

QorIQ LS1046A BSP v0.4



Contents

- Chapter 1 SDK Overview..... 17**
 - 1.1 What's New in LS1046A BSP v0.4..... 17
 - 1.2 Components..... 17
 - 1.3 Known Issues.....21

- Chapter 2 Getting Started.....23**
 - 2.1 SDK File System Images..... 23
 - 2.1.1 fsl-image-minimal..... 23
 - 2.1.2 fsl-image-mfgtool.....23
 - 2.1.3 fsl-image-full.....24
 - 2.1.4 fsl-image-core.....24
 - 2.1.5 fsl-image-virt.....25
 - 2.2 Essential Build Instructions.....25
 - 2.2.1 Install the SDK.....25
 - 2.2.2 Set Up Host Environment.....25
 - 2.2.3 Setup Poky.....26
 - 2.2.4 Builds.....27
 - 2.3 Additional Instructions for Developers.....27
 - 2.3.1 Customize U-Boot.....27
 - 2.3.2 Customize the Linux Kernel.....28
 - 2.3.3 Build Native Packages.....29

- Chapter 3 Deploy U-Boot, Linux Kernel, and Root Filesystem to a Reference Design Board (RDB).....30**
 - 3.1 Introduction.....30
 - 3.2 Basic Host Set-up.....30
 - 3.3 Supported Boards.....31
 - 3.3.1 LS1046ARDB.....31
 - 3.3.1.1 Overview.....31
 - 3.3.1.2 Switch Settings.....32
 - 3.3.1.3 U-Boot Environment Variables.....32
 - 3.3.1.3.1 U-Boot Environment Variable "hwconfig".....32
 - 3.3.1.3.2 Configuring U-Boot Network Parameters.....33
 - 3.3.1.4 Frame Manager Microcode (FMan Ucode).....33
 - 3.3.1.5 RCW (Reset Configuration Word) and Ethernet Interfaces.....34
 - 3.3.1.6 System Memory Map.....35
 - 3.3.1.7 Flash Bank Usage.....36
 - 3.3.1.8 Programming a New U-Boot, RCW and FMan Ucode.....38
 - 3.3.1.9 Deployment.....40
 - 3.3.1.9.1 FIT Image Deployment from TFTP.....40
 - 3.3.1.9.2 FIT Image Deployment from Flash.....40
 - 3.3.1.9.3 NFS Deployment.....41
 - 3.3.1.9.4 SD Deployment.....42
 - 3.3.1.9.5 QSPI Deployment.....43
 - 3.3.1.10 Check 'Link Up' for Serial Ethernet Interfaces.....44
 - 3.3.1.11 Basic Networking Ping Test.....45

Chapter 4 System Recovery	56
4.1 Environment Setup.....	56
4.1.1 Environment Setup (Common).....	56
4.2 Image Recovery.....	56
4.2.1 Recover system with already working U-Boot.....	56
4.2.2 Recover system using CodeWarrior Flash Programmer.....	57
Chapter 5 Linux Kernel Drivers	60
5.1 DMA Controller.....	60
5.1.1 Enhanced Direct Memory Access Driver (ARM).....	60
5.1.1.1 eDMA User Manual.....	60
5.2 DPAA 1.x Devices.....	62
5.2.1 DPAA Primer for Software Architecture.....	62
5.2.1.1 DPAA Primer.....	62
5.2.1.1.1 General Architectural Considerations.....	63
5.2.1.1.2 Multicore Design.....	63
5.2.1.1.3 Parse/classification Software Offload.....	63
5.2.1.1.4 Flow Order Considerations.....	64
5.2.1.1.5 Managing Flow-to-Core Affinity.....	66
5.2.1.2 DPAA Goals.....	67
5.2.1.3 FMan Overview.....	68
5.2.1.4 QMan Overview.....	70
5.2.1.5 QMan Scheduling.....	75
5.2.1.6 BMan.....	79
5.2.1.7 Order Handling.....	80
5.2.1.8 Pool Channels.....	83
5.2.1.9 Application Mapping.....	87
5.2.1.10 FQ/WQ/Channel.....	90
5.2.2 Queue Manager (QMan) and Buffer Manager (BMan).....	93
5.2.2.1 QMan/BMan Drivers Release Notes.....	93
5.2.2.2 QMan BMan API Reference Manual.....	101
5.2.2.2.1 About this document.....	101
5.2.2.2.2 Introduction to the Queue Manager and the Buffer Manager.....	101
5.2.2.2.3 Buffer Manager.....	101
5.2.2.2.4 BMan CoreNet portal APIs.....	106
5.2.2.2.5 Queue Manager.....	111
5.2.2.2.6 QMan portal APIs.....	120
5.2.2.2.7 QMAN CEETM APIs.....	136
5.2.2.2.8 Other QMan APIs.....	152
5.2.2.2.9 USDPAA-specific APIs.....	153
5.2.2.2.10 Sysfs and debugfs QMan/BMan interfaces.....	155
5.2.2.2.11 Error handling and reporting.....	169
5.2.2.2.12 Operating system specifics.....	170
5.2.3 Configuring DPAA Frame Queues.....	170
5.2.3.1 Introduction.....	170
5.2.3.2 FMan Network interface Frame Queue Configuration.....	171
5.2.3.3 FMan network interface ingress FQs configuration.....	171
5.2.3.4 Ingress FQs common configuration guidelines.....	172
5.2.3.5 Dynamic load balancing with order preservation - ingress FQs configuration guidelines...	173
5.2.3.6 Dynamic load balancing with order restoration - ingress FQs configuration guidelines.....	174
5.2.3.7 Static distribution - Ingress FQs Configuration Guidelines.....	175
5.2.3.8 FMan network interface egress FQs configuration.....	175

5.2.3.9 Accelerator Frame Queue Configuration.....	176
5.2.3.10 DPAA Frame Queue Configuration Guideline Summary.....	176
5.2.4 Frame Manager.....	179
5.2.4.1 Frame Manager Linux Driver User Guide.....	179
5.2.4.1.1 Introduction.....	179
5.2.4.1.2 The Linux FMD Devices.....	181
5.2.4.1.3 Linux FMD Programming Model.....	184
5.2.4.1.4 The Linux FMan Device.....	185
5.2.4.1.5 The Linux PCD Device.....	186
5.2.4.1.6 The Linux Port Devices.....	193
5.2.4.2 Frame Manager Linux Driver API Reference.....	197
5.2.4.3 Frame Manager Driver User Guide.....	197
5.2.4.3.1 Frame Manager Driver	197
5.2.4.4 Frame Manager Driver API Reference.....	249
5.2.4.5 Frame Manager Configuration Tool User Guide.....	249
5.2.4.5.1 Frame Manager Configuration Tool User Guide	249
5.2.5 Security Engine (SEC).....	335
5.2.5.1 SEC Device Driver User Manual.....	335
5.2.6 Pattern Matching Engine (PME).....	344
5.2.6.1 PME Driver Release Notes.....	344
5.2.6.2 Pattern Matcher 2.0 Software User's Guide.....	350
5.2.6.2.1 Preface.....	350
5.2.6.2.2 Overview.....	351
5.2.6.2.3 Regular Expressions.....	355
5.2.6.2.4 Application Interface.....	355
5.2.6.2.5 Stateful Rules.....	357
5.2.6.2.6 Pattern Management Software.....	359
5.2.6.2.7 Pattern Matcher Driver Software.....	365
5.2.6.2.8 Data Scan Sample Application.....	372
5.2.6.2.9 Software Components.....	377
5.2.6.2.10 Pattern Matcher FAQ.....	378
5.2.6.2.11 Revision History.....	381
5.2.6.3 Pattern Matcher 2.0, 2.1, 2.2 Software API Reference Manual.....	381
5.2.6.3.1 Introduction.....	381
5.2.6.3.2 Software Component Overview.....	382
5.2.6.3.3 Regular Expression Compiler.....	383
5.2.6.3.4 Stateful Rule Compiler.....	386
5.2.6.3.5 Pattern Matcher Configuration (PMC) API.....	390
5.2.6.3.6 Linker-Loader.....	397
5.2.6.3.7 Pattern Matcher Driver.....	420
5.2.6.3.8 Control Interface.....	455
5.2.6.3.9 Appendix: PMP Message Format.....	460
5.2.7 Decompression/Compression Acceleration (DCE).....	467
5.2.7.1 DCE Drivers Release Notes.....	468
5.3 Enhanced Secured Digital Host Controller (eSDHC).....	470
5.3.1 eSDHC Driver User Manual.....	470
5.4 Ethernet.....	475
5.4.1 Linux Ethernet Driver for DPAA 1.x Family.....	475
5.4.1.1 Linux DPAA 1.x Ethernet Primer.....	475
5.4.1.1.1 Introduction.....	475
5.4.1.1.2 Intended Use Cases.....	476
5.4.1.1.3 The DPAA-Eth View of the World.....	480
5.4.1.1.4 DPAA Resources Initialization.....	484
5.4.1.1.5 The (Simplified) Life of a Packet.....	484
5.4.1.1.6 Advanced Drivers Use Cases.....	491

5.4.1.1.7 Appendix A: Infrequently Asked Questions.....	493
5.4.1.1.8 Appendix B: Frequently Asked Questions.....	493
5.4.1.2 Linux DPAA 1.x Ethernet Drivers.....	494
5.4.1.2.1 Introduction.....	494
5.4.1.2.2 Private DPAA Ethernet Driver.....	495
5.4.1.2.3 Ethernet Advanced Drivers.....	510
5.4.1.2.4 Offline Parsing Port Driver.....	523
5.4.1.2.5 Link Management.....	525
5.4.1.2.6 Debugging.....	527
5.4.1.2.7 Adding support for DPAA Ethernet in Topaz Hypervisor.....	529
5.4.1.2.8 MACsec.....	531
5.4.1.2.9 Changes from previous versions.....	544
5.4.1.3 Quality of Service.....	547
5.4.1.3.1 Features.....	547
5.4.1.3.2 Policing.....	547
5.4.1.3.3 Scheduling and Shaping.....	547
5.5 Flash Memory.....	557
5.5.1 JFFS2 on NOR Flash Device Driver User Manual.....	557
5.5.2 JFFS2 on NAND Flash Device Driver User Manual.....	560
5.5.3 IFC NOR Flash User Manual.....	564
5.5.4 IFC NAND Flash User Manual.....	570
5.6 IEEE 1588 Timer Module.....	577
5.6.1 IEEE 1588 Device Driver User Manual.....	577
5.7 Low Power UART User Guide.....	584
5.7.1 Low Power UART User Guide.....	584
5.8 PCI Express Interface Controller.....	587
5.8.1 PCIe Linux Driver.....	587
5.8.2 EDAC Driver User Manual.....	590
5.8.3 PCIe Advanced Error Reporting User Manual.....	592
5.8.4 PCIe Remove and Rescan User Manual.....	594
5.8.5 PCIe Endpoint User Manual.....	596
5.9 SATA Controller.....	600
5.9.1 External SATA Controller User Manual.....	600
5.9.2 NXP Native SATA User Manual.....	606
5.10 Serial Peripheral Interface.....	609
5.10.1 QuadSPI Driver for TWR-LS1021A User Manual.....	609
5.10.1.1 QuadSPI Driver User Manual.....	609
5.11 Universal Serial Bus Interfaces.....	611
5.11.1 USB 2.0 Host Driver.....	611
5.11.1.1 USB 2.0 Host Driver User Manual.....	611
5.11.2 USB Gadget Memory Driver User Manual.....	623
5.11.2.1 USB Gadget for Memory Devices.....	624
5.11.3 USB Gadget Network Driver User Manual.....	629
5.11.3.1 USB Gadget for Network Devices.....	629
5.11.4 USB 3.0 Host/Peripheral Linux Driver User Manual.....	633
5.11.4.1 USB 3.0 Host/Peripheral Linux Driver User Manual.....	633
5.12 Watchdog Timers.....	643
5.12.1 Watchdog Device Driver User Manual.....	643
Chapter 6 Additional Linux Use Cases.....	648
6.1 Power Management.....	648
6.1.1 Power Management User Manual.....	648
6.1.2 CPU Frequency Switching User Manual.....	650
6.1.3 System Monitor.....	654

6.1.3.1 Power Monitor User Manual.....	654
6.1.3.2 Thermal Monitor User Manual.....	659
6.1.3.3 Web-based System Monitor User Guide.....	660
6.2 PCIe DMA Test User Manual.....	662
6.2.1 Introduction to PCI DMA Test.....	662
6.2.2 PCI DMA EP Application.....	664
6.2.3 PCI DMA Host Driver Module	667
6.2.4 Test Procedure.....	671

Chapter 7 Linux User Space.....673

7.1 Linux HugeTLBFS User Guide.....	673
7.1.1 Introduction.....	673
7.1.2 Objectives.....	673
7.1.3 TLBFS Basics.....	673
7.1.3.1 4KB TLB0 Miss Issues.....	673
7.1.3.2 e500 Family TLB1.....	674
7.1.3.3 HugeTLB Basics.....	674
7.1.3.4 Normal vs. Gigantic hugepages.....	675
7.1.4 Building and Booting Linux with Hugetlb.....	675
7.1.4.1 Booting Linux for HugeTLB.....	675
7.1.4.2 Setting Up Mount Points with hugeadm.....	676
7.1.4.3 Running Hugepage mount/query Example.....	677
7.1.5 Hugepage Allocation in User Programs.....	677
7.1.5.1 Using Shared Memory Hugepages: SHM_HUGETLB.....	678
7.1.5.2 Running the SHM_HUGETLB Example.....	678
7.1.5.3 Using Shared Memory Hugepages: Link with libhugetlbf.....	678
7.1.6 Manipulating Shared Memory System Tunables.....	679
7.1.6.1 Allocating hugepages for Heap Memory.....	679
7.1.6.2 Invoking Applications with Huge Heap (malloc()/new).....	679
7.1.6.3 Moving the Heap for Hugepage Allocations.....	680
7.1.7 Using Hugepages for .text, .data, and/or .bss (gnu ld 2.17+).....	680
7.1.8 Performing mmap on a Hugepage-backed File.....	681
7.1.8.1 Exercising the mmap() File Example.....	681
7.1.8.2 Requesting Anonymous mmap() with Hugepages.....	682
7.1.8.3 Using HugeTLB mmap(): Arguments and Error Checking.....	682
7.1.8.4 Using HugeTLB munmap(): Arguments and Error Checking.....	682
7.1.9 Running the Test.....	683
7.1.9.1 When to Use Hugepages.....	683
7.1.10 Matching Hugetlb Methods with Program Characteristics.....	684
7.1.11 Using Multiple Types of Hugepage Allocation Methods.....	684
7.1.12 Hugetlb Benefits and Limitations.....	684
7.1.13 Understand the Differences Between Applications.....	685
7.1.14 HugeTLB Status and Support.....	685
7.1.15 HugeTLB Resources.....	685
7.2 OpenSSL Offload User's Guide.....	685
7.2.1 Overview.....	685
7.2.1.1 OpenSSL Software architecture.....	686
7.2.1.1.1 OpenSSL's ENGINE Interface.....	687
7.2.1.1.2 NXP solution for OpenSSL hardware offloading	687
7.2.2 Valid TLS Ciphersuites based on TLS protocol version.....	687
7.3 USDPAA.....	692
7.3.1 USDPAA User Guide.....	692
7.3.1.1 Introduction.....	693
7.3.1.1.1 Intended audience.....	693

7.3.1.1.2	USDPAA overview.....	693
7.3.1.1.3	USDPAA and legacy Linux software.....	694
7.3.1.2	USDPAA assumptions and use cases.....	694
7.3.1.2.1	Assumptions.....	694
7.3.1.2.2	Use cases.....	695
7.3.1.3	USDPAA components.....	696
7.3.1.3.1	Device-tree handling.....	696
7.3.1.3.2	QMan and BMan drivers and C API.....	696
7.3.1.3.3	DMA memory management.....	699
7.3.1.3.4	Network configuration.....	700
7.3.1.3.5	CPU isolation.....	701
7.3.1.4	Relationship to SDK Linux ethernet subsystem.....	702
7.3.1.4.1	Selecting ethernet interfaces for USDPAA.....	703
7.3.1.4.2	FMC, FMD, and the ethernet Driver.....	704
7.3.1.5	Supported hardware platforms.....	705
7.3.1.5.1	P4080DS.....	705
7.3.1.5.2	P3041DS.....	706
7.3.1.5.3	P5020DS.....	706
7.3.1.5.4	P5040DS.....	706
7.3.1.5.5	P2041RDB.....	706
7.3.1.5.6	B4860QDS.....	706
7.3.1.5.7	T4240QDS.....	707
7.3.1.6	Example applications.....	707
7.3.1.7	USDPAA installation and execution.....	707
7.3.1.7.1	Files needed to boot Linux on the P4080DS system.....	707
7.3.1.7.2	About U-Boot and network interfaces.....	708
7.3.1.7.3	P4080DS NOR flash banks.....	710
7.3.1.7.4	Programming the P4080DS NOR flash bank 4.....	711
7.3.1.7.5	Boot into bank 4 and set more variables.....	711
7.3.1.7.6	Environment variable hwconfig and optical 10G.....	712
7.3.1.7.7	Booting Linux.....	712
7.3.1.7.8	Using tftp for the kernel, device-tree, and file-system.....	713
7.3.1.8	Using configurations other than SerDes 0xe.....	713
7.3.1.8.1	SGMII (4 x 1 Gbps) card and one XAUI (10 Gbps) card.....	713
7.3.1.8.2	SGMII (4 x 1 Gbps) card and no XAUI (10 Gbps) card.....	714
7.3.1.9	Known limitations	714
7.3.1.10	Document history.....	715
7.3.2	USDPAA Multiple Process Support.....	715
7.3.2.1	USDPAA Multiple Process Support.....	715
7.3.2.2	USDPAA User/Kernel Device Interface.....	716
7.3.2.3	USDPAA Resource Management.....	717
7.3.2.4	BMan and QMan API.....	719
7.3.2.5	USDPAA Thread and Global API.....	720
7.3.2.6	USDPAA DMA API.....	721
7.3.2.7	USDPAA netcfg.h.....	723
7.3.2.8	Kernel configuration.....	723
7.3.2.9	Device Tree (Excluding QMan/BMan Resource Ranges).....	723
7.3.2.10	Device Tree (QMan/BMan Resource Ranges).....	724
7.3.2.11	USDPAA Boot Arguments.....	726
7.3.2.12	USDPAA Virtualisation and Partitioning	727
7.3.2.13	Multi-process PPAC Applications.....	727
7.3.2.14	Limitations.....	728
7.4	USDPAA Applications.....	729
7.4.1	USDPAA ceetm Demo User Guide.....	729
7.4.1.1	Introduction.....	729

7.4.1.2 Overview of ceetm demo.....	729
7.4.1.3 Features of ceetm demo.....	729
7.4.1.4 CEETM use case.....	729
7.4.1.5 CEETM configuration file.....	731
7.4.1.6 Running ceetm_demo.....	733
7.4.1.7 Generate traffic flows.....	733
7.4.2 DPAA Offloading Applications Users Guide.....	734
7.4.2.1 Introduction.....	734
7.4.2.2 Altering the classifier_demo application.....	734
7.4.2.2.1 Overview.....	735
7.4.2.2.2 Running classifier_demo.....	738
7.4.2.3 Adapting the fragmentation_demo application.....	742
7.4.2.3.1 Overview.....	742
7.4.2.3.2 Running fragmentation_demo.....	742
7.4.2.4 Manipulating the reassembly_demo application.....	745
7.4.2.4.1 Overview.....	745
7.4.2.4.2 Running reassembly_demo.....	746
7.4.2.5 Customizing the ipsec_offload application.....	751
7.4.2.5.1 IPsec_offload application overview.....	751
7.4.2.5.2 Running the Application.....	756
7.4.2.5.3 ipsec_offload application flow.....	761
7.4.2.5.4 IPSEC tunnel setup using ipsec offload application and Strongswan with IKEv2.....	761
7.4.2.5.5 Ipsec offload application in multiple instances configuration.....	772
7.4.2.5.6 Host to host tunnels.....	774
7.4.2.6 Using the dpa_offload - the NF API offloading demo application.....	777
7.4.2.6.1 Overview.....	777
7.4.2.6.2 Running dpa_offload.....	778
7.4.2.7 References.....	783
7.4.2.8 Appendix A – Preparing DPA Offloading DTB Files.....	783
7.4.2.8.1 Compiling the device tree for USDPAA applications.....	783
7.4.2.8.2 Compiling the device tree for IP offloading.....	785
7.4.2.8.3 Compiling the device tree for IP reassembly.....	787
7.4.2.8.4 Compiling the device tree for ipsec offload.....	789
7.4.2.8.5 Compiling the device tree for Network Function Layer offloading.....	791
7.4.2.9 Appendix B - Enabling DPA Offloading Drivers in the Linux Kernel.....	792
7.4.2.10 Revision history.....	792
7.4.3 DPAA Offloading Drivers Reference Manual.....	792
7.4.3.1 Introduction.....	792
7.4.3.2 DPA Classifier.....	793
7.4.3.2.1 Table.....	793
7.4.3.2.2 Header Manipulation.....	810
7.4.3.2.3 Multicast.....	839
7.4.3.3 DPA IPsec.....	844
7.4.3.3.1 Initialization.....	845
7.4.3.3.2 DPA IPsec API.....	846
7.4.3.4 DPA Statistics.....	870
7.4.3.4.1 Initialization.....	870
7.4.3.4.2 DPA Statistics API.....	871
7.4.3.5 Network Function Layer.....	889
7.4.3.6 References.....	890
7.4.4 USDPAA PPAC User Manual.....	890
7.4.4.1 USDPAA PPAC Users Manual.....	890
7.4.4.2 Overview of PPAC.....	891
7.4.4.3 Overview of PPAC Method and Implementation.....	892
7.4.4.4 PPAC Files.....	893

7.4.4.5 PPAM Files.....	894
7.4.4.6 Packet-processing data structures.....	895
7.4.4.7 PPAM-provided Functions	897
7.4.4.8 PPAM Rx and Tx.....	898
7.4.4.9 PPAC Provided Functions.....	900
7.4.4.10 PPAC Setting.....	902
7.4.4.11 PPAC Buffers.....	903
7.4.4.12 Chronology of a DPAA application.....	904
7.4.4.13 A non-PPAC comparison, “hello_reflector”.....	906
7.4.5 USDPAA Reflector and PPAC User Guide SDK.....	906
7.4.5.1 Introduction.....	906
7.4.5.1.1 Intended audience.....	907
7.4.5.1.2 Change history.....	907
7.4.5.2 Overview of reflector.....	908
7.4.5.3 Overview of PPAC.....	908
7.4.5.4 PPAC details.....	908
7.4.5.4.1 IRQ mode for sleeping when idle.....	909
7.4.5.4.2 Buffers.....	909
7.4.5.4.3 Compile-time configuration.....	910
7.4.5.5 Running reflector.....	911
7.4.5.6 PPAC (and reflector) CLI commands.....	912
7.4.5.7 Running hello_reflector.....	913
7.4.5.8 Running hello_reflector (short circuit).....	913
7.4.5.9 Testing reflector.....	914
7.4.6 NXP USDPAA IPFWD User Manual Rev. 1.2.....	915
7.4.6.1 About this Book.....	915
7.4.6.2 Introduction.....	915
7.4.6.2.1 Purpose.....	915
7.4.6.3 Overview.....	915
7.4.6.3.1 USDPAA IPv4 forwarding application flow.....	915
7.4.6.4 Overview of PPAC.....	916
7.4.6.5 IPFwd related PPAC Details.....	916
7.4.6.5.1 Compile-time configuration.....	916
7.4.6.6 PPAM related compile time configuration.....	919
7.4.6.6.1 One million route support.....	919
7.4.6.7 IPFWD Application Suite.....	921
7.4.6.8 Possible configuration scenario for IPFWD.....	922
7.4.6.9 Using Two Computers to Test the IPFWD Application Suite.....	927
7.4.6.10 Flowchart for packet processing.....	930
7.4.6.10.1 Description of Flow chart.....	930
7.4.6.10.2 Running IPv4 forwarding on P4080DS board.....	931
7.4.6.10.3 Running IPv4 forwarding on P3041/P5020 board.....	933
7.4.6.10.4 Running IPv4 forwarding on T4240 board.....	934
7.4.6.10.5 Running IPv4 forwarding on B4860 board.....	935
7.4.6.10.6 Performance gap between 8 core and 6 core.....	935
7.4.6.10.7 PPAC (and IPFwd) CLI commands.....	935
7.4.6.11 IPv4 forward application Configuration command.....	936
7.4.6.11.1 Syntax.....	936
7.4.6.12 Traffic Generation.....	943
7.4.6.13 References.....	943
7.4.7 NXP USDPAA IPSecfwd User Manual.....	943
7.4.7.1 Introduction.....	943
7.4.7.1.1 Purpose.....	943
7.4.7.1.2 Change History.....	943
7.4.7.2 USDPAA IPSecfwd application.....	944

7.4.7.2.1 Application Overview.....	944
7.4.7.2.2 Packet Flow.....	944
7.4.7.2.3 Overview of IPSecfwd packet processing.....	945
7.4.7.2.4 Flow chart for IpSecfwd packet processing.....	948
7.4.7.3 Overview of PPAC.....	949
7.4.7.4 IPSecfwd related PPAM Details.....	949
7.4.7.4.1 In-Place Encryption/Decryption.....	949
7.4.7.5 Secfwd application suite.....	951
7.4.7.5.1 Using Two Computers to Test the IPFWD Application Suite.....	952
7.4.7.5.2 Running IPSecfwd on P4080DS board.....	954
7.4.7.5.3 Running IPv4 forwarding on P3041/P5020 board.....	955
7.4.7.5.4 Running IPv4 forwarding on T4240 board.....	956
7.4.7.5.5 Running IPv4 forwarding on B4860 board.....	957
7.4.7.5.6 PPAC (and IPSecfwd) CLI commands.....	957
7.4.7.5.7 IPSecfwd application Configuration command.....	958
7.4.7.6 References	967
7.4.7.7 Revision History.....	967
7.4.8 NXP Simple Crypto User Manual.....	967
7.4.8.1 Introduction.....	967
7.4.8.2 USDPAA Simple Crypto Application.....	967
7.4.8.2.1 Overview.....	967
7.4.8.2.2 Parameters to the application.....	968
7.4.8.2.3 Packet Flow.....	969
7.4.8.2.4 Throughput calculation.....	970
7.4.8.2.5 Running Simple Crypto Application on board.....	970
7.4.8.2.6 Simple Crypto command syntax.....	970
7.4.8.2.7 Snapshot of Simple Crypto output.....	972
7.4.9 NXP Simple Proto User Manual.....	972
7.4.9.1 Introduction.....	972
7.4.9.2 USDPAA Simple Proto Application.....	973
7.4.9.3 Overview.....	973
7.4.9.4 Parameters to the application.....	973
7.4.9.5 Packet Flow.....	975
7.4.9.6 Throughput calculation.....	976
7.4.9.7 Running Simple Proto Application on board.....	976
7.4.9.8 Simple Proto command syntax.....	976
7.4.9.9 MACSec protocol options.....	978
7.4.9.10 WiMAX protocol options.....	978
7.4.9.11 PDCP protocol options.....	978
7.4.9.12 RSA operations options.....	980
7.4.9.13 TLS protocol options.....	980
7.4.9.14 IPsec protocol options.....	981
7.4.9.15 MBMS protocol options.....	981
7.4.10 SEC Descriptor construction library (DCL).....	982
7.4.10.1 SEC Descriptor construction library (DCL).....	982
7.4.10.2 DCL Description.....	983
7.4.10.3 DCL Packaging.....	983
7.4.10.4 DCL Files.....	983
7.4.10.5 DCL Functional Description.....	983
7.4.10.6 Command Generator.....	983
7.4.10.7 Descriptor Disassembler.....	983
7.4.10.8 Upper-Tier DCL Descriptor Constructors.....	984
7.4.10.9 API Reference.....	984
7.4.10.9.1 API Reference Command Generator.....	984
7.4.10.9.2 Descriptor Constructors.....	997

7.4.10.9.3 Disassembler.....	1016
7.4.11 Runtime Assembler Library Reference.....	1016
7.4.11.1 Runtime Assembler Library Reference.....	1016
7.4.12 USDPAA PME Loopback User Guide.....	1016
7.4.12.1 Introduction.....	1016
7.4.12.1.1 Purpose.....	1017
7.4.12.1.2 Change history.....	1017
7.4.12.2 Overview of pme_loopback.....	1017
7.4.12.2.1 pme_loopback application flow.....	1018
7.4.12.3 pme_loopback application syntax.....	1020
7.4.12.3.1 pme_loopback_test.....	1020
7.4.12.3.2 create_ctx_direct_mode.....	1021
7.4.12.3.3 create_ctx_flow_mode.....	1021
7.4.12.3.4 prep_scan.....	1022
7.4.12.3.5 prep_scan_2.....	1024
7.4.12.3.6 start_scan.....	1025
7.4.12.3.7 stop_scan.....	1026
7.4.12.3.8 free_mem.....	1026
7.4.12.3.9 delete_ctx.....	1027
7.4.12.3.10 rm.....	1027
7.4.12.3.11 add.....	1028
7.4.12.3.12 list.....	1028
7.4.12.3.13 display_stats.....	1028
7.4.12.3.14 clear_stats.....	1029
7.4.12.3.15 help.....	1029
7.4.12.3.16 quit.....	1029
7.4.12.4 Running pme_loopback.....	1030
7.4.13 USDPAA IPFwd Longest Prefix Match User Manual.....	1031
7.4.13.1 NXP P4080/P5020/P3041 USDPAA IPFwd Longest Prefix Match User Manual.....	1031
7.4.13.1.1 Introduction.....	1031
7.4.13.1.2 Overview.....	1031
7.4.13.1.3 How is it different from existing Route cache based IPFwd?.....	1032
7.4.13.1.4 Longest Prefix Match algorithm	1032
7.4.13.1.5 Shared MAC Overview.....	1034
7.4.13.1.6 How to run shared MAC interface ?.....	1035
7.4.13.1.7 MAC-less use case.....	1037
7.4.13.1.8 How to ping MAC-less interface ?.....	1037
7.4.13.1.9 USDPAA LPM based IPv4 forwarding application flow.....	1038
7.4.13.1.10 Overview of packet flow:.....	1038
7.4.13.1.11 Overview of PPAC.....	1039
7.4.13.1.12 Compile-time configuration.....	1039
7.4.13.1.13 Order Preservation in LPM-IPFWD.....	1039
7.4.13.1.14 Order Restoration in LPM IPFWD.....	1039
7.4.13.1.15 Monitoring Rx/Tx fill-levels and flow-control via CGR.....	1040
7.4.13.1.16 LPM IPFWD Application Suite.....	1042
7.4.13.1.17 Possible configuration scenario for LPM based IPFWD.....	1043
7.4.13.1.18 Using Two Computers to Test the IPFWD Application Suite.....	1048
7.4.13.1.19 Running LPM IPv4 forwarding on P4080DS board.....	1050
7.4.13.1.20 Running LPM IPv4 forwarding on P3041/P5020 board.....	1052
7.4.13.1.21 USDPAA LPM IP Fwd performance gap between 6 core and 8 core.....	1053
7.4.13.1.22 PPAC (and IPFwd) CLI commands.....	1053
7.4.13.1.23 Syntax.....	1054
7.4.13.1.24 Command to show all enabled interfaces and their interface numbers.....	1055
7.4.13.1.25 Help for show all enabled interfaces command.....	1055
7.4.13.1.26 Assign IP address to interfaces.....	1056

7.4.13.127	Help for assign IP address to interfaces.....	1057
7.4.13.128	Adding a Route Entry.....	1057
7.4.13.129	Help for Route Entry Addition.....	1058
7.4.13.130	Deleting a Route Entry.....	1058
7.4.13.131	Help for Deleting a Route Entry.....	1058
7.4.13.132	Adding an ARP Entry.....	1058
7.4.13.133	Help for ARP Entry Addition.....	1059
7.4.13.134	Deleting an ARP Entry.....	1059
7.4.13.135	Help for Deleting an ARP Entry.....	1059
7.4.13.136	References.....	1060
7.4.14	NXP USDPAA FRA Configuration User Manual.....	1060
7.4.14.1	Introduction.....	1060
7.4.14.1.1	Purpose.....	1060
7.4.14.2	FRA Configuration.....	1060
7.4.14.2.1	Rman_cfg Element.....	1061
7.4.14.2.2	Network_cfg Element.....	1062
7.4.14.2.3	Transaction Element.....	1063
7.4.14.2.4	Distribution Element.....	1065
7.4.14.2.5	Policy Element.....	1070
7.4.14.3	Revision History.....	1071
7.4.15	NXP USDPAA FRA User Manual.....	1071
7.4.15.1	Overview.....	1072
7.4.15.2	Introduction.....	1072
7.4.15.2.1	Purpose	1072
7.4.15.2.2	Definitions and Acronyms	1072
7.4.15.3	Overview of FRA.....	1072
7.4.15.4	Running FRA on Two Boards.....	1080
7.4.15.4.1	Installation.....	1080
7.4.15.4.2	Compilation.....	1080
7.4.15.4.3	Configuring FRA.....	1080
7.4.15.4.4	Prepare the Hardware.....	1081
7.4.15.4.5	Managing RCW and U-boot Image.....	1082
7.4.15.4.6	Booting Linux.....	1083
7.4.15.4.7	Running FRA.....	1084
7.4.15.4.8	Testing FRA.....	1086
7.4.15.4.9	Debugging FRA.....	1087
7.4.15.4.10	Testing Port Write.....	1088
7.4.15.5	Running FRA with flow control.....	1088
7.4.15.5.1	Booting Linux.....	1088
7.4.15.5.2	Compilation FRA with flow control.....	1089
7.4.15.5.3	Running FRA.....	1089
7.4.15.6	Running FRA on One Board.....	1091
7.4.15.6.1	Prepare the Hardware (one board).....	1091
7.4.15.6.2	Configuring FRA (one board).....	1092
7.4.15.6.3	Managing RCW.....	1093
7.4.15.6.4	Running FRA.....	1093
7.4.15.6.5	Testing FRA (one board).....	1094
7.4.15.7	Revision History.....	1095
7.4.16	NXP USDPAA SRA User Manual.....	1095
7.4.16.1	Serial RapidIO application.....	1095
7.4.16.1.1	Overview.....	1096
7.4.16.1.2	SRA environment setup.....	1096
7.4.16.1.3	Boot	1099
7.4.16.1.4	SRA Demo.....	1099
7.4.16.1.5	SRA Command Description.....	1100

7.4.16.1.6 SRA command Usage.....	1101
7.4.16.1.7 Run SRA Demo.....	1105
7.4.16.2 Revision History.....	1108
7.4.17 USDPAAR MU User Manual.....	1109
7.4.17.1 RapidIO Message Unit Application.....	1109
7.4.17.2 Overview.....	1109
7.4.17.3 RMU Environment Setup.....	1109
7.4.17.3.1 Hardware Environment	1109
7.4.17.3.2 SDK Installation.....	1109
7.4.17.3.3 RCW Generation.....	1109
7.4.17.3.4 Kernel Building Configuration.....	1110
7.4.17.4 Boot	1110
7.4.17.5 RMU Demo.....	1111
7.4.17.6 RMU Commands.....	1112
7.4.17.7 Run RMU Demo.....	1116
7.4.18 USDPAAR SRIO IPsec Offload User Manual.....	1117
7.4.18.1 Introduction.....	1117
7.4.18.2 Overview of srio_ipsec_offload demo.....	1117
7.4.18.2.1 Srio_IPsec_offload outbound flows.....	1118
7.4.18.2.2 Srio_IPsec_offload inbound flows.....	1118
7.4.18.2.3 Limitations.....	1118
7.4.18.3 Running srio_ipsec_offload.....	1119
7.4.18.3.1 Application environment specifications	1119
7.4.18.3.2 Running srio_ipsec_offload.....	1119
7.4.18.3.3 Application configuration for IPsec.....	1120
7.4.18.3.4 Running Traffic.....	1120
7.4.18.4 Compiling the device tree and enabling kernel options	1120
7.4.18.4.1 Compiling the device tree for B4860.....	1120
7.4.18.4.2 Enabling DPA Offloading and RMAN Drivers in the Linux Kernel.....	1121

Chapter 8 Boot Loaders..... 1122

8.1 Primary Protected Application (PPA) User's Guide.....	1122
8.1.1 Introduction.....	1122
8.1.1.1 Rationale and Scope.....	1122
8.1.1.2 References.....	1122
8.1.1.3 Definitions.....	1123
8.1.2 Boot Flow Architecture.....	1123
8.1.2.1 LS1046A Boot Flow.....	1123
8.1.3 Loading and Initializing the PPA.....	1126
8.1.4 How to Call SMC/PSCI functions.....	1126
8.1.5 PSCI Function List.....	1127
8.1.5.1 PSCI_VERSION.....	1127
8.1.5.2 CPU_ON.....	1128
8.1.5.3 CPU_OFF.....	1128
8.1.5.4 CPU_SUSPEND.....	1129
8.1.5.5 AFFINITY_INFO.....	1129
8.1.5.6 SYSTEM_OFF.....	1130
8.1.5.7 SYSTEM_RESET.....	1130
8.1.5.8 PSCI Return Code Values.....	1130
8.1.5.9 PSCI Functions Implemented, by SoC.....	1131
8.1.6 SMC Function List.....	1131
8.1.6.1 Function Count - SMC64.....	1131
8.1.6.2 Function Count - SMC32.....	1132
8.1.6.3 Get UUID.....	1132

8.1.6.4 Get Revision.....	1132
8.1.7 Building the PPA.....	1133
8.1.8 System Considerations When Calling SMC & PSCI Functions.....	1133
8.2 Secure Boot: PBL Based Platforms.....	1134
8.2.1 Introduction.....	1134
8.2.2 Secure boot Process.....	1135
8.2.3 Pre-Boot Phase.....	1136
8.2.4 ISBC Phase.....	1138
8.2.4.1 Flow in the ISBC Code.....	1138
8.2.4.2 Super Root keys (SRKs) and signing keys.....	1139
8.2.4.3 Key Revocation.....	1139
8.2.4.4 Alternate Image Support.....	1140
8.2.4.5 ESBC with CSF Header.....	1140
8.2.5 ESBC Phase.....	1140
8.2.5.1 Boot script.....	1141
8.2.5.1.1 Where to place the boot script?.....	1141
8.2.5.1.2 Chain of Trust	1142
8.2.5.1.3 Chain of Trust with Confidentiality	1143
8.2.6 Next Executable (Linux Phase).....	1146
8.2.7 CST Tool.....	1146
8.2.7.1 KEY GENERATION.....	1146
8.2.7.1.1 gen_keys.....	1146
8.2.7.1.2 gen_drv_drbg.....	1148
8.2.7.1.3 gen_otpmk_drbg.....	1149
8.2.7.2 CSF Header Generation.....	1150
8.2.7.2.1 Default Usage.....	1151
8.2.7.2.2 Verbose Mode (--verbose).....	1154
8.2.7.2.3 Public Key/ SRK Hash Generation Only (--hash).....	1155
8.2.7.2.4 ISBC Key Extension (IE).....	1155
8.2.7.2.5 Image Hash Generation (--img_hash).....	1162
8.2.7.2.6 Help (--help).....	1163
8.2.7.3 Code Signing Tool Walkthrough.....	1163
8.2.8 Product execution.....	1165
8.2.8.1 Getting started.....	1165
8.2.8.1.1 Environment for Secure Boot.....	1165
8.2.8.1.2 SDK/ Images required for the demo.....	1165
8.2.8.2 Chain of Trust.....	1166
8.2.8.2.1 Other images required for the demo.....	1166
8.2.8.2.2 Boot Script and Signing the images.....	1166
8.2.8.2.3 Running secure boot (Chain of Trust).....	1170
8.2.8.3 Chain of Trust with Confidentiality.....	1171
8.2.8.3.1 Other images required for the demo.....	1172
8.2.8.3.2 Encap Bootscript.....	1172
8.2.8.3.3 Decap Bootscript.....	1172
8.2.8.3.4 Creating CSF Headers.....	1173
8.2.8.3.5 Running secure boot (Chain of Trust with Confidentiality).....	1173
8.2.8.4 NAND Secure Boot (Chain of Trust).....	1174
8.2.8.4.1 Running Secure Boot Chain of Trust (from NAND).....	1175
8.2.8.5 NAND Secure Boot (Chain of Trust with Confidentiality).....	1176
8.2.8.5.1 Running Secure Boot Chain of Trust with Confidentiality (from NAND).....	1177
8.2.9 Troubleshooting.....	1179
8.2.10 CSF Header Data Structure.....	1180
8.2.11 ISBC Validation Error Codes.....	1198
8.2.12 ESBC Validation Error Codes.....	1203
8.2.13 Trust Architecture and SFP Information.....	1204

8.2.14 Using QCVS Tool (Secure Boot From NAND)	1205
8.2.15 Appendix LS1046 Secure Boot demo.....	1211

Chapter 9 Virtualization..... 1215

9.1 KVM/QEMU User Guide and Reference.....	1215
9.1.1 KVM/QEMU Release Notes.....	1215
9.1.2 KVM for ARM Architecture Users Guide and Reference.....	1215
9.1.2.1 Introduction to KVM and QEMU.....	1215
9.1.2.1.1 Overview.....	1215
9.1.2.1.2 Organization of this Document.....	1216
9.1.2.1.3 Virtual Machine Overview.....	1217
9.1.2.1.4 Introduction to KVM and QEMU.....	1217
9.1.2.1.5 Device Tree Overview.....	1219
9.1.2.1.6 References.....	1219
9.1.2.1.7 For More Information.....	1220
9.1.2.2 Building QEMU and KVM.....	1220
9.1.2.2.1 Overview.....	1220
9.1.2.2.2 Building Linux with KVM.....	1220
9.1.2.2.3 Building QEMU.....	1224
9.1.2.2.4 Creating a host Linux root filesystem.....	1224
9.1.2.3 Using QEMU and KVM.....	1225
9.1.2.3.1 Overview of Using QEMU.....	1225
9.1.2.3.2 Virtual Machine Memory.....	1227
9.1.2.3.3 Virtual network interfaces.....	1228
9.1.2.3.4 VMs and the Linux Scheduler.....	1228
9.1.2.4 Virtual machine reference.....	1229
9.1.2.4.1 VM Overview.....	1229
9.1.2.4.2 Memory Map of Virtual I/O Devices.....	1229
9.1.2.4.3 Virtual machine state at initialization.....	1230
9.1.2.4.4 Virtual CPUs.....	1231
9.1.2.4.5 VGIC.....	1231
9.1.2.5 Debugging virtual machines.....	1232
9.1.2.5.1 QEMU Monitor.....	1232
9.1.2.5.2 QEMU GDB Stub.....	1232
9.1.2.6 KVM/QEMU How-to's.....	1234
9.1.2.6.1 Quick-start Steps to Build and Deploy KVM Using Yocto.....	1234
9.1.2.6.2 Quick-start Steps to Run KVM Using Hugetlbfs.....	1236
9.1.2.6.3 How to Use Virtual Network Interfaces Using Virtio.....	1238
9.1.2.6.4 How to use vhost-net with virtio.....	1239
9.1.2.6.5 Debugging: How to Examine Initial Virtual Machine State with QEMU.....	1240
9.1.2.6.6 Debugging: How to Profile Virtualization Overhead with KVM.....	1241
9.2 Libvirt Users Guide.....	1242
9.2.1 Introduction to libvirt.....	1242
9.2.1.1 Overview.....	1242
9.2.1.2 For Further Information.....	1243
9.2.1.3 Libvirt in the NXP QorIQ SDK -- Supported Features.....	1243
9.2.2 Build, Installation, and Configuration.....	1246
9.2.2.1 Building Libvirt with Yocto.....	1246
9.2.2.2 Running libvirtd.....	1246
9.2.2.3 Libvirt Domain Lifecycle.....	1247
9.2.2.4 Libvirt URIs.....	1248
9.2.2.5 virsh.....	1248
9.2.2.6 Libvirt xml.....	1248
9.2.3 Examples.....	1248

- 9.2.3.1 KVM Examples..... 1248
 - 9.2.3.1.1 Libvirt KVM/QEMU Example (ARM Architecture)..... 1248
- 9.2.3.2 Libvirt_lxc Examples..... 1252
 - 9.2.3.2.1 Basic Example..... 1252
 - 9.2.3.2.2 Custom Container Filesystem..... 1254
 - 9.2.3.2.3 Container Terminal Setup..... 1255
 - 9.2.3.2.4 Networking Examples..... 1257
- 9.3 Linux Containers User Guide..... 1261**
 - 9.3.1 Introduction to Linux Containers..... 1261
 - 9.3.1.1 NXP LXC Release Notes..... 1261
 - 9.3.1.2 Overview..... 1262
 - 9.3.1.3 Comparing LXC and Libvirt..... 1263
 - 9.3.1.4 For Further Information..... 1263
 - 9.3.2 Build, Installation, and Configuration..... 1264
 - 9.3.2.1 Summary..... 1264
 - 9.3.2.2 LXC: Building with Yocto..... 1264
 - 9.3.2.3 Building the Linux Kernel..... 1265
 - 9.3.2.4 Host Root Filesystem Configuration for Linux Containers..... 1267
 - 9.3.3 More Details..... 1267
 - 9.3.3.1 LXC: Command Reference..... 1267
 - 9.3.3.2 LXC: Configuration Files..... 1268
 - 9.3.3.3 LXC: Templates..... 1269
 - 9.3.3.4 Containers with Libvirt..... 1270
 - 9.3.3.5 Linux Control Groups (cgroups)..... 1271
 - 9.3.3.6 Linux Namespaces..... 1272
 - 9.3.3.7 POSIX Capabilities..... 1272
 - 9.3.4 LXC How To's..... 1273
 - 9.3.4.1 LXC: Getting Started (with a Busybox System Container)..... 1273
 - 9.3.4.2 LXC: How to configure non-virtualized networking (lxc-no-netns.conf)..... 1277
 - 9.3.4.3 LXC: How to assign a physical network interface to a container (lxc-phys.conf)..... 1277
 - 9.3.4.4 LXC: How to configure networking with virtual Ethernet pairs (lxc-veth.conf)..... 1279
 - 9.3.4.5 LXC: How to configure networking with macvlan (lxc-macvlan.conf)..... 1280
 - 9.3.4.6 LXC: How to configure networking using a VLAN (lxc-vlan.conf)..... 1282
 - 9.3.4.7 LXC: How to monitor containers..... 1284
 - 9.3.4.8 LXC: How to modify the capabilities of a container to provide additional isolation..... 1285
 - 9.3.4.9 LXC: How to use cgroups to manage and control a containers resources..... 1285
 - 9.3.4.10 LXC: How to run an application in a container with lxc-execute..... 1287
 - 9.3.4.11 LXC: How to run an unprivileged container..... 1288
 - 9.3.4.12 LXC: How to run containers with Seccomp protection..... 1290
 - 9.3.5 Libvirt How To's..... 1292
 - 9.3.5.1 Basic Example..... 1292
 - 9.3.6 Appendix..... 1294
 - 9.3.6.1 LXC Configuration File Reference..... 1294
 - 9.3.6.2 Documentation/cgroups/cgroups.txt..... 1310
- 9.4 Docker Containers..... 1321**
 - 9.4.1 Introduction to Docker Containers..... 1321
 - 9.4.1.1 Overview..... 1321
 - 9.4.2 Build and Installation..... 1322
 - 9.4.2.1 Building with Yocto..... 1322
 - 9.4.2.2 Building the Linux Kernel..... 1322
 - 9.4.3 Docker How To's..... 1323
 - 9.4.3.1 Running a webserver container..... 1323

Chapter 1

SDK Overview

1.1 What's New in LS1046A BSP v0.4

NXP Digital Networking is pleased to announce the release of the release of SDK 2.0, LS1046A BSP v0.4.

Highlights

- This is an early access release supporting LS1046A where loads are restricted to be issued in order. Future releases will add additional functionality and performance improvements.
- It is an incremental release to BSP v0.1, v0.2, v0.3 and v0.3.1, which are based on QorIQ SDK 2.0.
- This release includes the image tarball including ITB (kernel, dtb and core rootfs built by Yocto), u-boot, rcw, ramdisk, toolchain, 64MB QSPI flash image.
- Follow README file in the patch tarball to install this release.

See full list of features and changes below:

PPA

- LPM20

Linux Kernel Core, Virtualization

- No change since BSP v0.3.1

Linux Kernel Drivers

- LPUART (not present on RDB)
- Power Management: LPM20

User Space Datapath Acceleration Architecture (USDPA) and Reference Applications

- No change since BSP v0.3.1

U-Boot Boot Loader

- LPUART (not present on RDB)

Other Tools and Utilities

- No change since BSP v0.3.1

Benchmark

- NAS
- USDPA IPsec

For a list of known and fixed issues in LS1046A BSP v0.4, see "[Known Issues](#) on page 21" in the Overview section.

1.2 Components

Top-level components in SDK 2.0, LS1046A BSP v0.4

Overall

- PPA

SDK Overview

Components

- Linux Kernel and Virtualization
- Linux Kernel Drivers
- User Space Datapath Acceleration Architecture (USDPA) and Applications
- U-Boot Boot Loader
- Other Tools and Utilities
- Benchmark

PPA

- EL3 initialization of the platform
- CPU ON/OFF
- LPM20
- SMP boot in the Linux kernel

Linux Kernel Core and Virtualization

- Linux kernel 4.1.8
- ARM A72 (AArch64), Little Endian (default)
- ARM Generic timer
- 64-bit effective addressing
- 64-bit SMP kernel, 4 cores
- Huge Pages (hugetlbfs)
- Kernel-based Virtual Machine (KVM)
- Libvirt 1.2.19
- Linux Containers (LXC) 1.1.4 function support

Linux Kernel Driver

- LS1026A personality
- Booting kernel from QSPI flash and root filesystem in NVME card
- CAAM
- DUART
- GIC-400 (Interrupt Controller) support
- DPAA networking, 10G XFI (2 ports on Fman)
- DPAA networking, 1G RGMII
- DPAA networking, 1G SGMII
- eMMC/eSDXC card
- Flextimer
- FMD
- GPIO
- I2C
- IEEE1588 driver, ptpd stack
- IFC NAND
- IFC NOR (not present on RDB)

- KVM, LXC
- LPUART (not present on RDB)
- PCIe 3.0 Gen1 - e1000 card support
- PCIe endpoint
- PCIe MSI
- Power management: CPU idle, CPUFreq, DFS, LPM20
- Q/Bmana
- QDMA
- QSPI flash
- SATA controller
- USB 3.0: storage, audio
- Watchdog

User Space Datapath Acceleration Architecture (USDPA) and Applications

- Device-tree handling
- QMan and BMan drivers and C API
- DMA Memory Management
- Network Configuration
- CPU Isolation
- PPAC Reflector
- Hello Reflector
- IP Forward (route cache)
- IP Forward (longest prefix match)
- Simple Crypto
- Simple Proto

U-Boot Boot Loader

- U-boot version 2016.01
- 64-bit DDR4 SPD support (through I2C EEPROM)
- CCI400
- CGU (clock generation unit)
- DUART
- DSPI
- Fman IM mode, 1G RGMII
- IFC NAND
- IFC NOR, NOR boot (not present on RDB)
- LPUART (not present on RDB)
- PBI command
- PCIe Gen1 - e1000 card support
- PCIe Gen2

SDK Overview

Components

- PCIe Gen3 – msi display card
- QSGMII (not present on RDB)
- SD/MMC boot
- SD/MMC controller
- SGMII
- I2C
- OCRAM - bp140_wrapper (w/BP140) (Security Mem)(64KB)
- PPA integration
- QSGMII
- QSPI boot
- QSPI flash
- QSPI secure boot
- Serdes lane
- XFI (2 ports on Fman)
- SATA controller
- USB 3.0

Other Tools and Utilities

- Busybox
- CST
- FLIB
- FMC
- FMLIB
- Standalone application of Aquantia firmware programming

Benchmark

- Coremark
- Dhrystone
- EEMBC
- LMBENCH
- Linux IPv4 Forward
- Linux IPSec
- Linux TCP termination
- NAS
- USDPAA LPM and RC IP Forward (24G configuration)
- USDPAA IPSec

1.3 Known Issues

Known issues for this and previous releases of SDK 2.0, LS1046A BSP

The following table lists Known Issues in the BSP. Each issue has an identifier, a description, a disposition, a workaround (if available), and the release in which the issue was found or resolved.

ID	Description	Disposition	Opened in	Resolved in	Workaround
N/A	Error packet may occur on XFI2 port when not using Finisar optical module (shipped in the board kit) and blue fiber cable.	Hardware Issue	LS1046A BSP v0.1	N/A	
QLINUX-5835, QLINUX-5836, QLINUX-5837	Kernel crash encountered during long term performance test using netperf on 10G XFI port. 1. Occurrence: rare. Seen once over many iterations of the netperf test. 2. Not seen during any other type of tests. 3. Requires system reset due to kernel panic.	Resolved	LS1046A BSP v0.1	LS1046A BSP v0.4	
QLINUX-5840	Sandisk, Kingston and Apacer USB 3.0 drive were verified. Enumeration issue is faced with certain disks.	Open	LS1046A BSP v0.1		
QLINUX-5898	PCIe end point mode can't be detected with x86 PC host.	Open	LS1046A BSP v0.2		

Table continues on the next page...

Table continued from the previous page...

QLINUX-5988	There is kernel call trace while doing bridge performance test over PCIe and 10G ports in MSI mode	Open	LS1046A BSP v0.4		
QLINUX-5989	The system cannot enter sleep when enabling IFC.	Open	LS1046A BSP v0.4		
QLINUX-5930	The system cannot enter sleep when enabling Qman.	Open	LS1046A BSP v0.4		

Chapter 2

Getting Started

Yocto Project is an open-source collaboration project for embedded Linux developers. Yocto Project uses the Poky build system to make Linux images. For complete information about Yocto Project, see <https://www.yoctoproject.org/>.

2.1 SDK File System Images

This section describes the file system images that can be built using standard Yocto Project recipes included with the NXP SDK.

The file system images contain the programs, scripts, and other files that make up Linux user space. There are five standard images. They are described in the following sections.

In the SDK installation directory, look in `meta-freescale/recipes-fsl/images` for files that define these images.

Where to start: “`fsl-image-full`” contains a rich set of standard Linux features and all special NXP SDK-specific features. It is the best starting point for exploration and evaluation.

2.1.1 `fsl-image-minimal`: A Barebones Starting Point for Products

Contents:

This is a barebones image. It contains a small file set that allows Linux to boot and little else.

Purpose:

This image is intended as a starting point for product development. Users may add packages to it to form an image that targets their particular project or purpose. Packages may be added by editing `conf/local.conf` and adding new packages to be built and installed via

```
IMAGE_INSTALL_append = " package1 package2 etc"
```

Then, rebuild the image using `bitbake`. The result will be an image that is small enough for simple flash devices and is narrowly focused on a specific goal.

Users must add packages to “`fsl-image-minimal`” to make it useful. Thus it is not intended for “out-of-the-box” evaluation. Instead, use it as the basis for targeted images for your specific product.

2.1.2 `fsl-image-mfgtool`: A Small Flash Image for Managing Disks and Larger Images

Contents:

This is a small image that NXP preprograms into the flash on development boards. On many boards, the image is stored in a NOR flash and is loaded into a RAM disk when Linux boots. It contains disk management functions as described in the next section.

Purpose:

This image is intended to help users to load much larger images onto disks or disk-like devices such as SDHC cards or USB thumb drives. Thus, this image contains networking support to transfer images and also standard Linux disk and file system manipulation commands.

NXP preloads “`fsl-image-full`” (see below) onto disks on development boards that have them. For these boards, “`fsl-image-mfgtool`” can be used to restore the larger disk image if it becomes corrupted.

Users will find that it is often convenient to work with larger images and may wish to install “fsl-image-full” onto a disk-type device and use it with NXP development boards that do not come with a disk drive.

The networking and disk management commands that NXP supplies are standard Linux commands. They are, in fact, identical on all architectures and thus are not unique to NXP. Users with Linux experience will find them familiar. They include:

- ifconfig, ip, route, etc. (configure networking)
- ftp (transfer files via the network)
- scp (another way to transfer files via the network)
- date and rdate (set date and time)
- fdisk (partition disks and disk-type devices)
- mkfs (make file systems)
- tar (extract file system images, and more)
- fsck (check file systems)
- mount/umount (mount and umount file systems)

2.1.3 fsl-image-full: A Full-Featured Image, Useful Out-of-Box

Contents:

This is a large image that contains many standard Linux commands and features including native (target-resident) versions of the GNU tools including gcc and gdb.

If you boot this image, the resulting Linux environment will be much like a command-line on a full desktop-type Linux system rather than an embedded system.

Purpose:

While this type of image may not be appropriate for a final embedded product, it can be very helpful for many development and evaluation tasks. The reason is the full set of standard Linux facilities that are already present in the image. In fact, users may find that they can use this image instead of installing the Yocto Project-based NXP SDK onto a development system, at least initially.

For this reason, “fsl-image-full” is preinstalled on the disk drives of NXP development boards that have disks.

For example, the image is complete enough that the standard Linux open source command sequence “configure; make” stands a decent chance of working for arbitrary open source packages that do not happen to be on the image already.

To be clear, the NXP SDK is a Yocto Project/Poky-derived embedded distribution. However, the Yocto Project standard Linux package set is large enough that if one enables a lot of the available packages, the result begins to have the feel of desk top Linux. NXP added the special NXP-specific packages, and the result is “fsl-image-full.” It is intended for “out-of-the-box” evaluation because it is rather complete.

2.1.4 fsl-image-core: A Small Image with NXP-Specific Packages Present

Contents:

This is a small image somewhat like “fsl-image-minimal” except it contains all of the NXP-specific SDK packages.

Purpose:

This image is useful for evaluating the NXP-specific software packages in the context of a file system image that is much more embedded-oriented than “fsl-image-full”.

Embedded file system images contain fewer helpful tools and utilities by definition. They are intended to support an embedded product's functionality rather than a developer's tasks. Thus, it can be convenient to begin with "fsl-image-core" for evaluation and planning and then later narrowly extend "fsl-image-minimal" to support your embedded product.

2.1.5 fsl-image-virt: An image for KVM deployment

Contents:

This is an image which contains the specific packages needed to enable virtualization.

Purpose:

This image is useful for virtualization scenarios (KVM, libvirt, lxc). It contains:

- the guest root filesystem
- the guest image (ulmage format for Power based architectures and zImage for ARM based architectures)
- QEMU
- all necessary libraries and tools for libvirt and lxc support

2.2 Essential Build Instructions

The following sections are essential to the build process and must be performed when using Yocto Project to build the SDK. In order to install the SDK, prepare the host environment, setup Poky, and perform builds, follow the instructions in the subsequent sections. When these steps are completed, the build process will be complete. Linux images that have been built will be found in the following directory: `build_<machine>/tmp/deploy/images/<machine>`

See [Additional Instructions for Developers](#) for more information on using Yocto Project.

2.2.1 Install the SDK

1. Download and Install SDK 2.0 ISO:

- As a prerequisite, QorIQ-SDK-V2.0-SOURCE-20160527-yocto.iso is required to be installed
- In order to speed up the build, QorIQ-SDK-V2.0-AARCH64-CACHE-20160527-yocto.iso is also recommended to be installed.

2. Extract the LS1046A v0.4 tarball, and run the 'install' script to install the updates and sources to ISO install folder:

```
$ tar -xjf LS1046A-SDK-V0.4.tar.bz2
$ ./LS1046A-SDK-V0.4/install
```

- During the install process, the user will be prompted to input the SDK 2.0 ISO installed location, i.e., `<ISO_INSTALL_PATH>/QorIQ-SDK-V2.0-20160527-yocto`.

2.2.2 Set Up Host Environment

Yocto Project requires some packages to be installed on the host.

The following steps are used to prepare the Yocto Project environment.

In general, Yocto Project can work on most recent Linux distributions with Python-2.7.3 or greater (excluding python3 which is not supported), git-1.7.8 or greater, tar-1.24 or greater and required packages installed. The default Python is not 2.7.x on some Linux distros, e.g. CentOS 6.5 installs python 2.6.6. Please follow below instructions to install the Python 2.7.x in custom path instead of override the system default python, the override may cause system utilities breaking.

```
$ wget https://www.python.org/ftp/python/2.7.6/Python-2.7.6.tar.xz
[NOTE: Python 2.7.3 and python 2.7.5 can be used as well.]
```

Getting Started

Essential Build Instructions

```
$ tar -xf Python-2.7.6.tar.xz
$ cd Python-2.7.6
$ ./configure --prefix=/opt/python-2.7.6
$ make
$ sudo make install
```

Please run below export command to ensure python 2.7.x is used for Yocto build.

```
$ export PATH=/opt/python-2.7.6/bin:$PATH
```

Yocto Project supports typical Linux distributions: Ubuntu, Fedora, CentOS, Debian, OpenSUSE, etc. More Linux distributions are continually being verified. This SDK has been verified on following Linux distributions: Ubuntu 14.04, CentOS-7.1.1503, Debian 8.2, Fedora 22 and OpenSUSE 13.2

For a list of the Linux distributions tested by the Yocto Project community see SANITY_TESTED_DISTROS in poky/meta-yocto/conf/distro/poky.conf.

The following is the detailed package list on the CentOS hosts:

```
$ sudo yum install gawk make wget tar bzip2 gzip python unzip perl patch \
diffutils diffstat git cpp gcc gcc-c++ glibc-devel texinfo chrpath socat SDL-devel xterm
```

For the Fedora hosts:

```
$ sudo yum install gawk make wget tar bzip2 gzip python unzip perl patch \
diffutils diffstat git cpp gcc gcc-c++ glibc-devel texinfo chrpath \
ccache perl-Data-Dumper perl-Text-ParseWords perl-Thread-Queue socat \
findutils which SDL-devel xterm
```

For Ubuntu and Debian hosts:

```
$ sudo apt-get install gawk wget git-core diffstat unzip texinfo gcc-multilib \
build-essential chrpath socat libsdl1.2-dev xterm
```

Extra packages are needed for Ubuntu-64b:

```
$ sudo apt-get install lib32z1 lib32ncurses5 lib32bz2-1.0 ia32-libs lib32ncurses5-dev
```

For OpenSUSE host:

```
$ sudo zypper install python gcc gcc-c++ libtool subversion git chrpath automake make wget
diffstat makeinfo freeglut-devel libSDL-devel
```

2.2.3 Set Up Poky

Source the following poky script to set up your environment for your particular NXP platform. This script needs to be run once for each terminal, before you begin building source code.

```
$ . ./fsl-setup-env -m <machine>
```

For example:

```
$ . ./fsl-setup-env -m ls1046ardb
```

The following shows the usage text for the `fsl-setup-env` command:

Usage:

```
$ ./fsl-setup-env -h
Usage: . fsl-setup-env -m <machine>

Supported machines
ls1046ardb

Optional parameters:
* [-m machine]: the target machine to be built.
* [-b path]: non-default path of project build folder.
* [-j jobs]: number of jobs for make to spawn during the compilation stage.
* [-t tasks]: number of BitBake tasks that can be issued in parallel.
* [-d path]: non-default path of DL_DIR (downloaded source)
* [-c path]: non-default path of SSTATE_DIR (shared state Cache)
* [-g]: enable Carrier Grade Linux
* [-l]: lite mode. To help conserve disk space, deletes the building
        directory once the package is built.
* [-h]: help
```

2.2.4 Build Kernel itb

Follow these steps to do builds using Yocto Project. Be sure to set up the host environment before doing these steps.

1.

```
$ cd <ISO-install-dir>/QorIQ-SDK-V2.0-20160527-yocto
```
2.

```
$ bitbake fsl-image-kernelitb
```

NOTE

fsl-image-kernelitb: is a FIT image that includes the Linux image, dtb and rootfs image. For additional Yocto Project usage information, refer to <https://www.yoctoproject.org/>

2.3 Additional Instructions for Developers

This section describes additional "How To" instructions for getting started with Yocto Project.

Each set of instructions is aimed towards developers that are interested in modifying and configuring the source beyond the default build. Each section will describe instructions on how to use Yocto Project to achieve a specific development task.

2.3.1 Customize U-Boot

To Modify U-Boot Configuration:

Modify UBOOT_CONFIG. Values for UBOOT_CONFIG are listed in <sdk-install-dir>/sources/meta-nxp-npi-ls1046a/conf/machine/<machine>.conf

e.g. UBOOT_CONFIG ??= "nor"

To Modify U-Boot Source Code:

If source code has already been installed, please skip steps 1 & 2 and proceed modifying source code.

1.

```
$ bitbake -c cleansstate u-boot
```

NOTE

Other helpful bitbake cleaning commands:

```
bitbake -c clean <target>
```

- Removes work directory in build_<machine>/tmp/work

```
bitbake -c cleansstate <target>
```

- Removes work directory in build_<machine>/tmp/work
- Removes cache files in <sdk-install-dir>/sstate-cache/ directory.

2. \$ bitbake -c patch u-boot

3. \$ cd <S> and modify the source code. Follow instructions to "Rebuild U-Boot Image" when you are finished modifying source code.

NOTE

Use `bitbake -e <package-name> | grep ^S=` to get value of <S>, the package source code directory.

To Rebuild U-Boot Image:

1. \$ cd build_<machine>

2. \$ bitbake -c compile -f u-boot

3. \$ bitbake u-boot

NOTE

U-Boot image can be found in build_<machine>/tmp/deploy/images/<machine>/

2.3.2 Customize Linux Kernel

How to Configure, Modify, or Rebuild the Linux Kernel

To Modify Kernel Source Code:

If source code has already been installed, please skip steps 1 & 2 and proceed modifying source code.

1. \$ bitbake -c cleansstate virtual/kernel

2. \$ bitbake -c patch virtual/kernel

3. \$ cd <S> and change the source code. Follow instructions to "Rebuild Kernel Image" when you are finished modifying source code.

NOTE

Use `bitbake -e <package-name> | grep ^S=` get the value of <S> (package source code directory).

To Change the Kernel defconfig:

1. Update `KERNEL_DEFCONFIG` variable in <sdk-install-dir>/sources/meta-freescale/conf/machine/<machine>.conf

To Change dts:

1. Update `KERNEL_DEVICETREE` variable in <sdk-install-dir>/sources/meta-freescale/conf/machine/<machine>.conf

To Do menuconfig:

1. \$ bitbake -c menuconfig virtual/kernel

NOTE

If you are going to reuse this new kernel configuration for future builds, tell menuconfig to "Save Configuration to Alternate File" and give it an absolute path of `/tmp/my-defconfig`. If you do not do this, the new `defconfig` file will be removed when doing a `clean/cleansstate` for kernel.

NOTE

This runs the normal kernel menuconfig within the Yocto Project environment. If the kernel configure UI cannot open, edit `<yocto_install_path>/build_<machine>/conf/local.conf` and add the following based on your environment (prior to issuing the bitbake command).

For a non-X11 environment:

- `OE_TERMINAL = "screen"`

The following commands can be used for the other environments:

For a GNOME environment (default):

- `OE_TERMINAL = "gnome"`

For a KDE environment:

- `OE_TERMINAL = "konsole"`

For non-GNOME and non-KDE environments:

- `OE_TERMINAL = "auto"`
-

To Rebuild Kernel Image:

1. `$ cd build_<machine>`
2. `$ bitbake -c compile -f virtual/kernel`
3. `$ bitbake virtual/kernel`

NOTE

Kernel images can be found in `<sdk-install-dir>/build_<machine>/tmp/ deploy/ images/<machine>/`

2.3.3 Build Native Packages

How to Build Native Packages for the Host

Native packages such as `cst-native` are supported in Yocto Project. To build a native package, do the following:

```
$ bitbake cst-native
```

NOTE

The binaries can be found in `build_<machine>/tmp/sysroot/`

Chapter 3

Deploy U-Boot, Linux Kernel, and Root Filesystem to a Reference Design Board (RDB)

3.1 Introduction

This chapter describes how to deploy U-Boot, Linux kernel and root file system to the NXP Reference Design Board (RDB). The guide starts with generic host and target board pre-requisites. This is followed by board-specific configuration:

- Switch Settings
- U-Boot environment Variables
- Management Complex(MC), DPC, DPL
- Reset Configuration Word (RCW) and Ethernet Interfaces (if applicable)
- System Memory Map
- Flash Bank Usage

The "Switch Settings" section within each guide shows the default switch settings for the reference design board. If more information is needed beyond the scope of the default configuration, refer to the reference design board's Quick Start Guide and Reference Manual/ User Manual.

For reference design boards with more than one QSPI flash, the "Programming a New U-Boot, RCW, Management Complex" section describes how the user can individually or simultaneously update U-Boot, RCW, and Management Complex, DPC, DPL by flashing them to the board's second QSPI flash prior to deployment.

Once the board is set-up and configured appropriately, select one of the following deployment methods:

- Ramdisk deployment from TFTP
- Ramdisk deployment from Flash
- NFS deployment
- Harddisk deployment (if applicable)
- SD deployment (if applicable)
- Hypervisor deployment (if applicable)

Each of these guides will step you through the deployment method of your choice.

3.2 Basic Host Set-up

Since TFTP will be used to download files onto the target board, a TFTP server must be running on your host system. If you are going to use NFS deployment then an NFS server must also be running on your host system.

Once TFTP and NFS servers are installed, use the following generic instructions to complete the host set-up:

1. Create the tftboot directory.

```
mkdir /tftboot
```

2. Copy over kernel, bootloader, and flash filesystem images for your deployment to the /tftpboot directory:

```
cp <yocto_work_dir>/build_<platform>_release/tmp/deploy/images/* /tftpboot
```

3. Use Yocto Project to generate a tar.gz type file system, and uncompress it in <nfs_root_path>.

4. Edit /etc/exports and add the following line:

```
<nfs_root_path> <target_board_IP> (rw,no_root_squash, async)
```

5. Edit /etc/xinetd.d/tftp to enable TFTP server:

```
service tftp
{
  disable= no
  socket_type= dgram
  protocol= udp
  wait= yes
  user= root
  server= /usr/sbin/in.tftpd
  server_args= /tftpboot
}
```

6. Restart the nfs and tftp servers on your host:

```
/etc/init.d/xinetd restart
/etc/init.d/nfs restart
```

7. Connect the board to the network.
8. Connect the target to the host via a cross cable serial connection.
9. Open a serial console tool on the host system and set it up to talk to the target board:
 - Select appropriate serial device.
 - Configure the serial port with the following settings: Baud rate = 115,200; Data = 8 bit; Parity = none; Stop = 1 bit; Flow control = none.
 - Power on board and see the console prompt.

NOTE

- (i) The Linux distribution running on your host will determine the specific instructions to use.
- (ii) Steps 3 and 4 are only necessary when using NFS deployment.

3.3 Supported Boards

3.3.1 LS1046ARDB

3.3.1.1 Overview

The LS1046A reference design board (RDB) is the high-performance computing, evaluation, and development platform that supports the QorIQ LS1046A processor. This guide provides board-specific configurations and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

3.3.1.2 Switch Settings

The RDB has user selectable switches for evaluating different boot options for the LS1046A device. Table below lists the default switch settings and the description of these settings. ('0' is OFF, '1' is ON.)

Table 1. Default Switch Settings

	1	2	3	4	5	6	7	8
SW3	0	1	0	0	0	1	1	0
SW4	0	0	1	1	1	0	1	1
SW5	0	0	1	0	0	0	1	0

Below are additional switch settings for alternate boot devices. Please note that changing the boot device configuration may require additional changes in the RCW or in other code images.

Table 2. LS1046ARDB Switch Settings

Boot Source	Switch
QSPI flash 0 (bank0)	SW5[1-8] +SW4[1] = 0b'00100010_0 SW3[3-5]= 0b'000
QSPI flash 1 (bank4)	SW5[1-8] +SW4[1] = 0b'00100010_0 SW3[3-5]= 0b'001
SD	SW5[1-8] +SW4[1] = 0b'00100000_0

3.3.1.3 U-Boot Environment Variables

The following sections will guide the users on how to set the U-Boot environment and configure the U-Boot network parameters.

3.3.1.3.1 U-Boot Environment Variable "hwconfig"

Environment variable "hwconfig" is used within the U-Boot bootloader to convey information about desired hardware configurations. It is an ordinary environment variable in that:

- It can be set in the U-Boot prompt using the "setenv" command.
- It can be removed from the U-Boot environment by setting it to an empty value, i.e.

```
=>setenv hwconfig
```

- It can be modified in the U-Boot command prompt using the "editenv" command.
- It can be saved in the U-Boot environment via the "saveenv" command.

Variable "hwconfig" is set to a sequence of *option:value* entries separated by semicolons.

The default setting for for "hwconfig" on LS1046ARDB is as follows:

```
hwconfig = fsl_ddr:bank_intlv=auto
```

3.3.1.3.2 Configuring U-Boot Network Parameters

To support TFTP based deployments, set up the U-Boot environment once, and save it, so that settings persist on subsequent resets.

```
=>setenv ipaddr <board_ipaddress>  
=>setenv serverip <tftp_serverip>  
=>setenv gatewayip <your_gatewayip>  
=>setenv ethaddr <mac addr0>  
=>setenv eth1addr <mac addr1>  
=>setenv eth2addr <mac addr2>  
=>setenv eth3addr <mac addr3>  
=>setenv ethprime <ethx>  
=>setenv ethact <ethx>  
=>setenv netmask 255.255.x.x  
=>saveenv
```

NOTE

* <ethx> is the Ethernet port on the board connected to the Linux boot server. "netmask" is subnet mask for the Linux boot server's network.

Below is one example of the MAC address configuration corresponding to the set up above. Change these values to appropriate MAC addresses appropriate for your board.

```
=>setenv ethaddr 00:e0:0c:00:89:00  
=>setenv eth1addr 00:e0:0c:00:89:01  
=>setenv eth2addr 00:e0:0c:00:89:02  
=>setenv eth3addr 00:e0:0c:00:89:03  
=>saveenv
```

NOTE

1. For boards with more network interfaces, additional environment variables need to be set (e.g., eth6addr, eth7addr,...).
2. In the overwhelming majority of cases, eth<*>addr can be autoset.

Now the flashed version of U-Boot is ready for performing TFTP based deployments.

3.3.1.4 Frame Manager Microcode (FMan Ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for LS1046A version information and also for the version of FMan microcode currently flashed on the LS1046A (e.g., microcode version 108.4.19). For QorIQ SDK 2.0, one of the following FMan microcode binaries should be used:

```
fsl_fman_ucode_t2080_r1.1_106_4_18.bin
```

NOTE

- (i) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.
- (ii) For instructions on how to flash a new FMan microcode image, see [Programming a New U-boot, RCW, FMan Microcode](#).
- (iii) Using a microcode binary from an older SDK (e.g., SDK 1.9) with a Linux kernel from SDK 2.0 is not supported.

3.3.1.5 RCW (Reset Configuration Word) and Ethernet Interfaces

The RCW directories' names conform to the following naming convention:

```
ab_cdef_ghij
```

Table 3. RCW directories Naming Convention Legend

Slot	Convention
a	'R' indicates RGMII1 @DTSEC3 is supported 'N' if not available/not used
b	'R' indicates RGMII2@DTSEC4 is supported 'N' if not available/not used
c	What is available in SerDes1 Lane 0
d	What is available in SerDes1 Lane 1
e	What is available in SerDes1 Lane 2
f	What is available in SerDes1 Lane 3
g	What is available in SerDes2 Lane 0
h	What is available in SerDes2 Lane 1
i	What is available in SerDes2 Lane 2
j	What is available in SerDes2 Lane 3

Table 4. For Lanes (C through H)

Flag	Convention
'N'	NULL, not available/not used
'P'	PCIe
'X'	XAUI
'S'	SGMII
'Q'	QSGMII
'F'	XFI
'H'	SATA
'A'	AURORA

For example,

```
RR_FFPPPN_1133_5559
```

means:

- RGMII1@DTSEC3 on board
- RGMII2@DTSEC4 on board
- XFI9 on Copper port
- XFI10 on SFP cage
- SGMII5 on SGMII1 port
- SGMII6 on SGMII2 port
- PCIe1 on miniPCIe slot
- PCIe2 on Slot 2
- PCIe3 on Slot 3
- SATA
- SERDES1 Protocol is 0x1133
- SERDES2 Protocol is 0x5559

The RCW file names for the ls1046ardb conform to the following naming convention:

```
rcw_<frequency>_<specialsetting>.rcw
```

Table 5. RCW Files Naming Convention Legend

Code		Convention
frequency		Core frequency(MHZ)
specialsetting	bootmode	QSPI/SD/EMMC
	special support	emmc: eMMC boot sdboot: SD boot sben: Secure boot qspiboot: QSPI boot

For example,

```
rcw_1600_sdboot.rcw means rcw for core frequency of 1600MHz with sd boot.
```

```
ls1046ardb/RR_FFPPPN_1133_5559/rcw_1600_qspiboot.rcw means rcw for core frequency of 1600MHz with QSPI boot.
```

The following RCW binaries are used on the ls1046ardb:

```
RR_FFPPPN_1133_5559/rcw_1600_qspiboot.bin
```

```
RR_FFPPPN_1133_5559/rcw_1600_qspiboot_sben.bin
```

3.3.1.6 System Memory Map

In 64-bit u-boot, there is a 1:1 mapping of physical address and effective address. After system startup, the boot loader maps physical address and effective address as shown in the following table:

Start Physical Address	End Physical Address	Memory Type	Size
0x00_0000_0000	0x00_00FF_FFFF	Secure Boot ROM	1MB
0x00_0100_0000	0x00_0FFF_FFFF	CCSR	240MB
0x00_1000_0000	0x00_1000_FFFF	OCRAM0	64KB
0x00_1001_0000	0x00_1001_FFFF	OCRAM1	64KB
0x00_2000_0000	0x00_23FF_FFFF	DCSR	64MB
0x00_4000_0000	0x00_5FFF_FFFF	QSPI	512MB
0x00_6000_0000	0x00_7FFF_FFFF	IFC region	512MB
0x00_8000_0000	0x00_FFFF_FFFF	DRAM	2GB
0x05_0000_0000	0x05_07FF_FFFF	QMAN S/W Portal	128M
0x05_0800_0000	0x05_0FFF_FFFF	BMAN S/W Portal	128M
0x08_8000_0000	0x0F_FFFF_FFFF	DRAM	30GB
0x40_0000_0000	0x47_FFFF_FFFF	PCI Express1	32G
0x48_0000_0000	0x4F_FFFF_FFFF	PCI Express2	32G
0x50_0000_0000	0x57_FFFF_FFFF	PCI Express3	32G

3.3.1.7 Flash Bank Usage

LS1046ARDB has 2 QSPI flash connected over QSPI controller.

Only one QSPI flash is available at a time depending upon the board switch settings. These switch settings can also be overridden by CPLD commands.

To protect the default U-Boot in flash0 (aka bank0), it is a convention employed by NXP to deploy work images into the flash1 (aka bank4), and then switch to the flash1 (aka bank4) for testing. Switching to the flash1 (aka bank4) can be done in software using CPLD commands and effectively swaps the flash0 (aka bank0) with the flash1 (aka bank4). This protects flash1 and keeps the board bootable under all circumstances.

To determine the current bank, refer to the U-Boot log:

```
U-Boot 2016.01 (Aug 19 2016 - 16:59:27 +0800)

SoC: LS1046E (0x87070010)
Clock Configuration:
  CPU0(A72):1600 MHz CPU1(A72):1600 MHz CPU2(A72):1600 MHz
  CPU3(A72):1600 MHz
  Bus:      600 MHz DDR:      2100 MT/s FMAN:      700 MHz
Reset Configuration Word (RCW):
  00000000: 0c150010 0e000000 00000000 00000000
  00000010: 11335559 40005012 40025000 c1000000
  00000020: 00000000 00000000 00000000 00238800
  00000030: 20124000 00003101 00000096 00000001
I2C: ready
Model: LS1046A RDB Board
Board: LS1046ARDB, boot from QSPI vBank 0
CPLD: V2.2
PCBA: V2.0
SERDES Reference Clocks:
SD1_CLK1 = 156.25MHZ, SD1_CLK2 = 100.00MHZ
DRAM: Initializing DDR...using SPD
Detected UDIMM 18ASF1G72AZ-2G3B1
8 GiB (DDR4, 64-bit, CL=15, ECC on)
  DDR Chip-Select Interleaving Mode: CS0+CS1
SEC0: RNG instantiated
PPA Firmware: Version 0.2
Using SERDES1 Protocol: 4403 (0x1133)
Using SERDES2 Protocol: 21849 (0x5559)
NAND: 512 MiB
MMC: FSL_SDHC: 0
SF: Detected S25FL512S_256K with page size 256 Bytes, erase size 256 KiB, total 64 MiB
EEPROM: Invalid ID (5a 5a 5a 5a)
PCIE1: Root Complex no link, regs @ 0x3400000
PCIE2: Root Complex no link, regs @ 0x3500000
PCIE3: Root Complex no link, regs @ 0x3600000
In: serial
Out: serial
Err: serial
SATA link 0 timeout.
AHCI 0001.0301 32 slots 1 ports 6 Gbps 0x1 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc apst
Found 0 device(s).
SCSI: Net: SF: Detected S25FL512S_256K with page size 256 Bytes, erase size 256 KiB, total 64 MiB
Fman1: Uploading microcode version 106.4.15
FM1@DTSEC3 [PRIME], FM1@DTSEC4, FM1@DTSEC5, FM1@DTSEC6, FM1@TGEC1, FM1@TGEC2
Hit any key to stop autoboot: 0
=>
```

Bank switching can be done in U-Boot using the following statements:

- Switch to QSPI bank 0(default) for RDB:

```
=>cpld reset
```

- Switch to QSPI bank 4 for RDB:

```
=>cpld reset altbank
```

The table below shows the QSPI flash memory map for LS1046ARDB:

Table 6. QSPI Flash Memory Map for LS1046ARDB

Start	End Offset	Description	Size
0x40000000	0x400FFFFFFF	bank0 rcw + pbi	1 M
0x40100000	0x401FFFFFFF	bank0 U-Boot image	1 M
0x40200000	0x402FFFFFFF	bank0 U-Boot Env	1 M
0x40300000	0x403FFFFFFF	bank0 Fman ucode	1 M
0x40400000	0x404FFFFFFF	bank0 UEFI	1 M
0x40500000	0x406FFFFFFF	bank0 Primary Protected Application (PPA)	2 M
0x40700000	0x408FFFFFFF	bank0 Secure boot header + bootscript	2 M
0x40900000	0x40FFFFFFF	bank0 reserved	7 M
0x41000000	0x43FFFFFFF	bank0 FIT Image	48 M
0x44000000	0x440FFFFFFF	bank4 rcw + pbi	1 M
0x44100000	0x441FFFFFFF	bank4 U-Boot image	1 M
0x44200000	0x442FFFFFFF	bank4 U-Boot Env	1 M
0x44300000	0x443FFFFFFF	bank4 Fman ucode	1 M
0x44400000	0x444FFFFFFF	bank4 UEFI	1 M
0x44500000	0x446FFFFFFF	bank4 PPA	2 M
0x44700000	0x448FFFFFFF	bank4 Secure boot header + bootscript	2 M
0x44900000	0x44FFFFFFF	bank4 reserved	7 M
0x45000000	0x47FFFFFFF	bank4 FIT Image	48 M

3.3.1.8 Programming a New U-Boot, RCW and FMan Ucode

The following three sections will discuss how to individually update U-Boot, RCW. For specific addresses, please refer to the [QSPI Flash Memory Map](#) as a reference.

Prior to continuing with the following instructions, please refer to [Configuring U-Boot Network Parameters](#) to make sure all necessary U-Boot parameters have been set.

Here are U-Boot commands to switch between the three different boot sources:

1. `cpld reset`: reset to boot from current bank
2. `cpld reset altbank`: reset to boot from alternate bank
3. `cpld reset sd`: reset to boot from SD card

Programming a New U-Boot to QSPI flash

By default, an existing U-Boot is run in QSPI bank 0 after the system is powered on or after a hard reset is performed. To flash U-Boot to the alternate bank, first switch to bank 0 by performing a hard reset or by typing `reset`. Then use the following commands to flash a new U-Boot into the alternate bank and then switch to that alternate bank where the new U-Boot is flashed:

```
=>tftp <uboot_image_addr> <uboot_file_name>.bin
=>sf probe 0:1
```

```
=>sf erase 100000 +$filesize  
=>sf write <uboot_image_addr> 100000 $filesize  
=>cpld reset altbank
```

NOTE

1. Using "sf probe 0:0" could program U-Boot to the current bank.
2. The U-Boot image generated from code needs to be swapped. Yocto Project generates the swapped U-Boot image, which is programmed into QSPI flash. For steps to build U-Boot without using Yocto Project, refer to [QSPI Deployment](#).

Programming a New RCW to QSPI flash

To program a new RCW, first switch to QSPI bank 0 by performing a hard reset or by typing *reset*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash offset 0x0. 0x0 is the offset address of RCW in the QSPI flash. Execute the following commands at the U-Boot prompt to program the RCW to QSPI flash and reset to alternate bank.

```
=>tftp <rcw_image_addr> <rcw_file_name>.bin  
=>sf probe 0:1  
=>sf erase 0 +$filesize  
=>sf write <rcw_image_addr> 0 $filesize;  
=>cpld reset altbank
```

NOTE

Using "sf probe 0:0" could program RCW to the current bank.

Programming a New U-Boot to SD card

To program U-boot, first boot the board to u-boot. Next, load the new U-Boot SD boot image(u-boot-with-spl-pbl.bin) to RAM by downloading it via TFTP and then copying it to SD card with blk offset 0x8. Execute the following commands at the U-Boot prompt to program the U-Boot to SD card and reset to sd boot.

```
=>tftp <uboot_image_addr> u-boot-with-spl-pbl.bin  
=>mmc erase 8 0x800  
=>mmc write <uboot_image_addr> 8 0x800  
=>cpld reset sd
```

Program the image to SD card in Linux.

```
dd if=u-boot-with-spl-pbl-sd.bin of=/dev/sdb bs=512 seek=8
```

Programming a New FMan Microcode

Program a new microcode to QSPI flash 1(bank 4):

To program a new microcode, first switch to QSPI bank 0 by performing a hard reset or by typing *cpld reset*. Next, load the new microcode to RAM by downloading it via TFTP and then writing it to flash offset 0x300000. 0x300000 is the offset of ucode in the QSPI flash. Then execute the following commands at the U-Boot prompt to program the ucode to the boot device.

```
=>tftp <ucode_image_addr> <ucode_file_name>.bin  
=>sf probe 0:1  
=>sf erase 300000 +$filesize  
=>sf write <ucode_image_addr> 300000 $filesize
```

NOTE: Using "sf probe 0:0" could program microcode to the current bank.

Program a new microcode to SD Card:

```
=>tftp <uimage_addr> <uimage_file_name>.bin
=>mmc write <uimage_addr> 820 50
```

3.3.1.9 Deployment

Each of these guides will step you through the deployment method of your choice. Please refer to the board's NOR Flash Memory Map within *Flash Bank Usage* as reference for specific addresses.

3.3.1.9.1 FIT Image Deployment from TFTP

1. Setting U-Boot Environment

Before performing FIT image deployment, the U-Boot environment variables need to be configured.

Refer to [Configuring U-Boot Network Parameters](#) on page 33 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for FIT image deployment from TFTP:

```
=>setenv bootargs 'root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500 console=ttyS0,115200'
=>saveenv
```

2. Booting Up the System

Execute the following commands to TFTP the image to the board, then boot into Linux.

```
=>tftp a0000000 <fit_image_name>
=>bootm a0000000
```

Now the board will boot into Linux .

3.3.1.9.2 FIT Image Deployment from Flash

1. Setting U-Boot Environment

Before performing FIT image from flash, the U-Boot environment variables need to be configured. Refer to [Configuring U-Boot Network Parameters](#) to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for deployment from flash:

```
=>setenv bootargs root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500 console=ttyS0,115200
=>setenv bootcmd sf probe 0:0;sf read <fit_image_addr> 1000000 2800000;bootm <fit_image_addr>
=>saveenv
```

2. Programming FIT image to QSPI Flash

The FIT image should be downloaded to the RAM address using TFTP then copied to flash offset 0x1000000 as per "QSPI memory map" in [Flash Bank Usage](#). At the U-Boot prompt, use the following commands to program the image to QSPI current bank:

```
=>tftp <fit_image_addr> <fit_image_name>
=>sf probe 0:0
=>sf erase 0x1000000 +$filesize
=>sf write <fit_image_addr> 0x1000000 $filesize
```

3. Booting Up the System

The kernel can boot up automatically after the board is powered on, or the following command can be used to boot up the board at U-Boot prompt:

```
=>boot
```

or

```
=> bootm <fit_image_addr>
```

3.3.1.9.3 NFS Deployment

1. Generating File System

Use Yocto to generate a tar.gz type file system, and uncompress it for NFS deployment.

2. Setting Host NFS Server Environment

a. On the Linux host NFS server, add the following line in the file /etc/exports:

```
nfs_root_path board_ipaddress (rw,no_root_squash,async)
```

b. Restart the NFS service:

```
/etc/init.d/portmap restart
```

```
/etc/init.d/nfs-kernel-server restart
```

NOTE

nfs_root_path: the NFS root directory path on NFS server.

3. Setting U-Boot Environment

The NFS file system generated by Yocto allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to [Configuring U-Boot Network Parameters](#) on page 33 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

```
=>setenv bootargs root=/dev/nfs rw nfsroot=<tftp_serverip>:<nfs_root_path>  
ip=<board_ipaddr>:<tftp_serverip>: <your_gatewayip>:<your_netmask>:<board_name>:<ethx>:off  
console=ttyS0,115200 earlycon=uart8250,mmio,0x21c0500  
=>saveenv
```

NOTE

<ethx> is the port connected on the Linux boot network.

Now U-Boot is ready for NFS deployment.

4. Booting up the System

TFTP the kernel FIT image to the board, then boot it up.

```
=>tftp a0000000 kernel.itb;  
=>bootm a0000000:kernel@1 - a0000000:fdt@1
```

Now the board will boot up with NFS filesystem.

3.3.1.9.4 SD Deployment

Partition SD Card

1. Insert SD card into the Linux Host PC.
2. Use the "fdisk" command to repartition the SD card.

```
# fdisk /dev/sdb
```

3. Use the mkfs.ext2 command to create the filesystem.

```
#mkfs.ext2 /dev/sdb1
```

NOTE

The first 2056 sectors of SD card must be remained for u-boot image

FIT Kernel Image and Root File System Deployment from SD Card

1. Insert SD card into the Linux Host PC.
2. Create temp director in host PC and mount the ext2 partition to the temp

```
#mkdir temp  
#mount /dev/sdb1 temp
```

3. Copy the FIT Kernel Image to the SD card partition.

```
#cp kernel.itb temp/
```

4. Copy the Root File System to the SD card partition.

```
#cp fsl-image-core-ls1043ardb_<release date>.rootfs.tar.gz temp/  
#tar xvfz fsl-image-core-ls1043ardb_<release date>.rootfs.tar.gz  
#rm fsl-image-core-ls1043ardb_<release date>.rootfs.tar.gz temp/
```

5. Umount the temp director

```
#umount temp
```

U-Boot Image Deployment from SD Card

- Form Linux Host PC

1. Insert SD card into Host.
2. Use the "dd" command

```
# dd if=u-boot-with-spl-pbl.bin of=/dev/sdb seek=8 bs=512
```

- Form ls1043ardb Board

1. Insert SD card into target board and power on.
2. Programming U-boot image to SD Card

```
=> tftpbboot 82000000 u-boot-with-spl-pbl.bin  
=> mmc write 82000000 8 800
```

```
=> cpld reset sd
```

Setting U-Boot Environment

- Execute the following commands at the U-Boot prompt

```
=> setenv bootcmd "ext2load mmc 0 a0000000 kernel.itb && bootm a0000000"
```

- Using the Ramdisk as the Root File System

```
=> setenv bootargs "root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500 console=ttyS0,115200"
```

- Using the Ext2 Partition of SD card as the Root File System

```
=> setenv bootargs "root=/dev/mmcblk0p1 rw earlycon=uart8250,0x21c0500 console=ttyS0,115200"
```

- Saving the environment

```
=>saveenv
```

NOTE

The `kernel.itb` is the name of your FIT Image, you can use the `ext2ls` command to list it at the U-Boot prompt

3.3.1.9.5 QSPI Deployment

Build U-Boot image for QSPI without Yocto Project

The U-Boot image generated from code needs to be swapped for QSPI boot. Yocto generates the swapped U-Boot image, which is programmed into QSPI flash. Below are steps to build U-Boot without Yocto:

1. Compile QSPI boot image(enable QSPI):

```
$make distclean ARCH=aarch64 CROSS_COMPILE=${toolchain_path}/bin/aarch64-linux-gnu-
```

```
$make ARCH=aarch64 $ls1046ardb_qspi_defconfig
```

```
$make CROSS_COMPILE=${toolchain_path}/bin/aarch64-linux-gnu- -j4
```

2. Swap the bytes for QSPI boot:

```
$tclsh byte_swap.tcl rcw_1600_qspiboot.bin rcw_1600_qspiboot_swap.bin 8
```

```
$tclsh byte_swap.tcl u-boot-dtb.bin u-boot_swap.bin 8
```

The `byte_swap.tcl` script is a shareable tool and can be found under `rcw/ls1046ardb/` directory.

3. Switch to QSPI altbank:

```
=>cpld reset altbank
```

Or set switches referring to board configurations [Switch Settings](#) and power on the board from QSPI boot.

Deploy U-Boot, Linux Kernel, and Root Filesystem to a Reference Design Board (RDB)
Supported Boards

Build Linux image and FIT image for QSPI boot

1. Compile Linux kernel image:

```
$make distclean ARCH=aarch64 CROSS_COMPILE=${toolchain_path}/bin/aarch64-linux-gnu-
```

```
$make ARCH=arm64 $make defconfig freescale.config
```

```
$make CROSS_COMPILE=${toolchain_path}/bin/aarch64-linux-gnu- -j4
```

2. Make it into kernel.itb:

```
$mkimage -f kernel-ls1046a-rdb.its kernel.itb
```

To boot kernel, please refer to [FIT Image Deployment from TFTP](#) and [FIT Image Deployment from Flash](#).

3.3.1.10 Check 'Link Up' for Serial Ethernet Interfaces

Check Communication to External PHY

In order to check if U-Boot can communicate with the PHYs on the board, use the U-Boot command *mdio list*. The U-Boot command *mdio list* will display all manageable Ethernet PHYs.

Example:

```
=> mdio list
FSL_MDIO0:
 1 - RealTek RTL8211F <--> FM1@DTSEC3
 2 - RealTek RTL8211F <--> FM1@DTSEC4
 3 - RealTek RTL8211F <--> FM1@DTSEC5
 4 - RealTek RTL8211F <--> FM1@DTSEC6
FM_TGEC_MDIO:
 0 - Aquantia AQR107 <--> FM1@TGEC1
```

The results from the above *mdio list* command show that U-Boot was able to see PHYs on each of the 4 DTSEC interfaces and on the 10GEC interface. If you see "Generic" reported, it is an indication that something is there but the ls1046ardb can't communicate with the device/port.

Check Link Status for External PHY

In order to check the status of a SGMII link, you can use the *mdio read* command. Since this is a Clause 22 device, we pass two arguments to the *mdio read* command.

```
mdio read <PHY address> <REGISTER Address>
```

Example:

```
=> mdio read FM1@DTSEC3 1
Reading from bus FSL_MDIO0
PHY at address 1:
1 - 0x79ad
```

The link partner ("copper side") link status bit is in Register #1 on the PHY. The 'Link Status' bit is bit #2 (from the left) of the last nibble. In the above example the nibble of interest is "d" (d = b'1101'), and therefore the 'Link Status' = 1, which means 'link up'. If the link were down this bit would be a "0," and we would see 0x79a9.

3.3.1.11 Basic Networking Ping Test

U-BOOT

The LS1046ARDB has two RGMII ports and two SGMII ports. The log below shows how to ping from FM1@DTSEC3 and FM1@DTSEC4 interfaces.

```
U-Boot 2016.01 (Aug 19 2016 - 16:59:27 +0800)

SoC: LS1046E (0x87070010)
Clock Configuration:
  CPU0(A72):1600 MHz  CPU1(A72):1600 MHz  CPU2(A72):1600 MHz
  CPU3(A72):1600 MHz
  Bus:      600 MHz  DDR:      2100 MT/s  FMAN:      700 MHz
Reset Configuration Word (RCW):
  00000000: 0c150010 0e000000 00000000 00000000
  00000010: 11335559 40005012 40025000 c1000000
  00000020: 00000000 00000000 00000000 00238800
  00000030: 20124000 00003101 00000096 00000001
I2C:  ready
Model: LS1046A RDB Board
Board: LS1046ARDB, boot from QSPI vBank 0
CPLD: V2.2
PCBA: V2.0
SERDES Reference Clocks:
SD1_CLK1 = 156.25MHZ, SD1_CLK2 = 100.00MHZ
DRAM:  Initializing DDR...using SPD
Detected UDIMM 18ASF1G72AZ-2G3B1
8 GiB (DDR4, 64-bit, CL=15, ECC on)
      DDR Chip-Select Interleaving Mode: CS0+CS1
SEC0:  RNG instantiated
PPA Firmware: Version 0.2
Using SERDES1 Protocol: 4403 (0x1133)
Using SERDES2 Protocol: 21849 (0x5559)
NAND:  512 MiB
MMC:   FSL_SDHC: 0
SF: Detected S25FL512S_256K with page size 256 Bytes, erase size 256 KiB, total 64 MiB
EEPROM: Invalid ID (5a 5a 5a 5a)
PCIE1: Root Complex no link, regs @ 0x3400000
PCIE2: Root Complex no link, regs @ 0x3500000
PCIE3: Root Complex no link, regs @ 0x3600000
In:    serial
Out:   serial
Err:   serial
SATA link 0 timeout.
AHCI 0001.0301 32 slots 1 ports 6 Gbps 0x1 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc apst
Found 0 device(s).
SCSI: Net: SF: Detected S25FL512S_256K with page size 256 Bytes, erase size 256 KiB, total 64 MiB
Fman1: Uploading microcode version 106.4.15
FM1@DTSEC3 [PRIME], FM1@DTSEC4, FM1@DTSEC5, FM1@DTSEC6, FM1@TGEC1, FM1@TGEC2
Hit any key to stop autoboot:  0
=> setenv serverip 10.192.208.233
=> setenv ipaddr 10.193.20.129
=> setenv ethaddr 00:e0:0c:00:89:00
=> ping $serverip
Using FM1@DTSEC3 device
host 10.192.208.233 is alive
=> setenv ethact FM1@DTSEC4
```



```
## Loading ramdisk from FIT Image at a0000000 ...
Using 'config@1' configuration
Trying 'ramdisk@1' ramdisk subimage
  Description:  LS1046 Ramdisk
  Type:         RAMDisk Image
  Compression:  gzip compressed
  Data Start:   0xa053e5b8
  Data Size:    21686563 Bytes = 20.7 MiB
  Architecture: AArch64
  OS:          Linux
  Load Address: unavailable
  Entry Point:  unavailable
Verifying Hash Integrity ... OK
## Loading fdt from FIT Image at a0000000 ...
Using 'config@1' configuration
Trying 'fdt@1' fdt subimage
  Description:  Flattened Device Tree blob
  Type:         Flat Device Tree
  Compression:  uncompressed
  Data Start:   0xa053882c
  Data Size:    23809 Bytes = 23.3 KiB
  Architecture: AArch64
Verifying Hash Integrity ... OK
Loading fdt from 0xa053882c to 0x90000000
Booting using the fdt blob at 0x90000000
Uncompressing Kernel Image ... OK
Using Device Tree in place at 0000000090000000, end 0000000090018d00

Starting kernel ...

[ 0.000000] Booting Linux on physical CPU 0x0
[ 0.000000] Initializing cgroup subsys cpu
[ 0.000000] Linux version 4.1.8-gdce5972 (ggy@titan) (gcc version 4.9.1 20140529 (prerelease)
(crosstool-NG linaro-1.13.1-4.9-2014.07 - Linaro GCC 4.9-2014.06) ) #12 SMP PREEMPT Wed Aug 24
12:21:27 CST 2016
[ 0.000000] CPU: AArch64 Processor [410fd082] revision 2
[ 0.000000] Detected PIPT I-cache on CPU0
[ 0.000000] earlycon: Early serial console at MMIO 0x21c0500 (options '')
[ 0.000000] bootconsole [uart0] enabled
[ 0.000000] efi: Getting EFI parameters from FDT:
[ 0.000000] efi: UEFI not found.
[ 0.000000] Reserved memory: initialized node bman-fbpr, compatible id fsl,bman-fbpr
[ 0.000000] Reserved memory: initialized node qman-fqd, compatible id fsl,qman-fqd
[ 0.000000] Reserved memory: initialized node qman-pfdr, compatible id fsl,qman-pfdr
[ 0.000000] cma: Reserved 16 MiB at 0x00000000ff000000
[ 0.000000] psci: probing for conduit method from DT.
[ 0.000000] psci: PSCIv0.2 detected in firmware.
[ 0.000000] psci: Using standard PSCI v0.2 function IDs
[ 0.000000] PERCPU: Embedded 18 pages/cpu @ffff80097fd6d000 s33664 r8192 d31872 u73728
[ 0.000000] Built 1 zonelists in Zone order, mobility grouping on. Total pages: 2063880
[ 0.000000] Kernel command line: console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,mmio,
0x21c0500
[ 0.000000] log_buf_len individual max cpu contribution: 4096 bytes
[ 0.000000] log_buf_len total cpu_extra contributions: 12288 bytes
[ 0.000000] log_buf_len min size: 16384 bytes
[ 0.000000] log_buf_len: 32768 bytes
[ 0.000000] early log buf free: 14320(87%)
[ 0.000000] PID hash table entries: 4096 (order: 3, 32768 bytes)
[ 0.000000] Dentry cache hash table entries: 1048576 (order: 11, 8388608 bytes)
[ 0.000000] Inode-cache hash table entries: 524288 (order: 10, 4194304 bytes)
```

Deploy U-Boot, Linux Kernel, and Root Filesystem to a Reference Design Board (RDB)

Supported Boards

```
[ 0.000000] software IO TLB [mem 0xfafff000-0xfeffff000] (64MB) mapped at [ffff80007afff000-ffff80007effffff]
[ 0.000000] Memory: 8068852K/8386560K available (7754K kernel code, 582K rwdara, 3092K rodata, 476K init, 756K bss, 301324K reserved, 16384K cma-reserved)
[ 0.000000] Virtual kernel memory layout:
[ 0.000000]   vmalloc : 0xffff000000000000 - 0xffff7bffbfff0000 (126974 GB)
[ 0.000000]   vmemmap : 0xffff7bffc0000000 - 0xffff7ffffc0000000 ( 4096 GB maximum)
[ 0.000000]             0xffff7bffc2000000 - 0xffff7bffe7ff8000 ( 607 MB actual)
[ 0.000000]   fixed   : 0xffff7ffffabfd000 - 0xffff7ffffac00000 ( 12 KB)
[ 0.000000]   PCI I/O : 0xffff7ffffae00000 - 0xffff7ffffbe00000 ( 16 MB)
[ 0.000000]   modules : 0xffff7ffffc000000 - 0xffff800000000000 ( 64 MB)
[ 0.000000]   memory  : 0xffff800000000000 - 0xffff80097fe00000 ( 38910 MB)
[ 0.000000]     .init : 0xffff800000b1a000 - 0xffff800000b91000 ( 476 KB)
[ 0.000000]     .text : 0xffff800000080000 - 0xffff800000b19974 ( 10855 KB)
[ 0.000000]     .data : 0xffff800000ba2000 - 0xffff800000c33800 ( 582 KB)
[ 0.000000] Preemptible hierarchical RCU implementation.
[ 0.000000] Additional per-CPU info printed with stalls.
[ 0.000000] RCU restricting CPUs from NR_CPUS=64 to nr_cpu_ids=4.
[ 0.000000] RCU: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=4
[ 0.000000] NR_IRQS:64 nr_irqs:64 0
[ 0.000000] Architected cp15 timer(s) running at 25.00MHz (phys).
[ 0.000000] clocksource arch_sys_counter: mask: 0xffffffffffffff max_cycles: 0x5c40939b5,
max_idle_ns: 440795202646 ns
[ 0.000002] sched_clock: 56 bits at 25MHz, resolution 40ns, wraps every 4398046511100ns
[ 0.008237] Console: colour dummy device 80x25
[ 0.012708] Calibrating delay loop (skipped), value calculated using timer frequency.. 50.00
BogoMIPS (lpj=250000)
[ 0.023127] pid_max: default: 32768 minimum: 301
[ 0.027809] Security Framework initialized
[ 0.031948] Mount-cache hash table entries: 16384 (order: 5, 131072 bytes)
[ 0.038863] Mountpoint-cache hash table entries: 16384 (order: 5, 131072 bytes)
[ 0.046512] Initializing cgroup subsys memory
[ 0.050902] Initializing cgroup subsys hugetlb
[ 0.055451] hw perfevents: enabled with arm/armv8-pmuv3 PMU driver, 7 counters available
[ 0.063605] EFI services will not be available.
[ 0.143758] CPU1: Booted secondary processor
[ 0.143760] Detected PIPT I-cache on CPU1
[ 0.163758] CPU2: Booted secondary processor
[ 0.163761] Detected PIPT I-cache on CPU2
[ 0.183767] CPU3: Booted secondary processor
[ 0.183769] Detected PIPT I-cache on CPU3
[ 0.183802] Brought up 4 CPUs
[ 0.211728] SMP: Total of 4 processors activated.
[ 0.216455] CPU: All CPU(s) started at EL2
[ 0.220751] devtmpfs: initialized
[ 0.226415] DMI not present or invalid.
[ 0.230400] clocksource jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns:
19112604462750000 ns
[ 0.240504] pinctrl core: initialized pinctrl subsystem
[ 0.246350] NET: Registered protocol family 16
[ 0.280834] cpuidle: using governor ladder
[ 0.310840] cpuidle: using governor menu
[ 0.314830] fsl-mc bus type registered
[ 0.318648] MC object device driver fsl_mc_dprc registered
[ 0.324196] MC object device driver fsl_mc_allocator registered
[ 0.330165] Bman ver:0a02,02,01
[ 0.335432] qman-fqd addr 0x9ff000000 size 0x800000
[ 0.340335] qman-pfdr addr 0x9fc000000 size 0x2000000
[ 0.345418] Qman ver:0a01,03,02,01
[ 0.348875] vdso: 2 pages (1 code @ ffff800000ba9000, 1 data @ ffff800000ba8000)
```

```
[ 0.356336] hw-breakpoint: found 6 breakpoint and 4 watchpoint registers.
[ 0.363480] DMA: preallocated 256 KiB pool for atomic allocations
[ 0.369707] Serial: AMBA PL011 UART driver
[ 0.410976] vgaarb: loaded
[ 0.413838] SCSI subsystem initialized
[ 0.417925] usbcore: registered new interface driver usbfs
[ 0.423486] usbcore: registered new interface driver hub
[ 0.428863] usbcore: registered new device driver usb
[ 0.434393] i2c i2c-0: IMX I2C adapter registered
[ 0.439131] i2c i2c-0: can't use DMA
[ 0.442853] i2c i2c-1: IMX I2C adapter registered
[ 0.447588] i2c i2c-1: can't use DMA
[ 0.451279] pps_core: LinuxPPS API ver. 1 registered
[ 0.456269] pps_core: Software ver. 5.3.6 - Copyright 2005-2007 Rodolfo Giometti
[ 0.465487] PTP clock support registered
[ 0.469599] bman-fbpr addr 0x9fe000000 size 0x1000000
[ 0.474692] Bman err interrupt handler present
[ 0.479444] Bman portal initialised, cpu 0
[ 0.483625] Bman portal initialised, cpu 1
[ 0.487806] Bman portal initialised, cpu 2
[ 0.491993] Bman portal initialised, cpu 3
[ 0.496108] Bman portals initialised
[ 0.500587] Qman err interrupt handler present
[ 0.505348] QMan: Allocated lookup table at ffff0000001cd000, entry count 131073
[ 0.646309] Qman portal initialised, cpu 0
[ 0.650890] Qman portal initialised, cpu 1
[ 0.655466] Qman portal initialised, cpu 2
[ 0.660039] Qman portal initialised, cpu 3
[ 0.664158] Qman portals initialised
[ 0.667803] Bman: BPID allocator includes range 32:32
[ 0.672902] Qman: FQID allocator includes range 256:256
[ 0.678158] Qman: FQID allocator includes range 32768:32768
[ 0.683784] Qman: CGRID allocator includes range 0:256
[ 0.689085] Qman: pool channel allocator includes range 1025:15
[ 0.695072] No USDPAA memory, no 'fsl,usdpaa-mem' in device-tree
[ 0.701144] fsl-ifc 1530000.ifc: Freescale Integrated Flash Controller
[ 0.707716] fsl-ifc 1530000.ifc: IFC version 1.4, 8 banks
[ 0.713576] Switched to clocksource arch_sys_counter
[ 0.721658] NET: Registered protocol family 2
[ 0.726268] TCP established hash table entries: 65536 (order: 7, 524288 bytes)
[ 0.733717] TCP bind hash table entries: 65536 (order: 8, 1048576 bytes)
[ 0.740989] TCP: Hash tables configured (established 65536 bind 65536)
[ 0.747612] UDP hash table entries: 4096 (order: 5, 131072 bytes)
[ 0.753781] UDP-Lite hash table entries: 4096 (order: 5, 131072 bytes)
[ 0.760467] NET: Registered protocol family 1
[ 0.764943] RPC: Registered named UNIX socket transport module.
[ 0.770896] RPC: Registered udp transport module.
[ 0.775636] RPC: Registered tcp transport module.
[ 0.780363] RPC: Registered tcp NFSv4.1 backchannel transport module.
[ 0.786923] Trying to unpack rootfs image as initramfs...
[ 0.792505] rootfs image is not initramfs (no cpio magic); looks like an initrd
[ 0.818311] Freeing initrd memory: 21172K (ffff80002053f000 - ffff8000219ec000)
[ 0.826013] kvm [1]: interrupt-controller@1440000 IRQ9
[ 0.831246] kvm [1]: timer IRQ3
[ 0.834412] kvm [1]: Hyp mode initialized successfully
[ 0.840599] futex hash table entries: 1024 (order: 4, 65536 bytes)
[ 0.846867] audit: initializing netlink subsys (disabled)
[ 0.852307] audit: type=2000 audit(0.720:1): initialized
[ 0.857853] HugeTLB registered 2 MB page size, pre-allocated 0 pages
[ 0.864507] VFS: Disk quotas dquot_6.6.0
```

Deploy U-Boot, Linux Kernel, and Root Filesystem to a Reference Design Board (RDB)

Supported Boards

```
[ 0.868469] VFS: Dquot-cache hash table entries: 512 (order 0, 4096 bytes)
[ 0.875691] NFS: Registering the id_resolver key type
[ 0.880784] Key type id_resolver registered
[ 0.885033] Key type id_legacy registered
[ 0.889127] fuse init (API version 7.23)
[ 0.893182] 9p: Installing v9fs 9p2000 file system support
[ 0.899088] io scheduler noop registered
[ 0.903057] io scheduler cfq registered (default)
[ 0.908251] find msi-controller /soc/msi-controller@1580000
[ 0.914857] PCI host bridge /soc/pcie@3400000 ranges:
[ 0.919941]   IO 0x4000010000..0x400001ffff -> 0x00000000
[ 0.925468]   MEM 0x4040000000..0x407ffffff -> 0x40000000
[ 0.931053] layerscape-pcie 3400000.pcie: PCI host bridge to bus 0000:00
[ 0.937801] pci_bus 0000:00: root bus resource [bus 00-ff]
[ 0.943317] pci_bus 0000:00: root bus resource [io 0x0000-0xffff]
[ 0.949561] pci_bus 0000:00: root bus resource [mem 0x4040000000-0x407ffffff] (bus address
[0x40000000-0x7ffffff])
[ 0.960317] pci 0000:00:00.0: bridge configuration invalid ([bus 00-00]), reconfiguring
[ 0.968438] pci 0000:00:00.0: BAR 1: assigned [mem 0x4040000000-0x4043ffffff]
[ 0.975624] pci 0000:00:00.0: BAR 0: assigned [mem 0x4044000000-0x4044ffffff]
[ 0.982802] pci 0000:00:00.0: BAR 6: assigned [mem 0x4045000000-0x4045ffffff pref]
[ 0.990426] pci 0000:00:00.0: PCI bridge to [bus 01]
[ 0.995441] pcieport 0000:00:00.0: enabling device (0000 -> 0002)
[ 1.001640] pcieport 0000:00:00.0: Signaling PME through PCIe PME interrupt
[ 1.008818] PCI host bridge /soc/pcie@3500000 ranges:
[ 1.013907]   IO 0x4800010000..0x480001ffff -> 0x00000000
[ 1.019424]   MEM 0x4840000000..0x487ffffff -> 0x40000000
[ 1.025007] layerscape-pcie 3500000.pcie: PCI host bridge to bus 0001:00
[ 1.031747] pci_bus 0001:00: root bus resource [bus 00-ff]
[ 1.037270] pci_bus 0001:00: root bus resource [io 0x10000-0x1ffff] (bus address [0x0000-0xffff])
[ 1.046289] pci_bus 0001:00: root bus resource [mem 0x4840000000-0x487ffffff] (bus address
[0x40000000-0x7ffffff])
[ 1.057026] pci 0001:00:00.0: bridge configuration invalid ([bus 00-00]), reconfiguring
[ 1.065142] pci 0001:00:00.0: BAR 1: assigned [mem 0x4840000000-0x4843ffffff]
[ 1.072321] pci 0001:00:00.0: BAR 0: assigned [mem 0x4844000000-0x4844ffffff]
[ 1.079505] pci 0001:00:00.0: BAR 6: assigned [mem 0x4845000000-0x4845ffffff pref]
[ 1.087125] pci 0001:00:00.0: PCI bridge to [bus 01]
[ 1.092131] pcieport 0001:00:00.0: enabling device (0000 -> 0002)
[ 1.098334] pcieport 0001:00:00.0: Signaling PME through PCIe PME interrupt
[ 1.105508] PCI host bridge /soc/pcie@3600000 ranges:
[ 1.110592]   IO 0x5000010000..0x500001ffff -> 0x00000000
[ 1.116115]   MEM 0x5040000000..0x507ffffff -> 0x40000000
[ 1.121692] layerscape-pcie 3600000.pcie: PCI host bridge to bus 0002:00
[ 1.128458] pci_bus 0002:00: root bus resource [bus 00-ff]
[ 1.133979] pci_bus 0002:00: root bus resource [io 0x20000-0x2ffff] (bus address [0x0000-0xffff])
[ 1.142994] pci_bus 0002:00: root bus resource [mem 0x5040000000-0x507ffffff] (bus address
[0x40000000-0x7ffffff])
[ 1.153738] pci 0002:00:00.0: bridge configuration invalid ([bus 00-00]), reconfiguring
[ 1.161846] pci 0002:00:00.0: BAR 1: assigned [mem 0x5040000000-0x5043ffffff]
[ 1.169032] pci 0002:00:00.0: BAR 0: assigned [mem 0x5044000000-0x5044ffffff]
[ 1.176215] pci 0002:00:00.0: BAR 6: assigned [mem 0x5045000000-0x5045ffffff pref]
[ 1.183834] pci 0002:00:00.0: PCI bridge to [bus 01]
[ 1.188841] pcieport 0002:00:00.0: enabling device (0000 -> 0002)
[ 1.195046] pcieport 0002:00:00.0: Signaling PME through PCIe PME interrupt
[ 1.202340] Freescale LS2 console driver
[ 1.206336] fsl-ls2-console: device fsl_mc_console registered
[ 1.212151] fsl-ls2-console: device fsl_aiop_console registered
[ 1.219546] Serial: 8250/16550 driver, 4 ports, IRQ sharing disabled
[ 1.226540] msm_serial: driver initialized
[ 1.230858] console [ttyS0] disabled
```

```
[ 1.234470] 21c0500.serial: ttyS0 at MMIO 0x21c0500 (irq = 18, base_baud = 18750000) is a 16550A
[ 1.243320] console [ttyS0] enabled
[ 1.243320] console [ttyS0] enabled
[ 1.250309] bootconsole [uart0] disabled
[ 1.250309] bootconsole [uart0] disabled
[ 1.258337] 21c0600.serial: ttyS1 at MMIO 0x21c0600 (irq = 18, base_baud = 18750000) is a 16550A
[ 1.267291] 21d0500.serial: ttyS2 at MMIO 0x21d0500 (irq = 19, base_baud = 18750000) is a 16550A
[ 1.276245] 21d0600.serial: ttyS3 at MMIO 0x21d0600 (irq = 19, base_baud = 18750000) is a 16550A
[ 1.288701] brd: module loaded
[ 1.293380] loop: module loaded
[ 1.296585] at24 0-0052: 65536 byte 24c512 EEPROM, writable, 1 bytes/write
[ 1.303471] at24 0-0053: 65536 byte 24c512 EEPROM, writable, 1 bytes/write
[ 1.310790] ahci-qoriq 3200000.sata: AHCI 0001.0301 32 slots 1 ports 6 Gbps 0x1 impl platform mode
[ 1.319760] ahci-qoriq 3200000.sata: flags: 64bit ncq sntf pm clo only pmp fbs pio slum part ccc
sds apst
[ 1.329738] scsi host0: ahci-qoriq
[ 1.333229] ata1: SATA max UDMA/133 mmio [mem 0x03200000-0x0320ffff] port 0x100 irq 30
[ 1.341947] nand: device found, Manufacturer ID: 0x2c, Chip ID: 0xac
[ 1.348311] nand: Micron MT29F4G08ABBEAH4
[ 1.352315] nand: 512 MiB, SLC, erase size: 256 KiB, page size: 4096, OOB size: 224
[ 1.360623] Bad block table found at page 131008, version 0x01
[ 1.367676] Bad block table found at page 130944, version 0x01
[ 1.374551] fsl,ifc-nand 7e800000.nand: IFC NAND device at 0x7e800000, bank 0
[ 1.381947] fsl-quadspi 1550000.quadspi: s25fs512s (65536 Kbytes)
[ 1.388390] fsl-quadspi 1550000.quadspi: s25fs512s (65536 Kbytes)
[ 1.395613] libphy: Fixed MDIO Bus: probed
[ 1.399790] tun: Universal TUN/TAP device driver, 1.6
[ 1.404846] tun: (C) 1999-2004 Max Krasnyansky
[ 1.411315] libphy: Freescale XGMAC MDIO Bus: probed
[ 1.417174] libphy: Freescale XGMAC MDIO Bus: probed
[ 1.423299] libphy: Freescale XGMAC MDIO Bus: probed
[ 1.428336] libphy: Freescale XGMAC MDIO Bus: probed
[ 1.433361] libphy: Freescale XGMAC MDIO Bus: probed
[ 1.438394] libphy: Freescale XGMAC MDIO Bus: probed
[ 1.443419] libphy: Freescale XGMAC MDIO Bus: probed
[ 1.448451] libphy: Freescale XGMAC MDIO Bus: probed
[ 1.453478] libphy: Freescale XGMAC MDIO Bus: probed
[ 1.458505] libphy: Freescale XGMAC MDIO Bus: probed
[ 1.473182] Freescale FM module, FMD API version 21.1.0
[ 1.480715] Freescale FM Ports module
[ 1.484382] fsl_mac: fsl_mac: FSL FMan MAC API based driver
[ 1.490043] fsl_mac 1ae4000.ethernet: FMan MEMAC
[ 1.494665] fsl_mac 1ae4000.ethernet: FMan MAC address: 00:00:00:00:00:03
[ 1.501503] fsl_mac 1ae6000.ethernet: FMan MEMAC
[ 1.506123] fsl_mac 1ae6000.ethernet: FMan MAC address: 00:00:00:00:00:04
[ 1.513007] fsl_mac 1ae8000.ethernet: FMan MEMAC
[ 1.517627] fsl_mac 1ae8000.ethernet: FMan MAC address: 00:00:00:00:00:05
[ 1.524512] fsl_mac 1aea000.ethernet: FMan MEMAC
[ 1.529127] fsl_mac 1aea000.ethernet: FMan MAC address: 00:00:00:00:00:06
[ 1.535973] fsl_mac 1af0000.ethernet: FMan MEMAC
[ 1.540588] fsl_mac 1af0000.ethernet: FMan MAC address: 00:00:00:00:00:07
[ 1.547547] fsl_mac 1af2000.ethernet: FMan MEMAC
[ 1.552163] fsl_mac 1af2000.ethernet: FMan MAC address: 00:00:00:00:00:08
[ 1.559026] fsl_dpa: FSL DPAA Ethernet driver
[ 1.563471] fsl_dpa fsl,dpaa:ethernet@0: of_find_device_by_node(/soc/fman@1a00000/ethernet@e0000) failed
[ 1.572963] fsl_dpa: probe of fsl,dpaa:ethernet@0 failed with error -22
[ 1.579616] fsl_dpa fsl,dpaa:ethernet@1: of_find_device_by_node(/soc/fman@1a00000/ethernet@e2000) failed
[ 1.589106] fsl_dpa: probe of fsl,dpaa:ethernet@1 failed with error -22
```

Deploy U-Boot, Linux Kernel, and Root Filesystem to a Reference Design Board (RDB)

Supported Boards

```
[ 1.600629] fsl_dpa: fsl_dpa: Probed interface eth0
[ 1.610449] fsl_dpa: fsl_dpa: Probed interface eth1
[ 1.620543] fsl_dpa: fsl_dpa: Probed interface eth2
[ 1.630908] fsl_dpa: fsl_dpa: Probed interface eth3
[ 1.641565] fsl_dpa: fsl_dpa: Probed interface eth4
[ 1.652419] fsl_dpa: fsl_dpa: Probed interface eth5
[ 1.657362] fsl_advanced: FSL DPAA Advanced drivers:
[ 1.662324] fsl_proxy: FSL DPAA Proxy initialization driver
[ 1.668030] fsl_dpa_shared: FSL DPAA Shared Ethernet driver
[ 1.673688] fsl_dpa_macless: FSL DPAA MACless Ethernet driver
[ 1.679513] fsl_oh: FSL FMan Offline Parsing port driver
[ 1.683593] ata1: SATA link down (SStatus 0 SControl 300)
[ 1.690334] e1000: Intel(R) PRO/1000 Network Driver - version 7.3.21-k8-NAPI
[ 1.697385] e1000: Copyright (c) 1999-2006 Intel Corporation.
[ 1.703161] e1000e: Intel(R) PRO/1000 Network Driver - 2.3.2-k
[ 1.708994] e1000e: Copyright(c) 1999 - 2014 Intel Corporation.
[ 1.714940] sky2: driver version 1.30
[ 1.718848] VFIO - User Level meta-driver version: 0.3
[ 1.724110] vfio_fsl_mc_driver_init: Driver registration fails as no fsl_mc_bus found
[ 2.932972] xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
[ 2.938470] xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 1
[ 2.946283] xhci-hcd xhci-hcd.0.auto: hcc params 0x0220f66d hci version 0x100 quirks 0x00010010
[ 2.955006] xhci-hcd xhci-hcd.0.auto: irq 27, io mem 0x02f00000
[ 2.961164] hub 1-0:1.0: USB hub found
[ 2.964927] hub 1-0:1.0: 1 port detected
[ 2.968952] xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
[ 2.974445] xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 2
[ 2.982121] usb usb2: We don't know the algorithms for LPM for this host, disabling LPM.
[ 2.990427] hub 2-0:1.0: USB hub found
[ 2.994184] hub 2-0:1.0: 1 port detected
[ 2.998220] xhci-hcd xhci-hcd.1.auto: xHCI Host Controller
[ 3.003713] xhci-hcd xhci-hcd.1.auto: new USB bus registered, assigned bus number 3
[ 3.011513] xhci-hcd xhci-hcd.1.auto: hcc params 0x0220f66d hci version 0x100 quirks 0x00010010
[ 3.020231] xhci-hcd xhci-hcd.1.auto: irq 28, io mem 0x03000000
[ 3.026366] hub 3-0:1.0: USB hub found
[ 3.030117] hub 3-0:1.0: 1 port detected
[ 3.034144] xhci-hcd xhci-hcd.1.auto: xHCI Host Controller
[ 3.039629] xhci-hcd xhci-hcd.1.auto: new USB bus registered, assigned bus number 4
[ 3.047311] usb usb4: We don't know the algorithms for LPM for this host, disabling LPM.
[ 3.055613] hub 4-0:1.0: USB hub found
[ 3.059364] hub 4-0:1.0: 1 port detected
[ 3.063400] xhci-hcd xhci-hcd.2.auto: xHCI Host Controller
[ 3.068920] xhci-hcd xhci-hcd.2.auto: new USB bus registered, assigned bus number 5
[ 3.076727] xhci-hcd xhci-hcd.2.auto: hcc params 0x0220f66d hci version 0x100 quirks 0x00010010
[ 3.085446] xhci-hcd xhci-hcd.2.auto: irq 29, io mem 0x03100000
[ 3.091579] hub 5-0:1.0: USB hub found
[ 3.095366] hub 5-0:1.0: 1 port detected
[ 3.099384] xhci-hcd xhci-hcd.2.auto: xHCI Host Controller
[ 3.104876] xhci-hcd xhci-hcd.2.auto: new USB bus registered, assigned bus number 6
[ 3.112552] usb usb6: We don't know the algorithms for LPM for this host, disabling LPM.
[ 3.120850] hub 6-0:1.0: USB hub found
[ 3.124629] hub 6-0:1.0: 1 port detected
[ 3.128663] ehci_hcd: USB 2.0 'Enhanced' Host Controller (EHCI) Driver
[ 3.135220] ehci-pci: EHCI PCI platform driver
[ 3.139684] ehci-platform: EHCI generic platform driver
[ 3.145029] ohci_hcd: USB 1.1 'Open' Host Controller (OHCI) Driver
[ 3.151211] ohci-pci: OHCI PCI platform driver
[ 3.155707] ohci-platform: OHCI generic platform driver
[ 3.161061] usbcore: registered new interface driver usb-storage
[ 3.167189] mousedev: PS/2 mouse device common for all mice
```

```
[ 3.172941] rtc-pcf2127 1-0051: chip found, driver version 0.0.1
[ 3.181682] rtc-pcf2127 1-0051: rtc core: registered rtc-pcf2127 as rtc0
[ 3.188479] i2c /dev entries driver
[ 3.194275] ina2xx 0-0040: power monitor ina220 (Rshunt = 1000 uOhm)
[ 3.200675] 0-004c supply vcc not found, using dummy regulator
[ 3.209254] imx2-wdt 2ad0000.wdog: timeout 60 sec (nowayout=0)
[ 3.215370] sdhci: Secure Digital Host Controller Interface driver
[ 3.221548] sdhci: Copyright(c) Pierre Ossman
[ 3.225958] sdhci-pltfm: SDHCI platform and OF driver helper
[ 3.231727] sdhci-esdhc 1560000.esdhc: No vmmc regulator found
[ 3.237565] sdhci-esdhc 1560000.esdhc: No vqmmc regulator found
[ 3.283583] mmc0: SDHCI controller on 1560000.esdhc [1560000.esdhc] using ADMA 64-bit
[ 3.292162] platform caam_qi.0: Linux CAAM Queue I/F driver initialised
[ 3.298788] caam 1700000.crypto: Instantiated RNG4 SH1
[ 3.303933] caam 1700000.crypto: device ID = 0x0a11030100000000 (Era 8)
[ 3.310546] caam 1700000.crypto: job rings = 4, qi = 1
[ 3.317314] caam algorithms registered in /proc/crypto
[ 3.322993] platform caam_qi.0: fsl,sec-v5.4 algorithms registered in /proc/crypto
[ 3.331427] caam_jr 1710000.jr: registering rng-caam
[ 3.336493] caam 1700000.crypto: fsl,sec-v5.4 algorithms registered in /proc/crypto
[ 3.344195] MC object device driver fsl_dpaa2_caam registered
[ 3.350438] usbcore: registered new interface driver usbhid
[ 3.356101] usbhid: USB HID core driver
[ 3.359999] fsl-mc bus not found, restool driver registration failed
[ 3.366378] MC object device driver fsl_dprio_drv registered
[ 3.372405] Freescale USDPAA process driver
[ 3.376588] fsl-usdpaa: no region found
[ 3.380419] Freescale USDPAA process IRQ driver
[ 3.385008] MC object device driver fsl_dpaa2_eth registered
[ 3.390677] MC object device driver dpaa2_mac registered
[ 3.396003] MC object device driver dpaa2_ethsw registered
[ 3.401496] MC object device driver dpaa2_evb registered
[ 3.406863] MC object device driver fsl_dce_api registered
[ 3.412357] MC object device driver dpaa2_rtc registered
[ 3.417768] Initializing XFRM netlink socket
[ 3.422073] NET: Registered protocol family 10
[ 3.426910] sit: IPv6 over IPv4 tunneling driver
[ 3.431749] NET: Registered protocol family 17
[ 3.436204] NET: Registered protocol family 15
[ 3.440667] 8021q: 802.1Q VLAN Support v1.8
[ 3.444877] 9pnet: Installing 9P2000 support
[ 3.449173] Key type dns_resolver registered
[ 3.453670] registered taskstats version 1
[ 3.457893] fsl_generic: FSL DPAA Generic Ethernet driver
[ 3.466054] rtc-pcf2127 1-0051: setting system clock to 2016-08-23 22:21:29 UTC (1471990889)
[ 3.474584] fdt: not creating '/sys/firmware/fdt': CRC check failed
[ 3.481260] RAMDISK: gzip image found at block 0
[ 4.242385] VFS: Mounted root (ext2 filesystem) readonly on device 1:0.
[ 4.249050] devtmpfs: mounted
[ 4.252209] Freeing unused kernel memory: 476K (ffff800000b1a000 - ffff800000b91000)
[ 4.259991] Freeing alternatives memory: 48K (ffff800000b91000 - ffff800000b9d000)
INIT: version 2.88 booting
Starting udev
[ 4.365202] udevd[1890]: starting version 182
[ 4.393090] fsl_dpa fsl_dpaa:ethernet@2 fml-mac3: renamed from eth0
[ 4.493818] fsl_dpa fsl_dpaa:ethernet@8 fml-mac9: renamed from eth4
[ 4.493848] udevd[1893]: renamed network interface eth0 to fml-mac3
[ 4.563841] fsl_dpa fsl_dpaa:ethernet@5 fml-mac6: renamed from eth3
[ 4.563870] udevd[1898]: renamed network interface eth4 to fml-mac9
[ 4.663783] fsl_dpa fsl_dpaa:ethernet@3 fml-mac4: renamed from eth1
```

Deploy U-Boot, Linux Kernel, and Root Filesystem to a Reference Design Board (RDB)

Supported Boards

```
[ 4.663809] udevd[1896]: renamed network interface eth3 to fml-mac6
[ 4.723709] fsl_dpa fsl,dpaa:ethernet@4 fml-mac5: renamed from eth2
[ 4.730024] udevd[1894]: renamed network interface eth1 to fml-mac4
[ 4.773732] udevd[1895]: renamed network interface eth2 to fml-mac5
[ 4.820204] random: dd urandom read with 3 bits of entropy available
Populating dev cache
Running postinst /etc/rpm-postinsts/100-sysvinit-inittab...
INIT: Entering runlevel: 5un-postinsts exists during rc.d purge
Configuring network interfaces... eth0: ERROR while getting interface flags: No such device
Starting OpenBSD Secure Shell server: sshd
    generating ssh RSA key...
    generating ssh ECDSA key...
    generating ssh DSA key...
    generating ssh ED25519 key...
done.
Starting network benchmark server: netserver.
Starting system log daemon...0
Starting kernel log daemon...0

QorIQ SDK (FSL Reference Distro) 1.9 ls1043ardb /dev/ttyS0

ls1043ardb login: root

root@ls1043ardb:~# ifconfig
lo          Link encap:Local Loopback
            inet addr:127.0.0.1  Mask:255.0.0.0
            inet6 addr: ::1/128 Scope:Host
            UP LOOPBACK RUNNING  MTU:65536  Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:0
            RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@ls1043ardb:~# ifconfig -a
eth5       Link encap:Ethernet  HWaddr 00:00:00:00:00:08
            BROADCAST MULTICAST  MTU:1500  Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
            Memory:1af2000-1af2fff

fml-mac3   Link encap:Ethernet  HWaddr 00:00:00:00:00:03
            BROADCAST MULTICAST  MTU:1500  Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
            Memory:1ae4000-1ae4fff

fml-mac4   Link encap:Ethernet  HWaddr 00:00:00:00:00:04
            BROADCAST MULTICAST  MTU:1500  Metric:1
            RX packets:0 errors:0 dropped:0 overruns:0 frame:0
            TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
            Memory:1ae6000-1ae6fff

fml-mac5   Link encap:Ethernet  HWaddr 00:00:00:00:00:05
            BROADCAST MULTICAST  MTU:1500  Metric:1
```

```
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
Memory:1ae8000-1ae8fff

fm1-mac6 Link encap:Ethernet HWaddr 00:00:00:00:00:06
BROADCAST MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
Memory:1aea000-1aeafff

fm1-mac9 Link encap:Ethernet HWaddr 00:00:00:00:00:07
BROADCAST MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
Memory:1af0000-1af0fff

lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

sit0 Link encap:UNSPEC HWaddr 00-00-00-00-3A-30-30-30-00-00-00-00-00-00-00-00
NOARP MTU:1480 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

root@ls1043ardb:~#
root@ls1043ardb:~# ifconfig fm1-mac3 10.193.20.129
root@ls1043ardb:~# ping 10.192.208.233
PING 10.192.208.233 (10.192.208.233) 56(84) bytes of data.
64 bytes from 10.192.208.233: icmp_seq=1 ttl=63 time=0.303 ms
64 bytes from 10.192.208.233: icmp_seq=2 ttl=63 time=0.314 ms
64 bytes from 10.192.208.233: icmp_seq=3 ttl=63 time=0.281 ms
64 bytes from 10.192.208.233: icmp_seq=4 ttl=63 time=0.319 ms
64 bytes from 10.192.208.233: icmp_seq=5 ttl=63 time=0.318 ms
64 bytes from 10.192.208.233: icmp_seq=6 ttl=63 time=0.289 ms
64 bytes from 10.192.208.233: icmp_seq=7 ttl=63 time=0.267 ms
64 bytes from 10.192.208.233: icmp_seq=8 ttl=63 time=0.259 ms
64 bytes from 10.192.208.233: icmp_seq=9 ttl=63 time=0.318 ms
^C
--- 10.192.208.233 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 7998ms
rtt min/avg/max/mdev = 0.259/0.296/0.319/0.027 ms
root@ls1043ardb:~#
```

Chapter 4

System Recovery

4.1 Environment Setup

4.1.1 Environment Setup (Common)

The section describes the related setup for system recovery

1. Required Materials

- Target board
- The related recovery image files

2. Host PC setup

The host PC should have a serial-terminal program capable of running at 115,200bps, 8-N-1, for communicating with U-Boot running on the target board.

3. Target board setup

- a. Power off the target board system if the power is already on.
- b. If U-Boot runs on this board, and U-Boot commands will be used to reflash the U-Boot images, connect the target board to the network via the eTSEC port on the board.
- c. Connect the target board to the host machine via the serial port with an RS-232 cable and the joined NXP adapter cable, if needed.
- d. Set up the serial terminal in the host machine with 115,200bps, 8-N-1, no flow control.
- e. Verify all the switches and jumpers are set up correctly to default values described in <Hardware configurations> as described in the Switch Settings section of the board's Software Deployment Guide
- f. Connect the JTAG cable for your CodeWarrior TAP or Gigabit TAP to the board if you will be using the CodeWarrior Flash Programmer to recover the board image.
- g. Power on the board.

4.2 Image Recovery

4.2.1 Recover system with already working U-Boot

Target Board Setup

1. Power off the target board system if the power is already on.
2. Connect the target board to the network via the eTSEC port on the board.
3. Connect the target board to the host machine via the serial port with an RS-232 cable and the joined NXP adaptor cable, if needed.
4. Set up the serial terminal in the host machine with 115,200bps, 8-N-1, no flow control.
5. Verify all the switches and jumpers are setup correctly to default value described in the board's "Switch Settings" section in the board's Software Deployment Guide.

6. Power on the board.

Refer to the “Programming a New U-Boot...” section in the Software Deployment Guide for the target board to be recovered.

4.2.2 Recover system using CodeWarrior Flash Programmer

Environment Setup

Required Materials

- Target board.
- CodeWarrior for Power Architecture v10.x (for Windows or Linux)
- CodeWarrior TAP or Gigabit TAP run control device.
- The related recovery image files.

Host PC setup

The host PC is assumed to be running Windows 7, or one of the supported distributions of Linux (refer to the CodeWarrior PA10 Release Notes for the list of supported Linux distributions).

This machine should have latest CodeWarrior PA10 installed and working correctly. If the run control device is a CodeWarrior TAP used over USB, then the USB drivers should be installed automatically when the device is plugged in. If the run control device is a CodeWarrior TAP used over Ethernet, or a Gigabit TAP, then both the host PC and TAP should be connected to the network, and communications between them should be verified.

Target board setup

1. Power off the Target board system if the power is already on.
2. Connect the Target board to the host machine via the serial port with an RS-232 cable and the joined NXP adaptor cable, if needed.
3. Set up the serial terminal in the host machine with 115,200bps, 8-N-1, no flow control.
4. Verify all the switches and jumpers are set up correctly to default values described in the "Switch Settings" section in the board's Software Deployment Guide.
5. Connect the TAP's JTAG cable to the board.
6. Power on the board.

System Recovery

1. Start the CodeWarrior PA10 or ARMv7 IDE.
2. For LS102x targets, see Chapter 3 of *Getting Started for ARMv7 Processors.pdf* in the CodeWarrior ARMv7 installation for steps on creating a BareBoard Core0 project for the LS102x processor on this target board. For other QorIQ targets, see *Quick Start for PA 10 Processors.pdf* in the CodeWarrior PA10 installation for steps on creating a BareBoard AMP Core0 project for the QorIQ processor on this target board. In the "Debug Target Settings Page", of the procedure for creating a new project, uncheck the 'Download' option, and enable the 'Download SRAM' option, if available.
3. Select your CodeWarrior TAP or Gigabit TAP as your debug connection type. For CodeWarrior TAP, select “USB” or “Ethernet” as the connection medium.
4. Build the project.
5. Bring up the Target Tasks view: go to Window>Show View>Other>Debug>Target Task.
6. Import the Flash Profile:
 - a. In the Target Tasks view, click on the Import button. A file-browser window will appear, showing the "Flash_Programmer" folder.

- b. Open the "Flash_Programmer" folder, then the folder associated with the processor family on this target board.
 - c. Select the configuration file for the particular target and flash device to be programmed on this target board, and click OK to import it. This file will appear in the Target Tasks view.
7. In the board's Software Deployment Guide, locate the "Flash Bank Usage" section for the target board to be recovered.
- a. Identify the NOR/NAND/SPI flash memory map that applies to the flash to be programmed. For the following steps, if the target flash supports multiple banks, choose the starting addresses for 'Bank0' or 'current bank', as appropriate.
 - b. Identify the starting address for the u-boot image.
 - c. Identify the starting address for the RCW image (if applicable).
 - d. Identify the starting address for the ucode/microcode (if applicable).
 - e. Identify the starting address for the dtb image.
 - f. Identify the starting address for the RamDisk image.
 - g. Identify the starting address for the Linux Kernel image. For example:

Table 7. T4240QDS NOR flash

Binaries	Starting Address
U-Boot	0xEFF40000
RCW	0xE8000000
ucode	0xEFF00000
dtb	0xE8800000
RamDisk (rootfs)	0xE9300000
Linux Kernel (ulmage)	0xE8020000

8. Configure Flash Programmer.
- a. Double-click on the file name that was imported with the flash profile, to bring up the Flash Programmer Task view.
 - b. Click on 'Add Action' > 'Program/Verify'.
 - c. Set 'File Type' to "Binary".
 - d. Click on 'File System' and navigate to the folder containing the u-boot binary image.
 - e. Enable "Erase sectors before program".
 - f. Enable "Apply address offset", and enter the starting address where this binary recovery image will be flashed (see the tables in the previous step for examples).
 - g. (OPTIONAL) Enable 'Verify after program' to verify that the flash programming was successful.
 - h. Repeat steps (starting with Click 'Add Action') above for each binary image file to be programmed into flash.
9. Execute Flash Programming.
- a. In the Target Tasks view, right-click on the imported filename and select the green Execute button to launch the programmer.
 - b. If Execute is not green, the debugger is not running. The debugger must be running for this flash programmer to work.
 - c. When finished flashing, terminate the debugger.
10. This is the end of the process. Now the boot loader, kernel and root file system are programmed to flash.

11. Reset or power-cycle the board and verify that u-boot appears in the board's serial terminal.

Chapter 5 Linux Kernel Drivers

5.1 DMA Controller

5.1.1 Enhanced Direct Memory Access Driver (ARM)

5.1.1.1 eDMA User Manual

Description

The SoC integrates NXP's Enhanced Direct Memory Access module. Slave device such as I2C or SAI can deploy the DMA functionality to accelerate the transfer and release the CPU from heavy load.

Kernel Configure Options

Tree View

Below are the configure options need to be set/unset while doing "make menuconfig" for kernel

Kernel Configure Tree View Options	Description
<pre>Device Drivers ---> [*] DMA Engine support ---> ---> <*> Freescale eDMA engine support</pre>	DMA engine subsystem driver and eDMA driver support

Identifier

Below are the configure identifiers which are used in kernel source code and default configuration files.

Option	Values	Default Value	Description
CONFIG_FSL_EDMA	y/m/n	n	eDMA Driver

Device Tree Binding

Device Tree Node

Below is an example device tree node required by this feature. Note that it may has differences among platforms.

```
edma0: edma@2c00000 {
    #dma-cells = <2>;
    compatible = "fsl,vf610-edma";
    reg = <0x0 0x2c00000 0x0 0x10000>,
        <0x0 0x2c10000 0x0 0x10000>,
        <0x0 0x2c20000 0x0 0x10000>;
    interrupts = <GIC_SPI 135 IRQ_TYPE_LEVEL_HIGH>,
        <GIC_SPI 135 IRQ_TYPE_LEVEL_HIGH>;
}
```

```

interrupt-names = "edma-tx", "edma-err";
dma-channels = <32>;
big-endian;
clock-names = "dmamux0", "dmamux1";
clocks = <&platform_clk 1>,
        <&platform_clk 1>;
};

```

Device Tree Node Binding for Slave Device

Below is the device tree node binding for a slave device which deploy the eDMA functionality.

```

i2c0: i2c@2180000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl,vf610-i2c";
    reg = <0x0 0x2180000 0x0 0x10000>;
    interrupts = <GIC_SPI 88 IRQ_TYPE_LEVEL_HIGH>;
    clock-names = "i2c";
    clocks = <&platform_clk 1>;
    dmas = <&edma0 1 39>,
          <&edma0 1 38>;
    dma-names = "tx", "rx";
    status = "disabled";
};

```

Source Files

The following source files are related the this feature in Linux kernel.

Table 8. Source Files

Source File	Description
drivers/dma/fsl-edma.c	The eDMA driver file

Verification in Linux

1. Use the slave device which deploy the eDMA functionality to verify the eDMA driver, below is a verification with the I2C salve.

```

root@ls1021aqds:~# i2cdetect 0
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-0.
I will probe address range 0x03-0x77.
Continue? [Y/n]
   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
10:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
20:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
30:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
40:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
50:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
60:  --  --  --  --  --  --  --  --  --  69  --  --  --  --  --  --
70:  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --  --
root@ls1021aqds:~# i2cdump 0 0x69 i
   0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef

```

```

00: 05 07 ff ff 5d 55 10 55 11 05 1e 00 e8 03 b5 ff    ???]U?U???..???.
10: ff e8 03 95 00 00 00 00 aa fe 9a 00 00 00 00 78    .???....???...x
20: 05 12 04 ff 00 7f 40 14 1d 60 3c 83 05 00 40 00    ???..?@??`<??.@.
30: fe 80 c6 29 00 00 00 7a 00 ff ff ff ff ff ff ff    ???)...z.....
40: 05 07 ff ff 5d 55 10 55 11 05 1e 00 e8 03 b5 ff    ???]U?U???..???.
50: ff e8 03 95 00 00 00 00 aa fe 9a 00 00 00 00 78    .???....???...x
60: 05 12 04 ff 00 7f 40 14 1d 60 3c 83 05 00 40 00    ???..?@??`<??.@.
70: fe 80 c6 29 00 00 00 7a 00 ff ff ff ff ff ff ff    ???)...z.....
80: 07 ff ff 5d 55 10 55 11 05 1e 00 e8 03 b5 ff ff    ?..]U?U???..???.
90: e8 03 95 00 00 00 00 aa fe 9a 00 00 00 00 78 00    ???....???...x.
a0: 12 04 ff 00 7f 40 14 1d 60 3c 83 05 00 40 00 fe    ???..?@??`<??.@.?
b0: 80 c6 29 00 00 00 7a 00 ff ff ff ff ff ff ff ff    ??)...z.....
c0: 07 ff ff 5d 55 10 55 11 05 1e 00 e8 03 b5 ff ff    ?..]U?U???..???.
d0: e8 03 95 00 00 00 00 aa fe 9a 00 00 00 00 78 00    ???....???...x.
e0: 12 04 ff 00 7f 40 14 1d 60 3c 83 05 00 40 00 fe    ???..?@??`<??.@.?
f0: 80 c6 29 00 00 00 7a 00 ff ff ff ff ff ff ff ff    ??)...z.....
root@ls1021aqds:~# cat /proc/interrupts
          CPU0           CPU1
 29:             0             0          GIC 29  arch_timer
 30:          5563          5567          GIC 30  arch_timer
112:             260             0          GIC 112 fsl-lpuart
120:             32             0          GIC 120 2180000.i2c
121:             0             0          GIC 121 2190000.i2c
167:             8             0          GIC 167  eDMA
IPI0:             0             1 CPU wakeup interrupts
IPI1:             0             0 Timer broadcast interrupts
IPI2:          1388          1653 Rescheduling interrupts
IPI3:             0             0 Function call interrupts
IPI4:             2             4 Single function call interrupts
IPI5:             0             0 CPU stop interrupts
Err:             0
root@ls1021aqds:~#

```

5.2 DPAA 1.x Devices

5.2.1 DPAA Primer for Software Architecture

5.2.1.1 DPAA Primer

Multicore processing, or multiple execution thread processing, introduces unique considerations to the architecture of networking systems, including processor load balancing/utilization, flow order maintenance, and efficient cache utilization. Herein is a review of the key features, functions, and properties enabled by the QorIQ DPAA (Data Path Acceleration Architecture) hardware and demonstrates how to best architect software to leverage the DPAA hardware.

By exploring how the DPAA is configured and leveraged in a particular application, the user can determine which elements and features to use. This streamlines the software development stage of implementation by allowing programmers to focus their technical understanding on the elements and features that are implemented in the system under development, thereby reducing the overall DPAA learning curve required to implement the application.

Our target audience is familiar with the material in **QorIQ Data Path Acceleration Architecture (DPAA) Reference Manual**.

Benefits of the DPAA

The primary intent of the DPAA is to provide intelligence within the IO portion of the System On Chip (SOC) to route and manage the processing work associated with traffic flows to simplify ordering and load balance concerns associated with

multi core processing. The DPAA hardware inspects ingress traffic and extracts user defined flows from the port traffic. It then steers specific flows (or related traffic) to a specific core or set of cores.

Architecting a networking application with a multicore processor presents challenges (such as workload balance and maintaining flow order), which are not present in a single core design. Without hardware assistance, the software must implement techniques that are inefficient and cumbersome, reducing the performance benefit of multiple cores. To address workload balance and flow order challenges, the DPAA determines and separates ingress flows then manages the temporary, permanent, or semi-permanent flow-to-core affinity. The DPAA also provides a work priority scheme, which ensures ingress critical flows are addressed properly and frees software from the need to implement a queuing mechanism on egress. As the hardware determines which core will process which packet, performance is boosted by direct cache warming/stashing as well as by providing biasing for core-to-flow affinity, which ensures that flow-specific data structures can remain in the core's cache.

By understanding how the DPAA is leveraged in a particular design, the system architect can map out the application to meet the performance goals and to utilize the DPAA features to leverage any legacy application software that may exist. Once this application map is defined, the architect can focus on more specific details of the implementation.

5.2.1.1.1 General Architectural Considerations

As the need for processing capability has grown, the only practical way to increase the performance on a single silicon part is to increase the number of general purpose processing cores (CPUs). However, many legacy designs run on a single processor; introducing multiple processors into the system creates special considerations, especially for a networking application.

5.2.1.1.2 Multicore Design

Multicore processing, or multiple execution thread processing, introduces unique considerations. Most networking applications are split between data and control plane tasks. In general, control plane tasks manage the system within the broad network of equipment. While the control plane may process control packets between systems, the control plane process is not involved in the bulk processing of the data traffic. This task is left to the data plane processing task or program.

The general flow of the data plane program is to receive data traffic (in general, packets or frames), process or transform the data in some way and then send the packets to the next hop or device in the network. In many cases, the processing of the traffic depends on the type of traffic. In addition, the traffic usually exists in terms of a flow, a stream of traffic where the packets are related. A simple example could be a connection between two clients that, at the packet level, is defined by the source and destination IP address. Typically, multiple flows are interleaved on a single interface port; the number of flows per port depends on the interface bandwidth as well as on the bandwidth and type of flows involved.

5.2.1.1.3 Parse/classification Software Offload

The DPAA provides intelligence within the IO subsection of the SoC (system-on-chip) to split traffic into user-defined queues. One benefit is that the intelligence used to divide the traffic can be leveraged at a system level.

In addition to sorting and separating the traffic, the DPAA can append useful processing information into the stream; offloading the need for the software to perform these functions (see the following two figures).

Note that the DPAA is not intended to replace significant packet processing or to perform extensive classification tasks. However, some applications may benefit from the streamlining that results from the parse/classify/distribute function within the DPAA. The ability to identify and separate flow traffic is fundamental to how the DPAA solves other multicore application issues.

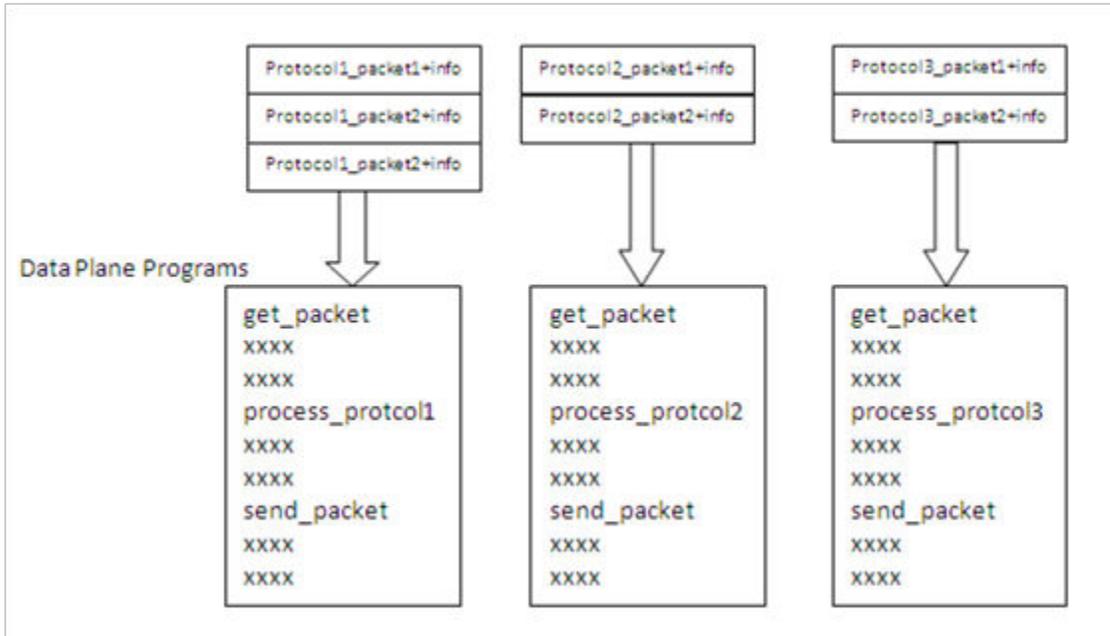


Figure 1. Hardware-sorted Protocol Flow

5.2.1.1.4 Flow Order Considerations

In most networking applications, individual traffic flows through the system require that the egress packets remain in the order in which they are received. In a single processor core system, this requirement is easy to implement. As long as the data plane software follows a run-to-completion model on a per-packet basis, the egress order will match the ingress packet order. However, if multiple processors are implemented in a true symmetrical multicore processing (SMP) model in the system, the egress packet flow may be out of order with respect to the ingress flow. This may be caused when multiple cores simultaneously process packets from the same flow.

Even if the code flow is identical, factors such as cache hits/misses, DRAM page hits/misses, interrupts, control plane and OS tasks can cause some variability in the processing path, allowing egress packets to “pass” within the same flow, as shown in this figure

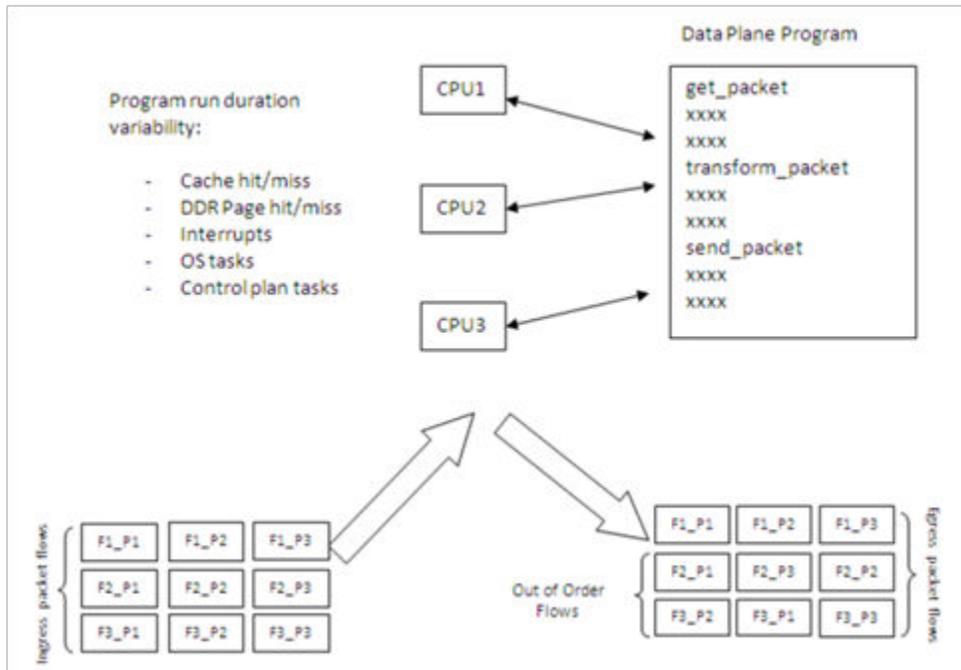


Figure 2. Multicore Flow Reordering

For some applications, it is acceptable to reorder the flows from ingress to egress. However, most applications require that order is maintained. When no hardware is available to assist with this ordering, the software must maintain flow order. Typically, this requires additional code to determine the sequence of the packet currently being processed, as well as a reference to a data structure that maintains order information for each flow in the system. Because multiple processors need to access and update this state information, access to this structure must be carefully controlled, typically by using a lock mechanism that can be expensive with regard to program cycles and processing flow (see the next figure). One of goals of the DPAA architecture is to provide the system designer with hardware to assist with packet ordering issues.

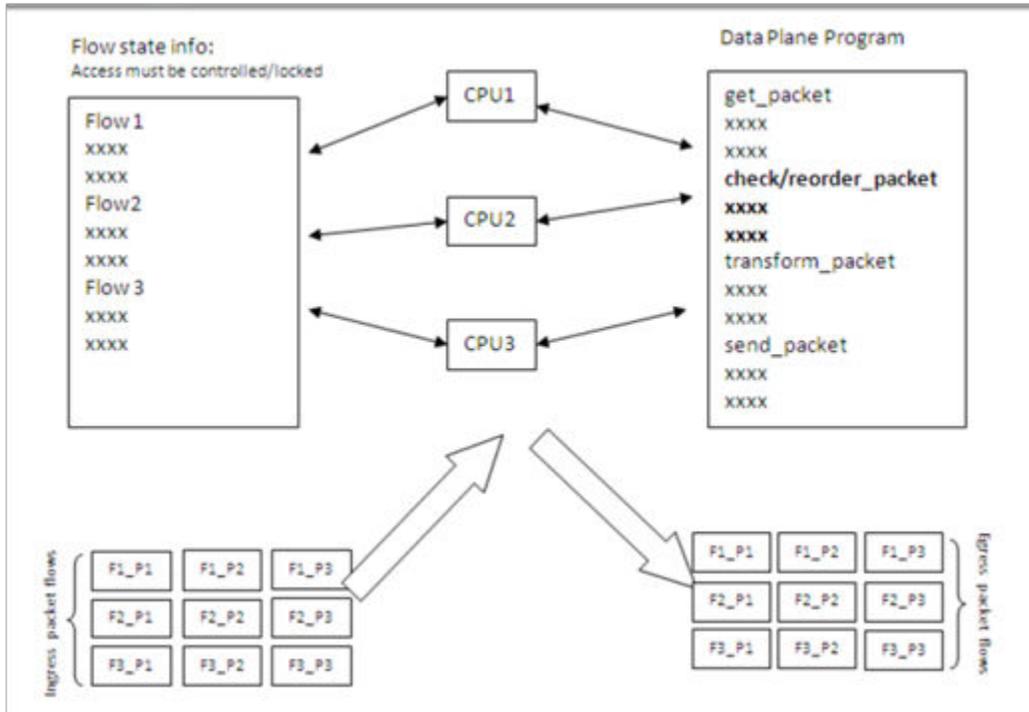


Figure 3. Implementing Order in Software

5.2.1.15 Managing Flow-to-Core Affinity

Multicore processing, or multiple execution thread processing, introduces unique considerations to the architecture of networking systems, including processor load balancing/utilization, flow order maintenance, and efficient cache utilization. Herein is a review of the key features, functions, and properties enabled by the QorIQ DPAA (Data Path Acceleration Architecture) hardware and demonstrates how to best architect software to leverage the DPAA hardware.

In a multicore networking system, if the hardware configuration always allows a specific core to process a specific flow then the binding of the flow to the core is described as providing flow affinity. If a specific flow is always processed by a specific processor core then for that flow the system acts like a single core system. In this case, flow order is preserved because there is a single thread of operation processing the flow; with a run-to-completion model, there is no opportunity for egress packets to be reordered with respect to ingress packets.

Another benefit of a specific flow being affined to a core is that the cache local to that core (L1 and possibly L2, depending on the specific core type) is less likely to miss when processing the packets because the core's data cache will not fetch flow state information for flows to which it is not affined. Also, because multiple processing entities have no need to access the same individual flow state information, the system need not lock the access to the individual flow state data. The DPAA offers several options to define and manage flow-to-core affinity.

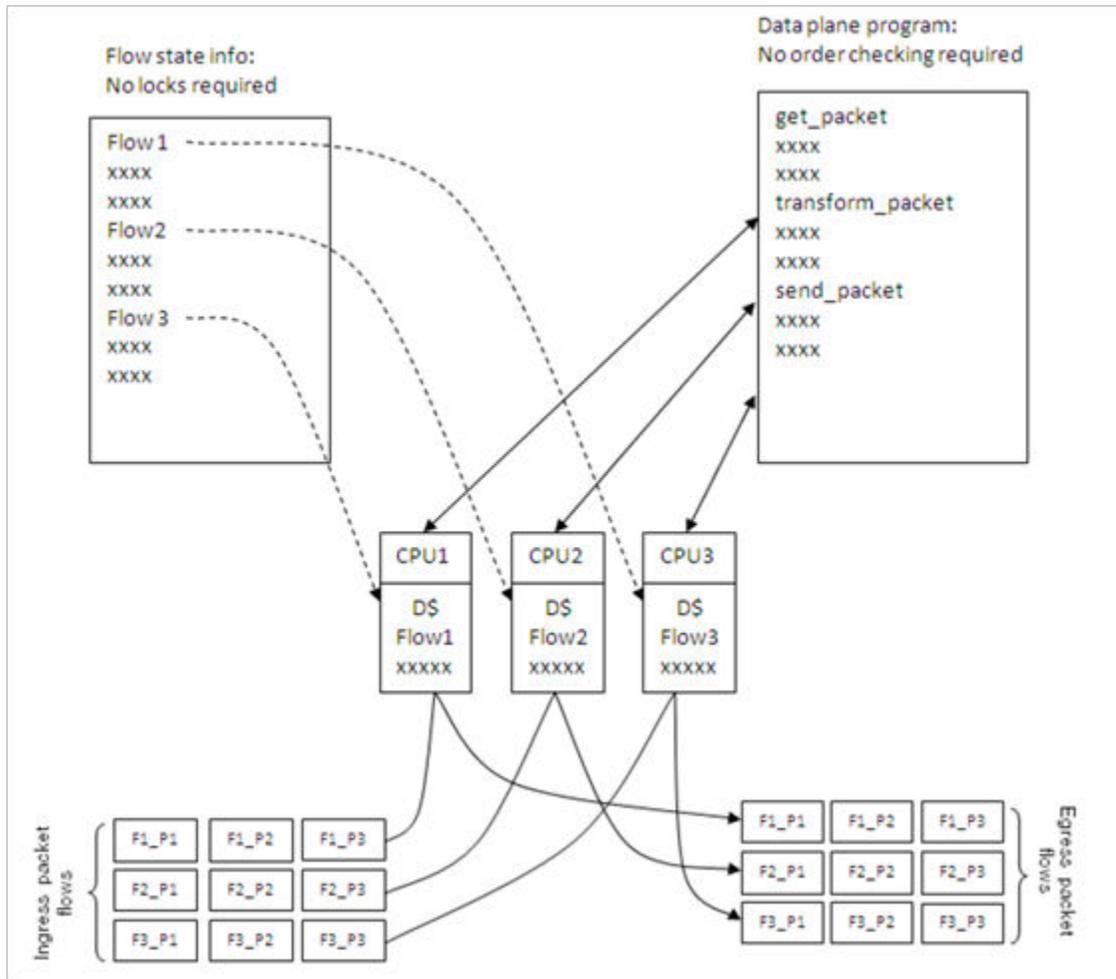


Figure 4. Managing Flow-to-Core Affinity

Many networking applications require intensive, repetitive algorithms to be performed on large portions of the data stream(s). While software in the processor cores could perform these algorithms, specific hardware offload engines often better address specific algorithms. Cryptographic and pattern matching accelerators are examples of this in the QorIQ family. These accelerators act as standalone hardware elements that are fed blocks or streams of data, perform the required processing, and then provide the output in a separate (or perhaps overwritten) data block within the system. The performance boost is significant for tasks that can be done by these hardware accelerators as compared to a software implementation.

In DPAA-equipped SoCs, these offload engines exist as peers to the cores and IO elements, and they use the same queuing mechanism to obtain and transfer data. The details of the specific processing performed by these offload engines is beyond the scope of this document; however, it is important to determine which of these engines will be leveraged in the specific application. The DPAA can then be properly defined to implement the most efficient configuration/definition of the DPAA elements.

5.2.1.2 DPAA Goals

A brief overview of the DPAA elements in order to contextualize the application mapping activities. For more details on the DPAA architecture, see the **QorIQ Data Path Acceleration Architecture (DPAA) Reference Manual**

The primary goals of the DPAA are as follows:

- To provide intelligence within the IO portion of the SoC.
- To route and manage the processing work associated with traffic flows.

- To simplify the ordering and load balance concerns associated with multicore processing.

The DPAA achieves these goals by inspecting and separating ingress traffic into Frame Queues (FQs). In general, the intent is to define a flow or set of flows as the traffic in a particular FQ. The FQs are associated to a specific core or set of cores via a channel. Within the channel definition, the FQs can be prioritized among each other using the Work Queue (WQ) mechanism. The egress flow is similar to the ingress flow. The processors place traffic on a specific FQ, which is associated to a particular physical port via a channel. The same priority scheme using WQs exists on egress, allowing higher priority traffic to pass lower priority traffic on egress without software intervention.

5.2.1.3 FMan Overview

The FMan inspects traffic, splits it into FQs on ingress, and sends traffic from the FQs to the interface on egress.

On ingress traffic, the FMan is configured to determine the traffic split required by the PCD (Parse, Classify, Distribute) function. This allows the user to decide how he wants to define his traffic: typically, by flows or classes of traffic. The PCD can be configured to route all traffic on one port to a single queue or with a higher level of complexity where large numbers of queues are defined and managed. The PCD can identify traffic based on the specific content of the incoming packets (usually within the header) or packet reception rates (policing).

The parse function is used to identify which fields within the data frame determine the traffic split. The fields used may be defined by industry standards, or the user may employ a programmable soft parse feature to accommodate proprietary field (typically header) definition(s). The results of the parse function may be used directly to determine the frame queue; or, the contents of the fields selected by the parse function may be used as a key to select the frame queue. The parse function employs a programmed mask to allow the use of selected fields.

The resultant key from the parse function may be used to assign traffic to a specific queue based on a specific exact match definition of fields in the header. Alternatively, a range of queues can be defined either by using the resultant key directly (if there are a small number of bits in the key) or by performing a hash of the key to use a large number of bits in the flow identifier and create a manageable number of queues.

The FMan also provides a policer function, which is rate-based and allows the user to mark or drop a specific frame that exceeds a traffic threshold. The policing is based on a two-rate, three-color marking algorithm (RFC2698). The sustained and peak rates as well as the burst sizes are user-configurable.

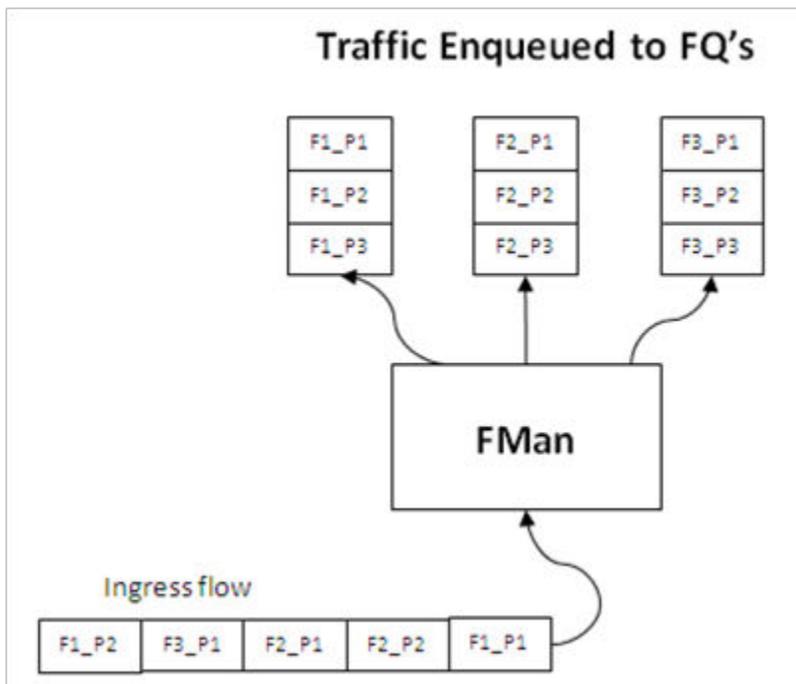


Figure 5. Ingress FMan Flow

The figure above shows the FMan splitting ingress traffic on an external port into a number of queues. However, the FMan works in a similar way on egress: it receives traffic from FQs then transmits the traffic on the designated external port. Alternatively, the FMan can be used to process flows internally via the offline port mechanism: traffic is dequeued (from some other element in the system), processed, then enqueued onto a frame queue processing further within the system.

On ingress traffic, the FMan generates an internal context (IC) data block, which it uses as it performs the PCD function. Optionally, some or all of this information may be added into the frames as they are passed along for further processing. For egress or offline processing, the IC data can be passed with each frame to be processed. This data is mostly the result of the PCD actions and includes the results of the parser, which may be useful for the application software. See the QorIQ DPAA Reference Manual for details on the contents of the IC data block.

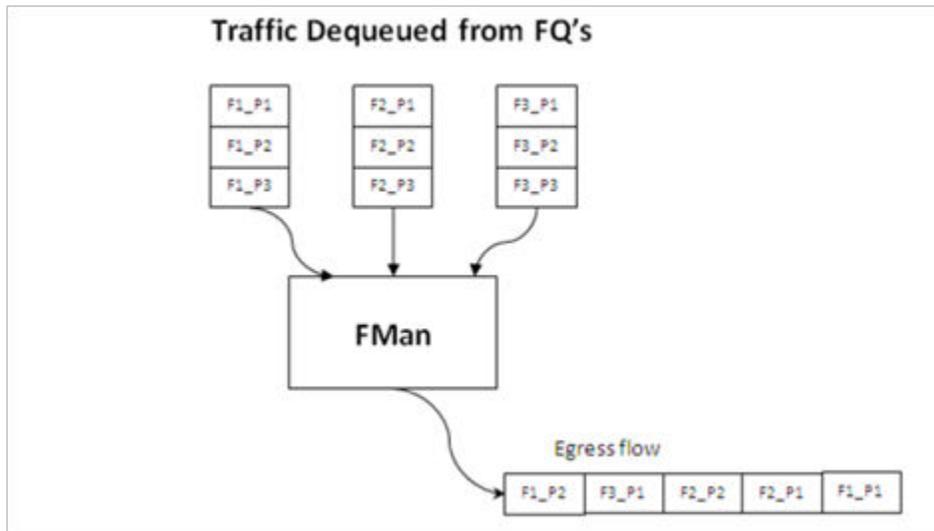


Figure 6. FMan Egress Flow

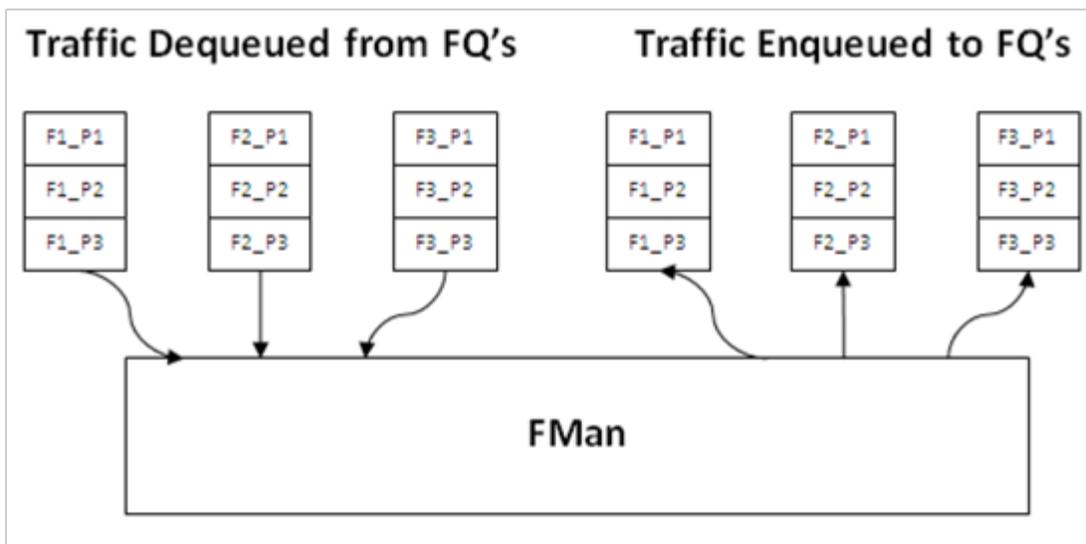


Figure 7. FMan Offline Flow

5.2.1.4 QMan Overview

The QMan links the FQs to producers and consumers (of data traffic) within the SoC. The producers/consumers are either FMan, acceleration blocks, or CPU cores.

All the producers/consumers have one channel, each of which is referred to as a dedicated channel. Additionally, there are a number of pool channels available to allow multiple cores (not FMan or accelerators) to service the same channel. Note that there are channels for each external FMan port, the number of which depends on the SoC, as well as the internal offline ports.

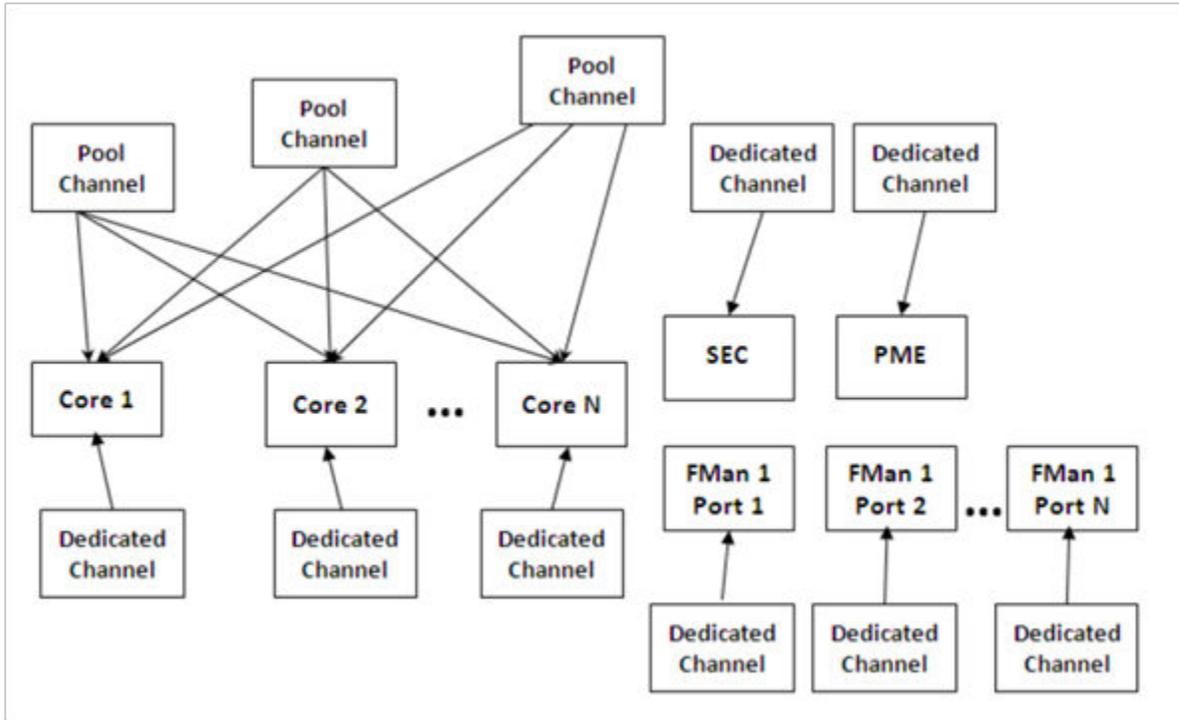


Figure 8. DPAA Channel Types

Each channel provides for eight levels of priority, each of which has its own work queue (WQ). The two highest level WQs are strict priority: traffic from WQ0 must be drained before any other traffic flows; then traffic from WQ1 and then traffic from the other six WQs is allowed to pass. The last six WQs are grouped together in two groups of three, which are configurable in a weighted round robin fashion. Most applications do not need to use all priority levels. When multiple FQs are assigned to the same WQ, QMan implements a credit-based scheme to determine which FQ is scheduled (providing frames to be processed) and how many frames (actually the credit is defined by the number of bytes in the frames) it can dequeue before QMan switches the scheduling to the next FQ on the WQ. If a higher priority WQ becomes active (that is, one of the FQs in the higher priority WQ receives a frame to become non-empty) then the dequeue from the lower priority FQ is suspended until the higher priority frames are dequeued. After the higher priority FQ is serviced, when the lower priority FQ restarts servicing, it does so with the remaining credit it had before being pre-empted by the higher priority FQ.

When the DPAA elements of the SoC are initialized, the FQs are associated with WQs, allowing the traffic to be steered to the desired core (dedicated connect channel), set of cores (pool channel), FMan, or accelerator, using a defined priority.

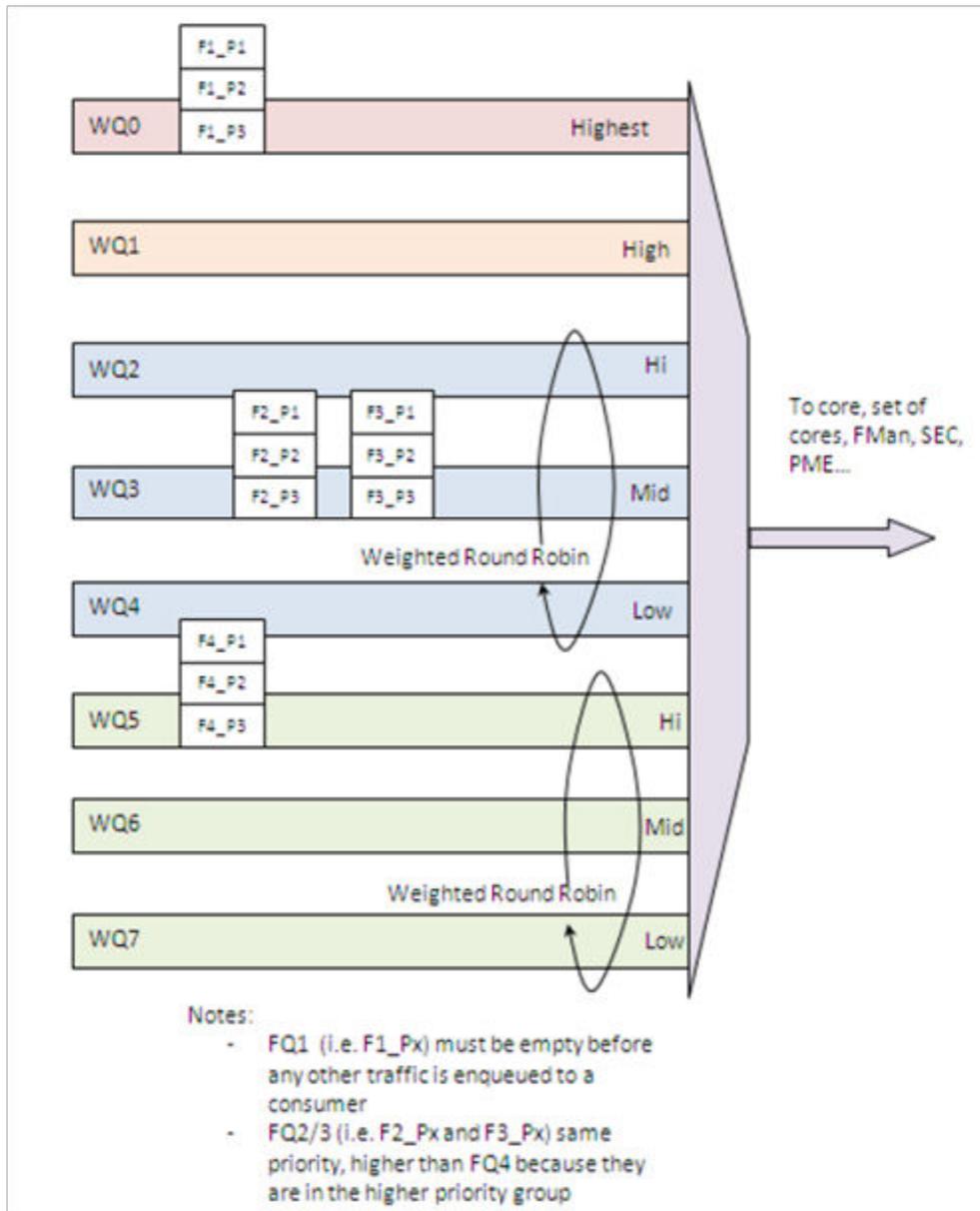


Figure 9. Prioritizing Work

QMan: Portals

A single portal exists for each non-core DPAA producer/consumer (FMan, SEC, and PME). This is a data structure internal to the SoC that passes data directly to/from the consumer’s direct connect channel.

Software portals are associated with the processor cores and are, effectively, data structures that the cores use to pass (enqueue) packets to or receive (dequeue) packets from the channels associated with that portal (core). Each SoC has at least as many software portals as there are cores. Software portals are the interface through which the DPAA provides the data processing workload for a single thread of execution.

The portal structure consists of the following:

- The Dequeue Response Ring (DQRR) determines the next packet to be processed.
- The Enqueue Command Ring (EQCR) sends packets from the core to the other elements.
- The Message Ring (MR) notifies the core of the action (for example, attempted dequeue rejected, and so on).

- The Management command and response control registers.

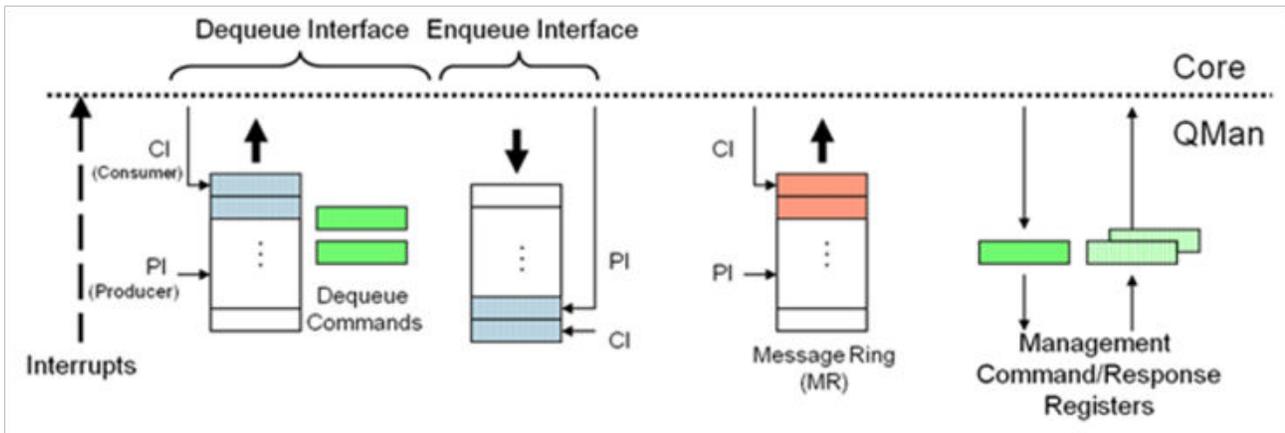


Figure 10. Processor Core Portal

On ingress, the DQRR acts as a small buffer of incoming packets to a particular core. When a section of software performs a get packet type operation, it gets the packet from a pointer provided as an entry in the DQRR for the specific core running that software. Note that the DQRR consolidates all potential channels that may be feeding frames to a particular core. There are up to 16 entries in each DQRR. Each DQRR entry contains:

- a pointer to the packet to be processed,
- an identifier of the frame queue from which the packet originated,
- a sequence number (when configured),
- and additional FMan-determined data (when configured).

When configured for push mode, QMan attempts to fill the DQRR from all the potential incoming channels. When configured in pull mode, QMan only adds one DQRR entry when it is told to by the requesting core. Pull mode may be useful in cases where the traffic flows must be very tightly controlled; however, push mode is normally considered the preferred mode for most applications.

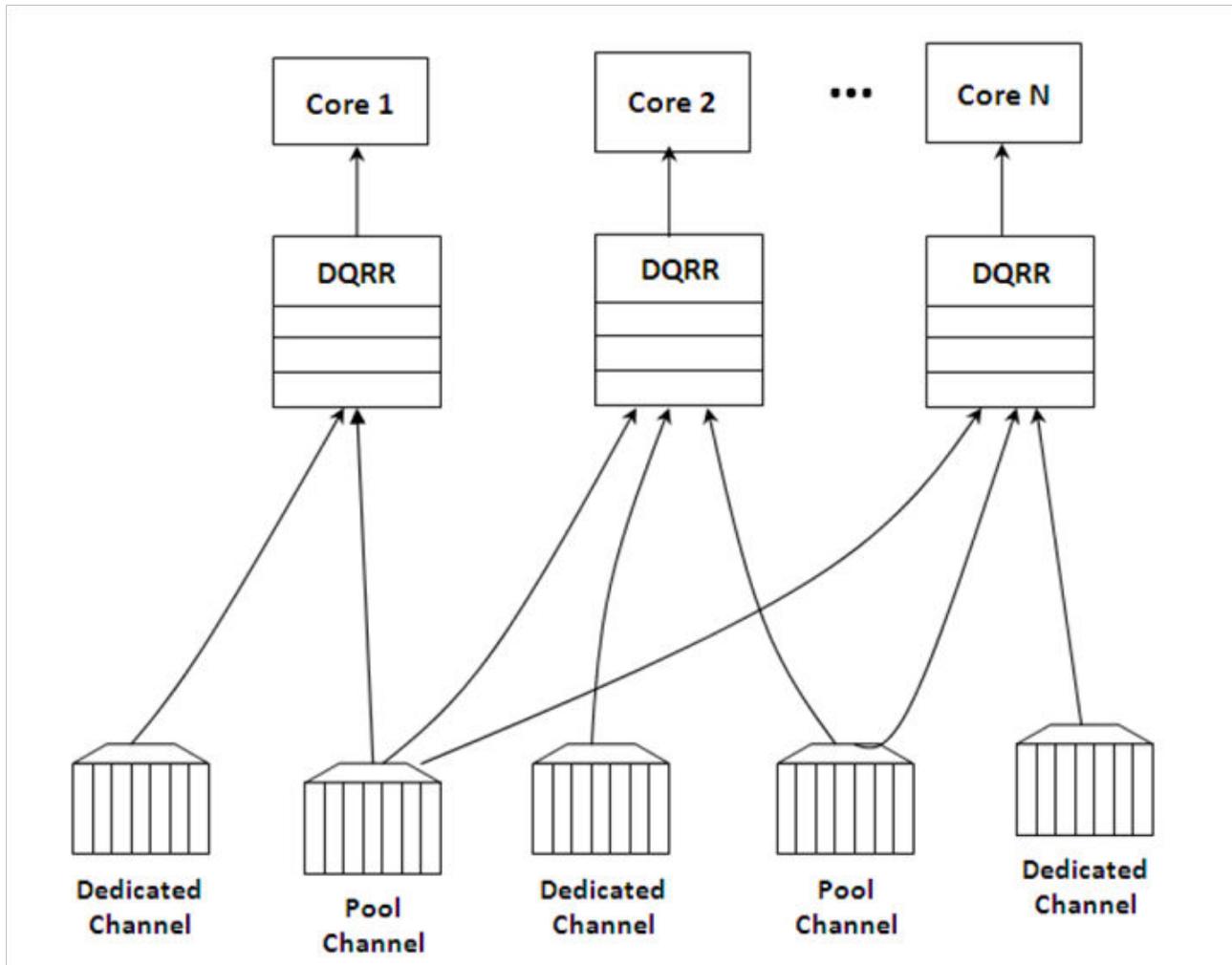


Figure 11. Ingress Channel to Portal Options

The DQRRs are tightly coupled to a processor core. The DPAA implements a feature that allows the DQRR mechanism to pre-allocate, or stash, the L1 and/or L2 cache with data related to the packet to be processed by that core. The intent is to have the data required for packet processing in the cache before the processor runs the “get packet” routine, thereby reducing the overall time spent processing a particular packet.

The following is data that may be warmed into the caches:

- The DQRR entry
- The packet or portion of the packet for a single buffer packet
- The scatter gather list for a multi-buffer packet
- Additional data added by FMan
- FQ context (A and B)

The FQ context is a user-defined space in memory that contains data associated with the FQ (per flow) to be processed. The intent is to place in this data area the state information required when processing a packet for this flow. The FQ context is part of the FQ definition, which is performed when the FQ is initialized.

The cache warming feature is enabled by configuring the capability and some definition of the FQs and QMan at system initialization time. This can provide a significant performance boost and requires little to no change in the processing flow.

When defining the system architecture, it is highly recommended that the user enable this feature and consider how to maximize its impact.

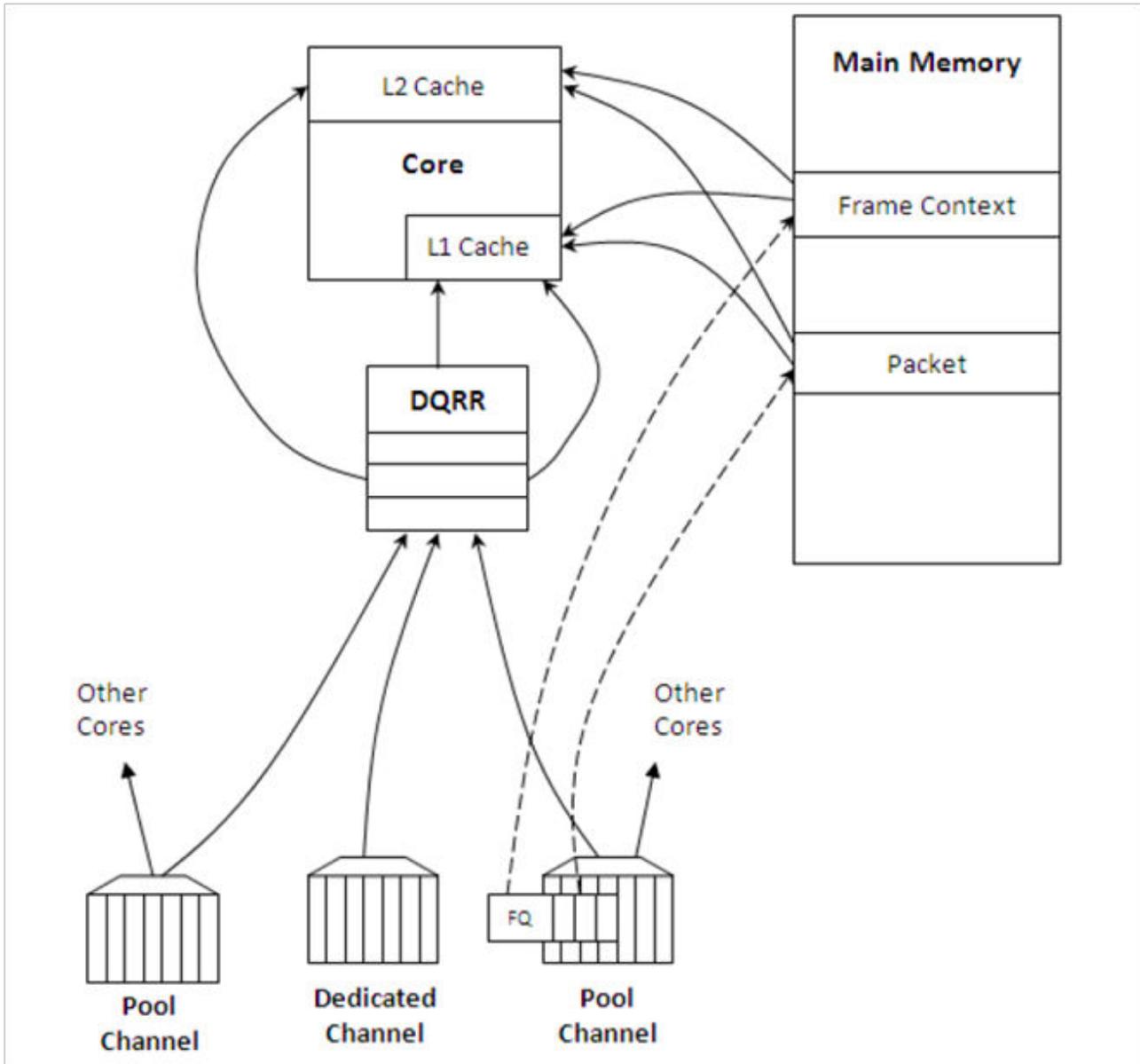


Figure 12. Cache Warming Options

In addition to getting packet information from the DQRR, the software also manages the DQRR by indicating which DQRR entry it will consume next. This is how the QMan determines when the DQRR (portal) is ready to process more frames. Two basic options are provided. In the first option, the software can update the ring pointer after one or several entries have been consumed. By waiting to indicate the consumption of multiple frames, the performance impact of the write doing this is minimized. The second option is to use the discrete consumption acknowledgment (DCA) mode. This mode allows the consumption indication to be directly associated with a frame enqueue operation from the portal (that is, after the frame has been processed and is on the way to the egress queue). This tracking of the DQRR Ring Pointer CI (Consumer Index) helps implement frame ordering by ensuring that QMan does not dequeue a frame from the same FQ (or flow) to a different core until the processing is completed.

5.2.1.5 QMan Scheduling

The QMan links the FQs to producers and consumers (of data traffic) within the SoC.

QMan: Queue schedule options

The primary communication path between QMan and the processor cores is the portal memory structure. QMan uses this interface to schedule the frames to be processed on a per-core basis. For a dedicated channel, the process is straightforward: the QMan places an entry in the DQRR for the portal (processor) of the dedicated channel and dequeues the frame from an FQ to the portal. To do this, QMan determines, based on the priority scheme (previously described) for the channel, which frame should be processed next and then adds an entry to the DQRR for the portal associated with the channel.

When configured for push mode, once the portal requests QMan to provide frames for processing, QMan provides frames until halted. When the DQRR is full and more frames are destined for the portal, QMan waits for an empty slot to become available in the DQRR and then adds more entries (frames to be processed) as slots become available.

When configured for pull mode, the QMan only adds entries to the DQRR at the direct request of the portal (software request). The command to the QMan that determines if a push or pull mode is implemented and tells QMan to provide either one or from one to three (up to three if there are that many frames to be dequeued) frames at a time. This is a tradeoff of smaller granularity (for one frame only) versus memory access consolidation (if the up to three frames option is selected).

When the system is configured to use pool channels, a portal may get frames from more than one channel and a channel may provide frames (work) to more than one portal (core). QMan dequeues frames using the same mechanism described above (updating DQRR) and QMan also provides for some specific scheduling options to account for the pool channel case in which multiple cores may process the same channel.

QMan: Default Scheduling

The default scheduling is to have an FQ send frames to the same core for as long as that FQ is active. An FQ is active until it uses up its allocated credit or becomes empty. After an FQ uses its credit, it is rescheduled again, until it is empty. For its schedule opportunity, the FQ is active and all frames dequeued during the opportunity go to the same core. After the credit is consumed, QMan reactivates that FQ but may assign the flow processing to a different core. This provides for a sticky affinity during the period of the schedule opportunity. The schedule opportunity is managed by the amount of credit assigned to the FQ.

NOTE

A larger credit assigned to an FQ provides for a stickier core affinity, but this makes the processing work granularity larger and may affect load balancing.

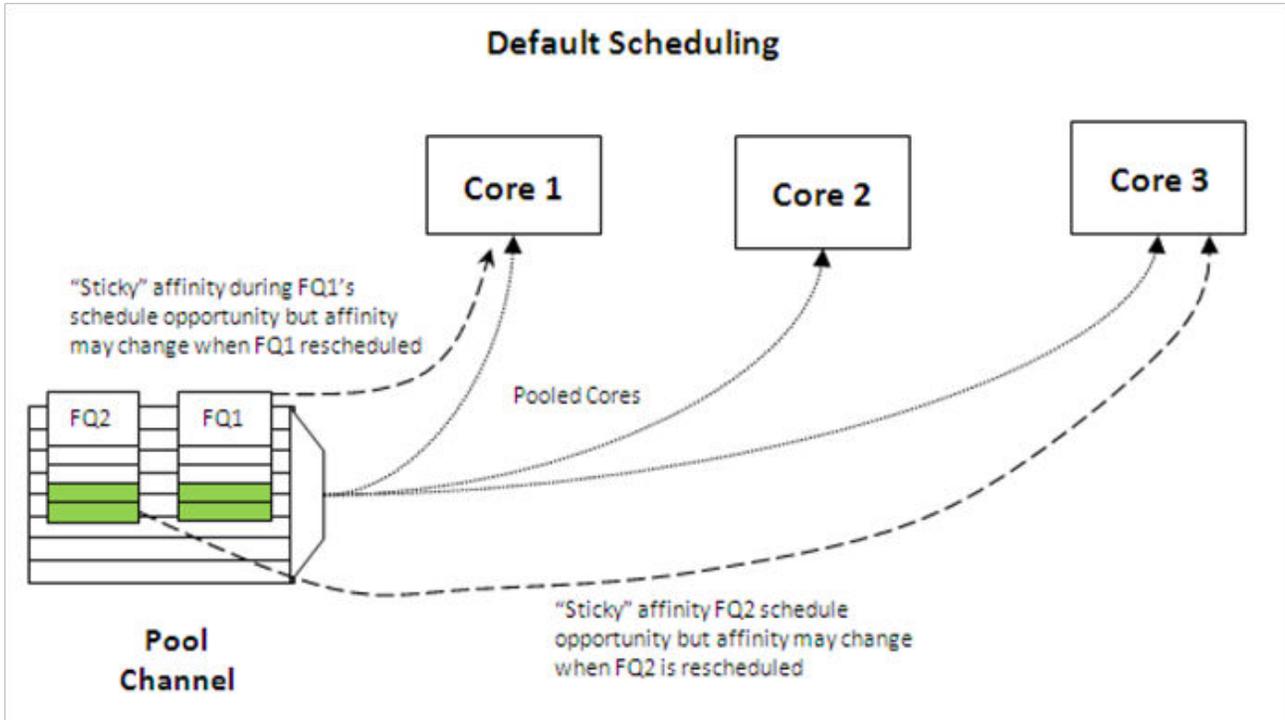


Figure 13. Default Scheduling

QMan: Hold Active Scheduling

With the hold active option, when the QMan assigns an FQ to a particular core, that Q is affined to that core until it is empty. Even after the FQ's credit is consumed, when it is rescheduled with the next schedule opportunity, the frames go to the same core for processing. This effectively makes the flow-to-core affinity stickier than the default case, ensuring the same flow is processed by the same core for as long as there are frames queued up for processing. Because the flow-to-core affinity is not hard-wired as in the dedicated channel case, the software may still need to account for potential order issues. However, because of flow-to-core biasing, the flow state data is more likely to remain in L1 or L2 cache, increasing hit rates and thus improving processing performance. Because of the specific QMan implementation, only four FQs may be in held active state at a given time.

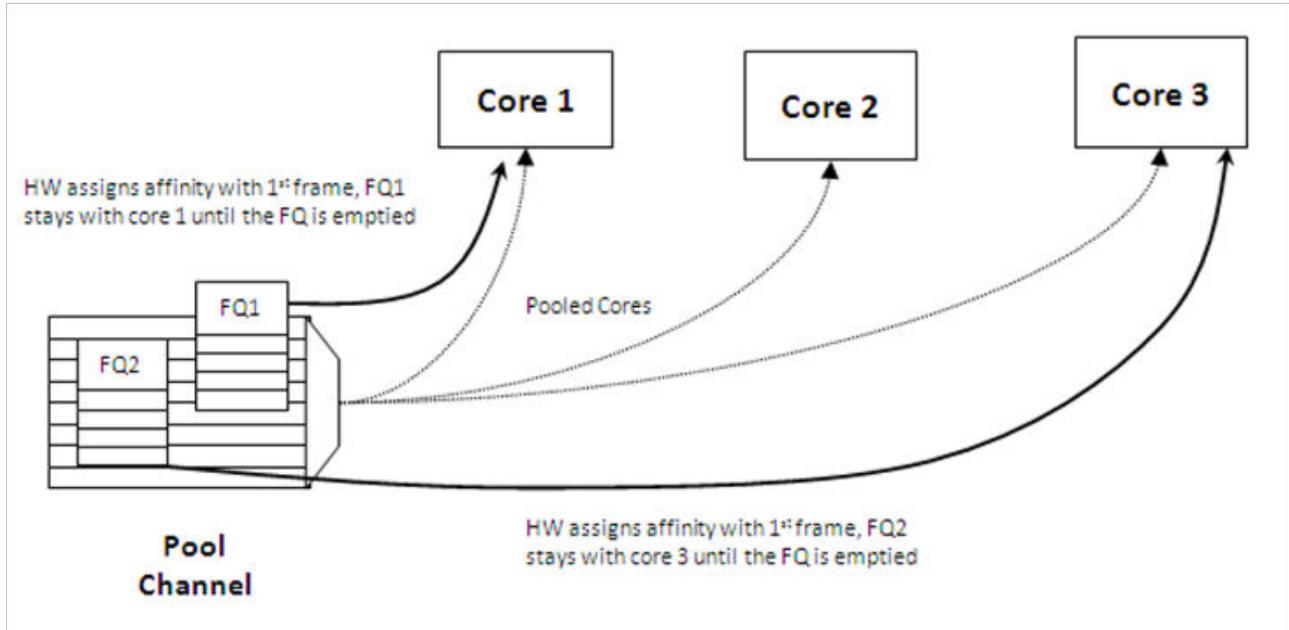


Figure 14. Hold Active Scheduling

QMan: Avoid blocking scheduling

Avoid blocking scheduling QMan can also be scheduled in the avoid blocking mode, which is mutually exclusive to hold active. In this mode, QMan schedules frames for an FQ to any available core in the pool channel. For example, if the credit allows for three frames to be dequeued, the first frame may go to core 1. But, when that dequeue fills core 1's DQRR, QMan finds the next available DQRR entry in any core in the pool. With avoid blocking mode there is no biasing of the flow to core affinity. This mode is useful if a particular flow either has no specific order requirements or the anticipated processing required for a single flow is expected to consume more than one core's worth of processing capability.

Alternatively, software can bypass QMan scheduling and directly control the dequeue of frame descriptors from the FQ. This mode is implemented by placing the FQ in parked state. This allows software to determine precisely which flow will be processed (by the core running the software). However, it requires software to manage the scheduling, which can add overhead and impact performance.

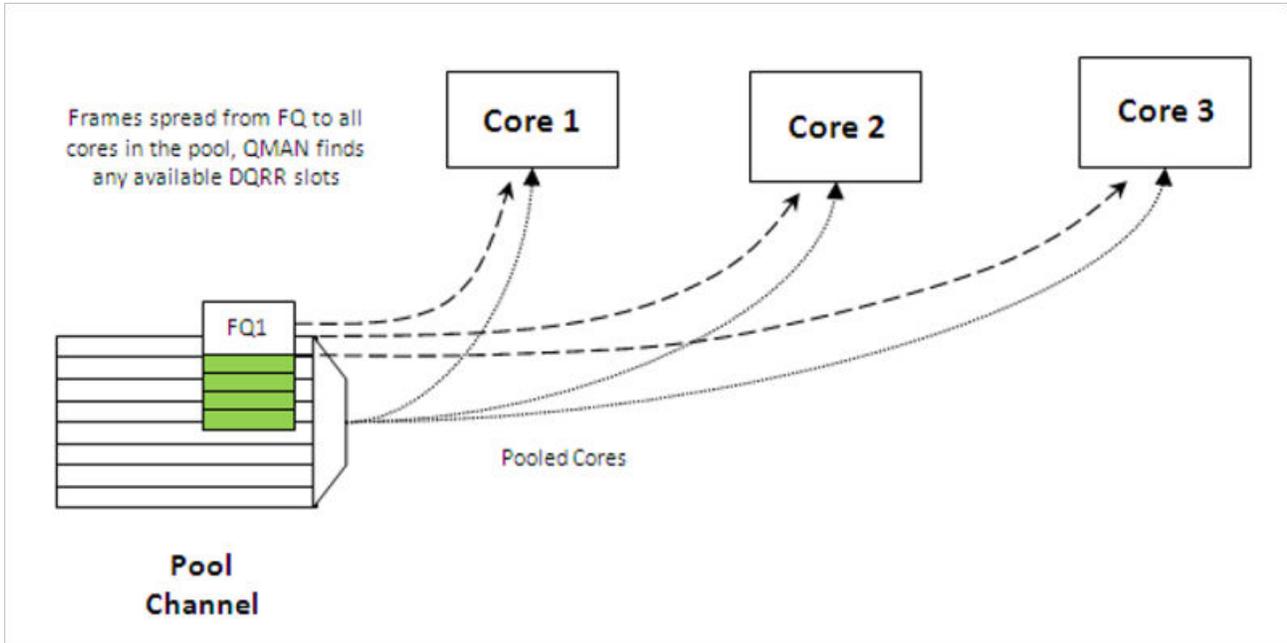


Figure 15. Avoid Blocking Scheduling

QMan: Order Definition/ Restoration

The QMan provides a mechanism to strictly enforce ordering. Each FQ may be defined to participate in the process of an order definition point and/or an order restoration point. On ingress, an order definition point provides for a 14 bit sequence number assigned to each frame (incremented per frame) in a FQ in the order in which they were received on the interface. The sequence number is placed in the DQRR entry for the frame when it is dequeued to a portal. This allows software to efficiently determine which packet it is currently processing in the sequence without the need to access a shared (between cores) data structure. On egress, an order restoration point delays placing a frame onto the FQ until the expected next sequence number is encountered. From the software standpoint, once it has determined the relative sequence of a packet, it can enqueue it and resume other processing in a fire-and-forget manner.

NOTE

The order definition points and order restoration points are not dependent on each other; it is possible to have one without the other depending on application requirements. To effectively use these mechanisms, the packet software must be aware of the sequence tagging.

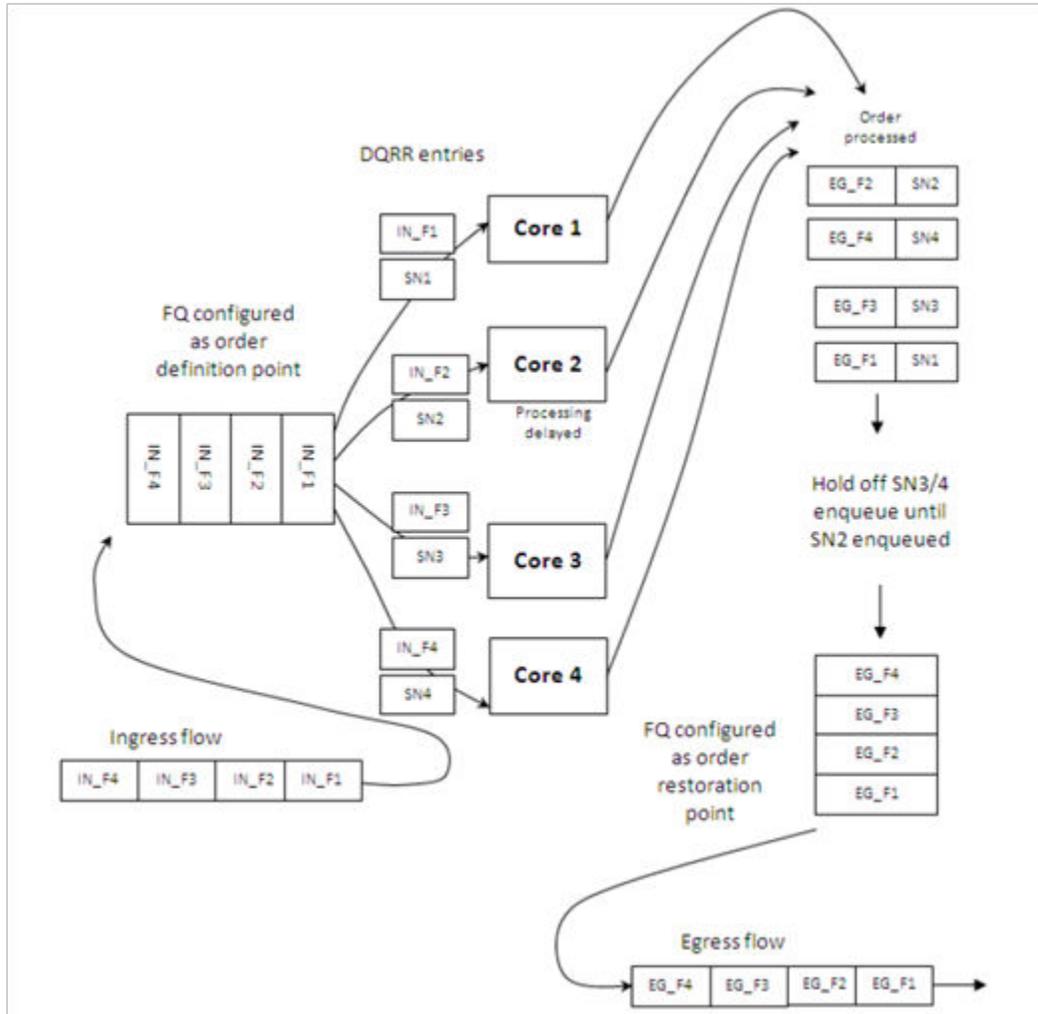


Figure 16. Order Definition/Restoration

As processors enqueue packets for egress, it is possible that they may skip a sequence number because of the nature of the protocol being processed. To handle this situation, each FQ that participates in the order restoration service maintains its own Next Expected Sequence Number (NESN). When the difference between the sequence number of the next expected and the most recently received sequence number exceeds the configurable ORP threshold, QMan gives up on the missing frame(s) and autonomously advances the NESN to bring the skew within threshold. This causes any deferred enqueues that are currently held in the ORP link list to become unblocked and immediately enqueue them to their destination FQ. If the “skipped” frame arrives after this, the ORP can be configured to reject or immediately enqueue the late arriving frame.

5.2.1.6 BMan

The BMan block manages the data buffers in memory. Processing cores, FMan, SEC and PME all may get a buffer directly from the BMan without additional software intervention. These elements are also responsible for releasing the buffers back to the pool when the buffer is no longer in use.

Typically, the FMan directly acquires a buffer from the BMan on ingress. When the traffic is terminated in the system, the core generally releases the buffer. When the traffic is received, processed, and then transmitted, the same buffer may be used throughout the process. In this case, the FMan may be configured to release the buffer automatically, when the transmit completes.

The BMan also supports single or multi-buffer frames. Single buffer frames generally require the adequately defined (or allocated) buffer size to contain the largest data frame and minimize system overhead. Multi-buffer frames potentially allow

better memory utilization, but the entity passed between the producers/consumers is a scatter-gather table (that then points to the buffers within the frame) rather than the pointer to the entire frame, which adds an extra processing requirement to the processing element.

The software defines pools of buffers when the system is initialized. The BMan unit itself manages the pointers to the buffers provided by the software and can be configured to interrupt the software when it reaches a condition where the number of free buffers is depleted (so that software may provide more buffers as needed).

5.2.1.7 Order Handling

The DPAA helps address packet order issues that may occur as a result of running an application in a multiple processor environment. And there are several ways to leverage the DPAA to handle flow order in a system. The order preservation technique maps flows such that a specific flow always executes on a specific processor core.

For the case that DPAA handles flow order, the individual flow will not have multiple execution threads and the system will run much like a single core system. This option generally requires less impact to legacy, single-core software but may not effectively utilize all the processing cores in the system because it requires using a dedicated channel to the processors. The FMan PCD can be configured to either directly match a flow to a core or to use the hashing to provide traffic spreading that offers a permanent flow-to-core affinity.

If the application must use pool channels to balance the processing load then the software must be more involved in the ordering. The software can make use of the order restoration point function in QMan, which requires the software to manage a sequence number for frames enqueued on egress. Alternatively, the software can be implemented to maintain order by biasing the stickiness of flow affinity with default or hold active scheduling; lock contention and cache misses can be biased to increase performance.

If there are no order requirements then load balancing can be achieved by associating the non-ordered traffic to a pool of cores.

NOTE

All of these techniques may be implemented simultaneously on the same SoC; as long as the flow definition is precise enough to split the traffic types, it is simply a matter of proper defining the FQs and associating them to the proper channels in the system.

Using the exact match flow definition to preserve order

The simplest technique for preserving order is to route the ingress traffic of an individual flow to a particular core. For the particular flow in question, the system appears as a legacy, single-core programming model and, therefore, has minimal impact on the structure of the software. In this case, the flow definition determines the core affinity of a flow.

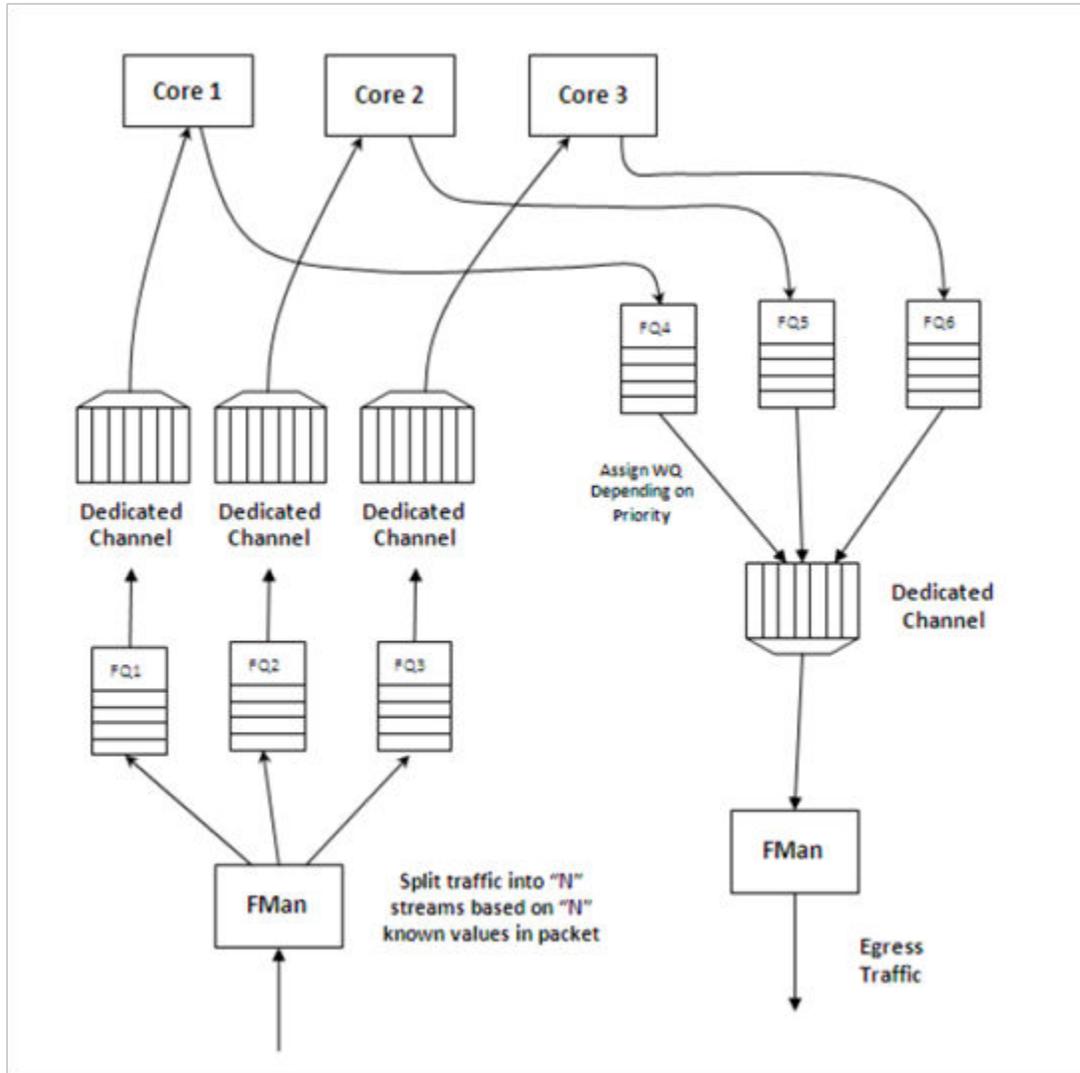


Figure 17. Direct Flow-to-Core Mapping (Order Preserved)

This technique is completely deterministic: the DPAA forces specific flows to a specific processor, so it may be easier to determine the performance assuming the ingress flows are completely understood and well defined. Notice that a particular processor core may become overloaded with traffic while another sits idle for increasingly random flow traffic rates.

To implement this sort of scheme, the FMan must be configured to exactly match fields in the traffic stream. This approach can only be used for a limited number of total flows before the FMan's internal resources are consumed.

In general, this sort of hard-wired approach should be reserved for either critical out-of-band traffic or for systems with a small number of flows that can benefit from the highly deterministic nature of the processing.

Using hashing to distribute flows across cores

The FMan can be configured to extract data from a field or fields within the data frame, build a key from that, and then hash the resultant key into a smaller number. This is a useful technique to handle a larger number of flows while ensuring that a particular flow is always associated with a particular core. An example is to define a flow as an IPv4 source + IPv4 destination address. Both fields together constitute 64 bits, so there are 264 possible combinations for the flow in that definition. The FMan then uses a hash algorithm to compress this into a manageable number of bits. Note that, because the hash algorithm is consistent, packets from a particular flow always go to the same FQ. By utilizing this technique, the flows can be spread in a pseudo-random, consistent (per flow) manner to a smaller number of FQs. For example, hashing the 64 bits down to 2

bits spreads the flows among four queues. Then these queues can be assigned to four separate cores by using a dedicated channel; effectively, this appears as a single-core implementation to any specific flow.

This spreading technique works best with a large number of possible flows to allow the hash algorithm to evenly spread the traffic between the FQs. In the example below, when the system is only expected to have eight flows at a given time, there is a good chance the hash will not assign exactly two flows per FQ to evenly distribute the flows between the four cores shown. However, when the number of flows handled is in the hundreds, the odds are good that the hash will evenly spread the flows for processing.

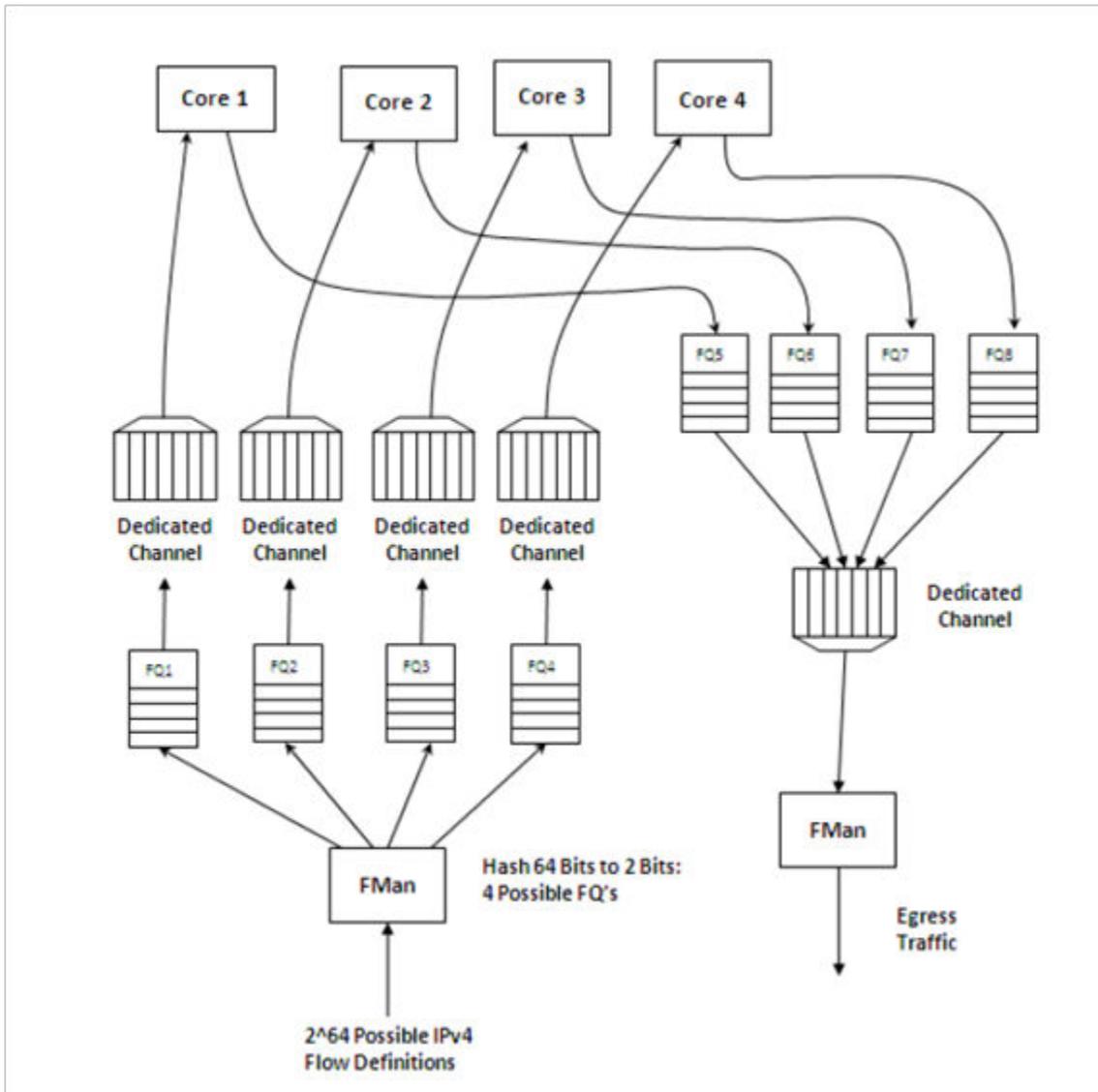


Figure 18. Simple flow distribution via hash (order preserved)

To optimize cache warming, the total number of hash buckets can be increased with flow-to-core affinity maintained. When the number of hash values is larger than the number of expected flows at a given time, it is likely though not guaranteed that each FQ will contain a single flow. For most applications, the penalty of a hash collision is two or more flows within a single FQ. In the case of multiple flows within a single FQ, the cache warming and temporary core affinity benefits are reduced unless the flow order is maintained per flow.

Note that there are 24 bits architected for the FQ ID, so there may be as many as 16 million FQs in the system. Although this total may be impractical, this does allow for the user to define more FQs than expected flows in order to reduce the likelihood of a hash collision; it also allows flexibility in assigning FQID's in some meaningful manner. It is also possible to hash some

fields in the data frame and concatenate other parse results, possibly allowing a defined one-to-one flow to FQ implementation without hash collisions.

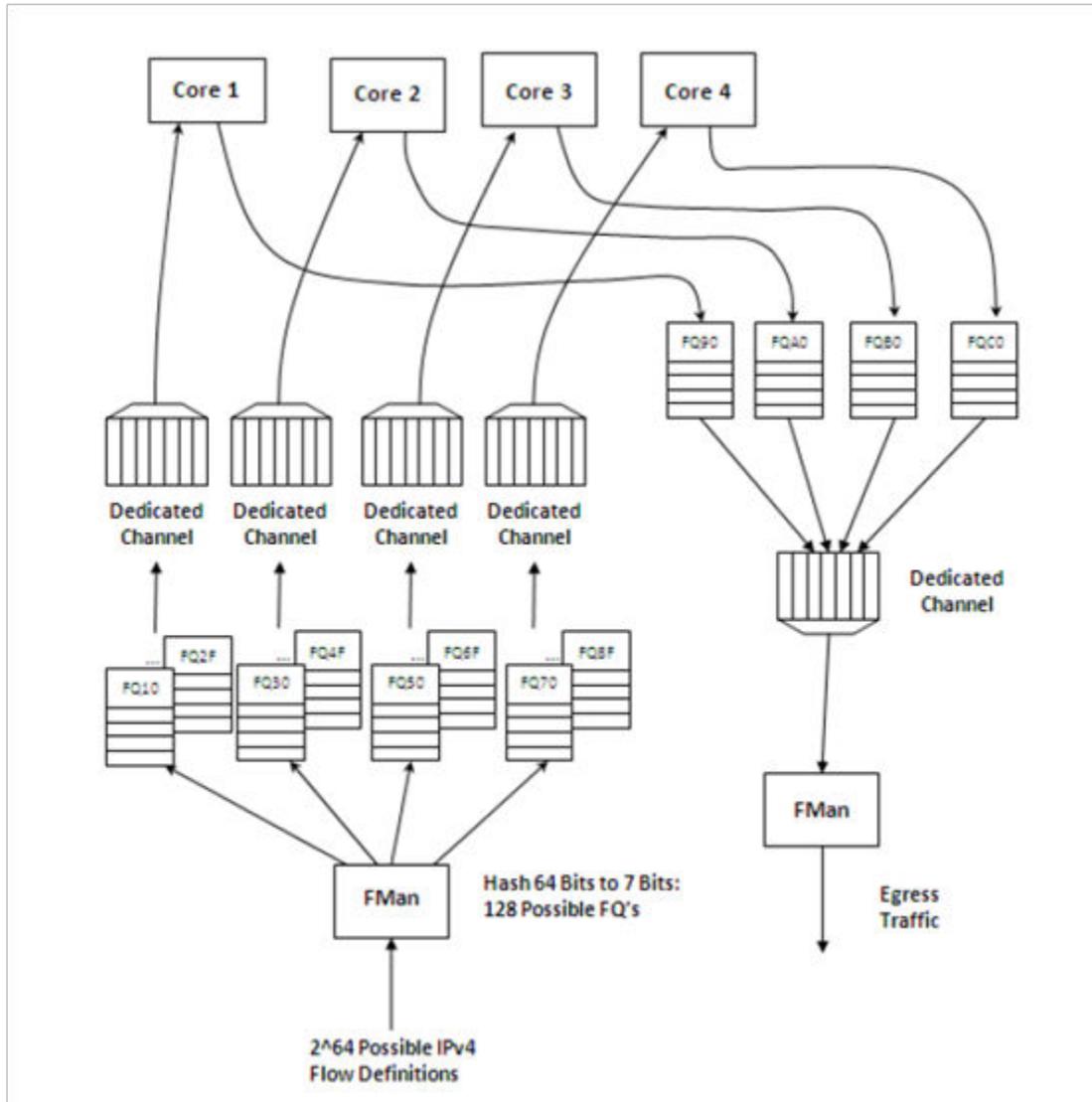


Figure 19. . Using hash to assign one flow per FQ (order preserved and cache stashing effective)

5.2.1.8 Pool Channels

A user may employ a pool channel approach where multiple cores may pool together to service a specific set of flows. This alternative approach allows potentially better processing balance, but increases the likelihood that packets may be processed out of order allowing egress packets to pass ingress packets.

So far, the techniques discussed in this white paper have involved assigning specific flows to the same core to ensure that the same core always processes the same flow or set of flows, thereby preserving flow order. However, depending on the nature of the flows being processed (that is, variable frame sizes, difficulty efficiently spreading due to the nature of the flow contents, and so on), this may not effectively balance the processing load among the cores. Alternatively, a user may employ a pool channel approach where multiple cores may pool together to service a specific set of flows. This alternative approach allows potentially better processing balance, but increases the likelihood that packets may be processed out of order allowing egress packets to pass ingress packets. When the application does not require flows to be processed in order, the pool channel approach allows the easiest method for balancing the processing load. When a pool channel is used and order is required, the software must maintain order. The hardware order preservation may be used by the software to implement order

without requiring locked access to shared state information. When the system uses a software lock to handle order then the default scheduling and hold active scheduling tends to minimize lock contention.

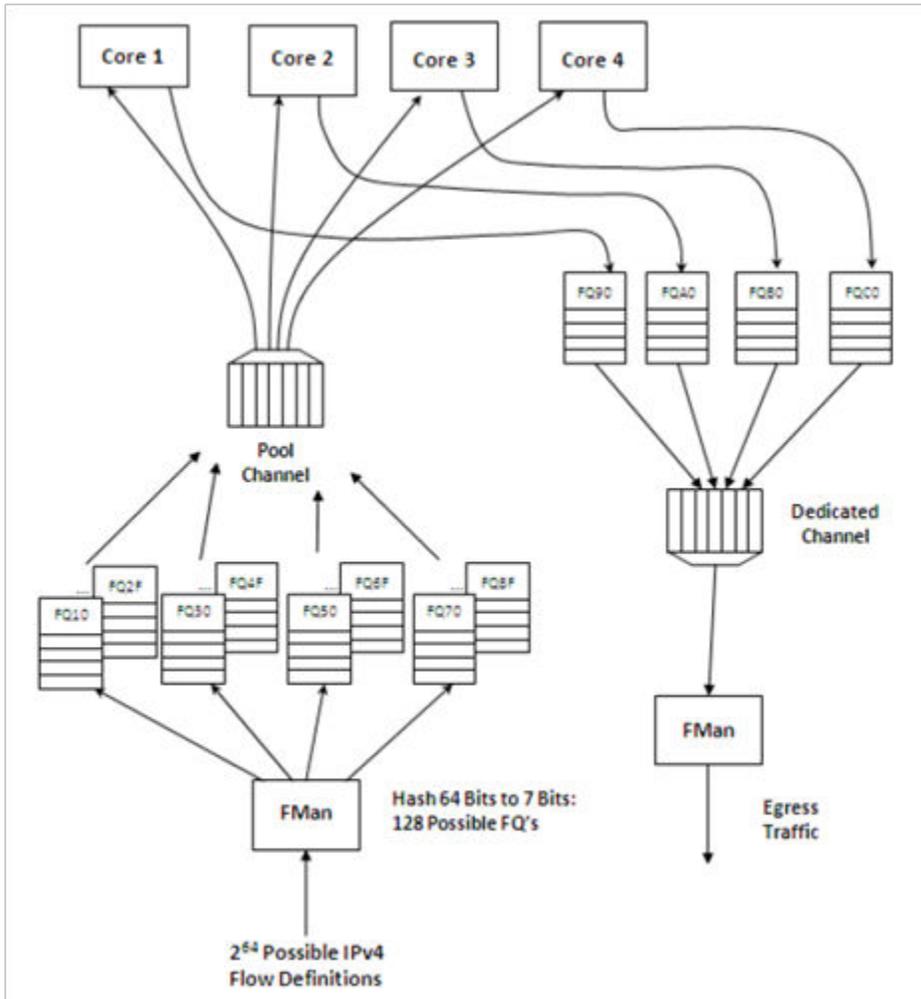


Figure 20. Using pool channel to balance processing

Order preservation using hold active scheduling and DCA mode

As shown in the examples above, order is preserved as long as two or more cores never process frames from the same flow at the same time. This can also be accomplished by using hold active scheduling along with discrete consumption acknowledgment (DCA) mode associated with the DQRR. Although flow affinity may change for an FQ with hold active scheduling when the FQ is emptied, if the new work (from frames received after the FQ is emptied) is held off until all previous work completes, then the flow will not be processed by multiple cores simultaneously, thereby preserving order.

When the FQ is emptied, QMan places the FQ in hold suspended state, which means that no further work for that FQ is enqueued to any core until all previously enqueued work is completed. Because DCA mode effectively holds off the consumption notification (from the core to QMan) until the resultant processed frame is enqueued for egress, this implies processing is completely finished for any frames in flight to the core. After all the in-flight frames have been processed, QMan reschedules the FQ to the appropriate core.

NOTE

After the FQ is empty and when in hold active mode, the affinity is not likely to change. This is because the indication of “completeness” from the core currently processing the flow frees up some DQRR slots that could be used by QMan when it restarts enqueueing work for the flow. The possibility of the flow-to-core affinity changing when the FQ empties is only discussed as a worst case possibility with regards to order preservation.

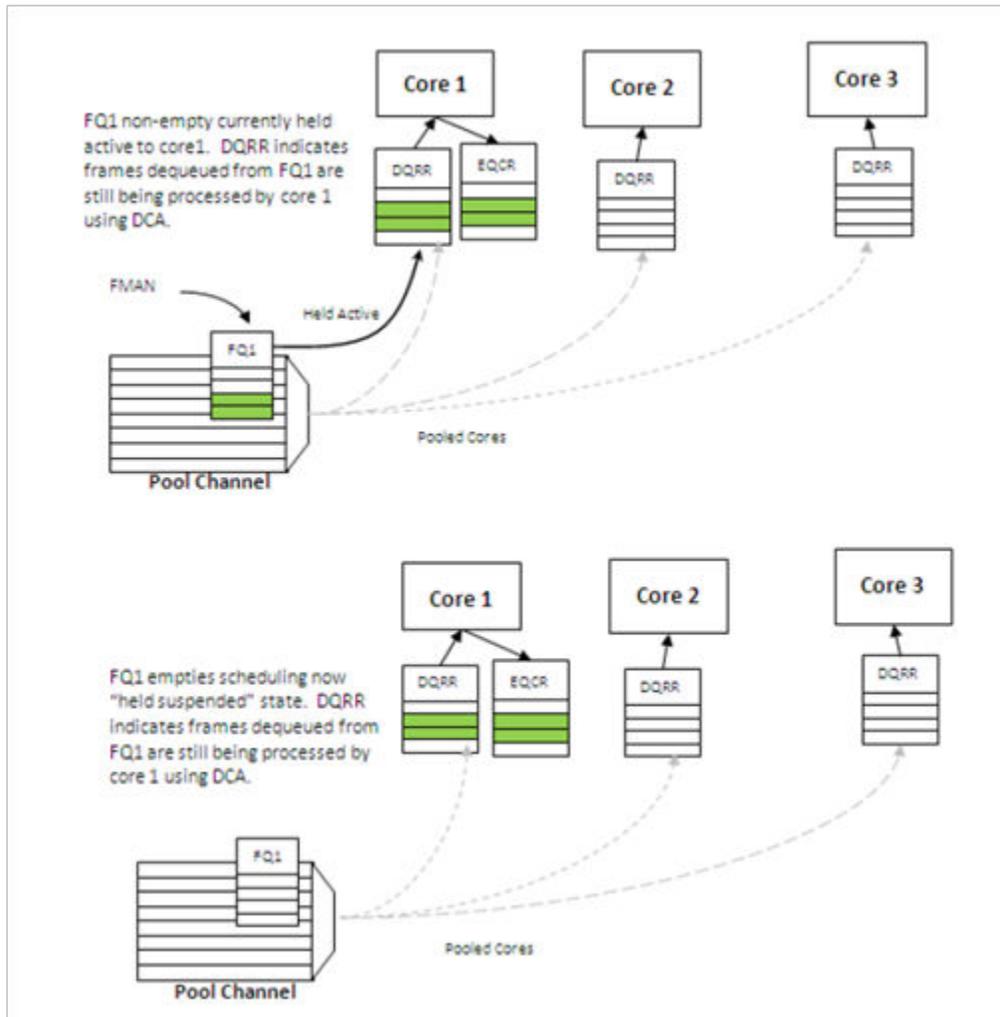


Figure 21. Hold active to held suspended mode

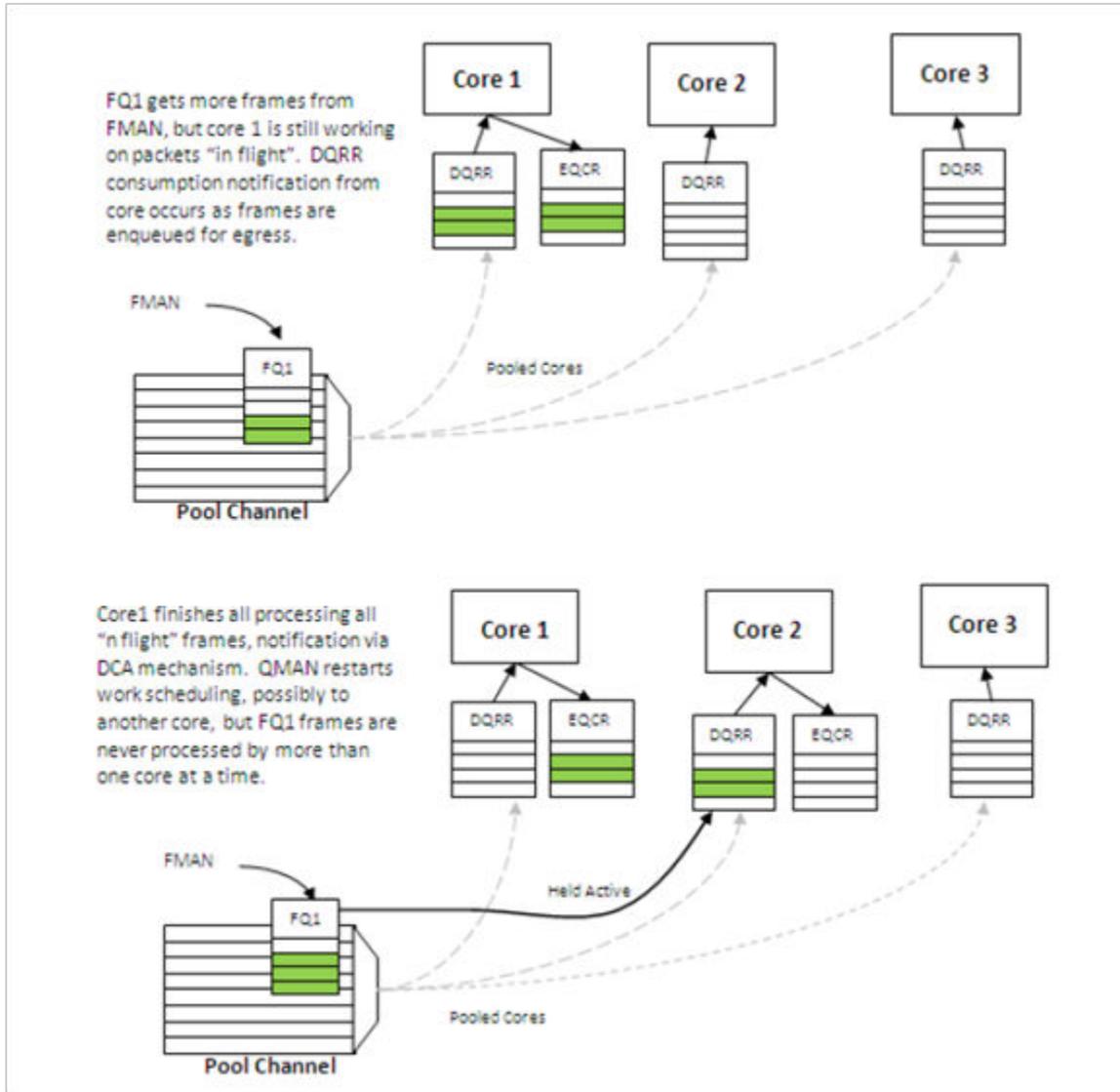


Figure 22. Held suspended to hold active mode

Congestion management

From an overall system perspective, there are multiple potential overflow conditions to consider. The maximum number of frames active in the system (the number of frames in flight) is determined by the amount of memory allocated to the Packed Frame Queue Descriptors (PQFD's). Each PQFD is 64 bytes and can identify up to three frames, so the total number of frames that can be identified by the PQFDs is equal to the amount of memory allocated for PQFD space divided by 64 bytes (per entry) multiplied by three (frames per entry).

A pool of buffers may deplete in BMan. This depends on how many buffers have been assigned by software for BMan. BMan may raise an interrupt to request more buffers when in a depleted state for a given pool; the software can manage the congestion state of the buffer pools in this manner.

In addition to these high-level system mechanisms, congestion management may also be identified specific to the FQs. A number of FQs can be grouped together to form a congestion group (up to 256 congestion groups per system for most DPAA SoCs). These FQs need not be on the same channel. The system may be configured to indicate congestion either by consider the aggregate number of bytes within the FQ's in the congestion group or by the aggregate number of frames within the congestion group. The frame count option is useful when attempting to manage the number of buffers in a buffer pool as they

are used by a particular core or group of cores. The byte count is useful to manage the amount of system memory used by a particular core or group of cores.

When the total number of frames/bytes within the frames in the congestion group exceeds the set threshold, subsequent enqueues to any of the FQs in the group are rejected; in general, the frame is dropped. For the congestion group mechanism, the decision to reject is defined by a programmed weighted random early discard (WRED) algorithm programmed when the congestion group is defined.

In addition, a specific FQ can be set to a particular maximum allowable depth (in bytes); after the threshold is reached enqueue attempts will be rejected. This is a maximum threshold: there is no WRED algorithm for this mechanism. Note that, when the FQ threshold is not set, a specific FQ may fill until some other mechanism (because it's part of a congestion group or system PQFD depletion or BMAN depletion) prevents the FQ from getting frames. Typically, FQs within a congestion group are expected to have a maximum threshold set for each FQ in the group to ensure a single queue does not get stuck and unfairly consume the congestion group. Note that, when an FQ does not have a queue depth set and/or is not a part of a congestion group, the FQ has no maximum depth. It would be possible for a single queue to have all the frames in the system, until the PQFD space or the buffer pool is exhausted.

5.2.1.9 Application Mapping

The first step in application mapping is to determine how much processing capability is required for tasks that may be partitioned separately.

Processor core assignment

Consider a typical networking application with a set of distinct control and data plane functionality. Assigning two cores to perform control plane tasks and six cores to perform data plane tasks may be a reasonable partition in an eight-core device. When initially identifying the SoC required for the application, along with the number of cores and frequencies required, the designer makes some performance assumptions based on previous designs and/or applicable benchmark data.

Define flows

Next, define what flows will be in the system. Key considerations for flow definition include the following:

- Total number of flows expected at a given time within the system
- Desired flow-to-core affinity, ingress flow destination
- Processor load balancing
- Frame sizes (may be fixed or variable)
- Order preservation requirement
- Traffic priority relative to the flows
- Expected bandwidth requirement of specific flows or class of flows
- Desired congestion handling

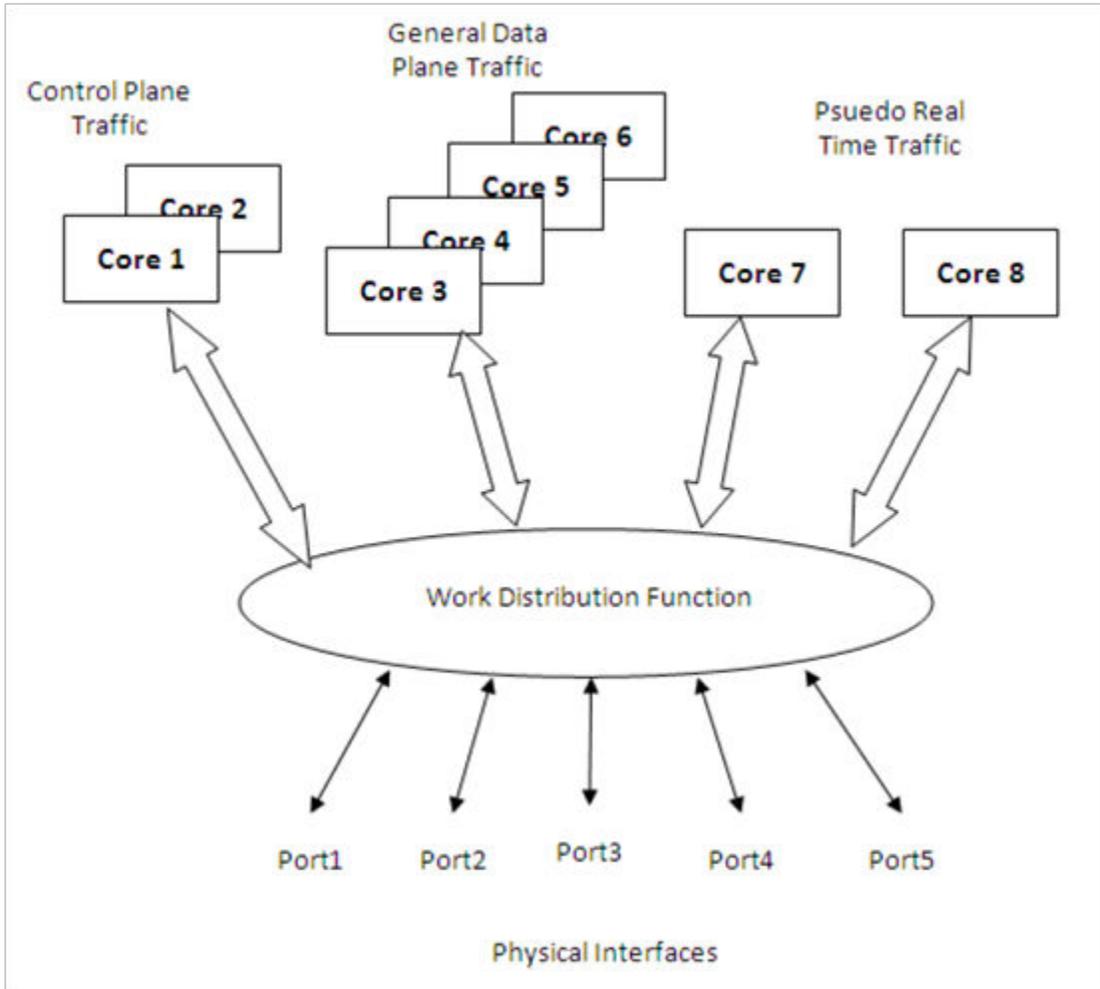


Figure 23. Example Application with Three Classes

In the figure above, two cores are dedicated to processing control plane traffic, four cores are assigned to process general data traffic and special time critical traffic is split between two other cores. In this case, assume the traffic characteristics in the following table. With this system-level definition, the designer can determine which flows are in the system and how to define the FQs needed.

Table 9. Traffic characteristics

Characteristic	Definition
Control plane traffic	<ul style="list-style-type: none"> Terminated in the system and any particular packet sent has no dependency on previous or subsequent packets (no order requirement). May occur on ports 1, 2 or 3. Ingress control plane traffic on port three is higher priority than the other ports. Any ICMP packet on ports 1, 2 or 3 is considered control plane traffic. Control plane traffic makes up a small portion of the overall port bandwidth.

Table continues on the next page...

Table 9. Traffic characteristics (continued)

Characteristic	Definition
General data plane traffic	<ul style="list-style-type: none"> • May occur on ports 1, 2 or 3 and is expected to comprise the bulk of the traffic on these ports. • The function performed is done on flows and egress packets must match the order of ingress packets. • A flow is identified by the IP source address. • The system can expect up to 50 flows at a time. • All flows have the same priority and a lower priority than any control plane traffic. • It is expected that software will not always be able to keep up with this traffic and the system should drop packets after some amount of packets are within the system.
Pseudo real-time traffic	<ul style="list-style-type: none"> • A high amount of determinism is required by the function. • This traffic only occurs on port 4 and port 5 and is identified by a proprietary field in the header; any traffic on these ports without the proper value in this field is dropped. • All valid ingress traffic on port 4 is to be processed by core 7, ingress traffic on port 5 processed by core 8. • There are only two flows, one from port 4 to port 5 and one from port 5 to port 4, egress order must match ingress order. • The traffic on these flows are the highest priority.

Identify ingress and egress frame queues (FQs)

For many applications, because the ingress flow has more implications for processing, it is easier to consider ingress flows first. In the example above, the control plane and pseudo real-time traffic FQ definitions are fairly straightforward. For the control plane ingress, one FQ for lower priority traffic on ports 1 and 2 and one for the higher priority traffic would work. Note that two ports can share the same queue on ingress when it does not matter for which core the traffic is destined. For ingress pseudo real-time traffic, there is one FQ on port 4 and one FQ on port 5.

The general data plane ingress traffic is more complicated. Multiple options exist which maintain the required ordering for this traffic. While this traffic would certainly benefit from some of the control features of the QMan (cache warming, and so on), it is best to have one FQ per flow. Per the example, the flow is identified by the IP source (32-bits), which consists of too many bits to directly use as the FQID. The hash algorithm can be used to reduce the 32-bits to a smaller number; in this case, six bits would generate 64 queues, which is more than the anticipated maximum flows at a given time. However, this is not significantly more than maximum flow expected, so more FQs can be defined to reduce hash collisions. Note that, in this case, a hash collision implies that two flows are assigned to the same FQ. As the ingress FQs fill directly from the port, the packet order is still maintained when there is a collision (two flows into one FQ). However, having two flows in the same FQ tends to minimize the impact of cache warming. There may be other possibilities to refine the definition of flows to ensure a one-to-one mapping of flows to FQs (for example, concatenating other fields in the frame) but for this example assume that an 8 bit hash (256 FQs) minimizes the likelihood of two flows in the FQ to an acceptable level.

Consider the case in which, on ingress, there is traffic that does not match any of the intended flow definitions. The design can handle these by placing unidentifiable packets into a separate garbage FQ or by simply having the FMan discard the packets.

On egress control traffic, because the traffic may go out on three different ports, three FQs are required. For the egress pseudo real-time traffic, there is one queue for each port as well.

For the egress data plane traffic, there are multiple options. When the flows are pinned to a specific core, it might be possible to simply have one queue per port. In this case, the cores would effectively be maintaining order. However, for this example, assume that the system uses the order definition/order restoration mechanism previously described. In this case, the system needs to define an FQ for each egress flow. Note that, since software is managing this, there is no need for any sort of hash algorithm to spread the traffic; the cores will enqueue to the FQ associated with the flow. When there are no more than 50 flows in the system at one time, and number of egress flows per port is unknown, the system could define 50 FQs for each port when the DPAA is initialized.

Define PCD configuration for ingress FQs

This step involves defining how the FMan splits the incoming port traffic into the FQs. In general, this is accomplished using the PCD (Parse, Classify, Distribute) function and results in specific traffic assigned to a specific FQID. Fields in the incoming packet may be used to identify and split the traffic as required. For this key optimization case, the user must determine the correct field. The example is as follows:

- For the ingress control traffic, the ICMP protocol identifier is the selector or key. If the traffic is from ports 1 or 2 then that traffic goes to one FQID. If it is from port 3, the traffic goes to a different FQID because this needs to be separated and given a higher priority than the other two ports.
- For the ingress data plane traffic, the IP source field is used to determine the FQID. The PCD is then configured to hash the IP source to 8 bits, which will generate 256 possible FQs. Note that this is the same, regardless of whether the packet came from ports 1, 2, or 3.
- For the ingress pseudo real-time traffic, the PCD is configured to check for the proprietary identifier. If there is a match then the traffic goes to an FQID based on the ingress port. If there is no match then the incoming packet is discarded. Also, the soft parser needs to be configured/programmed to locate the proprietary identifier.

Note that the FQID number itself can be anything (within the 24 bits to define the FQ). To maintain meaning, use a numbering scheme to help identify the type of traffic. For the example, define the following ingress FQIDs:

- High priority control: FQID `0x100`
- Low priority control: FQID `0x200`
- General data plane: FQID `0x1000 - 0x10FF`
- Pseudo real-time traffic: FQID `0x2000` (port 4), FQID `0x2100` (port 5)

The specifics for configuring the PCDs are described in the **DPAA Reference Manual (link)** and in the Software Developer Kit (SDK) used to develop the software.

5.2.1.10 FQ/WQ/Channel

For each class of traffic in the system, the FQs must be defined together with both the channel and the WQ to which they are associated. The channel association affines to a specific processor core while the WQ determines priority.

Consider the following by class of traffic:

- The control traffic goes to a pool of two cores with priority given to traffic on port 3.
- The general data plane traffic goes to a pool of 4 cores.
- The pseudo real-time traffic goes to two separate cores as a dedicated channel.

Note that, when the FQ is defined, in addition to the channel association, other parameters may be configured. In the application example, the FQs from 1000 to 10FF are all assigned to the same congestion group; this is done when the FQ is initialized. Also, for these FQs it is desirable to limit the individual FQ length; this would also be configured when the FQ is initialized.

Because the example application is going to use order definition/order restoration mode, this setting needs to be configured for each FQ in the general data plane traffic (FQID `0x1000-0x10FF`). Note that order is not required for the control plane traffic and that order is preserved in the pseudo real-time traffic because the ingress traffic flows are mapped to specific cores.

QMan configuration considerations include the congestion management and pool channel scheduling. A congestion group must be defined as part of QMan initialization. (Note that the FQ initialization is where the FQ is bound to a congestion group.) This is where the total number of frames and the discard policy of the congestion group are defined. Also, consider the QMan scheduling for pool channels. In this case, the default of temporarily attaching an FQ to a core until the FQ is empty will likely work best. This tends to keep the caches current, especially for the general data plane traffic on cores 3-6.

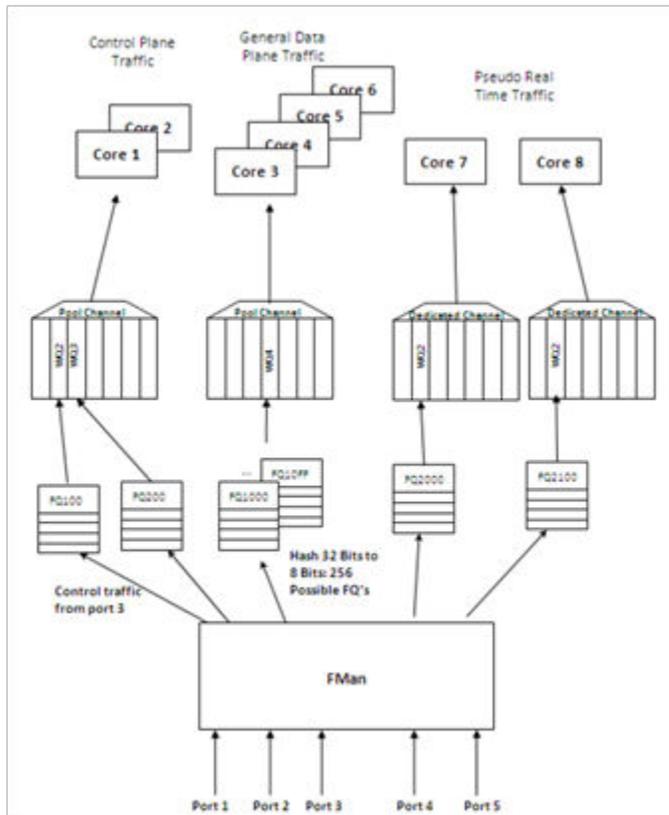


Figure 24. Ingress application map

Define egress FQ/WQ/channel configuration

For egress, the packets still flow through the system using the DPAA, but the considerations are somewhat different. Note that each external port has its own dedicated channel; therefore, to send traffic out of a specific port, the cores enqueue a frame to an FQ associated with the dedicated channel for that port. Depending on the priority level required, the FQ is associated with a specific work queue.

For the example, the egress configuration is as follows:

- For control plane traffic, there needs to be separate queues for each port this traffic may use. These FQs must be assigned to a WQ that is higher in priority than the WQ used for the data plane traffic. The example shown includes a strict priority (over the data plane traffic) for ports 1 and 2 with the possibility of WRED with the data plane traffic on port 3.
- Because the example assumes that the order restoration facility in the FQs will be utilized, there must be one egress FQ for each flow. The initial system assumptions are for up to 50 flows of this type; however, the division by port is unknown, the FQs can be assigned so that there are at least 50 for each port. Note that FQs can be added when the flow is discovered or they can be defined at system initialization time.
- For the pseudo real-time traffic, per the initial assumptions, core 7 sends traffic out of port 4 and core 8 sends traffic out of port 5. As the flows are per core, the order is preserved because of this mapping. These are assigned to WQ2, which allows definition for even higher priority traffic (to WQ1) or lower priority traffic for future definition on these ports.

As stated before, the FQIDs can be whatever the user desires and should be selected to help keep track of what type of traffic the FQ's are associated. For this example:

- Control traffic for ports 1, 2, 3 are FQID 300, 400, 500 respectively.
- Data plane traffic for ports 1, 2, 3 are FQID 3000-303F, 4000-403F, and 5000-503F respectively, this provides for 64 FQ's per port on egress.
- The pseudo real-time traffic uses FQID 6000 for port 4 and 7000 for port 5.

Because this application makes use of the order restoration feature, an order restoration point must be defined for each data plane traffic flow. Also, congestion management on the FQs may be desirable. Consider that the data plane traffic may come in on multiple ports but may potentially be consolidated such that it egresses out a single port. In this case, more traffic may be attempted to be enqueued to a port than the port interface rate may allow, which may cause congestion. To manage this possibility, three congestion groups can be defined each containing all the FQs on each of the three ports that may have the control plus data plane traffic. As previously discussed, it may be desirable to set the length of the individual FQs to further manage this potential congestion.

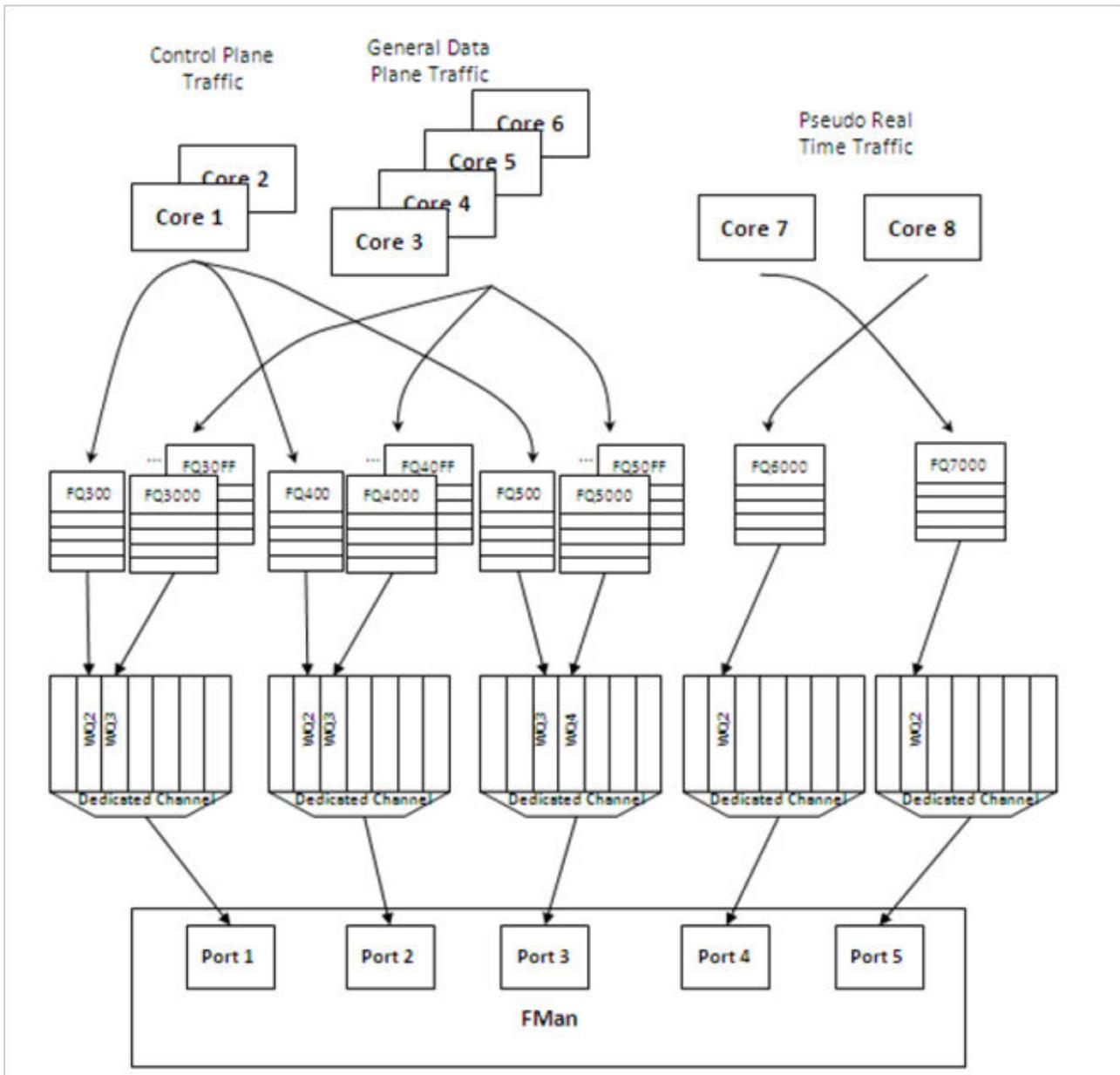


Figure 25. Egress application map

End of Document

5.2.2 Queue Manager (QMan) and Buffer Manager (BMan)

5.2.2.1 QMan/BMan Drivers Release Notes

Description

This document describes Linux and USDPAA drivers for the QMan and BMan hardware blocks underlying the QorIQ data path. QMan and BMan have independent drivers but their implementation and interfaces are very much analogous due to the similar CCSR and Corenet programming interfaces for each. As such, we will describe here "the driver", when in fact the description applies to both the QMan and BMan drivers equally and independently.

The driver targets the Linux and USDPAA environments. The majority of the code is shared between the environments. Environmental differences are dealt with by including a compatibility layer in the USDPAA code. This code redefines Linux-specific functionality for use in the other environments (for example `irqs` and `spinlocks`).

The driver has two parts to it, "config" and "portal", corresponding to the two complimentary programming interfaces exposed by the device itself - these are described below. Additionally there is a self-test module for each driver that uses the portal interface to perform some basic tests provided one or more portals are made available to the OS via its device-tree.

CCSR, or "global config"

The CCSR map and associated registers allows the device to be configured and controlled in a global/un-partitioned manner. This includes such basic notions as configuring the device's private memory region(s), configuring the hardware interfaces that are exposed by QMan/BMan to the dependent hardware blocks (CAAM, PME, Fman), managing global device error interrupts, etc. Only one "control" operating system should map to this CCSR register space in the case that a hypervisor is managing multiple guests. Other operating systems like secondary Linux instances or USDPAA applications do not have access to CCSR registers.

Corenet portals

Use of QMan/BMan is via a number of independent portals located within sub-regions of a corenet memory map. Each portal has its own copy of the device interfaces that allow independent and parallel use of QMan/BMan functionality, possibly by different operating systems or by different processors (or threads) within a single operating system. When a hypervisor is managing multiple guests, each guest device tree indicates to the guest the portal it has access to. The device tree specifies a one to one mapping between cores and QMan/BMan portals. This mapping is assumed by the high-level QMan/BMan APIs. This should not be changed unless only low-level QMan/BMan APIs are to be used.

Functionality

Configuration

The QMan device is configured via device-tree nodes and by some compile-time options controlled via Linux's Kconfig system. See the "QMan and BMan Kernel Configure Options" section for more info.

API

For the Linux kernel, the C interface of the QMan and BMan drivers provides access to portal-based functionality for arbitrary higher-layer code, hiding all the mux/demux/locking details required for shared use by multiple driver layers (networking, pattern matching, encryption, IPC, etc.) The driver makes 1-to-1 associations between cpus and portals to improve cache locality and reduce locking requirements. The QMan API permits users to work with Frame Queues and callbacks, independently of other users and associated portal details. The BMan API permits users to work with Buffer Pools in a similar manner.

For USDPAA, the driver associates portals with threads (in the *pthread*s sense), so the above comments about "shared use by multiple driver layers" only applies with respect to code executed within the thread owning a portal. To benefit from cache locality, and particularly from portal stashing, USDPAA-enabled threads are generally expected to be configured to execute

on the same core that the portal is assigned to. Indeed, the USDPAA API for threads to call to initialise a portal takes the core as a function parameter. Please see the USDPAA User Guide for more information (as well as the “Queue Manager, Buffer Manager API Reference Manual”).

DPAA allocator

The DPAA allocator is a purely software-based range-allocator, but this must be explicitly seeded with a hard-coded range of values and is not shared between operating systems. The DPAA allocator is used to allocate all QMan and BMan resource, i.e bman-bpid, qman-fqid, qman-pool, qman-cgrid, ceetm-sp, ceetm-lni, ceetm-lfqid, ceetm-ccgrid.

Sysfs Interface

QMan and BMan have a sysfs interface. Refer to the Queue Manager, Buffer Manager API reference Manual for details

Debugfs Interface

Both the QMan and BMan have a debugfs interface available to assist in device debugging. The code can be built either as a loadable module or statically.

Module Loading

The drivers are statically linked into the kernel. Driver self-tests and the debugfs interface may be built as dynamically loadable modules.

QMan and BMan Kernel Configure Options

Common Kernel Configure Options	Description
CONFIG_STAGING	Required in order to make “staging” drivers such as QMan/BMan available.
CONFIG_FSL_DPA	Required to build either QMan and/or BMan drivers.
CONFIG_FSL_DPA_CHECKING	Compiles in additional sanity-checks, at the expense of minor performance degradation. Recommended for debugging, but not for benchmarking.
CONFIG_FSL_DPA_CAN_WAIT	Compiles in support for interfaces and functionality that allow callers to optionally be put to “sleep” waiting for temporarily blocked resources to become available rather than returning errors. Eg. enqueueing when an enqueue ring is full. This is enabled unconditionally on linux.
CONFIG_FSL_DPA_CAN_WAIT_SYNC	Similar to “_CAN_WAIT”, but supports additional API flags for waiting for asynchronous operations to complete. Eg. after starting a volatile dequeue, wait for all dequeues to complete. This is enabled unconditionally on linux.
CONFIG_FSL_DPA_PIRQ_FAST	If set, causes portals to initialise with fast-path interrupt sources enabled. (Otherwise, polling APIs must be called to perform fast-path processing.) This is enabled unconditionally on linux.
CONFIG_FSL_DPA_PIRQ_SLOW	If set, causes portals to initialise with slow-path interrupt sources enabled. (Otherwise, polling APIs must be called to perform slow-path processing.) This is enabled unconditionally on linux.
CONFIG_FSL_DPA_PORTAL_SHARE	Compiles in support for sharing one CPU's portal with all online CPUs that do not have their own. Useful when assigning most portals to USDPAA applications and leaving only a minimum for kernel requirements, in which case Tx events on all CPUs can be handled by the network driver. This is enabled by default, as the microscopic performance overhead of checking this option is not noticeable in the kernel environment.

QMan Kernel Configure Options	Description
CONFIG_FSL_QMAN	Required to build the QMan driver
CONFIG_FSL_QMAN_CONFIG	Handles config/CCSR nodes in the device-tree and initialises the corresponding devices
CONFIG_FSL_QMAN_TEST	Builds a self-test kernel module (static or dynamic) that will, if QMan portal nodes are available in the device-tree, exercise one of the portals and panic() the kernel if any errors are detected.
CONFIG_FSL_QMAN_TEST_STASH_POTATO	This requires the presence of multiple unused cpu-affine portals, and performs a "hot potato" style test enqueueing/dequeueing a frame across a series of FQs scheduled to different portals (and cpus). The intention is to test stashing. The "potato" will visit each "spoon" (portal/cpu pair) during the test. Each "potato" frame has a single cacheline of data that is read-modify-written by each cpu/portal before passing it to the next.
CONFIG_FSL_QMAN_TEST_HIGH	This requires the presence of cpu-affine portals, and performs high-level API testing with them (whichever portal(s) are affine to the cpu(s) the test executes on).
CONFIG_FSL_QMAN_TEST_ERRATA	This requires the presence of cpu-affine portals, and performs testing that handling for known hardware-errata is correct.
CONFIG_FSL_QMAN_DEBUGFS	This option enables files in the debugfs filesystem.

BMan Kernel Configure Options	Description
CONFIG_FSL_BMAN	Required to build the BMan driver
CONFIG_FSL_BMAN_CONFIG	Handles config/CCSR nodes in the device-tree and initialises the corresponding devices
CONFIG_FSL_BMAN_TEST	Builds a self-test kernel module (static or dynamic) that will, if BMan portal nodes are available in the device-tree, exercise one of the portals and panic() the kernel if any errors are detected.
CONFIG_FSL_BMAN_TEST_HIGH	Performs high-level API testing.
CONFIG_FSL_BMAN_TEST_THRESH	Multi-threaded testing of BMan pool depletion handling.
CONFIG_FSL_BMAN_DEBUGFS	This option enables files in the debugfs filesystem.

Device-tree nodes

Device tree nodes are used to describe QMan/BMan resources to the driver, some of which are specific to control-plane s/w (i.e. depending on CCSR access) and some of which relate to portal usage for control and data plane s/w.

CCSR, or "global config"

The "fsl,qman" and "fsl,bman" nodes (i.e. these "compatible" property types) indicate the presence and location of the 4Kb "Configuration, Control, and Status Register" (CCSR) space, for use by a single control-plane driver instance to initialise and manage the device. The device-tree usually groups all such CCSR maps as sub-nodes under a parent node that represents the SoCs entire CCSR map, usually named "soc" or "ccsr". For example;

```

soc {
    #address-cells = <1>;
    #size-cells = <1>;
    device_type = "soc";

```

```
compatible = "simple-bus";

ddr1: memory-controller@8000{
    [...]
};
i2c@118000 {
    [...]
};
mpic: pic@40000 {
    [...]
};

qman: qman@318000 {
    compatible = "fsl,qman";
    reg = <0x318000 0x1000>;
    interrupts = <16 2 1 3>;
    /* Commented out, use default allocation */
    /* fsl,qman-fqd = <0x0 0x20000000 0x0 0x01000000>; */
    /* fsl,qman-pfdr = <0x0 0x21000000 0x0 0x01000000>; */
};
bman: bman@31a000 {
    compatible = "fsl,bman";
    reg = <0x31a000 0x1000>;
    interrupts = <16 2 1 3>;
    /* Same as fsl,qman-*, use default allocation */
    /* fsl,bman-fbpr = <0x0 0x22000000 0x0 0x01000000>; */
};
[...]
```

Contiguous memory

The `fsl,qman-fqd`, `fsl,qman-pfdr`, and `fsl,bman-fbpr` properties can be used to specify which contiguous sub-regions of memory should be used for the various memory requirements of QMan/BMan. The properties use 64-bit values, so 4 cells express the address/size 2-tuple to use. In the above example, if uncommented, the QMan/BMan resources would be allocated in the range `0x20000000-0x221ffffff`, with 16MB each for QMan FQD and PFDR memory and BMan FBPR memory. If these properties are not specified (or they are commented out) in the device tree, then default values hard-coded within the QMan and BMan drivers are used instead. The linux kernel will reserve these memory ranges early on boot-up. Note that in the case of a hypervisor scenario, these memory ranges are relative to the partition memory space of the control-plane guest OS.

QMan and BMan Corenet portals

The QMan Corenet portal interface in QorIQ P4080/P3041/P5020 provides up to 10 distinct memory-mapped interfaces for use by software to interact efficiently with QMan functionality. The number may be different for other SoCs - the QMan driver determines the available corenet portals from the device tree. The portal nodes are at the physical address scope (unlike the device-tree node for the BMan device itself, which is within the “soc” physical address node that represents CCSR). These nodes indicate the physical address ranges of the cache-enabled and cache-inhibited sub-regions of the portal (respectively), and look something like the following;

```
qman-portals@ffe420000 {
    #address-cells = <0x1>;
    #size-cells = <0x1>;
    compatible = "simple-bus";
    qportal0: qman-portal@0 {
        [...]
    };
    [...]
};
```

```

qportal3: qman-portal@c000 {
    cell-index = <0x3>;
    compatible = "fsl,qman-portal";
    reg = <0xc000 0x4000 0x103000 0x1000>;
    interrupts = <110 0x2 0 0>;
    fsl,qman-channel-id = <0x3>;
};
[...]
};

```

QMan FQID-range allocation

The "fsl,fqid-range" node (i.e. these "compatible" property types) indicates a range of FQIDs to use for FQID allocation by the QMan driver. The range within the node is specified using a property of the same name, and whose two cells are the starting FQID value and the count. Multiple nodes can be provided to seed the allocator with a discontinuous set of FQIDs.

Eg. to specify that the allocator use FQIDs between 256 and 512 inclusive;

```

qman-fqids@0 {
    compatible = "fsl,fqid-range";
    fsl,fqid-range = <256 256>;
};

```

BMan BPID-range allocation

The "fsl,bpool-range" node (i.e. these "compatible" property types) indicates a range of BPIDs to use for BPID allocation by the BMan driver. The range within the node is specified using a property of the same name, and whose two cells are the starting BPID value and the count. Multiple nodes can be provided to seed the allocator with a discontinuous set of BPIDs.

Eg. to specify that the allocator use BPIDs between 32 and 64 inclusive;

```

bman-bpids@0 {
    compatible = "fsl,bpid-range";
    fsl,bpid-range = <32 32>;
};

```

Compile-time Configuration Options

The "Kernel Configure Options" above describe the compile-time configuration options for the kernel. The device tree entries are also "compile-time", and are described above.

Source Files

As mentioned earlier, the QMan/BMan drivers support Linux and USDPAA environments. Many of the files have the same contents between the different environments, though the files are located at different paths to satisfy the different build systems for each.

For DPAA QMan drivers, all the files are located in `drivers/staging/fsl_qbman/` directory

For the device trees, all files are located in `arch/powerpc/boot/dts/fsl/` and `arch/arm64/boot/dts/freescale/`

USDPAA

Source Files	Description
include/usdpaa/fsl_qman.h	The QMan driver APIs
<i>Table continues on the next page...</i>	

Table continued from the previous page...

Source Files	Description
include/usdpaa/fsl_bman.h	The BMan driver APIs
include/usdpaa/fsl_usd.h	The USDPAA-specific APIs for QMan/BMan (eg. Binding portals to threads, support for UIO-based interrupt handling, etc.)
include/usdpaa/compat.h	The QMan/BMan driver compatibility shims
include/usdpaa/compat_list.h	The QMan/BMan driver compatibility shims, linked-list support.
src/qbman/qman_*.c	The QMan driver
src/qbman/bman_*.c	The BMan driver
src/qbman/dpa_sys.h	USDPAA-specific definitions shared by the QMan/BMan drivers.
src/qbman/dpa_alloc.c	USDPAA support for dpa allocator.
src/qbman/06-usdpaa-uio.rules	Udev rules to create appropriately-named /dev entries when the kernel registers portals as UIO devices.

Build Procedure

The procedure is a standard SDK build, which includes Linux kernel and USDPAA drivers by default.

Test Procedure

The QMan/BMan drivers are used by all Linux kernel software that communicates with datapath functionality such as CAAM, PME, and/or Fman. (The exception is that kernel cryptographic acceleration presently bypasses QMan/BMan interfaces by using the device's own "job queue" interface.) Use of such datapath-based functionality provides test-coverage of user-facing features of the QMan/BMan drivers in the Linux environment. This complements the QMan/BMan unit tests that are run during development but are not part of the release. For USDPAA, all applications and tests use QMan and BMan interfaces in a fundamental way, so all imply a degree of test-coverage.

Additionally, for Linux, the QMan and BMan self-tests target QMan and BMan directly without involving other datapath blocks. If these are built statically into the kernel and the device-tree makes one or more QMan and/or BMan portals available, then the self-tests will run during the kernel boots and log output to the boot console. The output of both QMan and BMan tests resembles the following excerpts;

Detecting the CCSR and portal device-tree nodes;

```
[...]
Qman ver:0a01,01,02
[...]
Bman ver:0a02,01,00
[...]
BMan err interrupt handler present

BMan portal initialised, cpu 0

BMan portal initialised, cpu 1

BMan portal initialised, cpu 2

BMan portal initialised, cpu 3

BMan portal initialised, cpu 4
```

```
BMan portal initialised, cpu 5
BMan portal initialised, cpu 6
BMan portal initialised, cpu 7
BMan portals initialised
BMan: BPID allocator includes range 32:32
QMan err interrupt handler present
QMan portal initialised, cpu 0
QMan portal initialised, cpu 1
QMan portal initialised, cpu 2
QMan portal initialised, cpu 3
QMan portal initialised, cpu 4
QMan portal initialised, cpu 5
QMan portal initialised, cpu 6
QMan portal initialised, cpu 7
QMan portals initialised
QMan: FQID allocator includes range 256:256
QMan: FQID allocator includes range 32768:32768
QMan: CGRID allocator includes range 0:256
QMan: pool channel allocator includes range 33:15
[...]
```

Running the QMan and BMan self-tests;

```
[...]
BMAN: --- starting high-level test ---
BMAN: --- finished high-level test ---
[...]
qman_test_high starting
VDQCR (till-empty);
VDQCR (4 of 10);
VDQCR (6 of 10);
scheduled dequeue (till-empty)
Retirement message received
qman_test_high finished
[...]
```

Running the BMan threshold test;

```
[...]
bman_test_thresh: start
```

```
bman_test_thresh: buffers are in
thread 0: starting
thread 1: starting
thread 2: starting
thread 3: starting
thread 4: starting
thread 5: starting
thread 6: starting
thread 7: starting
thread 0: draining...
cb_depletion: bpid=62, depleted=2, cpu=0
cb_depletion: bpid=62, depleted=2, cpu=1
cb_depletion: bpid=62, depleted=2, cpu=2
cb_depletion: bpid=62, depleted=2, cpu=3
cb_depletion: bpid=62, depleted=2, cpu=4
cb_depletion: bpid=62, depleted=2, cpu=5
cb_depletion: bpid=62, depleted=2, cpu=6
cb_depletion: bpid=62, depleted=2, cpu=7
thread 0: draining done.
thread 0: exiting
thread 1: exiting
thread 2: exiting
thread 3: exiting
thread 4: exiting
thread 5: exiting
thread 6: exiting
thread 7: exiting
bman_test_thresh: done
[...]
```

Running the QMan hot potato test;

```
[...]
qman_test_hotpotato starting
Creating 2 handlers per cpu...
Number of cpus: 8, total of 16 handlers
Sending first frame
Received final (8th) frame
qman_test_hotpotato finished
[...]
```

If the self-tests detect any errors, they will `panic()` the kernel immediately, so if the kernel gets beyond the QMan/BMan self-tests then the tests passed.

Changes since SDKv1.7

- Add new CEETM APIs to allow the user to set the LNI and channel shaping rate in bps format directly.
See API reference manual for details.
- Add new CEETM API to check the LNI/channel shaping enablement.
- Add new CEETM API to set the CQ's weight in the ratio format. The ratio to weight code conversion is done inside this new API
- Add new CEETM API to drain the Class Queue till empty, and call this API inside the function to release CQ so that there will be no frames left in CQ when it is released

Known Bugs, Limitations, or Technical Issues

- QMan and BMan portals are core affine, with each core assigned one QMan and one BMan portal. This assignment also assumes that interrupts associated with these portals are directed to the same cores. This is a fundamental assumption for QMan/BMan drivers. Users should not change the core affinity of portal interrupts for any reason as this would cause the portal to become non-functional and possibly cause a kernel crash.
- QMan enqueue command ring (EQCR) stashing is only supported on QMan rev ≥ 3.0 on T4240 and B4860, not on other QorIQ Pxxxx silicon. Therefore, run-to-completion software that is attempting an enqueue operation (ie. it is not providing any WAIT flag) should implement its own “back off” after an enqueue returns an EBUSY return code (a 1000-cycle back off is a recommended guide-line). Failure to do so can consume excessive memory bandwidth and reduce overall throughput supported by the system.

5.2.2.2 QMan BMan API Reference

5.2.2.2.1 About this document

This document describes drivers for the Queue Manager and Buffer Manager hardware blocks underlying the datapath architecture within the NXP QorIQ multicore SoC's (P4080, P3041, P5020). There are also explanations given to various aspects of the QMan and BMan hardware itself, insofar as this is essential knowledge in using these devices effectively. The driver descriptions cover some driver implementation details, usage details (loading and configuring), but the main emphasis is on the programming interfaces (APIs) exposed by these drivers.

5.2.2.2.1.1 Suggested Reading

1. *QorIQ P4080, P3041, P5020 Reference Manuals (P4080RM, P3041RM, P5020RM)*
2. Power.org Standard for Embedded Power Architecture Platform Requirements (ePAPR). power.org, 2008.

5.2.2.2.2 Introduction to the Queue Manager and the Buffer Manager

The Queue Manager (QMan) and Buffer Manager (BMan) devices each expose two interfaces to software control. One interface is the Configuration and Control Status Register map (CCSR), which provides global configuration of the device, registers related to global device errors, performance, statistics, debugging, etc. The other interface is the CoreNet interface, which provides a memory map with multiple “portals” located in separable sub-regions for independent/parallel run-time use of the devices.

5.2.2.2.2.1 O/S specifics

The software described in this document is targeted to the Linux kernel and Linux user-space (USDPAAs) system targets. However, only Linux supports operating as the controller for the devices, so all interfaces related to CCSR access are Linux-only. Also, remember platform-specific considerations when working with the interfaces described here. See [Operating system specifics](#) on page 170 for more details.

5.2.2.2.3 Buffer Manager

5.2.2.2.3.1 BMan Overview

5.2.2.2.3.1.1 Buffer Manager's Function

The QorIQ Buffer Manager (BMan) SoC block manages pools of buffers for use by software and hardware in the “Datapath” architecture. On the P4080, BMan maintains state for 64 “Buffer Pools,” which are typically used by hardware blocks for constructing output data for returning to software, where software can not (or does not wish to) pre-allocate an output descriptor. *For non-P4080 specifications, refer to the appropriate QorIQ SoC Reference Manual.*

In particular;

1. provides an efficient use of buffer resources because the output will only occupy as many buffers as required (whereas pre-allocation must provide for the worst-case scenario each time if it wishes to avoid truncation and information-loss),
2. software does not need to provision resources for every queued operation nor handle the complications of recycling unused output buffers, etc.,

3. the footprint for buffer resources for a variety of different flows (and even different guest operating systems) can be "pooled".

With respect to "buffers", BMan really acts as an allocator of any 48-bit tokens the user wishes - BMan does not interpret these tokens at all, it is only the software and hardware blocks that use BMan that may assume these to be memory addresses. In many cases, the BMan acquire and release interfaces are likely to be more efficient than software-managed allocators due to the proximity of BMan's corenet-based interfaces to each CPU and its on-board caching and pre-fetching of pool data. Possible examples include; a BMan-oriented page-allocator for operating system memory-management, a "frame queue" allocator to manage unused QMan frame queue descriptors (FQD), etc. In particular, the frame queue example provides a simple mechanism for sharing a range of frame-queue IDs across different partitions/operating systems in a virtualized environment without needing inter-partition communications in software.

5.2.2.3.1.2 BMan's interfaces

The BMan block has a CCSR register space and interrupt line associated with the block for global configuration and management, specifically;

- the private system memory range (invisible to software) needed by BMan,
- software and hardware depletion interrupt thresholds for each pool,
- device error handling uses the global interrupt line and the CCSR register space contains error-capture and error-status registers.

The BMan block also exposes a Corenet memory space for low-latency interaction by the multiple SoC cores, and this corenet region is divided into a geometry of "portals" to allow independent access to BMan functionality in a partitioned (and/or virtualized) environment. Each portal consists of one 16KB cache-enabled and one 4KB cache-inhibited sub-range of the Corenet region, as well as a per-portal interrupt line. There are a variety of possible reasons for using distinct portals;

- for partitioning between distinct guest operating systems,
- to dedicate a portal for each CPU to reduce locking and improve cache-affinity,
- to make distinct portal configurations available,
- to give certain applications their own portal rather than enforcing a mux/demux layer to share a portal between applications,
- [etc.]

Each portal presents the following BMan functionality;

- a "release command ring" (RCR), a pipelined mechanism for software to hardware commands that release buffers to BMan-managed buffer pools,
- a "management command" interface (MC), a low-latency command/response interface for acquiring buffers from buffer pools, and querying the status of all buffer pools,
- an interrupt line and associated status, disable, enable, and inhibit registers.

These portal interfaces will be described in more detail in their respective sections.

5.2.2.3.2 BMan configuration interface

The BMan configuration interface is an encapsulation of the BMan CCSR register space and the global/error interrupt line. Whereas BMan portals provide independent channels for accessing BMan functionality, the configuration interface represents the BMan device itself. The BMan configuration interface is presently limited to the device-tree node that represents it, with one exception: an API exists to set per-buffer-pool depletion thresholds. This API is only available in the linux control-plane - that is, a kernel compiled with BMan control support that has the BMan CCSR device-tree node present. In a hypervisor scenario, this implies that only the control-plane linux guest OS can set buffer pool depletion thresholds.

5.2.2.3.2.1 BMan Device-Tree Node

The BMan device tree node represents the BMan device and its CCSR configuration space. When a linux kernel has BMan control support compiled in, it will react to this device tree node by configuring and managing the BMan device. The device-tree node sits within the CCSR node ("soc") and is of the following form;

```
soc@fe000000 {
    [...]
    bman: bman@31a000 {
        compatible = "fsl,bman";
        reg = <0x31a000 0x1000>;
        fsl,liodn = <0x20>;
    };
    [...]
};
```

'compatible' and 'reg' are standard ePAPR properties.

5.2.2.3.2.1.1 Free Buffer Proxy Records

As previously mentioned, BMan buffer pools needn't be used only for managing memory buffers, but in fact can manage pools of arbitrary 48-bit token values, whatever those tokens might represent. This is possible because BMan never uses those token values as memory locations - all management of buffer pools is maintained in memory that is private to the BMan block. Specifically, BMan uses some internal memory together with a private range of contiguous system memory for backing store. The internal units of the backing store memory are called "free buffer proxy records" (FBPRs), each of which occupies a 64-byte cacheline of memory, and can hold 8 tokens.

The current driver implementation allows this memory resource to be specified via the 'fsl,bman-fbpr' device-tree property, or by resorting to a default allocation of contiguous memory early during kernel boot. The 'fsl,bman-fbpr' property specifies a 2-tuple of address and size, specifying the physical address range to assign to BMan. The example given configures 16MB for FBPR memory (262,144 buffer tokens). These elements are expressed as 64-bit values, so take two cells each;

```
fsl,fbpr = <0x0 0x20000000 0x0 0x01000000>;
```

If the hypervisor is in use, this address range is "guest physical". If the given memory range falls within the range used by the linux control-plane OS, it will attempt to reserve the range against use by the OS.

NOTE

For all BMan and QMan private memory resources, the alignment of the memory region must match its size.

5.2.2.3.2.1.2 Logical I/O Device Number (BMan)

Reads and writes to BMan's FBPR memory are subject to processing by the PAMU IO-MMU configuration of the SoC. In particular, BMan has an LIODN (Logical I/O Device Number) register setting that will be used by PAMU to authorize and possibly translate memory accesses. The bootloader (u-boot) will program BMan's LIODN register and it will add this value as the "fsl,liodn" property before passing it along to the booted software.

```
fsl,liodn = <0x20>;
```

This property is only used by the hypervisor, in order to ensure that any translation between guest physical and real physical memory for the linux guest OS is similarly applied to BMan transactions. If linux is booted natively (no hypervisor), then the PAMU is either left in bypass mode or it is configured without translation. In any case the LIODN is of little practical importance to the configuration or use of BMan driver software.

5.2.2.2.3.2.2 Buffer Pool Node

The BMan buffer pool device tree node represents one of a BMan device's buffer pools and its associated configuration. When a linux kernel has BMan control support compiled in, it will react to this device tree node by configuring and managing the BMan buffer pool, in particular the pool will be marked as reserved by the driver so that it is not available for dynamic assignment. The device-tree nodes usually sit within a BMan portals parent node ("bman-portals") and is of the following form;

```
bman-portals@f4000000 {  
  
    [...]  
  
    buffer-pool@0 {  
  
        compatible = "fsl,bpool";  
  
        fsl,bpid = <0x0>;  
  
        fsl,bpool-cfg = <0x0 0x100 0x0 0x1 0x0 0x100>;  
  
        fsl,bpool-thresholds = <0x8 0x20 0x0 0x0>;  
  
    };  
  
    [...]  
  
};
```

5.2.2.2.3.2.2.1 Buffer Pool ID

The BMan device in QorIQ P4080 supports 64 hardware managed buffer pools, so valid IDs range from 0 to 63. For non-P4080 specifications, refer to the appropriate QorIQ SoC Reference Manual. The above example configures buffer pool 0, which is used by the QMan driver as an inter-partition allocator of unused QMan Frame Queue IDs;

```
fsl,bpid = <0x0>;
```

Buffer pool nodes in the device-tree indicate that the corresponding buffer pool IDs are reserved, ie. that they are not to be used for ad-hoc allocation of unused pools.

5.2.2.2.3.2.2.2 Seeding Buffer Pools

It is also possible to have the control plane linux BMan driver seed the buffer pool with an arbitrary arithmetic sequence of values, using the "fsl,bpool-cfg" property. This property is a 3-tuple of 64-bit values (each taking 2 cells) defining the arithmetic sequence; the count, the increment, and the base.

```
fsl,bpool-cfg = <0x0 0x100 0x0 0x1 0x0 0x100>;
```

In this example, the QMan FQ allocator implemented using BMan buffer pool ID 0 is seeded with 256 FQIDs in the range [256...511].

5.2.2.3.2.2.3 Depletion Thresholds

Each of the 64 buffer pools has CCSR registers related to depletion-handling. A pool is considered "depleted" once the number of buffers in that pool crosses a "depletion-entry" threshold from above, and this ends when the number of buffers subsequently crosses a "depletion-exit" threshold from below (the depletion-exit threshold should be higher than the depletion-entry threshold).

Each pool maintains two independent depletion states - one for software use and another for hardware blocks. Hardware blocks (like CAAM, FMan, PME) use the hardware depletion state primarily for the purpose of implementing push back (e.g. by stalling input-processing, issuing "pause frames", etc). There is a depletion-entry and -exit threshold for each buffer pool related to this hardware depletion state. The software depletion state serves two possible purposes - one is to allow software to implement push back too. The other use of software depletion thresholds is to allow software to manage "replenishment" of buffer pools. It is software that seeds buffer pools with blocks of memory initially and if desired, it can also use this mechanism to selectively provide additional blocks at run-time during depletion.

```
fsl,bpool-thresholds = <0x8 0x20 0x0 0x0>;
```

Here, software depletion thresholds have been set for the buffer pool used for the FQ allocator, but hardware depletion thresholds are disabled (the pool is for software use only). The pool will enter depletion when it drops below 8 "buffers" (in this case, FQIDs), and exit depletion when it rises above 32.

5.2.2.3.2.3 BMan Portal Device-Tree Node

The BMan Corenet portal interface in QorIQ P4080 provides up to 10 distinct memory-mapped interfaces for use by software to interact efficiently with BMan functionality. Specifically, each portal provides the following sub-interfaces; RCR (Release Command Ring), MC (Management Command), and ISR (Interrupt Status Register). For non-P4080 specifications, refer to the appropriate QorIQ SoC Reference Manual.

The BMan driver determines the available corenet portals from the device tree. The portal nodes are at the physical address scope (unlike the device-tree node for the BMan device itself, which is within the "soc" physical address node that represents CCSR). These nodes indicate the physical address ranges of the cache-enabled and cache-inhibited sub-regions of the portal (respectively), and look something like the following;

```
bman-portal@0 {
    compatible = "fsl,bman-portal";
    reg = <0xe4000000 0x4000 0xe4100000 0x1000>;
    interrupts = <0x69 2>;
    interrupt-parent = <&mpic>;
    cell-index = <0x0>;
    cpu-handle = <&cpu3>;
};
```

The most note-worthy property is "cpu-handle", which is used to express an affinity/association between the given BMan portal and the CPU represented by the referenced device-tree node.

5.2.2.3.2.3.1 Portal Initialization (BMan)

The driver is informed of the BMan portals that are available to it via the device-tree passed to the system from the boot process. For those portals that aren't reserved for USDPAAs usage via the "fsl,usdpaa-portal" property, it will automatically create TLB entries to map the BMan portal corenet sub-regions as cpu-addressable and cache-inhibited or cache-enabled as appropriate.

The BMan driver will automatically associate initialised BMan portals with the CPU to which they are configured, only a one-per-CPU basis (if multiple portals are configured for the same CPU, only one is used). The purpose of this is to provide a canonical portal that software can use for whichever CPU it is running on, with the advantages of a cpu-affine interface being improved cache-locality and reduced locking. This requires that each CPU have at least one portal device-tree node dedicated to it using the "cpu-handle" property.

5.2.2.3.2.3.2 Portal sharing

If there are CPUs that have no affine portal associated with them (for example if most portals have been reserved for USDPAA use), then the driver will select the highest-index portal to be configured for “sharing” with the CPUs that have no affine portal, otherwise called “slave CPUs” in this document. In this mode of operation, a coarser locking scheme is used for the portal in order to properly synchronise use by more than one CPU.

One key point to understand with portal sharing is that hardware-instigated portal events will continue to be processed only by the CPU to which the portal is affine, they are not shared. One consequence of this is that slave CPUs can not use `*_irqsource_*`() APIs to alter the interrupt-vs-polling state of the portal, nor can they call `*_poll_*`() APIs to perform run-to-completion servicing of the portal. The sharing of the portal is only to allow software-instigated portal functionality to be available to slave CPUs, such as creating and manipulating objects, performing commands, etc.

5.2.2.2.4 BMan CoreNet portal APIs

The following sections describe interfaces provided by the BMan driver for manipulating portals, as defined in [BMan Portal Device-Tree Node](#) on page 105.

5.2.2.2.4.1 BMan High-Level Portal Interface

5.2.2.2.4.1.1 Overview (BMan)

The high-level portal interface provides management and encapsulation of a portal hardware interface. The operations performed on the portal are co-ordinated internally, hiding the user from the I/O semantics, and allowing multiple users/contexts to share portals without collaboration between them. This interface also provides an object representation for buffer pools, with optional assists for cases where the user wishes to track depletion entry and exit events.

This interface provides locking and arbitration of portal operations from multiple software contexts and/or threads (ie. the portal is shared). In cases where a resource is busy, the interface also gives callers the option of blocking/sleeping until the resource is available. In any case where sleeping is an option, the caller can also specify whether the sleep should be interruptible.

NOTE

Support for blocking/sleeping is limited to linux, it is not available on run-to-completion systems such as USDPAA.

5.2.2.2.4.1.2 Portal management (BMan)

The portal management API provides `bman_affine_cpus()`, which returns a mask that indicates which CPUs have auto-initialized portals associated with them. See [BMan Portal Device-Tree Node](#) on page 105. All other BMan API functions must be executed on CPUs contained within this mask, and any interactions they require with h/w will be performed on the corresponding portals.

```
/**
 * bman_affine_cpus - return a mask of cpus that have portal access
 */
const cpumask_t *bman_affine_cpus(void);
```

5.2.2.2.4.1.2.1 Modifying interrupt-driven portal duties (BMan)

Portals have various servicing duties they must perform in reaction to hardware events. The portal management API allows applications to control which of these duties/events are triggered by interrupt-handling versus those which are performed at the application's explicit request via `bman_poll()`. If portal-sharing is in effect (see [Portal sharing](#) on page 105), these APIs won't succeed when called from a slave CPU.

```
#define BM_PIRQ_RCRI      0x00000002    /* RCR Ring (below threshold) */
#define BM_PIRQ_BSCN     0x00000001    /* Buffer depletion State Change */
/**
 * bman_irqsource_get - return the portal work that is interrupt-driven
 *
 * Returns a bitmask of BM_PIRQ_**I processing sources that are currently
 * enabled for interrupt handling on the current cpu's affine portal. These
```

```

* sources will trigger the portal interrupt and the interrupt handler (or a
* tasklet/bottom-half it defers to) will perform the corresponding processing
* work. The bman_poll_***() functions will only process sources that are not in
* this bitmask. If the current CPU is sharing a portal hosted on another CPU,
* this always returns zero.
*/
u32 bman_irqsource_get(void);
/**
 * bman_irqsource_add - add processing sources to be interrupt-driven
 * @bits: bitmask of BM_PIRQ_**I processing sources
 *
 * Adds processing sources that should be interrupt-driven, (rather than
 * processed via bman_poll_***() functions). Returns zero for success, or
 * -EINVAL if the current CPU is sharing a portal hosted on another CPU.
 */
int bman_irqsource_add(u32 bits);
/**
 * bman_irqsource_remove - remove processing sources from being interrupt-driven
 * @bits: bitmask of BM_PIRQ_**I processing sources
 *
 * Removes processing sources from being interrupt-driven, so that they will
 * instead be processed via bman_poll_***() functions. Returns zero for success,
 * or -EINVAL if the current CPU is sharing a portal hosted on another CPU. */
int bman_irqsource_remove(u32 bits);

```

5.2.2.4.12.2 Processing non-interrupt-driven portal duties (BMan)

If portal-sharing is in effect (see [Portal sharing](#) on page 105), these APIs won't succeed when called from a slave CPU.

```

/**
 * bman_poll_slow - process anything that isn't interrupt-driven.
 *
 * This function does any portal processing that isn't interrupt-driven. NB,
 * unlike the legacy wrapper bman_poll(), this function will deterministically
 * check for the presence of portal processing work and do it, which implies
 * some latency even if there's nothing to do. The bman_poll() wrapper on the
 * other hand (like the qman_poll() wrapper) attenuates this by checking for
 * (and doing) portal processing infrequently. Ie. such that qman_poll() and
 * bman_poll() can be called from core-processing loops. Use bman_poll_slow()
 * when you yourself are deciding when to incur the overhead of processing. If
 * the current CPU is sharing a portal hosted on another CPU, this function will
 * return -EINVAL, otherwise returns zero for success.
 */
int bman_poll_slow(void);
/**
 * bman_poll - process anything that isn't interrupt-driven.
 *
 * Dispatcher logic on a cpu can use this to trigger any maintenance of the
 * affine portal. This function does whatever processing is not triggered by
 * interrupts. This is a legacy wrapper that can be used in core-processing
 * loops but mitigates the performance overhead of portal processing by
 * adaptively bypassing true portal processing most of the time. (Processing is
 * done once every 10 calls if the previous processing revealed that work needed
 * to be done, or once every 1000 calls if the previous processing revealed no
 * work needed doing.) If you wish to control this yourself, call
 * bman_poll_slow() instead, which always checks for portal processing work.
 */
void bman_poll(void);

```

5.2.2.2.4.1.2.3 Recovery support (BMan)

Note that the following functions require the BMan portal to have been initialized in "recovery mode", which is not possible with the current release. As such, these functions are for future use only (and documented here only because they're declared in the API header).

```
/**
 * bman_recovery_cleanup_bpid - in recovery mode, cleanup a buffer pool
 */
int bman_recovery_cleanup_bpid(u32 bpid);
/**
 * bman_recovery_exit - leave recovery mode
 */
int bman_recovery_exit(void);
```

5.2.2.2.4.1.2.4 Determining if the release ring is empty

```
/**
 * bman_rcr_is_empty - Determine if portal's RCR is empty
 *
 * For use in situations where a cpu-affine caller needs to determine when all
 * releases for the local portal have been processed by BMan but can't use the
 * BMAN_RELEASE_FLAG_WAIT_SYNC flag to do this from the final bman_release().
 * The function forces tracking of RCR consumption (which normally doesn't
 * happen until release processing needs to find space to put new release
 * commands), and returns zero if the ring still has unprocessed entries,
 * non-zero if it is empty.
 */
int bman_rcr_is_empty(void);
```

5.2.2.2.4.1.3 Pool Management

To work with BMan buffer pools, a pool object must be created. As explained in [Depletion State](#) on page 111, the pool may be created with the `BMAN_POOL_FLAG_DEPLETION` flag and corresponding depletion-entry/exit callbacks if the owner wishes to be notified of changes in the pool's depletion state. Creation of the pool object can also modify the pool's depletion entry and exit thresholds with the `BMAN_POOL_FLAG_THRESH` flag, so long as the `BMAN_POOL_FLAG_DYNAMIC_BPID` flag is specified (which will allocate an unreserved BPID) and when running in the control-plane (where reserved BPIDs are tracked). Depletion thresholds for reserved BPIDs can be set in the device-tree within the nodes that reserve them, so support for setting them in the API is not provided. The pool object can also maintain an internal buffer stockpile to optimize releases and acquires of buffers by specifying the `BMAN_POOL_FLAG_STOCKPILE` flag - actual releases to and acquires from h/w will only occur when the stockpile needs flushing or replenishing, ensuring that the interactions with hardware occur less often and are always optimized to release/acquire the maximum number of buffers at once. If a pool object is being freed and it has been configured to use stockpiling, a flush operation must be performed on the pool object. This will ensure that all buffers in the stockpile are flushed to h/w. The pool object can then be freed. The stockpiling option is recommended wherever possible. One implementation note is that applications will sometimes want to create multiple pool objects for the same BPID in order to have one for each CPU (for performance reasons) - this means that each pool object will have its own stockpile. As a consequence, to drain a buffer pool empty would require that all pool objects for that BPID be drained independently (whereas without stockpiling enabled, only one pool object needs to be drained).

```
struct bman_pool;
/* This callback type is used when handling pool depletion entry/exit. The
 * 'cb_ctx' value is the opaque value associated with the pool object in
 * bman_new_pool(). 'depleted' is non-zero on depletion-entry, and zero on
 * depletion-exit. */
typedef void (*bman_cb_depletion)(struct bman_portal *bm,
                                  struct bman_pool *pool, void *cb_ctx, int depleted);
/* Flags to bman_new_pool() */
```

```

#define BMAN_POOL_FLAG_NO_RELEASE    0x00000001 /* can't release to pool */
#define BMAN_POOL_FLAG_ONLY_RELEASE  0x00000002 /* can only release to pool */
#define BMAN_POOL_FLAG_DEPLETION     0x00000004 /* track depletion entry/exit */
#define BMAN_POOL_FLAG_DYNAMIC_BPID  0x00000008 /* (de)allocate bpid */
#define BMAN_POOL_FLAG_THRESH        0x00000010 /* set depletion thresholds */
#define BMAN_POOL_FLAG_STOCKPILE     0x00000020 /* stockpile to reduce hw ops */
/* This struct specifies parameters for a bman_pool object. */
struct bman_pool_params {
    /* index of the buffer pool to encapsulate (0-63), ignored if
     * BMAN_POOL_FLAG_DYNAMIC_BPID is set. */
    u32 bpid;
    /* bit-mask of BMAN_POOL_FLAG_*** options */
    u32 flags;
    /* depletion-entry/exit callback, if BMAN_POOL_FLAG_DEPLETION is set */
    bman_cb_depletion cb;
    /* opaque user value passed as a parameter to 'cb' */
    void *cb_ctx;
    /* depletion-entry/exit thresholds, if BMAN_POOL_FLAG_THRESH is set. NB:
     * this is only allowed if BMAN_POOL_FLAG_DYNAMIC_BPID is used *and*
     * when run in the control plane (which controls BMan CCSR). This array
     * matches the definition of bm_pool_set(). */
    u32 thresholds[4];
};
/**
 * bman_new_pool - Allocates a Buffer Pool object
 * @params: parameters specifying the buffer pool behavior
 *
 * Creates a pool object for the given @params. A portal and the depletion
 * callback field of @params are only used if the BMAN_POOL_FLAG_DEPLETION flag
 * is set. NB, the fields from @params are copied into the new pool object, so
 * the structure provided by the caller can be released or reused after the
 * function returns.
 */
struct bman_pool *bman_new_pool(const struct bman_pool_params *params);
/**
 * bman_free_pool - Deallocates a Buffer Pool object
 * @pool: the pool object to release
 */
void bman_free_pool(struct bman_pool *pool);
/**
 * bman_flush_stockpile - Flush stockpile buffer(s) to the buffer pool
 * @pool: the buffer pool object the stockpile belongs
 * @flags: bit-mask of BMAN_RELEASE_FLAG_*** options
 *
 * Adds stockpile buffers to RCR entries until the stockpile is empty.
 * The return value will be a negative error code if a h/w error occurred.
 * If BMAN_RELEASE_FLAG_NOW flag is passed and RCR ring is full,
 * -EAGAIN will be returned.
 */
int bman_flush_stockpile(struct bman_pool *pool, u32 flags);
/**
 * bman_get_params - Returns a pool object's parameters.
 * @pool: the pool object
 *
 * The returned pointer refers to state within the pool object so must not be
 * modified and can no longer be read once the pool object is destroyed.
 */
const struct bman_pool_params *bman_get_params(const struct bman_pool *pool);
/**
 * bman_query_free_buffers - Query how many free buffers are in buffer pool

```

```
* @pool: the buffer pool object to query
*
* Return the number of the free buffers
*/
u32 bman_query_free_buffers(struct bman_pool *pool);
/**
 * bman_update_pool_thresholds - Change the buffer pool's depletion thresholds
 * @pool: the buffer pool object to which the thresholds will be set
 * @thresholds: the new thresholds
 */
int bman_update_pool_thresholds(struct bman_pool *pool, const u32 *thresholds);
```

5.2.2.2.4.1.4 Releasing and Acquiring Buffers

The following API functions allow applications to release buffers to a pool and acquire buffers from a pool. Note that the various "WAIT" flags for `bman_release()` are only available on linux.

```
/* Flags to bman_release() */
#define BMAN_RELEASE_FLAG_WAIT      0x00000001 /* wait if RCR is full */
#define BMAN_RELEASE_FLAG_WAIT_INT  0x00000002 /* if we wait, interruptible? */
#define BMAN_RELEASE_FLAG_WAIT_SYNC 0x00000004 /* if wait, until consumed? */
/**
 * bman_release - Release buffer(s) to the buffer pool
 * @pool: the buffer pool object to release to
 * @bufs: an array of buffers to release
 * @num: the number of buffers in @bufs (1-8)
 * @flags: bit-mask of BMAN_RELEASE_FLAG_*** options
 *
 * Releases the specified buffers to the buffer pool. If stockpiling is
 * enabled, this may not require a release command to be issued via the RCR
 * ring, otherwise it certainly will. If the RCR ring is full, the function
 * will return -EBUSY unless BMAN_RELEASE_FLAG_WAIT is selected, in which case
 * it will sleep waiting for space to become available in RCR. If
 * BMAN_RELEASE_FLAG_WAIT_SYNC is also specified then it will sleep until
 * hardware has processed the command from the RCR (otherwise the same
 * information can be obtained by polling bman_rcr_is_empty() until it returns
 * TRUE). If the BMAN_RELEASE_FLAG_WAIT_INT is set, then any sleeps will be
 * interruptible. If it is interrupted before producing the release command, it
 * returns -EINTR. Otherwise, it will return zero to indicate the release was
 * successfully issued. (In the case of interruptible sleeps and WAIT_SYNC,
 * check signal_pending() upon return to determine whether the wait was
 * interrupted.)
 */
int bman_release(struct bman_pool *pool, const struct bm_buffer *bufs,
                u8 num, u32 flags);
/**
 * bman_acquire - Acquire buffer(s) from a buffer pool
 * @pool: the buffer pool object to acquire from
 * @bufs: array for storing the acquired buffers
 * @num: the number of buffers desired (@bufs is at least this big)
 *
 * Acquires buffers from the buffer pool. If stockpiling is enabled, this may
 * not require an acquire command to be issued via the MC interface, otherwise
 * it certainly will. The return value will be the number of buffers obtained
 * from the pool, or a negative error code if a h/w error or pool starvation
 * was encountered.
 */
```

```
int bman_acquire(struct bman_pool *pool, struct bm_buffer *bufs, u8 num,
                u32 flags);
```

5.2.2.2.4.1.5 Depletion State

It is possible for portals to track depletion state changes to any of the 64 buffer pools supported in BMan. As described in [Pool Management](#) on page 108, a pool object can invoke callbacks to convey depletion-entry and depletion-exit events if created with the `BMAN_POOL_FLAG_DEPLETION` flag.

Conversely, software can issue a portal management command to obtain a snapshot of the depletion and availability status of all BMan 64 pools at once, which is what the following interface does. Here "availability" implies that the pool is not completely empty. Depletion on the other hand is relative to the pools depletion-entry and exit-thresholds. The state of all 64 buffer pools is represented by the following structure types, accessor macros, and `bman_query_pools()` API;

```
struct bm_pool_state {
    [...]
};
/**
 * bman_query_pools - Query all buffer pool states
 * @state: storage for the queried availability and depletion states
 */
int bman_query_pools(struct bm_pool_state *state);
/* Determine the "availability state" of BPID 'p' from a query result 'r' */
#define BM_MCR_QUERY_AVAILABILITY(r,p) [...]
/* Determine the "depletion state" of BPID 'p' from a query result 'r' */
#define BM_MCR_QUERY_DEPLETION(r,p) [...]
```

5.2.2.2.5 Queue Manager

5.2.2.2.5.1 QMan Overview

5.2.2.2.5.1.1 Queue Manager's Function

The QorIQ Queue Manager (QMan) SoC block manages the movement of data ("frames") along uni-directional flows ("frame queues") between different software and hardware end-points ("portals"). This allows software instances to communicate with other software instances and/or datapath hardware blocks (CAAM, PME, FMan) using a hardware-managed queueing mechanism. QMan provides a variety of features in the way this data movement can be managed, including tail-drop or weighted-red congestion/flow-control, congestion group depletion notification, order restoration, and order preservation.

It is beyond the scope of this document to fully explain all the QMan-related notions that are essential to using datapath functionality effectively. But unlike the BMan reference, we will cover at least some of the basic elements here that are fundamental to the software interface, because QMan is more complicated than BMan and some simplistic definitions can be helpful as a place to start. For any more information about what QMan does and how it behaves, please consult the appropriate QorIQ SoC Reference Manual.

5.2.2.2.5.1.2 Frame Descriptors

Frames are represented by "frame descriptors" (or "FD"s) which are 16-byte structures consisting of fields to describe;

- contiguous or scatter-gather data,
- a 32-bit per-frame-descriptor token value (called "cmd/status" because of its common usage in processing data to/from hardware blocks),
- trace-debugging bits,
- a partition ID, used for virtualizing memory access to frame data by datapath hardware blocks (CAAM, PME, FMan),
- a BMan buffer pool ID, used to identify frames whose buffers are sourced from (or are to be recycled to) a BMan buffer pool.

A third ("nested") mode of the scatter-gather representation allows a frame-descriptor to reference more than one frame - this is referred to as a *compound frame*, and is a mechanism for creating an indissociable binding of more than one data descriptor, eg. this is used when sending an input descriptor to PME or CAAM and providing an output descriptor to go with it.

Frame descriptors that are under QMan's control reside in QMan-private resources, comprised of dedicated on-board cache as well as system memory assigned to QMan on initialization. When frames are enqueued to (and dequeued from) frame queues by QMan on behalf of software portals or hardware blocks, the frame descriptor fields are copied in to (and out of) these QMan-private resources.

As with BMan not caring whether the 48-bit tokens it manages are real buffer addresses or not, the same is mostly true for QMan with respect to the frame descriptors it manages. QMan ignores the memory addresses present in the frame descriptor, unless it is dequeued via a portal configured for data stashing and is dequeued from a frame queue that is configured for frame data (or annotation) stashing. However QMan always pays attention to the length field of frame descriptors. In general, the only field that can be safely used as a "pass-through" value without any QMan consequences is the 32-bit cmd/status field.

5.2.2.2.5.1.3 *Frame Queue Descriptors (QMan)*

Frame queues are uni-directional queues of frames, where frames are enqueued to the tail of the frame queue and dequeued from the head. A frame queue is represented in QMan by a "frame queue descriptor" (or "FQD"), and these reside in a private system memory resource configured for QMan on initialization. A frame queue is referred to by a "frame queue identifier" (or "FQID"), which is literally the index of that FQD within QMan's memory resource. As such, FQIDs form a global name-space, even in an otherwise virtualized environment, so two entities of software can not simultaneously use the same FQID for different purposes.

5.2.2.2.5.1.4 *Work Queues*

Work queues (or "WQ"s) are uni-directional queues of "scheduled" frame queues. We will see shortly what is meant here by a "scheduled" frame queue, but suffice it to say that QMan supports a fixed collection of work queues, to which QMan appends frame queues when they are due to be serviced. To summarize, multiple FDs can be linked to a single FQ, and multiple FQs can be linked to a single WQ.

5.2.2.2.5.1.5 *Channels*

A channel is a fixed, hardware-defined association of 8 work queues, also thought of as "priority work queues". This grouping is convenient in that QMan provides sophisticated prioritization support for dequeuing from entire channels rather than specific work queues. Specifically, the 8 work queues within a channel are divided into 3 tiers according to QMan's "class scheduler" logic - work queues 0 and 1 form the high-priority tier and are treated with a strict priority semantic, work queues 2, 3, and 4 form the medium-priority tier and are treated with a weighted interleaved round-robin semantic, and work queues 5, 6, and 7 form the low-priority tier and are also treated with a weighted interleaved round-robin semantic. Apart from the top-tier, the weighting within and between the other two tiers is programmable.

5.2.2.2.5.1.6 *Portals*

A QMan portal is similar in nature to a BMan portal. There are hardware portals (also called "direct connect portals", or "DCP"s) that allow QMan to be used by other hardware blocks, and there are software portals that allow QMan to be used by logically separated units of software. A software portal consists of two sub-regions of QMan's corenet region, in precisely the same way as with BMan.

5.2.2.2.5.1.7 *Dedicated Portal Channels*

Each software portal has its own dedicated channel (of 8 work queues), that only it may dequeue from. As a shorthand, one sometimes says that a frame queue is "scheduled to a portal", when what is really meant is that the frame queue is scheduled to a work queue within that portal's *dedicated channel*. Hardware portals also have their own dedicated channels, though sometimes more than one (FMan blocks have multiple dedicated channels).

5.2.2.2.5.1.8 Pool Channels

There are also 15 "pool channels" from which any software portal can dequeue - this is typically used for load-balancing or load-spreading.

5.2.2.2.5.1.9 Portal Sub-Interfaces

Each portal exposes cache-inhibited and cache-enabled registers that can be read and/or written by software to achieve various ends. With some necessary exceptions, the software interface hides most of these details. However an important conceptual point regarding portals is that they have essentially four decoupled sub-interfaces;

- EQCR (EnQueue Command Ring), this is an 8-cacheline ring containing commands from software to QMan. These commands perform enqueues of frame descriptors to frame queues.
- DQRR (DeQueue Response Ring), this is a 16-cacheline ring containing dequeue processing results from QMan to software. These entries usually contain a frame descriptor (except when the dequeue action produced no valid frame descriptor) as well as status information about the dequeue action, the frame queue being dequeued from, and other context for software's use. This ring is unique in that QMan can be configured to stash new ring entries to processor cache, rather than relying on software to (pre)fetch ring entries into cache explicitly.
- MR (Message Ring), this is an 8-cacheline ring containing messages from QMan to software, most notably for enqueue rejection messages and asynchronous retirement processing events. Unlike DQRR, this ring does not support stashing.
- Management commands, consisting of a Command Register (CR) and two Response Register locations (RR0 and RR1), used for issuing a variety of other commands to QMan. EQCR and DQRR (and to a lesser extent, MR) are intended to provide the communications with QMan that represent the fast-path of data processing logic, and the management command interface is where "everything else happens".

5.2.2.2.5.1.10 Frame queue dequeuing

Enqueuing a frame to a frame queue is an unambiguous mechanism; an enqueue command in the EQCR specifies a frame descriptor and a frame queue ID, and the intention is clear. Dequeuing is more subtle, and falls into two general classes depending on *what* one is dequeuing *from* - these are "scheduled" or "unscheduled" dequeues.

5.2.2.2.5.1.10.1 Unscheduled Dequeues

One can dequeue from a specific frame queue, but that frame queue must necessarily be "idle" - or in QMan terminology, "unscheduled". It is an illegal action to attempt to dequeue directly from a frame queue that is in a "scheduled" state. Specifically, unscheduled dequeues require the frame queue to be in the "Parked" or "Retired" state (described in [Frame Queue States](#) on page 115).

5.2.2.2.5.1.10.2 Scheduled Dequeues

Conversely, if a frame queue is "scheduled" then, by definition, management of the frame queue is (until further notice) under QMan's control and may at any point change state according to events within QMan or via actions on other software or hardware portals. So a "scheduled dequeue" does not target a specific FQ, but either a specific WQ or collection of channels. QMan processes scheduled dequeue commands within a portal by selecting from among the non-empty WQs, dequeuing a FQ from that selected WQ, and then dequeuing a FD from that FQ.

QMan portals implement two dequeue command modes, "push" and "pull";

5.2.2.2.5.1.10.3 Pull Mode

The "pull" mode is the less conventional of the two, as it is driven by software writing a dequeue command to a single cache-inhibited register that will, in response, perform a single instance of that command and publish its result to DQRR. This "pull" command (PDQCR - Pull DeQueue Command Register) could generate anywhere between 1 and 3 DQRR entries, and software must ensure that it does not write a new command to PDQCR until it knows at least one of these DQRR entries has been published (otherwise writing a new command could clobber the previous command before QMan has prepared its execution). The PDQCR command register can perform scheduled and unscheduled dequeues.

5.2.2.2.5.1.10.4 Push Mode

The "push" mode is the mode that gives software a familiar "DMA-style" interface, ie. where hardware performs work and fills in a kind of "Rx ring" autonomously. In the case of the QMan portal's DQRR sub-interface, this push mode is driven by two dequeue command registers, one for scheduled dequeues (SDQCR - Static DeQueue Command Register), and one for unscheduled dequeues (VDQCR - Volatile DeQueue Command Register). The reason for the static/volatile terminology (rather than scheduled/unscheduled), as well as the presence of two command registers instead of one, relates to how QMan schedules execution of the dequeue commands.

Unlike "pull" mode, QMan is not prodded by a write to the command register each time a dequeue command should occur, it must autonomously execute commands when appropriate. So it is clear that scheduled dequeues can only be performed when the targetted work queue or channels have Truly Scheduled frame queues available to dequeue from. Note that this is not an issue with "pull" mode, as a scheduled dequeue command can be issued when there are no available frame queues and QMan will simply publish a DQRR entry containing no frame descriptor to mark completion of the command - for "push" mode, this semantic cannot work. When in "push" mode, the QMan portal has a (possibly NULL) scheduled dequeue command for dequeuing from a selection of available channels. QMan executes this command only when there is matching scheduled dequeue work available on one of the channels - ie. the scheduled dequeue command (for channels) is *static*. If software writes SDQCR with a command to dequeue from a specific WQ, the command is executed only once (like the pull command), at which point it reverts to the static dequeue command for channels.

For unscheduled dequeues, a single Parked or Retired frame queue is identified for dequeuing, and as QMan does not manipulate the state of such frame queues in reaction to enqueue or dequeue activity (ie. there is no "scheduling"), there is no mechanism for QMan to "know" when this frame queue becomes non-empty some time in the future. So like "pull" mode, unscheduled dequeues must be done when explicitly demanded by software, and as such they must also (a) expire after a configurable number of frame descriptors are dequeued from frame queue or once it is empty, and (b) even if the frame queue is already empty, a DQRR entry with no frame descriptor should be used to notify software that the unscheduled dequeue command has expired. Ie. the unscheduled command "goes live" when written and becomes inactive once completed - it is *volatile*. Unlike "pull" mode however, the volatile command can perform more than a single dequeue action, and it can even block or flow-control while active, however it always runs to completion and then stops.

As "push" mode supports two dequeue commands (in fact one of them, SDQCR, encompasses two commands in its own right - it has a persistent channel-dequeue command, and an optional one-shot workqueue-dequeue command can be issued without clobbering it), it is worth pointing out that it can service both at once. The VDQCR command register contains a precedence option that QMan uses to determine whether SDQCR or VDQCR work be favoured in the situation where both are active.

5.2.2.2.5.1.10.5 Stashing to Processor Cache

When dequeuing frame queues and publishing entries in DQRR, QMan provides stashing features that involve repositioning data in processor cache. The main benefit of hardware-instigated stashing is that the data will already be in cache when the processor needs it, avoiding the need to explicitly prefetch it in advance or stalling the processor to fetch it on-demand. As we will see, there is another benefit in the specific case of DQRR stashing.

Each portal supports two types of stashing, for which distinct PAMU entries are configured.

DLIODN

The DLIODN setting configures PAMU authorization and/or translation of transactions to stash DQRR ring entries as they are produced by QMan. The stashing of DQRR entries is not just a performance tweak, it changes the way driver software operates the portal. Rather than needing to invalidate and prefetch the DQRR cachelines to see (or poll for) new DQRR entries, software can simply reread the cached version until it "magically changes". The stashing transaction is then the only implied traffic across the corenet bus (reducing bandwidth) and it is initiated by hardware at the first instant at which a software-initiated prefetch could have seen anything new (minimum possible latency).

Note that if the driver does not enable DQRR stashing, then it is a requirement to manipulate the processor cache directly, so its run time mode of operation must match device configuration. Note also that if DQRR stashing is used, software can not trust the DQRI interrupt source nor read PI index registers to determine that a new DQRR entry is available, as they may race against the stash transaction. On the other hand, software may use the interrupt source to avoid polling for DQRR production unnecessarily, but it does not guarantee that the first read would show the new DQRR entry.

NOTE

P1023 supports DQRR stashing but since it doesn't have Corenet and PAMU, the FLIODN is not applicable to P1023.

FLIODN

QMan can also stash per-frame-descriptor information, specifically;

1. Frame data, pointed to by the frame descriptor
2. Frame annotations, which is anything prior to the data due to a non-zero offset
3. Frame queue context (for the frame queue from which the frame descriptor was dequeued).

In all cases, the FLIODN setting is used by PAMU to authorize/translate these stashing transactions.

5.2.2.2.5.11 *Frame Queue States*

Frame queues are managed by QMan via state-transitions, and some of these states are of interest to software. From software's perspective, a simplification of the frame queue states is to group them as follows;

- **Out of service:** the frame queue is not in use and must be initialized. Neither enqueues nor dequeues are permitted.
- **Parked:** the frame queue is initialized and in an idle state. Enqueues are permitted, as are unscheduled dequeues, neither of which change the frame queue's state. Scheduled dequeues will not result in dequeues from parked frame queues, as a parked frame queue is never linked to a work queue.
- **Scheduled:** the frame queue has been scheduled, implying that hardware will modify its state as/when relevant events occur. Enqueues are permitted, but unscheduled dequeues are not. This is not a real state, but actually a set of states that a frame queue moves between - as hardware performs these moves internally, it's useful to treat them as one, because changes between them are asynchronous to software. The real states are;
 - **Tentatively Scheduled:** the frame queue is not linked to a work queue (yet), the frame queue must therefore be empty and no retirement or force-eligible command has been issued against the frame queue.
 - **Truly Scheduled:** the frame queue is linked to a work queue, either because it has become non-empty or a force-eligible command has occurred.
 - **Active:** the frame queue has been selected by a portal for scheduled dequeue and so is removed from the work queue.
 - **Held Active:** the frame queue is still held by the portal after scheduled dequeuing has been performed, it may yet be dequeued from again, depending on scheduling configuration, priorities, etc.
 - **Held Suspended:** the frame queue is still held by the portal after scheduled dequeuing has been performed but another frame queue has been selected "active" and so no further dequeuing will occur on this frame queue.
- **Retired:** the frame queue is being "closed". A frame queue can be put into the retired state as a means of (a) getting it back under software's control (not under QMan's control nor the control of another hardware block), eg. for closing down "Tx" frame queues, and (b) blocking further enqueues to the frame queue so that it can be drained to empty in a deterministic manner. Enqueues are therefore not permitted in this state. Unscheduled dequeues are permitted, and are the only way to dequeue frames from a frame queue in this state.

See the appropriate QorIQ SoC Reference Manual for more detailed information.

5.2.2.2.5.12 *Hold active*

The QMan portal sub-interfaces are generally decoupled or asynchronous in their operation. For example: The processing of software-produced enqueue commands in EQCR is asynchronous to the processing of dequeue commands into DQRR, and both of these are asynchronous to the production of messages into MR and the processing of management commands.

There is however a specific coupling mechanism between EQCR and DQRR to address a certain class of requirements for datapath processing. Consider first that it is possible for multiple portals to dequeue independently from the same data source,

eg. for the purposes of load-balancing, or perhaps idle-time processing of low-priority work. This could occur because multiple portals issue unscheduled dequeue commands from the same Parked (or Retired) frame queue, or because they issue scheduled dequeue commands that target the same pool channels (or the same specific work queue within a pool channel). So we describe here the "hold active" mechanisms that help maintain some synchronicity of hardware dequeue processing (and optionally software *post*-processing) on multiple portals/CPUs.

The unscheduled dequeue case is not covered by the mechanisms described here - QMan will correctly handle multiple unscheduled dequeues from the same frame queue, but the "hold active" mechanisms have no effect in this case. For scheduled dequeues however, there are two levels of "hold active" functionality that can be used for software to synchronise multiple portals dequeuing from the same source.

5.2.2.2.5.1.12.1 Dequeue Atomicity

As described in the previous section ("Frame queue states"), the Active, Held Active, and Held Suspended states are for frame queues that have been selected by a portal for *scheduled* dequeuing. These states imply that the frame queue has been detached from the work queue that it was previously "scheduled" to, but not yet moved to the Parked state nor rescheduled to the Tentatively Scheduled or Truly Scheduled state after the completion of dequeuing.

Normally, a frame queue is rescheduled by QMan as soon as it is done dequeuing, potentially even before the resulting DQRR entries are visible to software. However, if the frame queue has been configured for "Held active" behavior, then this will not happen - the frame queue will remain in the Held Active or Held Suspended state once QMan has finished dequeuing from it. QMan will only reschedule or park the frame queue once software consumes all DQRR entries that correspond to that frame queue - the default behavior is to reschedule, but this "held" state of the frame queue allows software an opportunity to request that the final action for the frame queue be to park it instead.

A consequence of this mechanism is that if a DQRR entry is seen that corresponds to a frame queue configured for "held active" behavior, software implicitly knows that there can be no other (unconsumed) DQRR entry on any other portal for that same frame queue. (Proof: if there was, the frame queue would be currently "held" in that portal and not in this one.) For an SMP system where each core has its own portal, this would obviate the need to (spin)lock software context related to a frame queue when handling incoming frames - the "lock" is implicitly obtained when the DQRR entry is seen, and it is implicitly released when the DQRR entries are consumed. This is what is meant by "dequeue atomicity".

5.2.2.2.5.1.12.2 Parking Scheduled FQs

As noted above in [Dequeue Atomicity](#) on page 116, if a FQ is currently "held active" in the portal, software can request that it be move to the Parked state once its final DQRR entry is consumed, rather than rescheduled which is the normal behavior. As we will also see in [Force Eligible](#) on page 117, this is not necessarily limited to FQs that are configured for "hold active" behavior, but can also be applied to regular FQs by issuing a Force Eligible command on them.

5.2.2.2.5.1.12.3 Order Preservation & Discrete Consumption Acknowledgement

In addition to the dequeue atomicity feature, it is possible to obtain a stronger property from QMan to aid with datapath situations that "spread" incoming data over multiple portals. Specifically, if incoming frames are to be forwarded via subsequent enqueues, then dequeue atomicity does not prevent the forwarded frames from getting out of order. Ie. multiple CPUs (using multiple portals) may be using dequeue atomicity in order to write enqueue commands to their EQCR rings before consuming the DQRR entries, and thus ensuring that EQCR entries are *published* in the same order as the incoming frames. But as there are multiple portals, this does not ensure that QMan will necessarily *process* those EQCR entries in the same order. Indeed if the portals' EQCR rings have significantly varied fill-levels, then there is a reasonable chance that two enqueue commands published in quick succession via different portals could get processed in the opposite order by QMan.

Instead, software can elect to only consume DQRR entries when no forwarding is to be performed on the corresponding frames (eg. when dropping a packet), and for the others, it can encode the EQCR enqueue commands to perform an implicit "Discrete Consumption Acknowledgement" (or "DCA") - the result of which is that QMan will consume the corresponding DQRR entry on software's behalf *once it has finished processing the enqueue command*. This provides a cross-portal, order preservation semantic from end-to-end (from dequeue to enqueue) using hardware assists.

Note, QMan has other functionality called Order Restoration that is completely unrelated to the above - Order Restoration is a mechanism to restore frames into their intended order once they been allowed to get out of order, using sequence numbers

and "reassembly windows" within QMan, see [Order Restoration](#) on page 117. The above "hold active" mechanisms are to prevent frames from getting out of order in the first place.

5.2.2.2.5.1.13 Force Eligible

QMan portals support a management command called "Force Eligible" which allows software to regain control of a scheduled frame queue, usually with the intention to park it. When a frame queue is scheduled, QMan is responsible for its state and software can not meaningfully query it, as any snapshots are implicitly out of date by the time software sees them. Software only knows the frame queue state once QMan-generated events indicate that the frame queue is "quiesced" somehow. Moreover, if the frame queue is not configured for "hold active" behavior, then even the presence of DQRR entries does not help in this regard, as the portal may well have rescheduled the frame queue before software sees the first DQRR entry.

When QMan processes a Force Eligible command, it does two things - it tags the frame queue descriptor with a flag that is visible in subsequent DQRR entries, and, if the frame queue is Tentatively Scheduled (because it is empty), it will move the frame queue to the Truly Scheduled state (linked to a work queue). The result is that the frame queue "will receive dequeue processing soon", whether that was already happening or not. Fundamentally, when QMan is dequeuing from the FQ a short while later, it will treat the frame queue as "hold active", even if it isn't configured for hold active treatment. As such, software can request that the FQ be parked rather than rescheduled once the DQRR entry is consumed. See [Parking Scheduled FQs](#) on page 116.

5.2.2.2.5.1.14 Enqueue Rejections

Enqueues may be rejected, immediately or after any delay due to order restoration, and the enqueue mechanisms themselves do not provide any meaningful way to convey the rejection event to the software portal. For this reason, Enqueue Rejection Notifications (ERNs) are messages received on a message ring that carry frames that did not successfully enqueue together with the reason for their rejection.

5.2.2.2.5.1.15 Order Restoration

Frame queue descriptors can serve one or both of two complimentary purposes. A small subset of fields in the FQDs are used to implement an "Order Restoration Point", which allows an FQD to act as a reassembly window for out-of-sequence enqueues. FQDs also contain a sequence number field that generates increasing sequence numbers for all frames dequeued from the FQ. This dequeue activity sequence number is also called an "Order Definition Point". The idea is that frames dequeued from a given FQ (ODP) may get out-of-sequence during processing before they're enqueued onto an egress FQ, so the enqueue function allows one to not only specify the destination FQD, but also an ORP that the enqueue command should first pass through - which might hold up the intended enqueue until other, missing, sequence elements are enqueued. I.e. an ORP-enabled enqueue command requires 2 FQID parameters, which need not necessarily be the same - indeed in many networking examples, the Rx FQ serves as both the ODP and the ORP when enqueueing to the Tx FQ. To see why this choice of ORP FQ makes sense, consider that many Rx flows may need to be order-restored independently, even if all of them are ultimately enqueued to the same destination Tx FQ. It's also possible to enqueue using software-generated sequence numbers, i.e. without any FQ dequeue activity acting as an ODP. An ODP is any source of sequence numbers starting at zero and wrapping to zero at 0x3fff ($2^{14}-1$).

ORP-enabled enqueue functions provide various features, such as filling in missing sequence numbers (eg. when dropping frames), advancing the "Next Expected Sequence Number" despite missing frames (that may or may not show up later), etc. These features are options in the enqueue interfaces, eg. see [Enqueue Command \(without ORP\)](#) on page 129, specifically the `qman_enqueue_orp()` API.

There are also numerous options that can be set in ORP-enabled FQDs, and these are achieved via the same functions that allow you to manipulate FQDs for any other purpose. Eg. see [Frame queue management](#) on page 124, specifically the `qman_init_fq()` API. Care should be taken when using a FQD as both a FQ and an ORP - in particular, a FQD can not be retired and put out-of-service while the ORP component of the descriptor is still in use, and vice versa.

5.2.2.2.5.2 QMan configuration interface

The QMan configuration interface is an encapsulation of the QMan CCSR register space and the global/error interrupt source. Whereas QMan portals provide independent channels for accessing QMan functionality, the configuration interface

represents the QMan device itself. The QMan configuration interface is presently limited to the device-tree node that represents it.

5.2.2.2.5.2.1 QMan device-tree node

The QMan device tree node represents the QMan device and its CCSR configuration space (as distinct from its corenet portals). When a linux kernel has QMan control support built in, it will react to this device tree node by configuring and managing the QMan device. The device-tree node sits within the CCSR node ("soc") and is of the following form;

```
soc@fe000000 {
    [...]
    qman: qman@318000 {
        compatible = "fsl,qman";
        reg = <0x318000 0x1000>;
        fsl,qman-fqd = <0x0 0x22000000 0x0 0x00200000>;
        fsl,qman-pfdr = <0x0 0x21000000 0x0 0x01000000>;
        fsl,liodn = <0x1f>;
    };
    [...]
};
```

'compatible' and 'reg' are standard ePAPR properties.

5.2.2.2.5.2.1.1 Frame Queue Descriptors

This property configures the memory used by QMan for storing frame queue descriptors. Each FQD occupies a 64-byte cacheline of memory, so as the above example configures 2MB for FQD memory, the valid range of FQIDs is [1...32767];

```
fsl,qman-fqd = <0x0 0x22000000 0x0 0x00200000>;
```

The treatment and alignment requirements of this property are the same as in [Free Buffer Proxy Records](#) on page 103.

5.2.2.2.5.2.1.2 Packed Frame Descriptor Records

This property configures the memory used by QMan for storing Packed Frame Descriptor Records. Each PFDR occupies a 64-byte cacheline of memory, and can hold 3 Frame Descriptors. QMan maintains an onboard cache for holding recently enqueued (and/or soon to be dequeued) frames, and in responsive systems that remain within their operating capacity (ie. no spikes) it can often be unnecessary for frames to ever be stored in system memory at all. However, to handle spikes or buffering, a storage density of 3 enqueued frames per-cacheline can be used for estimating a suitable allocation of memory to QMan for PFDRs. In the case of handling ERNs (eg. if congestion controls exist elsewhere than on an ingress network interface), then a storage density of 1 ERN per-cacheline should be used. The above example configures 16MB for PFDR memory (786,432 enqueued frames, or 262,144 ERNs);

```
fsl,qman-pfdr = <0x0 0x21000000 0x0 0x01000000>;
```

The treatment and alignment requirements of this property are the same as in [Free Buffer Proxy Records](#) on page 103.

5.2.2.2.5.2.1.3 Logical I/O Device Number (QMan)

This property is the same as described in [Logical I/O Device Number \(BMan\)](#) on page 103, but for use by QMan when accessing FQD and PFDR memory (rather than BMan's FBPR memory).

5.2.2.2.5.2.2 QMan pool channel device-tree node

Each QMan software portal has its own dedicated channel of work queues. QMan also provides "pool channels" that all software portals can optionally dequeue from - this is described in [Portals](#) on page 112. The device-tree should declare pool channels using device-tree nodes as follows;

```
qman-pool@1 {
    compatible = "fsl,qman-pool-channel";
    cell-index = <0x1>;
};
```

```

        fsl,qman-channel-id = <0x21>;
    };

```

5.2.2.5.2.2.1 Channel ID

When FQs are initialized for scheduling, the target work queue is identified by the channel id (a hardware-assigned identifier) and by one of the 8 priority levels within that channel. Channel ids are hardware constants, as conveyed by this device-tree property;

```

fsl,qman-channel-id = <0x21>;

```

5.2.2.5.2.3 QMan portal device-tree node

The QMan Corenet portal interface in QorIQ P4080 provides up to 10 distinct memory-mapped interfaces for use by software to interact efficiently with QMan functionality. These are described in [Portals](#) on page 112 and [Portal Sub-Interfaces](#) on page 113. Refer to the appropriate SoC reference manuals for non-P4080 specifications.

The QMan driver determines the available corenet portals from the device tree. The portal nodes are at the physical address scope (unlike the device-tree node for the BMan device itself, which is within the "soc" physical address node that represents CCSR). These nodes indicate the physical address ranges of the cache-enabled and cache-inhibited sub-regions of the portal (respectively), and look something like the following;

```

qman-portal@c000 {
    compatible = "fsl,qman-portal";
    reg = <0xf420c000 0x4000 0xf4303000 0x1000>;
    interrupts = <0x6e 2>;
    interrupt-parent = <&mpic>;
    cell-index = <0x3>;
    cpu-handle = <&cpu3>;
    fsl,qman-channel-id = <0x3>;
    fsl,qman-pool-channels = <&qpool1 &qpool2>;
    fsl,liodn = <0x7 0x8>;
};

```

As with BMan portal nodes, the "cpu-handle" property is used to express an affinity/association between the given QMan portal and the CPU represented by the referenced device-tree node. Unlike BMan however, the "cpu-handle" property is also used by PAMU configuration, to determine which CPU's L1 or L2 cache should receive stashing transactions emanating from this portal. The "fsl,qman-channel-id" property is already documented in [Channel ID](#) on page 119, the other QMan-specific portal properties are described below.

5.2.2.5.2.3.1 Portal Access to Pool Channels

In QorIQ P4080, P3041, P5020 hardware, all software portals can dequeue from any/all pool channels. Nonetheless, the portal device-tree nodes allow the architect to specify this and optionally limit the range of pool channels a given portal can dequeue from. This can be particularly useful when partitioning multiple guest operating systems, it essentially allows the architect to partition the use of pool channels as they partition the use of portals. In the above example, the portal is only able to dequeue from 2 pool channels;

```

fsl,qman-pool-channels = <&qpool1 &qpool2>;

```

5.2.2.5.2.3.2 Stashing Logical I/O Device Number

This property, when used in QMan portal nodes, declares two LIODN values for use by QMan when performing dequeue stashing to processor cache. These are documented in [Stashing to Processor Cache](#) on page 114. This property is filled in automatically by u-boot, and if hypervisor is in use then it will fill in this property for guest device-trees also. PAMU drivers

(linux-native or within the hypervisor) will configure the settings for these LIODNs according to the CPU that stashing should be directed towards, as per the `cpu-handle` property;

```
fsl,liodn = <0x7 0x8>;  
cpu-handle = <&cpu3>;
```

5.2.2.2.5.2.3.3 Portal Initialization (QMan)

The driver is informed of the QMan portals that are available to it via the device-tree passed to the system from the boot process. For those portals that aren't reserved for USDPAA usage via the "fsl,usdpaa-portal" property, it will automatically create TLB entries to map the QMan portal corenet sub-regions as `cpu-addressable` and `cache-inhibited` or `cache-enabled` as appropriate.

As with the BMan driver, the QMan driver will automatically associate initialised QMan portals with the CPU to which they are configured, only one a one-per-CPU basis (if multiple portals are configured for the same CPU, only one is used). Please see [Portal sharing](#) on page 105 for an explanation of this behaviour in the BMan documentation, the QMan behaviour is identical.

5.2.2.2.5.2.3.4 Auto-Initialization

As with the BMan driver, the QMan driver will, by default, automatically initialize QMan portals as they are parsed out of the device-tree. Please see [Portal sharing](#) on page 105 for an explanation of this behavior in the BMan documentation. The QMan behavior is identical.

5.2.2.2.6 QMan portal APIs

The following sections describe interfaces provided by the QMan driver for manipulating portals. These are defined in [QMan portal device-tree node](#) on page 119, and described in [Portals](#) on page 112 and [Portal Sub-Interfaces](#) on page 113.

Note, unlike the BMan documentation, we will not include many of the QMan-related data structures within this documentation as they are significantly more elaborate. It is presumed the reader will consult the corresponding header files for structure data details that aren't sufficiently described here.

5.2.2.2.6.1 QMan High-Level Portal Interface

5.2.2.2.6.1.1 Overview (QMan)

The high-level portal interface provides management and encapsulation of a portal hardware interface. The operations performed on the "portal" are coordinated internally, hiding the user from the I/O semantics, and allowing multiple users/contexts to share portals without collaboration between them. This interface also provides an object representation for congestion group records (CGRs), with optional assists for cases where the user wishes to track congestion entry and exit events, eg. to apply back-pressure on the affected frame queues, etc. There is also an object representation for frame queues that internally coordinates FQ operations, demuxes incoming dequeued frames and messages to the corresponding owner's callbacks, and interprets hardware-provided indications of changes to FQ state.

This interface provides locking and arbitration of portal operations from multiple software contexts and/or threads (ie. the portal is shared). In cases where a resource is busy, the interface also gives callers the option of blocking/sleeping until the resource is available (and in the case of volatile dequeue commands, the caller may also optionally sleep until the volatile dequeue command has finished). In any case where sleeping is an option, the caller can also specify whether the sleep should be interruptible.

NOTE

Support for blocking/sleeping is limited to Linux, it is not available on run-to-completion systems such as USDPAA.

The demux logic within the portal interface assumes ownership of the "contextB" field of frame queue descriptors (FQDs), so users of this interface can not modify this field. However, callers provide the cache line of memory to be used within the driver for each FQ object when calling `qman_create_fq()`, so they can extend this structure into adjacent cachelines with their own data and use this instead of contextB for their own purposes. Ie. when callbacks are invoked because of dequeued

frames, enqueue rejections, or retirement notifications, those callbacks will find their custom per-FQ data adjacent to the FQ object pointer they are passed. Moreover, if context-stashing is enabled for the portal and the FQD is configured to stash 1 or more cachelines of context, the QMan driver's demux function will be implicitly accelerated because the FQ object will be prefetched into processor cache. Any adjacent data that is covered by the FQ's stashing configuration could likewise lead to acceleration of the owner's dequeue callbacks, ie. by reducing or eliminating cache misses in fast-path processing.

5.2.2.2.6.1.2 Frame and Message Handling

When DQRR or MR ring entries are produced by hardware to software, callbacks that have been provided by the API user are invoked to allow those entries to be handled prior to the driver consuming them. These callbacks are provided in the 'qman_fq_cb' structure type.

```
struct qman_fq_cb {
    qman_cb_dqrr dqrr; /* for dequeued frames */
    qman_cb_mr ern;    /* for software ERNs */
    qman_cb_mr dc_ern; /* for diverted hardware ERNs */
    qman_cb_mr fqr;    /* retirement messages */
};
typedef enum qman_cb_dqrr_result (*qman_cb_dqrr)(struct qman_portal *qm,
                                                struct qman_fq *fq, const struct qm_dqrr_entry *dqrr);
typedef void (*qman_cb_mr)(struct qman_portal *qm, struct qman_fq *fq,
                          const struct qm_mr_entry *msg);
enum qman_cb_dqrr_result {
    /* DQRR entry can be consumed */
    qman_cb_dqrr_consume,
    /* Like _consume, but requests parking - FQ must be held-active */
    qman_cb_dqrr_park,
    /* Does not consume, for DCA mode only. This allows out-of-order
     * consumes by explicit calls to qman_dca() and/or the use of implicit
     * DCA via EQCR entries. */
    qman_cb_dqrr_defer
};
```

5.2.2.2.6.1.3 Portal management (QMan)

The portal management API provides `qman_affine_cpus()`, which returns a mask that indicates which CPUs have auto-initialized portals associated with them. See [QMan portal device-tree node](#) on page 119. All other QMan API functions must be executed on CPUs contained within this mask, and any interactions they require with h/w will be performed on the corresponding portals.

```
/**
 * qman_affine_cpus - return a mask of cpus that have portal access
 */
const cpumask_t *qman_affine_cpus(void);
```

5.2.2.2.6.1.3.1 Modifying interrupt-driven portal duties (QMan)

Portals have various servicing duties they must perform in reaction to hardware events. The portal management API allows applications to control which of these duties/events are triggered by interrupt-handling versus those which are performed at the application's explicit request via `qman_poll()` (or more specifically, via `qman_poll_dqrr()` and `qman_poll_slow()`). If portal-sharing is in effect (see [Portal sharing](#) on page 105), these APIs won't succeed when called from a slave CPU.

```
#define QM_PIRQ_CSCI    0x00100000    /* Congestion State Change */
#define QM_PIRQ_EQCI    0x00080000    /* Enqueue Command Committed */
#define QM_PIRQ_EQRI    0x00040000    /* EQCR Ring (below threshold) */
#define QM_PIRQ_DQRI    0x00020000    /* DQRR Ring (non-empty) */
#define QM_PIRQ_MRI    0x00010000    /* MR Ring (non-empty) */
#define QM_PIRQ_SLOW    (QM_PIRQ_CSCI | QM_PIRQ_EQCI | QM_PIRQ_EQRI | \
                        QM_PIRQ_MRI)
```

```
/**
 * qman_irqsource_get - return the portal work that is interrupt-driven
 *
 * Returns a bitmask of QM_PIRQ_**I processing sources that are currently
 * enabled for interrupt handling on the current cpu's affine portal. These
 * sources will trigger the portal interrupt and the interrupt handler (or a
 * tasklet/bottom-half it defers to) will perform the corresponding processing
 * work. The qman_poll_***() functions will only process sources that are not in
 * this bitmask. If the current CPU is sharing a portal hosted on another CPU,
 * this always returns zero.
 */
u32 qman_irqsource_get(void);
/**
 * qman_irqsource_add - add processing sources to be interrupt-driven
 * @bits: bitmask of QM_PIRQ_**I processing sources
 *
 * Adds processing sources that should be interrupt-driven (rather than
 * processed via qman_poll_***() functions). Returns zero for success, or
 * -EINVAL if the current CPU is sharing a portal hosted on another CPU.
 */
int qman_irqsource_add(u32 bits);
/**
 * qman_irqsource_remove - remove processing sources from being interrupt-driven
 * @bits: bitmask of QM_PIRQ_**I processing sources
 *
 * Removes processing sources from being interrupt-driven, so that they will
 * instead be processed via qman_poll_***() functions. Returns zero for success,
 * or -EINVAL if the current CPU is sharing a portal hosted on another CPU.
 */
int qman_irqsource_remove(u32 bits);
```

5.2.2.6.1.3.2 Processing non-interrupt-driven portal duties (QMan)

If portal-sharing is in effect (see [Portal sharing](#) on page 105), these APIs won't succeed when called from a slave CPU.

```
/**
 * qman_poll_dqrr - process DQRR (fast-path) entries
 * @limit: the maximum number of DQRR entries to process
 *
 * Use of this function requires that DQRR processing not be interrupt-driven.
 * Ie. the value returned by qman_irqsource_get() should not include
 * QM_PIRQ_DQRI. If the current CPU is sharing a portal hosted on another CPU,
 * this function will return -EINVAL, otherwise the return value is >=0 and
 * represents the number of DQRR entries processed.
 */
int qman_poll_dqrr(unsigned int limit);
/**
QMan Portal APIs
QMan, BMan API RM, Rev. 0.13
6-34 NXP Confidential Proprietary NXP Semiconductors
Preliminary—Subject to Change Without Notice
 * qman_poll_slow - process anything (except DQRR) that isn't interrupt-driven.
 *
 * This function does any portal processing that isn't interrupt-driven. If the
 * current CPU is sharing a portal hosted on another CPU, this function will
 * return -EINVAL, otherwise returns zero for success.
 */
void qman_poll_slow(void);
/**
```

```

* qman_poll - legacy wrapper for qman_poll_dqrr() and qman_poll_slow()
*
* Dispatcher logic on a cpu can use this to trigger any maintenance of the
* affine portal. There are two classes of portal processing in question;
* fast-path (which involves demuxing dequeue ring (DQRR) entries and tracking
* enqueue ring (EQCR) consumption), and slow-path (which involves EQCR
* thresholds, congestion state changes, etc). This function does whatever
* processing is not triggered by interrupts.
*
* Note, if DQRR and some slow-path processing are poll-driven (rather than
* interrupt-driven) then this function uses a heuristic to determine how often
* to run slow-path processing - as slow-path processing introduces at least a
* minimum latency each time it is run, whereas fast-path (DQRR) processing is
* close to zero-cost if there is no work to be done. Applications can tune this
* behavior themselves by using qman_poll_dqrr() and qman_poll_slow() directly
* rather than going via this wrapper.
*/
void qman_poll(void);

```

5.2.2.6.1.3.3 Recovery support (QMan)

Note that the following functions require the QMan portal to have been initialized in "recovery mode", which is not possible with the current release. As such, these functions are for future use only (and documented here only because they're declared in the API header).

```

/**
 * qman_recovery_cleanup_fq - in recovery mode, cleanup a FQ of unknown state
 */
int qman_recovery_cleanup_fq(u32 fqid);
/**
 * qman_recovery_exit - leave recovery mode
 */
int qman_recovery_exit(void);

```

5.2.2.6.1.3.4 Stopping and restarting dequeues to the portal

```

/**
 * qman_stop_dequeues - Stop h/w dequeuing to the s/w portal
 *
 * Disables DQRR processing of the portal. This is reference-counted, so
 * qman_start_dequeues() must be called as many times as qman_stop_dequeues() to
 * truly re-enable dequeuing.
 */
void qman_stop_dequeues(void);
/**
 * qman_start_dequeues - (Re)start h/w dequeuing to the s/w portal
 *
 * Enables DQRR processing of the portal. This is reference-counted, so
 * qman_start_dequeues() must be called as many times as qman_stop_dequeues() to
 * truly re-enable dequeuing.
 */
void qman_start_dequeues(void);

```

5.2.2.6.1.3.5 Manipulating the portal static dequeue command

```

/**
 * qman_static_dequeue_add - Add pool channels to the portal SDQCR

```

```
* @pools: bit-mask of pool channels, using QM_SDQCR_CHANNELS_POOL(n)
*
* Adds a set of pool channels to the portal's static dequeue command register
* (SDQCR). The requested pools are limited to those the portal has dequeue
* access to.
*/
void qman_static_dequeue_add(u32 pools);
/**
 * qman_static_dequeue_del - Remove pool channels from the portal SDQCR
 * @pools: bit-mask of pool channels, using QM_SDQCR_CHANNELS_POOL(n)
 *
 * Removes a set of pool channels from the portal's static dequeue command
 * register (SDQCR). The requested pools are limited to those the portal has
 * dequeue access to.
 */
void qman_static_dequeue_del(u32 pools);
/**
 * qman_static_dequeue_get - return the portal's current SDQCR
 *
 * Returns the portal's current static dequeue command register (SDQCR). The
 * entire register is returned, so if only the currently-enabled pool channels
 * are desired, mask the return value with QM_SDQCR_CHANNELS_POOL_MASK.
 */
u32 qman_static_dequeue_get(void);
```

5.2.2.2.6.1.3.6 Determining if the enqueue ring is empty

```
/**
 * qman_eqcr_is_empty - Determine if portal's EQCR is empty
 *
 * For use in situations where a cpu-affine caller needs to determine when all
 * enqueues for the local portal have been processed by QMan but can't use the
 * QMAN_ENQUEUE_FLAG_WAIT_SYNC flag to do this from the final qman_enqueue().
 * The function forces tracking of EQCR consumption (which normally doesn't
 * happen until enqueue processing needs to find space to put new enqueue
 * commands), and returns zero if the ring still has unprocessed entries,
 * non-zero if it is empty.
 */
int qman_eqcr_is_empty(void);
```

5.2.2.2.6.1.4 Frame queue management

Frame queue objects are stored in memory provided by the caller, which makes the API for this object representation a little peculiar at first sight. The motivating factors are memory management and stashing of frame queue context. Another factor is that frame queue objects are the only objects in the QMan (or BMan) high level interfaces that are essentially arbitrary in number, so having the caller provide storage relieves the driver of having to know the best allocation scheme for all applications.

The `qman_create_fq()` API creates a new frame queue object, using the caller-supplied storage, and in which the caller has already configured the callback functions to be used for handling hardware-produced data - namely, DQRR entries and MR entries, the latter divided according to the type of message (software-enqueue rejections, hardware-enqueue rejections, or frame queue state changes).

```
#define QMAN_FQ_FLAG_NO_ENQUEUE      0x00000001 /* can't enqueue */
#define QMAN_FQ_FLAG_NO_MODIFY      0x00000002 /* can only enqueue */
#define QMAN_FQ_FLAG_TO_DCPORTAL    0x00000004 /* consumed by CAAM/PME/FMan */
#define QMAN_FQ_FLAG_LOCKED         0x00000008 /* multi-core locking */
#define QMAN_FQ_FLAG_AS_I           0x00000010 /* query h/w state */
```

```

#define QMAN_FQ_FLAG_DYNAMIC_FQID    0x00000020 /* (de)allocate fqid */
struct qman_fq {
    /* Caller of qman_create_fq() provides these demux callbacks */
    struct qman_fq_cb {
        qman_cb_dqrr dqrr;      /* for dequeued frames */
        qman_cb_mr_ern;        /* for s/w ERNs */
        qman_cb_mr_dc_ern;     /* for diverted h/w ERNs */
        qman_cb_mr_fqs;        /* frame-queue state changes*/
    } cb;
    /* Internal to the driver, don't touch. */
    [...]
};
/**
 * qman_create_fq - Allocates a FQ
 * @fqid: the index of the FQD to encapsulate, must be "Out of Service"
 * @flags: bit-mask of QMAN_FQ_FLAG_*** options
 * @fq: memory for storing the 'fq', with callbacks filled in
 *
 * Creates a frame queue object for the given @fqid, unless the
 * QMAN_FQ_FLAG_DYNAMIC_FQID flag is set in @flags, in which case a FQID is
 * dynamically allocated (or the function fails if none are available). Once
 * created, the caller should not touch the memory at 'fq' except as extended to
 *
 * adjacent memory for user-defined fields (see the definition of "struct
 * qman_fq" for more info). NO_MODIFY is only intended for enqueueing to
 * pre-existing frame-queues that aren't to be otherwise interfered with, it
 * prevents all other modifications to the frame queue. The TO_DCPORTAL flag
 * causes the driver to honour any contextB modifications requested in the
 * qm_init_fq() API, as this indicates the frame queue will be consumed by a
 * direct-connect portal (PME, CAAM, or FMan). When frame queues are consumed by
 *
 * software portals, the contextB field is controlled by the driver and can't be
 *
 * modified by the caller. If the AS_SI flag is specified, management commands
 * will be used on portal @p to query state for frame queue @fqid and construct
 * a frame queue object based on that, rather than assuming/requiring that it be
 * Out of Service.
 */
int qman_create_fq(u32 fqid, u32 flags, struct qman_fq *fq);
#define QMAN_FQ_DESTROY_PARKED      0x00000001 /* FQ can be parked or OOS */
/**
 * qman_destroy_fq - Deallocates a FQ
 * @fq: the frame queue object to release
 * @flags: bit-mask of QMAN_FQ_DESTROY_*** options
 *
 * The memory for this frame queue object ('fq' provided in qman_create_fq()) is
 * not deallocated but the caller regains ownership, to do with as desired. The
 * FQ must be in the 'out-of-service' state unless the QMAN_FQ_DESTROY_PARKED
 * flag is specified, in which case it may also be in the 'parked' state.
 */
void qman_destroy_fq(struct qman_fq *fq, u32 flags);

```

5.2.2.6.14.1 Querying a FQ object

The following functions do not interact with h/w, they simply return the state that the QMan driver tracks within the FQ object.

```

/**
 * qman_fq_fqid - Queries the frame queue ID of a FQ object
 * @fq: the frame queue object to query
 */

```

```

u32 qman_fq_fqid(struct qman_fq *fq);
enum qman_fq_state {
    qman_fq_state_oos,
    qman_fq_state_parked,
    qman_fq_state_sched,
    qman_fq_state_retired
};
#define QMAN_FQ_STATE_CHANGING      0x80000000 /* 'state' is changing */
#define QMAN_FQ_STATE_NE           0x40000000 /* retired FQ isn't empty */
#define QMAN_FQ_STATE_ORL         0x20000000 /* retired FQ has ORL */
#define QMAN_FQ_STATE_BLOCKOOS    0xe0000000 /* if any are set, no OOS */
#define QMAN_FQ_STATE_CGR_EN      0x10000000 /* CGR enabled */
/**
 * qman_fq_state - Queries the state of a FQ object
 * @fq: the frame queue object to query
 * @state: pointer to state enum to return the FQ scheduling state
 * @flags: pointer to state flags to receive QMAN_FQ_STATE_*** bitmask
 *
 * Queries the state of the FQ object, without performing any h/w commands.
 * This captures the state, as seen by the driver, at the time the function
 * executes.
 */
void qman_fq_state(struct qman_fq *fq, enum qman_fq_state *state, u32 *flags);

```

5.2.2.2.6.1.4.2 Initialize a FQ

The `qman_init_fq()` API requires that the caller fill in the details of the Initialize FQ command that they desire, and uses the 'struct `qm_mcc_initfq`' structure type to this end. This structure is quite elaborate, please consult the API header file and SDK examples for more informatoin.

```

#define QMAN_INITFQ_FLAG_SCHED      0x00000001 /* schedule rather than park */
#define QMAN_INITFQ_FLAG_NULL      0x00000002 /* zero 'contextB', no demux */
#define QMAN_INITFQ_FLAG_LOCAL     0x00000004 /* set dest portal */
/**
 * qman_init_fq - Initialises FQ fields, leaves the FQ "parked" or "scheduled"
 * @fq: the frame queue object to modify, must be 'parked' or new.
 * @flags: bit-mask of QMAN_INITFQ_FLAG_*** options
 * @opts: the FQ-modification settings, as defined in the low-level API
 *
 * @opts: the FQ-modification settings
 *
 * Select QMAN_INITFQ_FLAG_SCHED in @flags to cause the frame queue to be
 * scheduled rather than parked. Select QMAN_INITFQ_FLAG_NULL in @flags to
 * configure a frame queue that will not demux to a 'struct qman_fq' object when
 * dequeued frames or messages arrive at a software portal, but which will
 * instead trigger the portal's 'null_cb' callbacks (see qman_create_portal()).
 * NB, @opts can be NULL.
 *
 * Note that some fields and options within @opts may be ignored or overwritten
 * by the driver;
 * 1. the 'count' and 'fqid' fields are always ignored (this operation only
 * affects one frame queue: @fq).
 * 2. the QM_INITFQ_WE_CONTEXTB option of the 'we_mask' field and the associated
 * 'fqd' structure's 'context_b' field are sometimes overwritten;
 * - if @flags contains QMAN_INITFQ_FLAG_NULL, then context_b is initialized
 * to zero by the driver,
 * - if @fq was not created with QMAN_FQ_FLAG_TO_DCPORTAL, then context_b is
 * initialized to a value used by the driver for demux.
 * - if context_b is initialized for demux, so is context_a in case stashing

```

```

*   is requested (see item 4).
* (So caller control of context_b is only possible for TO_DCPORTAL frame queue
* objects.)
* 3. if @flags contains QMAN_INITFQ_FLAG_LOCAL, the 'fqd' structure's
* 'dest::channel' field will be overwritten to match the portal used to issue
* the command. If the WE_DESTWQ write-enable bit had already been set by the
* caller, the channel workqueue will be left as-is, otherwise the write-enable
* bit is set and the workqueue is set to a default of 4. If the "LOCAL" flag
* isn't set, the destination channel/workqueue fields and the write-enable bit
* are left as-is.
* 4. if the driver overwrites context_a/b for demux, then if
* QM_INITFQ_WE_CONTEXTA is set, the driver will only overwrite
* context_a.address fields and will leave the stashing fields provided by the
* user alone, otherwise it will zero out the context_a.stashing fields.
*/
int qman_init_fq(struct qman_fq *fq, u32 flags, struct qm_mcc_initfq *opts);

```

5.2.2.6.14.3 Schedule a FQ

```

/**
 * qman_schedule_fq - Schedules a FQ
 * @fq: the frame queue object to schedule, must be 'parked'
 *
 * Schedules the frame queue, which must be Parked, which takes it to
 * Tentatively-Scheduled or Truly-Scheduled depending on its fill-level.
 */
int qman_schedule_fq(struct qman_fq *fq);

```

5.2.2.6.14.4 Retire a FQ

```

/**
 * qman_retire_fq - Retires a FQ
 * @fq: the frame queue object to retire
 * @flags: FQ flags (as per qman_fq_state) if retirement completes immediately
 *
 * Retires the frame queue. This returns zero if it succeeds immediately, +1 if
 * the retirement was started asynchronously, otherwise it returns negative for
 * failure. When this function returns zero, @flags is set to indicate whether
 * the retired FQ is empty and/or whether it has any ORL fragments (to show up
 * as ERNs). Otherwise the corresponding flags will be known when a subsequent
 * FQRN message shows up on the portal's message ring.
 *
 * NB, if the retirement is asynchronous (the FQ was in the Truly Scheduled or
 * Active state), the completion will be via the message ring as a FQRN - but
 * the corresponding callback may occur before this function returns!! Ie. the
 * caller should be prepared to accept the callback as the function is called,
 * not only once it has returned.
 */
int qman_retire_fq(struct qman_fq *fq, u32 *flags);

```

5.2.2.6.14.5 Put a FQ out of service

```

/**
 * qman_oos_fq - Puts a FQ "out of service"
 * @fq: the frame queue object to be put out-of-service, must be 'retired'
 *
 * The frame queue must be retired and empty, and if any order restoration list

```

```
* was released as ERNs at the time of retirement, they must all be consumed.
*/
int qman_oos_fq(struct qman_fq *fq);
```

5.2.2.2.6.1.4.6 Query a FQD from QMan

The following functions perform query commands via the QMan software portal to obtain information about the FQD corresponding to the given FQ object. The data structures used by the query are quite elaborate, please consult the API header file and SDK examples for more information.

```
/**
 * qman_query_fq - Queries FQD fields (via h/w query command)
 * @fq: the frame queue object to be queried
 * @fqd: storage for the queried FQD fields
 */
int qman_query_fq(struct qman_fq *fq, struct qm_fqd *fqd);
/**
 * qman_query_fq_np - Queries non-programmable FQD fields
 * @fq: the frame queue object to be queried
 * @np: storage for the queried FQD fields
 */
int qman_query_fq_np(struct qman_fq *fq, struct qm_mcr_queryfq_np *np);
```

5.2.2.2.6.1.4.7 Unscheduled (volatile) dequeuing of a FQ

```
#define QMAN_VOLATILE_FLAG_WAIT      0x00000001 /* wait if VDQCR is in use */
#define QMAN_VOLATILE_FLAG_WAIT_INT 0x00000002 /* if wait, interruptible? */
#define QMAN_VOLATILE_FLAG_FINISH    0x00000004 /* wait till VDQCR completes */
/**
 * qman_volatile_dequeue - Issue a volatile dequeue command
 * @fq: the frame queue object to dequeue from (or NULL)
 * @flags: a bit-mask of QMAN_VOLATILE_FLAG_*** options
 * @vdqcr: bit mask of QM_VDQCR_*** options, as per qm_dqrr_vdqcr_set()
 *
 * Attempts to lock access to the portal's VDQCR volatile dequeue functionality.
 * The function will block and sleep if QMAN_VOLATILE_FLAG_WAIT is specified and
 * the VDQCR is already in use, otherwise returns non-zero for failure. If
 * QMAN_VOLATILE_FLAG_FINISH is specified, the function will only return once
 * the VDQCR command has finished executing (ie. once the callback for the last
 * DQRR entry resulting from the VDQCR command has been called). If @fq is
 * non-NULL, the corresponding FQID will be substituted in to the VDQCR command,
 * otherwise it is assumed that @vdqcr already contains the FQID to dequeue
 * from.
 */
int qman_volatile_dequeue(struct qman_fq *fq, u32 flags, u32 vdqcr)
```

5.2.2.2.6.1.4.8 Set FQ flow control state

```
/**
 * qman_fq_flow_control - Set the XON/XOFF state of a FQ
 * @fq: the frame queue object to be set to XON/XOFF state, must not be 'oos',
 * or 'retired' or 'parked' state
 * @xon: boolean to set fq in XON or XOFF state
 *
 * The frame should be in Tentatively Scheduled state or Truly Schedule sate,
 * otherwise the IFSI interrupt will be asserted.
```

```
*/
int qman_fq_flow_control(struct qman_fq *fq, int xon);
```

5.2.2.2.6.1.5 Enqueue Command (without ORP)

```
#define QMAN_ENQUEUE_FLAG_WAIT      0x00010000 /* wait if EQCR is full */
#define QMAN_ENQUEUE_FLAG_WAIT_INT  0x00020000 /* if wait, interruptible? */
#define QMAN_ENQUEUE_FLAG_WAIT_SYNC 0x00000004 /* if wait, until consumed? */
#define QMAN_ENQUEUE_FLAG_WATCH_CGR 0x00080000 /* watch congestion state */
#define QMAN_ENQUEUE_FLAG_DCA       0x00008000 /* perform enqueue-DCA */
#define QMAN_ENQUEUE_FLAG_DCA_PARK  0x00004000 /* If DCA, requests park */
#define QMAN_ENQUEUE_FLAG_DCA_PTR(p) /* If DCA, p is DQRR entry */ \
    (((u32)(p) << 2) & 0x00000f00)
#define QMAN_ENQUEUE_FLAG_C_GREEN   0x00000000 /* choose one C_*** flag */
#define QMAN_ENQUEUE_FLAG_C_YELLOW  0x00000008
#define QMAN_ENQUEUE_FLAG_C_RED     0x00000010
#define QMAN_ENQUEUE_FLAG_C_OVERRIDE 0x00000018
/**
 * qman_enqueue - Enqueue a frame to a frame queue
 * @fq: the frame queue object to enqueue to
 * @fd: a descriptor of the frame to be enqueued
 * @flags: bit-mask of QMAN_ENQUEUE_FLAG_*** options
 *
 * Fills an entry in the EQCR of portal @qm to enqueue the frame described by
 * @fd. The descriptor details are copied from @fd to the EQCR entry, the 'pid'
 * field is ignored. The return value is non-zero on error, such as ring full
 * (and FLAG_WAIT not specified), congestion avoidance (FLAG_WATCH_CGR
 * specified), etc. If the ring is full and FLAG_WAIT is specified, this
 * function will block. If FLAG_INTERRUPT is set, the EQCI bit of the portal
 * interrupt will assert when QMan consumes the EQCR entry (subject to "status
 * disable", "enable", and "inhibit" registers). If FLAG_DCA is set, QMan will
 * perform an implied "discrete consumption acknowledgement" on the dequeue
 * ring's (DQRR) entry, at the ring index specified by the FLAG_DCA_IDX(x)
 * macro. (As an alternative to issuing explicit DCA actions on DQRR entries,
 * this implicit DCA can delay the release of a "held active" frame queue
 * corresponding to a DQRR entry until QMan consumes the EQCR entry - providing
 * order-preservation semantics in packet-forwarding scenarios.) If FLAG_DCA is
 * set, then FLAG_DCA_PARK can also be set to imply that the DQRR consumption
 * acknowledgement should "park request" the "held active" frame queue. Ie.
 * when the portal eventually releases that frame queue, it will be left in the
 * Parked state rather than Tentatively Scheduled or Truly Scheduled. If the
 * portal is watching congestion groups, the QMAN_ENQUEUE_FLAG_WATCH_CGR flag
 * is requested, and the FQ is a member of a congestion group, then this
 * function returns -EAGAIN if the congestion group is currently congested.
 * Note, this does not eliminate ERNs, as the async interface means we can be
 * sending enqueue commands to an un-congested FQ that becomes congested before
 * the enqueue commands are processed, but it does minimise needless thrashing
 * of an already busy hardware resource by throttling many of the to-be-dropped
 * enqueues "at the source".
 */
int qman_enqueue(struct qman_fq *fq, const struct qm_fd *fd, u32 flags);
```

5.2.2.2.6.1.6 Enqueue Command with ORP

```
/* Same flags as qman_enqueue(), with the following additions;

 * - this flag indicates "Not Last In Sequence", ie. all but the final fragment
```

```
* of a frame. */
#define QMAN_ENQUEUE_FLAG_NLIS      0x01000000
/* - this flag performs no enqueue but fills in an ORP sequence number that
 * would otherwise block it (eg. if a frame has been dropped). */
#define QMAN_ENQUEUE_FLAG_HOLE      0x02000000
/* - this flag performs no enqueue but advances NESN to the given sequence
 * number. */
#define QMAN_ENQUEUE_FLAG_NESN      0x04000000
/*
 * qman_enqueue_orp - Enqueue a frame to a frame queue using an ORP
 * @fq: the frame queue object to enqueue to
 * @fd: a descriptor of the frame to be enqueued
 * @flags: bit-mask of QMAN_ENQUEUE_FLAG_*** options
 * @orp: the frame queue object used as an order restoration point.
 * @orp_seqnum: the sequence number of this frame in the order restoration path
 *
 * Similar to qman_enqueue(), but with the addition of an Order Restoration
 * Point (@orp) and corresponding sequence number (@orp_seqnum) for this
 * enqueue operation to employ order restoration. Each frame queue object acts
 * as an Order Definition Point (ODP) by providing each frame dequeued from it
 * with an incrementing sequence number, this value is generally ignored unless
 * that sequence of dequeued frames will need order restoration later. Each
 * frame queue object also encapsulates an Order Restoration Point (ORP), which
 * is a re-assembly context for re-ordering frames relative to their sequence
 * numbers as they are enqueued. The ORP does not have to be within the frame
 * queue that receives the enqueued frame, in fact it is usually the frame
 * queue from which the frames were originally dequeued. For the purposes of
 * order restoration, multiple frames (or "fragments") can be enqueued for a
 * single sequence number by setting the QMAN_ENQUEUE_FLAG_NLIS flag for all
 * enqueues except the final fragment of a given sequence number. Ordering
 * between sequence numbers is guaranteed, even if fragments of different
 * sequence numbers are interlaced with one another. Fragments of the same
 * sequence number will retain the order in which they are enqueued. If no
 * enqueue is performed, QMAN_ENQUEUE_FLAG_HOLE indicates that the given
 * sequence number is to be "skipped" by the ORP logic (eg. if a frame has been
 * dropped from a sequence), or QMAN_ENQUEUE_FLAG_NESN indicates that the given
 * sequence number should become the ORP's "Next Expected Sequence Number".
 *
 * Side note: a frame queue object can be used purely as an ORP, without
 * carrying any frames at all. Care should be taken not to deallocate a frame
 * queue object that is being actively used as an ORP, as a future allocation
 * of the frame queue object may start using the internal ORP before the
 * previous use has finished.
 */
int qman_enqueue_orp(struct qman_fq *fq, const struct qm_fd *fd, u32 flags,
                    struct qman_fq *orp, u16 orp_seqnum);
```

5.2.2.2.6.1.7 DCA Mode

As described in [Order Preservation & Discrete Consumption Acknowledgement](#) on page 116, FQs initialized for "hold active" behavior can have order-preservation behavior if their DQRR entries are consumed either by implicit DCA in the enqueue command when forwarding, or by explicit DCA if the frame is not going to be forwarded. The implicit DCA via enqueue is described in [Enqueue Command \(without ORP\)](#) on page 129, this section describes the API for performing an explicit DCA on a DQRR entry. As with the implicit DCA via enqueue, explicit DCA commands also allow the caller to specify that the FQ be Parked rather than rescheduled once all its DQRR entries are consumed.

```
/**
 * qman_dca - Perform a Discrete Consumption Acknowledgement
```

```

* @dq: the DQRR entry to be consumed
* @park_request: indicates whether the held-active @fq should be parked
*
* Only allowed in DCA-mode portals, for DQRR entries whose handler callback had
* previously returned 'qman_cb_dqrr_defer'. NB, as with the other APIs, this
* does not take a 'portal' argument but implies the core affine portal from the
*
* cpu that is currently executing the function. For reasons of locking, this
* function must be called from the same CPU as that which processed the DQRR
* entry in the first place.
*/
void qman_dca(struct qm_dqrr_entry *dq, int park_request);

```

5.2.2.2.6.1.8 Congestion Management Records

QMan supports a fixed number^[1] of built-in resources called Congestion Group Records (CGRs), that can be used as containers for related frame queues that should collectively benefit from congestion management. The precise algorithms used for congestion management with these records is beyond the scope of the document, please see the Queue Manager section of the appropriate QorIQ SoC Reference Manual for details.

The CGR kernel structure enables access to the CGR hardware functionality. Each object refers to an underlining hardware record via the `cgrid` field. Many CGR object may reference the same `cgrid`, but care must be taken when this object resides on different cores as no inter-core protection is provided.

The init frame queue functionality allows the caller to associate a CGR with the associated frame queue. The interface permits the management and modification of the underlining CGRs and notifies the user of congestion state changed. The current interface does not provide a mechanism to manage CGR ids. The application software is expected to arbitrate use of CGR ids.

```

/* Flags to qman_modify_cgr() */
#define QMAN_CGR_FLAG_USE_INIT      0x00000001
/**
 * This is a qman cgr callback function which gets invoked when the
typedef void (*qman_cb_cgr)(struct qman_portal *qm,
        struct qman_cgr *cgr, int congested);
struct qman_cgr {
    /* Set these prior to qman_create_cgr() */
    u32 cgrid; /* 0..255 */
    qman_cb_cgr cb;
    enum qm_channel chan; /* portal channel this object is created on */
    struct list_head node;
};
/* When Weighted Random Early Discard (WRED) is used then the following
 * structure is used to configure the WRED parameters. Refer to the QMan
 * Block Guide for a detailed description of the various parameters.
 */
struct qm_cgr_wr_parm {
    union {
        u32 word;
        struct {
            u32 MA:8;
            u32 Mn:5;
            u32 SA:7; /* must be between 64-127 */
            u32 Sn:6;
            u32 Pn:6;
        } __packed;
    };
};

```

[1] 256 for P4080/P5020/P3041

```

} __packed;
/* This struct represents the 13-bit "CS_THRES" CGR field. In the corresponding
 * management commands, this is padded to a 16-bit structure field, so that's
 * how we represent it here. The congestion state threshold is calculated from
 * these fields as follows;
 *   CS threshold = TA * (2 ^ Tn)
 */
struct qm_cgr_cs_thres {
    u16 __reserved:3;
    u16 TA:8;
    u16 Tn:5;
} __packed;
/* This identical structure of CGR fields is present in the "Init/Modify CGR"
 * commands and the "Query CGR" result. It's suctioned out here into its own
 * struct. */
struct __qm_mc_cgr {
    struct qm_cgr_wr_parm wr_parm_g;
    struct qm_cgr_wr_parm wr_parm_y;
    struct qm_cgr_wr_parm wr_parm_r;
    u8 wr_en_g; /* boolean, use QM_CGR_EN */
    u8 wr_en_y; /* boolean, use QM_CGR_EN */
    u8 wr_en_r; /* boolean, use QM_CGR_EN */
    u8 cscn_en; /* boolean, use QM_CGR_EN */
    union {
        struct {
            u16 cscn_targ_upd_ctrl; /* use QM_CSCN_TARG_UDP_ */
            u16 cscn_targ_dcp_low; /* CSCN_TARG_DCP low-16bits */
        };
        u32 cscn_targ; /* use QM_CGR_TARG_ */
    };
    u8 cstd_en; /* boolean, use QM_CGR_EN */
    u8 cs; /* boolean, only used in query response */
    struct qm_cgr_cs_thres cs_thres;
    u8 mode; /* QMAN_CRG_MODE_FRAME not supported in rev1.0 */
} __packed
struct qm_mcc_initcgr {
    u8 __reserved1;
    u16 we_mask; /* Write Enable Mask */
    struct __qm_mc_cgr cgr; /* CGR fields */
    u8 __reserved2[2];
    u8 cgid;
    u8 __reserved4[32];
} __packed;
/**
 * qman_create_cgr - Register a congestion group object
 * @cgr: the 'cgr' object, with fields filled in
 * @flags: QMAN_CGR_FLAG_* values
 * @opts: optional state of CGR settings
 *
 * Registers this object to receiving congestion entry/exit callbacks on the
 * portal affine to the cpu portal on which this API is executed. If opts is
 * NULL then only the callback (cgr->cb) function is registered. If @flags
 * contains QMAN_CGR_FLAG_USE_INIT, then an init hw command (which will reset
 * any unspecified parameters) will be used rather than a modify hw hardware
 * (which only modifies the specified parameters).
 */
int qman_create_cgr(struct qman_cgr *cgr, u32 flags, struct qm_mcc_initcgr *opts);
/**
 * qman_create_cgr_to_dcp - Register a congestion group object to DCP portal
 * @cgr: the 'cgr' object, with fields filled in

```

```

* @flags: QMAN_CGR_FLAG_* values
* @dcp_portal: the DCP portal to which the cgr object is registered
* @opts: optional state of CGR settings
*
*/
int qman_create_cgr_to_dcp(struct qman_cgr *cgr, u32 flags, u16 dcp_portal,
                        struct qm_mcc_initcgr *opts);
/**
* qman_delete_cgr - Deregisters a congestion group object
* @cgr: the 'cgr' object to deregister
*
* "Unplugs" this CGR object from the portal affine to the cpu on which this API
* is executed. This must be excuted on the same affine portal on which it was
* created.
*/
int qman_delete_cgr(struct qman_cgr *cgr);
/**
* qman_modify_cgr - Modify CGR fields
* @cgr: the 'cgr' object to modify
* @flags: QMAN_CGR_FLAG_* values
* @opts: the CGR-modification settings
*
* The @opts parameter can be NULL. Note that some fields and options within
* @opts may be ignored or overwritten by the driver, in particular the 'cgrid'
* field is ignored (this operation only affects the given CGR object). If
* @flags contains QMAN_CGR_FLAG_USE_INIT, then an init hw command (which will
* reset any unspecified parameters) will be used rather than a modify hw
* hardware (which only modifies the specified parameters).
*/
int qman_modify_cgr(struct qman_cgr *cgr, u32 flags, struct qm_mcc_initcgr *opts);
/**
* qman_query_cgr - Queries CGR fields
* @cgr: the 'cgr' object to query
* @result: storage for the queried congestion group record
*/
int qman_query_cgr(struct qman_cgr *cgr, struct qm_mcr_querycgr *result);

```

5.2.2.2.6.1.9 Zero-Configuration Messaging

As described in [Overview \(QMan\)](#) on page 120, the demux logic of the QMan portal driver uses the contextB field of FQDs, as published in DQRR and MR entries, to determine the corresponding FQ object, and from there the DQRR or MR callback to invoke. However, "default callbacks" can also be associated with a portal that will be used if a "NULL" FQ is dequeued from, where NULL refers to a FQD whose contextB entry has been initialized to NULL (this occurs when using the QMAN_INITFQ_FLAG_NULL flag to the qman_init_fq() API, described in [Initialize a FQ](#) on page 126).

The purpose of this mechanism is to allow the user of one portal to enqueue frames on any frame queue that is configured in this way and schedule it to another portal. For virtualization or AMP scenarios, it is a difficult architectural problem to configure all guest operating systems to agree, in advance, on run-time parameters. The use of NULL frame queues allows a control plane guest OS to use any frame queue, configured with a NULL "contextB" field (see the QMAN_INITFQ_FLAG_NULL flag in the "Frame queue management" section below), to send any and all such configuration to another guest by scheduling that NULL frame queue to one of the target guest's portals. The target guest will have the portal's "NULL" callbacks invoked rather than those of any frame queue objects, and as such this provides what could be considered a "zero-configuration" interface - no agreement is required over what frame queue that configuration information will be arriving on, only that the configuration will arrive via the portal as a message on a NULL frame queue.

NOTE

Unless the payload of FDs passed over a zero-config FQ fits entirely within the 32-bit cmd/status field, buffers will presumably be required and the zero-configuration mechanism described here does not address how the sending and receiving ends should agree on what memory resources and management to use for this.

```
/**
 * qman_get_null_cb - get callbacks currently used for "null" frame queues
 *
 * Copies the callbacks used for the affine portal of the current cpu.
 */
void qman_get_null_cb(struct qman_fq_cb *null_cb);
/**
 * qman_set_null_cb - set callbacks to use for "null" frame queues
 *
 * Sets the callbacks to use for the affine portal of the current cpu, whenever
 * a DQRR or MR entry refers to a "null" FQ object. (Eg. zero-conf messaging.)
 */
void qman_set_null_cb(const struct qman_fq_cb *null_cb);
```

5.2.2.2.6.1.10 FQ allocation

5.2.2.2.6.1.10.1 Ad-hoc FQ allocator

As described in [Seeding Buffer Pools](#) on page 104, BMan buffer pool ID zero is currently reserved for use as an ad-hoc FQ allocator. As seen in [Frame queue management](#) on page 124, this feature can be used implicitly when creating a FQ object by passing the QMAN_FQ_FLAG_DYNAMIC_FQID flag to `qman_init_fq()`. The advantage of this mechanism is that it works across all cpus/portals, independent of any hypervisor or other system partitioning. The disadvantage of this mechanism is that does not permit the atomic nor contiguous allocation of more than one FQ at a time, and in particular most high-performance uses of FMan require contiguous ranges of FQIDs that also meet certain alignment requirements (ie. that the FQID range begins on an aligned FQID value).

5.2.2.2.6.1.10.2 FQ range allocator

The following APIs allow software to allocate and release arbitrary ranges of FQIDs, but it should be noted that the current version of the NXP Datapath software implements this without any hardware interaction. As such, multiple (guest) systems running on the same chip will each have their own allocator and are not aware of each other's (de)allocations. The range allocator's default state is empty, and it can be seeded by calling `qman_release_fqid_range()` on initialization with an appropriate FQID range to manage. The intention is for the control-plane software to initialize this range and to perform all allocations and deallocations on behalf of any software running on different system instances.

```
/**
 * qman_alloc_fqid_range - Allocate a contiguous range of FQIDs
 * @result: is set by the API to the base FQID of the allocated range
 * @count: the number of FQIDs required
 * @align: required alignment of the allocated range
 * @partial: non-zero if the API can return fewer than @count FQIDs
 * Returns the number of frame queues allocated, or a negative error code. If
 * @partial is non zero, the allocation request may return a smaller range of
 * FQs than requested (though alignment will be as requested). If @partial is
 * zero, the return value will either be 'count' or negative.
 */
int qman_alloc_fqid_range(u32 *result, u32 count, u32 align, int partial);
/**
 * qman_release_fqid_range - Release the specified range of frame queue IDs
 * @fqid: the base FQID of the range to deallocate
 * @count: the number of FQIDs in the range
 */
```

```

* This function can also be used to seed the allocator with ranges of FQIDs
* that it can subsequently use. Returns zero for success.
*/
void qman_release_fqid_range(u32 fqid, unsigned int count);

```

5.2.2.2.6.1.10.3 Future FQ allocator changes

Please note that a future version of the NXP Datapath software will automatically seed the range allocator with all FQIDs available to QMan, it will reimplement these APIs over an IPC layer such that all system instances share a common allocator instance, and the BMan-based FQ allocator will be removed and the corresponding APIs being reimplemented to use this range allocator.

5.2.2.2.6.1.11 Helper functions

In cases where software running on different CPUs communicate using QMan frame queues, there can arise an initialization problem related to synchronisation. If one side is termed the producer and the other the consumer, then the question becomes one of when it is safe for the producer to enqueue to that FQ. It is normal for software consumers to take care of initializing and scheduling FQs, because they must provide initialization and scheduling details in order for dequeue-handling to function correctly. But on the producer side, any attempt to enqueue to the FQ prior to the FQ being initialized will be rejected (enqueues are not permitted to OutOfService FQs). The following inline function can be used directly or as an example of how to determine when a FQ has changed state.

NOTE

It is safe for the producer to enqueue once the FQ has been initialized but not yet scheduled by the consumer.

```

/**
 * qman_poll_fq_for_init - Check if an FQ has been initialized from OOS
 * @fqid: the FQID that will be initialized by other s/w
 *
 * In many situations, a FQID is provided for communication between s/w
 * entities, and whilst the consumer is responsible for initialising and
 * scheduling the FQ, the producer(s) generally create a wrapper FQ object using
 * and only call qman_enqueue() (no FQ initialisation, scheduling, etc). Ie;
 *   qman_create_fq(..., QMAN_FQ_FLAG_NO_MODIFY, ...);
 * However, data can not be enqueued to the FQ until it is initialized out of
 * the OOS state - this function polls for that condition. It is particularly
 * useful for users of IPC functions - each endpoint's Rx FQ is the other
 * endpoint's Tx FQ, so each side can initialise and schedule their Rx FQ object
 * and then use this API on the (NO_MODIFY) Tx FQ object in order to
 * synchronise. The function returns zero for success, +1 if the FQ is still in
 * the OOS state, or negative if there was an error.
 */
static inline int qman_poll_fq_for_init(struct qman_fq *fq)
{
    struct qm_mcr_queryfq_np np;
    int err;
    err = qman_query_fq_np(fq, &np);
    if (err)
        return err;
    if ((np.state & QM_MCR_NP_STATE_MASK) == QM_MCR_NP_STATE_OOS)
        return 1;
    return 0;
}

```

5.2.2.2.7 QMan CEETM APIs

CEETM is a version of egress traffic management in QMan that provides hierarchical class based scheduling and traffic shaping. It is intended for use on links leading to a Wide Area Network (WAN).

5.2.2.2.7.1 QMan CEETM Device-Tree Node

The QMan driver determines the available CEETM resources from the device tree. The definition of CEETM node is defined as:

```
qman-ceetm@0 {
    compatible = "fsl,qman-ceetm";
    fsl,ceetm-lfqid-range = <0xf00000 0x1000>;
    fsl,ceetm-sp-range = <0 12>;
    fsl,ceetm-lni-range = <0 8>;
    fsl,ceetm-channel-range = <0 32>;
};
```

Each *-range node will specify the CEETM resource(lfqid/sp/lni/cq channel) by a 2-cell value <x y>, in which x is the first value and y is the total number.

There are two ceetm device trees under arch/powerpc/boot/dts/fsl: qoriq-qman-ceetm0.dtsi and qoriq-qman-ceetm1.dts, which are used for DCP0 and DCP1 CEETM resources separately. To enable the CEETM initialization routine, the ceetm device tree should be added in the board's device tree.

For example, in t4240qds.dts, add the following lines at the bottom:

```
/include/ "fsl/qoriq-qman-ceetm0.dtsi"
/include/ "fsl/qoriq-qman-ceetm1.dtsi"
```

In b4860qds.dts, add the following line at the bottom: (because b4860 only support CEETM instance on DCP0).

```
/include/ "fsl/qoriq-qman-ceetm0.dtsi"
```

5.2.2.2.7.2 The token rate of CEETM shaper

5.2.2.2.7.2.1 The token rate structure

The QMan CEETM provides two rate shapers – CR and ER shapers to shape the traffic with the given rate. Both shapers use the token credit rate and credit update reference period to determine the shaper's output rate (in bits/second). The token rate is specified in bytes with an 11 bit integer and a 13 bit fractional part, and can be configured via CEETM shaper configuration commands. Here is the definition of the token rate structure:

```
/* Token Rate Structure
/* The shaping rates are based on a "credit" system and a pre-configured h/w
* internal timer. The following type represents a shaper "rate" parameter as a
* fractional number of "tokens". Here's how it works. This (fractional) number
* of tokens is added to the shaper's "credit" every time the h/w timer elapses
* (up to a limit which is set by another shaper parameter). Every time a frame
* is enqueued through a shaper, the shaper deducts as many tokens as there are
* bytes of data in the enqueued frame. A shaper will not allow itself to
* enqueue any frames if its token count is negative. As such:
*
*           The rate at which data is enqueued is limited by the
*           rate at which tokens are added.
*
* Therefore if the user knows the period between these h/w timer updates in
* seconds, they can calculate the maximum traffic rate of the shaper (in
```

```

* bytes-per-second) from the token rate. And vice versa, they can calculate
* the token rate to use in order to achieve a given traffic rate.
*/
struct qm_ceetm_rate {
    /* The token rate is; whole + (fraction/8192) */
    u32 whole:11; /* 0..2047 */
    u32 fraction:13; /* 0..8191 */
};

```

5.2.2.2.72.2 The APIs to convert token rate and shapers output rate

The suggested value for credit update reference period is 1000ns as stated in the PRES field's description of CEETM_CFG_PRES register in Reference Manual. The calculation for shapers output rate is based on:

*shaper output rate (in bits/sec) = token credit rate * 8 / credit update reference period*

For more detail, please refer to the Appendix in this document. The following APIs are used to convert shaper output rate in bps to token rate and vice versa, which can be applied to both LNI and channel shaping:

```

/**
 * qman_ceetm_bps2tokenrate - Given a desired rate shaper rate measured in bps
 * (ie. bits-per-second), compute the 'token_rate' fraction that best
 * approximates that rate.
 *
 * @bps: the input shaper rate in bps.
 * @token_rate: the output token rate computed with the given bps.
 * @rounding: dictates how to round if an exact conversion is not possible; if
 * it is negative then 'token_rate' will round down to the highest value that
 * does not exceed the desired rate, if it is positive then 'token_rate' will
 * round up to the lowest value that is greater than or equal to the desired
 * rate, and if it is zero then it will round to the nearest approximation,
 * whether that be up or down.
 *
 * Returns zero for success, or -EINVAL if prescaler or qman clock is not available.
 */
int qman_ceetm_bps2tokenrate(u32 bps,
                            struct qm_ceetm_rate *token_rate,
                            int rounding);

/**
 * qman_ceetm_tokenrate2bps - Given a 'token_rate', compute the corresponding
 * number of 'bps'.
 * @token_rate: the input token rate fraction.
 * @bps: the output shaper rate in bps.
 * @rounding: has the same semantics as the previous function.
 *
 * Return zero for success, or -EINVAL if prescaler or qman clock is not available.
 */
int qman_ceetm_tokenrate2bps(const struct qm_ceetm_rate *token_rate,
                             u32 *kbps,
                             int rounding);

```

5.2.2.2.73 CEETM Sub-portal

CEETM is implemented as a mode of scheduling for sub-portals on specific QMan Direct-Connect Portals. We need to claim the sub-portal on a DCP to support CEETM.

5.2.2.2.73.1 Claim/release sub-portal

```
/* *
 * qman_ceetm_sp_claim - Claims the given sub-portal, provided it is available
 * to use and configured for traffic-management.
 * @sp: the returned sub-portal object, if successful.
 * @dcp_id: specifies the desired FMan block (and thus the relevant CEETM instance).
 * The type enum qm_dc_portal has already been defined in fsl_qman.h
 * @sp_idx: is the desired sub-portal index from 0 to 15.
 *
 * Returns zero for success, or -ENODEV if the sub-portal is in use, or -EINVAL
 * if the sp_idx is out of range.
 *
 *
 * Note that if there are multiple driver domains (eg. a linux kernel versus
 * user-space drivers in USDPAA, or multiple guests running under a
 * hypervisor) then a sub-portal may be accessible by more than one instance
 * of a qman driver and so it may be claimed multiple times. If this is the
 * case, it is up to the system architect to prevent conflicting configuration
 * actions coming from the different driver domains. The qman drivers do not
 * have any behind-the-scenes coordination to prevent this from happening.
 */
int qman_ceetm_sp_claim(struct qm_ceetm_sp **sp,
                      enum qm_dc_portal dcp_id,
                      unsigned int sp_idx);

/**
 * qman_ceetm_sp_release - Releases a previously claimed sub-portal.
 * @sp: the sub-portal to be released.
 *
 * Returns 0 for success, or -EBUSY for failure if the dependencies are not
 * released yet.
 */
int qman_ceetm_sp_release(struct qm_ceetm_sp *sp);
```

5.2.2.2.74 CEETM LNI - Logic Network Interface

Each QMan CEETM instance supports up to 8 Logical Network Interfaces (LNIs) which can be mapped to a DCP sub-portal. Each LNI aggregates frames from one or more QMan CEETM channels and priority schedules unshaped frames, CR frames and ER frames. It applies a dual rate shaper to aggregate CR/ER frames from shaped channel. The user can enable and disable the shaper, change the shaper rate for LNI, as well as control the CEETM traffic class flow control because it is maintained on a per LNI basis and applied to all class queue channels within the LNI.

5.2.2.2.74.1 Claim/release LNI

```
/**
 * qman_ceetm_lni_claim - Claims an unclaimed LNI.
 * @lni: the returned LNI object, if successful.
 * @dcp_id: specifies the desired FMan block (and thus the relevant CEETM instance).
 * @lni_idx: the desired LNI index.
 *
 * Returns zero for success, or -EINVAL for failure, which will happen
 * if the LNI is not available or has already been claimed (and not yet
 * successfully released), or lni_idx is out of range.
```

```

*
* Note that there may be multiple driver domains (or instances) that need to transmit
* out the same LNI, so this claim is only guaranteeing exclusivity within the
* domain of the driver being called. See qman_ceetm_sp_claim() and
* qman_ceetm_sp_get_lni() for more information.
*/
int qman_ceetm_lni_claim(struct qm_ceetm_lni **lni,
                        enum qm_dc_portal dcp_id,
                        unsigned int lni_idx);

/**
 * qman_ceetm_lni_release - Releases a previously claimed LNI.
 * @lni: the LNI needs to be released.
 *
 * This will only succeed if all dependent objects have been released.
 * Returns zero for success. Returns -EBUSY if the dependencies are not released.
 */
int qman_ceetm_lni_release(struct qm_ceetm_lni *lni);

```

5.2.2.2.74.2 Map sub-portal with LNI

```

/**
 * qman_ceetm_sp_set_lni
 * qman_ceetm_sp_get_lni - Set/get the LNI that the sub-portal is currently mapped to.
 * @sp: the given sub-portal.
 * @lni(in "set" function): the LNI object which the sub-portal will be mapped to.
 * @lni_idx(in "get" function): the LNI index which the sub-portal is mapped to.
 *
 * Returns zero for success. * Returns -EINVAL for "set" function when this sp-lni mapping has
 * been set, or
 * configure mapping command returns error, and
 * -EINVAL for "get" function when this sp-lni mapping is not set, or the query
 * mapping command returns error.
 *
 * This may be useful in situations where multiple driver
 * domains have access to the same sub-portals in order to all be able to
 * transmit out the same physical interface (perhaps they're on different IP
 * addresses or VPNs, so FMan is splitting Rx traffic and here we need to
 * converge Tx traffic). In that case, a control-plane is likely to use
 * qman_ceetm_lni_claim() followed by qman_ceetm_sp_set_lni() to configure the
 * sub-portal, and other domains are likely to use qman_ceetm_sp_get_lni()
 * followed by qman_ceetm_lni_claim() in order to determine the LNI that the
 * control-plane had assigned. This is why the "get" returns an index, whereas
 * the "set" takes an (already claimed) LNI object.
 */
int qman_ceetm_sp_set_lni(struct qm_ceetm_sp *sp,
                          struct qm_ceetm_lni *lni);
int qman_ceetm_sp_get_lni(struct qm_ceetm_sp *sp,
                           unsigned int *lni_idx);

```

5.2.2.2.74.3 Configure LNI shaper

The LNI provides two rate shapers that can be used to shape the traffic from all class queue channels which are shaped on input to the channel scheduler. The LNI shapers are used to shape or rate limit the aggregate of the class queue channels

which have each been shaped. The two shaper rates are applied only to frames shaped by the corresponding shaper at the channel shaper level.

```
/**
 * qman_ceetm_lni_enable_shaper
 * qman_ceetm_lni_disable_shaper - Enables/disables shaping on the LNI.
 * @lni: the given LNI.
 * @coupled: indicates whether CR and ER shapers are coupled.
 * @oal: the overhead accounting length which is added to the actual length of
 * each frame when performing shaper calculations.
 *
 * When the number of (unused) committed-rate tokens reach the committed-rate
 * token limit, @coupled indicates whether surplus tokens should be added to
 * the excess-rate token count (up to the excess-rate token limit). Whenever
 * a claimed LNI is first enabled for shaping, its committed and excess token
 * rates and limits are zero, so will need to be changed to do anything useful.
 * The shaper can subsequently be enabled/disabled without resetting the shaping
 * parameters, but the shaping parameters will be reset when the LNI is released.
 *
 * Returns 0 for success, or errno for "enable" function in the cases as:
 * a) -EINVAL if the shaper is already enabled.
 * b) -EIO if the configure shaper command returns error.
 * For "disable" function, returns:
 * a) -EINVAL if the shaper has been disabled.
 * b) -EIO if the query shaper command returns error.
 */
int qman_ceetm_lni_enable_shaper(struct qm_ceetm_lni *, int coupled, int oal);
int qman_ceetm_lni_disable_shaper(struct qm_ceetm_lni *);

/**
 * qman_ceetm_lni_is_shaper_enabled - Check LNI shaper status
 * @lni: the give LNI
 */
int qman_ceetm_lni_is_shaper_enabled(struct qm_ceetm_lni *lni);

/**
 * qman_ceetm_lni_set_commit_rate
 * qman_ceetm_lni_get_commit_rate
 * qman_ceetm_lni_set_excess_rate
 * qman_ceetm_lni_get_excess_rate - Set/get the shaper CR/ER token rate and token
 * limit of the given LNI.
 * @lni: the given LNI.
 * @token_rate: the desired token rate for "set" function, or the token rate of
 * the LNI queried by "get" function.
 * @token_limit: the desired token limit for "set" function, or the token limit of
 * the LNI queried by "get" function.
 *
 * Returns zero for success. The "set" function returns -EINVAL if the given
 * LNI is unshaped, or -EIO if the configure shaper command returns error.
 * The "get" function returns -EINVAL if the token rate or token limit is not set,
 * or the query command returns error
 */
int qman_ceetm_lni_set_commit_rate(struct qm_ceetm_lni *,
                                   const struct qm_ceetm_rate *token_rate,
                                   u16 token_limit);
int qman_ceetm_lni_get_commit_rate(struct qm_ceetm_lni *,
                                   struct qm_ceetm_rate *token_rate,
```

```

        u16 *token_limit);
int qman_ceetm_lni_set_excess_rate(struct qm_ceetm_lni *,
        const struct qm_ceetm_rate *token_rate,
        u16 token_limit);
int qman_ceetm_lni_get_excess_rate(struct qm_ceetm_lni *,
        struct qm_ceetm_rate *token_rate,
        u16 *token_limit);

/**
 * qman_ceetm_lni_set_commit_rate_bps
 * qman_ceetm_lni_get_commit_rate_bps
 * qman_ceetm_lni_set_excess_rate_bps
 * qman_ceetm_lni_get_excess_rate_bps - Set/get the shaper CR/ER rate
 * and token limit for the given LNI.
 * @lni: the given LNI.
 * @bps: the desired shaping rate in bps for "set" function, or the shaping rate
 * of the LNI queried by "get" function.
 * @token_limit: the desired token bucket limit for "set" function, or the token
 * limit of the given LNI queried by "get" function.
 *
 * Returns zero for success. The "set" function returns -EINVAL if the given
 * LNI is unshaped or -EIO if the configure shaper command returns error.
 * The "get" function returns -EINVAL if the token rate or the token limit is
 * not set or the query command returns error.
 */
int qman_ceetm_lni_set_commit_rate_bps(struct qm_ceetm_lni *lni,
        u64 bps,
        u16 token_limit);
int qman_ceetm_lni_get_commit_rate_bps(struct qm_ceetm_lni *lni,
        u64 *bps, u16 *token_limit);
int qman_ceetm_lni_set_excess_rate_bps(struct qm_ceetm_lni *lni,
        u64 bps,
        u16 token_limit);
int qman_ceetm_lni_get_excess_rate_bps(struct qm_ceetm_lni *lni,
        u64 *bps, u16 *token_limit);

```

5.2.2.7.4.4 Configure LNI traffic class flow control

```

/**
 * qman_ceetm_lni_set_tcfcc -configure "Traffic Class Flow Control".
 * qman_ceetm_lni_get_tcfcc - query "Traffic Class Flow Control".
 * @lni: the given LNI.
 * @cq_level: between 0 and 15, representing individual class queue levels (CQ0 to
 * CQ7 for every channel) and grouped class queue levels (CQ8 to CQ15 for every
 * channel).
 * @traffic_class: between 0 and 7 when associating a given class queue level to a
 * traffic class, or -1 when disabling traffic class flow control for this class
 * queue level.
 *
 * Returns zero for success, or -EINVAL if the cq_level or traffic_class is out of
 * the range or -EIO if configure/query tcfcc command returns error.
 *
 * Refer to the section of QMan CEETM traffic class flow control in the
 * reference manual.
 */

```

```
int qman_ceetm_lni_set_tcfcc(struct qm_ceetm_lni *lni,
                            unsigned int cq_level,
                            int traffic_class);
int qman_ceetm_lni_get_tcfcc(struct qm_ceetm_lni *lni,
                             unsigned int *cq_level,
                             int *traffic_class);
```

5.2.2.2.75 CEETM class queue channel

Each instance of CEETM supports 32 class queue channels for allocation across the 8 LNIs. Each channel can be configured to deliver frames to any one of the LNIs, can be configured as shaped or unshaped channel. When shaped, a dual-rate shaper applies to the aggregate of CR/ER frames from the channel. Each channel contains a total of 16 class queues (CQ), with 8 independent classes and 8 grouped classes which can be configured as 1 group of 8 classes or 2 groups of 4 classes. The weighted bandwidth fairness scheduling applies within the grouped classes, and strict priority scheduling applies to 8 independent classes and 2 class groups. Once the channel is configured as shaped, the 8 independent classes and 2 class groups can be configured to supply CR frames, ER frames or both. The channel is configured independently from other channels.

5.2.2.2.75.1 Claim/release class queue channel

```
/**
 * qman_ceetm_channel_claim - Claims an unclaimed CQ channel that is mapped to the
 * given LNI.
 * @channel: the returned class queue channel object, if successful.
 * @lni: the LNI that the channel belongs to.
 *
 * Channels are always initially "unshaped".
 *
 * Returns zero for success, or -ENODEV if there is no channel available(all 32
 * channels are claimed) or -EINVAL if the channel mapping command returns error.
 */
int qman_ceetm_channel_claim(struct qm_ceetm_channel **channel,
                             struct qm_ceetm_lni *lni);

/**
 * qman_ceetm_channel_release - Releases a previously claimed CQ channel.
 * @channel: the channel needs to be released.
 *
 * Returns zero for success, or -EBUSY if the dependencies are still
 * in use. Note any shaping of the channel will be
 * cleared to leave it in an unshaped state.
 */
int qman_ceetm_channel_release(struct qm_ceetm_channel *channel);
```

5.2.2.2.75.2 Configure the shaper of class queue channel

```
/**
 * qman_ceetm_channel_enable_shaper
 * qman_ceetm_channel_disable_shaper - Enable/Disable shaping on the given channel.
 * @channel: the given channel.
 * @coupled: indicates whether surplus tokens should be added to the excess-rate
 * token count (up to the excess-rate token limit) when the number of (unused)
 * committed-rate tokens reach the committed-rate token limit.
 *
 * Whenever a claimed channel is first enabled for shaping, its committed and
 * excess token rates and limits are zero, so will need to be changed to do
```

```

* anything useful. The shaper can subsequently be enabled/disabled without
* resetting the shaping parameters, but the shaping parameters will be reset
* when the channel is released.
*
* Returns 0 for success and -EINVAL for failure if the channel shaping has been
* enabled or disabled, or the management command returns error
*/
int qman_ceetm_channel_enable_shaper(struct qm_ceetm_channel *channel, int coupled);
int qman_ceetm_channel_disable_shaper(struct qm_ceetm_channel *channel);

/**
 * qman_ceetm_channel_is_shaper_enabled - Check channel shaper status.
 * @channel: the give channel.
 */
int qman_ceetm_channel_is_shaper_enabled(struct qm_ceetm_channel *channel);

/**
 * qman_ceetm_channel_set_commit_rate
 * qman_ceetm_channel_get_commit_rate
 * qman_ceetm_channel_set_excess_rate
 * qman_ceetm_channel_get_excess_rate - Set/get the channel shaper CR/ER token rate
 * and token limit.
 * @channel: the given channel.
 * @token_rate: the desired token rate for "set" function, or the queried token rate
 * for "get"function.
 * @token_limit: the desired token limit for "set" function, or the queried token
 * limit for "get" function.
 *
 * Returns zero for success. The "set" function returns -EINVAL if the channel
 * is unshaped or -EIO if the configure shaper command returns error. The
 * "get" function returns -EINVAL if the token rate or token limit is not set, or
 * the query shaper command returns error.
 */
int qman_ceetm_channel_set_commit_rate(struct qm_ceetm_channel *channel,
                                     const struct qm_ceetm_rate *token_rate,
                                     u16 token_limit);
int qman_ceetm_channel_get_commit_rate(struct qm_ceetm_channel *channel,
                                     struct qm_ceetm_rate *token_rate,
                                     u16 *token_limit);
int qman_ceetm_channel_set_excess_rate(struct qm_ceetm_channel *channel,
                                     const struct qm_ceetm_rate *token_rate,
                                     u16 token_limit);
int qman_ceetm_channel_get_excess_rate(struct qm_ceetm_channel *channel,
                                     struct qm_ceetm_rate *token_rate,
                                     u16 *token_limit);

/**
 * qman_ceetm_channel_set_commit_rate_bps
 * qman_ceetm_channel_get_commit_rate_bps
 * qman_ceetm_channel_set_excess_rate_bps
 * qman_ceetm_channel_get_excess_rate_bps - Set/get channel CR/ER shaper
 * parameters.
 * @channel: the given channel.
 * @token_rate: the desired shaper rate in bps for "set" function, or the
 * shaper rate in bps for "get" function.
 * @token_limit: the desired token limit for "set" function, or the queried
 * token limit for "get" function.
 *

```

```
* Return zero for success. The "set" function returns -EINVAL if the channel
* is unshaped, or -EIO if the configure shaper command returns error. The
* "get" function returns -EINVAL if token rate of token limit is not set, or
* the query shaper command returns error.
*/
int qman_ceetm_channel_set_commit_rate_bps(struct qm_ceetm_channel *channel,
                                           u64 bps, u16 token_limit);
int qman_ceetm_channel_get_commit_rate_bps(struct qm_ceetm_channel *channel,
                                           u64 *bps, u16 *token_limit);
int qman_ceetm_channel_set_excess_rate_bps(struct qm_ceetm_channel *channel,
                                           u64 bps, u16 token_limit);
int qman_ceetm_channel_get_excess_rate_bps(struct qm_ceetm_channel *channel,
                                           u64 *bps, u16 *token_limit);
```

5.2.2.2.75.3 Configure the token limit as the weight for the unshaped channel

QMan CEETM uses an algorithm called unshaped fair queuing (uFQ) for the unshaped channel. The algorithm allows unshaped channels to be included in the CR time eligible list, and thus use the configured commit rate token bucket limit value as their fair queuing weight.

```
/**
 * qman_ceetm_channel_set_weight
 * qman_ceetm_channel_get_weight - Set/get the weight for unshaped channel.
 * @channel: the given channel.
 * @token_limit: the desired token limit as the weight of unshaped channel for "set"
 * function, or the queried token limit for "get" function.
 *
 * Returns zero for success, or -EINVAL if the channel is a shaped channel, or
 * the management command returns error
 */
int qman_ceetm_channel_set_weight(struct qm_ceetm_channel *channel,
                                  u16 token_limit);
int qman_ceetm_channel_get_weight(struct qm_ceetm_channel *channel,
                                  u16 *token_limit);
```

5.2.2.2.75.4 Set CR/ER eligibility for CQs within a CEETM channel

The APIs below allow the user to set the 8 independent CQs and 2 CQ groups within a shaped CEETM channel to be CR and/or ER eligible.

```
/**
 * qman_ceetm_channel_set_group_cr_eligibility
 * qman_ceetm_channel_set_group_er_eligibility - Set channel group eligibility
 * @channel: the given channel object
 * @group_b: indicates whether there is group B in this channel.
 * @cre: the commit rate eligibility, 1 for enable, 0 for disable.
 *
 * Return zero for success, or -EINVAL if eligibility setting fails.
 */
int qman_ceetm_channel_set_group_cr_eligibility(struct qm_ceetm_channel
*channel, int group_b, int cre);
int qman_ceetm_channel_set_group_er_eligibility(struct qm_ceetm_channel
*channel, int group_b, int ere);
```

```

/**
 * qman_ceetm_channel_set_cq_cr_eligibility
 * qman_ceetm_channel_set_cq_er_eligibility - Set channel cq eligibility
 * @channel: the given channel object
 * @idx: is from 0 to 7 (representing CQ0 to CQ7).
 * @cre: the commit rate eligibility, 1 for enable, 0 for disable.
 *
 * Return zero for success, or -EINVAL if eligiblity setting fails.
 */
int qman_ceetm_channel_set_cq_cr_eligiblility(struct qm_ceetm_channel *channel,
unsigned int idx, int cre);
int qman_ceetm_channel_set_cq_er_eligiblility(struct qm_ceetm_channel *channel,
unsigned int idx, int ere);

```

5.2.2.2.76 CEETM class queue

Each CEETM class has a dedicated class queue (CQ), it can have dedicated or shared Class Congestion Group Record.

5.2.2.2.76.1 Claim/release CQ

```

/**
 * qman_ceetm_cq_claim - Claims an individual class queue.
 * @cq: the returned class queue object, if successful.
 * @channel: the given class queue channel.
 * @idx: is from 0 to 7 (representing CQ0 to CQ7).
 * @ccg: represents the class congestion group that this class queue
 * should be subscribed to, or NULL if no congestion group membership is desired.
 *
 * Returns zero for success, or -EINVAL if @idx is out of range 0 to 7 or this class
 * queue has been claimed, or configure class queue command returns error, or
 * returns -ENOMEM if allocating CQ memory fails.
 */
int qman_ceetm_cq_claim(struct qm_ceetm_cq **cq,
                      struct qm_ceetm_channel *channel,
                      unsigned int idx,
                      struct qm_ceetm_ccg *ccg);

/**
 * qman_ceetm_cq_claim_A - Claims a class queue within the channel group A.
 * @cq: the returned class queue object, if successful.
 * @channel: the given class queue channel.
 * @idx: If the channel is configured for 1 group only, @idx is from 8 to 15 (CQ8 to
 * CQ15) and only group A exists, otherwise @idx is from 8 to 11 (CQ8 to CQ11 for
 * group A).
 * @ccg: represents the class congestion group that this class queue should be
 * subscribed to, or NULL if no ccg is desired.
 *
 * Returns zero for success, or -EINVAL if @idx is out of range or if the class
 * queue has been claimed, or configure class queue command returns error, or
 * returns -ENOMEM if allocating CQ memory fails.
 */
int qman_ceetm_cq_claim_A(struct qm_ceetm_cq **cq,
                        struct qm_ceetm_channel *channel,
                        unsigned int idx,
                        struct qm_ceetm_ccg *ccg);

```

```
/**
 * qman_ceetm_cq_claim_B - Claims a class queue within the channel group B.
 * @cq: the returned class queue object, if successful.
 * @channel: the given class queue channel.
 * @idx: if the channel is configured with 2 groups, 'idx' is from 12 to 15(CQ12 to
 * CQ15 for group B).
 * @ccg: represents the class congestion group that this class queue should be
 * subscribed to, or NULL if no ccg is desired.
 *
 * Returns zero for success, or -EINVAL if @idx is out of range or the class
 * queue has been claimed, or configure class queue command returns error, or
 * returns -ENOMEM if allocating CQ memory fails.
 */
int qman_ceetm_cq_claim_B(struct qm_ceetm_cq **cq,
                        struct qm_ceetm_channel *channel,
                        unsigned int idx,
                        struct qm_ceetm_ccg *ccg);

/**
 * qman_ceetm_cq_release - Releases a previously claimed class queue.
 * @cq: the class queue to be released.
 *
 * This will only succeed if all dependent objects (eg. logical FQIDs) have been
 * released.
 * Returns zero for success or -EBUSY for failure if the dependencies are not
 * released (e.g. LFQID is not released)
 */
int qman_ceetm_cq_release(struct qm_ceetm_cq *cq);

/**
 * qman_ceetm_drain_cq - drain the CQ till it is empty.
 * @cq: the give CQ object.
 * Return 0 for success or -EINVAL for unsuccessful command to empty CQ.
 */
int qman_ceetm_drain_cq(struct qm_ceetm_cq *cq);
```

5.2.2.2.7.6.2 Change CQ weight

```
/**
 * qman_ceetm_queue_set_weight
 * qman_ceetm_queue_get_weight - Configure/query the weight of a grouped class queue.
 * @cq: the given class queue.
 * @weight_code: the desired weight code to change for this given class queue for
 * "set" function or the queried weight code of the give class queue for "get"
 * function.
 *
 * Grouped class queues have a default weight code of zero, which corresponds to
 * a scheduler weighting of 1. This function can be used to modify a grouped
 * class queue to another weight, valid values are from 0 to 255. (Use the
 * helpers qman_ceetm_wbfs2ratio() and qman_ceetm_ratio2wbfs() to convert
 * between these 'weight_code' values and the corresponding sharing weight.) As
 * the weight code ranges from 0 to 255, the corresponding scheduling weight
 * ranges from 1.00 to 248 in pseudo-exponential steps).
 *
 * Returns zero for success or -EIO if the configure weight code command returns
 * error for "set" function, or -EINVAL if the query command returns error for "get"
 * function.
```

```

* Please refer to section "CEETM Weighted Scheduling among Grouped Classes" in
* Reference Manual for weight and weight code.
*/
int qman_ceetm_queue_set_weight(struct qm_ceetm_cq *cq,
                               struct qm_ceetm_weight_code *weight_code);
int qman_ceetm_queue_get_weight(struct qm_ceetm_cq *cq,
                                struct qm_ceetm_weight_code *weight_code);

/**
 * qman_ceetm_wbfs2ratio - Given a weight code ('wbfs'), an accurate fractional
 * representation of the corresponding weight is given (in order to not lose
 * any precision).
 * @weight_code: The given weight code in WBFS.
 * @numerator: the numerator part of the weight computed by the weight code.
 * @denominator: the denominator part of the weight computed by the weight code
 *
 * Returns zero for success, or -EINVAL if the given weight code is illegal.
 */
int qman_ceetm_wbfs2ratio(struct qm_ceetm_weight_code *weight_code,
                          u32 *numerator,
                          u32 *denominator);

/**
 * qman_ceetm_ratio2wbfs - Given a weight, find the nearest possible weight code.
 * @numerator: numerator part of the given weight.
 * @denominator: denominator part of the given weight.
 * @weight_code: the weight code computed from the given weight.
 *
 * If the user needs to know how close this is, convert the resulting weight code
 * back to a weight and compare.
 *
 * Returns zero for success, or -ERANGE if "numerator/denominator" is outside the
 * range of weights.
 */
int qman_ceetm_ratio2wbfs(u32 numerator,
                          u32 denominator,
                          struct qm_ceetm_weight_code *weight_code);

/**
 * qman_ceetm_set_queue_weight_in_ratio
 * qman_ceetm_get_queue_weight_in_ratio - Configure/query the weight of a
 * grouped class queue.
 * @cq: the given class queue.
 * @ratio: the weight in ratio. It should be the real ratio number multiplied
 * by 100 to get rid of fraction. User needs to check the "WBFS weight code
 * to weight mapping" table in BG for the possible weight values to be used.
 *
 * Returns zero for success, or -EIO if the configure weight command returns
 * error for "set" function, or -EINVAL if the query command returns
 * error for "get" function.
 */
int qman_ceetm_set_queue_weight_in_ratio(struct qm_ceetm_cq *cq, u32 ratio);
int qman_ceetm_get_queue_weight_in_ratio(struct qm_ceetm_cq *cq, u32 *ratio);

```

5.2.2.2.76.3 Query CQ statistics

```
/**
 * qman_ceetm_cq_get_dequeue_statistics - Get the statistics provided by CEETM
 * CQ counters.
 * @cq: the given CQ object.
 * @flags: indicates whether the statistics counter will be cleared after query.
 * @frame_count: The number of the frames that have been counted since the
 * counter was cleared last time.
 * @byte_count: the number of bytes in all frames that have been counted.
 *
 * Return zero for success or -EINVAL if query statistics command returns error.
 */
int qman_ceetm_cq_get_dequeue_statistics(struct qm_ceetm_cq *cq, u32 flags,
                                       u64 *frame_count, u64 *byte_count);
```

5.2.2.2.77 CEETM logical FQID

Each class queue in CEETM mode is identified via a “logical frame queue identifier (LFQID)” to maintain semantic compatibility with enqueue commands to FQs (non-CEETM queues). The upper 1M FQIDs is used for the LFQID, and each DCP owns 4K LFQIDs. Any enqueue to these LFQIDs will be directed to the CEETM logic used by one of the DCP portals.

5.2.2.2.77.1 Claim/release LFQID

```
/**
 * qman_ceetm_lfq_claim - Claims an unused logical FQID, associates it with the
 * given class queue.
 * @lfq: the returned lfq object, if successful.
 * @cq: the given class queue which needs to claim a LFQID.
 *
 * Returns 0 for success, or -ENODEV if no LFQID is available, or -ENOMEM if
 * allocating memory for lfq fails, or -EINVAL if configuring LfqMT fails
 */
int qman_ceetm_lfq_claim(struct qm_ceetm_lfq **lfq,
                        struct qm_ceetm_cq *cq);

/**
 * qman_ceetm_lfq_release - Releases a previously claimed logical FQID.
 * @lfq: the logic fq to be released.
 *
 * Return zero for success. The failure condition is TBD.
 */
int qman_ceetm_lfq_release(struct qm_ceetm_lfq *lfq);
```

5.2.2.2.77.2 Configure/Query Dequeue Context Table

```
/**
 * qman_ceetm_lfq_set_context
 * qman_ceetm_lfq_get_context - Set/get the context_a/context_b pair to the
```

```

* "dequeue context table" associated with the logical FQID.
* @lfq: the given lfq object.
* @context_a: contextA of the dequeue context.
* @context_b: contextB of the dequeue context.
*
* Returns zero for success, or -EINVAL if there is error to set/get the context
* pair.
*/
int qman_ceetm_lfq_set_context(struct qm_ceetm_lfq *lfq,
                             u64 context_a,
                             u32 context_b);
int qman_ceetm_lfq_get_context(struct qm_ceetm_lfq *lfq,
                              u64 *context_a,
                              u32 *context_b);

```

5.2.2.2.773 Create FQ for LFQ

```

/**
 * qman_ceetm_create_fq - Initialise a FQ object for the LFQ.
 * @lfq: the given logic FQ.
 * @fq: the FQ object created for the LFQ
 *
 * The FQ object can be used in qman_enqueue() and qman_enqueue_orp() APIs to target
 * a logical FQID (and the class queue it is associated with). Note that this FQ
 * object can only be used for enqueues, and in the case of qman_enqueue_orp()
 * it can not be used as the 'orp' parameter, only as 'fq'. This FQ object can not
 * (and shouldn't) be destroyed, it is only valid as long as the underlying 'lfq'
 * remains claimed. It is the user's responsibility to ensure that the underlying
 * 'lfq' is not released until any enqueues to this FQ object have completed.
 * The only field the user needs to fill in is fq->cb.ern, as that enqueue rejection
 * handler is the callback that could conceivably be called on this FQ object. This
 * API can be called multiple times to create multiple FQ objects referring to the
 * same logical FQID, and any enqueue rejections will respect the callback of the
 * object that issued the enqueue (and will identify the object via the parameter
 * passed to the callback too). There is no 'flags' parameter to this API as there
 * is for qman_create_fq() - the created FQ object behaves as though
 * qman_create_fq() had been called with the single flag QMAN_FQ_FLAG_NO_MODIFY.
 *
 * Returns 0 for success. The failure case is TBD
 */
int qman_ceetm_create_fq(struct qm_ceetm_lfq *lfq, struct qman_fq *fq);

```

5.2.2.2.78 CEETM Class Congestion Group(CCG)

CEETM supports both Weighted Random Early Discard (WRED) and tail drop congestion management with its 512 Class Congestion Groups (CCGs). The CCG are divided into channels. Each channel contains 16 CCGs, and each CCG channel is tied one-to-one with a CQ channel. The CCG to be used for a particular CQ can be assigned to any of the 16 CCG within the channel. The description of CCG can be found at section “CEETM Class Congestion Management and Avoidance” in the Reference Manual.

5.2.2.2.78.1 Claim/release CCG

```

/**
 * qman_ceetm_ccg_claim - Claims an unused CCG.

```

```
* @ccg: the returned CCG object, if successful.
* @idx: the ccg index from 0-15 within the given channel.
* @channel: the given class queue channel which tied to the CCG channel.
* @cscn: the callback function of this CCG.
* @cb_ctx: specify the callback and corresponding context to be used if state
* change notifications are later enabled for this CCG.
*
* The congestion group is local to the given class queue channel, so only
* class queues within the channel can be associated with that congestion
* group. The association of class queues to congestion groups occurs when the
* class queues are claimed, see qman_ceetm_cq_claim() and related functions.
* Congestion groups are in a "zero" state when initially claimed, and they are
* returned to that state when released.
*
* Returns 0 for success, or -EINVAL if no CCG is available.
*/
int qman_ceetm_ccg_claim(struct qm_ceetm_ccg **ccg,
                       struct qm_ceetm_channel *channel,
                       void (*cscn)(struct qm_ceetm_ccg *,
                                     void *cb_ctx,
                                     int congested),
                       void *cb_ctx);

/**
 * qman_ceetm_ccg_release - Releases a previously claimed CCG.
 * ccg: the CCG to be released.
 *
 * Returns zero for success, or -EBUSY if the given CCG's dependent objects(class
 * queues that are associated with the CCG) have not been released.
 */
int qman_ceetm_ccg_release(struct qm_ceetm_ccg *ccg);
```

5.2.2.2.78.2 Configure CCG

```
/* This struct is used to specify attributes for a CCG. The 'we_mask' field
 * controls which CCG attributes are to be updated, and the remainder specify
 * the values for those attributes. A CCG counts either frames or the bytes
 * within those frames, but not both ('mode'). A CCG can optionally cause
 * enqueues to be rejected, due to tail-drop or WRED, or both (they are
 * independent options, 'td_en' and 'wr_en_g,wr_en_y,wr_en_r'). Tail-drop can be
 * level-triggered due to a single threshold ('td_thres') or edge-triggered due
 * to a "congestion state", but not both ('td_mode'). Congestion state has
 * distinct entry and exit thresholds ('cs_thres_in' and 'cs_thres_out'), and
 * notifications can be sent to software the CCG goes in to and out of this
 * congested state ('cscn_en').
 * The struct qm_cgr_cs_thres has already been defined in fsl_qman.h
 */
struct qm_ceetm_ccg_params {
    /* Boolean fields together in a single bitfield struct */
    struct {
        /* Whether to count bytes or frames. 1==frames */
        int mode:1;
        /* En/disable tail-drop. 1==enable */
        int td_en:1;
        /* Tail-drop on congestion-state or threshold. 1=threshold */
        int td_mode:1;
        /* Generate congestion state change notifications. 1==enable */
        int cscn_en:1;
    };
};
```

```

        /* Enable WRED rejections (per colour). 1==enable */
        int wr_en_g:1;
        int wr_en_y:1;
        int wr_en_r:1;
    } __packed;
    /* Tail-drop threshold. See qm_cgr_thres_[gs]et64(). */
    struct qm_cgr_cs_thres td_thres;
    /* Congestion state thresholds, for entry and exit. */
    struct qm_cgr_cs_thres cs_thres_in;
    struct qm_cgr_cs_thres cs_thres_out;
    /* Overhead accounting length. Per-packet "tax", from -128 to +127 */
    signed char oal;
    /* WRED parameters. */
    struct qm_cgr_wr_parm wr_parm_g;
    struct qm_cgr_wr_parm wr_parm_y;
    struct qm_cgr_wr_parm wr_parm_r;
};

/* Bits used in 'we_mask' to qman_ceetm_ccg_set(), controls which attributes of
 * the CCG are to be updated. */
#define QM_CCGR_WE_CDV      0x0000 /* cdv */
#define QM_CCGR_WE_MODE    0x0001 /* mode (bytes/frames) */
#define QM_CCGR_WE_TD_EN   0x0004 /* congestion state tail-drop enable */
#define QM_CCGR_WE_TD_MODE 0x4000 /* tail-drop mode (state/threshold) */
#define QM_CCGR_WE_TD_THRES 0x2000 /* tail-drop threshold */
#define QM_CCGR_WE_CS_THRES_IN 0x0002 /* congestion state entry threshold */
#define QM_CCGR_WE_CS_THRES_OUT 0x1000 /* congestion state exit threshold */
#define QM_CCGR_WE_CSCN_EN 0x0010 /* congestion notification enable */
#define QM_CCGR_WE_CSCN_TUPD 0x0008 /* CSCN target update */
#define QM_CCGR_WE_OAL     0x0800 /* overhead accounting length */
#define QM_CCGR_WE_WR_PARM_G 0x0400 /* WRED parameters - green */
#define QM_CCGR_WE_WR_PARM_Y 0x0200 /* WRED parameters - yellow */
#define QM_CCGR_WE_WR_PARM_R 0x0100 /* WRED parameters - red */
#define QM_CCGR_WE_WR_EN_G   0x0080 /* WRED enable - green */
#define QM_CCGR_WE_WR_EN_Y   0x0040 /* WRED enable - yellow */
#define QM_CCGR_WE_WR_EN_R   0x0020 /* WRED enable - red */

/**
 * qman_ceetm_ccg_set
 * qman_ceetm_ccg_get - Configure/query a subset of CCG attributes.
 * @ccg: the given CCG.
 * @we_mask: the write enable mask for the CCG attributes.
 * @params: the parameters set for this CCG.
 *
 * Returns zero for success, or -EINVAL if there is error for this CCG setting.
 */
int qman_ceetm_ccg_set(struct qm_ceetm_ccg *ccg,
                      u32 we_mask,
                      const struct qm_ceetm_ccg_params *params);
int qman_ceetm_ccg_get(struct qm_ceetm_ccg *ccg,
                      struct qm_ceetm_ccg_params *params);

```

5.2.2.2.78.3 Query CCG statistics

```

/**
 * qman_ceetm_ccg_get_reject_statistics - Get the statistics provided by
 * CEETM CCG counters.
 * @ccg: the given CCG object.
 * @flags: indicates whether the statistics counter will be cleared after query.

```

```
* @frame_count: The number of the frames that have been counted since the
* counter was cleared last time.
* @byte_count: the number of bytes in all frames that have been counted.
*
* Return zero for success or -EINVAL if query statistics command returns error.
*
*/
int qman_ceetm_ccg_get_reject_statistics(struct qm_ceetm_ccg *ccg, u32 flags,
                                       u64 *frame_count, u64 *byte_count);
```

5.2.2.2.7.8.4 Set/get Congestion State Change Notification Target

```
/** qman_ceetm_cscn_swp_get - Query whether a given software portal index is
* in the cscn target mask.
* @ccg: the give CCG object.
* @swp_idx: the index of the software portal.
* @cscn_enabled: 1: cscn is enabled in this swp. 0: cscn is not enabled
* in this swp.
*
* Return 0 for success, or -EINVAL if command in set/get function fails.
*/
int qman_ceetm_cscn_swp_get(struct qm_ceetm_ccg *ccg,
                           u16 swp_idx,
                           unsigned int *cscn_enabled);

/** qman_ceetm_cscn_dcp_set - Add or remove a direct connect portal from the\
* target mask.
* qman_ceetm_cscn_swp_get - Query whether a given direct connect portal index
* is in the cscn target mask.
* @ccg: the give CCG object.
* @dcp_idx: the index of the direct connect portal.
* @cscn_enabled: 1: Set the dcp to be cscn target. 0: remove the dcp from
* the target mask.
*
* Return 0 for success, or -EINVAL if command in set/get function fails.
*/
int qman_ceetm_cscn_dcp_set(struct qm_ceetm_ccg *ccg,
                           u16 dcp_idx,
                           unsigned int cscn_enabled);

int qman_ceetm_cscn_dcp_get(struct qm_ceetm_ccg *ccg,
                           u16 dcp_idx,
                           unsigned int *cscn_enabled);
```

5.2.2.2.8 Other QMan APIs

The following sections describe the interfaces provided by the QMan driver for manipulating QMan device.

5.2.2.2.8.1 Waterfall Power Management

Waterfall power management is a mechanism that works between the QMan and the e6500's Drowsy Core mode. The basic idea is that when using multiple cores and pool channels to forward frames, and the QMan receives less traffic than required to keep all cores busy, it no longer assigns to the higher numbered cores (i.e. software portals) in a pool. This feature is supported from QMan3.0, and the APIs below can only be called at SoC with QMan3.0

```
/**
* qman_set_wpm - Set waterfall power management
```

```

*
* @wpm_enable: boolean, 1 = enable wpm, 0 = disable wpm.
*
* Return 0 for success, return -ENODEV if QMan misc_cfg register is not
* accessible.
*/
int qman_set_wpm(int wpm_enable);
/**
 * qman_get_swp - Query the waterfall power management setting
 *
 * @wpm_enable: boolean, 1 = enable wpm, 0 = disable wpm.
 *
 * Return 0 for success, return -ENODEV if QMan misc_cfg register is not
 * accessible.
 */
int qman_get_wpm(int *wpm_enable);

```

5.2.2.2.9 USDPAA-specific APIs

5.2.2.2.9.1 Overview

The USDPAA SDK includes a port of the QMan and BMan drivers to linux user space, and assumes a pthreads-based programming model, and the use of a single application/process instance.

Unlike conventional user space interfaces to hardware (in which the kernel manipulates hardware interfaces on behalf of user space processes), the USDPAA QMan and BMan drivers operate on the hardware directly from user space. The underlying mechanisms for this are based on the USDPAA Linux character device. Put briefly, the kernel exposes the devices to user space as character devices with various attributes, which allow the user space drivers to `mmap()` the Corenet portal regions directly. Interrupt handling works by disabling the portal interrupt when it asserts and conveying the interrupt event to user-space via the USDPAA device's file-descriptor (the user-space portal driver can re-enable the interrupt whenever it so chooses).

The key distinction between the QMan (or BMan) drivers found in the Linux kernel and the one in USDPAA is that the latter does not perform a global initialization step to parse all available portals and initialize and assign them to CPUs/threads. The model in USDPAA assumes that the application is responsible for creating its own pthreads, and is responsible for making those threads affine to the appropriate CPUs (if indeed it chooses to). Such applications simply call into a USDPAA-specific API to “enable” the thread for QMan/BMan portal usage (specifying the desired CPU-affinity for the portal), and the USDPAA driver at that point will “find” an unused portal suitable for the requested CPU and initialize and bind it to the pthread via thread-local storage, or fail if no such portal was available. Portals can likewise be released from threads when they are no longer required, without requiring the pthread itself to be destroyed.

Not only do both the Linux kernel and USDPAA drivers initialize all portals at start-up, but their portals are managed as CPU-*local* associations. USDPAA on the other hand manages portals as *thread-local* associations. Nonetheless, portals are (typically) configured for a particular CPU-affinity, meaning that ring and data stashing will target the designated CPU, so USDPAA applications are advised to run their threads on the CPUs for which their portals are configured - failing to do so will not break the system nor the application, but will yield significant performance degradation relative to an optimized system.

NOTE

The "recovery_mode" parameters in the following APIs are currently unsupported, they are reserved until a future release, so should always be set to 0 (or "FALSE").

5.2.2.2.9.2 Thread initialization

An application pthread can request that the QMan drivers initialize and bind a portal for use by that thread by calling the following APIs. This API must be called and return success prior to calling any of the interfaces mentioned in [QMan portal](#)

[APIs](#) on page 120 or [BMan CoreNet portal APIs](#) on page 106. Note, this interface is dependent on the device-tree layer having been initialized, see [Device-tree dependency](#) on page 155.

```
int qman_thread_init(void);
int qman_thread_init_idx(uint32_t idx);
int bman_thread_init(void);
int bman_thread_init_idx(uint32_t idx);
```

Note that the 'idx' parameter influences the driver's search for the portal with the specified portal index. The pthread is required to already be affine to the given CPU.

As usual with "int"-valued APIs, a return value of zero indicates success, otherwise a standard negative error number is returned in event of failure.

Likewise, the thread-portal association can be ended (and thus make the underlying portal available for another user/thread) by calling the following APIs.

```
int qman_thread_finish(void);
int bman_thread_finish(void);
```

5.2.2.2.9.3 FQID/BPID allocation

As of the current release of USDPAA, support for allocation of FQIDs and BPIDs is via hard-coded ranges encoded within the drivers. This will be revised in a future release.

To enable the allocation mechanisms in the USDPAA QMan (and BMan) drivers, the following APIs must be called once from a thread that has already successfully bound to a QMan (or BMan) portal via `qman_thread_init()` (or `bman_thread_init()`, respectively).

```
int qman_global_init(int recovery_mode);
int bman_global_init(int recovery_mode);
```

Note that this API does not need to be called from all threads, only from one, but that it should be called before using any QMan/BMan APIs that require dynamic allocation of FQIDs or BPIDs.

5.2.2.2.9.4 Interrupt handling

5.2.2.2.9.4.1 USDPAA file-descriptors

Interrupt handling in USDPAA is done by reading a standard file descriptor that is created when a portal is assigned to a thread. The application should use the `poll()` or `select()` API to determine when the file descriptor becomes readable which indicates an interrupt has occurred. The following APIs return the current USDPAA file-descriptors for portals bound to the calling thread.

```
int qman_thread_fd(void);
int bman_thread_fd(void);
```

5.2.2.2.9.4.2 Processing interrupt-driven portal duties

It should be noted that the QMan and BMan drivers allow applications to dynamically configure which portal "duties" are to be triggered and performed by interrupts versus polling. See [Modifying interrupt-driven portal duties \(BMan\)](#) on page 106 and [Modifying interrupt-driven portal duties \(QMan\)](#) on page 121 for details. Prior to going into a blocking `read()`, `select()`, `poll()`, [...] on the file-descriptor, the application will need to ensure that the "duties" it wishes to wait for are put into IRQ mode (ie. added to the portal's "irqsource") prior to blocking, otherwise the presence of such work will not trigger any interrupt and so will not wake up the sleeping process. Likewise, if going back into polling mode and expecting polling APIs to process such portal duties, the application will need to remove such duties from the "irqsource".

For portal duties that are in the "irqsource", and after any USDPAA-supported file-descriptor operation that indicates that an interrupt was received, the application pthread can call the following APIs to post-process any such interrupt-driven duties:

```
/* Post-process interrupts. NB, the kernel IRQ handler disables the interrupt
 * line before notifying us, and this post-processing re-enables it once
 * processing is complete. As such, it is essential to call this before going
 * into another blocking read/select/poll. */
void qman_thread_irq(void);
void bman_thread_irq(void);
```

5.2.2.2.9.5 Device-tree dependency

The QMan/BMan drivers “find” portals by referring to information extracted from the device-tree, as represented in the procs filesystem located at /proc/device-tree. This information is handled by a USDPAA “of” driver, which must be initialised before any attempts are made to initialise threads for QMan/BMan use as described in [Thread initialization](#) on page 153. As of the current USDPAA release, the “of” driver must be explicitly initialised by applications prior to QMan/BMan thread initialisation, though this may change in future releases. (As a side note, other USDPAA mechanisms, such as application-side configuration parsing may also be dependent on this device-tree layer, and so the “of” init should occur prior to all such dependencies.)

The API to initialise the “of” driver is of_init(), which parses the /proc/device-tree representation into an internal representation of C data-structures.

```
#ifndef OF_INIT_DEFAULT_PATH
#define OF_INIT_DEFAULT_PATH "/proc/device-tree"
#endif
int of_init_path(const char *dt_path);
/* Use of this wrapper is recommended. */
static inline int of_init(void)
{
    return of_init_path(OF_INIT_DEFAULT_PATH);
}
```

Conversely, the of_finish() API cleans up all data structures and handles created by of_init(), which may be useful to satisfy leak-checking tools, or persistent process/thread recycling schemes.

```
void of_finish(void);
```

5.2.2.2.10 Sysfs and debugfs QMan/BMan interfaces

This chapter describes the qman and bman interfaces available via sysfs and debugfs.

NOTE

For non-P4080 devices, it is generally recommended to determine these from the device-tree, the SoC-specific Reference Manual, and/or by examining the sysfs filesystem at run-time.

5.2.2.2.10.1 QMan sysfs

5.2.2.2.10.1.1 /sys/devices/ffe000000.soc/ffe318000.qman

Description:

This directory contains a snapshot of the internal state of the qman device.

5.2.2.2.10.1.2 /sys/devices/ffe000000.soc/ffe318000.qman/error_capture

Description:

This directory contains a snapshot of error related qman attributes.

5.2.2.2.10.1.3 /sys/devices/ffe000000.soc/ffe318000.qman/error_capture/sbec_<0..6>

Description:

Provides a count of the number of single bit ECC errors that have occurred when reading from one of the QMan internal memories. The range <0..6> represent a QMAN internal memory region defined as follows:

- 0: FQD cache memory
- 1: FQD cache tag memory
- 2: SFDR memory
- 3: WQ context memory
- 4: Congestion Group Record memory
- 5: Internal Order Restoration List memory
- 6: Software Portal ring memory

This file is read-reset.

5.2.2.2.10.1.4 /sys/devices/ffe000000.soc/ffe318000.qman/sfdr_in_use

Description:

Reports the number of SFDR currently in use. The minimum value is 1. This file is not available on Rev 1.0 of P4080 QorIQ.

This file is read-only

/sys/devices/ffe000000.soc/ffe318000.qman/pfdr_fpc

Description:

Total Packed Frame Descriptor Record Free Pool Count in external memory.

This file is read-only

5.2.2.2.10.1.5 /sys/devices/ffe000000.soc/ffe318000.qman/pfdr_cfg

Description:

Used to read the configuration of the dynamic allocation policy for PFDRs. The value is used to account for PFDR that may be required to complete any currently executing operations in the sequencers.

This file is read-only.

5.2.2.2.10.1.6 /sys/devices/ffe000000.soc/ffe318000.qman/idle_stat

Description:

This file can be used to determine when QMan is both idle and empty. The possible values are:

- 0: All work queues in QMan are NOT empty and QMan is NOT idle.
- 1: All work queues in QMan are NOT empty and QMan is idle.
- 2: All work queues in QMan are empty
- 3: All work queues in QMan are empty and QMan is idle.

This file is read-only.

5.2.2.2.10.1.7 */sys/devices/ffe000000.soc/ffe318000.qman/err_isr*

Description:

QMan contains one dedicated interrupt line for signaling error conditions to software. This file identifies the source of the error interrupt within QMan. The value is displayed in hexadecimal format. Refer to the appropriate QorIQ SOC Reference Manual for a description of the QMAN_ERR_ISR register.

This file is read-only.

5.2.2.2.10.1.8 */sys/devices/ffe000000.soc/ffe318000.qman/dcp<0..3>_dlm_avg*

Description:

These files contain an EWMA (exponentially weighted moving average) of dequeue latency samples for dequeue commands received on the sub portal. The range <0..3> refers to each of the direct-connect portals. The display format is as follows: <avg_interger>.<avg_fraction>

This file can be seeded with a interger value. The input interger is processed in the following manner: <avg_fraction> = lowest 8 bits / 256 , <avg_interger> = next 12 bits

ex: echo 0x201 > dcp0_dlm_avg

```
cat dcp0_dlm_avg
```

```
0.00390625
```

This file is read-write

5.2.2.2.10.1.9 */sys/devices/ffe000000.soc/ffe318000.qman/ci_rlm_avg*

Description:

This file contains an EWMA (exponentially weighted moving average) of read latency samples for reads on CoreNet initiated by QMan. The display format is as follows: <avg_interger>.<avg_fraction>

This file can be seeded with a interger value. The input interger is processed in the following manner: <avg_fraction> = lowest 8 bits / 256 , <avg_interger> = next 12 bits

ex: echo 0x201 > ci_rlm_avg

```
cat ci_rlm_avg
```

```
0.00390625
```

This file is read-write

5.2.2.2.10.2 BMan sysfs

5.2.2.2.10.2.1 */sys/devices/ffe000000.soc/ffe31a000.bman*

Description:

This directory contains a snapshot of the internal state of the BMan device.

5.2.2.2.10.2.2 */sys/devices/ffe000000.soc/ffe31a000.bman/error_capture*

Description:

This directory contains a snapshot of error related BMan attributes.

5.2.2.2.10.2.3 */sys/devices/ffe000000.soc/ffe31a000.bman/error_capture/sbec_<0..1>*

Description:

Provides a count of the number of single bit ECC errors that have occurred when reading from one of the BMan internal memories. The range <0..1> represent a BMAN internal memory region defined as follows:

- 0: Stockpile memory 0
- 1: Software Portal ring memory

This file is read-reset.

5.2.2.2.10.2.4 /sys/devices/ffe000000.soc/ffe31a000.bman/pool_count

Description:

This directory contains a snapshot of the number of free buffers available in any of the buffer pools.

5.2.2.2.10.2.5 /sys/devices/ffe000000.soc/ffe31a000.bman/fbpr_fpc

Description:

This file returns a snapshot of the Free Buffer Proxy Record free pool size. Total Free Buffer Proxy Record Free Pool Count in external memory.

This file is read-only

5.2.2.2.10.2.6 /sys/devices/ffe000000.soc/ffe31a000.bman/err_isr

Description:

BMan contains one dedicated interrupt line for signaling error conditions to software. This file identifies the source of the error interrupt within BMan. The value is displayed in hexadecimal format. Refer to the appropriate QorIQ SOC Reference Manual for a description of the BMAN_ERR_ISR register.

This file is read-only.

5.2.2.2.10.3 QMan debugfs

5.2.2.2.10.3.1 /sys/kernel/debug/qman

Description:

This directory contains various QMan device debugging attributes.

5.2.2.2.10.3.2 /sys/kernel/debug/qman/query_cgr

Description:

Query the entire contents of a Congestion Group Record. The file takes as input the Congestion Group Record ID. The output of the file returns the various CGR fields.

For example, if we want to query cgr_id 10 we would do the following:

```
# echo 10 > query_cgr
```

```
# cat query_cgr
```

Query CGR id 0xa

```
wr_parm_g MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0
```

```
wr_parm_y MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0
```

```
wr_parm_r MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0
```

```
wr_en_g: 0, wr_en_y: 0, we_en_r: 0
```

```
cscn_en: 0
```

```
cscn_targ: 0
```

```
cstd_en: 0
cs: 0
cs_thresh_TA: 0, cs_thresh_Tn: 0
i_bcmt: 0
a_bcmt: 0
```

5.2.2.2.10.3.3 */sys/kernel/debug/qman/query_congestion*

Description:

Query the state of all 256 Congestion Groups in QMan. This is a read-only file. The output of the file returns the state of all congestion group records. The state of a congestion group is either "in congestion" or "not in congestion". Since CGR are normally not in congestion, only CGR which are in congestion are returned. If no CGR are in congestion, then this is indicated.

For example, if we want to perform a query we would do the following:

```
# cat query_congestion
```

```
Query Congestion Result
```

```
All congestion groups not congested.
```

5.2.2.2.10.3.4 */sys/kernel/debug/qman/query_fq_fields*

Description:

Query the frame queue programmable fields. This file takes as input the frame queue id to be queried on a subsequent read. The output of this file returns all the frame queue programmable fields. The default frame queue id is 1.

Refer to the appropriate QorIQ SOC Reference Manual for detailed explanation on the return values.

For example, if we determine that our application is using frame queue 482 we could use this file in the following manner:

```
# echo 482 > query_fq_fields
```

```
# cat query_fq_fields
```

```
Query FQ Programmable Fields Result fqid 0x1e2
```

```
orprws: 0
```

```
oa: 0
```

```
olws: 0
```

```
cgid: 0
```

```
fq_ctrl:
```

```
Aggressively cache FQ
```

```
Don't block active
```

```
Context-A stashing
```

```
Tail-Drop Enable
```

```
dest_channel: 33
```

```
dest_wq: 7
```

```
ics_cred: 0
```

```
td_mant: 128
```

```
td_exp: 7
```

```
ctx_b: 0x19e
```

ctx_a: 0x78b59e18

ctx_a_stash_exclusive:

FQ Ctx Stash

Frame Annotation Stash

ctx_a_stash_annotation_cl: 1

ctx_a_stash_data_cl: 2

ctx_a_stash_context_cl: 2

5.2.2.2.10.3.5 /sys/kernel/debug/qman/query_fq_np_fields

Description:

Query the frame queue non programmable fields. This file takes as input the frame queue id to be queried on a subsequent read. The output of this file returns all the frame queue non programmable fields. The default frame queue id is 1.

Refer to the appropriate QorIQ SOC Reference Manual for detailed explanation on the return values.

For example, if we determine that our application is using frame queue 482 we could use this file in the following manner:

```
# echo 482 > query_fq_np_fields
```

```
# cat query_fq_np_fields
```

Query FQ Non Programmable Fields Result fqid 0x1e2

force eligible pending: no

retirement pending: no

state: Out of Service

fq_link: 0x0

odp_seq: 0

orp_nesn: 0

orp_ea_hseq: 0

orp_ea_tseq: 0

orp_ea_hptr: 0x0

orp_ea_tptr: 0x0

pfd_r_hptr: 0x0

pfd_r_tptr: 0x0

is: ics_surp contains a surplus

ics_surp: 0

byte_cnt: 0

frm_cnt: 0

ra1_sfdr: 0x0

ra2_sfdr: 0x0

od1_sfdr: 0x0

od2_sfdr: 0x0

od3_sfdr: 0x0

5.2.2.2.10.3.6 /sys/kernel/debug/qman/query_cq_fields

Description:

Query all the fields of in a particular CQD. This file takes input as the DCP id plus the class queue id to be queried on a subsequent read. The output of this file returns all the class queue fields. The default class queue id is 1 of DCP 0

Refer to the appropriate QorIQ SOC Reference Manual for detailed explanation on the return values.

For example, if we determine that our application is using class queue 4 of DCP 1, we could use this file in the following manner:

```
# echo 0x01000004 > query_cq_fields
```

(The most left 8 bits are used to specify DCP id, and the rest of 24 bits are used to specify the class queue id)

```
# cat query_cq_fields
```

```
Query CQ Fields Result cqid 0x4 on DCP 1
```

```
ccqid: 4
```

```
state: 0
```

```
pfd_r_hptr: 0
```

```
pfd_r_tptr: 0
```

```
od1_xsfdr: 0
```

```
od2_xsfdr: 0
```

```
od3_xsfdr: 0
```

```
od4_xsfdr: 0
```

```
od5_xsfdr: 0
```

```
od6_xsfdr: 0
```

```
ra1_xsfdr: 0
```

```
ra2_xsfdr: 0
```

```
frame_count: 0
```

5.2.2.2.10.3.7 /sys/kernel/debug/qman/query_ceetm_ccgr

Description:

Query the configuration and state fields within a CEETM Congestion Group Record that relate to congestion management(CM). This file takes input as the DCP id(most left 8 bits) and CEETM Congestion Group Record ID(most right 24 bits). The output of the file returns the various CCGR fields.

For example, if we want to query ccgr_id 7 of DCP 0, we would do the following:

```
# echo 0x00000007 > query_ceetm_ccgr
```

```
# cat query_ceetm_ccgr
```

```
Query CCGID 7
```

```
Query CCGR id 7 in DCP 0
```

```
wr_parm_g MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0
```

```
wr_parm_y MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0
```

```
wr_parm_r MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0
```

```
wr_en_g: 0,
```

wr_en_y: 0,
we_en_r: 0
cscn_en: 0
cscn_targ_dcp:
cscn_targ_swp:
td_en: 0
cs_thresh_in_TA: 0,
cs_thresh_in_Tn: 0
cs_thresh_out_TA: 0,
cs_thresh_out_Tn: 0
td_thresh_TA: 0,
td_thresh_Tn: 0
mode: byte count
i_cnt: 0
a_cnt: 0

5.2.2.2.10.3.8 /sys/kernel/debug/qman/query_wq_lengths

Description:

Query the length of the Work Queues in a particular channel. This file takes as input a specified channel id. The output of this file returns the lengths of the work queues on the specified channel.

For example, if we want to query channel 1 we would do the following:

```
# echo 1 > query_wq_lengths
```

```
# cat query_wq_lengths
```

Query Result For Channel: 0x1

```
wq0_len : 0  
wq1_len : 0  
wq2_len : 0  
wq3_len : 0  
wq4_len : 0  
wq5_len : 0  
wq6_len : 0  
wq7_len : 0
```

5.2.2.2.10.3.9 /sys/kernel/debug/qman/fqd/avoid_blocking_[enable | disable]

Description:

Query Avoid_Blocking bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Avoid_Blocking bit mask enabled or disabled.

For example, if we want to find all frame queues with Avoid_Blocking enabled, we would do the following:

```
# cat avoid_blocking_enable
List of fq ids with: Avoid Blocking :enabled
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001de,0x0001df,0x0001e0,0x0001e1,
0x0001ea,0x0001eb,0x0001ec,0x0001ed,0x0001f6,0x0001f7,0x0001f8,0x0001f9,
...
Total FQD with: Avoid Blocking : enabled = 528
Total FQD with: Avoid Blocking : disabled = 32239
```

5.2.2.2.10.3.10 /sys/kernel/debug/qman/fqd/prefer_in_cache_[enable | disable]

Description:

Query Prefer_in_Cache bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Prefer_in_Cache bit mask enabled or disabled.

For example, if we want to find all frame queues with Prefer_in_Cache enabled, we would do the following:

```
# cat prefer_in_cache_enable
List of fq ids with: Prefer in cache :enabled
0x0001ca,0x0001cb,0x0001cc,0x0001cd,0x0001ce,0x0001cf,0x0001d0,0x0001d1,
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001d6,0x0001d7,0x0001d8,0x0001d9,
...
Total FQD with: Prefer in cache : enabled = 560
Total FQD with: Prefer in cache : disabled = 32207
```

5.2.2.2.10.3.11 /sys/kernel/debug/qman/fqd/cge_[enable | disable]

Description:

Query Congestion_Group_Enable bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Congestion_Group_Enable bit mask enabled or disabled.

For example, if we want to find all frame queues with Congestion_Group_Enable disabled, we would do the following:

```
# cat cge_disable
List of fq ids with: Congestion Group Enable :disabled
0x000001,0x000002,0x000003,0x000004,0x000005,0x000006,0x000007,0x000008,
0x000009,0x00000a,0x00000b,0x00000c,0x00000d,0x00000e,0x00000f,0x000010,
...
Total FQD with: Congestion Group Enable : enabled = 0
Total FQD with: Congestion Group Enable : disabled = 32767
```

5.2.2.2.10.3.12 /sys/kernel/debug/qman/fqd/cpc_[enable | disable]

Description:

Query CPC_Stash_Enable bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their CPC_Stash_Enable bit mask enabled or disabled.

For example, if we want to find all frame queues with CPC Stash disabled, we would do the following:

```
# cat cpc_disable
List of fq ids with: CPC Stash Enable :disabled
0x000001,0x000002,0x000003,0x000004,0x000005,0x000006,0x000007,0x000008,
```

```
0x000009,0x00000a,0x00000b,0x00000c,0x00000d,0x00000e,0x00000f,0x000010,  
...  
Total FQD with: CPC Stash Enable : enabled = 0  
Total FQD with: CPC Stash Enable : disabled = 32767
```

5.2.2.2.10.3.13 /sys/kernel/debug/qman/fqd/cred

Description:

Query Intra-Class Scheduling bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma separated list, which have their Intra-Class Scheduling Credit value greater than 0.

```
# cat cred  
List of fq ids with Intra-Class Scheduling Credit > 0  
Total FQD with ics_cred > 0 = 0
```

5.2.2.2.10.3.14 /sys/kernel/debug/qman/fqd/ctx_a_stashing_[enable | disable]

Description:

Query Context_A bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma separated list, which have their Context_A bit mask enabled or disabled.

For example, if we want to find all frame queues with Context_A enabled, we would do the following:

```
# cat ctx_a_stashing_enable  
List of fq ids with: Context-A stashing :enabled  
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001de,0x0001df,0x0001e0,0x0001e1,  
0x0001ea,0x0001eb,0x0001ec,0x0001ed,0x0001f6,0x0001f7,0x0001f8,0x0001f9,  
...  
Total FQD with: Context-A stashing : enabled = 528  
Total FQD with: Context-A stashing : disabled = 32239
```

5.2.2.2.10.3.15 /sys/kernel/debug/qman/fqd/hold_active_[enable | disable]

Description:

Query Hold_Active bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma separated list, which have their Hold_Active bit mask enabled or disabled.

For example, if we want find all frame queues with Hold_Active enabled, we would do the following:

```
# cat hold_active_enable  
List of fq ids with: Hold active in portal :enabled  
Total FQD with: Hold active in portal : enabled = 0  
Total FQD with: Hold active in portal : disabled = 32767
```

5.2.2.2.10.3.16 /sys/kernel/debug/qman/fqd/orp_[enable | disable]

Description:

Query ORP bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma separated list, which have their ORP bit mask enabled or disabled.

For example, if we want find all frame queues with ORP enabled, we would do the following:

```
# cat orp_enable
List of fq ids with: ORP Enable :enabled
Total FQD with: ORP Enable : enabled = 0
Total FQD with: ORP Enable : disabled = 32767
```

5.2.2.2.10.3.17 */sys/kernel/debug/qman/fqd/sfdr_[enable | disable]*

Description:

Query Force_SFDR_Allocate bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Force_SFDR_Allocate bit mask enabled or disabled.

For example, if we want to find all frame queues with Force_SFDR_Allocate enabled, we would do the following:

```
# cat sfdr_enable
List of fq ids with: High-priority SFDRs :enabled(1)
Total FQD with: High-priority SFDRs : enabled = 0
Total FQD with: High-priority SFDRs : disabled = 32767
```

5.2.2.2.10.3.18 *sys/kernel/debug/qman/fqd/state_[active | oos | parked | retired | tentatively_sched | truly_sched]*

Description:

Query Frame Queue State in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which are in the specified state: active, oos, parked, retired, tentatively scheduled or truly scheduled.

For example, the following returns all the frame queues in the Tentatively Scheduled state:

```
# cat state_tentatively_sched
List of fq ids in state: Tentatively Scheduled
0x0001ca,0x0001cb,0x0001cc,0x0001cd,0x0001ce,0x0001cf,0x0001d0,0x0001d1,
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001d6,0x0001d7,0x0001d8,0x0001d9,
...
Out of Service count = 32201
Retired count = 0
Tentatively Scheduled count = 566
Truly Scheduled count = 0
Parked count = 0
Active, Active Held or Held Suspended count = 0
```

5.2.2.2.10.3.19 */sys/kernel/debug/qman/fqd/tde_[enable | disable]*

Description:

Query Tail_Drop_Enable bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Tail_Drop_Enable bit mask enabled or disabled.

For example, the following returns all the frame queues with Tail_Drop_Enable bit enabled:

```
# cat tde_enable
List of fq ids with: Tail-Drop Enable :enabled(1)
```

```
0x0001ca,0x0001cb,0x0001cc,0x0001cd,0x0001ce,0x0001cf,0x0001d0,0x0001d1,  
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001d6,0x0001d7,0x0001d8,0x0001d9,  
...  
Total FQD with: Tail-Drop Enable : enabled = 560  
Total FQD with: Tail-Drop Enable : disabled = 32207
```

5.2.2.2.10.3.20 /sys/kernel/debug/qman/fqd/wq

Description:

Query Destination Work Queue in all frame queue descriptors. This file takes as input work queue id combined with channel id (destination work queue). The output of this file returns all the frame queues with destination work queue number as specified in the input.

For example, the following returns all the frame queues with their destination work queue number equal to 0x10f:

```
# echo 0x10f > wq  
# cat wq  
List of fq ids with destination work queue id = 0x10f  
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001de,0x0001df,0x0001e0,0x0001e1,  
0x0001ea,0x0001eb,0x0001ec,0x0001ed,0x0001f6,0x0001f7,0x0001f8,0x0001f9,  
0x0001fa,0x0001fb,0x0001fd,0x0001fe  
Summary of all FQD destination work queue values  
Channel: 0x0 WQ: 0x0 WQ_ID: 0x0, count = 32199  
Channel: 0x0 WQ: 0x0 WQ_ID: 0x4, count = 1  
Channel: 0x0 WQ: 0x3 WQ_ID: 0x7, count = 64  
Channel: 0x1 WQ: 0x3 WQ_ID: 0xf, count = 64  
Channel: 0x2 WQ: 0x3 WQ_ID: 0x17, count = 64  
Channel: 0x3 WQ: 0x3 WQ_ID: 0x1f, count = 64  
Channel: 0x4 WQ: 0x3 WQ_ID: 0x27, count = 64  
Channel: 0x5 WQ: 0x3 WQ_ID: 0x2f, count = 64  
Channel: 0x6 WQ: 0x3 WQ_ID: 0x37, count = 64  
Channel: 0x7 WQ: 0x3 WQ_ID: 0x3f, count = 64  
Channel: 0x21 WQ: 0x3 WQ_ID: 0x10f, count = 20  
Channel: 0x42 WQ: 0x3 WQ_ID: 0x217, count = 8  
Channel: 0x45 WQ: 0x0 WQ_ID: 0x228, count = 1  
Channel: 0x60 WQ: 0x3 WQ_ID: 0x307, count = 8  
Channel: 0x61 WQ: 0x3 WQ_ID: 0x30f, count = 8  
Sysfs and Debugfs QMan/BMan interfaces  
QMan, BMan API RM, Rev. 0.13  
NXP Semiconductors NXP Confidential Proprietary 8-67  
Preliminary-Subject to Change Without Notice  
Channel: 0x62 WQ: 0x3 WQ_ID: 0x317, count = 8  
Channel: 0x65 WQ: 0x0 WQ_ID: 0x328, count = 1  
Channel: 0xa0 WQ: 0x0 WQ_ID: 0x504, count = 1
```

5.2.2.2.10.3.21 /sys/kernel/debug/qman/fqd/summary

Description:

Provides a summary of all the fields in all frame queue descriptors. This is a read only file.

```
# cat summary  
Out of Service count = 32201  
Retired count = 0  
Tentatively Scheduled count = 566  
Truly Scheduled count = 0
```

```

Parked count = 0
Active, Active Held or Held Suspended count = 0
-----
Prefer in cache count = 560
Hold active in portal count = 0
Avoid Blocking count = 528
High-priority SFDRs count = 0
CPC Stash Enable count = 0
Context-A stashing count = 528
ORP Enable count = 0
Tail-Drop Enable count = 560

```

5.2.2.2.10.3.22 /sys/kernel/debug/qman/ccsrmempeek

Description:

Provides access to Queue Manager ccsr memory map. This file takes as input an offset from the QMan CCSR base address. The output of this file returns the 32-bit value of the memory address as specified in the input.

For example, to query the QM IP Block Revision 1 register (which is at offset 0xbf8 from the QMan CCSR base address), we would do the following:

```

# echo 0xbf8 > ccsrmempeek
# cat ccsrmempeek
QMan register offset = 0xbf8
value = 0x0a010101

```

5.2.2.2.10.3.23 /sys/kernel/debug/qman/query_ceetm_xsfdr_in_use

Description:

Query the number of XSFDRs currently in use by the CEETM logic of the DCP portal. This file takes input as the DCP id. The output of the file returns the number of XSFDR in use. Please note this feature is only available in T4/B4 rev2 silicon.

For example, if we want to query XSFDR in use number of DCP 0, we would do the following:

```

# echo 0 > query_ceetm_xsfdr_in_use
# cat query_ceetm_xsfdr_in_use
DCP0: CEETM_XSFDR_IN_USE number is 0

```

5.2.2.2.10.4 BMan debugfs

5.2.2.2.10.4.1 /sys/kernel/debug/bman

Description:

This directory contains various BMan device debugging attributes.

5.2.2.2.10.4.2 /sys/kernel/debug/bman/query_bp_state

Description:

This file requests a snapshot of the availability and depletion state of each of BMan's buffer pools. This is a read only file.

For example, if we want to perform a query we could use this file in the following manner:

```

# cat query_bp_state
bp_id free_buffers_avail bp_depleted
0 yes no

```

Linux Kernel Drivers
DPAA 1.x Devices

- 1 no no
- 2 no no
- 3 no no
- 4 no no
- 5 no no
- 6 no no
- 7 no no
- 8 no no
- 9 no no
- 10 no no
- 11 no no
- 12 no no
- 13 no no
- 14 no no
- 15 no no
- 16 no no
- 17 no no
- 18 no no
- 19 no no
- 20 no no
- 21 no no
- 22 no no
- 23 no no
- 24 no no
- 25 no no
- 26 no no
- 27 no no
- 28 no no
- 29 no no
- 30 no no
- 31 no no
- 32 no no
- 33 no no
- 34 no no
- 35 no no
- 36 no no
- 37 no no
- 38 no no

39 no no
40 no no
41 no no
42 no no
43 no no
44 no no
45 no no
46 no no
47 no no
48 no no
49 no no
50 no no
51 no no
52 no no
53 no no
54 no no
55 no no
56 no no
57 no no
58 no no
59 no no
60 no no
61 no no
62 no no
63 yes no

5.2.2.2.11 Error handling and reporting

This chapter describes the QMan and BMan error handling and reporting.

5.2.2.2.11.1 Handling and Reporting

The QMan and BMan error interrupt services routines log the occurrence of every error interrupt. Some error interrupts can be triggered multiple times. To prevent a flood of error logging when this interrupts are raised, they are only logged on their first occurrence at which time they are disabled. The logs are generated via the `pr_warning()` kernel api. At the end of the interrupt service routine the ISR register is cleared. These logs are available on the console, `dmesg` and related log file.

The following QMan error conditions are logged a single time:

QM_EIRQ_PLWI and QM_EIRQ_PEBI.

The following BMan error conditions are logged a single time:

BM_EIRQ_FLWI (low water mark).

5.2.2.2.12 Operating system specifics

This chapter captures O/S-specific issues and distinctions, as the rest of the document essentially describes the interfaces in a generalized manner.

5.2.2.2.12.1 Portal maintenance

By default, the Linux kernel initializes QMan and BMan portals to perform all processing via interrupt-handling. As such there are no persistent threads or polling requirements in order to use portals in the Linux kernel.

Whereas for USDPAA (linux user space), the default is for all processing to be driven by polling, and support for the use of interrupts is disabled. The applications are required to call `qman_poll()` and `bman_poll()` within their run-to-completion loops to ensure that portal processing occurs regularly.

As described in [Processing non-interrupt-driven portal duties \(BMan\)](#) on page 107 (for BMan) and [Processing non-interrupt-driven portal duties \(QMan\)](#) on page 122 (for QMan), it is also possible to dynamically control at run-time which portal duties are interrupt-driven versus poll-driven, so the aforementioned defaults for Linux are start-up defaults. However, USDPAA needs to be built with "CONFIG_FSL_DPA_IRQ_SAFETY" defined in order to allow any duties to be interrupt-driven, whereas it is disabled by default (in `inc/public/conf.h`) due to a very slight performance improvement that it yields.

5.2.2.2.12.2 Callback context

In the Linux kernel, all interrupt-driven portal duties are handled in interrupt context, whereas all other portal duties are invoked from within the `qman_poll()` and `bman_poll()` functions, which are invoked by the application.

In USDPAA, even interrupt-driven portal duties are handled in an application context. Interrupts are handled within the kernel and locally disabled, and the presence of such interrupt events is available to the application via the USDPAA file-descriptor representing the portal devices. See [Interrupt handling](#) on page 154 for more information. Interrupt-driven portal duties are thus processed when the application calls the `qman_thread_irq()` and `bman_thread_irq()` functions, and other portal duties are processed when the application calls `qman_poll()` and `bman_poll()`.

5.2.2.2.12.3 Blocking semantics

Many high-level QMan and BMan API functions provide "WAIT" flags, to allow the API to block as part of its operation.

In the Linux kernel, "WAIT" behavior is implemented by allowing the calling thread to sleep until a given condition is satisfied. The limitation then to using "WAIT" flags is that the caller can not be in atomic context - i. e. not executing within an interrupt handler, tasklet, bottom-half, etc, nor with any spinlocks held. One consequence is that "WAIT" flags can not be used within a callback.

On run-to-completion systems such as USDPAA, "WAIT" behavior is unsupported and unavailable.

5.2.3 Configuring DPAA Frame Queues

5.2.3.1 Introduction

Describes configurations of Queue Manager (QMan) Frame Queues (FQs) associated with Frame Manager (FMan) network interfaces for the QorIQ Data Path Acceleration Architecture (DPAA). The relationship of the FMan and the QMan channels and work queues are illustrated by examples.

The basic configuration examples for QMan FQs provided yield straightforward and reliable DPAA performance. These simple examples may then be fine tuned for special use cases. For additional information and understanding of advanced system level features please refer to the DPAA Reference Manual.

DPAA provides the networking specific I/Os, accelerator/offload functions, and basic infrastructure to enable efficient data passing, without locks or semaphores, within the multi-core QorIQ SoC between:

1. The Power Architecture cores (and software)
2. The network and I/O interfaces through which that data arrives and leaves
3. The accelerator blocks used by the software to assist in processing that data.

Hardware-managed queues which reside in and are managed by the QMan provide the basic infrastructure elements to enable efficient data path communication. The data resides in delimited work units of frames/packets between cores, hardware accelerators and network interfaces. These hardware-managed queues, known as Frame Queues (FQs), are FIFOs of related frames. These frames comprise buffers that hold a data element, generally a packet. Frames can be single buffers or multiple buffers (using scatter/gather lists).

FQ assignment to consumers i.e., cores, hardware accelerators, network interfaces, are programmable (not hard coded). Specifically, FQs are assigned to work queues which in turn are grouped into channels. Channels which represent a grouping of FQs from which a consumer can dequeue from, are of two types:

- Pool channel: a channel that can be shared between consumers which facilitates load balancing/spreading. (Applicable to cores only. Does not apply to hardware accelerators or network interfaces)
- Dedicated channel: a channel that is dedicated to a single consumer.

Each pool or dedicated channel has eight (8) work queues. There are two high priority work queues that have absolute, strict priority over the other six (6) work queues which are grouped into medium and low priority tiers. Each tier contains three work queues which are serviced using a weighted round robin based algorithm. More than one FQ can be assigned to the same work queue as channels implementing a 2-level hierarchical queuing structure. That is, FQs are enqueued/dequeued onto/from work queues. Within a work queue a modified deficit round algorithm is used to determine the number of bytes of data that can be consumed from a FQ at the head of a work queue. The FQ, if not empty, is enqueued back onto the tail of the same work queue once its consumption allowance has been met.

NOTE

- The configuration information provided in this document applies to the QorIQ family of SoCs built on Power Architecture and DPAA technology
- The configuration information provided in this document assumes a top bin platform frequency.

5.2.3.2 FMan Network interface Frame Queue Configuration

Configuring the QMan Frame Queues (FQs) associated with the FMan network interfaces for QorIQ DPAA.

Each network interface has an ingress and an egress direction. The ingress direction is defined as the direction from the network interface to the cores. The egress direction is defined as the direction from the cores to the network interfaces.

FQs associated with FMan network interfaces can be either ingress or egress FQs. Ingress FQs are referred to FQs used in the ingress direction to store packets received from network interfaces to be processed by the cores. Egress FQs are referred to FQs used in the egress direction to store packets to be transmitted by FMan out of its network interfaces.

5.2.3.3 FMan network interface ingress FQs configuration

Dependencies for configuration of the ingress Frame Queues (FQs) is dependent on the QMan mechanism used to load balance/spread received packets across the multiple cores in QorIQ DPAA.

Two mechanisms are offered:

1. Dynamic load balancing

- Load spread the packets (from ingress FQs) to the cores based on actual core availability/readiness.
- Achieved through the use of QMan pool channel (i.e. a channel which can be shared by multiple cores).
- Maintaining packet ordering (e.g. when packets are being forwarded) is achieved through the following two mechanisms:
 - a. Order preservation; ensures that related packets (e.g. a sequence of packets moving between two end points) are processed in order (and typically one at a time).
 - b. Order restoration; allows packets to be processed out of order and then restores their order later on before they are transmitted out to the network interfaces.

- Improves core work load balancing over a static distribution based approach scheme but will not maintain core affinity because a FQ may get processed by multiple cores.

2. Static distribution

- Static association between FQs and cores; FQs are always processed by the same core.
- Achieved through the use of QMan dedicated channel (i.e. a channel which supplies FQs to a specific core).
- Static not dynamic, doesn't react to core load, assigns work to the cores in a static or fixed manner.
- Does not require any special order preservation/restoration mechanism as packet ordering is implicitly preserved.

For all of these mechanisms, QMan requires that related packets, which must be processed and/or transmitted in order, be placed on the same FQ. This does not mean that only related packets are placed on a given FQ; many sets of related packets (“flows”) can be placed on a single FQ. FMan is responsible for achieving this placement/FQ selection function through its distribution capabilities. For instance, FMan can be configured to apply a hash function to a set of packet header fields and use the hash value to select the FQ. This set of packet header fields can be for example, a 5-tuple consisting of:

- source IP address
- destination IP address
- protocol
- source port
- destination port

Note that the FMan processing may be out of order, but it has internal mechanism to ensure that packets are enqueued in order of reception.

These mechanisms can be configured and used simultaneously on an SoC device.

5.2.3.4 Ingress FQs common configuration guidelines

Guidelines and examples for configuring ingress Frame Queues (FQs) in the QorIQ DPAA are shown.

Following guidelines apply regardless of the load balancing mechanism(s) configured:

- Maximum number of ingress FQs for all ingress interfaces on the device (including any of the separate FQs that are used to serve as an order restoration point (ORP)): 1024
- Maximum number of ingress FQs per work queue (FIFO of FQs):
 - 64 if the aggregate bandwidth of the configured network interface(s) on the device is higher than 10 Gbit/s.
 - 128 if the aggregate bandwidth of the configured network interface(s) on the device is 10 Gbit/s or lower.
- The aggregate bandwidth of the configured network interface(s) on the device receiving packets into FQs associated to the same work queue should not exceed 10 Gbit/s. In other words, the recommended maximum incoming rate into a single work queue is 10 Gbit/s. If the configured network interface(s) on the device is higher than 10 Gbit/s, then multiple work queues should be used.
- Since the Single Frame Descriptor Record (SFDRs) reservation scheme is recommended for the egress FQs ([FMan network interface egress FQs configuration](#)) and any other FQs assigned to high priority work queues will also use these reserved SFDRs, careful consideration should be given to the required number of ingress FQs assigned to the high priority work queues as SFDRs are a scarce QMan resource (there is a total of 2K SFDRs). One needs to leave sufficient SFDRs for FQs not using the reserved SFDRs (e.g. ingress FQs assigned to medium or low priority work queues).

As an example, if one allocates 1024 ingress FQs and the aggregate bandwidth of the configured network interface(s) on the device is higher than 10 Gbit/s, then a minimum of 16 work queues would be required based on the above guidelines.

Assuming that all 1024 FQs are to be scheduled at the same priority using a dynamic load balancing scheme, a minimum of 6 pool channels would need to be used (based on the fact that up to 3 work queues can be used within a medium or low priority tier).

The guideline “maximum of 1024 ingress FQs for all ingress interfaces” results from the size of the internal memory in QMan that is used to cache Frame Queue Descriptors (FQDs). This internal memory is sized to 2K entries. To achieve high, deterministic and reliable performance under worst-case packet workload (back-to-back 64-byte packets enqueued to FQs on a rotating basis), all ingress FQDs must remain in the QMan internal cache. FQD cache misses increase the time required to enqueue packets as the FQD may need to be read from external memory. This in return could result in received packets being discarded by the MAC due MAC FIFO overflow condition as a result of the back-pressure applied by the FMan to the MAC as there is little buffering between the MAC and the point at which incoming packets are enqueued onto the ingress FQs.

Although a device configured with a number of ingress FQs higher than the size of the QMan FQD internal cache would operate at high performance with no packet discards if the incoming traffic exhibited some level of temporal locality, it is generally recommended that the device be engineered such that ingress path operates at line rate under worst case packet workload to avoid unnecessary packets losses and to make effective use of QMan to prioritize and apply appropriate QoS if there is congestion in a downstream element (e.g. cores). Since all FQs defined on the device shared the QMan 2K internal FQD cache, the recommended maximum number of ingress and egress FQs is even more constrained so that there is adequate space left for caching FQDs assigned to accelerators.

With regards to congestion management, the default mechanism for managing ingress FQ lengths is through buffer management. Input to FQs is limited to the availability of buffers in the buffer pool used to supply buffers to the FQs. Although very efficient and simple, when a buffer pool is shared by multiple FQs, there is no protection between the FQs sharing the buffer pool and as a result a FQ could potentially occupy all the buffers.

Queue management mechanisms can be configured (e.g. tail drop/WRED) to improve congestion control however appropriate software must be in place to handle enqueue rejections as a result of queue congestion.

5.2.3.5 Dynamic load balancing with order preservation - ingress FQs configuration guidelines

Dynamic load balancing with order preservation provides a very effective workload distribution technique to achieve optimal utilization of all cores as it distributes packets to the cores based on actual core availability/readiness.

Order preservation allows FQs to be dynamically reassigned from one core to another while preserving per-FQ packet ordering. It never allows packets from the same FQ to be processed at multiple cores at the same time; a specific FQ is only processed by one core at any given time. Once the FQ is released by the core, it can be processed by any of the cores. To keep multiple cores active there must be multiple FQs distributing packets to the cores, each with a set of (potentially) related packets.

In packet-forwarding scenarios, Discrete Consumption Acknowledgement (DCA) embedded in the enqueue commands should be used to forward packets as this ensures that QMan will release the ingress FQ on software's behalf once it has finished processing the enqueue command. This provides order preservation semantic from end-to-end (from dequeue to enqueue). To support the above, software portals that will be issuing DCA notifications to QMan must be configured with DCA mode enabled.

Following are specific configuration guidelines for ingress FQs used for dynamic load balancing with order preservation:

- FQ must be associated to a pool channel (i.e. a channel which can be shared by multiple cores).
- Within a pool channel, minimum number of FQs per active portal (core): 4.
- Frame Queue Descriptor (FQD) attributes settings:
 - Prefer in cache.
 - Hold active set.
 - Don't set avoid blocking.
 - Intra-class scheduling (ICS) credit set to 0 unless a more advanced scheduling scheme is required.
 - Don't set force SFDR allocate unless FQ needs performance optimization.
 - FQD CPC stashing enabled.

- Dequeued Frame Data, Annotation, and FQ Context stashing: application dependent.
- Order Restoration Point (ORP) disabled.

5.2.3.6 Dynamic load balancing with order restoration - ingress FQs configuration guidelines

Dynamic load balancing with order restoration dispatches packets from the same Frame Queue (FQ) to different processor cores without attempting to maintain order. QMan provides order restoration with specific configurations shown.

The packet order in the original FQ (e.g. ingress FQ) is restored once the cores complete its processing and return the packets to QMan for sending to the next destination (e.g. egress FQ for transmission).

Dynamic load balancing with order restoration has the advantage that parallel processing of related traffic is possible; allows to process without packet drops a flow that exceed the processing rate of a core. However order restoration does make use of more resources than the other distribution schemes. Its usage must also be balanced with applications need to atomically access shared data.

Order restoration is achieved through the following two QMan components:

- Order Definition Points (ODPs)
 - A point through which packets pass, where their order or sequence relative to each other is defined.
 - For convenience each FQ has an ODP for packets dequeued from that FQ.
- Order Restoration Points (ORPs)
 - A point through which packets pass, where their order or sequence is restored to that defined at the related ODP.
 - If a packet is out of sequence it is held until it is in sequence.
 - ORP data structure is maintained in a FQ; it is recommended that a dedicated/separate FQ be allocated solely for this purpose.

Following are specific configuration guidelines for ingress FQs used for dynamic load balancing with order restoration:

- FQ must be associated to a pool channel (i.e. a channel which can be shared by multiple cores).
- For each ingress FQ supporting order restoration, a separate FQ should be allocated to serve as the ORP.
- Ingress FQ descriptor attributes settings.
 - Prefer in cache
 - Don't set hold active.
 - Set avoid blocking.
 - Intra-class scheduling (ICS) credit set to 0 unless a more advanced scheduling scheme is required.
 - Don't set force SFDR allocate unless FQ needs performance optimization.
 - FQD CPC stashing enabled.
 - Dequeued Frame Data, Annotation, and FQ Context stashing: application dependent.
 - ORP disabled.

Following are specific configuration guidelines for ORP FQs:

- FQs used for ORP don't need to be associated with a pool or dedicated channel.
- ORP FQ descriptor attributes settings:
 - Prefer in cache .
 - Don't set hold active.
 - Don't set avoid blocking.

- Intra-class scheduling credit set to 0.
- Don't set force SFDR allocate .
- FQD CPC stashing enabled.
- ORP enabled.
- Recommended ORP restoration window size: 128.

5.2.3.7 Static distribution - Ingress FQs Configuration Guidelines

With a static distribution approach, a single FQ is always processed by the same processor core. Specific guidelines for processor core affinity are presented.

Although not as effective as a dynamic based approach from a resource utilization aspect, static distribution maintains core affinity meaning that the mapping from the flow to the core is preserved.

Distribution of packets (selection of FQ) can be based on hash keys, ensuring that packets from the same traffic flow will always go to the same cores. The FQ selection function is achieved by FMan.

Following are specific configuration guidelines for ingress FQs used for static distribution:

- FQ must be associated to a dedicated channel (i.e. a channel which supplies FQs to a specific core); multiple FQs can be associated to a single dedicated channel.
- Within a dedicated channel, minimum number of FQs: 1.
- FQ descriptor attributes settings:
 - Prefer in cache .
 - Don't set hold active
 - Don't set avoid blocking.
 - Intra-class scheduling (ICS) credit set to 0 unless a more advanced scheduling scheme is required. On P4080/P3041/P5020, due to errata number QMAN-A002, allowable values for ICS are: 0 and 15'h7FFF.
 - Don't set force SFDR allocate unless FQ needs performance optimization.
 - FQD CPC stashing enabled.
 - Dequeued Frame Data, Annotation, and FQ Context stashing: application dependent.
 - ORP disabled.

5.2.3.8 FMan network interface egress FQs configuration

Configuration guidelines for egress Frame Queues (FQs) for QorIQ DPAA

FQ Configurations:

- Maximum number of egress FQs for all network interfaces: 128.
- Minimum number of egress FQs per network interface: 1.
- Maximum number of egress FQs per work queue: 8.
- Egress FQ descriptor attributes settings:
 - Prefer in cache.
 - Don't set hold active .
 - Don't set avoid blocking.
 - Set force SFDR allocate to ensure that egress queues make use of the reserved SFDRs; the SFDR reservation threshold field of the QMan SFDR configuration register must also be set accordingly (5 SFDRs per egress FQ + 3 extra SFDRs as required by QMan).

- Intra-class scheduling set to zero (0) unless a more advanced scheduling scheme is required.
- FQD CPC stashing enabled.
- ORP disabled.

5.2.3.9 Accelerator Frame Queue Configuration

Configurations for Frame Queues (FQs) used to communicate with accelerators for QorIQ DPAA are shown.

FQ accelerator Guidelines:

- Since the Single Frame Descriptor Record (SFDRs) reservation scheme is recommended for the egress FQs ([FMan network interface egress FQs configuration](#)) and any other FQs assigned to high priority work queues will also use these reserved SFDRs, careful consideration should be given to the required number of accelerator FQs assigned to the high priority work queues as SFDRs are a scarce QMan resource (there is a total of 2K SFDRs). One needs to leave sufficient SFDRs for FQs not using the reserved SFDRs (e.g. accelerator FQs assigned to medium or low priority work queues).
- Accelerator FQ descriptor attributes settings:
 - Don't set prefer in cache.
 - Don't set hold active .
 - Don't set avoid blocking.
 - FQD CPC stashing enabled.
 - Intra-class scheduling (ICS) credit set to 0 unless a more advanced scheduling scheme is required.
 - Don't set force SFDR allocate unless FQ needs performance optimization.
 - Dequeued Frame Data, Annotation, and FQ Context stashing: application dependent.
 - ORP disabled.

Generally accelerators are used in a request/response manner and in cases where a pair of FQs is needed per session/flow to communicate with accelerators, one may need to allocate a very large number of FQs (in the order of thousands). At times when many FQs allocated to an accelerator are active, this situation can result in having significant amount of cache consumed for storing the corresponding FQ descriptors. This in turn may negatively impact overall system performance.

To ensure optimal resource utilization (e.g. QorIQ caches), maximize throughput and avoid overload, it is recommended that the number of outstanding requests/responses to an accelerator be regulated. Typically, for a given accelerator, regulating the number of outstanding requests/responses across all its FQs to a few hundredths should be sufficient to maintain high throughput without overloading the system. Regulating the number of outstanding requests/responses to an accelerator can be achieved through various methods.

One method is to keep track in software of the total number of outstanding requests/responses to an accelerator and once this number exceeds a threshold, software would stop sending requests to that accelerator.

Another method is to make use of the congestion management capabilities of QMan. Specifically, all FQs allocated to an accelerator can be aggregated into a congestion group. Each congestion group can be configured to track the number of Frames in all FQs in the congestion group. Once this number exceeds a configured threshold, the congestion group enters congestion. When a congestion group enters congestion, QMan can be configured to rejects enqueues to any FQs in the congestion group and/or sent notification indicating that the congestion group has entered congestion. If a Frame (or request) is not going to be enqueued, it will be returned to the configured destination via an enqueue rejection notification. Congestion state change notifications are generated when the congestion group either enters congestion or exits congestion. On software portals, the congestion state change notification is sent via an interrupt.

5.2.3.10 DPAA Frame Queue Configuration Guideline Summary

Summary of Configurations for Frame Queue (FQ) communication with accelerators for QorIQ DPAA

Four tables comprise this summary:

- Global Configuration settings
- Network interface ingress FQ guidelines
- Network interface egress FQ guidelines
- Accelerator FQ guidelines

Table 10. Global Configuration Settings Summary

Parameter or subject	Guideline
FQD stashing	Recommend QMan explicitly stash FQDs: <ul style="list-style-type: none"> • QMan; both the global CPC stash enable bit in the QMan FQD_AR register and the CPC stash enable bit in the FQD must be set. • PAMU; PAACT tables used by PAMU also configured appropriately .
PFDR stashing	Recommend QMan explicitly stash PFDRs: <ul style="list-style-type: none"> • QMan; the global CPC stash enable bit in the QMan PFDR_AR register must be set . • PAMU; PAACT tables used by PAMU must also be configured appropriately .
SFDR reservation threshold	Set SFDR reservation threshold in QMan SFDR configuration register to: <ul style="list-style-type: none"> • Total number of FQs using reserved SFDRs times 5 (5 SFDRs per FQ) plus 3 extra SFDRs as required by QMan. Recommend that all egress FQs use reserved SFDRs .

Table 11. Network Interface Ingress FQs Guidelines Summary

Parameter or subject	Guideline
Maximum number of ingress FQs for all ingress interfaces on the device (including any of the separate FQs that are used to serve as an order restoration point (ORP))	1024 FQs
Maximum number of ingress FQs per work queue.	<ul style="list-style-type: none"> • 64 FQs per work queue if the aggregate bandwidth of the configured network interface(s) on the device is higher than 10 Gbit/s. • 128 FQs per work queue if the aggregate bandwidth of the configured network interface(s) on the device is 10 Gbit/s or lower.
The maximum aggregate bandwidth of the configured network interface(s) on the device receiving packets into FQs associated to the same work queue	10 Gbit/s

Table continues on the next page...

Table 11. Network Interface Ingress FQs Guidelines Summary (continued)

Parameter or subject	Guideline
Within a pool channel, minimum number of FQs per active portal (cores).	4 FQs
Within a dedicated channel, minimum number of FQs:	1 FQ
Assignment to high priority work queues.	Should be limited enough to leave sufficient SFDRs for FQs not using the reserved SFDRs (e.g. ingress FQs assigned to medium or low priority work queues).
Order restoration point (ORP).	A separate FQ should be allocated and dedicated to serve as the ORP for each ingress FQ supporting order restoration.
Ingress FQ descriptor load balancing and performance related settings.	<ul style="list-style-type: none"> • Prefer_in_Cache: 1 • CPC Stash Enable: 1 • ORP_Enable: 0 • Avoid_Blocking: <ul style="list-style-type: none"> • 0 if static distribution or dynamic load balancing with order preservation. • 1 if dynamic load balancing with order restoration. • Hold_Active <ul style="list-style-type: none"> • 0 if static distribution or dynamic load balancing with order restoration . • 1 if dynamic load balancing with order preservation. • Force_SFDR_Allocate: 0 unless FQ needs performance optimization. • Intra-Class Scheduling Credit: 0 unless a more advanced scheduling scheme is required.
ORP FQ descriptor order restoration and performance related settings.	<ul style="list-style-type: none"> • Prefer_in_Cache: 1 • CPC Stash Enable: 1 • ORP_Enable: 1 • Avoid_Blocking: 0 • Hold_Active: 0 • Force_SFDR_Allocate: 0 • ORP Restoration Window Size: 2 (corresponds to window size of 128 frames). • Class Scheduling Credit: 0

Table 12. Network Interface Egress FQs Guidelines Summary

Parameter or subject	Guideline
Maximum number of egress FQs for all network interfaces.	128 FQs
Minimum number of egress FQs per network interface.	1 FQ
Maximum number of egress FQs per work queue.	8 FQs
Egress FQ descriptor performance related settings.	<ul style="list-style-type: none"> • Prefer_in_Cache: 1 • CPC Stash Enable: 1 • ORP_Enable: 0 • Avoid_Blocking: 0 • Hold_Active: 0 • Force_SFDR_Allocate: 1 • Class Scheduling Credit: 0 unless a more advanced scheduling scheme is required.

Table 13. Accelerator FQs Guidelines Summary

Parameter or subject	Guideline
Assignment to high priority work queues.	Should be limited enough to leave sufficient SFDRs for FQs not using the reserved SFDRs (e.g. accelerator FQs assigned to medium or low priority work queues).
Egress FQ descriptor performance related settings.	<ul style="list-style-type: none"> • Prefer_in_Cache: 0 • CPC Stash Enable: 1 • ORP_Enable: 0 • Avoid_Blocking: 0 • Hold_Active: 0 • Force_SFDR_Allocate: 0 unless FQ needs performance optimization . • Class Scheduling Credit: 0 unless a more advanced scheduling scheme is required .

5.2.4 Frame Manager

5.2.4.1 Frame Manager Linux Driver User Guide

5.2.4.1.1 Introduction

This part is describing the Linux implementation of the driver for the Frame Manager, or FMD.

The Linux driver for the Frame Manager is based on the NetCommSw drivers, or NCSW. The NCSW drivers are written in an OS-agnostic fashion, so what the Linux FMD does is to implement a set of standard Linux character devices that rely on the NCSW drivers to do the actual communication with the hardware. The figure below describes this best:

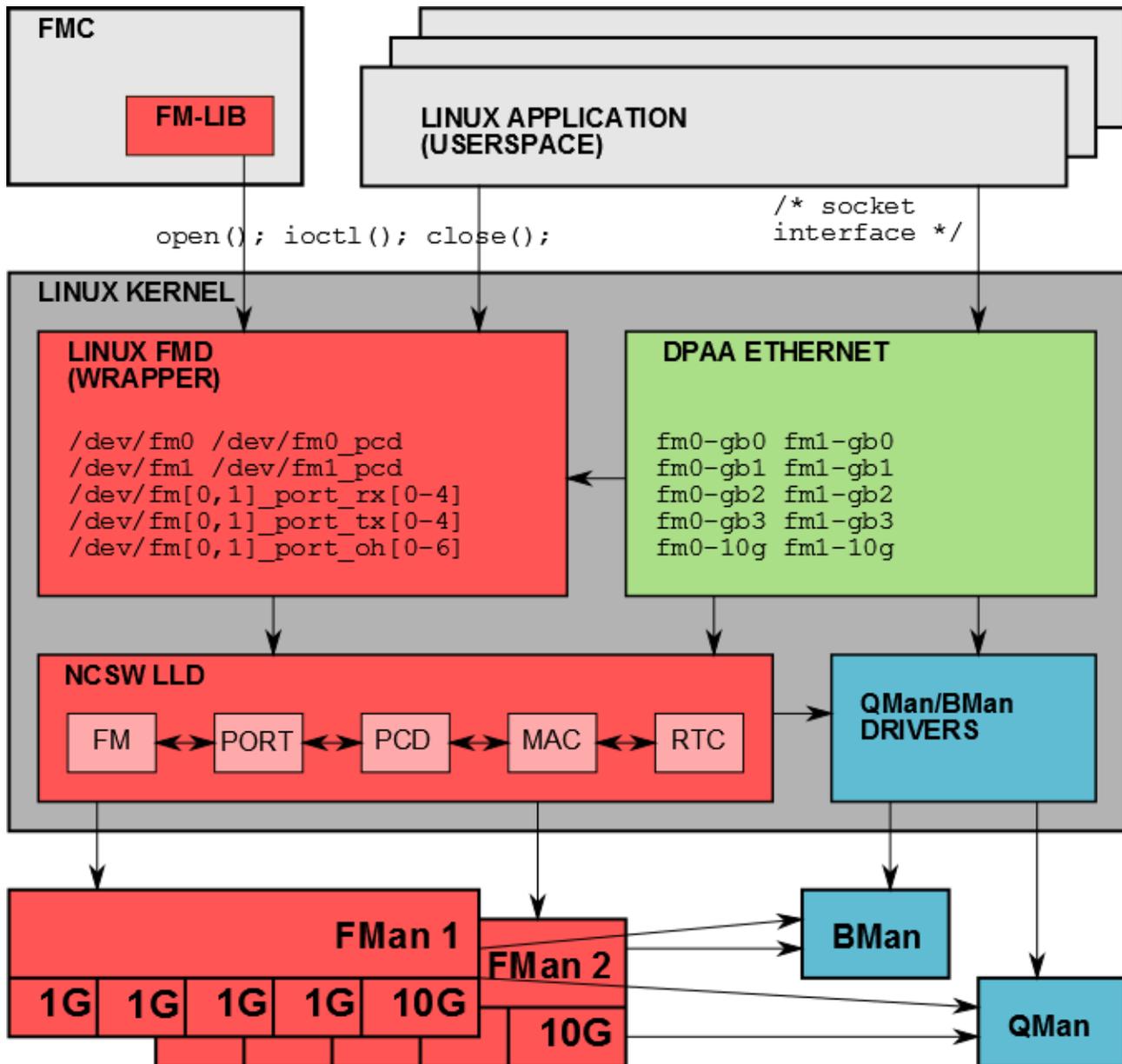


Figure 26. FMan-centric view of relationships between DPAA software and hardware blocks in the Linux environment.

The features of the Linux FMan Driver are the following:

- Performs initialization of the Frame Manager based on platform configuration (device tree), and on probing of the actual hardware;
- Supports Linux user space applications looking to create FMan PCD configurations;
- Attaches/detaches PCDs to/from FMan ports;
- Reports FMan and port status:
 - FMan registers

- FMan statistics
- FMan port and MAC counters

The Linux FMan driver does not handle actual network traffic. Network traffic in Linux is being handled exclusively by Linux network devices. Network traffic going through FMan can only be handled by the Linux DPAA Ethernet driver. Although the DPAA Ethernet and the Linux FMan Driver share strong links and interdependencies with the underlying low-level FMD and with each other, their feature sets do not overlap. The DPAA Ethernet driver is described in the *Linux Ethernet Driver User Manual*.

Since the Linux FMD implementation relies heavily on the NCSW FMD, we strongly recommend *NetCommSw's Frame Manager Driver User's Guide* as prerequisite reading.

The USDPAA is a special case not detailed here.

5.2.4.1.2 The Linux FMD Devices

The Linux interface to the FMD consists in several Linux character devices:

- `/dev/fm[0,1]`, each corresponding to an actual Frame Manager;
- `/dev/fm[0,1]_pcd` are PCD devices corresponding each to a Frame Manager;
- `/dev/fm[0,1]_port_rx[0-4]`, and `/dev/fm[0,1]_port_tx[0-4]` corresponding to the physical ports of each FMan: each rx/tx device in a pair corresponds to the receive and transmit sides of a physical port;
- `/dev/fm[0,1]_port_oh[0-6]` correspond to the Offline Parsing ports.

These devices' creation and initialization is performed at boot time, based on probing of the physical hardware, as well as on the parsing of the device tree. Each of the physical ports can thus be disabled from the device tree, but also from the Reset Configuration Word (for details please consult the **Reset Configuration Word (RCW)** section of the **P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual**, or similar RM for other platform; for other useful reading please consult *Selecting Ethernet Interfaces* in the **QorIQ SDK Ethernet** manual).

NOTE

The assumption for the remainder of this document is that the device tree and the RCW are immutable, and therefore no attempt will be made to describe their formats in here. For details regarding the RCW/.dts, please consult the relevant documentation.

Therefore, depending on platform and on RCW/.dts configuration, some of these devices may be missing. The mapping of these devices to the physical ports is given in the following table:

Table 14. Mapping of Linux devices to low-level port IDs.

Linux Device	Low-Level ID	Identification
<code>/dev/fm0_port_rx0 /dev/fm0_port_tx0</code>	0	1st FMan's 1st 1GbE Receive, Transmit
<code>/dev/fm0_port_rx1 /dev/fm0_port_tx1</code>	1	1st FMan's 2nd GbE Receive, Transmit
<code>/dev/fm0_port_rx2 /dev/fm0_port_tx2</code>	2	1st FMan's 3rd GbE Receive, Transmit
<code>/dev/fm0_port_rx3 /dev/fm0_port_tx3</code>	3	1st FMan's 4th GbE Receive, Transmit

Table continues on the next page...

Table 14. Mapping of Linux devices to low-level port IDs. (continued)

Linux Device	Low-Level ID	Identification
/dev/fm0_port_rx4 /dev/fm0_port_tx4	4	1st FMan's 5th GbE ^[2] Receive, Transmit
/dev/fm0_port_rx5 /dev/fm0_port_tx5	5	1st FMan's 10Gb Receive, Transmit
N/A	0	1st FMan's Host Command
/dev/fm0_port_oh0	1	1st FMan's 1st Offline Parsing
/dev/fm0_port_oh1	2	1st FMan's 2nd Offline Parsing
/dev/fm0_port_oh2	3	1st FMan's 3rd Offline Parsing
/dev/fm0_port_oh3	4	1st FMan's 4th Offline Parsing
/dev/fm0_port_oh4	5	1st FMan's 5th Offline Parsing
/dev/fm0_port_oh5	6	1st FMan's 6th Offline Parsing
/dev/fm0_port_oh6	7	1st FMan's 7th Offline Parsing
/dev/fm1_port_rx0 /dev/fm1_port_tx0	0	2nd FMan's 1st 1GbE Receive, Transmit
/dev/fm1_port_rx1 /dev/fm1_port_tx1	1	2nd FMan's 2nd 1GbE Receive, Transmit
/dev/fm1_port_rx2 /dev/fm1_port_tx2	2	2nd FMan's 3rd 1GbE Receive, Transmit
/dev/fm1_port_rx3 /dev/fm1_port_tx3	3	2nd FMan's 4th 1GbE Receive, Transmit
/dev/fm1_port_rx4 /dev/fm1_port_tx4	4 ^[3]	N/A
/dev/fm1_port_rx5 /dev/fm1_port_tx5	5	2nd FMan's 10Gb Receive, Transmit
N/A	0	2nd ^[4] FMan's Host Command
/dev/fm1_port_oh0	1	2nd FMan's 1st Offline Parsing Port
/dev/fm1_port_oh1	2	2nd FMan's 2nd Offline Parsing Port
/dev/fm1_port_oh2	3	2nd FMan's 3rd Offline Parsing Port
/dev/fm1_port_oh3	4	2nd FMan's 4th Offline Parsing Port
/dev/fm1_port_oh4	5	2nd FMan's 5th Offline Parsing Port
/dev/fm1_port_oh5	6	2nd FMan's 6th Offline Parsing Port

Table continues on the next page...

[2] Only on P5020.

[3] This port & devices are not available on any of the platforms supported by this release!

[4] Only P4080 has a second FMan.

Table 14. Mapping of Linux devices to low-level port IDs. (continued)

Linux Device	Low-Level ID	Identification
/dev/fm1_port_oh6	7	2nd FMan's 7th Offline Parsing Port

The Low Level IDs are the IDs that are used by the Low Level Drivers (upon which the Linux FMan Driver is based) to distinguish between the physical ports. It is obvious from the above table that the port ID alone does not allow for uniquely identifying a single port. It has to be combined with the following information in order to successfully point to the desired port:

- FMan ID: 0 or 1 for FMan1 or 2, respectively;
- Port type: 1G, 10G or O/H (Offline Parsing/Host Command).

Although all this may seem confusing at first, the LLD API provides convenient enums/macros to deal with these aspects. Furthermore, the FMD driver API tries its best to hide these details from the userspace Linux programmer, specifically by using dedicated /dev entries for each port, etc. However, not all userspace-visible API is free of such port IDs, so this is why we even mention them here.

The FMD LLD uses no distinct port IDs for Rx and Tx, the distinction between Receive and Transmit being made by calling distinct Rx/Tx-specific functions, or by specifying the "RX" or "TX" direction as a separate argument.

The Host Command ports are invisible to the Linux application. One needs to be aware, though, of their mere existence at the least, since the LLD allocates the first physical O/H port of every FMan to this purpose ("O/H" standing for "Offline Parsing/Host Command"). There are 8 such O/H ports on each FMan that can be used for these purposes; the first of these having been dedicated by the LLD to Host Commands, while the remaining 7 being available for Offline Parsing. Host Commands are just one of the vehicles through which the LLD exercises control of the FMan hardware.

NOTE

Please note that depending on the platform, RCW, and .dts configuration not all the possible combinations of devices and ports are possible, and most certainly some will be missing from any existing configuration. For details regarding possible port & device configurations for a specific platform, please consult the Reference Manuals for that platform, as well as the relevant chapters from the SDK documentation for that platform.

Alongside these character devices, and out of the scope of this writing, are the Linux network devices, labeled using the `fm[1,2]-gb[0-4]` (e.g. `fm1-gb0`, `fm2-gb3`) and `fm[1,2]-10g` (i.e. `fm1-10g`, `fm2-10g`) schemes, which provide the means for Linux to handle actual network traffic, i.e. "traffic termination". These network devices are instances of the Linux DPAA Ethernet Driver, which is architected as a separate entity from the Linux FMan Driver, but which both make use at some point of the same Low-Level Driver FMD API. The feature sets of the DPAA Ethernet and of the Linux FMan drivers are disjunct, though, which is the main reason for their coexistence.

NOTE

There is no requirement that these are the only network devices in the system. You may find the well known `eth0`, `eth1`, etc. devices alongside e.g. `fm1-gb0`, except that these other network devices will correspond to other vendors' NICs that may be installed in the system and will be serviced by vendor-specific, non-DPAA, Ethernet drivers.

There are a few constants #defined in the headers that need to be included when working with the Linux FMD (in both kernel and user spaces) that may come in handy when having to deal with devices and port IDs:

- `FM_MAX_NUM_OF_1G_RX_PORTS`
- `FM_MAX_NUM_OF_10G_RX_PORTS`
- `FM_MAX_NUM_OF_1G_TX_PORTS`
- `FM_MAX_NUM_OF_10G_RX_PORTS`
- `FM_MAX_NUM_OF_RX_PORTS`

- `FM_MAX_NUM_OF_TX_PORTS`
- `FM_MAX_NUM_OF_OH_PORTS`
- `IOC_FM_MAX_NUM_OF_VALID_PORTS`

that together with `INTG_MAX_NUM_OF_FM` can give the programmer the essential tools to get around in a specific configuration (this list, though, is not exhaustive: please consult the relevant API Reference/header files before attempting to #define your own).

Also, the

```
$ ls /dev/fm*
```

Linux shell command can conveniently show all the FMD devices currently available in the target system.

5.2.4.13 Linux FMD Programming Model

Given the Linux devices presented earlier, a Linux application looking to use the FMan features can use the general Linux character device syscall interface:

- `open()/close()` - this is essential API when working with Linux devices.
- `read()/write()` - although `read()` and `write()` operations are mandatory to be implemented by all Linux devices, there are no read/write semantics associated with the FMD devices.
- `ioctl()` calls are used extensively as the only means to communicate with the hardware. The `ioctl` API does little more than delegating the `ioctl()` syscall to the underlying LLD API (for the actual mapping of IOCTLs to actual LLD APIs, please consult the tables available in the following sections).

We'll state here once more that the programming model is essentially that of the FMD LLD. The Linux wrapper merely adapts the LLD to the Linux interface requirements. This part of the SDK documentation focuses only on the Linux specifics. For details regarding individual API calls, please refer to the *Frame Manager Driver API Reference Manual*.

As is the case with any Linux device, the general sequence of actions when using the FMD devices is the following:

1. Linux boots: all `/dev/fm*` devices are being created, FMan resources initialized according to `platform/RCW/dts`;
2. User launches FMD-aware application;
3. User app. performs `open()` on selected `/dev/fm*` device/s;
4. User app. performs `ioctl()` call/s on the `fd` returned by the previous successful `open()` call;
5. When the user app. decides it has finished working with selected `/dev/fm*` device, it must call `close()` on its `fd`, just like on any other Linux device.

Not all the LLD functions have a correspondent in the FMD IOCTLs. Only those functions have been selected which makes sense from an architectural standpoint. The same/other LLD functions are also being called by the Linux wrapper unrestrictedly, as needed to perform its required actions, and not only in response to `ioctl()` calls.

The arguments of the `ioctl()` calls can be quite complex, and may have complex requirements, as they are described in the **LLD API Reference** (Frame Manager Driver API Documentation).

The following required low-level initialization APIs: `FM_Config()`, `FM_PCD_Config()`, `FM_PORT_Config()`, and subsequently `FM_Init()`, `FM_PCD_Init()`, `FM_PORT_Init()` are being called from within the Linux FMD initialization code at boot time. They are therefore not accessible to the user space application. Any configuration of FMan hardware resources will be performed using Linux-specific means: device tree, kernel build configuration, etc. Code in the DPAA Ethernet driver also initializes the configured MACs using `FM_MAC_Config()`, then `FM_MAC_Init()`, as required by the *Frame Manager Driver API Reference Manual*, and as described in *The DPAA Ethernet Driver's User Manual*.

The correspondence between FMD Linux devices and DPAA ETH network devices is intuitive: there is a pair of `/dev/fmX_port_(rxY|txY)` devices for each `fmX-gbY` or `fmX-10g` device in the system. However, due to configuration, it is possible that at boot time not all FMan ports be probed by the DPAA Ethernet driver, hence not all `/dev/fmX_port_(rxY|txY)` may have a corresponding `netdev`. This is because the FMan port devices and the DPAA Ethernet devices are being configured in different sections of the device tree. The binding between these devices is also done in the device tree.

While Offline Parsing ports are being fully supported by the FMan Driver, currently it is not possible to inject traffic from user space to these ports, as there is no netdev being created for them, as the Linux FMD does not handle traffic. There is indeed a way for kernel space drivers or e.g. USDPAA apps. to use them, but that is out of scope here.

It is not to be expected that a FMan port device for which a corresponding DPAA Ethernet netdev has not been configured, to be fully functional. That is because port functionality is reliant also upon additional DPAA resources (i.e. frame queues, buffer pools) that are being initialized exclusively by the DPAA Ethernet driver. Therefore, even though `/dev/fmX_port_*` devices may exist for such ports, trying to access them may result in an error.

`FM_PORT_Enable()` and `FM_PORT_Disable()` are called for specific ports during `ifconfig up/down` of the corresponding network device (DPAA Ethernet-specific). They are also available as IOCTLS for the `/dev/fmX_port*` devices, but while in the DPAA Ethernet they are called for both ports of the RX/TX pair, the `/dev/fmX_port_(rxY|txY)` allow for selectively enabling/disabling of only one of the RX/TX sides, as desired.

The `ioctl()` API conforms to Linux rules for all FMD devices. However, errors originating within the LLD will invariably be reported to the user as `-EFAULT`. All such errors should be considered non-recoverable and should be immediately followed by a `close()` on the device for which they were reported. A more descriptive message should be printed on the bootup console only, identifying the LLD function, and the line in the source file where the error has occurred. One can look at the documentation for `enum e_ErrorType` in the **LLD API Reference** (Frame Manager Driver API Documentation) for details regarding all the possible LLD error codes and their general meaning.

The following sections will present a brief description of each type of Linux device, as well as their IOCTLS' mapping to the FMD LLD API.

5.2.4.14 The Linux FMan Device

This device corresponds to an individual Frame Manager, and is required for performing FMan-wide actions. The FMan device merely acts as a portal for the IOCTLS that are listed in the table below:

Table 15. IOCTLS for the FMan Device

IOCTL	LLD Mapping	Brief
FM_IOC_SET_PORTS_BANDWIDTH	FM_SetPortsBandwidth()	Sets ports' bandwidths as percentage of total bandwidth.
FM_IOC_GET_REVISION	FM_GetRevision()	API to get the FMan's revision.
FM_IOC_GET_COUNTER	FM_GetCounter()	API to read FMan hardware counters (also available through sysfs).
FM_IOC_SET_COUNTER	FM_ModifyCounter()	API to modify/reset FMan's counters.
FM_IOC_FORCE_INTR	FM_ForceIntr()	Forces an FMan interrupt (or exception). Dangerous! Use for debugging only!
FM_IOC_GET_API_VERSION	FM_GetApiVersion()	Reads the FMD IOCTL API version.
FM_IOC_VSP_CONFIG	FM_VSP_Config()	Creates descriptor for the FM VSP module.
FM_IOC_VSP_INIT	FM_VSP_Init()	Initializes the FM VSP module
FM_IOC_VSP_FREE	FM_VSP_Free()	Frees all resources that were assigned to FM VSP module.

Table continues on the next page...

Table 15. IOCTLs for the FMan Device (continued)

IOCTL	LLD Mapping	Brief
FM_IOC_VSP_CONFIG_POOL_DEPLETION	FM_VSP_ConfigPoolDepletion()	Calling this routine enables pause frame generation depending on the depletion status of BM pools. It also defines the conditions to activate this functionality. By default, this functionality is disabled.
FM_IOC_VSP_CONFIG_BUFFER_PREFIX_CONTENT	FM_VSP_ConfigBufferPrefixContent()	Defines the structure, size and content of the application buffer.
FM_IOC_VSP_CONFIG_NO_SG	FM_VSP_ConfigNoScatherGather()	Returns the pointer to the parse result in the data buffer. In Rx ports this is relevant after reception, if parse result is configured to be part of the data passed to the application. For non Rx ports it may be used to get the pointer of the area in the buffer where parse result should be initialized - if so configured. See FM_VSP_ConfigBufferPrefixContent for data buffer prefix configuration.
FM_IOC_CTRL_MON_START	FM_CtrlMonStart()	Start monitoring utilization of all available FM controllers.
FM_IOC_CTRL_MON_STOP	FM_CtrlMonStop()	Stop monitoring utilization of all available FM controllers.
FM_IOC_CTRL_MON_GET_COUNTERS	FM_CtrlMonGetCounters()	Obtain FM controller utilization parameters.

All the IOCTL-mapped LLD APIs are what the LLD terms as "callable at runtime", i.e. callable after the LLD `Init()` function for the corresponding entity has been called. This is so because by the time the user app. gets to invoke `ioctl()`, all the `Init()` functions have already been called by the initialization code of the Linux FMD at boot time.

5.2.4.15 The Linux PCD Device

There is exactly one PCD device, or `/dev/fmX_pcd`, for each Frame Manager. The reason for that is that PCDs are FMan-wide constructs, and are applied simultaneously to traffic being received on possibly more than one port.

"PCD" is a generic term designating a Parse-Classify-Distribute configuration for a group of ports, as described in detail in the **QorIQ Data Path Acceleration Architecture (DPAA) Reference Manual**. In short, what a PCD does is to route incoming traffic from a set of RX ports onto several frame queues managed by the Queue Manager. Such frame queues may be attached to a DPAA Ethernet network device, in which case the traffic is received by the CPUs (or "terminated"), or they can be connected to a TX port, in which case the traffic is being forwarded onto that port. Also, frame queues can be further grouped into work queues & policed, etc. (please read the QMan documentation). However, one thing is not supported in the Linux environment, and that is: direct access to frame queues from user space (please note that this is not a limitation of the Linux FMD, but one enforced by design in the Linux driver for the QMan). Not in the classical meaning of "Linux environment", that is. If one needs to create complex DPAA scenarios that are not possible/cumbersome using the current Linux FMD, then USDPAA may hold the answer they're looking for.

There's still a lot that can be achieved with the Linux FMD, and the Linux PCD device is there to help. Its role is to manage the PCDs for its associated FMan. The `ioctl`s for this device are mapped to the similarly-sounding `FM_PCD_*()` LLD APIs:

Table 16. IOCTL List for the PCD Device

IOCTL	LLD Mapping	Brief
FM_PCD_IOC_ENABLE	FM_PCD_Enable()	Should be called after PCD is initialized for enabling all PCD engines according to their existing configuration.
FM_PCD_IOC_DISABLE	FM_PCD_Disable()	Disables an existing PCD.
FM_PCD_IOC_PRs_LOAD_SW[_COMPAT]	FM_PCD_PrLoadSw()	This routine may be called only when all ports in the system are actively using the classification plan scheme. In such cases it is recommended in order to save resources. The driver automatically saves 8 classification plans for ports that do NOT use the classification plan mechanism; to avoid this (in order to save those entries) this routine may be called.
FM_PCD_IOC_KG_SET_DFLT_VALUE	FM_PCD_KgSetDfltValue()	Sets a global default value to be used by the key generator when the parser does not recognize a required field/header (default 0).
FM_PCD_IOC_KG_SET_ADDITIONAL_DATA_AFTER_PARSING	FM_PCD_KgSetAdditionalDataAfterParsing()	Calling this routine allows the keygen to access data past the parser finishing point.
FM_PCD_IOC_SET_EXCEPTION	FM_PCD_SetException()	Enables/disables PCD interrupts.
FM_PCD_IOC_GET_COUNTER	N/A	Unimplemented, do not use!
FM_PCD_IOC_SET_COUNTER	N/A	Placeholder, do not use!
FM_PCD_IOC_FORCE_INTR	FM_PCD_ForceIntr()	Forces a PCD interrupt (exception) of specified type. Dangerous! Use only for debugging!
FM_PCD_IOC_NET_ENV_CHARACTERISTICS_SET[_COMPAT]	FM_PCD_NetEnvCharacteristicsSet()	Establishes a minimal set of networking protocols ("Network Environment Characteristics") that can be discovered by this PCD (please refer to the Reference Manual for details).
FM_PCD_IOC_NET_ENV_CHARACTERISTICS_DELETE[_COMPAT]	FM_PCD_NetEnvCharacteristicsDelete()	Deletes a set of "Network Environment Characteristics".

Table continues on the next page...

Table 16. IOCTL List for the PCD Device (continued)

IOCTL	LLD Mapping	Brief
FM_PCD_IOC_KG_SCHEME_SET[_COMPAT]	FM_PCD_KgSchemeSet()	Initializes or modifies and enables a scheme for the KeyGen. This routine should be called for adding or modifying a scheme. When a scheme needs modifying, the API requires that it be rewritten. In such a case <code>modify</code> should be TRUE. If the routine is called for a valid scheme and <code>modify</code> is FALSE, it will return error.
FM_PCD_IOC_KG_SCHEME_DELETE[_COMPAT]	FM_PCD_KgSchemeDelete()	Deletes an initialized scheme.
FM_PCD_IOC_CC_ROOT_BUILD[_COMPAT]	FM_PCD_CcRootBuild()	This routine must be called to define a complete coarse classification tree. This is the way to define coarse classification to a certain flow - the KeyGen schemes may point only to trees defined in this way.
FM_PCD_IOC_CC_ROOT_DELETE[_COMPAT]	FM_PCD_CcRootDelete()	Deletes an existing coarse classification tree.
FM_PCD_IOC_MATCH_TABLE_SET[_COMPAT]	FM_PCD_MatchTableSet()	This routine should be called for each CC (coarse classification) node. The whole CC tree should be built bottom up so that each node points to already defined nodes. <code>p_node_id</code> returns the node Id to be used by other nodes.
FM_PCD_IOC_MATCH_TABLE_DELETE[_COMPAT]	FM_PCD_MatchTableDelete()	Deletes a built node.
FM_PCD_IOC_CC_ROOT_MODIFY_NEXT_ENGINE[_COMPAT]	FM_PCD_CcRootModifyNextEngine()	Modifies the Next Engine Parameters in the entry of the tree (allowed only after <code>FM_PCD_CcBuildTree()</code>).
FM_PCD_IOC_MATCH_TABLE_MODIFY_NEXT_ENGINE[_COMPAT]	FM_PCD_MatchTableModifyNextEngine()	Modifies the Next Engine Parameters in the relevant key entry of the node (possible only after a call to <code>FM_PCD_MatchTableSet()</code>).
FM_PCD_IOC_MATCH_TABLE_MODIFY_MISS_NEXT_ENGINE[_COMPAT]	FM_PCD_MatchTableModifyMissNextEngine()	Modifies the Next Engine Parameters of the Miss key case of the node (allowed only after a previous call to <code>FM_PCD_MatchTableSet()</code>).
<i>Table continues on the next page...</i>		

Table 16. IOCTL List for the PCD Device (continued)

IOCTL	LLD Mapping	Brief
FM_PCD_IOC_MATCH_TABLE_REMOVE_KEY[_COMPAT]	FM_PCD_MatchTableRemoveKey()	Removes the key (including its next engine parameters) defined by the index of the relevant node (allowed only after a previous call to FM_PCD_MatchTableSet())
FM_PCD_IOC_MATCH_TABLE_ADD_KEY[_COMPAT]	FM_PCD_MatchTableAddKey()	Adds the key (including next engine parameters of this key) in the index defined by key_index (allowed only after a previous call to FM_PCD_MatchTableSet())
FM_PCD_IOC_MATCH_TABLE_MODIFY_KEY_AND_NEXT_ENGINE[_COMPAT]	FM_PCD_MatchTableModifyKeyAndNextEngine()	Modifies the key and Next Engine Parameters of this key in the index defined by key_index (allowed only after a previous call to FM_PCD_MatchTableSet()).
FM_PCD_IOC_MATCH_TABLE_MODIFY_KEY[_COMPAT]	FM_PCD_MatchTableModifyKey()	Modifies the key at the index defined by key_index (allowed only after a previous call to FM_PCD_MatchTableSet()).
FM_PCD_IOC_HASH_TABLE_SET[_COMPAT]	FM_PCD_HashTableSet()	Initializes a hash table structure.
FM_PCD_IOC_HASH_TABLE_DELETE[_COMPAT]	FM_PCD_HashTableDelete()	Deletes the provided hash table and released all its allocated resources.
FM_PCD_IOC_HASH_TABLE_ADD_KEY[_COMPAT]	FM_PCD_HashTableAddKey()	Adds the provided key (including next engine parameters of this key) to the hash table. The key is added as the last key of the bucket that it is mapped to.
FM_PCD_IOC_HASH_TABLE_REMOVE_KEY[_COMPAT]	FM_PCD_HashTableRemoveKey()	Removes the requested key (including its next engine parameters) from the hash table.
FM_PCD_IOC_PLCR_PROFILE_SET[_COMPAT]	FM_PCD_PlcrProfileSet()	Sets a profile entry in the policer profile table, overriding any existing value.
FM_PCD_IOC_PLCR_PROFILE_DELETE[_COMPAT]	FM_PCD_PlcrProfileDelete()	Deletes a profile entry in the policer profile table. It sets the entry to invalid.
FM_PCD_IOC_MANIP_NODE_SET[_COMPAT]	FM_PCD_ManipNodeSet()	This routine should be called for defining a manipulation node. A manipulation node must be defined before the CC node that precedes it.

Table continues on the next page...

Table 16. IOCTL List for the PCD Device (continued)

IOCTL	LLD Mapping	Brief
FM_PCD_IOC_MANIP_NODE_REPLACE[_COMPAT]	FM_PCD_ManipNodeReplace()	Change existing manipulation node to be according to new requirement.
FM_PCD_IOC_MANIP_NODE_DELETE[_COMPAT]	FM_PCD_ManipNodeDelete()	Deletes an existing manipulation node.
FM_PCD_IOC_SET_ADVANCED_OFFLOAD_SUPPORT	FM_PCD_SetAdvancedOffloadSupport()	This routine must be called in order to support the following features: IP-fragmentation, IP-reassembly, IPsec, header manipulation, frame replicator.
FM_PCD_IOC_FRM_REPLIC_GROUP_SET[_COMPAT]	FM_PCD_FrmReplicSetGroup()	Initialize a Frame Replicator group.
FM_PCD_IOC_FRM_REPLIC_GROUP_DELETE[_COMPAT]	FM_PCD_FrmReplicDeleteGroup()	Delete a Frame Replicator group.
FM_PCD_IOC_FRM_REPLIC_MEMBER_ADD[_COMPAT]	FM_PCD_FrmReplicAddMember()	Add the member in the index defined by the memberIndex.
FM_PCD_IOC_FRM_REPLIC_MEMBER_REMOVE[_COMPAT]	FM_PCD_FrmReplicRemoveMember()	Remove the member defined by the index from the relevant group.
FM_PCD_IOC_STATISTICS_SET_NODE[_COMPAT]	FM_PCD_StatisticsSetNode()	Not implemented in this release. Do not use!
FM_PCD_IOC_KG_SCHEME_GET_CNTR	FM_PCD_KgSchemeGetCounter()	Reads scheme packet counter.

NOTE

The `_COMPAT` variants of certain IOCTLs in the above table are required for supporting 32-bit user space apps. on 64-bit Linux kernels. The specifics of the `COMPAT` mappings are documented by Linux.

The programming model for defining and managing PCDs for a group of ports is the same as described in the **FMD LLD User's Guide**.

What follows is a step-by-step description of an example of `ioctl()` call mapping to a LLD API call.

The example chosen for this walk-through is that of `FM_PCD_IOC_MATCH_TABLE_SET`. Here's a reminder of the `ioctl()` prototype:

```
extern int ioctl (int __fd, unsigned long int __request, ...) __THROW;
```

and below is how it appears to kernel space:

```
struct file_operations {
  [...]
  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
  [...]
};
```

The `ioctl()` function is actually a pointer to a driver-supplied function having the specified signature. The glue between the two is kernel code.

The semantics associated with the second and third function arguments are entirely the driver's business, but usually the `unsigned int` argument is used to discriminate between various `ioctl` commands (actually, it should obey some Linux good-behavior rules, which we are not going to detail here). In our case, it should be `FM_PCD_IOC_MATCH_TABLE_SET`.

Linux attaches no predefined semantics to the third argument, the `unsigned long` one. In some cases it is unused, or its semantics are those of an unsigned integer number, but in most cases it is treated as a (32-bit, on most platforms) pointer to a driver-defined structure in user space. The driver defines the format, but the user space allocates and fills in the data prior to invoking `ioctl()` on the open device `fd`. This is also the case with our example.

The format of the third argument of the `FM_PCD_IOC_MATCH_TABLE_SET` `ioctl` is (as it actually appears in the header file where it's defined):

```

/*****
 @Description  A structure for defining the CC node params
 **/
typedef struct ioc_fm_pcd_cc_node_params_t {
    ioc_fm_pcd_extract_entry_t extract_cc_params;
                                /**< params which defines extraction
                                parameters */

    ioc_keys_params_t          keys_params;    /**< params which defines Keys
                                parameters of the extraction defined
                                in extract_cc_params */

    void                       *id;          /**< output parameter;
                                Returns the CC node Id to be used */
} ioc_fm_pcd_cc_node_params_t;

```

We'll detail the `ioc_*` types of the first two members later. The third member of this structure is apparently a pointer to some data structure being returned back to user space. It is not the case. This actual pointer should be handled as an opaque handle to some abstract item, in our case the "CC Node" that's being created for us by this `ioctl()` call if successful. This handle can be later passed to e.g. the `FM_PCD_IOC_MATCH_TABLE_DELETE` IOCTL for deletion. It corresponds to an actual `t_Handle`, as defined by the LLD.

NOTE

Failing to cleanup FMan resources that the LLD allocates in this manner can cause serious hardware resource leaks, which neither the Linux FMD, nor the LLD have the means to detect & cleanup automatically!

The LLD function that this IOCTL maps to has the following prototype:

```

t_Handle FM_PCD_MatchTableSet(t_Handle, t_FmPcdCcNodeParams *);

```

The first argument corresponds to the LLD resource that the Linux PCD device maps to. Most of the LLD resources are managed within the Linux FMD driver and not exposed to the user, but there are exceptions and the `FM_PCD_MatchTableSet()` function here is the best example, as it returns a `t_Handle` to such a LLD resource. This returned `t_Handle` is then passed over to the user space in the opaque `id` member of `ioctl()`'s third argument.

The second argument is a pointer to a structure of type `t_FmPcdCcNodeParams`. This maps to the `ioc_fm_pcd_cc_node_params_t` type that `ioctl()`'s third argument points to.

NOTE

Passing to `ioctl()` a pointer to something of a type other than the required one will cause the user application to segfault, or an error, at best, but may also cause undefined FMan behavior from that point onward, with errors being possibly reported only later downstream as the worst case. Linux/the FMD can do very little to prevent this worst case from occurring, so hopefully one can catch such coding errors early during the development cycle.

A side-by-side comparison of the two structures is given in the following table:

Table 17. Side-by-side comparison of IOCTL and LLD types

IOCTL Types	LLD Types
<pre>typedef struct ioc_fm_pcd_cc_node_params_t { ioc_fm_pcd_extract_entry_t extract_cc_params; ioc_keys_params_t keys_params; void *id; } ioc_fm_pcd_cc_node_params_t;</pre>	<pre>typedef struct t_FmPcdCcNodeParams { t_FmPcdExtractEntry extractCcParams; t_KeysParams keysParams; } t_FmPcdCcNodeParams;</pre>
<pre>typedef struct ioc_fm_pcd_extract_entry_t { ioc_fm_pcd_extract_type type; union { struct { ioc_net_header_type hdr; bool ignore_protocol_validation; ioc_fm_pcd_hdr_index hdr_index; ioc_fm_pcd_extract_by_hdr_type type; union { ioc_fm_pcd_from_hdr_t from_hdr; ioc_fm_pcd_from_field_t from_field; ioc_fm_pcd_fields_u full_field; } extract_by_hdr_type; } extract_by_hdr; struct{ ioc_fm_pcd_extract_from src; ioc_fm_pcd_action action; uint16_t ic_indx_mask; uint8_t offset; uint8_t size; } extract_non_hdr; } extract_params; } ioc_fm_pcd_extract_entry_t;</pre>	<pre>typedef struct t_FmPcdExtractEntry { e_FmPcdExtractType type; union { struct { e_NetHeaderType hdr; bool ignoreProtocolValidation; e_FmPcdHdrIndex hdrIndex; e_FmPcdExtractByHdrType type; union { t_FmPcdFromHdr fromHdr; t_FmPcdFromField fromField; t_FmPcdFields fullField; } extractByHdrType; } extractByHdr; struct { e_FmPcdExtractFrom src; e_FmPcdAction action; uint16_t icIndxMask; uint8_t offset; uint8_t size; } extractNonHdr; }; } t_FmPcdExtractEntry;</pre>
<pre>typedef struct ioc_keys_params_t { uint16_t max_num_of_keys; bool mask_support; ioc_fm_pcd_cc_stats_mode statistics_mode; uint16_t num_of_keys; uint8_t key_size; ioc_fm_pcd_cc_key_params_t key_params[IOC_FM_PCD_MAX_NUM_OF_KEYS]; ioc_fm_pcd_cc_next_engine_params_t cc_next_engine_params_for_miss; } ioc_keys_params_t;</pre>	<pre>typedef struct t_KeysParams { uint16_t maxNumOfKeys; bool maskSupport; ioc_fm_pcd_cc_stats_mode statisticsMode; uint16_t numOfKeys; uint8_t keySize; t_FmPcdCcKeyParams keyParams[FM_PCD_MAX_NUM_OF_KEYS]; t_FmPcdCcNextEngineParams ccNextEngineParamsForMiss; } t_KeysParams;</pre>

While the structure members have resembling names on both sides, most are not identical. That's because style has prevailed over the need to port existing LLD applications to the Linux environment, when the Linux FMD was designed (there is a more complete porting guide included in the NCSW software bundle, though, that's not also included in this SDK; one may refer to that for the specific topic of porting existing NCSW apps. to various platforms). Except for the occasional `*id` pointer, there is a 1:1 mapping between the struct members on the two sides, and that is consistent throughout the FMD.

The constituent structures of the two APIs' argument types given above are for illustration only. Their semantics are documented in the [Frame Manager Driver API Documentation](#).

NOTE

The existence of two separate definitions for otherwise two identical data structures may appear as an unfortunate design decision. However, since a `memcpy` from user space to kernel space is unavoidable, this design decision has no impact over performance. Moreover, the user space only sees one variant (i.e. the `ioc_*` one), hence the even smaller user impact. The larger impact is on code maintenance and on documentation.

5.2.4.1.6 The Linux Port Devices

There is a pair of RX/TX Linux character devices for each physical port of every Frame Manager. These devices are created irrespectively of the DPAAs Ethernet network devices and they are strictly reflecting the available Frame Manager hardware on the given platform. The port Linux devices are labeled as follows:

- `/dev/fmX_port_rxY` for receive, where `X=[0,1]` represents the FMan number, and `Y=[0-5]` represents the physical port ID (0 corresponding to the first 1 Gb port, and 5 to the 10 Gb port), and
- `/dev/fmX_port_txY` correspondingly for the transmit side.

Each FMan also has a number of Offline Parsing ports. These are labeled as `/dev/fmX_port_ohY`, where `Y=[0-6]`.

The port devices are created based on configuration information taken from the relevant Linux device tree section.

For instance, P5020 only has one FMan with 5 x 1Gb ports and one 10Gb port, while P4080 has two FMans, each having 4 x 1Gb and 1 x 10Gb ports. A side-by-side comparison of the corresponding port devices is given in the following table:

Table 18. Side-by-side comparison of port devices for P5020 and P4080.

P5020	P4080
<p>For the Receive side:</p> <pre> /dev/fm0_port_rx0 /dev/fm0_port_rx1 /dev/fm0_port_rx2 /dev/fm0_port_rx4 /dev/fm0_port_rx5 </pre>	<p>For the Receive side:</p> <pre> /dev/fm0_port_rx0 /dev/fm0_port_rx1 /dev/fm0_port_rx2 /dev/fm0_port_rx3 /dev/fm0_port_rx5 /dev/fm1_port_rx0 /dev/fm1_port_rx1 /dev/fm1_port_rx2 /dev/fm1_port_rx3 /dev/fm1_port_rx5 </pre>
<i>Table continues on the next page...</i>	

Table 18. Side-by-side comparison of port devices for P5020 and P4080. (continued)

P5020	P4080
For the Transmit side: <pre> /dev/fm0_port_tx0 /dev/fm0_port_tx1 /dev/fm0_port_tx2 /dev/fm0_port_tx3 /dev/fm0_port_tx4 /dev/fm0_port_tx5 </pre>	For the Transmit side: <pre> /dev/fm0_port_tx0 /dev/fm0_port_tx1 /dev/fm0_port_tx2 /dev/fm0_port_tx3 /dev/fm0_port_tx5 /dev/fm1_port_tx0 /dev/fm1_port_tx1 /dev/fm1_port_tx2 /dev/fm1_port_tx3 /dev/fm1_port_tx5 </pre>
For Offline Parsing: <pre> /dev/fm0_port_oh0 /dev/fm0_port_oh1 /dev/fm0_port_oh2 /dev/fm0_port_oh3 /dev/fm0_port_oh4 /dev/fm0_port_oh5 /dev/fm0_port_oh6 </pre>	For Offline Parsing: <pre> /dev/fm0_port_oh0 /dev/fm0_port_oh1 /dev/fm0_port_oh2 /dev/fm0_port_oh3 /dev/fm0_port_oh4 /dev/fm0_port_oh5 /dev/fm0_port_oh6 /dev/fm1_port_oh0 /dev/fm1_port_oh1 /dev/fm1_port_oh2 /dev/fm1_port_oh3 /dev/fm1_port_oh4 /dev/fm1_port_oh5 /dev/fm1_port_oh6 </pre>

NOTE

The `/dev/fmX_port_(rx4|tx4)` pairs of port devices are missing in the P4080, for numbering consistency reasons: this way the 10G ports will have the same IDs on every QorIQ platform that has them, thus greatly reducing the possibility for confusion, and adding to cross-platform code portability.

The table below summarizes the IOCTLs available for the port device.

Table 19. IOCTLs of the Port Device

IOCTLS	LLD Mapping	Brief
FM_PORT_IOC_DISABLE	FM_PORT_Disable()	Disables the port: all port settings are preserved, but all traffic stops.
FM_PORT_IOC_ENABLE	FM_PORT_Enable()	Enables the port: causes the port to start processing traffic.

Table continues on the next page...

Table 19. IOCTLs of the Port Device (continued)

IOCTLS	LLD Mapping	Brief
FM_PORT_IOC_SET_RATE_LIMIT	FM_PORT_SetRateLimit()	(TX & O/H Only) Activates the Rate Limiting Algorithm for the port.
FM_PORT_IOC_DELETE_RATE_LIMIT	FM_PORT_DeleteRateLimit()	(TX & O/H Only) Deactivates any Rate Limiting.
FM_PORT_IOC_SET_ERRORS_ROUTE	FM_PORT_SetErrorsRoute()	(RX & O/H Only) Instructs the FMD to enqueue frames w/specific errors onto the normal port queues, rather than onto the error queue (i.e. the default).
FM_PORT_IOC_ALLOC_PCD_FQIDS	N/A	For testing/debugging. Do not use!
FM_PORT_IOC_FREE_PCD_FQIDS	N/A	For testing/debugging. Do not use!
FM_PORT_IOC_SET_PCD[_COMPAT]	FM_PORT_SetPCD()	(RX & O/H Only) Defines a PCD configuration for the port.
FM_PORT_IOC_DELETE_PCD	FM_PORT_DeletePCD()	(RX & O/H Only) Deletes the port's PCD configuration.
FM_PORT_IOC_DETACH_PCD	FM_PORT_DetachPCD()	(RX & O/H Only) Disables the PCD configuration for the port (only allowed after FM_PORT_SetPCD() has been called for the port).
FM_PORT_IOC_ATTACH_PCD	FM_PORT_AttachPCD()	(RX & O/H Only) Re-enables the PCD configuration for the port (only valid after a call to FM_PORT_DetachPCD()).
FM_PORT_IOC_PCD_PLCR_ALLOC_PROFILES	FM_PORT_PcdPlcrAllocProfiles()	(RX & O/H Only) Allocates private policer profiles for the port (only allowed before a call to FM_PORT_SetPCD()).
FM_PORT_IOC_PCD_PLCR_FREE_PROFILES	FM_PORT_PcdPlcrFreeProfiles()	(RX & O/H Only) Frees any private policer profiles allocated for the port (callable only before FM_PORT_SetPCD()).
FM_PORT_IOC_PCD_KG_MODIFY_INITIAL_SCHEME[_COMPAT]	FM_PORT_PcdKgModifyInitialScheme()	(RX & O/H Only) Modifies key generation scheme following frame parsing (callable only after FM_PORT_SetPCD()).
FM_PORT_IOC_PCD_PLCR_MODIFY_INITIAL_PROFILE[_COMPAT]	FM_PORT_PcdPlcrModifyInitialProfile()	(RX & O/H Only) Changes the initial policer profile for the port (callable only after FM_PORT_SetPCD()).

Table continues on the next page...

Table 19. IOCTLs of the Port Device (continued)

IOCTLS	LLD Mapping	Brief
FM_PORT_IOC_PCD_CC_MODIFY_TREE[_COMPAT]	FM_PORT_PcdCcModifyTree()	(RX & O/H Only) Replaces the coarse classification tree if one is used for the port (callable only after FM_PORT_DetachPCD() and before FM_PORT_AttachPCD()).
FM_PORT_IOC_PCD_KG_BIND_SCHEMES[_COMPAT]	FM_PORT_PcdKgBindSchemes()	(RX & O/H Only) Adds more KeyGen schemes for the port to be bound to (callable only after FM_PORT_SetPCD()).
FM_PORT_IOC_PCD_KG_UNBIND_SCHEMES[_COMPAT]	FM_PORT_PcdKgUnbindSchemes()	(RX & O/H Only) Prevents the port from using the specified KG schemes (callable only after FM_PORT_SetPCD())
FM_PORT_IOC_PCD_PRS_MODIFY_START_OFFSET	FM_PORT_PcdPrsModifyStartOffset()	(RX & O/H Only) Changes the frame offset at which parsing starts (callable only after FM_PORT_DetachPCD() and before FM_PORT_AttachPCD()).
FM_PORT_IOC_ADD_CONGESTION_GRP	FM_PORT_AddCongestionGrps()	(RX & O/H Only) Should be called in order to enable pause frame transmission in case of congestion in one or more of the congestion groups relevant to this port. Each call to this routine may add one or more congestion groups to be considered relevant to this port.
FM_PORT_IOC_REMOVE_CONGESTION_GROUPS	FM_PORT_RemoveCongestionGrps()	(RX & O/H Only) Should be called when congestion groups were defined for this port and are no longer relevant, or pause frames transmitting is not required on their behalf. Each call to this routine may remove one or more congestion groups to be considered relevant to this port.
FM_PORT_IOC_ADD_RX_HASH_MAC_ADDR	FM_MAC_AddHashMacAddr()	Add an Address to the hash table. This is for filter purpose only.
FM_PORT_IOC_REMOVE_RX_HASH_MAC_ADDR	FM_MAC_RemoveHashMacAddr()	Delete an Address to the hash table. This is for filter purpose only.
FM_PORT_IOC_SET_TX_PAUSE_FRAMES	FM_MAC_SetTxPauseFrames()	Enable/Disable transmission of Pause-Frames. The routine changes the default configuration: pause-time - [0xf000], threshold-time - [0]

Table continues on the next page...

Table 19. IOCTLs of the Port Device (continued)

IOCTLS	LLD Mapping	Brief
FM_PORT_IOC_GET_MAC_STATISTICS	FM_MAC_GetStatistics()	Get all MAC statistics counters.
FM_PORT_IOC_CONFIG_BUFFER_PREFIX_CONTENT	FM_PORT_ConfigBufferPrefixContent()	Defines the structure, size and content of the application buffer.
FM_PORT_IOC_VSP_ALLOC[COMPAT]	FM_PORT_VSPAlloc()	This routine allocated VSPs per port and forces the port to work in VSP mode. Note that the port is initialized by default with the physical-storage-profile only.

NOTE

The COMPAT variants of certain IOCTLs in the above table are required for supporting 32-bit user space apps. on 64-bit Linux kernels. The specifics of the COMPAT mappings are documented by Linux.

The programming model for managing the FMan's ports is the same as described in the *Frame Manager Driver API Reference*. A few notable mentions though:

Although all the above IOCTLs are implemented by the Linux FMD, due to the asymmetry between RX and TX, not all are available for any port type. E.g. FM_PORT_IOC_SET_PCD will generate an error if called on a TX port device. Similarly, FM_PORT_IOC_SET_RATE_LIMIT will fail for an RX port. That is because the checking of the port type is being done late, inside the LLD, and not in the Linux FMD (i.e. the ioctl() calls for all port devices delegate to the same function inside the Linux kernel)!

The Offline Parsing ports have the best of both worlds. That is because conceptually, an O/H port is no different from a "regular" FMan port that has the TX side looped back internally to its RX side.

5.2.4.2 Frame Manager Linux Driver API Reference

This document describes the interface (IOCTLs) to the Frame Manager Linux Driver as apparent to user space Linux applications that need to use any of the Frame Manager's features. It describes the structure, concept, functionality, and high level API.

The link below leads to a supplemental directory. Download the file you need from this set.

For more information see [Frame Manager Linux Driver API Reference](#)

5.2.4.3 Frame Manager Driver User's Guide

5.2.4.3.1 Frame Manager Driver

5.2.4.3.1.1 Introduction

The Frame Manager is a hardware accelerator responsible for preprocessing and moving packets into and out of the datapath. It supports in-line/off-line packet parsing and initial classification to enable policing and flow/QoS based packet distribution to the CPUs for further processing of the packets.

The Frame Manager consists of a number of packet processing elements (also referred to as engines) and supports a flexible pipeline. Usually, the main Rx flow (simplified) follows these steps: packets are received from one of the Ethernet MACs, are temporarily stored in the FMan internal memory, then delivered to SoC memory via the FMan DMA. The packet header (max size 256 bytes) is stored and the modules common database structure is allocated. Then the packet is parsed by the parser or by the FMan controller. According to parsing results a key may be extracted by KeyGen, a destination frame-queue-id may be set, the packet may be classified by the FMan controller. In that stage, some offloads may be done like re-assembly,

fragmentation, header-manipulation and frame-replication. At the end of the classification and manipulations stage, the packet may be colored by policer. At the end of this process, packets are delivered to SoC memory via the FMan DMA and then are enqueued to a frame queue or dropped. The processing order is Parse-Classify-Distribute (PCD) flow dependant, based on user configurations. Each step is dependant on previous state results. This structure enables flexibility, which efficiently supports many flows.

On Tx the frames are transmitted via the desired MAC with optional checksum generation.

5.2.4.3.1.2 Frame Manager Features

The FMan driver aims to support the majority of the hardware features. It also includes exclusive software features designed to provides facilitation through abstraction.

Following are the features of the FMan driver:

- Simple initialization and configuration API for the following FMan blocks: DMA, FPM, IRAM, QMI, BMI, and RTC.
- Simple initialization and configuration for the following FMan PCD blocks: Parser, Keygen, Custom-Classifer (CC), Manipulations (e.g. Header-manipulations, IP-reassembly, IP-fragmentation, etc.) and Policer.
- FMan memory (MURAM) management.
- FMan-controller code loading.
- Software-Parser loading.
- Supported all FMan port types-Rx, Tx, Offline-Parsing, and Host-Command (internal use of the driver only)
- Common MAC API for dTSEC, 10G-MAC and mEMAC.
- Provides API for accessing the MII management interface.
- FMan Rx and Tx ports can run in one of the following modes:
 - Independent-Mode
 - Simple BMI-to-BMI (regular) mode
 - Advance PCD mode (using FMan PCD blocks such as parser, Keygen, CC, and Policer).
- FMan Offline ports can run in one of the following modes:
 - Simple BMI-to-BMI (regular) mode
 - Advance PCD mode (using FMan PCD blocks such as parser, Keygen, CC, and Policer)
- Internal (optional) Host-Command port initialization, based on user's parameters.
- FMan IRQ handling - events and exceptions.
- Supports both SMP and AMP operation modes.

5.2.4.3.1.3 Frame Manager Driver Components

The FMan driver contains following low-level modules, as shown in this figure.

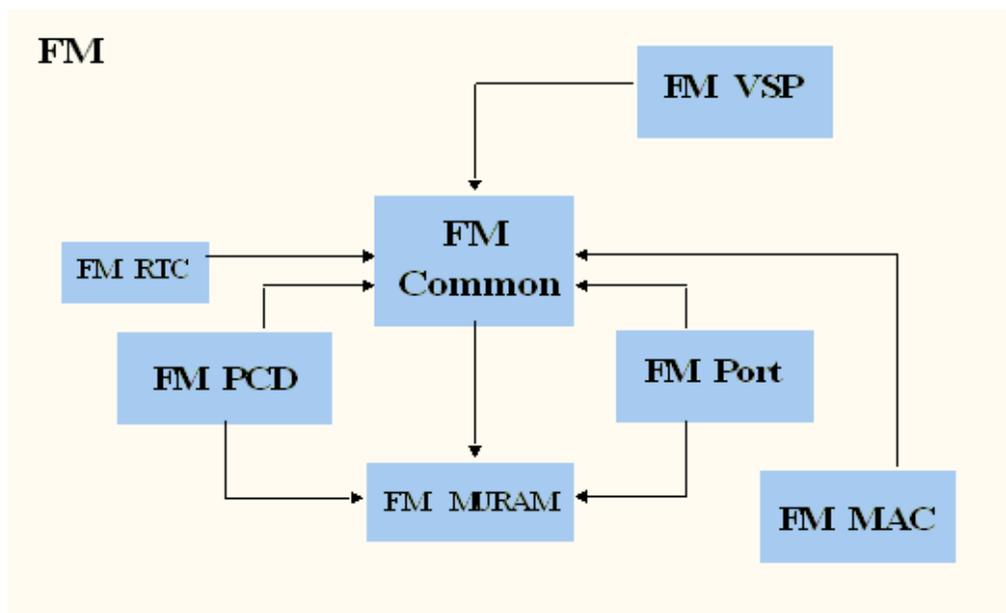


Figure 27. FMan Driver Modules (from a partition point of view)

The modules are as follows:

- **Frame Manager (common)**-The FMan module is a singleton module within its partition. It is responsible for the common hardware modules: FPM, DMA, common QMI, common BMI, FMan controller's initialization, and runtime control routines. This module must always be initialized when working with any FMan module. The module will mainly be used internally by the other FMan modules except for its initialization by the user.

This module has an instance for each partition. However, only the driver that is on the master-partition has access to the hardware registers.

- **Frame Manager Parser-Classifer-Distributor (FMan-PCD)**-The FMan PCD module is a singleton module within its partition. It is responsible of all common parts of the PCD, such as the hardware parser, software parser, Keygen, policer, and custom-classifier blocks. It is responsible for building the PCD graphs.

This module has an instance for each partition. However, only the driver on the master-partition has access to the hardware registers.

- **Frame Manager Memory (FMan-MURAM)**-This module is responsible for the specific memory partition of the FMan Memory. Each partition may have its own FMan Memory partition that is managed by the FMan Memory driver. For example, an FMan Memory instance will be created for each partition that has its own FMan ports.

This module has an instance for each partition.

- **Frame Manager Real-Time-Clock (FMan-RTC)**-This module is responsible for the FMan RTC module.

This module is a "singleton" and should be created once only for the master-partition.

- **Frame Manger Port (FMan-Port)**-This module is responsible for all FMan port-related register space, such as all registers related to a port in QMI or BMI.

This module can be run by each core or partition independently.

- **Frame Manager MAC (FMan-MAC)**-This module is responsible for the mEMAC dTSEC and the 10G MAC controllers.

This module can be run by each core or partition independently.

- **Frame Manager Virtual-Storage-Profile (FMan-VSP)**-This module is responsible for allocating and managing virtual storage profiles that may used for virtualization purposes. More of the VSP is described in [FMan VSP Driver](#) on page 242.

This module can be run by each core or partition independently.

5.2.4.3.1.4 Driver Modules in the System

The FMan driver is designed to support single or multi partition environment. In addition, the FMan driver is designed to support environment with multicore that are running in SMP mode.

The following figure shows a typical single-partition (maybe SMP or not) environment and its FMan driver building blocks.

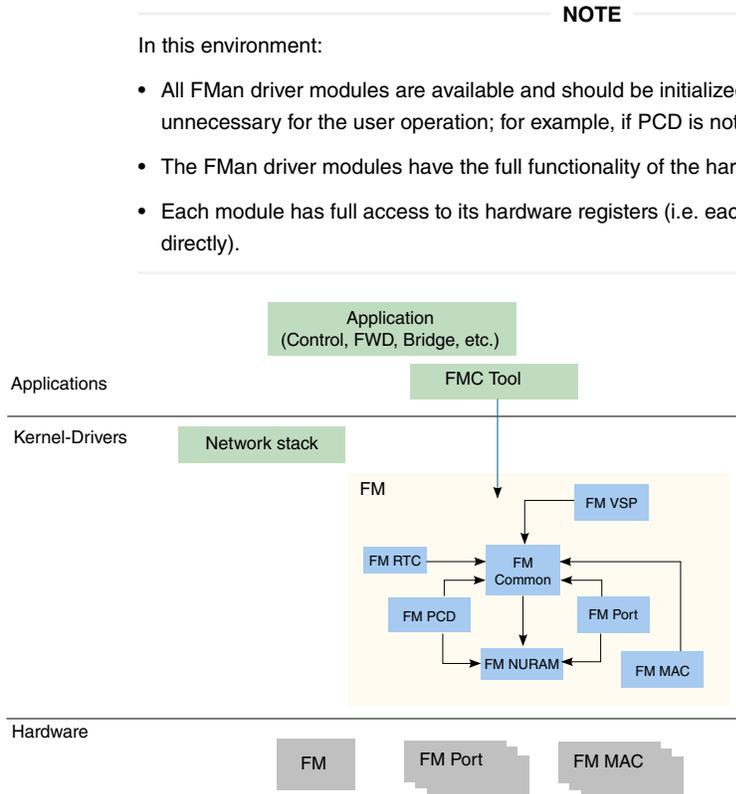


Figure 28. Single-Partition FM Building Blocks

5.2.4.3.1.4.1 Multicore Approach

The driver supports both Symmetric Multi-Processing (SMP) and Asymmetric Multi-Processing (AMP) operation methods.

5.2.4.3.1.4.1.1 SMP

As a rule, driver routines are not SMP safe. It is user's responsibility to lock all routines that might be in risk in his environment, for example, if `FM_PORT_Enable/FM_PORT_Disable` may be used by several cores, it is user's responsibility to protect the routine call using a spinlock.

An exception to this rule is the set of PCD routines. Due to the complexity of this module, and in order to support SMP and maintain coherency, PCD routines are protected using two mechanisms, spinlocks and flags.

Each PCD resource (i.e. software module such as scheme, CC Node, NetEnv, etc.) may have one or more spinlocks which are used to protect short code sections where specific resources such as hardware registers or software structures are accessed. In some cases, a spinlock of a higher level is used (i.e. CC locks the whole PCD).

The second mechanism is defined globally. The PCD global module provides a `PcdLock` mechanism, which is a list of lock objects containing a flag and a spinlock rotating that flag. On initialization of each PCD resource (i.e. software module such as scheme, CC Node, NetEnv, etc.), a `PcdLock` is allocated for this module. Critical sections that may not be protected by spinlocks (due to reasons of sections length, Host Commands and other lengthy operations) are protected by these flags. Note that this is a try-lock mechanism and the calling routine returns with `E_BUSY` error on failure. The try-locks are used by all PCD resources modification routines, in which case the application is expected to recall the routine until it is not busy.

In Addition, PCD and FM Port inter-module complex sections may be protected by try-locking all the initialized PcdLock modules in the global PCD, thus providing a safe PCD environment where influence and connections between modules may take effect.

On top of PCD routines, all FM Port PCD related routines are also protected by Port try-lock, meaning no two cores can access the same port to run a PCD routine. As in the PCD routines, these routines may return `E_BUSY` on failure and should then be recalled.

The driver SMP protection mechanism assumes the following:

- Only one core may initialize and delete a specific PCD software module (i.e. scheme x may not be initialized by two cores).
- A core should not attempt to delete a PCD software module when there is a risk of another core operating on that specific module.

5.2.4.3.1.4.1.2 AMP

FMan driver supports multi-partitioned system in a way that the common modules (`FM`, `FM_PCD`) are designed as "front-end" and "back-end". When configuring the FM driver module, the user should pass the driver its "guestId" ('0' identifies the "back-end"/"master-partition").

NOTE

In order to support AMP, FMan driver uses IPC calls for synchronization between the master module and the guests, i.e. user must have some IPC available in his system in order to have this functionality available in FMan driver.

There is no "hard" relation between "partitions" and "guests". A number of guests may co-exist on the same physical partition, creating a logical partitioning. Note that the FM "master"/"back-end" may run on any partition in the system.

When the FM driver is configured to run as a guest, it may access registers only if there is no race condition. If register access may end with some risk, the FM "front-end" driver access the HW registers through IPC call to the "back-end".

In the driver implementation, the FM-Port and the FM-MAC drivers are indifferent to guest/master considerations and may run on either partition. They do not have a "front-end" and both initialization and runtime routines are locally implemented. This means that a port may not be controlled from several partitions. However, a few ports may be controlled by the same partition.

The FM-MURAM module is pre-allocated and managed independently by each guest that has MURAM consumers (typically for Custom Classifier tables). It does not have a "front-end" and once allocated it may be accessed directly.

The FM-RTC module is a singleton module and is available only on the "back-end"/"master-partition" and therefore has no "front-end".

In the figure below, we can see two partition (A and B), where the FMan driver is configured as a master (the "back-end") on partition A and it is configured as a guest ("front-end") on partition B. In addition, each partition has its own FM-ports (also FM-MACs) that are working with the "front-end" and host-commands.

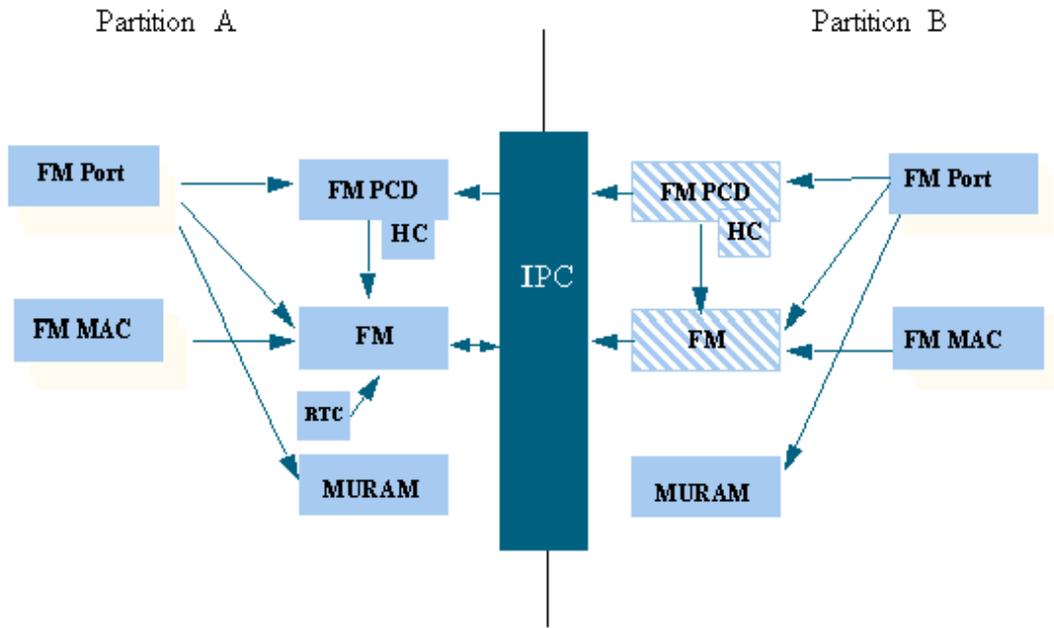


Figure 29. Multi-Partition FM Building Blocks with IPC (DPAA 1.0)

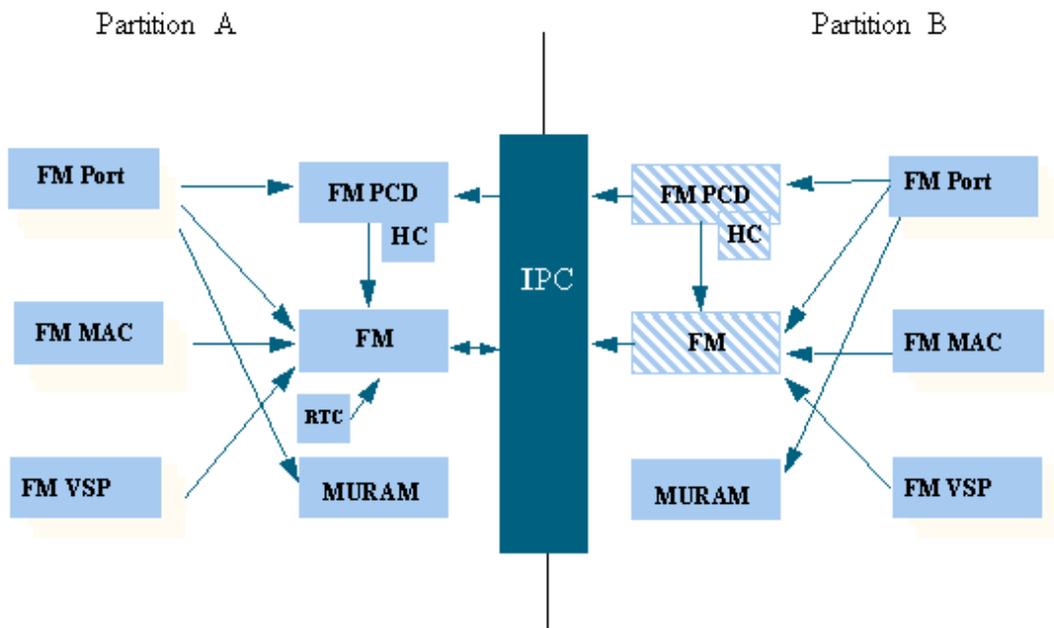


Figure 30. Multi-Partition FM Building Blocks with IPC (DPAA 1.1)

There may be two ways for user to work in AMP:

- AMP with IPC-In systems that has IPC mechanism (as in the figure above), the FM drivers (including FM, FM-PCD, FM-Port, etc.) will utilize the IPC mechanism to actually synchronize information between them; i.e. FM-PCD driver in guest mode will call the FM-PCD master to allocate schemes for its partition (according to user configuration). In this system, it is possible to create FM-Ports and FM-MACs local for guest partition; i.e. the HW of FM-Port and FM-MAC will be accessed and controlled locally for the partition that actually owns the resource.
- AMP without IPC-The FM driver itself doesn't support this kind of system. As shown in the figure below, it is still possible for a user to create DPA-Ports and to send and receive packets to and from MAC. In addition, user may also initialize

and access MACs from guest partitions, but, other operations like creating PCDs or accessing the FM-Ports registers may be done only by the master partition.

NOTE

Although FM-MAC driver can be initialized and accessed from guest partition even if there is no IPC present, some functions may be disabled (e.g. events of the MAC).

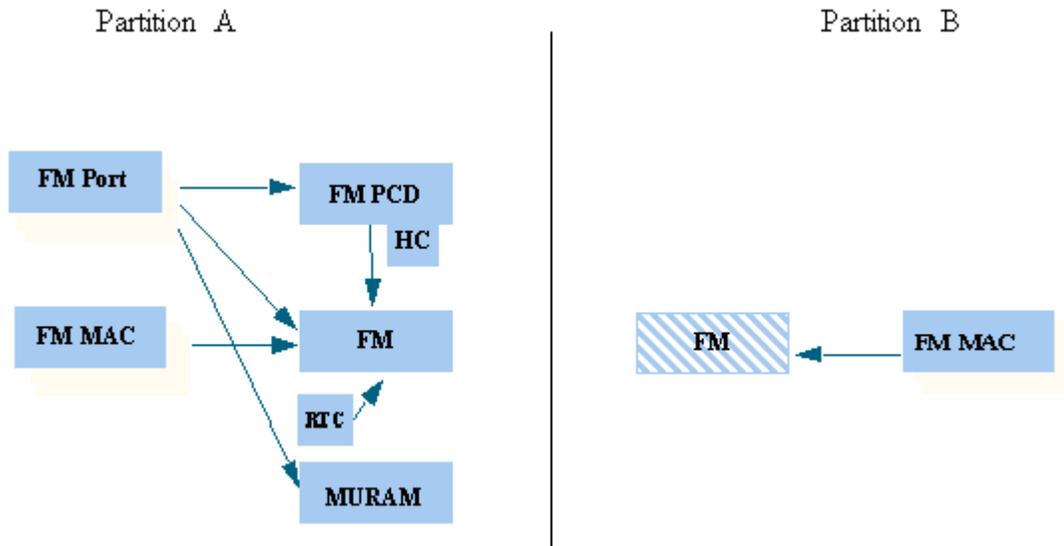


Figure 31. Multi-Partition FM Building Blocks without IPC (DPAA 1.0)

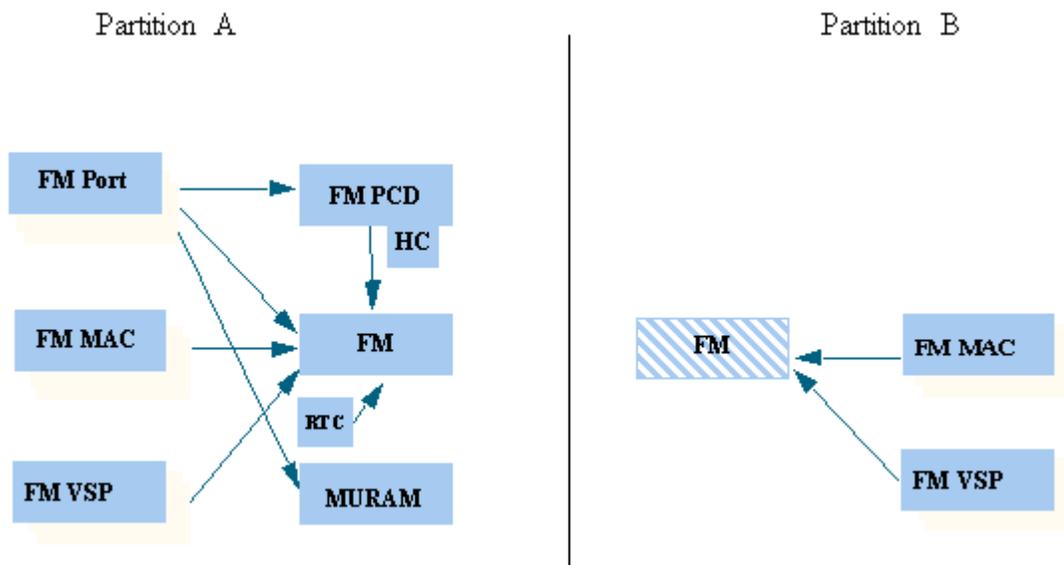


Figure 32. Multi-Partition FM Building Blocks without IPC (DPAA 1.1)

5.2.4.3.1.5 FMan Driver Calling Sequence

Initialization of the FMan driver is carried out by the application according to the following sequence:

1. MURAM configuration & Initialization
2. FMan (common) configuration & Initialization

3. [Optional] FMan RTC configuration & Initialization
4. For each MAC required by the user:
 - a. MAC Configuration & Initialization
 - b. PHY Initialization
5. For each FMan Port required by the user:
 - a. FMan Port Configuration & Initialization
 - b. [optional] If the FMan Port required to be virtualized, a set of VSPs need to be allocated and one of them should be set as the default.
 - c. [optional] If VSPs were allocated in previous step, the default VSP need to be configured & initialized
 - d. in that stage, user should configure and initialize everything that is needed for the operation of a port outside the fman; e.g. buffer-pools, frame-queues, etc.
 - e. Port Enablement
 - f. MAC Enablement
 - g. Calling 'AdjustLink' MAC API routine with the relevant link parameters

NOTE

Now, the FMan is operational. The ports operate in independent mode or BMI-to-BMI mode. From that point, all the following steps are optional.

6. FMan PCD Configuration & Initialization
7. If a physical port is being "virtualized" into several software entities (using some classification to distribute the traffic), user should configure and initialize the relevant buffer-pools and frame-queues.
8. If VSP is enabled, in that stage, user should configure and initialize the relevant profile.
9. FMan PCD Graph initialization:
 - a. Calling restricted runtime routines (that may be called only when PCD is disabled)
 - b. Calling the PCD enable routine
 - c. Initialization of a all PCD Graph objects (i.e. KG-schemes, Match-Tables, etc)
10. FMan port-PCD related initialization; calling the run-time control routines to set the PCD related parameters

NOTE

In case the PCD is "set" to a FMan OP port, it should be disabled first (i.e. before calling 'FM_PORT_SetPCD' routine).

11. FMan runtime routines
12. FMan Free sequence - in reverse order from initialization

5.2.4.3.1.6 Global FMan Driver

The Global FMan driver refers to the common FMan features - i.e. functionality that is not defined per-port and does not belong to a span of the specific modules such as PCD, RTC, MURAM, MAC etc.

5.2.4.3.1.6.1 FMan Hardware Overview

The following Frame Manager processing elements are considered general FMan components and are controlled by the FMan common driver:

- The Frame Processor Manager (FPM) schedules frames for processing by the different elements to create the appropriate pipeline.

- The BMI is intended to transfer data between network and internal FMan memory, generate frame descriptor (FD), initialize the internal context (IC), manage the internal buffers, allocate/deallocate external buffers with the help of BMan and activate the DMA to transfer data between internal and external RAMs
- The DMA is responsible for frames data transfer from and to external memory
- The queue manager interface (QMI) is responsible for transferring packet-based work assignments between the queue manager (QMan) and the frame manager (FMan). It provides an interface to the QMan for enqueueing and dequeuing new frames to/from the multicore system.

5.2.4.3.1.6.1.1 Global FMan Driver Software Abstraction

The FMan global driver covers all the logically common FMan functionality, i.e functionality which is not port related. The different hardware modules within the FMan (i.e. BMI, DMA, etc.) are encapsulated within the FMan module. The terms "BMI", "DMA" are used for resources identification such as exceptions, counters and some configuration parameters, but logically, the only module used for functional operations is the FMan.

5.2.4.3.1.6.2 *How to use the Global FMan Driver?*

The following sections provide practical information for using the software drivers.

5.2.4.3.1.6.2.1 Global FMan Driver Scope

This module represents the common parts of the FMan. It includes:

- FMan hardware structures configuration and enablement
- Resource allocation and management
- Interrupt handling
- Statistics support
- ECC support for the FMan RAM's
- Load balancing between ports

5.2.4.3.1.6.2.2 Global FMan Driver Sequence

- FMan config routine
- [Optional] FMan advance configuration routines
- FMan Init routine
- FMan runtime routines
- FMan free routine

5.2.4.3.1.6.2.3 Global FMan Driver Functional Description

The following sections describe main driver functionalities and their usage.

5.2.4.3.1.6.2.3.1 FMan Configuration and Initialization

On FMan driver initialization, the software configures all FMan registers and relevant memory. It supplies default values where no other values are specified, it allocates MURAM, it loads FMan controller code. It defines IRQ's and sets IRQ handles. It enables hardware mechanisms and initializes software data structures for software management.

By the time initialization is done, FMan is ready to be used and any of the FMan sub-modules (FMan-Ports, MACs, etc.) may be initialized.

5.2.4.3.1.6.2.3.2 Resource Management & Tuning

The FMan provides resources used by its sub-modules. Generally, the driver selects default resource allocation, but when initializing the global FMan module, the user may specify a different allocation for some or all of the resources.

The resources relevant for this discussion are resources used by the BMI only. These resources should be further distributed between the different ports, but the initial allocation is for the BMI in opposed to some internal use of the FMan controller. The main and most important resources of the FMan are TNUMs (i.e. the FMan "tasks"), DMAs, FIFOs and "pipeline-depth".

The total available resources may vary based on SoC. The recommended default values are designed to fit most applications but as the resource allocation depends on system configuration, it therefore may vary between applications. I.e. the default value that are being set by the driver will be sufficient in use-cases were the user utilizing most of the FMan bandwidth and the user application is mostly using the FMan. In other cases such as if user uses some advance PCD settings and/or overloads the SoC (e.g. PCI is being massively used), the resources may need some special treatment and tuning by user as the default may not be sufficient enough.

Most MURAM is used as a temporary location for data transaction. This part's size is referred to as "FIFO size". The rest of the MURAM may be used for other utilizations such as Custom Classifier and its size is effected by the use of these features, i.e. if Custom Classifier is not used, "FIFO size" may be enlarged. The user may call `FM_ConfigTotalFifoSize` in order to modify the default value of the MURAM. However, one should bear in mind that when FIFO size is enlarged - Custom Classifier space is decreased.

5.2.4.3.1.6.2.3.3 Load Balancing

The FMan provides a mechanism to optimize the internal arbitration of different ports over the shared resources of the hardware.

The driver supports this feature by providing an API for dividing the bandwidth between the different ports (`FM_SetPortsBandwidth`). The API is given in terms of percentage - i.e. for each port, the user should specify its percentage relative to the other ports. This API is optional and may be modified at runtime. If not used, or if all ports get the same bandwidth (whether its {50,50} or {25,25,25,25}), then no one port will have priority over other ports. If ports get different values, for example 3 ports used and get {25,50,25}, than the first and third ports will get the same access to shared resources but the second one will get twice as much. i.e. The numerical values given to each port are not important, but only the relation between the ports.

5.2.4.3.1.6.2.3.4 Statistics

The FMan API provides access to all the statistics gathered by the FMan hardware. The API routine `FM_GetCounter` may be called at any time after initialization to retrieve any of the FMan counters.

5.2.4.3.1.7 FMan Parse-Classify-Distribute Driver

The Parse-Classify-Distribute (PCD) driver module refers to the parts of the drivers handling the different PCD engines and services such as Parser, Keygen, Custom Classifier, Policer, Header Manipulation, Reassembly, Fragmentation and Frame Replication. It deals both with the common configuration and runtime features and the specific PCD resources such as Keygen Schemes, Custom Classifier graphs, etc.

5.2.4.3.1.7.1 FMan PCD Hardware Overview

- **Parser**-The parser performs protocol header parsing and validation for a wide range of frame formats with varying protocols and encapsulation. A hard-coded parser function is used for the known and stable protocols. The hardware parser capabilities can be expanded by software parser functions to support protocols not supported by the hardware parser including proprietary protocols and shim headers. The parser parses the frame according to a per-port configuration. It reads the frame header from the FMan Memory and writes the frame parse results to the Internal Context of the frame. The Lineup Confirmation Vector is a part of the parser result. It represents a list of all the protocols recognized by the hardware parser, and may be extended to contain information added by the software parser.
- **Keygen**-The Keygen is located on the FMan receive path, and enables high performance implementation of pre-classification. It holds a SoC dependent number of key generation schemes in internal memory. Each scheme can generate different frame queue ID (FQID), a Storage-Profile ID (SPID) and policer profile (PP). One main function of the Keygen module is to separate network data into different flows, each requiring different processing. Another function of the Keygen, is the Classification Plan. This is a mechanism provided in order to mask LCV bits according to per-port definition. The Classification Plan is implemented as a table of SoC dependent number of entries, logically divided or shared between the FMan Ports.

- Custom Classifier-The Frame Manager (FMan) Custom Classifier module performs a look-up using a specific key from the received frame or internal frame context according to Parser results. The FMan Custom Classifier logically occurs after the Keygen processing has completed and can be operational in both the MAC receive flow and the offline parsing flow. The look-up produces an action descriptor which contains the necessary information for the continuation of the frame processing in the next module or the next look-up table.
- Policier-The Policier supports implementation of differentiated services at line speed on the Frame Manager (FMan) receive or offline parsing paths. It holds a SoC dependent number of traffic profiles in internal memory, each profile implementing RFC-2698 or RFC-4115 or Pass-Through mode. Each mode can work in either color-blind or color aware mode, and pass or drop packets according to their resulting color.

5.2.4.3.1.7.1.1 FMan PCD Software Abstraction

The FMan PCD driver aims to provide a high-level, abstract, network oriented, logical interface. It is designed to allow a glue logic between the different PCD engines and the PCD "user" - the FMan port, and to define an interface to these features to be used by the application. In this process, new non-hardware modules may be created - such as "Network Environment", while existing hardware modules - such as "Classification Plan" - may be hidden from the user. The following sections makes an attempt to describe the driver design decisions in abstracting the engines' hardware and the gap between the hardware programming model and the drivers API.

5.2.4.3.1.7.1.1.1 FMan PCD Flow

The FMan opens the FPM scheduling capabilities to the application, which allows significant flexibility in defining the packet flow. At various points in the flow, the FMan user must configure the next engine to handle the packet and the next operation it will perform. The driver minimizes this flexibility by assuming a basic flow for each port. The driver can expand this flow to include all FMan PCD capabilities, but in a limited manner that will be described below.

The basic flow reflects the expected use of the FMan PCD. When a port is initialized, the default setup that received packets are passed to the port's default Rx frame queue, as configured by the user. When the PCD is linked to the port, the user chooses one of the provided PCD support options which selects which PCD engines (parser, Keygen, FMan-Controller, and Policier) are included in the frames. The selected PCD support option adds the selected engine or engines to the flow according to the following PCD organization.

- When parser is used, it is always the first PCD engine working on the received frames.
- If parser is not activated, Keygen, and FMan-Controller may not be activated.
- Keygen's first use follows the parser, but it may be used again following the Fman-Controller or the policier.
- If FMan-Controller is used, it will follow the Keygen. It may not be activated if Keygen is not used.
- Policier may be activated by itself or follow any of the engines.

In all cases, the frame returns to the buffer manager interface (BMI) for enqueueing. The application may not change the main flow at runtime.

The following figure shows the default ports flows (in terms of next invoked action (NIA) registers' initialization):

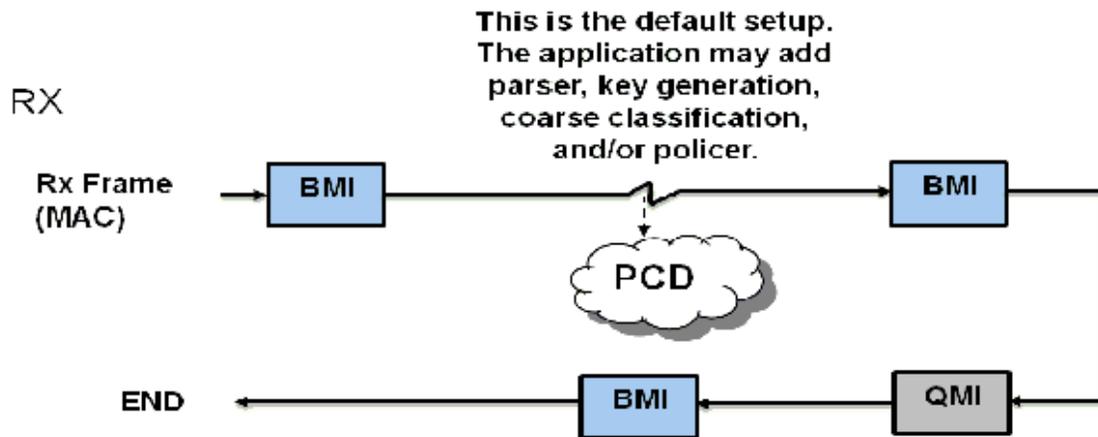


Figure 33. Default Rx Flow

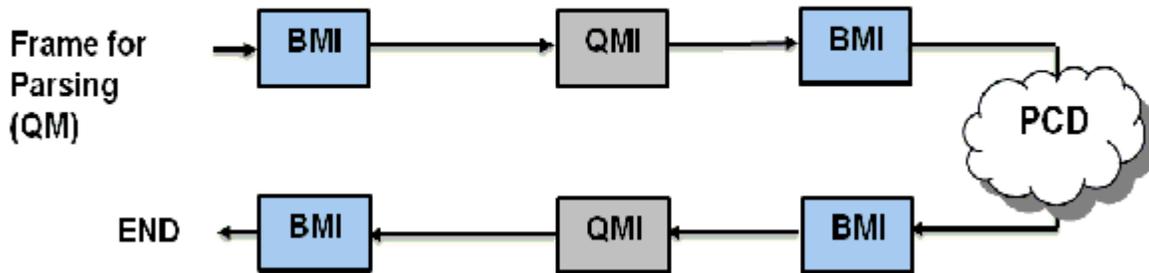


Figure 34. Default Offline Parsing Flow

NOTE

In independent mode, both Tx and Rx BMI NIA are FMan Controller. Other NIAs are not applicable.

After basic initialization, the default Rx flow, as shown in [Figure 33. Default Rx Flow](#) on page 208, is the configured flow. A PCD flow is initially defined by FMan Port level, although it is effected both by the port configuration and the PCD resources configuration. Following figure shows the PCD flows supported by the driver.

- Custom Classifier Roots
- Custom Classifier Match-Tables
- Custom Classifier Hahs-Tables
- Custom Classifier Manipulations
- Policer Profiles

The **Network Environment (NetEnv) Characteristics** are a pure SW resource. It is used in creating multiple HW PCD resources. Logically, it represents the NetEnv of a port or a number of port and supplies the glue between the parser, the Keygen, the Custom Classifier and the port. It ensures they all "speak the same language". Physically, it defines the LCV for all the participating protocols for each FMan Port.

Keygen Schemes and **Policer Profiles** are closely bound to their hardware programming model

Custom Classifier process is represented by a software graph. Each node in the graph represents a logical action. The driver defines different types of Custom Classifier nodes. One type of node is one of an Exact-Match which is a software representation of an Action-Descriptor (AD) that performs a lookup according to the key defined. Another type of node is one of Indexed-Lookup which is again a software representation of an Action-Descriptor of that type. A higher level of abstraction is performed on Hash-Table nodes, where the driver manages a hash table. Each node, may also contain a handle to a Manipulation action - which is the software abstraction for one or more AD's used for manipulating the frame by inserting and/or removing data. Generally, any Custom Classifier software node may be translated to one or more HW action descriptors.

The driver defines a notion of a Custom Classifier graph. The CC graph is the total set of lookups and manipulations performed by the Custom Classifier. The user builds the graph only after defining the CC Nodes. The finalization of the graph is done by building the root nodes and defining their grouping. This refers to the 16 entries array that functions as the entry point of the CC. Generally, the indexing into this array is performed by using 4 bits out of the LCV. This driver supports a division of this array into 2-16 unrelated groups to increase the flexibility of the programming and allow usage of more LCV bits.

5.2.4.3.1.7.1.1.4 How to Associate PCD Resources

The NetEnv is the link between the port and all the PCD resources it is using.

- Parser-The driver configures the LCV (lineup confirmation vector) in the parser configuration for every FMan Port according to the specific NetEnv it is bound to. When using SW parser, a private shim header should be added as a NetEnv unit, and may be used later as a regular unit.
- Keygen-Classification plan: The driver hides this resource from the user and configures classification plan entries to support and expand the HW parser capabilities according to the user definition of its NetEnv Characteristics
- Keygen-Schemes: The user describes the scheme in terms of NetEnv units, and the match vector is configured by the driver.
- Custom Classifier: The user describes the entry point of a CC root in terms of NetEnv units. The driver internally passes this information to the Keygen that uses it in selecting the entry point in the CC root when passing a frame from the Keygen to the Custom Classifier.

After defining PCD resources, the user may bind any FM Port to the initialized resources. A port must be bound to a single NetEnv, and may be bound to a Custom Classifier root and KeyGen schemes.

The set of figures below demonstrate a single example of the use of the driver's resources and their interaction with the hardware structures.

The following table demonstrates a NetEnv of 7 units. Unit 0, for example, is a simple unit recognizing ethernet frame, while unit 2 recognizes IP frames of either version.

Unit 0	Unit 1	Unit 2	Unit 3	Unit 4	Unit 5	Unit 6
Ethernet	Ethernet [Broadcast]	IPV4	IPV4	UDP	MPLS [stacked]	IPV4 [Multicast]
		IPV6		TCP		

Figure 36. Network Environment Example

When a port is bound to a NetEnv, the driver translates its units into the parser's hardware Line-up Confirmation Vector (LCV). The table below shows the LCV configured for a port that has the NetEnv above.

	LCV[0]	LCV[1]	LCV[2]	LCV[3]	LCV[4]	LCV[5]	LCV[6]	LCV[7-31]
Ethernet	1	1	0	0	0	0	0	0...0
IPv4	0	0	1	1	0	0	1	0...0
IPv6	0	0	1	0	0	0	0	0...0
UDP	0	0	0	0	1	0	0	0...0
TCP	0	0	0	0	1	0	0	0...0
MPLS	0	0	0	0	0	1	0	0...0

Figure 37. LCV Example (for a specific port that is bound to the Network environment above)

Based on the NetEnv, the driver also defines a set of Classification Plan entries to be used by each port using that NetEnv.

	Bit [0]	Bit [1]	Bit [2]	Bit [3]	Bit [4]	Bit [5]	Bit [6]	Bits [7-31]	Comments
0	1	0	1	1	1	0	0	1...1	No cls. Plan
1	1	1	1	1	1	0	0	1...1	Eth Broadcast
2	1	0	1	1	1	1	0	1...1	MPLS Stacked
3	1	1	1	1	1	1	0	1...1	1+2
4	1	0	1	1	1	0	1	1...1	IPv4 MC
5	1	1	1	1	1	0	1	1...1	1+4
6	1	0	1	1	1	1	1	1...1	2+4
7	1	1	1	1	1	1	1	1...1	1+2+4

Figure 38. Classification Plan entries for the Network environment above

When a frame is received its LCV is masked by one of the vectors in the Classification Plan. The FMan selects the entry based on the parser output and the port parameters.

To support this operation, the driver initializes the HXS plan offset field for each relevant header in the port parser parameters. The table below, is the driver's translation of the Network environment above into the port classification plan parameters. When a frame is being parsed, the classification plan offset for each header found is accumulated to construct the offset of the result classification plan. For example, a hypothetic frame of Ethernet BC/Stacked MPLS/IPv4 unicast frame, will have an LCV=0xF6000000 and a classification plan id of $2^{(1-1)} + 2^{(2-1)} = 3$, so its classification plan vector is 0xFDFFFFFF, and QLCV = 0xF4000000.

	HXS [plan offset]	Cls plan entry
Eth. BroadCast	1	$2^{(1-1)}=1$
MPLS Stacked	2	$2^{(2-1)}=2$
IPv6	0	0
UDP	-	-
TCP	-	-
IPv4 Multicast	3	$2^{(3-1)}=4$

Figure 39. Parser HXS (for a port that is bound to the Network environment above)

Given the driver's automatic initialization of the LCV and classification plan based on only the NetEnv, the user may now initialize Keygen schemes by passing as match criteria only the NetEnv unit id's. As in the other cases, the driver will translate the unit id's to the schemes' match vectors as can be seen in the figure below.

Id	Scheme Match Criteria	Units	Match vector
0	Ethernet broadcast	1	0x40000000
1	IPV4 MC+MPLS stacked	5+6	0x06000000
2	IPV4 MC	6	0x02000000
3	IPV4+(TCP or UDP)	3+4	0x18000000
4	match on IPv4 or IPv6 frames	2	0x20000000
5	Ethernet	0	0x80000000
6	Direct scheme	--	0xffffffff

Figure 40. Keygen schemes example

Finally, the driver will also take care of initializing the Keygen-to-Custom Classifier configuration registers. When initializing a Custom Classifier root, the user may create groups based on NetEnv units (in opposed to a simple group of a single entry; for more information refer to [Custom Classifier Root](#) on page 218).

When initializing a scheme, the user should only pass the handle to the Custom Classifier root. The driver will translate the group LCV dependent parameters into the scheme required register.

For example, Group 0 is a simple group that is not dependent on the NetEnv. Group 1 is based on a single unit - so a frame may be forwarded to 1 of 2 root nodes, and group 2 is based on 3 units - so a frame may be forwarded to 1 of 8 root nodes.

	CC Tree group Num of units	units	Keygen FMKG_SE_CCBS	Possible offsets within group depending on PR[LCV] AND FMKG_SE_CCBS
Group 0	0	--	0x00000000	0
Group 1	1	3	0x10000000 (Scheme 4 in the example)	0,1
Group 2	3	1,3,4	0x58000000	0-7

Figure 41. Keygen scheme configuration for CC next engine

The Policer Profiles are the one resource that does not rely on the Parser Results or the NetEnv. It is therefore managed independent of the other PCD resources.

5.2.4.3.1.7.1.15 FMan Header Manipulation

The FMan controller defines a set of header manipulation commands, and supports listing of these commands. The FMan driver allows limited listing by a single Manipulation node, limited to a single use of each command and to a defined order (e.g. remove + insert may be defined in a single node, but insert + remove or remove + remove may not). Alternatively, full listing and ordering is supported by chaining more than one Manipulation nodes. In such a case, the driver will unify HMCT's to optimize performance and MURAM usage unless parsing is required in between the different commands.

The following list maps each FMan controller command to the driver parameters in the Header Manipulation structure:

1. Generic removal-Set 'rmv' and use the corresponding parameters structure. Select generic enum and parameters.
2. Generic insertion-Set 'insrt' and use the corresponding parameters structure. Select generic enum and parameters.
3. Generic replace-Set 'insrt' and use the corresponding parameters structure. Select generic enum and parameters and set 'replace'.
4. Protocol specific removal-Set 'rmv' and use the corresponding parameters structure. Select byHdr enum and parameters.
5. Protocol specific insert-Set 'insrt' and use the corresponding parameters structure. Select byHdr enum and parameters.
6. Vlan priority update-Set 'fieldUpdate' and use the corresponding parameters structure. Select vlan enum and parameters.
7. IPv4 update-Set 'fieldUpdate' and use the corresponding parameters structure. Select IPv4 enum and parameters.
8. IPv6 update-Set 'fieldUpdate' and use the corresponding parameters structure. Select IPv6 enum and parameters.
9. TCP/UDP update-Set 'fieldUpdate' and use the corresponding parameters structure. Select TCP/UDP enum and parameters.
10. TCP/UDP checksum calculation-Set 'fieldUpdate' and use the corresponding parameters structure. Select TCP/UDP enum and parameters.
11. IP replace-Set 'custom' and use the corresponding parameters structure. Select TCP/UDP enum and parameters.

5.2.4.3.1.7.1.16 Custom Classifier Hash-Table Node

The driver provides a high level Hash-Table mechanism implemented over the FMan controller Custom Classifier structures. The driver implements the Hash-Table by using a Match-Table node of type Indexed-Hash, where each entry points to a hash bucket implemented by a Match-Table node of type Exact-Match (For more information on these nodes, refer to [Custom Classifier Root](#) on page 218). The driver uses the Keygen key and hash result as a key for the lookup. A selected part of the hash result is used to select the entry in the Indexed-Hash table (i.e. the bucket), and the full key possible values are used as the Match-Table keys in the selected bucket.

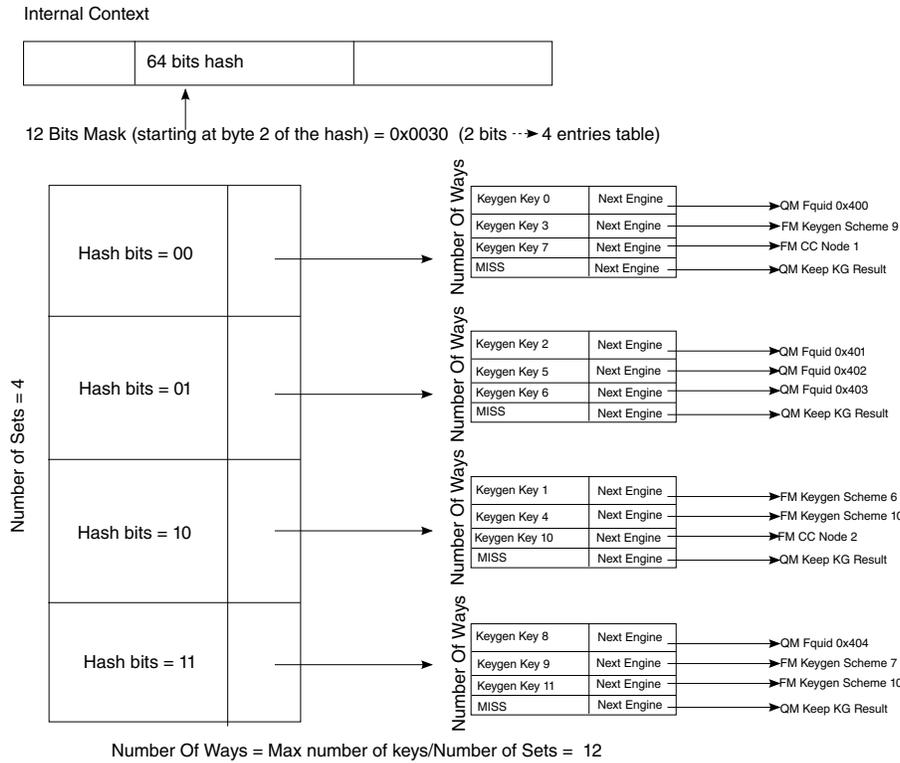


Figure 42. Hash_Table node example

5.2.4.3.1.7.2 How to use the FMan PCD Driver?

The following sections provide practical information for using the software drivers.

5.2.4.3.1.7.2.1 FMan PCD Driver Scope

- FMan Parser, Keygen, Custom Classifier & Policer configuration and initialization
- PCD Enable/Disable
- Resources allocation and management
- Interrupt handling
- Statistics support
- Support for FMan PCD operations

5.2.4.3.1.7.2.2 FMan PCD Driver Sequence

- FMan PCD Config routine
- [Optional] FMan PCD advance configuration routines
- FMan PCD Init routine
- Specific one-time pre-enable routines (e.g. load SW parser)
- FMan PCD Enable routine
- FMan PCD runtime routines
- FMan PCD specific resources runtime routines (for defining, modifying and deleting Keygen schemes, Custom Classifier nodes, etc.)
- FMan PCD Free routine

5.2.4.3.1.7.2.3 FMan PCD Driver Functional Description

The following sections describe main driver functionalities and their usage.

5.2.4.3.1.7.2.3.1 Global PCD Initialization

PCD initialization is divided into two parts. During the first part of the initialization, `FM_PCD_Config`, advance config routines, and `FM_PCD_Init` are called to configure and set all basic PCD capabilities, including pre-defining which engines are supported and may be used later. This stage is done in the kernel, and PCD is not yet enabled. During the second part of the initialization, PCD is enabled by a runtime routine (`FM_PCD_Enable`).

This division creates a gap during which some functionality may be added. The most important is the loading of the SW parser code. Note that this functionality is allowed only when PCD is disabled (i.e. between init and enable) or, with some restriction, in runtime after disable.

Once PCD basic initialization is complete (`FM_PCD_Init` and `FM_PCD_Enable` are called and returned), the PCD capabilities of the frame manager are reflected by the driver as a set of API runtime routines designed to define the PCD environment for a specific partition. PCD resources are defined per partition and may be used by all ports within a specific partition. The different PCD resources are first initialized and only later may be used by the FMan ports.

The order of PCD resources initialization is strict and relies on the PCD graph being initialized bottom up, which means that no resource may be initialized before its next engine is initialized. However, the use of port relative profiles is an exception to this rule. A scheme's next engine may be a port relative profile. In such a case, the scheme is initialized but not yet bound to a port, i.e. the actual policer profile is not yet specified. Therefore, its validity may not be verified. It is the user's responsibility to ensure that when a port using that scheme is activated (for using the PCD), its relative policer profile must be validated.

The PCD graph is partition based i.e. may be shared by ports on the same partition. Refer to [Port-PCD Binding](#) on page 236 for more details on port-PCD binding.

5.2.4.3.1.7.2.3.2 PCD Resources

The following subsections describe each of the driver's PCD resources in detail. In a single-partition environment, most resources are available and do not need explicit allocation. The port policer profiles are an exception. They must be allocated by the user, using the FMan Port API. In multipartition, some of the resources, specifically resources limited by hardware, must be first allocated by a partition and only then used by the partition's ports. The following sections specify the requirements for each of the PCD resources:

5.2.4.3.1.7.2.3.3 Network Environment Characteristics

The Network Environment (NetEnv) is a software entity that lists the network protocols used by the FM-PCD for classification and distribution. The total number of NetEnvs defined depends on the system configuration. A NetEnv may be defined per port or shared among some or all ports. The definition of a NetEnv must be done with care while considering the use of the FM-PCD module. The NetEnv is, in fact, the key for frames parsing, distribution, and classification.

The NetEnv is a list of distinction units. Each distinction unit consists of at least one or more headers. A header may either be one header from the list of supported headers or one of the supported headers plus an option (For more details on list and options available, refer to [Supported Network Protocols](#) on page 245).

The hardware parser implements header recognition. If the software parser is used, a distinction unit may also be one of the shim headers. The driver saves a number of units (that may be redefined in `fm_pcd_ext.h`) for private use. The user may then use this unit ID to recognize the private header by the Keygen or CC.

The following figure shows an example of a NetEnv. It has four units, two of which consist of a single header. One of the headers has an option. The final two units consist of two interchangeable headers. This example will be used throughout the following sections

.

Ethernet [Broadcast]	IPv4	IPv4	TCP	
	IPv6		UDP	

Figure 43. Network Environment Example

The distinction units list should reflect what the user wants to do with the PCD mechanisms to parse-classify-distribute incoming frames. Specifying a distinction unit means that the user wants to use that specification to either activate the parser on the specified headers or distinguish between frames with the Keygen or the Custom Classifier. Using interchangeable headers to define a unit means that the user is indifferent to which of the interchangeable headers is present in the frame, but instead wants the distinction to be based on the presence of either one of them. For example, if it is required that a selection of scheme is based on having L3 header of either IPv4 OR IPv6, but it is of no importance which of the two is present, than a unit should be defined with 2 interchangeable headers: IPv4, IPv6.

The initialization routine returns a NetEnv handle to be used later to specify that Network Environment.

Depending on context, there are limitations to the use of NetEnvs. A port using the PCD functionality is bound to a NetEnv. Some, or even all, ports may share a NetEnv, but it is also possible to have one NetEnv per port. When initializing a scheme, a Custom Classifier root, or when binding a port to the PCD, one of the required parameters is the handle of an initialized NetEnv. The driver uses the definitions of that NetEnv to initialize that scheme or Custom Classifier root. When a port is bound to a Keygen scheme or a Custom Classifier root, it must be bound to the same NetEnv.

For the flow's definition, the different PCD modules may only rely on distinction units as defined by their environment. When initializing a scheme for example, a PCD module may not choose to select IPv4 as a match for recognizing flows unless IPv4 was defined in the relating environment. In fact, to guide the user through the configuration of the PCD, each module's characterization in terms of flows is not done using protocol names, but rather environment indices.

In terms of hardware implementation, the list of distinction units sets the Lineup Confirmation Vectors (LCVs) and are later used for match vector and CC indexing. The shim header LCVs are conventionally assigned from LSB up, so the first shim header is 0x0000_0001. For more details on the implementation, refer to [Global FMan-PCD Resources](#) on page 209.

Runtime Modifications: A Network Environment may not be changed at runtime. New NetEnvs may be set, and unused NetEnvs may be deleted anytime.

Available API:

- FM_PCD_NetEnvCharacteristicsSet
- FM_PCD_NetEnvCharacteristicsDelete

5.2.4.3.1.7.2.3.4 Software Parser

The PCD allows the extension of the hardware parser by loading the software parser code for further manipulation. When this is required, the user passes the image of the software parser code and a table of labels to the driver. This represents the entry-points in the software parser code. If more than one code piece is required for a specific protocol (for example, to be used by different ports) an index is added to the labels table. Later, when configuring a port that uses one or more software parsing attachments, each protocol header may be bound to one of the previously declared labels. This is done by setting the software parser enable indication for one or more protocols headers, and indicating the software parser index (relative to that protocol header). The software parser code will run for that port after the hardware parser recognizes that header. In other words, the specified protocol header is in fact the trigger for the software parser to be activated. It is typical for the software parser to parse a private header that was previously defined as a NetEnv unit and then mark its existence for classification and distribution.

The software parser loading routine must be called only when the PCD is disabled and no ports in the system are using the parser. On initialization this means that the routine, if needed, must be called after FM_PCD_Init and before FM_PCD_Enable.

Runtime Modifications: Software parser may not be changed at runtime.

Available API:

- `FM_PCD_PrsLoadSw`

5.2.4.3.1.7.2.3.5 Keygen Schemes

The scheme entity relies on the hardware entity. There are 32 Keygen schemes in a frame manager. When a PCD is defined in a single partition environment, it is the owner of all 32 schemes. When a PCD is defined in a multipartition environment, the user must specify how many schemes are required for this partition. Once schemes are allocated for a specific partition, it may be used only by ports on that partition.

Within a partition, the schemes order is relevant. When initializing a scheme, the user must specify the following:

- Relative index, relative to the partition's schemes.
- Network environment handle.
- Match criteria, or which frames should be processed by the scheme.
- Keygen action (such as hash, FQID mask, and manipulation).
- Distribution FQIDs.

The match criteria (if used), is based on the NetEnv characteristics units. Schemes that are to be used directly should be configured as such, by specifying a scheme ID rather than using match criteria or specifying distinction units. Upon initialization, the driver returns a handle to the initialized scheme. This handle can be used later to specify the scheme.

Keygen schemes are dependant on parser results. They may be used immediately after the parser by direct mode or by using the match criteria. Schemes may also be used after the Custom Classifier or the policer. This flow is typically used for flow control redistribution. In this case, to avoid infinite loops the scheme is reached only in direct manner and not by match criteria.

The keygen action consists of the construction of the key and the definition of the distribution. The key is constructed by a set of extract actions arranged in the driver as an array of extractions. Extractions may be done from data, from Parse Result, from default values, but most commonly - from the header. When extraction is taken from the header it may be described generically by size and offset, or it may be an extraction of the full field. For a full list of supported headers and fields, see [Supported Network Protocols](#) on page 245.

When a scheme is initialized, the user must specify the next engine to which the frame should pass after it is processed. The next specified engine must be initialized and valid at this point. Frames may pass to the Custom Classifier or the policer, or they may be directly enqueued to an FQID.

Once schemes are defined, ports may be bound to them. A port may be bound to as many schemes as needed, as long as they are from the same partition and the same NetEnv.

Following figure shows an example of scheme setting and connection to the NetEnv, as shown in [Network Environment Characteristics](#) on page 215.

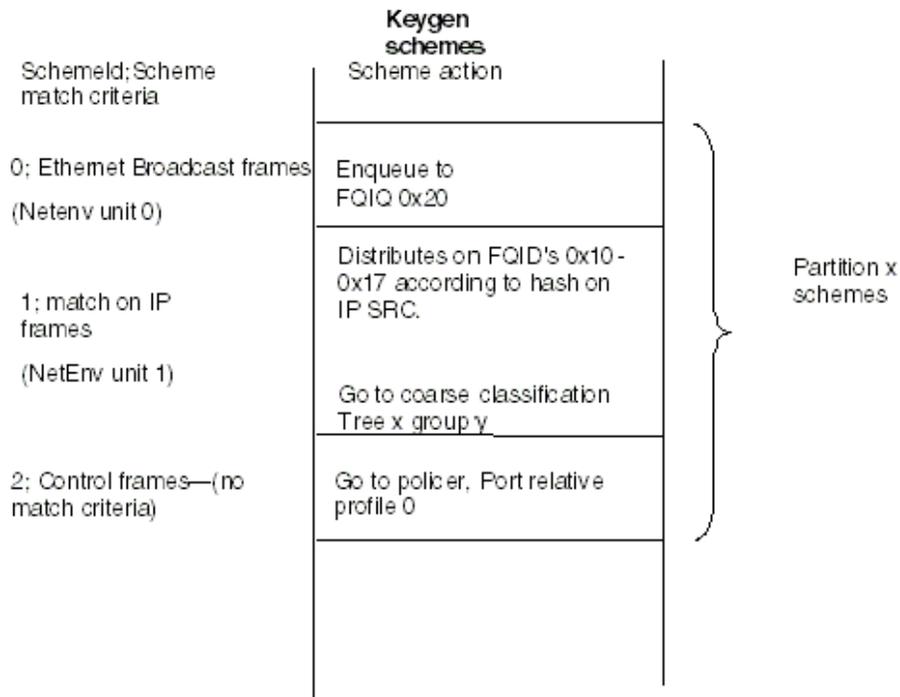


Figure 44. Schemes Example

Runtime Modifications: Valid schemes may be modified at runtime by calling the scheme initialization routine for an existing scheme with the following differences:

1. Passing the scheme handle as returned by the original initialization routine (instead of the scheme's relative ID).
2. Setting 'modify' to be 'TRUE'.

New schemes may be set and unused schemes may be deleted anytime.

Available API:

- FM_PCD_KgSchemeSet
- FM_PCD_KgSchemeDelete

5.2.4.3.1.7.2.3.6 Custom Classifier Root

A Custom Classifier root (or actually the entire CC graph) may be defined per FMan Port or shared by ports on the same partition. It is a set of lookups defined to classify, route and perform manipulation on a flow of frames. The CC graph is built bottom up by connecting CC Nodes. When a node (which is not a leaf in the graph) is set, it points to other nodes. These other nodes must already be initialized.

A CC root is defined by a set of entries that construct the root of the graph, and Custom Classifier Nodes of different types.

Once all nodes in the graph are ready and connected, the root is built by calling the FM_PCD_CcRootBuild routine. The root of the graph is in fact an array of up to 16 root entry nodes. The entry point for a frame is one of the CC root entries, depending on the engine that precedes the CC which is the Keygen.

According to the parser results (which is defined by the NetEnv setting) and Keygen configuration, a frame is directed to one of the entries in the CC root array.

When building the CC root, the user must specify its NetEnv id. Up to four distinction units may define the selection of one node (out of the 16), in a simple bit selection method. The following table shows the CC Root nodes selection (0 = unrecognized by parser, 1 = recognized by parser).

Table 20. CC Root Nodes Selection

Unit0	Unit1	Unit2	Unit3	Selected Node
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14

To allow more than 4 units to be involved in the selection, the 16 entries may be divided into groups. The table above demonstrates an organization of one group of 16 nodes, but other organizations are possible:

2 groups of 8 -> each group selected by 3 units (to select nodes 0-7 relative to this group's base)

4 groups of 4 -> each group selected by 2 units (to select nodes 0-3 relative to this group's base)

8 groups of 2 -> each group selected by 1 units (to select nodes 0-1 relative to this group's base)

16 groups of 1 -> indifferent to units (single node group always selected)

2-8 groups of varied sizes (8-1)

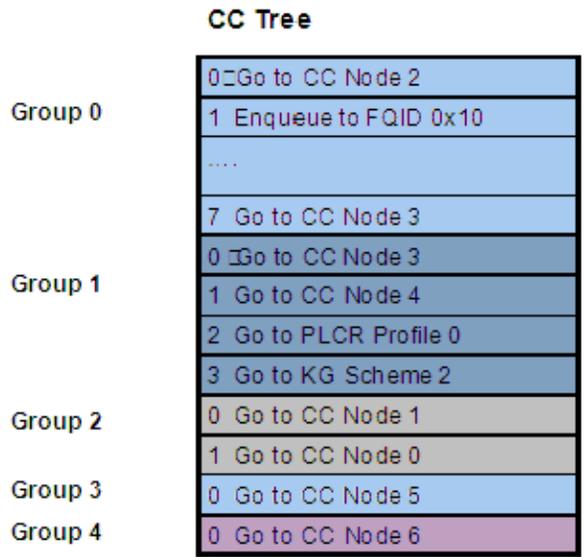


Figure 45. CC Root - 5 groups example

When building the CC Root, the user must specify the number and size of groups. Then, for each group, an array of per-root-node parameters is passed. The array is ordered according to the table above.

A simplified way of using the CC, is to define up to 16 different groups of one root-node each. In this way, all traffic from a specific Keygen scheme is going to the same group, which is a single node, and no NetEnv unit are selected. Groups 3 and 4 in figure above are an example of a single root group.

The following figure shows a combined use of the NetEnv units in Keygen and Custom Classifier, based on the previous NetEnv and Keygen scheme examples.

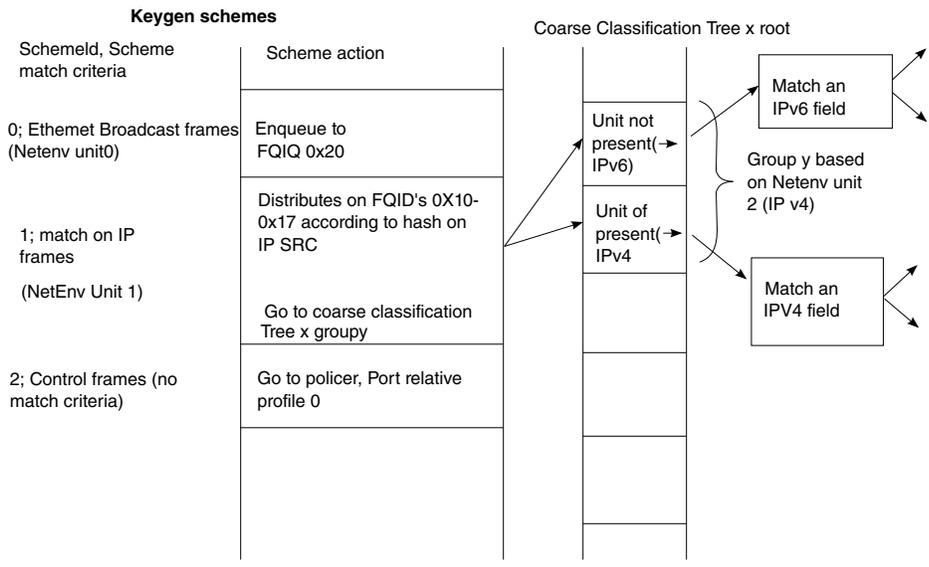


Figure 46. Keygen -> Custom Classifier Example

When a CC root or node is initialized, the driver returns a handle to the root or node respectively. This handle may be used later for specifying the root or node. For example, to build a root, the nodes are specified by passing their handles, and a root

handle must be passed when defining a port that uses the Custom Classifier. A port may be bound only to one root, from the same partition and NetEnv as the port.

Runtime Modifications: Custom Classifier nodes may be modified by using one of the routines listed in the "Available API" below.

Custom Classifier Roots may not be changed at runtime. New nodes and roots may be defined and unused ones may be deleted anytime.

Available API:

- FM_PCD_MatchTableSet
- FM_PCD_MatchTableDelete
- FM_PCD_HashTableSet
- FM_PCD_HashTableDelete
- FM_PCD_CcRootBuild
- FM_PCD_CcRootDelete

Specific runtime API:

- FM_PCD_CcRootModifyNextEngine
- FM_PCD_MatchTableModifyNextEngine
- FM_PCD_MatchTableModifyMissNextEngine
- FM_PCD_MatchTableRemoveKey
- FM_PCD_MatchTableAddKey
- FM_PCD_MatchTableModifyKey
- FM_PCD_MatchTableModifyKeyAndNextEngine
- FM_PCD_MatchTableFindNModifyNextEngine
- FM_PCD_MatchTableFindNRemoveKey
- FM_PCD_MatchTableFindNModifyKeyAndNextEngine
- FM_PCD_MatchTableFindNModifyKey
- FM_PCD_HashTableAddKey
- FM_PCD_HashTableRemoveKey
- FM_PCD_HashTableModifyNextEngine
- FM_PCD_HashTableModifyMissNextEngine

5.2.4.3.1.7.2.3.7 Match-Table Nodes

The driver defines two types of Match-Table nodes - Exact-Match nodes and Indexed-Lookup nodes. On both types of nodes a table of entries is defined where each entry leads to a selected next-engine with a selected action. The next-engines may be another CC Node, a Keygen scheme, a Policer profile or an enqueue action to a QM queue. In the last case, the queue may be either an Fqid (frame queue id) that was previously defined - typically by the Keygen, or an explicitly specified new Fqid that overrides any previous Fqid selection.

The difference between the two types of nodes is in the way an entry is selected in the node's table.

On an exact-match node, the user defines an extraction of data taken from the frame or the Internal-Context. The table of entries represent different possible values (keys) for this extraction, so that for each key a next-action is selected. An extra 'MISS' entry is also specified.

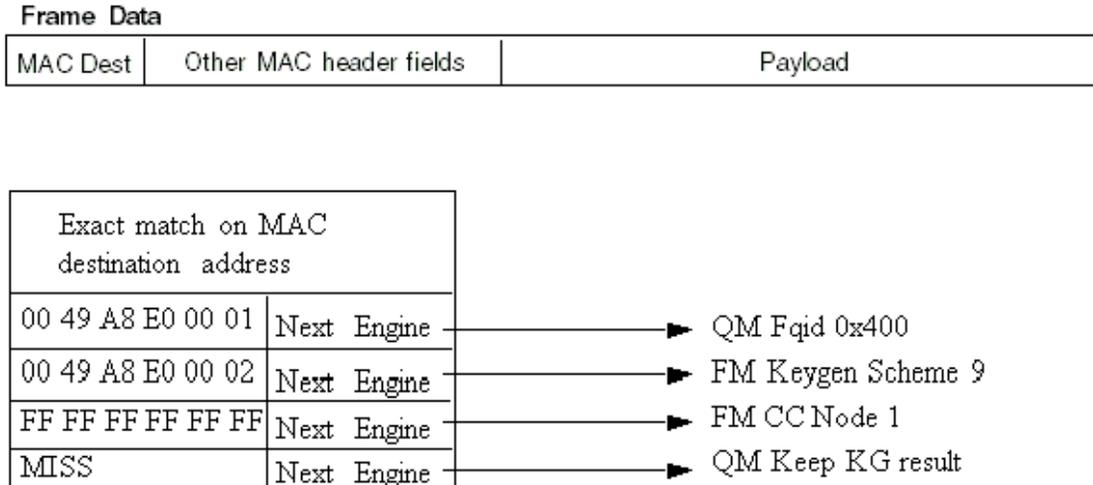


Figure 47. Exact-Match Node Example

On an Indexed-lookup node, up to 2¹² may be defined. The user selects 12 bits out of the Internal Context as an index to an entry in the table. The 12 bits may be masked to select less bits and a smaller table.

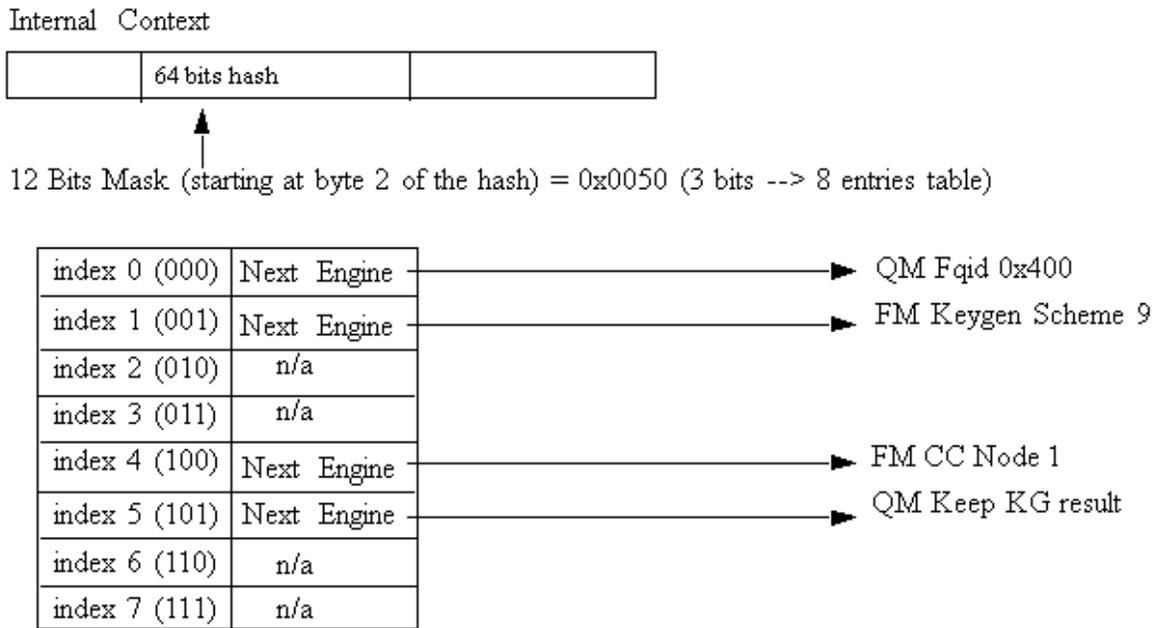


Figure 48. Indexed-Lookup node example

Two methods for CC node allocation are available: dynamic and static. Static mode was created in order to prevent runtime alloc/free of FMan memory (MURAM), which may cause fragmentation; in this mode, the driver automatically allocates the memory according to maximal number of keys, as received from the user. The driver calculates the maximal memory size that may be used for this CC node, taking into consideration whether key masks are required and node's statistics mode.

In dynamic mode, maximal number of keys is not provided (equals zero). At initialization, all required structures are allocated according to current number of keys. During runtime modification, these structures are re-allocated according to the updated number of keys.

5.2.4.3.1.7.2.3.8 Hash-Table Nodes

The Hash-Table node is a driver managed Hash table. It is defined as a next engine and may follow other CC nodes. The Hash-Table module uses driver lower level CC structures and provides an abstraction layer API consisting of AddKey/RemoveKey routines. By using this module, the user may easily use a hash table based on Keygen key extraction and hash calculation. When initializing this node, the user should define parameters regarding the basic key used for hashing and the structure and size of the hash table (sets/ways).

5.2.4.3.1.7.2.3.9 Manipulations

On the structural aspect, Manipulation nodes are not graph nodes in the way that they do not effect the flow of a frame, and they are not in fact a graph junctions. Manipulations nodes are defined as extensions to existing CC nodes of all types. Any key on any CC node may have a manipulation characterization on top of the next engine definition. This is realized by CC node parameter `h_Manip` which is a handle to a previously initialized Manipulation node (according to the bottom-up principle). The Manipulation node itself does not have a next engine definition and the frame's flow is determined by the last CC node.

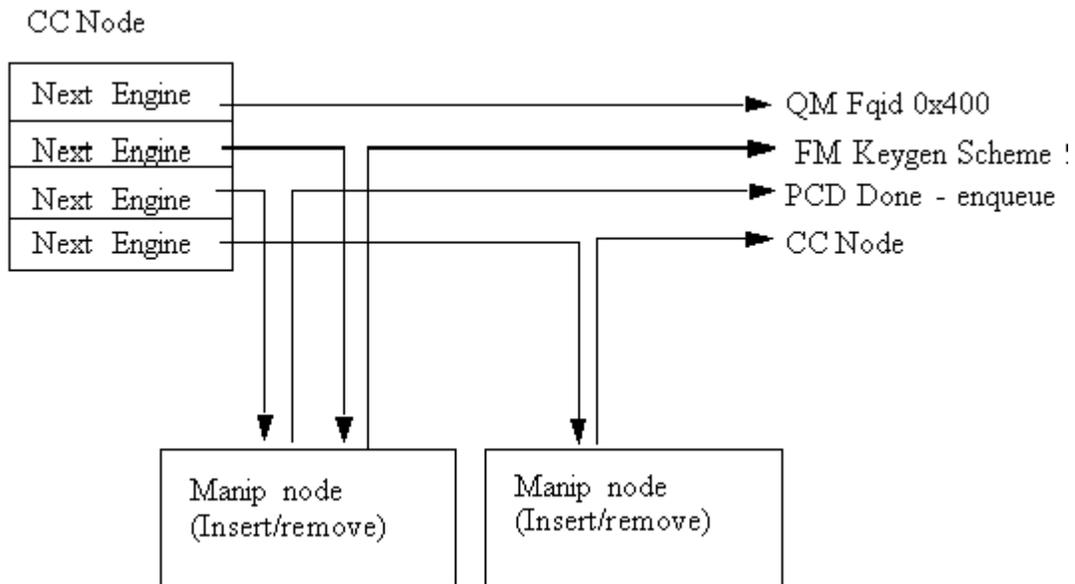


Figure 49. CC Node With Manipulation

Available API:

- `FM_PCD_ManipNodeSet`
- `FM_PCD_ManipNodeDelete`

Specific runtime API:

- `FM_PCD_ManipNodeReplace` (only available for Header-manipulation)
- `FM_PCD_ManipGetStatistics`

NOTE

- For all manipulation types below, the user must call `'FM_PCD_SetAdvancedOffloadSupport'` before calling `'FM_PCD_Enable'`.
- For each RX/OP-Ports that will work with the above FM-PCD, the user should have at least 16 trnms (num of tasks). in order to set the trnms the user should call `'FM_PORT_ConfigNumOfTasks'`.
- It is also required to set the DMA transactions to be per port by calling `'FM_ConfigDmaAidOverride'` with `'FALSE'` and calling `'FM_ConfigDmaAidMode'` with `'e_FM_DMA_AID_OUT_PORT_ID'`

5.2.4.3.1.7.2.3.9.1 Header Manipulation

The header manipulation is implemented by the FMan controller block, and is designed to change the incoming frame header for termination or interworking flow requirements. Header modification can be configured on a per-flow basis or for a user-determined group of flows.

The firmware defines some header manipulation structures which hold parameters for the definition of header manipulation action. It defines a basic table descriptor (Header Manipulation Table Descriptor HMTD) and a table of commands (HMCT), allowing a sequence of manipulations to be performed. The commands table may reside in either internal or external memory. The manipulation may be performed at any stage of the Custom Classifier process. As the manipulation changes the frame, the process allows an additional parsing of the processed frame once the manipulation process had ended.

The Header Manipulation (HM) mechanism is viewed by the driver as an extension to other Custom Classifier Nodes. It may take place at the beginning, the middle or the end of a CC graph, but it may not have an effect on the flow, i.e. the selection of the next action.

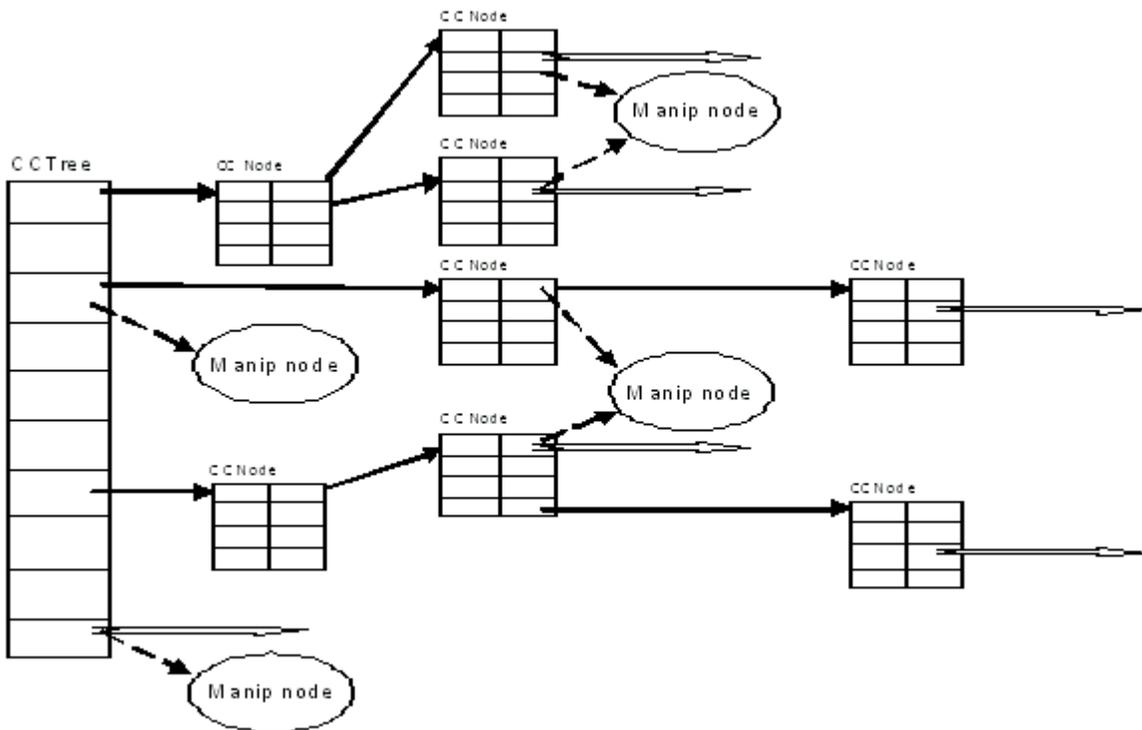


Figure 50. Header Manipulation CC Perspective

The HM action is represented by the driver's Manip node which is a driver sub-module (i.e. initialized by the user, its initialization routine returns an HM handle).

A Header Manipulation node is an independent unit that has no external information regarding other modules in the PCD graph, its users, its location in the flow, or the next engine it will be followed by.

A CC key or a CC root node may lead to a Header Manipulation node. The CC key/root node will define the next engine that should follow the manipulation. The next engine may be Keygen, Policer, another CC node, or PCD termination (enqueue).

In order to use the HM, the user should first create a Manip node, and then use its handle when defining the CC Node that points to this manipulation action.

A Header Manipulation action may be defined as one of the following manipulations:

- Remove
- Insert

- Fields Update
- Custom

More than one manipulation is allowed only if they are to be performed in the order above and only one manipulation of each type.

Other orders or a list of manipulations of the same type may be achieved by chaining some manipulation nodes by using the `h_NextManip` handle of the Manipulation parameters structure.

HM nodes may be shared, so that the same HM handle can be passed to more than one CC key.

By default, each frame goes back to the parser to be re-parsed after the manipulation. However this behavior may be disabled and may have an effect on performance as will be explained in the restrictions note below. It is controlled by the Header Manipulation node parameters.

The parsing option applies to whatever the user initialize as a Manip node - i.e. if the node contains a number of commands, the parsing can be done after all the commands and not between them. However, if the set of commands is initialized as a number of nodes that are chained together, the parser may be run after each node.

The driver aims to optimize performance and MURAM utilization. It does so by internally creating a single command table for chained nodes. Note that this optimization is NOT possible if parsing is required between manipulations and in this case the manip nodes are cascaded.

Note that when manipulations are chained, some restrictions apply:

1. Sharing of chained nodes is only possible on the head of the manipulation and not on inner nodes, i.e. all the manipulation is shared and not parts of it.
2. When parsing is required between manip nodes, the optimization described above is NOT possible and in this case the manip nodes are cascaded.
3. When parsing is required between manip nodes, the next engine of the last CC node may NOT be another CC node; i.e. chained nodes with parsing between them may only exist at the end (and not in the middle) of the CC graph.

5.2.4.3.1.7.2.3.9.2 IP Reassembly

The FM supports IP reassembly for both IPv4 and IPv6. The FMan accumulates IP fragments until enough have arrived to completely reconstitute the original datagram. IP Reassembly supports a maximum of 16 fragments per frame. Each fragment must reside in a single buffer (not in a Scatter/Gather frame).

The IP Reassembly driver utilizes the FMan Controller and FMan PCD resources in order to provide a full IP Reassembly solution.

The driver's interface is not identical to the hardware resources and provide an abstraction layer to the hardware resources. All IP Reassembly hardware data structures used for IP reassembly manipulation are represented by the software Custom Classifier Manipulation node. On top of the CC Manipulation, the driver internally defines the other resources needed for the full flow.

IP Reassembly flow

Fragments arriving on an Rx (or offline parsing) FMan Port that was configured to support IP Reassembly are recognized and marked by the software parser extension. These frames are steered to direct schemes the Keygen and caught by dedicated schemes that pass them to the Custom Classifier. The CC Root object is configured so that the IP fragments will reach a dedicated root entry node that contains a CC manipulation node. At this point, the IP Reassembly is performed. When a full frame is gathered, it is passed by the FMan controller back to the parser as a full reassembled frame. It is then passed to the Keygen and may be distributed and classified as any other frame.

What should the user do?

The following sequence describes the steps the user must take in order for the flow above to work.

- Initialize general DPAA (BM, BM Portal, BM Pools, QM, QM Portal, FMan and FMan PCD)
- Initialize the Rx/Offline FMan Port on which reassembly should run

- Define PCD as follows:
 - Set a Network Environment with one of the following options:
 - `HEADER_TYPE_IPV4` unit with `IPV4_FRAG_1` option for IPv4 reassembly manipulation.
 - `HEADER_TYPE_IPV6` unit with `IPV6_FRAG_1` option for IPv6 reassembly manipulation.

Note that if the user needs IPv4 or IPv6 units for other use, the fragmentation units may not be shared and dedicated units must be defined.

 - Allocate the first one or two schemes - one if only IPv4 is used, 2 if IPv6 is also used. The user should not configure those schemes, just save these schemes from other usage. The driver will use the first scheme for IPv4, and if needed, it will use the second for IPv6.
 - Create reassembly manipulation using `FM_PCD_ManipNodeSet` routine. Pass the relative id's of the schemes allocated above (A single manipulation module should be created for both IPv4 and IPv6 fragmented frames, passing all relevant parameters).
 - If CC is used, it is user's responsibility to leave two unused entries when building the CC root nodes (i.e. the total number of entries between all groups should not exceed 14).
 - Set at least one scheme to catch regular/reassembled frames.
 - When binding the Rx/Offline FMan Port to the PCD properties (i.e. calling `FM_PCD_SetPCD`), pass a handle to the created Reassembly Manipulation node.

Note that in order to perform distribution or classification on IPv4/IPv6 frames (unrelated to reassembly of IPv4/IPv6 fragments), independent IPv4/IPv6 units with no option must be explicitly defined.

What does the driver do?

In order to provide the required support for IP Reassembly, the driver performs some internal actions triggered by the user configuration. The following information describes the actions the driver internally performs and has no functional relevance to the user:

- When reassembly is required, the driver internally enables parser recognition of IPv4/IPv6 and shim2 - which is the IP Reassembly extension. This is triggered by the user defining NetEnv units with options: `IPV4_FRAG_1/IPv6_FRAG_1`.
- The driver loads the software parser that identifies IP fragments and enables its operation for the required FMan Port.
- The driver defines one or two (one for each IP version) Keygen schemes that recognize IP fragments and are programmed to generate an IP Reassembly key. When a frame is recognized as an IP fragment (by the Parser), it is steered to these Keygen schemes. The user should allocate the first one or two (for IPv4 and/or IPv6) schemes and pass their relative id's to the driver. The driver will internally initialize the relevant reassembly schemes when required.
- Each of the schemes above is programmed by the driver to point to a group in the Custom Classifier Root. If the user did not create a CC Root, the driver internally creates a new one. In both cases, the driver creates the needed group/s in the CC Root. It always uses the last two groups. It is user's responsibility to have at least two empty entries (one for a single IP version, two for both).
- The driver attaches the Manipulation sequence (created by the user) to the appropriate root entry node in the CC Root, causing the reassembly of IP fragments.

NOTE

The software parser code required to support reassembly may not coexist with user software parser code. If the user supplies IPv4 or IPv6 software parser code, it must include the code for handling IPv4/IPv6 reassembly according to the FMan controller spec.

Suggestions of how to use IPR in a system

The PCD with the IPR should identify frames up to L3; i.e. if the frame is IP or not.

In case it isn't an IP frame it should pass the desire PCD. IP frames should pass the reassembly process and than be directed to OP-Port to be classified according to their L3 and above.

5.2.4.3.1.7.2.3.9.3 IP Fragmentation

The FMan supports IP fragmentation for both IPv4 and IPv6. The fragmentation mechanism is implemented by the PCD, specifically by the Custom Classifier. IP fragmentation may be performed using an Offline Parsing FMan Port with a specific PCD configuration that will be described in this section.

The software driver provides API for initializing the IP fragmentation mechanism. driver's interface is not identical to the hardware resources and provide an abstraction layer to the hardware resources. Both of the AD (action descriptor) tables that used for IP fragmentation manipulation represented by the software Custom Classifier nodes using CC Manipulation. IP Fragmentation manipulation is used for fragmentation of IPv4 and IPv6 frames according to a specific MTU. This manipulation can be used on Offline Parsing ports only and as a part of the port's PCD definition. CC Nodes should have an IP fragmentation manipulation characterization in order to trigger this manipulation. This means that in order to create and initialize the IP fragmentation hardware, the user should create a Custom Classifier Node with Manipulation (refer to [Custom Classifier Root](#) on page 218). All relevant parameters such as MTU are defined during this module creation.

Following is the sequence that should be followed:

- Initialize general DPAA (BM, BM Portal, BM Pools, QM, QM Portal, FMan and FMan PCD)
- Initialize FMan Port of type Offline Parsing
- Define fragmentation PCD as follows:
 - Initialize an empty Network Environment (without any units)
 - Create fragmentation manipulation using `FM_PCD_ManipNodeSet` routine.
 - Create CC Node by calling `FM_PCD_MatchTableSet/FM_PCD_HashTableSet` and attached the fragmentation manipulation previously created to the desired key.
 - Build a CC Root with 1 group that points to the previously defined CC Node .
- Bind the Offline Parsing FMan Port to the PCD properties by calling `FM_PORT_SetPCD`

Manipulation parameters

- MTU of the fragmentation manipulation.
- Scratch Buffer Pool ID is a buffer pool that is required by the fragmentation process in order to ensure correct release operation of the frames and fragments. All IP Fragmentation Table Descriptors should use the same Scratch Buffer Pool ID. This pool must not be used by any other process or engine in the system.
- Don't Fragment Action - by setting this parameter the user can determine the action to be taken in case the IP packet is larger than the defined MTU and the 'Don't Fragment' (DF) bit of the frame is set.

NOTE

The software parser code required to support fragmentation may not coexist with user software parser code. If the user supplies IPv6 software parser code, it must include the code for handling IPv6 fragmentation according to the FMan controller spec.

Restrictions:

1. Tx confirmation is not supported.
2. Only Bman buffers shall be used for frames to be fragmented.
3. IP-Fragmentation will not work on OP-Port with VSP enabled.
4. fragmentation of IP-fragments is not supported
5. IPv4 packets containing header option field are fragments by copying all option fields to each fragment, regardless of the copy bit value.
6. Maximum number of fragments per frame is 16.

Suggestions of how to use IPF in a system:

In case one of the #1-#2 3 restrictions above is critical than it is suggested not to use IPF on OP-Ports that receive frames from the GPP and to do it on the GPP itself. We also suggest to put the IPF on a OP-Port just before the TX-Port.

5.2.4.3.1.7.2.3.9.4 IPsec Manipulation

The IPsec Manipulation is a specific instantiation of the special offload manipulation. It is designed to handle IPsec traffic in order to support the following actions:

- Support of variable outer header size

The user should initialize a Manipulation node of this type passing the relevant parameters

- Support for both ipv4/ipv6 IP version within SA

The user should initialize a Manipulation node of this type passing the relevant parameters.

- ECN/DSCP copying from inner/outer IP header to outer/inner.

In order to use this functionality the user must follow the following steps:

- Define a Manipulation node of this type passing the relevant parameters
- For the relevant Rx/OP port, define a buffer prefix that includes at least the Keygen hash result.
- Use SEC parameters to support this operation

5.2.4.3.1.7.2.3.10 Frame Replicator

The Frame Replicator (FR) is designed to duplicate incoming packets and route them to separate destinations. It is defined as a next engine and may follow other CC nodes, i.e. Match-table key, Hash-Table key or a CC-Root entry.

A Frame Replicator is realized by a group of members, where each member defines a replication of the incoming frame and a route to continue.

The next engine after FR is restricted to one of the following:

- Enqueue (PCD Termination)
- Policer
- Keygen (Direct scheme that leads to either Policer or PCD Termination)

When initializing an FR node, the user must define the maximum number of members this node may contain. The actual number of members may be modified on runtime by adding and removing FR group members.

Runtime modifications of add/remove members to/from the group can be done at any point in the system and in any location of the members group (first, middle or last). Note that runtime-modifications require the use of Host Command.

The order of the members in the group is of significance as the implementation of the replication is serial.

Manipulation may be applied to:

1. The whole group. The manipulation node should be placed before the replication group. That means that the FR is the next-engine of the Manip node. The Manip node is the next-engine of a key in a Match-table or Hash-table.
2. The last member of a FR group. That means that the manip node is the next-engine of the last member of the FR group.

NOTE

No support of Manip node after the "non-last" members.

The driver supports sharing of FR nodes means that FR group may be shared by more than one source.

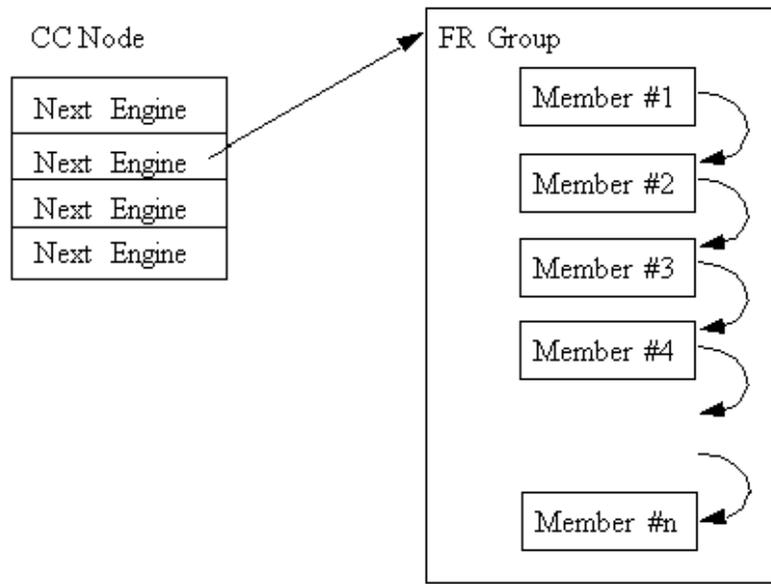


Figure 51. Frame Replicator Following a CC Node

Available API:

- FM_PCD_FrmReplicSetGroup
- FM_PCD_FrmReplicDeleteGroup

Specific runtime API:

- FM_PCD_FrmReplicAddMember
- FM_PCD_FrmReplicRemoveMember

NOTE

- For all manipulation types below, the user must call 'FM_PCD_SetAdvancedOffloadSupport' before calling 'FM_PCD_Enable'.
- For each RX/OP-Ports that will work with the above FM-PCD, the user should have at least 16 tnums (num of tasks). In order to set the tnums, the user should call 'FM_PORT_ConfigNumOfTasks'.
- It is also required to set the DMA transactions to be per port by calling 'FM_ConfigDmaAidOverride' with 'FALSE' and calling 'FM_ConfigDmaAidMode' with 'e_FM_DMA_AID_OUT_PORT_ID'

5.2.4.3.1.7.2.3.11 Policer Profiles

The policer profile entity relies on the hardware entity. It defines rules for policing for a certain flow. There are 256 different profiles in a frame manager that may be organized in per port windows. Some profiles may be shared between ports on the same PCD. By default, the number of shared profiles is set by the driver, but the user can also configure it to a different value. Shared profiles are typically used for aggregation.

When a PCD is defined in a single partition environment, it is the owner of all 256 profiles. When a PCD is defined in a multipartition environment, it is the owner of its shared profiles along with all the profiles that will be allocated per port for ports on this partition. The user must explicitly allocate per-port profiles for each port (if required), after PCD is initialized and prior to the profile initialization. Note that per-port profiles are the only PCD resource that is explicitly allocated and initialized for a specific port.

After profiles are mapped, the user may initialize each of the profiles by stating the following:

- Type
 - Shared
 - Per-port
- Offset relative to the port or to the shared group of profiles
- Characteristics

Once initialized, a handle is assigned to the profile for later use.

The Policer may be used after the Parser, Keygen or Custom Classifier, or solely - without activating any of the other PCD engines. It is not dependant on any previous output such as parser result. The policer may be used more than once in a frame flow. The next action after a police profile is either to pass the frame to a direct Keygen scheme for a new distribution (typically for control frames coming from the Custom Classifier), to pass the frame to another profile (always a shared profile, typically an aggregators), or to enqueue the frame to an FQID.

When other engines select a policer profile as the next engine, its handle must be passed. An exception is when a per-port profile is specified as the next engine of a scheme or of a "overrideParams" CC key. In these cases a port-relative index is required instead. The reason for this is that the required Policer Profile may not be initialized at this stage and hence have no handle. This irregular behavior is because CC Roots and KG schemes may be shared by ports, and at the time of scheme/ root initialization, they are not yet bound to specific ports. In this context, the profile selected may in fact be uninitialized and therefore can't be verified by the driver. It is therefor user's responsibility to make sure it is set prior to port- PCD binding.

Runtime Modifications: Valid profiles may be modified at runtime by calling the profile initialization routine for an existing profile, passing the profile handle as returned by the original initialization routine, and specifying modify (instead of the profile's relative id). New profiles may be set and unused profiles may be deleted anytime.

Available API:

- FM_PCD_PlcrProfileSet
- FM_PCD_DeleteProfilePlcr

5.2.4.3.1.7.2.3.12 PCD Organization

By initializing PCD resources, the user creates a directed graph in which the parser is the source of the graph and the FQIDs are its endpoints. Following figure shows a generalized example of a basic PCD graph.

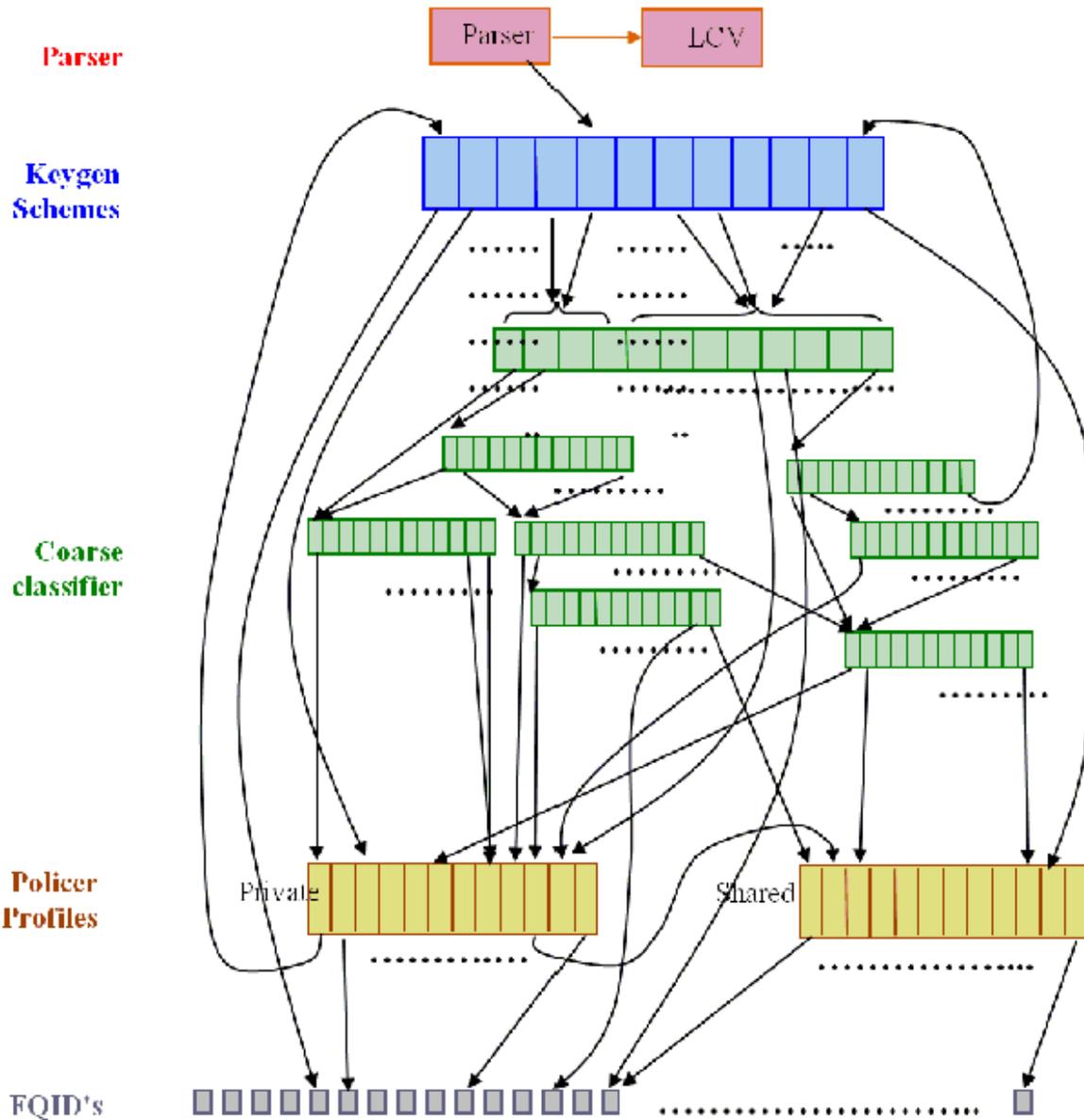


Figure 52. PCD Organization

5.2.4.3.1.7.2.3.13 PCD Definition Sequence

When a PCD graph is defined, its resources must be initialized bottom up when there's a dependency between them. Following figure shows the order of initialization (starting at the top of the figure) in a specific sequence.

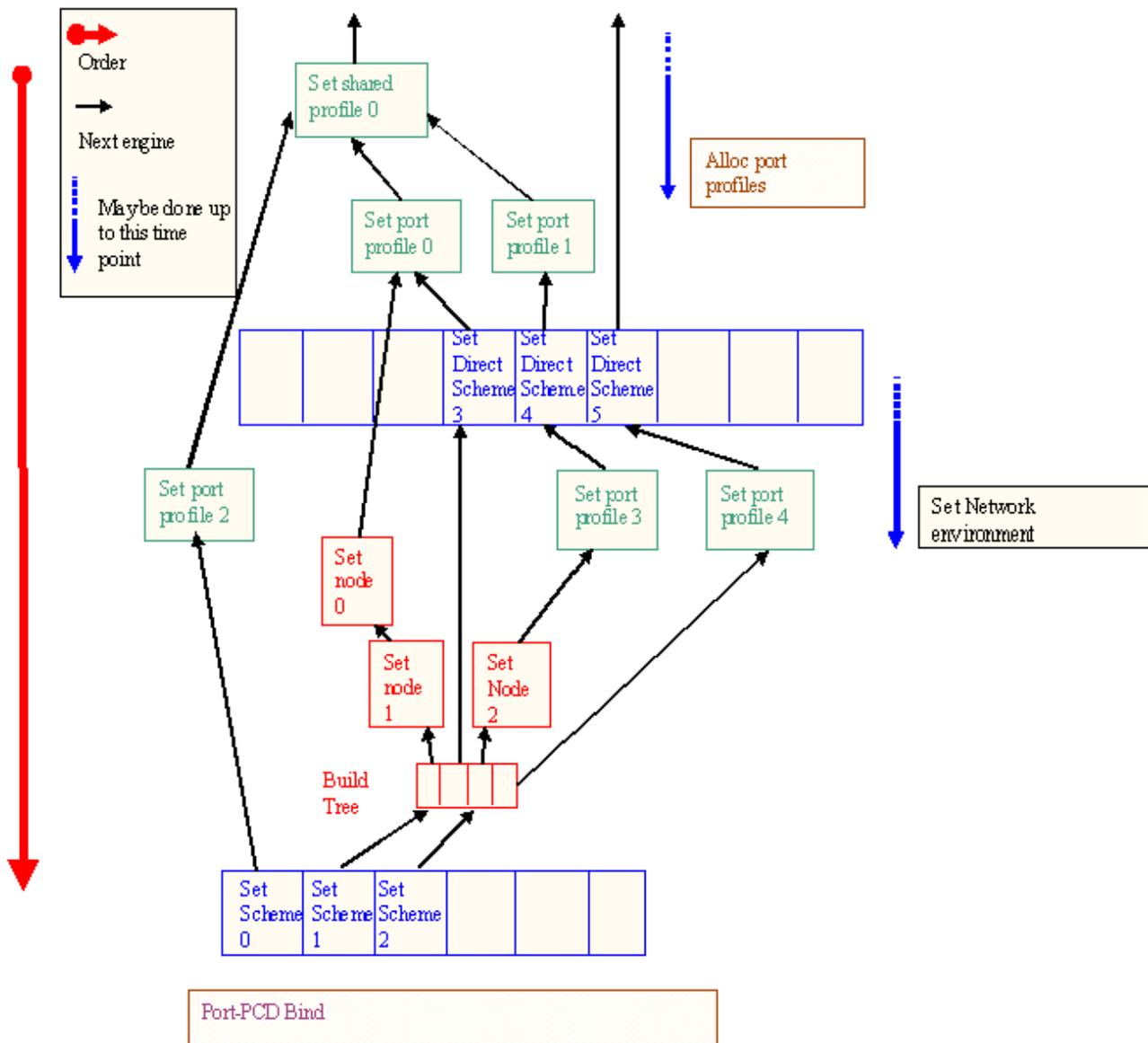


Figure 53. Definition Sequence

5.2.4.3.1.7.2.3.14 Host Command

Some PCD functionalities may be managed by either memory-mapped registers or by the host command mechanism to allow independent programming in a multipartition environment. In a single partition environment in the FMan driver, the host command mechanism is optionally used, but in a multipartition environment, wherever available, only the host command is used to prevent a risk of racing. The host command driver is a part of the PCD driver and is initialized internally by the driver, using user parameters.

When PCD is first initialized in a single-partition environment, the user must specify whether the host command should be used, and if so, host command parameters are required. In a multipartition environment, the use of the host command is forced and all host command parameters are required. When PCD initialization routine is called by the master/single partition driver, the user parameters include host command port parameters (such as port id, virtual address, and default queues) and the FMan Port for the host command is internally initialized.

5.2.4.3.1.7.2.3.15 PCD Statistics

The FMan PCD API provides access to all the statistics gathered by the FMan PCD engines hardware. Statistics is enabled by default but may be disabled/enabled at runtime using the dedicated API.

The following API routines may be called at any time after initialization to retrieve any of the following FMan PCD counters:

- `FM_PCD_GetCounter`
- `FM_PCD_KgSchemeGetCounter`
- `FM_PCD_PlcrProfileGetCounter`

5.2.4.3.1.7.2.3.15.1 Custom Classifier Statistics

A CC node supports statistics gathering on per-key basis. In order to enable statistics gathering by a CC node (Match table or Hash table), statistics mode must be provided upon initialization of that node and this will determine the statistics mode for all keys of the CC node.

Next, statistics should be enabled per-key, meaning statistics should be enabled for every key that the user wishes to monitor.

After these steps, the following API routines may be called to retrieve the statistics:

- `FM_PCD_MatchTableGetKeyCounter`
- `FM_PCD_MatchTableGetKeyStatistics`
- `FM_PCD_MatchTableFindNGetKeyStatistics`
- `FM_PCD_HashTableFindNGetKeyStatistics`

5.2.4.3.1.8 FMan Port Driver

The FMan Port driver module refers to the per-port features of the FMan, including port configuration and initialization, runtime functionalities and PCD binding.

5.2.4.3.1.8.1 FMan Port Hardware Overview

The FMan hardware supports a SoC dependent number of inline and offline FMan Ports of the following types:

- 1G Rx Ports
- 1G Tx Ports
- 10G Rx Ports (may be eliminated on some SoCs)
- 10G Tx Ports
- Offline/Host-command ports

Port configuration is controlled through a set of per-port, type-dependent memory mapped registers. I.e. Each port has its own memory map area. In addition, some FMan common registers also effect port behavior - for example, global resources such as tasks number are declared in the common registers are.

5.2.4.3.1.8.1.1 FMan Port Driver Software Abstraction

The FMan Port module is an independent module. On port configuration, the user selects the type and the mode of each port (Tx/Rx, 1G/10G, online/offline/Host command, regular/independent), and specifies the port index relative to its type. This index is not related to the hardware port id as described in the hardware spec.

The driver provides abstraction to the common/private division of registers location in the memory map. i.e. all registers that are logically relevant to the port are handled by the FMan Port driver, even if they physically belong to the common FMan memory map.

5.2.4.3.1.8.2 How to use the FMan Port Driver?

The following sections provide practical information for using the software drivers.

5.2.4.3.1.8.2.1 FMan Port Driver Scope

- FMan Port hardware structures configuration and enablement
- Resource allocation and management
- FMan port types support
- Offline-Parsing ports
- Independent-Mode
- Simple BMI-to-BMI (regular) mode
- PCD Binding
- Rate limiting
- Interrupt handling
- Statistics support

5.2.4.3.1.8.2.2 FMan Port Driver Sequence

- FMan Port Config routine
- [Optional] FMan Port advance configuration routines
- FMan Port Init routine
- FMan Port runtime routines
- FMan Port Free routine

5.2.4.3.1.8.2.3 FMan Port Driver Functional Description

The following sections describe main driver functionalities and their usage.

5.2.4.3.1.8.2.3.1 FMan Port Configuration and Initialization

On FMan Port driver initialization, the software configures all FMan Port registers. It supplies default values where no other values are specified, it enables hardware mechanisms and initializes software data structures for software management.

By the time initialization is done, FMan is ready to be used and any of the FMan sub-modules (FMan-Ports, MAC's etc.) may be initialized.

5.2.4.3.1.8.2.3.2 FMan Port Types

The driver provides API for the initialization of the following port types/modes:

- Tx 1G port
- Tx 1G port - independent mode
- Rx 1G port
- Rx 1G port - independent mode
- Tx 10G port
- Tx 10G port - independent mode
- Rx 10G port
- Rx 10G port - independent mode
- Offline Parsing Port

The driver also holds a single host-command port internally when mandatory (multi-partition environments) or when user explicitly requires it.

5.2.4.3.1.8.2.3.3 Independent-Mode

A port may be configured to operate in independent-mode. In such case no PCD may be defined. A slightly different set of parameters is required as the FMan functions differently.

5.2.4.3.1.8.2.3.4 Resource Management

FMan Port related resources (TNUMs, DMAs, FIFOs, etc.)- These resources are used by the BMI. The driver selects default values for these resources but they may be need some tuning depending on the specific application, based on the total number of ports used and the performance requirements of the system. The driver provides an API routine `FM_PORT_AnalyzePerformanceParams` that uses performance monitoring mechanism in order to see the resources utilization at runtime.

The FMan Port driver allocates its resources by calling the FMan "front-end" driver. The FMan "front-end" allocates the resources by calling the "back-end" through IPC if its in guest-mode or through direct call if its not in master-mode. The port driver does not access those resources at run-time; the resources are being used only by the hardware of a port.

PCD related resources (Keygen-schemes, policer-profiles, etc.)-During the initialization of the FMan-PCD driver on each partition, the driver allocates all the required resources (configurable by the user) through IPC call to the "back-end" driver. From that point, all the resources are being handled locally on the partition. Note, that all access to these resources are still done through host-command and that assures proper synchronization between different partitions (i.e. one can access these resources by mistake from a different partition in the system).

PCD Custom-Classifer tables-The CC tables are being allocated on the MURAM memory. This means that upon initialization of this partition, piece of MURAM should be allocated to the partition (according to how much the partition requires). From that point, the local PCD driver will manage the MURAM allocation by itself.

5.2.4.3.1.8.2.3.5 Virtual Storage Profiles Support

An FMan Port may use the legacy Physical Storage Profile or the Virtual Storage Profiles (VSP). This section will discuss the usage of VSP by an FMan port, while more information about the VSP mechanism which is implemented by the driver as separate entity `FM_VSP`, can be found in [FMan VSP Driver](#) on page 242.

When a user wants to set an Rx or OP port to work in virtualization mode using VSP's rather than the physical SP, user should call the function which allocates a storage profile window (range of VSPs allocated in continuously manner) to a port. The user should also define which profile in this range should be used as default SP; note that the default profile should be a relative index within the allocated window. Upon calling the window allocation routine, the driver enables virtual mode (i.e. using VSPs) for this port, allocates its profiles and defines default SP. In order to redirect a packet into a certain VSP, user may set the 'relative-VSP-id' within the PCD graph nodes (e.g. in the match-table entries). The value in the PCD graph nodes is port relative so if two ports are sharing the same PCD graph node (e.g. a match-table), the actual VSP will be selected by the 'relative-VSP-id' plus the port's base VSP as shown in the figure below.

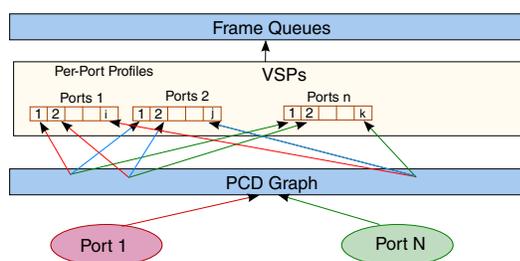


Figure 54. FMan PCD graph and the VSP selection

Rules and restrictions regarding the use of VSP:

- When called for Rx ports, the allocation routine expects also the handle of coupled Tx port as a parameter; the driver sets automatically the Tx port to work in VSP mode also and use the same default profile for this port.
- Storage Profiles windows may not overlap; i.e. sharing of VSPs between FM ports is not allowed by the driver.

- A call to the allocation routine requires that the FM port will be disabled. In the case of Rx port, coupled Tx port should also be disabled. When an FM Port (that has VSP mechanism enabled) is enabled, at least the default profile must be initialized.
- A call to the allocation routine may not be reverted, i.e. it's impossible to disable virtualization mode.
- Number of profiles to be allocated must be a power-of-2. In addition, the "base-profile" that will be allocated by the driver will be aligned to the number-of-profiles provided by the user.
- For FM-Port that works with VSP, its classification should also use VSP; i.e. classification (e.g. KG scheme or CC-node) should NOT try to revert from VSP to the FM-Port "physical" SP.
- When user frees all resources of FM port, the driver frees automatically VSP window which have been allocated for this port.

Initialization Sequence:

- Initialize FM Tx Port
- Initialize FM Rx Port
- Allocate VSP for FM Rx Port (thus enabling virtualization mode)
- Initialize default VSP (See [FMan VSP Driver](#) on page 242)
- Enable FM Ports

Free Sequence

- Disable Ports
- Free the default VSP
- Free FM Tx Port
- Free FM Rx Port

5.2.4.3.1.8.2.3.6 Rate Limiting

The driver supports the hardware mechanism of rate limiting for Tx ports. The runtime API consists of a number of parameters including a definition of the required rate (in KB/sec for Tx ports, in frame/sec for offline parsing ports) and refers to data rate rather than line-rate.

5.2.4.3.1.8.2.3.7 Simple BMI-to-BMI (regular) mode

This is the default FMan Rx/Offline Parsing Port mode. After Port initialization and prior to Port-PCD binding, all traffic will be received on the default Rx queue. This mode is called "BMI-To-BMI" as no PCD is involved in the data reception.

This mode is useful for the early state of a port as well as when major runtime PCD modification takes place. In such a case, sometimes the whole PCD functionality needs to be manipulated and the user should temporarily detach the Port from the PCD, receive all frames on the default Rx queue and only re-attach it to the PCD after the modifications have completed.

5.2.4.3.1.8.2.3.8 Port LIODN

An FMan Port LIODN is constructed out of a base and offset.

Upon FMan Port configuration, the user must specify the port's base LIODN.

For Rx ports, the user must also specify the LIODN offset for each port. No such configuration is required for Tx and Offline Parsing ports since on transmission, the offset LIODN is taken from the frames' FD. The FD is set according to the source of the frame - if transmitted by CPU, it is dynamically set by the QM SW portal. Another scenario is frames forwarded by other engines, in such a case their FD must contain the correct LIODN offset.

5.2.4.3.1.8.2.3.9 Port-PCD Binding

Ports may be linked to the PCD graph according to their PCD binding specifications and considering partition and Network Environment restrictions.

Following figure shows a schematic demonstration of possible port > PCD binding.

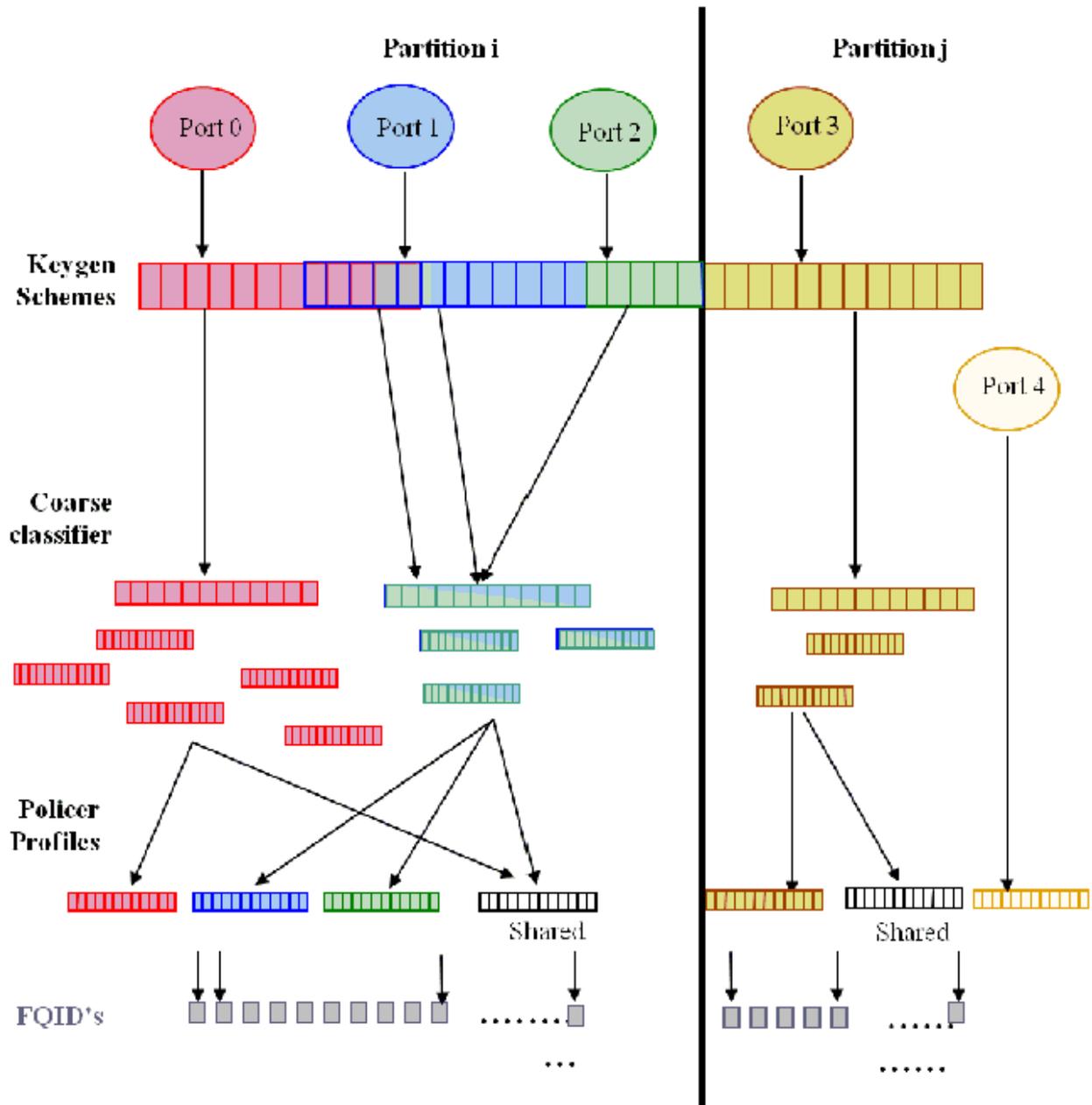


Figure 55. Port-To-PCD binding example

Once a set of PCD resources is set and organized as described above, a port may be bound to all or some of the resources by calling the `FM_PORT_SetPCD` routine. This routine, is referred to as the Port-PCD bind routine. It accepts a set of parameters that specify the PCD resources used by the port, configures PCD related parameters in the port, and binds PCD resources to the port. The `FM_PORT_DeletePCD` should be called when the port no longer needs the configured PCD functionality. This action is referred to as Port-PCD unbinding.

Another possible action that affects the Port-PCD relationship is calling `FM_PORT_DettachPCD` for a port that is bound to PCD. This causes the port to stop using the PCD functionalities, which results in all frames being passed to the default FQID. Note that calling `FM_PORT_DeletePCD` unbinds the port from the PCD functionalities by removing the connections, while `FM_PORT_DettachPCD` does not remove them but only causes the port to stop using them. To return to using the PCD, `FM_PORT_AttachPCD` should be called.

Certain runtime modifications may not be done directly, but require either the unbinding of PCD functionalities or PCD detaching. This should be done by calling the required delete/detach routines, making the desired changes, and calling set or attach to return to using the PCD. These actions will be referred to as resetting/detaching the Port-PCD. In the time between the calls of the two routines, the port continues to work, but its PCD functionalities are disabled. In both cases, all frames arriving at this time are enqueued to the default receive queue.

In the sections below, the relationship between the port and each of the PCD resources will be explained in terms of initialization and runtime modifications.

General

The port-PCD binding affects the flow of received frames on that port in terms of PCD functionality. The user must first define the general PCD for the port, using the following enumeration types, which define the superset of engines that may be used.

- `e_FM_PORT_PCD_SUPPORT_PRS_ONLY` (Use only Parser)
- `e_FM_PORT_PCD_SUPPORT_PLCR_ONLY` (Use only Policer)
- `e_FM_PORT_PCD_SUPPORT_PRS_AND_PLCR` (Use Parser and Policer)
- `e_FM_PORT_PCD_SUPPORT_PRS_AND_KG` (Use Parser and Keygen)
- `e_FM_PORT_PCD_SUPPORT_PRS_AND_KG_AND_CC` (Use Parser, Keygen and Custom Classifier)
- `e_FM_PORT_PCD_SUPPORT_PRS_AND_KG_AND_CC_AND_PLCR` (Use all PCD engines)
- `e_FM_PORT_PCD_SUPPORT_PRS_AND_KG_AND_PLCR` (Use Parser, Keygen and Policer)

Runtime Modifications: The engines set may be changed at runtime only by resetting the Port-PCD.

Available General Port API:

- `FM_PORT_SetPCD`
- `FM_PORT_DeletePCD`

Network Environment

When calling the Port-PCD binding routine, the user must specify a single NetEnv by passing its handle. This setting is used for the port parser and affects the PCD behavior.

Runtime Modifications: The NetEnv may not be modified at runtime. If the port requires a change of its NetEnv, it must first reset its Port-PCD connection, than use the PCD routines to do the required changes, and than re-connect to the PCD.

Parser

The hardware parser port configuration is taken directly from the NetEnv specified for the port. Other parsing configurations are explicitly defined by the user at the parameter's structure.

The software parser may be used on a per-port-per-header basis. When PCD is set per port, there is an option in the parser parameters to choose additional parameters per header. One of the optional per-header additional parameters is to enable the software parser for that header. When set, an index should be declared to select the software parser code. The header and index must be specified in the labels' table of the software parser code that was loaded on PCD initialization. Software parser enablement may be done for as many headers as required.

Runtime Modifications: Only the starting point of the parser may be changed on the fly. Any other changes require PCD resetting.

Available Port API:

- `FM_PORT_PcdPrsModifyStartOffset`

Keygen Schemes

In order for a port to use Keygen schemes, the port must be bound to those resources. The port may be bound to any number of schemes. At the port bind routine, the user passes a list of scheme handles, as returned by the server at scheme setting, for binding to the port. At least one scheme must be specified. All specified schemes must be valid at that time. If the initial scheme after the parser is used directly without using the match criteria, its id should be passed as one of the parameters to the Port-PCD binding routine.

Runtime Modifications: During runtime, new schemes may be set and then bound to an existing enabled port or existing schemes may be modified. Schemes that are not required by the port may be unbound. Note that when modifying existing schemes, all ports bound to those schemes are affected. If specific schemes are not required anymore, they must first be unbound from the port. If no other port is using them, they may be deleted. The selection of the initial scheme after parser (from direct to indirect and vice versa) may be also changed at runtime.

Available Port API:

- `FM_PORT_PcdKgBindScheme`
- `FM_PORT_PcdKgUnbindScheme`
- `FM_PORT_PcdKgModifyInitialScheme`

Custom Classifier graphs

If a port is using the Custom Classifier graph, an initialized Custom Classifier Root handle (as returned by the RootBuild routine) must be passed when calling the port bind routine.

Runtime Modifications: The CC graph (as well as the CC Root) itself may be modified at runtime, but ports binding to a CC Root may be changed only by detaching and then re-attaching the Port-PCD.

- `FM_PORT_PcdCcModifyTree`

Policer Profiles

Before any port profile is set, the profile allocation routine must be called to bind the port to the policer profile. This is required as the port's binding to the policer profile is not done using the port bind routine. It is only then that per-port profiles may be set, and the port bind routine is subsequently called. If Keygen or parser are not used (i.e. policer is reached directly after parser or from BMI), the port bind routine parameters must specify which policer profile is used (otherwise, no policer parameters are required).

Runtime Modifications: The initial profile selection may be changed during runtime. All profiles allocated to a port are in fact bound to this port, so no runtime binding/unbinding is possible. Uninitialized port profiles (profiles that were allocated for this port but not used) may also be set during runtime, or existing profiles may be modified. If specific profiles are not required anymore, they may be deleted. If a change in port profile allocation is required, follow the steps given below to reset the Port-PCD:

1. Port-PCD deleted
2. Profiles deleted and freed
3. New profiles allocated and set
4. Port-PCD set

Available Port API:

- `FM_PORT_PcdPlcrModifyInitialProfile`
- `FM_PORT_PcdPlcrFreeProfiles`
- `FM_PORT_PcdPlcrAllocProfiles`

5.2.4.3.1.8.2.3.10 Port-PCD Binding Changes

There are three levels of Port-PCD binding changes:

- **Basic Runtime Modifications**-May be invoked while PCD is active and on enabled ports using PCD.

- Port routines responsible for binding/unbinding to/from the modified resources.
 - FM_PORT_PcdKgBindScheme
 - FM_PORT_PcdKgUnbindScheme
- Port routines responsible for PCD change of behavior.
 - FM_PORT_PcdKgModifyInitialScheme
 - FM_PORT_PcdPlcrModifyInitialProfile
 - FM_PORT_PcdPrsModifyStartOffset
- **Port-PCD Detach Runtime Modifications**-For changes that require detaching the Port-PCD connection:
 - FM_PORT_PcdCcModifyTree

For these modifications, take the following steps:

 - Detach the port from its PCD resources by calling the Detach PCD routine (FM_PORT_DettachPCD). After this action, the port continues to work enqueueing all frames to the default receive FQID.
 - Call one of the two routines above.
 - Re-attach port to PCD resources by recalling the set PCD routine (FM_PORT_AttachPCD).
- **Port-PCD Reset Runtime Modifications**-For changes that require resetting of the port-PCD binding.

The following steps should be taken for any modification that is not listed under the last two items:

 - Unbind port from its PCD resources by calling the delete PCD routine (FM_PORT_DeletePCD). After this action the port will continue to work, enqueueing all frames to the default receive FQID.
 - Modify PCD resources-optional. The change may be only in the binding of the port and not on the resources. Note that the freeing and deleting of resources, and then allocating and setting resources, must be orderly, in the same manner as for initial PCD setting and final PCD deleting.
 - Bind port to PCD resources by recalling the set PCD routine (FM_PORT_DeletePCD)

All PCD routines listed above may be used for deleting and setting PCD resources. The following two routines below are used if a change of port profiles window is required (Other PORT routines are not needed as binding is done using SetPCD routine.):

- FM_PORT_PcdPlcrFreeProfiles
- FM_PORT_PcdPlcrAllocProfiles

5.2.4.3.1.9 FMan MAC Driver

The FMan MAC driver module refers to the FMan MAC controller functionalities including configuration and initialization as well as runtime and control.

5.2.4.3.1.9.1 FMan MAC Hardware Overview

The FMan hardware supports one or two kinds of MAC controllers - depending on SoC. All SoCs support three-speed Ethernet controller (dTSEC) interfaces to 10 Mbps, 100 Mbps, and 1 Gbps Ethernet/IEEE 802.3 networks which interfaces the media through external phy or SerDes device. Some SoCs also support 10 Gigabit Ethernet media access controller (10GEC) which interfaces to 10 Gbps Ethernet/IEEE 802.3ae networks via XAUI using the high-speed SerDes interface.

5.2.4.3.1.9.1.1 FMan MAC Software Abstraction

The driver provides a unique API serving both interfaces. If user tries to configure features that are supported only by one of the interfaces, an "unsupported" message will be displayed.

5.2.4.3.1.9.2 How To Use The FMan MAC Driver?

The following sections provide practical information for using the software drivers.

5.2.4.3.1.9.2.1 FMan MAC Driver Scope

This module represents the FMan MAC. It includes:

- FMan MAC hardware structures configuration and enablement
- FMan MAC controller runtime support
- PTP IEEE 1588 support
- MAC hash addressing
- Interrupt handling
- Statistics support

5.2.4.3.1.9.2.2 FMan MAC Driver Sequence

- FMan MAC Config routine
- [Optional] FMan MAC advance configuration routines
- FMan MAC Init routine
- FMan MAC runtime routines
- FMan MAC Free routine

5.2.4.3.1.9.2.3 FMan MAC Driver Functional Description

The following sections describe main driver functionalities and their usage.

5.2.4.3.1.9.2.3.1 FMan MAC Configuration and Initialization

On FMan MAC driver initialization, the software configures all FMan MAC registers. If required, MAC may be reset at that time. The driver supplies default values where no other values are specified, it defines IRQ's and sets IRQ handles. It enables hardware mechanisms and initializes software data structures for software management.

By the time initialization is done, FMan MAC is ready to be used and the relative FMan Ports may be initialized.

5.2.4.3.1.9.2.3.2 FMan MAC Addressing

On MAC initialization, the user must define a single MAC address. During runtime, the driver provides API for modifying this address and adding other addresses (depending on the specific MAC hardware support).

In addition, the driver supports the addition and removal of addresses to the MAC hash mechanism.

5.2.4.3.1.9.2.3.3 IEEE1588 Support

The driver provides the API to support the hardware IEEE1588 time-stamping. In order to use this feature, the user must first initialize the FM-RTC module. IEEE1588 functionality is always enabled on FM-MAC. Thus, no additional settings are required for the MAC. and the FM-MAC and only then they can enable this feature by calling `FM_MAC_Enable1588TimeStamp` routine. Once enabled, the user may also set the exception for receiving 1588 relevant interrupts on the MAC.

5.2.4.3.1.9.2.3.4 MAC Statistics

The driver provides statistics gathering support for all the standard (MIB) counters. For some controllers, it is necessary to use an interrupt driven mechanism for accounting for counters overflow and in order to keep track on the accurate counters. This mechanism may have some influence on performance, and therefor the driver supports statistics gathering in 3 levels:

- Full statistics-provides all standard counters but may reduce performance.
- Partial statistics-provides only special event counters (errors etc.). If selected, regular counters (such as byte/packet) will be invalid and will return -1.
- No statistics gathering.

5.2.4.3.1.10 FMan VSP Driver

The FMan VSP driver module refers to the software support provided for the Virtual Storage Profile mechanism.

5.2.4.3.1.10.1 FMan VSP Hardware Overview

VSPs may be used by user for virtualization. If a user is running with a multi-partitioned (or with a multiple software entities) system where a single MAC may be used by several software partitions/entities simultaneously, except for using a different FQID (that is already available in DPAA1.0), user may use a different VSP for each SW partition/entity; that way, the buffer may be private (rather than being shared as in DPAA1.0). It allows the virtualization of the buffer pool selection for frame storage (and other parameters related to storage in external memory) from the physical hardware ports. Using this mechanism, different packets received on the same physical port may be stored in different BM pools based on the frame header, in a similar way to FQID selection. VSPs are replacing the legacy, "physical", per-port BM Pool selection. A backward compatible mode exists and it is possible to use the original BM Pool selection, now referred to as "Physical SP".

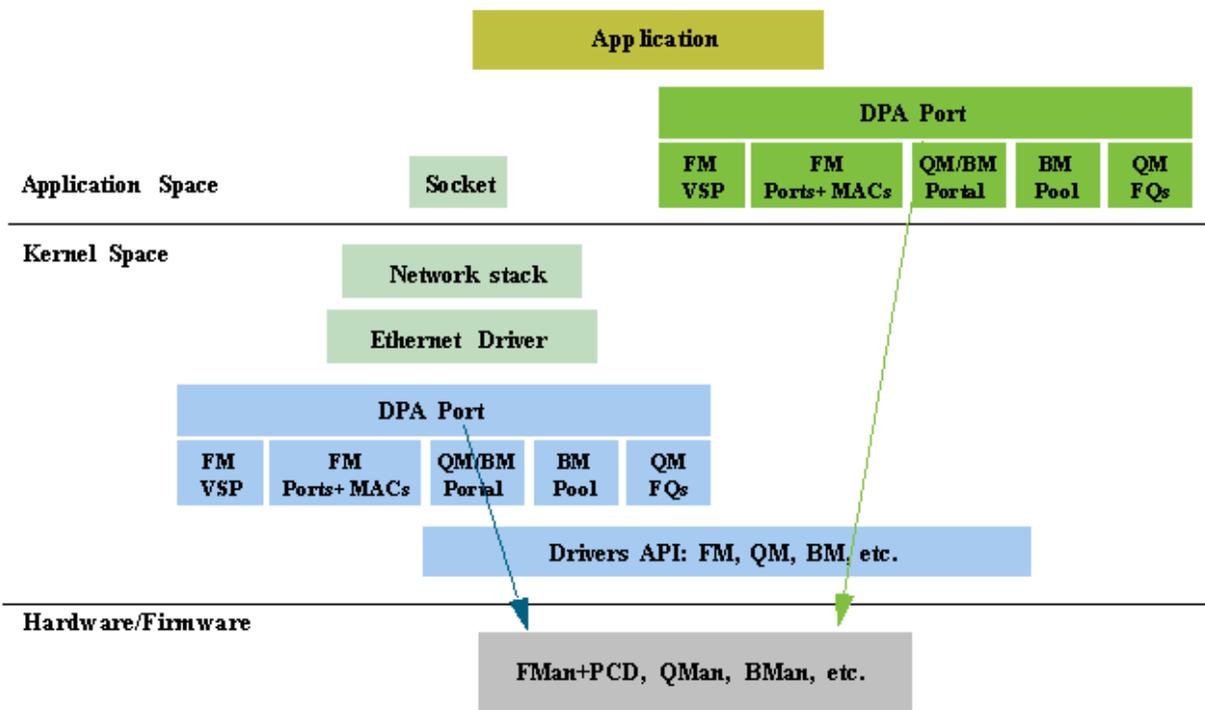


Figure 56. Virtualization Using VSPs

The global FMan module is in charge of the Virtual Storage Profiles entries management. On FMan initialization, the first VSP index dedicated to this partition must be defined (it should be an absolute index), and so is the total number of VSP's for this partition. Later, for each port using VSP's, a window of entries should be defined. VSPs may not be shared among FMan ports.

Each port has a default VSP. On each PCD classification, a VSP may be selected. Received packets will be written into the destination buffer according to the VSP parameters, while the VSP is selected according to the frame headers and the PCD configuration.

The VSP is implemented by the driver as separate entity, however, other modules of the FM driver are aware of this entity and interact with it. An FM VSP module represents a single storage profile.

The global FMan module is in charge of the Virtual Storage Profiles entries management. On FMan port initialization, if using VSP mode, it should allocate and bind to a range of VSP's. On the PCD, A decision is being taken by user on every node of the PCD graph whether to continue to work with previously defined VSP or to override with a new profile.

5.2.4.3.1.10.2 How To Use The FMan VSP Driver?

The VSP is implemented by the driver as separate entity, however, other modules of the FM driver are aware of this entity and interact with it. An FM VSP module represents a single storage profile.

The global FMan module is in charge of the Virtual Storage Profiles entries management. On FMan port initialization, if using VSP mode, it should allocate and bind to a range of VSP's. On the PCD, A decision is being taken by user on every node of the PCD graph whether to continue to work with previously defined VSP or to override with a new profile.

5.2.4.3.1.10.2.1 FMan VSP Driver Scope

This module represents the FMan VSP driver. It includes:

- FMan VSP hardware structures configuration and enablement
- Parsing of the buffer
- Statistics

5.2.4.3.1.10.2.2 FMan VSP Driver Sequence

This sequence includes other modules required for the VSP

- Definition of general VSP parameters on global FMan initialization
- FM Port initialization
- FM Port VSP window allocation
- FM Port enablement
- FMan VSP Config routine (for specific VSP's)
- [Optional] FMan VSP advance configuration routines (for specific VSP's)
- FMan VSP Init routine (for specific VSP's)

5.2.4.3.1.10.2.3 FMan VSP Driver Functional Description

The following sections describe main driver functionalities and their usage.

5.2.4.3.1.10.2.3.1 Virtual Storage Profile Initialization

The VSP's must be initialized prior to their usage. It is user's responsibility to initialize at least the default VSP for each port before enabling it. Similarly, it is their responsibility to initialize all other VSPs before a classification that may use some VSP is enabled.

Initializing a VSP defines the destination BM Pool buffer for a specific type of packets. It also defines the structure of the buffer - i.e. the data offset, the prefix content, etc.

5.2.4.3.1.10.2.3.2 Virtual Storage Profile Parsing

On VSP initialization, the user defines the buffer prefix content. Based on these requirements, the driver then defines the buffer prefix structure, i.e. data offset, whether certain information such as parse result should be copied to the external buffer and where it will be located. On buffer reception, the user may call VSP routines in order to get the data, as well as the buffer prefix sections such as parse result, time stamp, or Keygen output.

5.2.4.3.1.11 FMan RTC (IEEE 1588) Driver

The FMan RTC driver module refers to the software support provided for the IEEE 1588 hardware of the FMan.

5.2.4.3.1.11.1 FMan RTC Hardware Overview

The 1588 timer module interfaces to up to four 10/100/1000 or one 10G Ethernet MACs, providing current time, 2 alarms, and 2 fiper periodic pulse generators.

5.2.4.3.1.11.2 How To Use The RTC Driver?

The following sections provide practical information for using the software drivers.

5.2.4.3.1.11.2.1 RTC Driver Scope

This module represents the FMan 1588 driver. It includes:

- IEEE 1588 hardware configuration and enablement
- Support for alarm mechanism
- Support for periodic pulse
- Support for external trigger
- Runtime compensation tuning
- Interrupt handling

5.2.4.3.1.11.2.2 RTC Driver Sequence

- FMan RTC Config routine
- [Optional] FMan RTC advance configuration routines
- FMan RTC Init routine
- FMan RTC Enable routine
- FMan RTC runtime routines
- FMan RTC Free routine

5.2.4.3.1.11.2.3 RTC Driver Functional Description

The following sections describe main driver functionalities and their usage.

5.2.4.3.1.11.2.3.1 FMan RTC 1588 module utilization

The driver API provides interface to the 1588 hardware module. It initializes its registers to define the clock period and it supports the definition of the alarms and periodic pulses. Note that When setting periodic pulse, the RTC module must be disabled.

5.2.4.3.1.11.2.3.2 Utilizing IEEE1588 for MAC frames time stamping

Several FMan driver modules are involved in having the 1588 time stamping functionality activated: FMan-RTC, FMan-MAC, FMan-Port and FMan-PCD.

The initialization sequence is as described below:

After the Frame Manager is initialized, the FMan-RTC needs to be initialized by calling (with the appropriate parameters):

- FM_RTC_Config
- FM_RTC_Init

Next, the following routine should be called, only after MAC is initialized.

- FM_MAC_Enable1588TimeStamp

From this point and on all the Ethernet frames on this MAC are time-stamped. In order to obtain the timestamp, during the FMan Port configuration, the user must call the advance config routine:

- FM_PORT_ConfigBufferPrefixContent (with 'passTimeStamp' parameter set).

At run-time, for each received/confirmed frame, the user should call the following routine, passing it the frame's data pointer:

- FM_PORT_GetBufferTimeStamp

The routine will return the pointer to the time stamp.

5.2.4.3.1.11.2.3.3 Utilizing IEEE1588 for PTP

The sequence described in the previous section causes all the frames that are being received or transmitted by FMan to be time-stamped. However, if the user wants to distinguish PTP frames from other frames on a specific port, PCD rules need to be applied on the PCD graph for this port; i.e using the parser to recognize the PTP frame and then using an appropriate scheme to distinguish PTP frames and route them to the desired destination queues.

5.2.4.3.1.12 FMan MURAM Driver

The FMan MURAM driver module refers to the memory management of the FMan Multi User RAM.

5.2.4.3.1.12.1 FMan MURAM Hardware Overview

The MURAM is the internal memory of the FMan.

5.2.4.3.1.12.1.1 FMan MURAM Driver Software Abstraction

The FMan MURAM driver is a memory manager that allows partitioning of the MURAM. Upon initialization the user receives a handle that may be used by other modules in order to allocate and de-allocate memory blocks out of that MURAM partition.

5.2.4.3.1.12.2 How To Use The FMan MURAM Driver?

The following sections provide practical information for using the software drivers.

5.2.4.3.1.12.2.1 FMan MURAM Driver Scope

This module manages the FMan MURAM. It includes MURAM allocation and de-allocation of different sizes of required memory blocks.

5.2.4.3.1.12.2.2 FMan MURAM Driver Sequence

- FMan MURAM config and init routine
- FMan MURAM allot and free runtime routines
- FMan MURAM free routine

5.2.4.3.1.12.2.3 FMan MURAM Driver Functional Description

The FMan MURAM drivers supports MURAM memory blocks allocation and de-allocation. After initializing an MURAM partition, the user is normally required to pass its handle to other FMan driver modules. In this way, these modules may allocate and de-allocate memory blocks from this partition.

5.2.4.3.1.13 Supported Network Protocols

The following sections show the protocols that may be selected when defining NetEnv characteristics.

5.2.4.3.1.13.1 L2 Protocols

The following list shows the L2 protocols:

- `HEADER_TYPE_ETH`, with the following two options
 - `ETH_BROADCAST`
 - `ETH_MULTICAST`
- `HEADER_TYPE_VLAN`, with the following option
 - `VLAN_STACKED`
- `HEADER_TYPE_MPLS`, with the following option

Linux Kernel Drivers
DPAA 1.x Devices

- MPLS_STACKED
- HEADER_TYPE_PPPOE
- HEADER_TYPE_LLC_SNAP

5.2.4.3.1.13.2 L3 Protocols

The following list shows the L3 protocols:

- HEADER_TYPE_IPV4, with the following options
 - IPV4_BROADCAST_1
 - IPV4_MULTICAST_1
 - IPV4_UNICAST_2
 - IPV4_MULTICAST_BROADCAST_2
 - IPV4_FRAG_1
- HEADER_TYPE_IPV6, with the following options
 - IPV6_MULTICAST_1
 - IPV6_UNICAST_2
 - IPV6_MULTICAST_2
 - IPV6_FRAG_1
- HEADER_TYPE_GRE
- HEADER_TYPE_MINENCAP
- HEADER_TYPE_USER_DEFINED_L3

5.2.4.3.1.13.3 L4 Protocols

The following list shows the L4 protocols:

- HEADER_TYPE_TCP
- HEADER_TYPE_UDP
- HEADER_TYPE_SCTP
- HEADER_TYPE_DCCP
- HEADER_TYPE_IPSEC_AH
- HEADER_TYPE_IPSEC_ESP
- HEADER_TYPE_USER_DEFINED_L4

5.2.4.3.1.13.4 Private Headers

- HEADER_TYPE_USER_DEFINED_SHIM1
- HEADER_TYPE_USER_DEFINED_SHIM2

5.2.4.3.1.13.5 Fields Supported By Driver for Keygen Extraction

Fields supported as "full fields":

- HEADER_TYPE_ETH
 - NET_HEADER_FIELD_ETH_DA

- NET_HEADER_FIELD_ETH_SA
- NET_HEADER_FIELD_ETH_TYPE
- HEADER_TYPE_LLC_SNAP
 - NET_HEADER_FIELD_LLC_SNAP_TYPE
- HEADER_TYPE_VLAN
 - NET_HEADER_FIELD_VLAN_TCI
 - (index may apply:
 - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1,
 - e_FM_PCD_HDR_INDEX_LAST))
- HEADER_TYPE_MPLS
 - NET_HEADER_FIELD_MPLS_LABEL_STACK
 - (index may apply:
 - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1,
 - e_FM_PCD_HDR_INDEX_2,
 - e_FM_PCD_HDR_INDEX_LAST))
- HEADER_TYPE_IPv4
 - NET_HEADER_FIELD_IPv4_SRC_IP
 - NET_HEADER_FIELD_IPv4_DST_IP
 - NET_HEADER_FIELD_IPv4_PROTO
 - NET_HEADER_FIELD_IPv4_TOS
 - (index may apply:
 - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1,
 - e_FM_PCD_HDR_INDEX_2/e_FM_PCD_HDR_INDEX_LAST))
- HEADER_TYPE_IPv6
 - NET_HEADER_FIELD_IPv6_SRC_IP
 - NET_HEADER_FIELD_IPv6_DST_IP
 - NET_HEADER_FIELD_IPv6_NEXT_HDR
 - NET_HEADER_FIELD_IPv6_VER | NET_HEADER_FIELD_IPv6_FL | NET_HEADER_FIELD_IPv6_TC (must come together!)
 - (index may apply:
 - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1,
 - e_FM_PCD_HDR_INDEX_2/e_FM_PCD_HDR_INDEX_LAST))

NOTE

NET_HEADER_FIELD_IPv6_NEXT_HDR with e_FM_PCD_HDR_INDEX_LAST indication, applies to the very last next header indication, meaning the next L4, which may be present at the Ipv6 last extension. On earlier revisions this field applies to the Next-Header field of the main IPv6 header)

- HEADER_TYPE_IP
 - NET_HEADER_FIELD_IP_PROTO
 - (index may apply:

Linux Kernel Drivers

DPAA 1.x Devices

- e_FM_PCD_HDR_INDEX_LAST)
- NET_HEADER_FIELD_IP_DCSP
(index may apply:
 - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1)
- HEADER_TYPE_GRE
 - NET_HEADER_FIELD_GRE_TYPE
- HEADER_TYPE_ETH
 - NET_HEADER_FIELD_ETH_DA
 - NET_HEADER_FIELD_ETH_SA
 - NET_HEADER_FIELD_ETH_TYPE
- HEADER_TYPE_MINENCAP
 - NET_HEADER_FIELD_MINENCAP_SRC_IP
 - NET_HEADER_FIELD_MINENCAP_DST_IP
 - NET_HEADER_FIELD_MINENCAP_TYPE
- HEADER_TYPE_TCP
 - NET_HEADER_FIELD_TCP_PORT_SRC
 - NET_HEADER_FIELD_TCP_PORT_DST
 - NET_HEADER_FIELD_TCP_FLAGS
- HEADER_TYPE_UDP
 - NET_HEADER_FIELD_UDP_PORT_SRC
 - NET_HEADER_FIELD_UDP_PORT_DST
- HEADER_TYPE_UDP_LITE (relevant only if FM_CAPWAP_SUPPORT define)
 - NET_HEADER_FIELD_UDP_LITE_PORT_SRC
 - NET_HEADER_FIELD_UDP_LITE_PORT_DST
- HEADER_TYPE_IPSEC_AH
 - NET_HEADER_FIELD_IPSEC_AH_SPI
 - NET_HEADER_FIELD_IPSEC_AH_NH
- HEADER_TYPE_IPSEC_ESP
 - NET_HEADER_FIELD_IPSEC_ESP_SPI
- HEADER_TYPE_SCTP
 - NET_HEADER_FIELD_SCTP_PORT_SRC
 - NET_HEADER_FIELD_SCTP_PORT_DST
- HEADER_TYPE_DCCP
 - NET_HEADER_FIELD_DCCP_PORT_SRC
 - NET_HEADER_FIELD_DCCP_PORT_DST
- HEADER_TYPE_PPPOE
 - NET_HEADER_FIELD_PPPOE_PID
 - NET_HEADER_FIELD_PPPOE_SID

Fields supported as "from fields":

- `HEADER_TYPE_ETH` (with or without validation):
 - `NET_HEADER_FIELD_ETH_TYPE`
- `HEADER_TYPE_VLAN` (with or without validation):
 - `NET_HEADER_FIELD_VLAN_TCI`
 - (index may apply:
 - `e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1`,
 - `e_FM_PCD_HDR_INDEX_LAST`)
- `HEADER_TYPE_IPv4` (without validation):
 - `NET_HEADER_FIELD_IPv4_PROTO`
 - (index may apply:
 - `e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1`,
 - `e_FM_PCD_HDR_INDEX_2/e_FM_PCD_HDR_INDEX_LAST`)
- `HEADER_TYPE_IPv6` (without validation):
 - `NET_HEADER_FIELD_IPv6_NEXT_HDR`
 - (index may apply:
 - `e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1`,
 - `e_FM_PCD_HDR_INDEX_2/e_FM_PCD_HDR_INDEX_LAST`)

5.2.4.4 Frame Manager Driver API Reference

This document describes the interface (IOCTLs) to the Frame Manager Driver as apparent to user space Linux applications that need to use any of the Frame Manager's features. It describes the structure, concept, functionality, and low level API.

The link below leads to a supplemental directory. Download the file you need from this set.

[Frame Manager Driver API Reference](#)

5.2.4.5 Frame Manager Configuration Tool User's Guide

5.2.4.5.1 Frame Manager Configuration Tool User Guide

5.2.4.5.1.1 Introduction

The Frame Manager (FMan) is part of NXP's Data Path Acceleration Architecture (DPAA), a set of logical blocks that lets multiple processors (cores) interact with multiple network interfaces and accelerators with low software overhead.

The Frame Manager Configuration Tool (FMC Tool) is a command-line program that converts Parse-Classify-Police-Distribute (PCD) descriptions of network packet flows into hardware configuration code for the FMan's KeyGen, Controller, and Policer functions.

The tool provides an abstraction layer: You define your application's PCD requirements in a high-level, XML markup language (NetPDL with NXP extensions). The tool translates these definitions into code that initializes the FMan's registers and data structures. This abstraction makes learning low-level hardware details unnecessary, allows new users to be productive more quickly, and simplifies the programming task for everyone.

5.2.4.5.1.2 Frame Manager (FMan) Overview

The FMan block handles both inbound and outbound Ethernet frames.

For inbound frames, the FMan can perform multi-layer protocol header parsing, classification, policing, and distribution. For outbound frames, the FMan transmits frames in priority order.

Note: The FMan is a modular engine designed for use by many QorIQ multicore embedded devices. This document describes the FMan hardware and software as implemented for the QorIQ P4080 chip. That said, the information contained herein applies to all FMan-equipped devices.

5.2.4.5.13 FMan hardware architecture

The P4080 has two, identical Frame Manager instances. The architecture of a single FMan block is shown in [Figure 57. Frame manager \(FMan\) logical block diagram](#) on page 250.

As the figure shows, the FMan consists of several sub blocks. While this document assumes that you are familiar with the DPAA in general and the FMan in particular, the sections below provide a high-level description of these sub blocks as a refresher. These sections should help you better understand the FMC Tool documentation that follows.

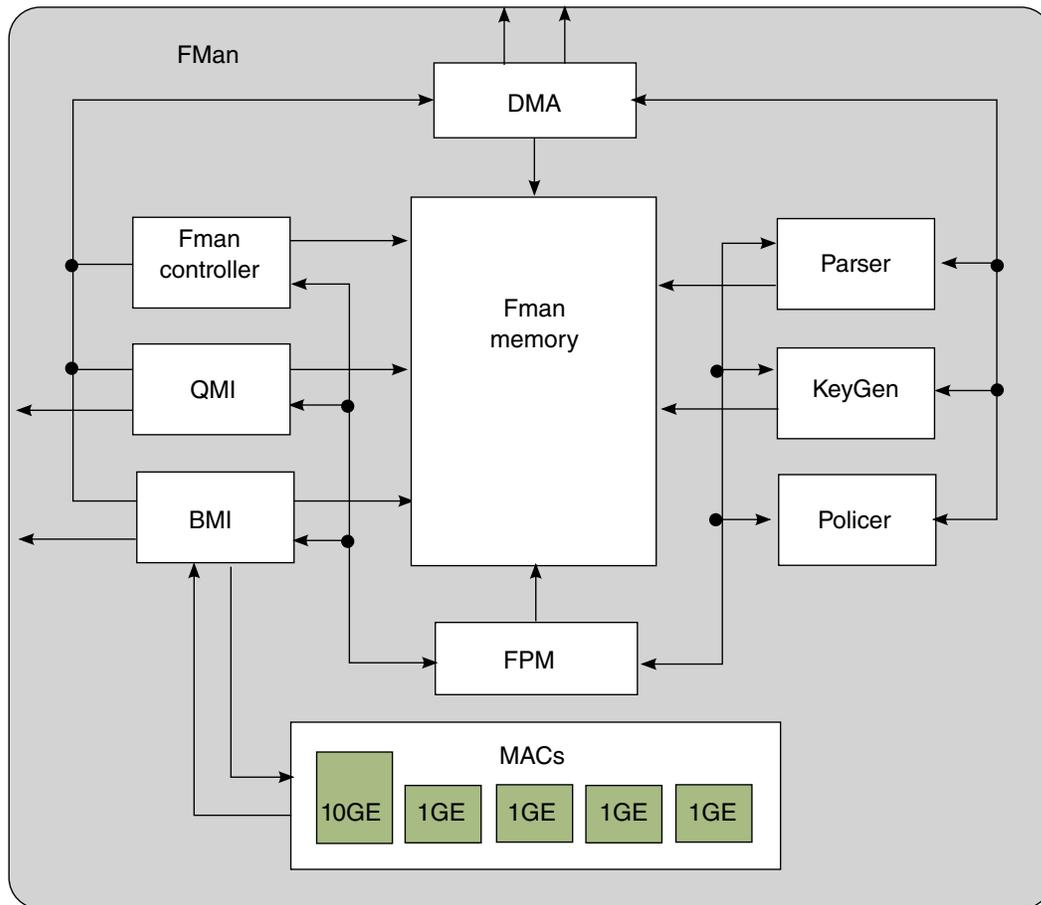


Figure 57. Frame manager (FMan) logical block diagram

5.2.4.5.13.1 FMan Media Access Controllers (MACs)

Each of the P4080's FMan blocks contains the following:

- (1) 10 Gigabit Ethernet media access controller (10GEC), interfacing to 10 Gbps Ethernet/IEEE 802.3ae networks via XAUI using the high-speed SerDes interface.
- (4) Data Path three-speed Ethernet controllers (dTSEC), interfacing to 10 Mbps, 100 Mbps, and 1 Gbps Ethernet/IEEE 802.3 networks. Individual dTSECs can use an RGMII or an SGMII interface.

See the QorIQ Data Path Acceleration Architecture (DPAA) Reference Manual for supported combinations of 10GEC and dTSECs per FMan and per P4080, as well as for detailed information about these MACs.

5.2.4.5.1.3.2 FMan FPM, DMA Controller, and Internal Memory

The Frame Processing Manager (FPM) coordinates tasks within the FMan block. The FPM ensures that frames arriving on a particular port (or destined for transmission on a particular port) are processed by the various FMan sub blocks in the appropriate order.

The FMan DMA controller performs the physical reads and writes required to move data between FMan memory and external memory, upon command from the FPM.

The FMan internal memory is multi-ported RAM within the FMan block that is used to buffer Ethernet frames (in both the Tx and Rx directions) and to hold data structures with both external relevance (Frame Descriptors) and internal relevance (Next Invoked Action (NIA) descriptors used by FMan sub blocks to direct the flow of frames through the FMan).

5.2.4.5.1.3.3 FMan Buffer Manager Interface (BMI)

The Buffer Manager Interface (BMI) is the FMan sub block that interacts with the DPAA's Buffer Manager (BMan) block to store and retrieve Ethernet frames to/from buffers in external memory.

On an inbound Ethernet frame, the BMI performs these operations:

1. Receives the frame from an FMan MAC
2. Stores the frame in FMan memory until the entire frame has been received
3. Calculates a raw, L4 checksum and passes it to the FMan's Parser sub block
4. Requests that the BMan allocate external memory in which to store the frame
5. Creates a frame descriptor (FD) for the frame (either a single-frame FD or a scatter/gather FD for frames that require more than one buffer)
6. Initiates DMA of the frame from FMan memory to external memory
7. Informs the FMan's Queue Manager Interface (QMI) sub block that a FD is available for enqueue to one of the Queue Manager block's frame queues (FQs)

On an outbound Ethernet frame, the BMI performs these operations:

1. Receives a FD dequeued by the QMI from one of the QMan block's FQs
2. Initiates DMA of the frame from external memory to FMan memory
3. Calculates a raw, L4 checksum for the frame
4. Transfers the frame to an FMan MAC
5. Passes the L4 checksum (offset/value) to the MAC
6. Signals the BMan to de-allocate the frame's buffer(s) once transfer to the MAC is complete

5.2.4.5.1.3.4 FMan Queue Manager Interface (QMI)

The FMan's Queue Manager Interface (QMI) sub block enqueues and dequeues work to frame queues (FQs), which are resources managed by the DPAA's QMan block. Typically, the QMI enqueues/dequeues a frame descriptor (FD) - a data structure that points to a buffer containing a frame; however, the QMI can also enqueue/dequeue other kinds of objects, such as those related to host commands/responses, offline parsing, Tx confirmation, and FMan error reporting.

5.2.4.5.1.3.5 FMan Parser

The FMan's Parser sub block parses fields in the hierarchy of protocol headers contained in a received frame.

The Parser is idle until notified by the FPM that a complete frame is available for parsing. The Parser then begins to parse the frame according to the rules specified in a user-written configuration (Policy file) that specifies a header at any offset from the beginning of the frame. The Parser can examine up to 256 bytes of a frame.

The result of parsing is a 'Next Invoked Action' descriptor that determines whether the frame is filtered out or passed to the FMan's KeyGen, Controller (also called the Classifier), or Policer sub block for further processing.

The Parser can be configured to write its 32-byte parse results to a reserved area of the buffer in which the BMI stored the frame. This feature makes it unnecessary for software to re-parse a frame in cases in which the FMan's only action is to distribute incoming frames across a number of frame queues.

The Parser performs multi-layer protocol header parsing and validation for a wide range of protocols and encapsulations. For the set of standard, stable protocols shown in [Figure 58. Standard Protocols - Hard-Coded Parse Tree](#) on page 253, the Parser uses hard-coded logic to parse the protocol headers. Parsing normally progresses from the lowest layer to the upper layers shown in the figure.

You can supplement the hard-coded support for standard protocols with 'soft' parse routines, thereby extending the Parser's capabilities to unforeseen combinations of standard protocols, new standards, and proprietary protocols with shim headers inserted between otherwise well-known protocols.

The FMC Tool can configure the Parser to use both hard- and soft-parser protocol header definitions. Details of the FMC Tool files used to describe both standard and custom protocol headers are provided in [Protocol files](#) on page 259.

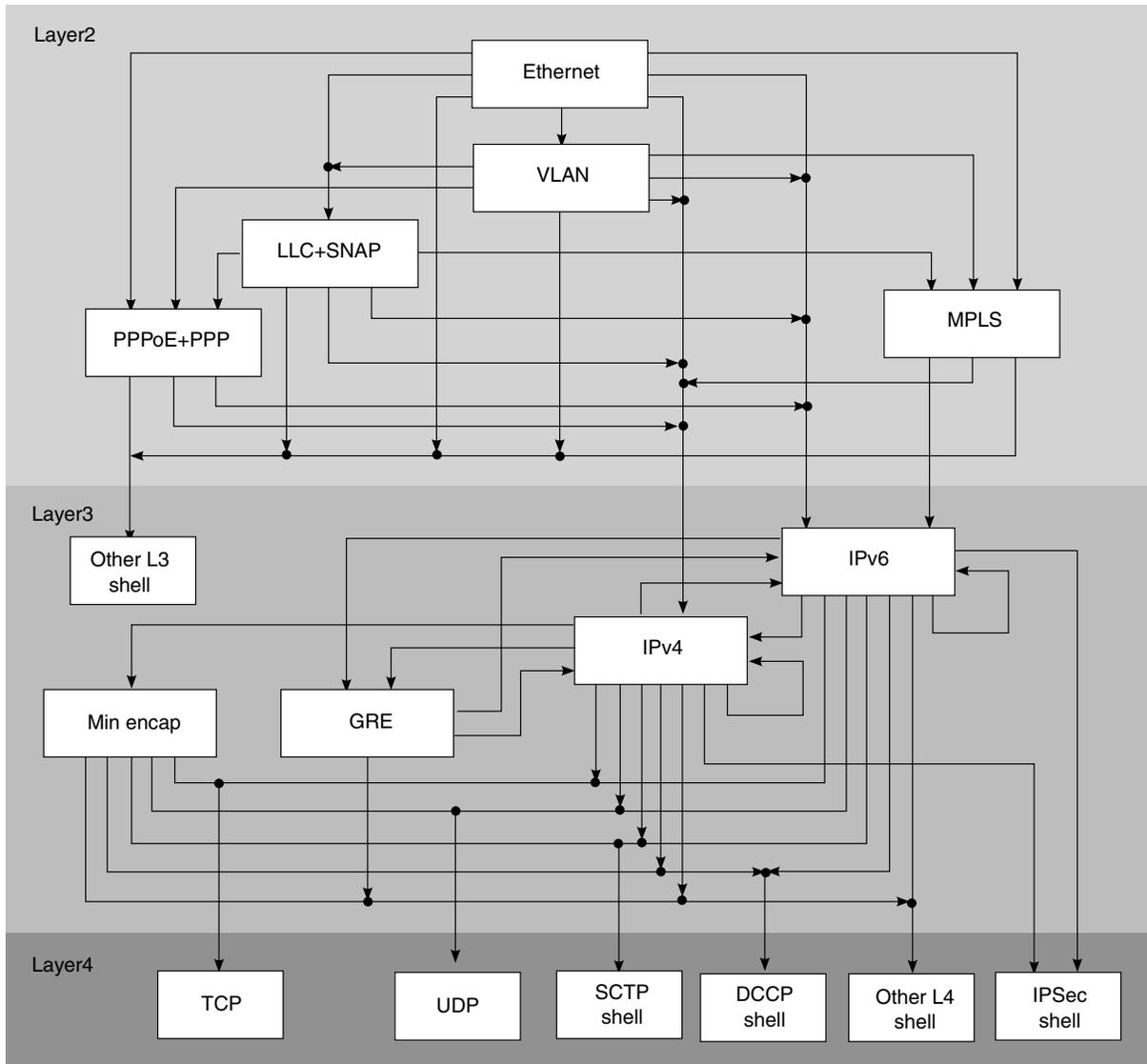


Figure 58. Standard Protocols - Hard-Coded Parse Tree

5.2.4.5.1.3.6 FMan Controller

The FMan Controller sub block (also called the Classifier) performs exact match frame classification. This capability lets your application identify control traffic (or other important subsets of traffic) that can be distinguished by a small number of exact-match rules and place these frames on a frame queue dedicated to this traffic type. All frames that do *not* meet an exact-match rule proceed to the KeyGen sub block, where they are distributed across a range of frame queues based on a hash function.

Exact-match frame classification uses rules tables stored in FMan memory. An FMan block can support classification at a line rate of up to 12 Gbps, assuming a three-level tree search. Each tree level can use up to 256 bytes of table space; however, the total rules table size for all three tree levels cannot exceed 512 bytes. For example, if classification is configured to perform an exact match on a 4-byte field and there are 32 exact-match rules defined, 128 bytes of rules table space is required.

The FMan Controller treats each rules table as an ordered list of entries and takes action on the first match.

5.2.4.5.1.3.7 FMan KeyGen

The FMan's KeyGen sub block performs line-rate packet distribution onto a range of frame queues (FQs) based on hashes of inbound frame header fields extracted by the Parser. The KeyGen sub block holds 32 key-generation schemes in internal memory, with each scheme generating a different Frame Queue ID (FQID) and Policer Profile (PP). Further, the KeyGen supports an option for a post-hash index.

By applying the same KeyGen hash scheme to all arriving frames, all frames belonging to the same coarse-grained flow are enqueued to the same frame queue (FQ). Ensuring that frames belonging to the same coarse-grained flow are enqueued to a single FQ guarantees that the fine-grained flows within the coarse-grained flow are dequeued and processed by the cores in order. [Figure 59. Algorithm for Calculating a Frame Queue ID](#) on page 254 shows the steps in the KeyGen sub block's frame queue ID calculation.



Figure 59. Algorithm for Calculating a Frame Queue ID

5.2.4.5.1.3.8 FMan Policer

The FMan's Policer sub block implements a two rate, three color marker (trTCM) traffic policing algorithm. Using the Policer, you can implement differentiated services at line speed on the FMan's receive path or its offline parse path.

The Policer holds 256 policing profiles in internal memory and can apply these profiles to the outputs of the FMan Controller and KeyGen frame processing stages to limit enqueues to frame queues.

Policer features:

- Implements RFC2698 and RFC4115
- Supports three-color traffic marking: "Green", "Yellow", and "Red"
- Supports four traffic measurements:
 - CIR - Committed Information Rate
 - CBS - Committed Burst Size
 - PIR - Peak Information Rate
 - PBS - Peak Burst Size
- Supports color-aware and color-blind metering
- Supports policing based on packet count, as well as byte count
- Includes a quick drop mode for "Red" packets
- Stores up to 256 policing profiles in internal FMan memory
- Maintains traffic statistics on a per-profile basis

5.2.4.5.1.4 FMC Tool Features

The FMC Tool can analyze input NetPDL and NetPCD XML files that define the parse, classify, police, and distribute behavior your application requires. The tool can then:

- Passes this information directly to the FMan by calling the appropriate FMan driver API functions. (See [FMC Tool - Runtime Environment Mode](#) on page 255.)
- Generate C source files containing this information that you can include in your application. (See [FMC Tool - Host Mode](#) on page 256.)

In more detail, the FMC Tool can perform the tasks listed below. The particular actions taken depend upon your application's requirements.

- Define the protocol stack
- Define a soft header examination sequence
- Configure the Policer sub block
- Configure frame distribution by defining how frames are assigned to particular frame queues
- Call hardware drivers to execute the current configuration
- Directly configure the FMan by executing on a target running embedded Linux (See [FMC Tool - Runtime Environment Mode](#) on page 255.)
- Indirectly configure the FMan by executing on a Linux or Windows host by generating C source code that configures the FMan. You include this code in your application. (See [FMC Tool - Host Mode](#) on page 256.)

5.2.4.5.15 FMC Tool Components and Packaging

The FMC Tool package contains these files:

- Host version of FMC Tool for desktop versions of Linux and Windows
- FMC Tool application for embedded Linux
- NetPDL file containing a description of each standard network protocol that the FMan's Hard Parser supports. This file is named `hxs_pdl_v3.xml` and is in the directory `/etc/fmc/config/`.

NOTE

For detailed information on NetPDL, go to <http://www.nbee.org/doku.php?id=netpdl:index>.

For documentation of NXP's customized version of NetPDL, see [NXP NetPDL Reference](#) on page 275.

5.2.4.5.16 FMC Tool - Runtime Environment Mode

In runtime environment mode, you run the FMC Tool on a target board from the Linux command line, passing several configuration files as arguments. The tool then calls the FMan Driver API functions required to configure the FMan block as specified in the supplied files.

When used in this way, the FMC Tool *directly* configures the FMan. In more detail, the FMC Tool passes the configuration it finds in its input files (along with compiled Soft Parser firmware) to the FMan driver which, in turn, modifies the FMan's configuration.

Note: The FMC Tool does *not* support dynamic FMan configuration; you can use the tool to configure the FMan just once, typically at application initialization.

As [Figure 60. FMC Tool, Runtime Environment - Input XML Files / FMan Driver API Calls](#) on page 256 shows, you pass these files to the FMC Tool as command-line arguments:

- Standard Protocol file - Optional; included in DPAA SDK; see [Standard Protocol File](#) on page 259 for more information.
- Custom Protocol file - Optional; user written; see [Custom Protocol File](#) on page 260 for more information.
- Policy file - Required; user written; see [Policy file](#) on page 261 for more information.
- Configuration file - Required; user written; see [Configuration File](#) on page 273 for more information.

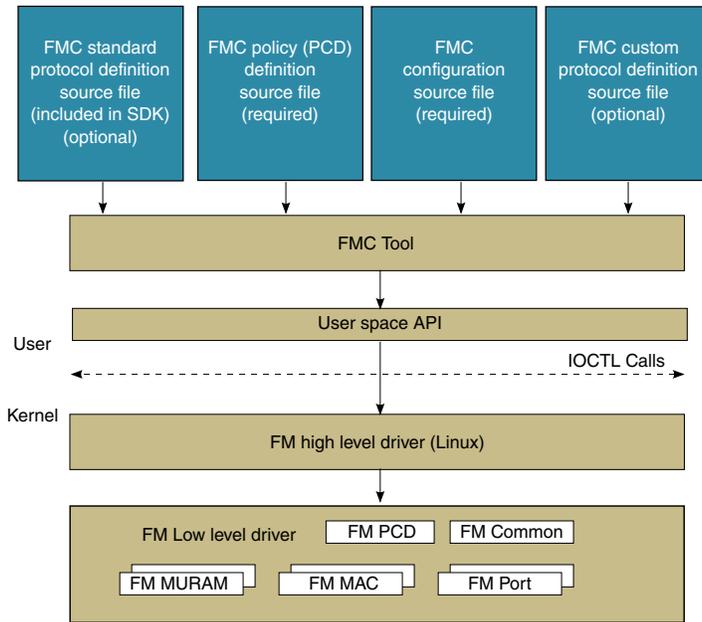


Figure 60. FMC Tool, Runtime Environment - Input XML Files / FMan Driver API Calls

See [FMC Tool Command-Line Arguments](#) on page 258 for documentation of each of the tool's command-line arguments.

Note: You should configure the FMan before you enable your Rx/Tx ports to send/receive traffic. If you do not, the FMan uses the default Rx and default Tx frame queues.

5.2.4.5.1.7 FMC Tool - Host Mode

In addition to running on a target board, the FMC Tool can execute on a host computer running Linux or Windows. When run on a host, the FMC Tool accepts the same input files as in runtime environment mode.

However, in host mode, the FMC Tool generates C source code files. This code calls the FMan driver functions required to implement the rules defined in the supplied input files. You can compile and link these files to produce a standalone executable that you can run by itself, or you can add them to your application.

Note: The FMC Tool does not support dynamic FMan configuration; you can use the tool to configure the FMan just once, typically at application initialization.

As [Figure 61. FMC Tool, Host Mode - Input XML Files / Generated C Source Code Files](#) on page 257 shows, in host mode, the FMC Tool generates C source code files from the input files listed below. (See [Host Mode Output - C Source Code Files](#) on page 257 for more information.)

- Standard Protocol File - Optional; included in DPAA SDK; see [Standard Protocol File](#) on page 259 for more information.
- Custom Protocol File - Optional; user written; see [Custom Protocol File](#) on page 260 for more information.
- Policy File - Required; user written; see [Policy file](#) on page 261 for more information.
- Configuration File - Required; user written; see [Configuration File](#) on page 273 for more information.

You pass these files to the FMC Tool as command-line arguments.

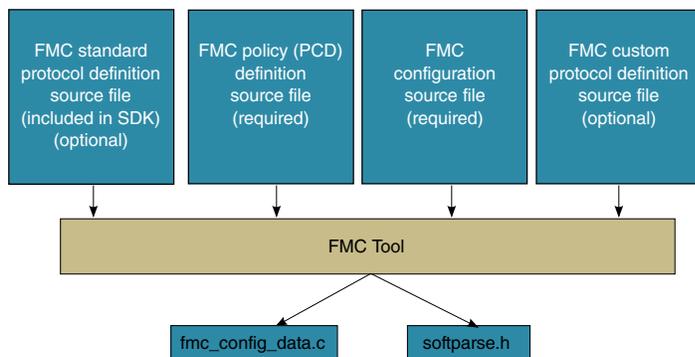


Figure 61. FMC Tool, Host Mode - Input XML Files / Generated C Source Code Files

See [FMC Tool Command-Line Arguments](#) on page 258 for documentation of each of the tool's command-line arguments.

5.2.4.5.1.7.1 Host Mode Output - C Source Code Files

When run in host mode, the FMC Tool generates C language source code files that make calls to FMan Driver API functions. These calls implement the behavior defined in the Configuration file, Policy file, and (optionally) Custom Protocol file passed to the tool from the command line. Typically, you include these source files in your project, so they are compiled and linked into your application binary. As a result, when you run your application, it automatically sets up the FMan to behave as required.

In more detail:

- When you supply a Policy file and a Configuration file, the tool generates a single source code file named "fmc_config_data.c".
- When you supply a Policy file, a Configuration file, *and* a Custom Protocol file, the tool generates two source code files: "fmc_config_data.c" and "softparse.h".

Contents of fmc_config_data.c

- #include software parser configuration "softparse.h" at the top of the file
- Initialization of FMC model structure 'fmc_model_t' with configuration data - This structure represents the data model for FMan hardware configuration according to input files

Using fmc_config_data.c

- FMC model structure must be used together with FMC model definition and FMC executer: 'fmc.h' and 'fmc_exec.c' files - These file are available in FMC source files location
- FMC model definition contains 'fmc_model' structure definition - This structure represents the FMC configuration model
- FMC executer contains 'fmc_execute' routine - This function configures the FMan hardware to behave as specified in the input files

Usage options:

- Compile and link these files together ('fmc_config_data.c', 'fmc.h', 'fmc_exec.c') and generate a standalone binary and run this binary to configure the FMan - In this case you must add a main() function that calls fmc_execute()
- Have your application call fmc_execute() - In this case you don't need to add a main() function

Contents of softparse.h

- Contains compiled firmware that controls the FMan sub blocks involved in parsing a custom protocol header
- Defines parameters such as code size, protocol to attach, and download base address

Using softparse.h - Automatically included in fmc_config.c if you pass the FMC Tool a Custom Protocol file

Note: You should configure the FMan before you enable your Rx/Tx ports to send/receive traffic. If you do not, the FMan uses the default Rx and default Tx frame queues.

5.2.4.5.18 FMC Tool Command-Line Arguments

The table below lists and describes the FMC Tool's command-line arguments.

Table 21. FMC Tool Command-Line Arguments

Command-Line Argument Syntax <i>(Both the verbose and abbreviated command forms are shown)</i>	Description
-d <pdl_file>, --pdl <pdl_file>	Path to and name of the Standard Protocol file. <i>(Optional)</i> You can use a full path or a relative path. See Standard Protocol File on page 259 for more information.
-p <pcd_file>, --pcd <pcd_file>	Path to and name of a Policy file. <i>(Required unless '--sp_only' is used)</i> You can use a full path or a relative path. See Policy file on page 261 for more information.
-c <data_file>, --config <data_file>	Path to and name of the Configuration file. <i>(Required unless '--sp_only' is used)</i> You can use a full path or a relative path. See Configuration File on page 273 for more information.
-s <custom_protocol_file>, --custom_protocol <custom_protocol_file>	Path to and name of the Custom Protocol file. <i>(Optional unless the '--sp_only' flag is used, in which case, this Custom Protocol file name is required.)</i> You can use a full path or a relative path. See Custom Protocol File on page 260 for more information.
-f --force	Applies the new configuration, without regard for the temporary information the FMC tool stored about the previous configuration. (Optional) When used in runtime environment mode (that is, on the target), the FMC Tool stores information about the previous configuration that the tool uses to properly unroll this configuration before applying the new one. There are cases (such as a non-destructive reboot) when such unrolling is not desirable. In such cases, use the --force switch.
-a, --apply	Apply the supplied configuration to the FMan rather than generating C source code. <i>(Optional; valid only when FMC Tool is executed in runtime environment)</i>

Table continues on the next page...

Table 21. FMC Tool Command-Line Arguments (continued)

Command-Line Argument Syntax (Both the verbose and abbreviated command forms are shown)	Description
--sp_only	Perform Soft Parser processing only. When this argument is supplied, the FMC Tool compiles just the Custom Protocol file, generates the file softparse.h, and exits. The file softparse.h contains C source code and custom protocol offsets. The tool creates softparse.h in the path from which the FMC Tool was executed. (Optional)
-w	Do not report warnings. (Optional)
--version	Display version information, then exit. (Optional)
-h, --help	Display usage information, then exit. (Optional)

5.2.4.5.1.9 The NetPDL and NetPCD XML Markup Languages

The Network Protocol Description Language (NetPDL) is an XML dialect that defines elements for describing protocols from OSI layer 2 to OSI layer 7. (For more information on NetPDL, see <http://www.nbee.org/doku.php?id=netpdl:index>.)

NXP uses NetPDL to define the standard protocols that are parsed by the FMan's Hard Parser. You cannot change these protocol descriptions. However, the SDK includes a Standard Protocol file that you can use as a reference.

In addition, you can use NetPDL (with slight semantic and syntactic differences) to define custom protocols that are parsed by the FMan's Soft Parser. This feature allows the FMan to handle any protocol that exists or that you define yourself.

Finally, NXP has extended NetPDL to create a language called NetPCD. You use the elements and attributes of NetPCD to define FMan parse, classify, police, and distribute behavior. The processing thus defined determines how frames move from block to block of the FMan.

The FMC Tool accepts files in NetPCD and NetPDL format as input.

5.2.4.5.1.10 Protocol files

For a protocol to be recognized by the FMC Tool, the protocol must be defined in one of two ways:

1. As a standard protocol within the Standard Protocol file (included in the SDK)
2. As a custom protocol within the Custom Protocol file.

Each file type is described in the sections that follow.

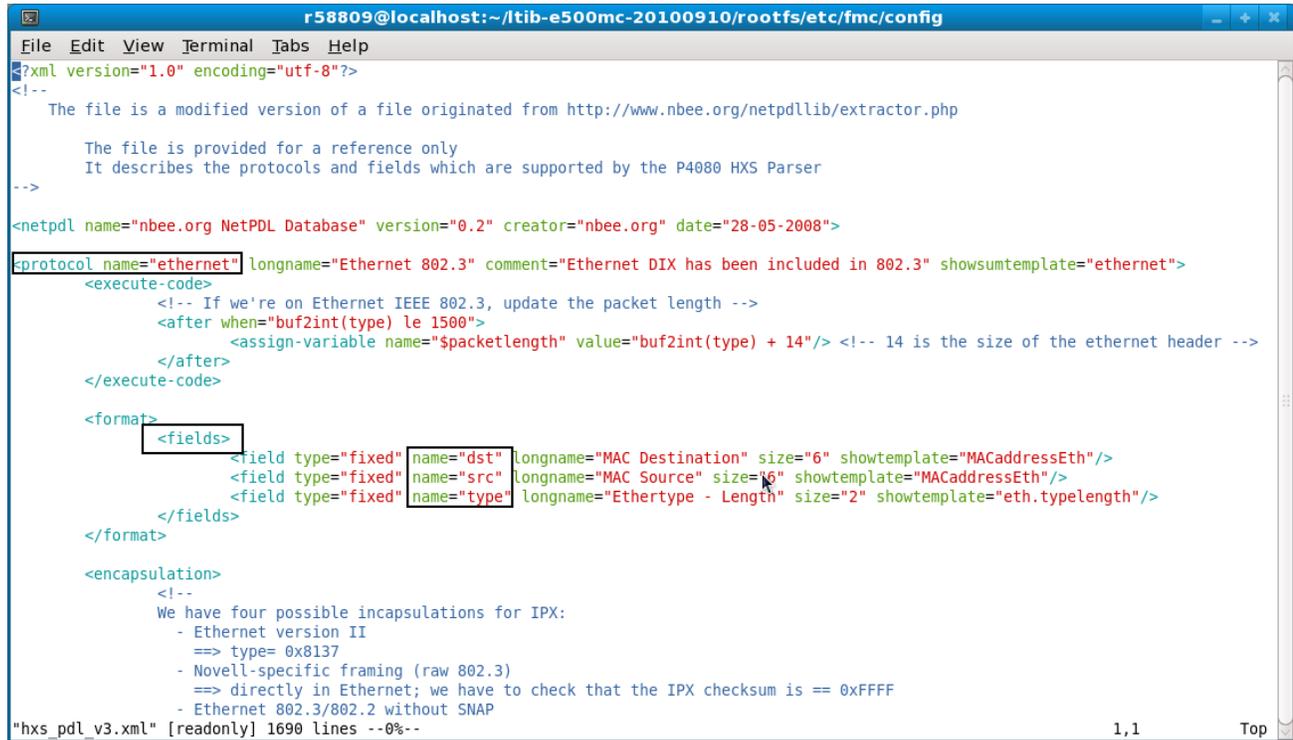
5.2.4.5.1.10.1 Standard Protocol File

The DPAA SDK includes a file called the Standard Protocol file. This file contains NetPDL (Network Protocol Description Language) markup that defines the fields in each standard protocol header that the FMan's Hard Parser can handle. In addition, for each standard protocol, the file includes NetPDL statements that define actions for the Hard Parser to take upon encountering an inbound instance of this protocol.

The Standard Protocol file is for the FMan's internal use only; you must therefore not change it. However, to write a Custom Protocol file and/or a Policy file, you sometimes need information the Standard Protocol file contains, such as the names of fields in a protocol's header.

For this reason, the SDK includes a copy of the Standard Protocol file in this directory: `/etc/fmc/config/hxs_pdl_v3.xml`.

To give you an idea what the markup in this file looks like, [Figure 62. Standard Protocol File NetPDL that Defines the Ethernet Protocol](#) on page 260 shows the NetPDL in the Standard Protocol file that defines the Ethernet protocol.



```
<?xml version="1.0" encoding="utf-8"?>
<!--
  The file is a modified version of a file originated from http://www.nbee.org/netpdllib/extractor.php

  The file is provided for a reference only
  It describes the protocols and fields which are supported by the P4000 HXS Parser
-->
<netpdl name="nbee.org NetPDL Database" version="0.2" creator="nbee.org" date="28-05-2008">
  <protocol name="ethernet" longname="Ethernet 802.3" comment="Ethernet DIX has been included in 802.3" showsumtemplate="ethernet">
    <execute-code>
      <!-- If we're on Ethernet IEEE 802.3, update the packet length -->
      <after when="buf2int(type) le 1500">
        <assign-variable name="$packetlength" value="buf2int(type) + 14"/> <!-- 14 is the size of the ethernet header -->
      </after>
    </execute-code>

    <format>
      <fields>
        <field type="fixed" name="dst" longname="MAC Destination" size="6" showtemplate="MACaddressEth"/>
        <field type="fixed" name="src" longname="MAC Source" size="6" showtemplate="MACaddressEth"/>
        <field type="fixed" name="type" longname="Ether type - Length" size="2" showtemplate="eth.typeLength"/>
      </fields>
    </format>

    <encapsulation>
      <!--
        We have four possible encapsulations for IPX:
        - Ethernet version II
          ==> type= 0x8137
        - Novell-specific framing (raw 802.3)
          ==> directly in Ethernet; we have to check that the IPX checksum is == 0xFFFF
        - Ethernet 802.3/802.2 without SNAP
      -->
    </encapsulation>
  </protocol>
</netpdl>
```

Figure 62. Standard Protocol File NetPDL that Defines the Ethernet Protocol

See the [Standard Protocol File - Excerpt](#) on page 327 topic to see a larger portion of the Standard Protocol file.

5.2.4.5.1.10.2 Custom Protocol File

The FMan's Hard Parser has built-in capability to handle a set of widely used, standard protocols, such as IPv4. The FMan also has a Soft Parser, which has the ability to process custom protocols.

Of course, for the Soft Parser to recognize a custom protocol, you must first provide a definition of this protocol. To do this, you create a Custom Protocol file, which consists of NetPDL markup that defines the fields in a custom protocol's header along with the actions you want the Soft Parser to take upon these fields. You then pass this file to the FMC Tool, which compiles it and passes the result to the FMan.

Note: Some elements in the NetPDL language are relevant only if used with a protocol analysis tool. The FMC Tool does *not* support these elements; instead, the tool supports only those elements that are applicable to the FMan block. Further, although it is based on NetPDL, the markup for a custom protocol does not strictly follow NetPDL rules. As a result, it is highly recommended that you become familiar with the [NXP NetPDL Reference](#) on page 275 topic, which fully documents the custom version of NetPDL used in custom protocol definitions.

See [Custom Protocol File - GTP Protocol](#) on page 334, for an example of a custom protocol definition file containing XML that defines the GPRS Tunneling Protocol (GTP).

Note: If your application does not use a custom protocol, you do not have to create a Custom Protocol file. Further, if your application uses *multiple* custom protocols, you can (and must) define them in a single Custom Protocol file; you can pass just one Custom Protocol file to the FMC Tool.

The general structure of a Custom Protocol file is shown below.

```
<netpdl> <!-- only one instance -->
  <protocol> <!-- one or more instances -->

    <format> <!-- only one instance -->
      <fields> <!-- only one instance -->
        <field/> <!-- one or more instances -->
      </fields>
    </format>

    <execute-code> <!-- zero or one instance -->
      <before> <!-- zero or one instance -->
      </before>

      <after> <!-- zero or one instance -->
      </after>
    </execute-code>

  </protocol>
</netpdl>
```

5.2.4.5.11 Policy file

The policy file defines how each inbound frame is parsed, classified, policed, and distributed by the various FMan sub blocks.

A policy file consists of NetPCD markup, where NetPCD is NXP's extension to NetPDL, an XML markup language for describing networking protocols. The elements and attributes of NetPCD let you define the parse, classification, policing, and distribution behavior your application requires. See [NetPCD Reference](#) on page 297 for documentation of each NetPCD element and its attributes.

A Policy file can have these sections:

- Distribution (required) - Contains one or more distribution definitions, each of which:
 - Specifies the protocol(s) a frame must contain to match the distribution
 - Defines how to handle matching frames
- Policy - (required) - Contains one or more policy definitions, each of which:
 - Is associated with an FMan port
 - Contains a prioritized list of distributions
- Classification (optional) - Contains one or more classification blocks, each of which:
 - Defines key/value/action tuples, which the FMan's Controller sub block stores in a lookup table
 - Compares the specified fields in the current frame header to each value in this table and, upon a match, takes the specified action
- Policer (optional) - Contains up to 256 policer profiles, each of which can be used to:
 - Take action upon frames without regard to traffic flow rate
 - Take action upon frames based on the RFC-2698 two-rate, three-color policing scheme
 - Take action upon frames based on the RFC-4115 two-rate, three-color, differentiated services scheme

Note: When you run the FMC Tool, you must pass it a Policy file or the '--sp_only' flag. Otherwise, the program will exit and print an error message.

Figure 63. High-level Structure of a Policy File

```
<netpcd> <!-- only one instance -->
  <distribution> <!-- one or more instances -->
  </distribution>

  <policy> <!-- one or more instances -->
    <dist_order> <!-- one instance -->
      <distributionref/> <!-- one or more instances -->
    </dist_order>
  </policy>

  <classification> <!-- optional, may have more than one instance -->
  </classification>

  <policer> <!-- optional, may have more than one instance -->
  </policer>
</netpcd>
```

5.2.4.5.1.11.1 Distribution Section

The Distribution *section* of the Policy file contains one or more 'distribution' *elements*. While 'distribution' elements can appear anywhere in the Policy file, they often appear at the top of the file.

Typically a 'distribution' contains child elements that define:

- Frame match rules
 - These rules define the conditions an inbound frame must meet to match (and therefore be handled by) this distribution
 - Use the 'protocols' element and/or the 'key' element to define match rules
- Frame handling rules
 - These rules determine what a distribution does with matching frames
 - Use the 'queue' and 'key' elements to hash frames, so they are evenly spread over a range of frame queues
 - Use the 'action' element to pass the frame to another element in the Policy file for further processing

Figure 64. Example Distribution Elements

```
<!-- distribution that matches all frames containing an IPv4 header -->
<!-- hashes these frames, so they are spread evenly over 32 frame queues -->
<distribution name="hash_ipv4_src_dst_dist0">
  <!-- frame match rule -->
  <key>
    <fieldref name="ipv4.src"/>
    <fieldref name="ipv4.dst"/>
  </key>

  <!-- frame handling rule -->
  <queue count="32" base="0x400"/>
</distribution>

<!-- distribution that matches frames containing Eth/VLAN/IPv4/UDP/GTP headers -->
<!-- passes all matching frames to the "dl_vlan_clasifif" classification element -->
<distribution name="dl_eth_vlan_ipv4_udp_gtp_dist">
  <!-- frame match rule -->
  <protocols>
```

```

<protocolref name="ethernet"/>
<protocolref name="vlan"/>
<protocolref name="ipv4"/>
<protocolref name="udp"/>
<!--shim1 is custom protocol defined for GTP -->
<protocolref name="shim1"/>
</protocols>

<!-- frame handling rule
<action type="classification" name="dl_vlan_classif"/>
</distribution>

```

See [The distribution element](#) on page 299 for complete documentation of this element.

Evenly Distributing Frames over a Range of Frame Queues

One frequent use of the 'distribution' element is to distribute frames evenly over a range of frame queues. If each available core is configured to pull from the same number of queues in the range, this even spreading balances the work each core must perform.

In this scenario, the FMan's KeyGen sub block uses values in the frame's header and in the child elements of the distribution as inputs to a hash algorithm that generates a 24-bit FQID within a range of FQIDs. The KeyGen sub block then places the frame on the frame queue identified by this FQID.

Here is the KeyGen's algorithm for generating a FQID:

1. Extract and concatenate the protocol header fields specified by the 'key' child element
2. Hash the resulting string to a 64-bit CRC
3. Shift the CRC right by the number of bits specified in the 'shift' attribute of the 'key' element to move the desired bits to the 24 least significant bit positions
4. Zero-extend the bit mask specified by the 'queue' child element ('count' attribute – 1) to 24 bits
5. Bitwise AND the result with the shifted CRC
6. Bitwise OR the result with the value specified by the 'combine' child element - repeat for each 'combine' element
7. Add the result to the base FQID specified by the 'base' attribute of the 'queue' child element

[Figure 65. KeyGen Algorithm for FQID Calculation](#) on page 263 shows the algorithm the KeyGen sub block uses to calculate a FQID.

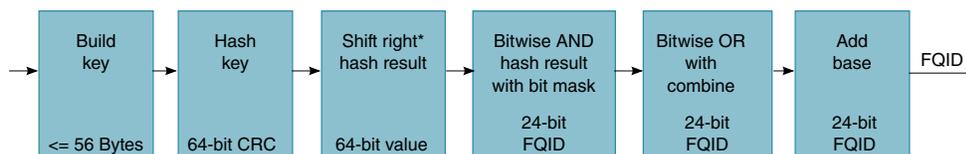


Figure 65. KeyGen Algorithm for FQID Calculation

* The 'key' element has an optional 'shift' attribute whose value defines the number of bits by which the hash result is right shifted. The default value for the shift attribute is zero.

Example KeyGen FQID Calculation

The series of figures that follow shows which child elements and attributes of a distribution block the KeyGen sub block uses in its FQID calculation.

[Figure 66. FQID Calculation - Elements/Attributes Used for Key, Bit Mask, and Base FQID](#) on page 264 shows where in the KeyGen sub block gets the inputs for the hash, shift right, bitwise AND, and "add base" parts of its FQID calculation.

```

r58809@localhost:~/l1b-no-hv/l1b-e500mc-20091218/rpm/BUILD/lw
File Edit View Terminal Tabs Help
<distribution name="eth_dist"
  description="Ethernet protocol based distribution">
  <queue count="0x400" base="0x81000"/>
  <key>
    <fieldref name="ethernet.src"/>
    <fieldref name="ethernet.dst"/>
  </key>
  <combine portid="true" offset="10" mask="0xFF" />
  <combine frame="112" offset="2" mask="0xFF" />
</distribution>
32,2-9 9%
  
```

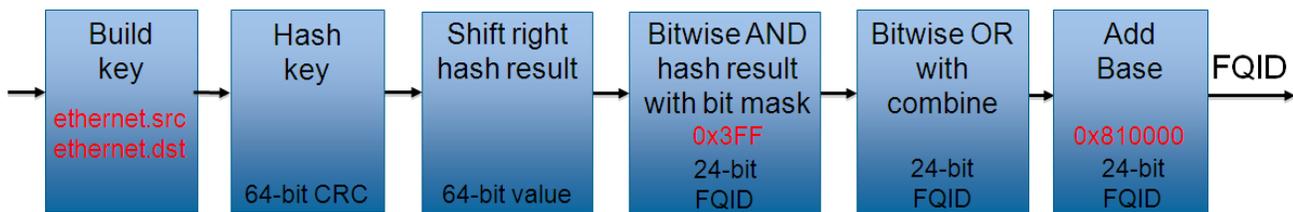


Figure 66. FQID Calculation - Elements/Attributes Used for Key, Bit Mask, and Base FQID

Figure 67. FQID Calculation - A 'combine' Element that Uses the 'portid' Attribute on page 264 shows a 'combine' element that includes a 'portid' attribute that is set to "true". In addition, the element's 'offset' attribute is "10", and its 'mask' is "0xFF". This markup instructs the KeyGen sub block to perform the "bitwise OR" part of the FQID calculation. In more detail, for this markup, the KeyGen does these things:

- Bitwise ANDs the 8-bit logical port ID (defined in the Configuration file) of the port on which the current frame arrived with the 8-bit mask in the 'combine' element.
- Bitwise ORs (inserts) the 8-bit result at the specified offset (10 bits) within the 24-bit FQID (where offset 0 signifies the FQID's most significant bit).

Note: Each FMan port can be assigned an 8-bit logical port ID by adding markup to the Configuration file. To do this, assign an 8-bit value to the 'portid' attribute of each 'port' element to which you want to assign a logical port ID. The Hard Parser puts this value (if defined) in the parse results array, where the a KeyGen sub block can get it.

```

r58809@localhost:~/l1b-no-hv/l1b-e500mc-20091218/rpm/BUILD/lw
File Edit View Terminal Tabs Help
<distribution name="eth_dist"
  description="Ethernet protocol based distribution">
  <queue count="0x400" base="0x81000"/>
  <key>
    <fieldref name="ethernet.src"/>
    <fieldref name="ethernet.dst"/>
  </key>
  <combine portid="true" offset="10" mask="0xFF" />
  <combine frame="112" offset="2" mask="0xFF" />
</distribution>
32,2-9 9%
  
```

Figure 67. FQID Calculation - A 'combine' Element that Uses the 'portid' Attribute

Figure 68. FQID Calculation - A 'combine' Element that Uses the 'frame' Attribute on page 265 shows a 'combine' element that includes a 'frame' attribute. This markup instructs the KeyGen sub block to:

- Get the 8 bits at offset 112 in the current frame header.

- Bitwise AND this value with the 8-bit mask (0xFF) specified in the 'combine' element
- Bitwise OR (insert) the 8-bit result at the specified offset within the 24-bit FQID (where offset 0 signifies the FQID's most significant bit).

Note: The value of the 'frame' attribute is an offset (in bits) from beginning of the current frame. The KeyGen sub block gets the byte at this offset for its FQID calculation. The value of 'frame' must be divisible by 8, so the bit it references is on a byte boundary.

```

r58809@localhost:~/ltib-no-hv/ltib-e500mc-20091218/rpm/BUILD/lw
File Edit View Terminal Tabs Help
<distribution name="eth_dist"
    description="Ethernet protocol based distribution">
  <queue count="0x400" base="0x81000"/>
  <key>
    <fieldref name="ethernet.src"/>
    <fieldref name="ethernet.dst"/>
  </key>
  <combine portid="true" offset="10" mask="0xFF" />
  <combine frame="112" offset="2" mask="0xFF" />
</distribution>
32,2-9 9%
  
```

Figure 68. FQID Calculation - A 'combine' Element that Uses the 'frame' Attribute

Finally, Figure 69. FQID Calculation - combine Elements Used in Bitwise OR on page 265 shows where the KeyGen sub block plugs the values from each of the combine elements into the bitwise OR part of the FQID calculation.

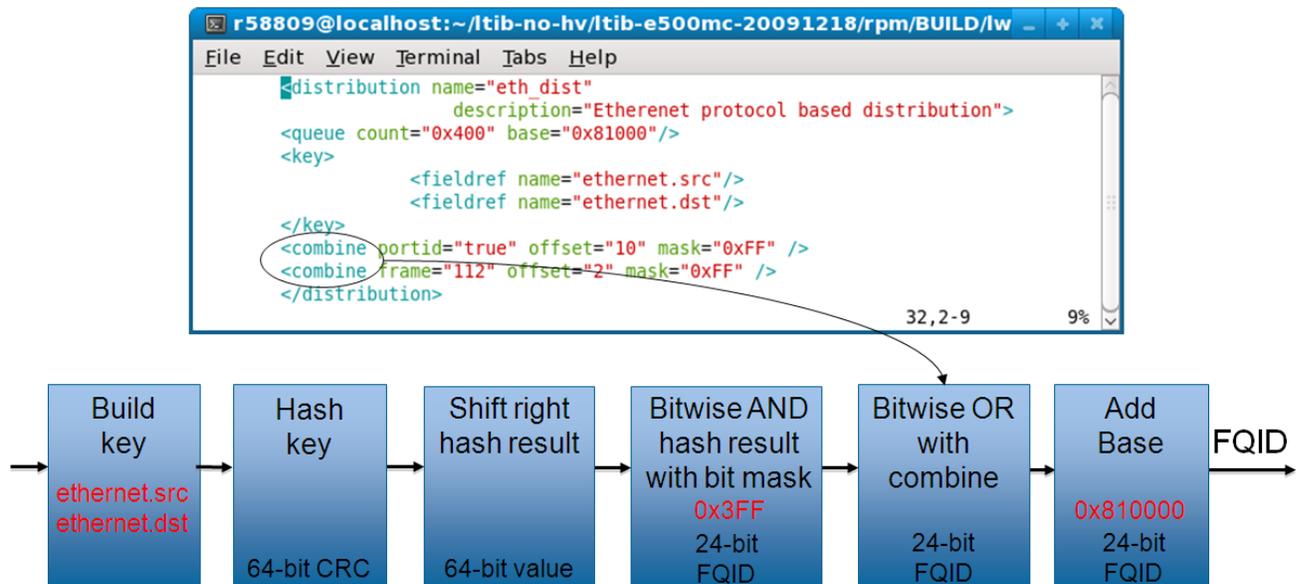


Figure 69. FQID Calculation - combine Elements Used in Bitwise OR

FQID Formula

$$\text{FQID}[0:23] = (\text{Shifted Hash Key}[0:23] \& \text{Hash Mask}) \mid \text{Data0}[0:23] \mid \text{Data1}[0:23] \mid \dots \mid \text{Data7}[0:23] \mid \text{FQID Base Address}$$

In sum, use the child elements/attributes of the 'distribution' element to provide the values on the right side of the FQID equation.

5.2.4.5.1.11.2 Policy Section

The *Policy section* of the *Policy file* consists of one or more 'policy' *elements*. While 'policy' elements can appear anywhere in the Policy file, they typically follow the last 'distribution' element in the file.

Each 'policy' element defines a set of *candidate* distributions that the FMan can apply to inbound frames. The particular distribution the FMan applies to a given frame depends on these factors:

- The position of each distribution in the 'policy' element's distribution order list
- The definition of each of these distributions

Candidate distributions are listed in *priority* order. As a result, if two or more distributions in the list match the current inbound frame, the FMan applies the first matching distribution because this distribution has higher priority.

How does the FMan know which policy (that is, which prioritized list of distributions) to apply to the traffic received on a particular Ethernet port? The Configuration file provides the connection.

In a Configuration file, you must enter one 'port' element for each FMan port your application uses. Further, the port element has a required attribute - the 'policy' attribute - whose value must match the name of one of the policy elements in the Policy file, thereby defining the policy (that is, the ordered list of distributions) that the FMan will apply to all traffic received on a port. In sum, the value of a port element's policy attribute in the *Configuration* file ties the port identified by this element to a policy element in the *Policy* file.

In a Configuration file:

- A port can be assigned a single policy
- Multiple ports can be assigned the same policy
- A port can have just one active policy at a time

Typically, you assign one policy to each port your application uses.

Example 1 - Simple Use of the Policy Element

Configuration File

```
<!-- The port element assigns the dl_policy policy to the 10 Gbps port of FMan 1 -->
<!-- Policy dl_policy is defined in the Policy file - see next code snippet -->
<cfgdata>
  <config>
    <engine name="fm1">
      <port type="10G" number="0" policy="dl_policy"/>
    </engine>
  </config>
</cfgdata>
```

Policy File

```
<!-- A policy element that defines how to apply two distributions -->
<!-- These distributions are defined elsewhere in the Policy file -->
<!-- This policy is assigned to an Ethernet port by the Configuration file above -->
<policy name="dl_policy">
  <dist_order>
    <distributionref name="dl_eth_vlan_ipv4_udp_gtp_dist"/>
    <distributionref name="garbage_dist"/>
  </dist_order>
```

```
</policy>
```

In the example above, the Configuration file assigns the policy named 'dl_policy' to the 10 Gbps port of the p4080 chip's second FMan (fm1). As a result, the FMan first tries to match each frame that arrives on this port to the 'dl_eth_vlan_ipv4_udp_gtp_dist' distribution since it appears first in the 'policy' element's distribution order list. Whether the frame matches depends on the definition of the 'dl_eth_vlan_ipv4_udp_gtp_dist' distribution, which is not shown. If the frame matches, it is handled according to the rules this distribution defines. If the frame does not match, the FMan next compares it to the 'garbage_dist' distribution since it appears second in the distribution order list. Because of this distribution's definition (also not shown), it matches all frames, thereby guaranteeing that every frame is handled in one way or the other.

See [The policy element](#) on page 298 for complete documentation of this element.

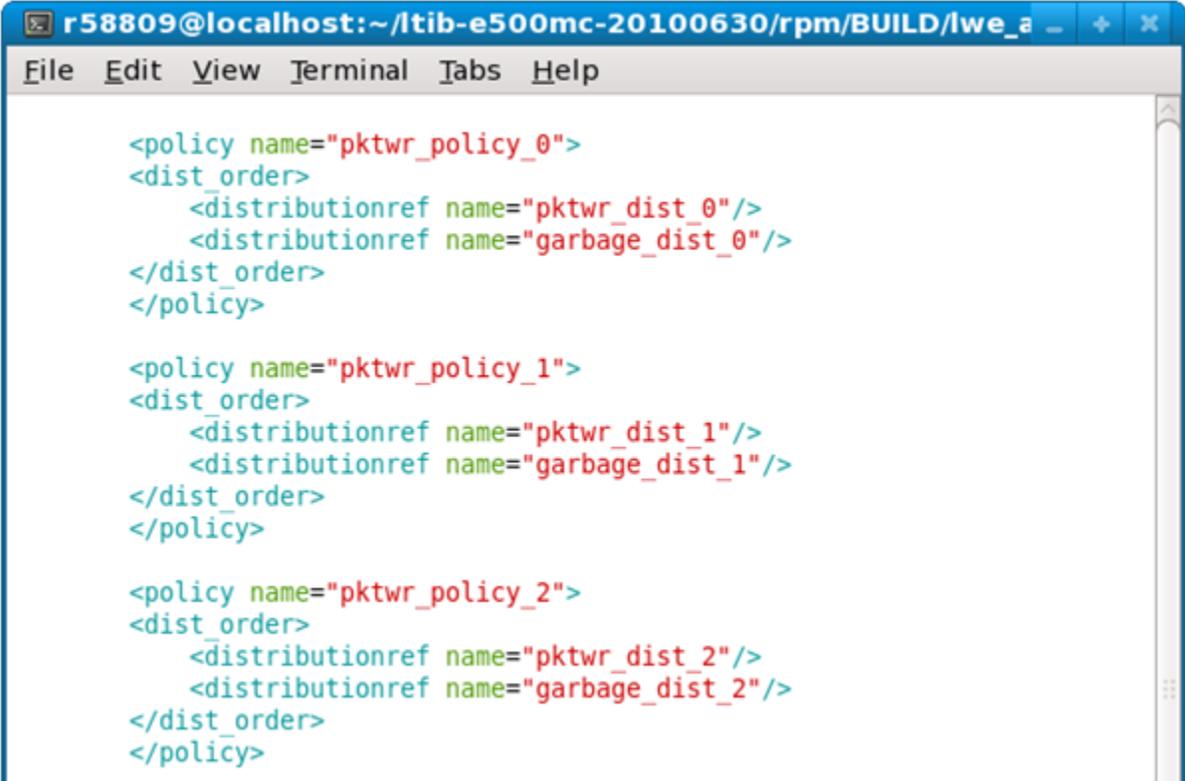
Example 2 - More Complex Use of the Policy Element

[Figure 70. More Complex Policy File - 1](#) on page 267 shows the Policy file from the pktwire application. This application requires a more complex use of policies and distributions than shown in the previous example.

This Policy file defines ten 'policy' elements - pktwr_policy_0, pktwr_policy_1, ... pktwr_policy_9 - some of which are shown in the figure.

A Configuration file (not shown) assigns each of these policies to one of the p4080's ten FMan ports - five on the first FMan (fm0) and five on the second FMan (fm1).

Note: Not all QorIQ devices have two FMans. Nor does every FMan have five Ethernet ports. See the reference manual for your QorIQ device to determine the number of FMans and FMan ports this device supports.



```
r58809@localhost:~/ltib-e500mc-20100630/rpm/BUILD/lwe_a
File Edit View Terminal Tabs Help

<policy name="pktwr_policy_0">
  <dist_order>
    <distributionref name="pktwr_dist_0"/>
    <distributionref name="garbage_dist_0"/>
  </dist_order>
</policy>

<policy name="pktwr_policy_1">
  <dist_order>
    <distributionref name="pktwr_dist_1"/>
    <distributionref name="garbage_dist_1"/>
  </dist_order>
</policy>

<policy name="pktwr_policy_2">
  <dist_order>
    <distributionref name="pktwr_dist_2"/>
    <distributionref name="garbage_dist_2"/>
  </dist_order>
</policy>
```

Figure 70. More Complex Policy File - 1

The Policy file also defines ten distributions - pktwr_dist_0, pktwr_dist_1, ... pktwr_dist_9 - some of which are shown in [Figure 71. More Complex Policy File - 2](#) on page 268.

As mentioned above, each of these distributions is assigned to a policy which, in turn, is assigned to a port. A frame "matches" the distribution assigned to the port on which the frame arrived if its header contains both the ipv4.src and ipv4.dst fields.

For each frame that matches, the KeyGen sub block computes a hash result using the concatenation of the `ipv4.src` and `ipv4.dst` fields as the hash key. The KeyGen sub block then uses the hash result to compute a FQID. (See the [Distribution Section](#) on page 262 topic for detailed coverage of the KeyGen's FQID calculation algorithm.)

The resulting FQID is in the range specified by the 'queue' element. For example, for distribution "pktwr_dist_0", the resulting FQID will be in range 0x2800 – 0x281F.



```
<netpcd xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="xmlProject/pcd.xsd" name="example"
description="PktWire configuration">

  <distribution name="pktwr_dist_0">
    <queue count="32" base="0x2800"/>
    <key>
      <fieldref name="ipv4.src"/>
      <fieldref name="ipv4.dst"/>
    </key>
  </distribution>

  <distribution name="pktwr_dist_1">
    <queue count="32" base="0x400"/>
    <key>
      <fieldref name="ipv4.src"/>
      <fieldref name="ipv4.dst"/>
    </key>
  </distribution>

  <distribution name="pktwr_dist_2">
    <queue count="32" base="0x800"/>
    <key>
      <fieldref name="ipv4.src"/>
      <fieldref name="ipv4.dst"/>
    </key>
  </distribution>
```

Figure 71. More Complex Policy File - 2

The Policy file also defines ten distributions - `garbage_dist_0`, `garbage_dist_1`, ... `garbage_dist_9` - some of which are shown in [Figure 72. More Complex Policy File - 3](#) on page 269.

Note that these distributions do not have a 'key' element. As a result, all frames "match" these distributions. For 'garbage_dist_0', the resulting FQID is always 0xb1 since the queue element specifies just one frame queue and the base FQID value is 0xb1.

A screenshot of a terminal window with a blue title bar. The title bar text is 'r58809@localhost:~/ltib-e500mc-20100630/rpm/BUILD/lwe_a'. The terminal has a menu bar with 'File', 'Edit', 'View', 'Terminal', 'Tabs', and 'Help'. The main content is XML code for policy distributions. The first distribution is 'pktwr_dist_9' with a queue count of 32 and base 0x2400, and a key containing 'ipv4.src' and 'ipv4.dst'. The following three are 'garbage_dist_0', 'garbage_dist_1', and 'garbage_dist_2', each with a queue count of 1 and different bases (0xb1, 0x51, 0x11).

```
<distribution name="pktwr_dist_9">
<queue count="32" base="0x2400"/>
<key>
    <fieldref name="ipv4.src"/>
    <fieldref name="ipv4.dst"/>
</key>
</distribution>

<distribution name="garbage_dist_0">
<queue count="1" base="0xb1"/>
</distribution>

<distribution name="garbage_dist_1">
<queue count="1" base="0x51"/>
</distribution>

<distribution name="garbage_dist_2">
<queue count="1" base="0x11"/>
</distribution>
```

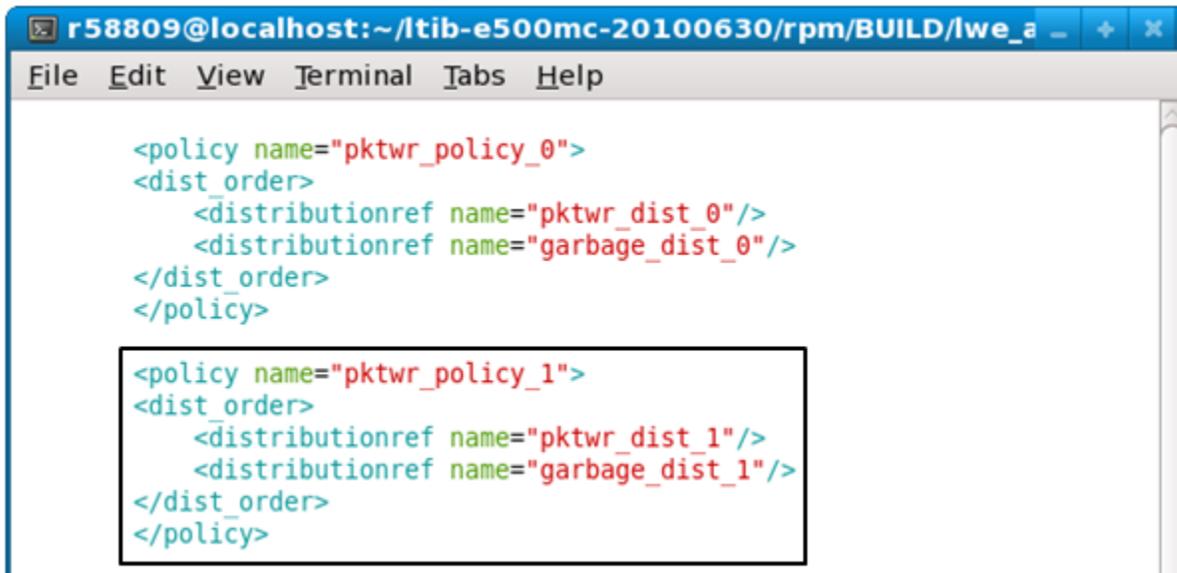
Figure 72. More Complex Policy File - 3

Let's say that an FMan port is tied to policy 'pktwr_policy_1' - highlighted in [Figure 73. More Complex Policy File - 4](#) on page 270.

This policy instructs the FMan to first attempt to distribute frames arriving on this port using the 'pktwr_dist_1' distribution. If the current frame does not include the ipv4.src and ipv4.dst fields, the policy instructs the FMan to try the next distribution in the policy's distribution order list.

In this example, the next distribution is "garbage_dist_1" which, due to the absence of a 'key' element, matches *all* frames and enqueues them to the single frame queue defined by the 'count' and 'base' attributes of its queue element.

Note: It is common for the last distribution in a distribution order list to be a "catch all", like the default case in a C switch statement; however, this is not a requirement.



```
<policy name="pktwr_policy_0">
<dist_order>
  <distributionref name="pktwr_dist_0"/>
  <distributionref name="garbage_dist_0"/>
</dist_order>
</policy>

<policy name="pktwr_policy_1">
<dist_order>
  <distributionref name="pktwr_dist_1"/>
  <distributionref name="garbage_dist_1"/>
</dist_order>
</policy>
```

Figure 73. More Complex Policy File - 4

5.2.4.5.1.11.3 Classification Section

The Classification section of the Policy file is optional. Use it to specify exact match frame classification.

A classification specifies the action to perform on a frame when the values of the specified fields in a frame's protocol header match a predefined value. You can specify as many predefined value/action pairs as desired, as well as a default action.

A classification starts with a 'classification' element, which is a container for these child elements:

- A 'key' element that defines the header fields (in protocol.field form) to use in the exact match operation
- One or more 'entry' elements, each of which defines a value to which the specified fields are compared and a 'queue' and/or 'action' element that defines what to do with the frame upon a match
- An optional 'action' element that defines the default action to take if none of the exact match conditions is met

The FMC Tool uses the information in these child elements to populate the FMan Controller's rules table. At runtime, the Controller uses this information to extract the specified fields from the specified protocol header, compare these fields to the specified values and, upon a match, take the specified action.

See [The classification element](#) on page 308 for complete documentation of this element.

Example

The example below shows a Policy file containing a 'classification' element.

The 'policy' element named 'policy_0' lists two distributions to try, 'udp_dist' and 'non_udp_dist'.

Note: For a classification block to be applied to a frame, the frame must first match a distribution that transfers control to this classification via an 'action' element. In other words, the "source engine" of the Classifier is always a 'distribution' element.

The 'udp_classif' classification element specifies an exact-match lookup on the ipv4.dst field. If this field's value is:

- 0xC0A81402, the frame is placed on the queue whose FQID is 0x200
- 0xC0A81404, the frame is placed on the queue whose FQID is 0x400
- 0xC0A81406, the frame is placed on the queue whose FQID is 0x600
- 0xC0A81408, the frame is placed on the queue whose FQID is 0x800

Otherwise, the 'action' element passes the frame to the 'unknown_dist' distribution for handling.

```

description="Course Classification configuration">
<policy name="policy_0">
  <dist_order>
    <distributionref name="udp_dist"/>
    <distributionref name="non_udp_dist"/>
  </dist_order>
</policy>

<distribution name="udp_dist">
  <protocols>
    <protocolref name="udp"/>
  </protocols>
  <action type="classified" name="udp_classif"/>
</distribution>

<classification name="udp_classif">
  <key>
    <fieldref name="ipv4.dst">
  </key>
  <entry>
    <data>0xC0A81402</data>
    <queue base="0x200"/>
  </entry>
  <entry>
    <data>0xC0A81404</data>
    <queue base="0x400"/>
  </entry>
  <entry>
    <data>0xC0A81406</data>
    <queue base="0x600"/>
  </entry>
  <entry>
    <data>0xC0A81408</data>
    <queue base="0x800"/>
  </entry>
  <action type="distribution" condition="on-miss" name="unknown_dist"/>
</classification>
"cc_policy.xml" 108 lines --61%--

```

5.2.4.5.1.11.4 Policer Section

The Policer section of the Policy file is optional.

If used, the section consists of up to 256 policer profiles. Each profile starts with a 'policer' element, which is a container for various child elements with which you implement a particular policing behavior.

Each profile works in one of these modes:

- Pass-through – Policer performs no traffic metering
- RFC-2698 - Policer employs a two-rate, three-color marker scheme
- RFC-4115 - Policer employs a differentiated service, two-rate, three-color marker scheme that efficiently handles in-profile traffic

Each of these modes can be configured to be color-aware or color-blind.

For RFC-2698 and RFC-4115 modes, you must specify these values:

- unit, the unit to be used for the following numeric parameters. Valid values for unit are "packet" and "byte."
- CIR, Committed Information Rate^[5]
- CBS, Committed Burst Size^[6]
- PIR, Peak Information Rate^[5]
- PBS, Peak Burst Size^[6]

In all three modes, you can specify the next invoked action (NIA) for each color result (drop the frame, proceed to the specified distribution, etc.)

Example 1 - Policer Markup for RFC2698 Mode

```
<policer name="policer2">
  <algorithm>rfc2698</algorithm>

  <color_mode>color_aware</color_mode>

  <CIR>12000</CIR>
  <EIR>34000</EIR>
  <CBS>56000</CBS>
  <EBS>78000</EBS>

  <unit>byte</unit>

  <action condition="on-green" type="distribution" name="green_dist"/>
  <action condition="on-yellow" type="distribution" name="yellow_dist"/>
  <action condition="on-red" type="drop"/>
</policer>
```

Example 2 - Policer Markup for Pass-through Mode

```
<policer name="vlan_congestion_control_green">
  <algorithm>pass_through</algorithm>

  <color_mode>color_blind</color_mode>

  <default_color>green</default_color>

  <action condition="on-green" type="distribution name="default_dist"/>
</policer>

<policer name="vlan_congestion_control_yellow">
  <algorithm>pass_through</algorithm>

  <color_mode>color_blind</color_mode>

  <default_color>yellow</default_color>

  <action condition="on-yellow" type="drop"/>
</policer>

<policer name="vlan_congestion_control_red">
  <algorithm>pass_through</algorithm>
```

[5] If "unit" attribute is "packet" specify CIR and PIR in packets/second. If "unit" attribute is "byte" specify CIR and PIR in Kbits/second

[6] If "unit" attribute is "packet" specify CBS and PBS in packets. If "unit" attribute is "byte" specify CBS and PBS in bytes.

```

<color_mode>color_blind</color_mode>

<default_color>red</default_color>

<action condition="on-red" type="drop"/>
</policer>

```

5.2.4.5.12 Configuration File

The Configuration file contains markup that defines the FMan instances (for devices with more than one FMan) and ports that are being used.

In addition, the Configuration file "connects" each port to the parse, classification, policing, and distribution rules defined in the Policy file. How? Each 'port' element in the Configuration file has a 'policy' attribute whose value must be the name of one of the 'policy' elements in the Policy file. This information tells the FMan which distributions to compare to each frame received on a given port.

[Figure 74. Example Configuration File](#) on page 273 shows the Configuration file's elements, attributes, and element hierarchy.

Note these element and attribute requirements:

- Valid engine names are "fm0" or "fm1"
- Valid values for the port type attribute are:
 - "MAC" (1/10 Gbps Ethernet port)
- Port numbering corresponds to hardware port number (as in dts) for each port.
- The value of the 'policy' attribute of a 'port' element must match the name of a 'policy' element in the Policy file.
- portid attribute (optional) - One byte numeric value that is attached to the port and that can be used in the 'distribution' and 'combine' elements of the Policy file.

The Configuration file's general structure is shown below.

[Figure 74. Example Configuration File](#) on page 273 shows an example configuration file. It uses the optional 'portid' attribute for the 1 Gbps ports.

Figure 74. Example Configuration File

```

<cfgdata>
  <config>
    <engine name="fm0">
      <port type="MAC" number="1" policy="ipv4_policy"/>
      <port type="MAC" number="2" policy="ipv4_policy" portid="0x96"/>
      <port type="MAC" number="3" policy="ipv4_policy" portid="0x97"/>
      <port type="MAC" number="4" policy="ipv4_policy" portid="0x97"/>
    </engine>
  </config>
</cfgdata>

```

Virtual Storage Profiles

The element 'vsp' (Virtual Storage Profile) is implemented in FMC as a standalone entity or can be defined directly in the element that uses it. The element 'vsp' can be used inside distributions, classification and entries (both classification and replicator). When used directly in the 'classification' element (not in 'entry') it counts for the on-miss action. If the 'action' of the 'entry' or on-miss goes to another 'classification' or 'replicator' the 'vsp' is ignored.

Table 22. 'fragmentation' Element Attributes:

Attribute	Requirement	Description
name	required	Name of the element. The name is used to refer the virtual storage profile inside the elements that are using it.
type	optional	The type of the VSP. Values: <ul style="list-style-type: none"> • direct – (default) the relative profile ID is selected directly by the 'base' attribute. • indirect – the relative profile ID is selected base on the attributes fqshift, vspoffset, and vspcount can be used only in distribution.
base	required for direct.	--
fqshift	required for indirect.	Shift of KeyGen results without the FQID base.
vspoffset	optional for indirect	OR of KeyGen results without the FQID base; should indicate the storage profile offset within the port's storage profiles window.
vspcount	optional for indirect	Range of profiles starting at base.

VSP examples (standalone, defined in element, direct/indirect): The action targets of the entry are restricted to:

```
<vsp name = "storage01" base = "6"/>
<vsp name = "storage02" type = "indirect" fqshift="2" vspoffset="3" vspcount="8"/>
<vsp name = "storage03" type = "direct" base = "7"/>
```

Usage:

```
...
<entry>
  <queue base="0x220"/>
  <vsp name="storage01">
</entry>
...
<distribution name="dist1">
  ...
  <queue count="8" base="0x230"/>
  <vsp type="indirect" fqshift="2" vspoffset="0" vspcount="4"/>
  ...
</distribution>
...
<classification name="eth_dest_clsif">
  <key>
    <fieldref name="ethernet.dst"/>
  </key>
  ...
  <vsp name="storage03">
    <action condition="on-miss" type="distribution" name="garbage"/>
</classification>
```

5.2.4.5.1.13 NXP NetPDL Reference

The FMan's Soft Parser can process non-standard, custom protocols that you define. To define a custom protocol, you enter NetPDL (Network Protocol Description Language) markup into a file called the Custom Protocol file. This markup defines each field in the custom protocol's header, as well as actions for the Soft Parser to take both before and after the custom header is loaded into the frame window.

Note: Although the markup used to define a custom protocol is based on NetPDL, this markup does not follow NetPDL rules strictly. As a result, you cannot rely on non-NXP documentation of NetPDL as you write your Custom Protocol file. Only the information in this appendix accurately explains how to write the NetPDL that goes in a Custom Protocol file.

You pass the name of the Custom Protocol file to the FMC Tool from the command line. The tool, in turn, passes the information in this file (directly or indirectly) to the FMan's Soft Parser.

5.2.4.5.1.13.1 Basic XML Rules

The Custom Protocol XML file follows standard XML rules.

The file is composed of several elements. Each element begins with a start tag and can contain attributes and/or child elements. If the element contains child elements, it must have a matching end tag. An element without child elements or text must end with a forward slash (/).

Note that element and attribute names are case sensitive. In the Custom Protocol file, all element and attribute names use only lower case alphabets.

Comments always begin with "<!--" and end with "-->"

Example

```
<one-element attribute1="value"> <!-- this is a comment -->
  <child-element myattribute="4"/>
</one-element>
<another-element attribute2="value2"/>
```

5.2.4.5.1.13.2 The netpdl Element

The Custom Protocol file always begins with the <netpdl> root element. As a result, the end netpdl tag must appear at the end of the file.

Attributes: No required attributes

Child Elements: protocol

Example

```
<netpdl>
...
</netpdl>
```

5.2.4.5.1.13.3 The protocol element

Use the 'protocol' element to bracket the definition of each custom protocol in the Custom Protocol file. The 'protocol' element is a container for all the other elements required to define a custom protocol.

Attributes

name - (required) alphanumeric string; defines the unique name of the custom protocol.

longname - (optional) alphanumeric string; provides a user-friendly name for the protocol.

prevproto - (required) alphanumeric string. This attribute defines the previous protocol, that is, the protocol whose header precedes the custom protocol's header.

[Table 23. Valid values for the prevproto attribute](#) on page 276 lists the values that you can assign to the 'prevproto' attribute.

Table 23. Valid values for the prevproto attribute

Protocol	Layer
ethernet	2
llc_snap	2
vlan	2
pppoe	2
mpls	2
ipv4	3
ipv6	3
gre	3
minencap	3
otherl3 ^[7]	3
tcp	4
udp	4
ipsec_ah	4
ipsec_esp	4
sctp	4
dccp	4
otherl4 ¹	4

Each time the frame window contains a header for a protocol specified in the 'prevproto' attribute of one of the 'protocol' elements in the Custom Protocol file, the Hard Parser transfers control to the Soft Parser.

The Soft Parser then executes the 'before' element code of the 'protocol' element whose prevproto attribute matches the current protocol. As long as the 'before' element code is executing, the previous protocol's header remains in the frame window. As a result, the 'before' element code can reference the fields in the previous protocol header.

Typically, the 'before' element includes code that determines whether the next protocol header is an instance of the custom protocol defined by this protocol element. If it is not, the 'before' code instructs the Soft Parser to return to the Hard Parser; if it is, the Soft Parser continues to execute the 'before' code.

[7] The Custom Protocol file's NetPDL XML has a somewhat different structure and behavior if either 'otherl3' or 'otherl4' is the previous protocol. See [Effect of Setting prevproto Attribute to otherl3 or otherl4](#) on page 277.

When the Soft Parser finishes executing the 'before' code (and if it does not return control to the Hard Parser), the Soft Parser advances the frame window to the custom protocol header and starts executing the 'after' element code (if any has been defined). Therefore, the code in the 'after' element can reference the fields in the custom protocol header.

Child Elements: format, execute-code

Example

```
<protocol name="gtpu" longname="GTP-U" prevproto="udp">
  ...
</protocol>

<protocol name="tcpExt" longname="tcp extension" prevproto="cp">
  ...
</protocol>
```

5.2.4.5.1.13.3.1 Effect of Setting prevproto Attribute to otherl3 or otherl4

When the 'prevproto' attribute of the 'protocol' element is set to otherl3 (for other layer 3 protocol) or otherl4 (for other layer 4 protocol), the first byte of the previous protocol header and the first byte of the custom protocol header are at the position in the frame window. Because they are not real protocols, neither otherl3 nor otherl4 has a real protocol header with a defined size and defined fields; these "protocols" are used just to provide the Soft Parser with an entry point (or a termination point) within the frame window. In effect, the size of the otherl3 and otherl4 "headers" is zero. Consequently, these "headers" have the same start offset in the frame window as does the custom protocol's header.

Note: Because the otherl3 and otherl4 protocols do not have real headers, they provide nothing for the Soft Parser to parse. As a result, you cannot use the 'before' element when either of these protocols is assigned to the 'prevproto' attribute. You can only use the 'after' element in these cases.

5.2.4.5.1.13.4 The format element

Use the 'format' element to bracket the definition of the structure of a custom protocol header. The 'format' element is a container for the 'fields' element which, in turn, is a container for the 'field' element. The 'field' element lets you define each field in a custom protocol's header.

Attributes: none

Child Elements: fields

5.2.4.5.1.13.4.1 The fields Element

Use the 'fields' element to define the structure of a custom protocol's header. This element is a container for the 'field' element, which lets you define each field in a custom protocol header.

Attributes: none

Child Elements: field

5.2.4.5.1.13.4.2 The field Element

Use the 'field' element to define one of the fields in a custom protocol header.

Attributes

type - (required) string; Defines the field size as either "fixed" for a byte-length field or "bit" for a bit-length field.

size - (required) integer; Defines the size of the field in bytes.

name - (required) string; Defines the unique name for the field.

longname - (optional) string; Defines the name of the field for display purposes.

mask - (required only for bit field) integer; Defines the specific bits in the current bytes which belong to this field.

The field elements appear one after the other to define a custom protocol's header frame. The first field begins in the first byte of the custom protocol's frame header and its size is determined by the size attribute. The following fields conform to the following rules:

- A fixed field or a field following a fixed field begins in the next byte, which is the previous field's offset + the previous field's size.
- A bit field following a bit field begins in the next byte only if the last bit in the previous field's mask is 1.
- If two fields share the same offset (which is possible only when both fields are bit fields and the mask of the first field does not end with 1), they should have the same value for their size attributes.

Example

```
<format>
  <fields>
    <field type="bit"   name="flags"   mask="0xE0" size="1"/>
    <field type="bit"   name="pt"     mask="0x80" size="1"/>
    <field type="bit"   name="version" mask="0x07" size="1"/>
    <field type="fixed" name="mtype"   size="1"/>
    <field type="fixed" name="length"  size="2"/>
  </fields>
</format>

<format>
  <fields>
    <field type="bit"   name="version" mask="0xE0" size="1"/>
    <field type="bit"   name="pt"     mask="0x10" size="1"/>
    <field type="bit"   name="flags"   mask="0x07" size="1"/>
    <field type="bit"   name="flags1"  mask="0x01" size="1"/>
    <field type="bit"   name="flags2"  mask="0x10" size="1"/>
    <field type="bit"   name="flags3"  mask="0x02" size="1"/>
    <field type="fixed" name="mtype"   size="1" longname="message type"/>
    <field type="fixed" name="length"  size="2"/>
  </fields>
</format>
```

The fields will, thus, be stored in the following bit offsets in the custom protocol header:

version: 0-2 pt: 3-3 flags: 5-7 flags1: 15-15 flags2: 19-19 flags3: 22-22 mtype: 24-31 length: 32-47

5.2.4.5.1.13.5 *The execute-code element*

Use the 'execute-code' element to define all code that should be executed for a custom protocol once the parser reaches the specified previous protocol header.

This element contains two child elements, 'before' and 'after'. At least one of these child elements must be defined. If both are defined, the 'before' element must appear before the 'after' element.

Attributes: none

Child Elements: before, after

Example

```
<execute-code>
  <before>
    ...
  </before>
```

```
<after headersize="8">
  </after>
</execute-code>
```

5.2.4.5.1.13.5.1 The before Element

The Soft Parser executes the code in the 'before' element before it moves the frame window from the previous protocol header to the custom protocol header. Therefore, use the 'before' element to specify logic that requires access to fields in the previous protocol header. This code is often used to determine whether the next protocol header is an instance of the custom protocol this protocol block defines. If it is not, the 'before' block instructs the Soft Parser to return control to the Hard Parser; if it is, the Soft Parser continues processing.

While the code in the 'before' element is analyzed, the frame window points to the previous protocol header. Therefore, the frame window variable (\$FW) references the fields in the previous protocol header and the header size variable (\$headerSize) variable returns the size of the previous protocol's header.

Once it reaches the end of the 'before' element, the Soft Parser moves the frame window to the custom protocol header. If no 'after' element has been defined, the Soft Parser then returns to the Hard Parser.

The 'before' element can only appear once in the 'execute-code' element and, if an 'after' element has been defined, the 'before' element must appear before the 'after' element.

Attributes

confirm - (optional) string; Valid values are "yes" and "no". The default value is "no" if an 'after' element has been defined. Otherwise, the default value is "yes". If confirm="yes", the Soft Parser confirms the presence of the 'prevproto' header by bitwise OR'ing the previous protocol's line-up enable confirmation mask with the current line-up confirmation vector (LCV) value.

confirmcustom - (optional) string; Valid values are "shim1", "shim2", and "no". The default value is "no". If 'confirmcustom' is set (!="no"), the Soft Parser confirms the presence of the custom protocol header by bitwise OR'ing the custom protocol's mask with the current line-up confirmation vector (LCV) value. The custom protocol can set one of the last two bits in the LCV. If "shim1" is selected, the least significant bit is set; if "shim2" is selected, the second least significant bit is set.

Child Elements: if, switch, assign, action

Note: When the previous protocol is 'otherl3' or 'otherl4', the previous protocol and the custom protocol are treated as if they are the same and each begins at the same offset within the frame window. Therefore, the 'before' element cannot be used when the 'prevproto' attribute is 'otherl3' or 'otherl4'; only an 'after' element be used when the 'prevproto' attribute is 'otherl3' or 'otherl4'. See [Effect of Setting prevproto Attribute to otherl3 or otherl4](#) on page 277 for more information.

5.2.4.5.1.13.5.2 The after Element

The 'after' element contains code which should be executed when a frame from the current custom protocol has been encountered. In contrast to the 'before' element, in the 'after' section, it is possible to access fields from the current protocol but not from the previous protocol. In the 'after' element the frame window variable (\$FW) manipulates the current custom protocol header and the header size variable (\$headerSize) returns the size of the current custom protocol header.

At the end of the 'after' element, the frame window jumps to the end of the custom protocol's header and control returns to the Hard Parser.

The 'after' element can appear only once in an 'execute-code' element and if a 'before' element has been defined, it must appear before the 'after' element.

Attributes

confirm - (optional) string; Valid values are "yes" and "no". The default value is "yes". If confirm="yes", the Soft Parser confirms the existence of the previous protocol header by bitwise OR'ing the previous protocol's line-up enable confirmation mask with the current line-up confirmation vector (LCV) value.

confirmcustom - (optional) string; Valid values are "shim1", "shim2", and "no". The default value is "no". If 'confirmcustom' is set (!="no"), the Soft Parser confirms the presence of the custom protocol header by bitwise OR'ing the custom protocol's

mask with the current line-up confirmation vector (LCV) value. The custom protocol can set one of the two last bits in the LCV. If "shim1" is selected, the least significant bit is set; if "shim2" is selected, the second least significant bit is set.

headerSize - (optional) integer; Possible values: arithmetic expression. (See [Arithmetic Expressions](#) on page 295) The default value is calculated using the fields contained by the 'format' element. You can specify the custom protocol's header size with this attribute. This information is needed so the parser returns to the right position following the custom protocol header. If header size is not specified, the FMC Tool assumes that the fields defined inside the 'format' element are the only fields in the custom protocol header and calculates the header size using these fields. The \$headerSize variable in the 'after' element returns the value defined in this attribute (or the value calculated by default if the header attribute is not defined).

Child Elements: if, switch, assign, action

Example

```
<protocol name="gtp" prevproto="udp">
  <format>
    <fields>
      <field type="bit" name="version" mask="0xE0" size="1"/>
    </fields>
  </format>

  <execute-code>
    <before confirm="no">
      <assign-variable name="$GPR1" value="udp.dport"/>
      <!-- Note that this is ILLEGAL: <assign-variable name="GPR1" value="version" -->
      <assign-variable name="$shimr" value="$headerSize"/>
      <!-- shimresult now holds udp's header size -->
    </before>

    <after headersize="4" confirmcustom="shim1">
      <!-- Note that this is ILLEGAL: <assign-variable name="$GPR1" value="udp.dport"> -->
      <assign-variable name="$GPR1" value="version"/>
      <assign-variable name="$shimr" value="$headerSize"/>
      <!-- shimresult now equals 4 -->
    </after>
  </execute-code>
</protocol>
```

5.2.4.5.1.13.5.3 Child Elements of the before and after Elements

5.2.4.5.1.13.5.3.1 The assign-variable Element

The 'assign-variable' element assigns an expression to a variable.

Attributes

name - (required) string; The name of the variable to which a value will be assigned. Valid values: Variables contained in the result array.

value - (required) integer; The expression assigned to the variable. Valid values: arithmetic expressions.

Child Elements: none

Example

```
<assign-variable name="$shimoffset_2" value="$shimoffset_1+12"/>
```

5.2.4.5.1.13.5.3.2 The if Element

This element tests the specified condition. If the condition is true, control transfers to the 'if-true' element; if the condition is false, control transfers to the 'if-false' element (if one is defined).

Attributes

expr - (required) string; Defines the condition to be checked before selecting the code block to execute. Valid values: logical expressions. (See [Logical Expressions](#) on page 295 for more information.)

Child Elements: if-true (required), if-false

Example

```
<if expr="$shimoffset_1==1">
  <if-true>
    ...
  </if-true>
  <if-false>
    ...
  </if-false>
</if>
```

5.2.4.5.1.13.5.3.2.1 The if-true Element

This element defines code to execute if the expression defined in the parent 'if' element is true.

Attributes: none

Child Elements: if, switch, assign, action (the same child elements as for the 'before' and 'after' elements)

Example

```
<if expr="$shimoffset_1==1">
  <if-true>
    ...
  </if-true>
  <if-false>
    ...
  </if-false>
</if>
```

5.2.4.5.1.13.5.3.2.2 The if-false Element

This element defines the code to execute if the expression defined in the parent 'if' element is false.

Attributes: none

Child Elements: if, switch, assign, action (the same child elements as for the 'before' and 'after' elements)

Example

```
<if expr="$shimoffset_1==1">
  <if-true>
    ...
  </if-true>
  <if-false>
    ...
  </if-false>
</if>
```

5.2.4.5.1.13.5.3.3 The switch Element

This element defines an expression and a set of cases. Each case consists of a value (or set of values) and code to be executed if the value equals the switch expression. Each 'switch' element must have at least one 'case' child element.

Note: Only the code of the first case that matches the switch expression is executed. Any following cases are skipped. In C language terms, a break is automatically added after the code of each case.

Attributes

expr - (required) string; Defines the value being checked. Valid values: arithmetic expressions.

Child Elements: case, default

Example

```
<switch expr="$shimoffset_1+1">
  <case value="2">
    <assign-variable name="$GPR[1:1]" value="0"/>
  </case>

  <case value="3" maxvalue="4">
    <assign-variable name="$GPR[1:1]" value="1"/>
  </case>

  <default>
    <assign-variable name="$GPR[1:1]" value="2"/>
  </default>
</switch>
```

5.2.4.5.1.13.5.3.3.1 The case Element

This element matches a value or range of values against the switch expression.

Attributes

value - (required) integer; If the value equals the switch expression and no earlier case has been matched, the code in the 'case' element is executed.

maxvalue - (optional) integer; If the switch expression is greater than or equal to the 'value' attribute and the expression is less than or equal to the 'maxvalue' attribute (and no earlier case has been matched), the code in the 'case' element is executed.

Child Elements: if, switch, assign, action (the same child elements as for the 'before' and 'after' elements)

Example

```
<switch expr="$shimoffset_1+1">
  <case value="2">
    <assign-variable name="$GPR[1:1]" value="0"/>
  </case>

  <case value="3" maxvalue="4">
    <assign-variable name="$GPR[1:1]" value="1"/>
  </case>

  <default>
    <assign-variable name="$GPR[1:1]" value="2"/>
  </default>
</switch>
```

5.2.4.5.1.13.5.3.3.2 The default Element

The 'default' element contains code that is executed if the expression in the 'switch' element is not matched by any of the candidate cases.

Attributes: none

Child Elements: if, switch, assign, action (the same child elements as for the 'before' and 'after' elements)

Example

```
<switch expr="$shimoffset_1+1">
  <case value="2">
    <assign-variable name="$GPR[1:1]" value="0"/>
  </case>

  <case value="3" maxvalue="4">
    <assign-variable name="$GPR[1:1]" value="1"/>
  </case>

  <default>
    <assign-variable name="$GPR[1:1]" value="2"/>
  </default>
</switch>
```

5.2.4.5.1.13.5.3.4 The action Element (for use in a Custom Protocol file)

Use the 'action' element in a 'before' or 'after' block to terminate soft parsing, jump to the specified next protocol header, and continue hard parsing.

Note: This topic defines the 'action' element used in a Custom Protocol file. See [The action element \(for use in a policy file\)](#) on page 306 for the definition of the 'action' element used in a Policy file.

Attributes

- type - (required) string; "exit" is the only valid value for the type attribute.
- advance - (optional) string; The 'advance' attribute controls whether the Soft Parser moves the frame window to the next frame header. This attribute has different meanings in the 'before' and 'after' elements. In the 'before' element, the Soft Parser moves the frame window from the previous protocol header to the custom protocol header. In the 'after' element, the Soft Parser moves the frame window from the custom protocol header to the specified next protocol header. The frame window is advanced according to the header size. The value of 'advance' must be 'yes' or 'no'. The default is 'yes' unless 'nextproto' is set to 'end_parse', 'return', or not set at all. In these cases, the default value is 'no'.
- confirm - (optional) string; If confirm="yes", the Soft Parser bitwise OR's the previous protocol's line-up enable confirmation mask with the current line-up confirmation vector (LCV) value. Valid values are "yes" and "no"; the default value is "yes".
- confirmcustom - (optional) string; Valid values are "shim1", "shim2", or "no". The default value is "no". If confirmcustom is set to a value other than "no", the Soft Parser bitwise ORs the custom protocol's mask with the current line-up confirmation vector (LCV) value. The custom protocol can set one of the two last bits in the LCV. If shim1 is specified, the least significant bit is set; if shim2 is specified, the second least significant bit is set.
- nextproto - (optional); If used, this attribute must be one of the values in [Table 24. Parse Action for each Value of the nextproto Attribute](#) on page 284. The default value is 'return'.

Table 24. Parse Action for each Value of the nextproto Attribute

If nextproto is ...	The parse action is ...
ethernet	Jump to the Ethernet header and continue hard parsing
llc_snap	Jump to the LLC_SNAP header and continue hard parsing
vlan	Jump to the VLAN header and continue hard parsing
pppoe	Jump to the PPPoE header and continue hard parsing
mpls	Jump to the MPLS header and continue hard parsing
ipv4	Jump to the IPv4 header and continue hard parsing
ipv6	Jump to the IPV6 header and continue hard parsing
gre	Jump to the GRE header and continue hard parsing
minencap	Jump to the MinEncap header and continue hard parsing
otherl3	Jump to the otherl3 header and continue hard parsing
tcp	Jump to the TCP header and continue hard parsing
udp	Jump to the UDP header and continue hard parsing
ipsec_ah	Jump to the IPsec_ah header and continue hard parsing
ipsec_esp	Jump to the IPsec_esp header and continue hard parsing
sctp	Jump to the SCTP header and continue hard parsing
dccp	Jump to the DCCP header and continue hard parsing
otherl4	Jump to the otherl4 header and continue hard parsing
after_ethernet	<p>Jump to the protocol that should follow the Ethernet header. The next protocol is determined from the value of the \$nxtHdr variable. See Table 25. Next Protocol for each \$nxtHdr Value if nextproto is 'after_ethernet' on page 285 to find the next protocol for each possible value of \$nxtHdr.</p> <p>Note:The 'advance' attribute must be set to 'yes' if 'nextproto' is set to 'after_ethernet'.</p>
after_ip	<p>Jump to the protocol that should follow the IP header. The next protocol is determined from the value of the \$nxtHdr variable. See Table 25. Next Protocol for each \$nxtHdr Value if nextproto is 'after_ethernet' on page 285 to find the next protocol for each possible value of \$nxtHdr.</p> <p>Note:The 'advance' attribute must be set to 'yes' if 'nextproto' is set to 'after_ip'.</p>
<i>Table continues on the next page...</i>	

Table 24. Parse Action for each Value of the nextproto Attribute (continued)

If nextproto is ...	The parse action is ...
return (default value)	Return to the Hard Parser without advancing the frame window. In this case, the Hard Parser starts parsing the frame header at the same position at which the Soft Parser began. The 'advance' attribute cannot be 'yes' when 'nextproto is set to return.
none/end_parse	Finish parsing the frame header; do not return to the Hard Parser.

Table 25. Next Protocol for each \$nxtHdr Value if nextproto is 'after_ethernet'

If \$nxtHdr is ...	The next protocol is ...
0x05DC or less	llc_snap
0x0800	ipv4
0x86DD	ipv6
0x8847, 0x8848	mpls
0x8100, 0x88A8, ConfigTPID1, ConfigTPID2	vlan
0x8864	pppoe
other value	otherI3

Table 26. Next Protocol for each \$nxtHdr Value if nextproto is 'after_ip'

If \$nxtHdr is ...	The next protocol is ...
4	ipv4
6	tcp
17	udp
33	dccp
41	ipv6
50, 51	ipsec
47	gre
55	minencap
132	sctp
other value	otherI4

Notes

- The frame window *must* be advanced when parsing jumps to the 'after_ethernet' or 'after_ip' protocols. Therefore, the 'advance' attribute cannot be set to 'no' in these cases.
- The frame window must *not* be advanced before a 'return' to the Hard Parser. Therefore, the 'advance' attribute cannot be set to 'yes' if nextproto is set to 'return' or not set at all (since 'return' is the default 'nextproto' value).

Child Elements: none

Example

```
<action type="exit"
  advance="yes"
  confirmcustom="shim2"
  confirm="no"
  nextproto="udp" />
```

5.2.4.5.1.13.6 Expressions

Expressions are constructed of operands and operators. The simplest expression can contain just one operand. Most operators are dyadic and separate two operands (such as +, -) and some operators are monadic and operate on just the operand that follows them (such as 'not').

5.2.4.5.1.13.6.1 Operands

These are the supported types of operands: numbers, variables, fields, and expressions.

Note: The maximum size of an operand is 64 bits (8 bytes).

5.2.4.5.1.13.6.1.1 Numbers

Numbers can appear in decimal (no prefix), binary (prefixed by '0b'), or hexadecimal (prefixed by '0x') format.

All numbers are 64-bit unsigned integers. However, some operators only use the 32 LSB of a number.

Note: Immediate, primitive negative numbers are not supported. For example, the number -2 cannot appear in an expression. However, artificial negative values can be created using arithmetic expressions such as 1-3 (which returns 0xffffffff).

5.2.4.5.1.13.6.1.2 Fields

Fields are defined with the 'format' element in a custom protocol header definition. There are two ways to access a field, by typing their name directly or by typing the name of the protocol header containing the field, followed by a period, followed by the name of the field.

In the 'before' element, it is only possible to access fields in the previous protocol header; in the 'after' element, it is only possible to access fields in the current custom protocol header.

Note: Fields longer than 8 bytes cannot be accessed individually. You can work around this limit by accessing the frame directly using the frame window (\$FW) variable or by splitting the field into several shorter fields.

Example

```
<protocol name="gptu" prevproto="#ethernet">
  <format>
    <fields>
      <field type="fixed" name="example" size="2"/>
    </fields>
  </format>

  <execute-code>
    <before>
      <assign-variable name="$l2r" value="ethernet.type"/>
    </before>
```

```

<after>
  <assign-variable name="$shimoffset_2" value="example"/>
</after>
</execute-code>
</protocol>

```

5.2.4.5.1.13.6.1.3 Variables

All variable names begin with the \$ prefix and are case-sensitive. These variables are supported: frame window, header size, prevprotoOffset, parameter array, and result array variables.

5.2.4.5.1.13.6.1.3.1 Result Array Variables

Result array variables return values contained in the parse results array.

Syntax for accessing result array variables:

- \$variableName - returns the entire variable
- \$variableName[byteOffset:byteNumber] - Returns the byteNumber number of bytes in the variable starting from byteOffset. This access method is useful for accessing a subset of the bytes in the variable. In bytesNumber equals zero, the entire variable is returned, starting from byteOffset.

Example: The variable \$actiondescriptor returns result array bytes 64-71. The expression \$actiondescriptor[2:4] returns result array bytes 66-69 since 66 is at offset 2 of the actiondescriptor variable and the requested size is 4. The expression \$actiondescriptor[3:0] returns result array bytes 67-71 since 67 is at offset 3 of the actiondescriptor variable and the requested size is 0, which means return the entire variable starting at the specified offset (3).

Other usage: In addition to expressions, result array variables can be used in the left side of 'assign-variable' elements to modify result array values.

[Table 27. Result Array Variables](#) on page 287 shows the available result array variables .

Table 27. Result Array Variables

Variable Name	Result Array Bytes Referenced
gpr1	0-7
gpr2	8-15
logicalportid	16-16
shimr	17-17
l2r	18-19
l3r	20-21
l4r	22-22
classificationplanid	23-23
nxthdr	24-25
runningsum	26-27
flags	28-28

Table continues on the next page...

Table 27. Result Array Variables (continued)

Variable Name	Result Array Bytes Referenced
fragoffset	28-29
routtype	30-30
rhp	31-31
ipvalid	31-31
shimoffset_1	32-32
shimoffset_2	33-33
ip_pidoffset	34-34
ethoffset	35-35
llcs_napoffset	36-36
vlandcioffset_1	37-37
vlandcioffset_n	38-38
lastetypeoffset	39-39
pppoeoffset	40-40
mplsoffset_1	41-41
mplsoffset_n	42-42
ipoffset_1	43-43
ipoffset_n	44-44
minencapo	44-44
minencapoffset	44-44
greoffset	45-45
l4offset	46-46
nxthdroffset	47-47
framedescriptor1	48-55
framedescriptor2	56-63
actiondescriptor	64-71
<i>Table continues on the next page...</i>	

Table 27. Result Array Variables (continued)

Variable Name	Result Array Bytes Referenced
ccbbase	72-75
ks	76-76
hpnia	77-79
sperc	80-80
ipver	85-85
iplength	86-87
icp	90-91
attr	92-92
nia	93-95
ipv4sa	96-99
ipv4da	100-103
ipv6sa1	96-103
ipv6sa2	104-111
ipv6da1	112-119
ipv6da2	120-127

Note: The \$GPR2 variable is used internally by the FMC Tool to calculate complex expressions, including checksum calculations. Using \$GPR2 for other purposes is possible, but is not supported or recommended.

5.2.4.5.1.13.6.1.3.2 Parameter Array Variable

This variable returns data from the parameter array. Because the parameter array is more than 8 bytes long, you must specify the particular bytes needed.

Accessing parameter array variables: \$PA[byteOffset:byteNumber] - returns the byteNumber number of bytes in the parameter array starting at byteOffset.

Example: The expression "\$PA[4:2]" accesses the fifth and sixth bytes (indexed at PA[4] and PA[5]) of the parameter array.

5.2.4.5.1.13.6.1.3.3 Header Size Variables

Header size variables return the header size or default header size of a protocol header.

Accessing header size variables: \$headerSize or \$defaultHeaderSize

- In the 'before' element, the \$headerSize of the previous protocol header is returned. Accessing \$defaultHeaderSize is not allowed.

- In the 'after' element, the \$defaultHeaderSize variable returns the number of bytes in the custom protocol's format fields. The \$headerSize variable returns the headerSize as defined by the 'headersize' attribute of the 'after' element. If the user has not specified a value for the 'headersize' attribute, \$headerSize returns the same value as \$defaultHeaderSize.

5.2.4.5.1.13.6.1.3.4 Frame Window Variable

The frame window variable (\$FW) returns data from the frame array. In the 'before' element, the frame window variable returns data from the previous protocol's header. In the 'after' element, the frame window variable returns data from the custom protocol header.

Using the frame window variable: \$variableName[bitOffset:bitNumber] - Returns the bitNumber number of bits in the frame header starting from bitOffset.

Note: The frame window uses similar syntax to the parameter array and result array variables; however, the frame window variable accesses bits instead of bytes.

Examples

To access the tenth and eleventh bits in the frame array (indexed at FW[9], FW[10]), use "\$FW[9:2]".

To access the entire third byte of the frame array, use "\$FW[16:8]".

The conditions in the example below are always true because the same bits can be accessed using either the \$FW variable or header field names.

```
<format>
  <fields>
    <field type="bit" name="first" size="1" mask="0xE0"/>
    <field type="bit" name="second" size="1" mask="0x1"/>
    <field type="bit" name="third" size="1" mask="0xF"/>
    <field type="fixed" name="fourth" size="2"/>
  </fields>
</format>
...
<after>
  <if expr="first==$FW[0:3]"> ... </if>
  <if expr="second==$FW[7:1]"> ... </if>
  <if expr="third==$FW[8:4]"> ... </if>
  <if expr="fourth==$FW[16:16]"> ... </if>
</after>
```

5.2.4.5.1.13.6.1.3.5 The prevprotoOffset Variable

This variable returns the offset of the previous protocol's frame header. This variable has the same value in the 'before' and 'after' sections and always refers to the protocol defined in the 'prevproto' attribute of the protocol element.

In the 'before' element, the frame window's current location is equal to prevprotoOffset. In the 'after' element, the frame window's current location is equal to prevprotoOffset+headerSize.

Note: This variable is actually a "shortcut" to the result array and returns or modifies values taken directly from this array.

Table 28. Previous Protocol RA Return Values

If the previous protocol is ...	The value returned from result array is ...
ethernet	\$ethoffset
gre	\$greoffset

Table continues on the next page...

Table 28. Previous Protocol RA Return Values (continued)

If the previous protocol is ...	The value returned from result array is ...
ipv4, ipv6	\$lpooffset_n
llc_snap	\$llcsnapoffset
minencap	\$minencapoffset
mpls	\$mplsoffset_n
pppoe	\$pppoeoffset
tcp, udp, sctp, dccp, ipsec_ah, ipsec_esp	\$l4offset
vlan	\$vlanoffset_n
otherl3, otherl4	\$NxtHdrOffset - When the previous protocol is otherl3 or other l4, the custom protocol and the previous protocol have the same offset. See Effect of Setting prevproto Attribute to otherl3 or otherl4 on page 277.

5.2.4.5.1.13.6.2 Operators

The parser supports many operators. These operators can receive arithmetic or logical operands and return an arithmetic or logical value. An arithmetic value is a number, while a logical value is true or false. (See [Arithmetic Expressions](#) on page 295 and [Logical Expressions](#) on page 295 for more information.)

[Table 29. Supported Operators and their Properties](#) on page 291 describes all operators and their associated properties. All dyadic operators (operators which receive two parameters) appear between two operands. All monadic operators appear before an operand.

Table 29. Supported Operators and their Properties

Name	Parameters	Description	Symbol
Greater than	Logical (Arithmetic, Arithmetic)	Checks if the value of the first expression is greater than the second	gt
Greater equal	Logical (Arithmetic, Arithmetic)	Checks if the value of the first expression is equal to or greater than the second	ge
Less than	Logical (Arithmetic, Arithmetic)	Checks if the value of the first expression is less than the second	lt
Less equal	Logical (Arithmetic, Arithmetic)	Checks if the value of the first expression is equal to or less than the second	le
Equal	Logical (Arithmetic, Arithmetic)	Checks if the two expressions are equal	==
Not equal	Logical (Arithmetic, Arithmetic)	Checks if the two expressions are not equal	!=

Table continues on the next page...

Table 29. Supported Operators and their Properties (continued)

Name	Parameters	Description	Symbol
Logical AND	Logical (Logical, Logical)	Checks if both expressions are true	and
Logical OR	Logical (Logical, Logical)	Checks if either one of the expressions is true	or
Logical NOT	Logical (Logical)	Returns true if the expression is false; returns false otherwise	not
Add	32-bit Arithmetic (32-bit Arithmetic, 32-bit arithmetic)	Return the sum of the expressions	+
Subtract	32-bit arithmetic (32-bit Arithmetic, 32-bit arithmetic)	Return the difference between the two expressions (result of subtraction)	-
Add carry	16-bit arithmetic (16-bit arithmetic, 16-bit arithmetic)	Return the sum of the two expressions summed with the carry after 32bit	addc
Bitwise OR	Arithmetic (Arithmetic, Arithmetic)	Returns the result of a bitwise OR operation on the two expressions	bitwor
Bitwise XOR	Arithmetic (Arithmetic, Arithmetic)	Returns the result of a bitwise XOR operation on the two expressions	bitwxor
Bitwise AND	Arithmetic (Arithmetic, Arithmetic)	Returns the result of a bitwise AND operation on the two expressions	bitwand
Bitwise NOT	Arithmetic (Arithmetic)	Returns the result of a bitwise NOT operation on the expression	bitwnot
Shift left	Arithmetic (Arithmetic, Integer - value up to 64 bits)	Return the left expression shifted left by the right expression	shl
Shift right	Arithmetic (Arithmetic, Integer - value up to 64 bits)	Return the left expression shifted right by the right expression	shr
Concat	Arithmetic (Arithmetic, Variable or Integer)	Special operator See The concat Operator on page 292 for full documentation	concat
Checksum	Arithmetic (Arithmetic - value up to 0xffff, Arithmetic - value up to 256, Arithmetic - value up to 256)	Special operator See The checksum Operator on page 293 for full documentation	checksum

5.2.4.5.1.13.6.2.1 The concat Operator

The concat operator shifts its first argument left and inserts its second argument to its right. The concat operation can be executed on variables or integers. If the second argument is a variable, the first argument is shifted left according to the known size of the variable. Result array variables have constant sizes and the size of the frame header's fields are set in the Custom Protocol file or the Standard Protocol file.

If the user accesses only specific bits in the second argument, the first argument is shifted left only by the number of bits specified.

If the second argument is an integer, the first argument is shifted left by the smallest word size into which the integer fits: 16, 32, 48, or 64.

Note: The second argument of a concat operation cannot be an expression because the FMC Tool does not know the size of an expression and therefore cannot shift the first argument properly. However, for expressions, you can replace the concat operation with a shift operation (as long as you know the number of bits to shift) and a bitwise OR operation.

Note: You should use concat instead of shift/bitwise OR when working with variables and integers in order to reduce code size.

For example, the following IF expression is true:

```
<assign-variable name="$shimr" value="2"/>
<assign-variable name="$GPR1[6:2]" value="3"/>
<if expr="1 concat $shimr concat $GPR1[6:2] concat 0x40000 == 0x102000300040000">
```

5.2.4.5.1.13.6.2.2 The checksum Operator

The checksum operator is a special operator with unique behavior and syntax. It appears before three operands that have parentheses around them. As a result, the concat operator looks like a function call - checksum(expression, integer, integer).

The first operand defines the initial checksum value. The second operand defines the frame window offset at which to start the checksum (relative to the current frame window location). The third operand defines the length of the data in bytes on which the checksum operation should be calculated.

Using these values, the checksum executes the add carry (addc) operation on 2-byte sized words in the frame window range specified. If the range specified contains an odd number of bytes to be checksummed, the last byte is padded on the right with zeros to form a 16-bit word for checksum purposes. The total sum is added to the initial checksum value using another addc operation. Therefore, the first argument that defined the initial sum value must be smaller than 0xffff. The result of the final addc operation is returned.

Note: Since it is only possible to access 256 bytes in the frame window, the last two arguments to the checksum operator must be less than or equal to 256.

Example

Suppose we have the following frame and the custom protocol header begins at offset 0xE (where 4500 appears):

```
FFFF FFFF FFFF 0CCB CC0D DDDD 0800 4500 002E 0000 4000 402F
2AA2 1000 0000 FFFE 0001 0308 0900 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 DA95 36D6 6F15 778C
```

The following IF conditions will always be true:

```
<after>
  <if expr="checksum(0x30A2,2,7+2)==0xDAFF">
    ...
  </if>

  <if expr="checksum(0,0,20)==0xFFFF">
    ...
  </if>
</after>
```

The first checksum operation above performs the following calculation:

```
0x30A2 + (0x002E add 0x0000 addc 0x4000 addc 0x402F addc 0x2A00)
```

The second checksum operation performs the following calculation:

```
0x0000 + (0x4500 addc 0x002E addc 0x0000 addc 0x4000 addc 0x402F addc 0x2AA2  
addc 0x1000 addc 0x0000 addc 0xFFFE addc 0x0001)
```

5.2.4.5.1.13.6.2.3 Expression Priorities

Expressions containing multiple operators perform the operation according to the following rules, in the order shown:

1. Operations in parentheses are performed
2. Operations that have a higher priority are performed
3. Multiple operations with the same priority are then executed from left to right

Note: Parentheses are recommended when several operators appear in the same expression to ensure correct calculation.

5.2.4.5.1.13.6.2.4 Operator Precedence

If several operators appear in the same expression (without separating parentheses), they are performed in the following order:

1. NOT, bitwise NOT, checksum
2. add, subtract, add carry
3. bitwise AND, bitwise OR, bitwise XOR
4. shift right, shift left, concat
5. greater than, greater equal, less than, less equal, equal, not equal
6. AND, OR

5.2.4.5.1.13.6.2.5 Variable Size

In most operations, expression size is limited to 64 bits. However, there are a few exceptions:

- When shifting variables, the shift value must be less than or equal to 64 bits since there are only 64 bits in an expression.
- The add carry operation can only be performed on 16-bit variables and always returns a 16-bit variable. The Soft Parser reports an error if an add carry operation is performed on a constant larger than 16 bits, but does not recognize a complex expression larger than 16 bits. Therefore, it is the responsibility of the user to perform the operation on 16-bit variables only.
- The subtract and add operators can only be performed on 32-bit variables and they always return a 32-bit result. If two 32-bit expressions are added and their result is larger than 32 bits, only the carry is returned, such that the returned value is a 32-bit variable. The Soft Parser reports a warning if an add carry operation is performed on a constant larger than 32 bits, but does not recognize a complex expression larger than 32 bits. Therefore, it is the responsibility of the user to perform the operation on 32-bit variables only.

For example, the following IF expressions are always true:

- ```
<if expr="0xFFFFFFFF+2==0x1">
```
- ```
<if expr="0x123456781+3==0x123456784">
```

The following IF expression is false (and should not be used):

- ```
<if expr="3+0x123456781==0x123456784">
```

#### 5.2.4.5.1.13.6.3 Expression Types

There are two main types of expressions: Logical expressions, which return "true" or "false", and arithmetic expressions, which return a numeric result.

##### 5.2.4.5.1.13.6.3.1 Logical Expressions

Logical expressions appear in the 'expr' attribute of the 'if' element.

These expressions always return "true" or "false" and, therefore, must use at least one logical operator that separates arithmetic and logical operators.

#### Examples

The following expressions are logical expressions:

- ```
(4+1==${shimoffset_1} or 5!=${shimoffset_2})
```
- ```
not(${shimoffset_2} ge ${shimoffset_1} or ${shimoffset_1} lt ${shimoffset_2})
```

The following expressions are NOT logical expressions:

- ```
(7 gt 3 and 2+7)
```
- ```
(5 lt 8 or 7)
```

##### 5.2.4.5.1.13.6.3.2 Arithmetic Expressions

Arithmetic expressions always have a numeric result. They can hold a single operand (a number, variable, or arithmetic expression) or more than one operand separated by arithmetic operators. Logical operators are not allowed in arithmetic expressions.

Arithmetic expressions can appear in the following places:

- The value attribute of the assign element
- The headersize attribute of the after element
- The expr attribute of the switch element

#### Examples

The following are arithmetic expressions:

- ```
(${FW[0:16]}+4)
```
- ```
(${shimoffset_1} concat 3)
```
- ```
(3+7+8+${shimoffset_2})
```
- ```
4
```

The following is NOT an arithmetic expression:

- ```
4==${shimoffset_2}
```

5.2.4.5.1.13.7 Tips and Recommendations

5.2.4.5.1.13.7.1 Result Array Fields that Must be Manually Updated

The FMC Tool lets you define custom protocol headers, and the Soft Parser parses these headers. However, the Soft Parser does not update header fields for you (other than advancing the frame window and updating the line-up confirm vector (LCV) with the previous protocol). (See [The before Element](#) on page 279, [The after Element](#) on page 279, and [The action Element \(for use in a Custom Protocol file\)](#) on page 283 topics for more information.)

Therefore, some result array fields are left empty unless you manually update them. These fields might be needed in later stages in order for the Soft Parser to correctly interpret the custom protocol header. A list of result array fields that should be updated appears in the Frame Manager Parser section of the *QorIQ Data Path Acceleration Architecture (DPAA) Reference Manual*. These fields include \$Classificationplanid, \$nxtHdr, \$Runningsum, HXS offsets, Last E Type Offset, and \$nxtHdrOffset. Note that the HXS offsets, \$nxtHdr, and \$nxtHdrOffset fields are also used internally by the Soft Parser; therefore, these fields should be modified carefully.

The \$nxtHdr fields should be modified only if the custom protocol does not jump to 'after_ip' or 'after_ethernet', or if you want to change the next protocol when jumping to 'after_ip' or 'after_ethernet'. You should only modify the HXS offsets and next header offsets in the 'after' element or in the 'before' element if the parser exits without advancing the frame window.

Finally, the LCV should be manually updated when a custom protocol is being parsed. This can be done using the 'confirmcustom' attribute, which is available in the 'before', 'after', and 'action' elements.

5.2.4.5.1.13.7.2 Result Array Fields that Should Not be Modified

Some fields in the result array are for the Soft Parser's exclusive use and therefore should not be modified by the user. These fields are:

- \$GPR1 is used to store temporary values in complex operations; therefore, you should not modify it.
- \$nxtHdr is used to calculate the position of the next protocol header when the 'protocol' element's 'nextproto' attribute is set to 'next_ethernet' or 'next_ip'. Therefore, this variable should not be modified when 'nextproto' equals one of these values.
- \$prevprotoOffset is used to advance the frame window between the 'before' and 'after' elements or when using the 'action' element with the 'advance' attribute in the 'before' element. Therefore, this variable should not be modified in the 'before' element unless the Soft Parser exits this element without advancing the frame window. In addition, \$prevprotoOffset can equal these result array variables: \$ethoffset, \$greoffset, \$ipoffset_n, \$llcsnapoffset, minencapoffset, mplsoffset_n, pppoeoffset, l4offset, vlanoffset_n, and \$nxtHdrOffset. As a result, these variable should also not be modified by code in the 'before' element.
- \$nxtHdrOffset is used to advance the frame window between the 'before' and 'after' elements or when using the 'action' element with the 'advance' attribute in the 'before' element. Therefore, this variable should not be modified in the 'before' element unless the Soft Parser exits this element without advancing the frame window.

5.2.4.5.1.13.7.3 Setting the Next Protocol

The Soft Parser can be used to add code for an existing protocol or to define an entirely new protocol. When it is used as an extension for an existing protocol and no new frame headers are being parsed, the 'nextproto' attribute of the 'action' element should be set to 'return'. In this case, the nextproto attribute can also be left empty since 'return' is the default value. If 'return' is set, the Soft Parser will execute its code and then the Hard Parser will continue parsing at the same position in the frame header at which it stopped.

When the Soft Parser is used for a custom protocol with its own header, the Hard Parser must skip this header (since it does not know how to parse it) and, therefore, the next protocol must be set to a specific protocol. If the next protocol is unknown, the 'nextproto' attribute in the 'action' element can be set to 'after_ip' or 'after_ethernet'. In these cases, the next protocol header is determined using the value of the \$nxtHdr field.

Example

1. If we want to execute the Soft Parser because when we parse the Ethernet protocol, our code will likely include an action similar to the action below, which will appear in the 'before' element.

```
<action type="exit" advance="no" next="return">
```

2. If we want to add a custom protocol after Ethernet and then jump to IPv6, our code will likely include an action similar to the action below, which will appear in the 'after' element...

```
<action type="exit" advance="yes" next="ipv6">
```

3. If we want to add a custom protocol after the Ethernet header, and we do not know where to jump next, our code will likely include an action similar to the action shown below, which will appear in the 'after' element.

```
<action type="exit" advance="yes" next="after_ethernet">
```

5.2.4.5.1.13.8 Limitations

This section discusses limitations you should consider when working with the FMC Tool's Soft Parser functionality.

5.2.4.5.1.13.8.1 Complex Expressions

Some expressions contain so many operations and parentheses that they are too complicated for the Soft Parser. If you receive an error stating that an expression is too complex, it may be necessary to simplify the expression by splitting it into multiple, smaller expressions, using parentheses, or storing temporary values in the result array variables.

Note: \$GPR1 is recommended for storing temporary variables. Do not use \$GPR2 for temporary variables because it is used internally by the tool).

Note that the checksum operation expressions can easily become too complex and must be simplified.

5.2.4.5.1.14 NetPCD Reference

5.2.4.5.1.14.1 The netpcd element

The 'netpcd' element is the root element of a NetPCD document (also known as a policy file). As a result, the 'netpcd' element must appear before any other NetPCD element.

5.2.4.5.1.14.1.1 netpcd Attribute Definitions

Table 30. netpcd Attribute Definitions

Attribute	Requirement	Description
name	optional	Free text. Use to describe the name and the purpose of the Policy file.
version="1.0"	optional	Version of the NetPCD DTD or XML schema. Currently there is only one version - "1.0," which is the default.
creator	optional	Author's name
date	optional	Date the document was created

5.2.4.5.1.14.1.2 netpcd Example

```
<?xml version="1.0"?>
<netpcd version="1.0" name="Example" creator="Serge Lamikhov">
  <!-- Other NetPCD elements like 'policy', 'distribution', etc -->
  <policy name="ipv4">
    <dist_order>
```

```

    <distributionref name="eth_dist"/>
    <distributionref name="default_dist"/>
  </dist_order>
</policy>
</netpcd>

```

5.2.4.5.1.14.2 The policy element

The 'policy' element defines a prioritized list of distributions.

A policy element is assigned (via its name attribute) to a port or ports using markup in the Configuration file. Thus, the 'policy' element is the means by which specific PCD rules defined in the Policy file are applied to traffic arriving on particular FMan ports.

Upon receipt of a frame on given port, the Hard Parser tries to match this frame to the distribution listed first in the policy assigned to this port. If the frame matches, this distribution handles the frame. If the frame does not match, the Hard Parser next tries to match the frame to the second distribution in the policy list. This process continues until a distribution in the list matches or no more distributions are left in the policy element's list, in which case, the frame is placed on the FMan's default receive queue.

5.2.4.5.1.14.2.1 policy Attribute Definitions

Table 31. policy Attribute Definitions

Attribute	Requirement	Description
name	required	Name of the policy. A port definition in the Configuration file references this name, thereby applying this policy to all frames arriving on this port.

5.2.4.5.1.14.2.2 policy Example

Policy File

```

<policy name="ipv4"> <!-- policy name is ipv4 -->
  <dist_order>
    <distributionref name="eth_dist"/>
    <distributionref name="default_dist"/>
  </dist_order>
</policy>

```

Configuration File

```

<cfgdata>
  <config>
    <engine="fm0">
      <port type="MAC" number="1" policy="ipv4"/> <!-- policy name ipv4 goes here -->
    </engine>
  </config>
</cfgdata>

```

5.2.4.5.1.14.3 The dist_order element

The 'dist_order' element is a container for a list of distribution references.

The Hard Parser chooses a particular distribution in this list at the moment when the protocol set made from the protocols participating in a distribution is a subset of the protocols found in the current network packet.

The distribution reference list contained within 'dist_order' element is processed sequentially, and the first conforming distribution is the distribution that is used. Thus, the order of distribution references is important.

5.2.4.5.1.14.3.1 dist_order Attribute Definitions

Table 32. dist_order Attribute Definitions

Attribute	Requirement	Description
none	n/a	n/a

5.2.4.5.1.14.3.2 dist_order Example

```
<policy name="ipv4">
  <dist_order>
    <distributionref name="tcp_dist"/>
    <distributionref name="udp_dist"/>
    <distributionref name="ethernet_dist"/>
    <distributionref name="default_dist"/>
  </dist_order>
</policy>
```

Note: In this example, putting "ethernet_dist" (which is supposed to process network traffic other than TCP and UDP) above "tcp_dist" will lead to all traffic be distributed according to "ethernet_dist" rule and no packets will reach "tcp_dist" or "udp_dist" rules. This is because the Ethernet protocol is a part of TCP and UDP frames as well.

5.2.4.5.1.14.4 The distributionref element

The 'distributionref' element references a 'distribution' element by its name.

The 'dist_order' element contains one or more 'distributionref' elements, thereby defining a prioritized list of distributions.

5.2.4.5.1.14.4.1 distributionref Attribute Definitions

Table 33. distributionref Attribute Definitions

Attribute	Requirement	Description
name	required	Name of the referenced 'distribution' element

5.2.4.5.1.14.4.2 distributionref Example

```
<policy name="ipv4">
  <dist_order>
    <distributionref name="eth_dist"/>
    <distributionref name="default_dist"/>
  </dist_order>
</policy>
```

5.2.4.5.1.14.5 The distribution element

The 'distribution' element is a container for child elements that define frame match rules and frame handling rules.

Frame match rules determine whether the current frame matches (and is therefore handled by) this distribution. Frame handling rules define what action is performed on matching frames.

Use the 'protocols' element and/or the 'key' element to define frame match rules.

Use the 'action', 'key', 'queue', and 'combine' elements to define frame handling rules.

An 'action' element within a the distribution passes the frame to the specified Policy file element for further processing

The 'key', 'queue' and (optional) 'combine' elements within a distribution together provide inputs to a hash algorithm that distributes frames evenly over a range of frame queues. The 'key' element defines the protocol header fields to use as the hash key, the 'queue' element defines the base value and number of FQIDs in the frame queue range, and the optional 'combine' elements give you fine control over the exact FQIDs that the algorithm generates.

Note: You can use an 'action' element in the hash scenario described above to pass the frame to a policer profile, which may abort the enqueue operation and drop the frame if traffic conditions warrant. In the absence of an 'action' element, frame processing concludes (and the frame leaves the FMan) at the end of the 'distribution' element.

A distribution's frame queue ID calculation is performed as follows:

- A hash key is formed by extracting and concatenating the protocol header fields specified by the 'key' element.
- The result value is hashed to a 64-bit CRC.
- The number of least significant bits is taken based on the 'count' attribute of the 'queue' element.
- The resulting value is ORed with the data retrieved according to the 'combine' elements.
- The resulting value is ORed with the 'base' attribute value of the 'queue' element.

All child elements are optional. Appropriate hardware dependent default values are used in cases where a child element does not exist in the 'distribution' definition.

5.2.4.5.1.14.5.1 distribution Attribute Definitions

Table 34. distribution Attribute Definitions

Attribute	Requirement	Description
name	required	Name of the distribution. Any references to a distribution are made using to this name.
description	optional	Free text describing the element purpose.
comment	optional	Free text providing any other information.

5.2.4.5.1.14.5.2 distribution Example

```
<distribution name="eth_dist" description="Ethernet protocol based distribution">
  <queue count="0x400" base="0x810000"/>
  <key>
    <fieldref name="ethernet.src"/>
    <fieldref name="ethernet.dst"/>
  </key>
  <combine portid="true" offset="10" mask="0xFF"/>
  <combine frame="112" offset="2" size="16" mask="0xFF"/>
  <action type="classification" name="eth_dest_clsif"/>
</distribution>
```

5.2.4.5.1.14.5.3 Default Groups

XML 'defaults' element is a container for parameters necessary for configuration of the default groups and private default registers. The element, if it exists, can be used as a child of element 'distribution'. This element contains a list of 'default' elements.

Table 35. 'default' Elements Attributes:

Attribute	Requirement	Description
private0	optional	The scheme default register 0.
private1	optional	The scheme default register 1.

Element 'default' attributes. This element can appear as a child to the element 'defaults':

Table 36. 'default' Element Attributes:

Attribute	Requirement	Description
type	required	Default type select. Possible values are: <ol style="list-style-type: none"> 1. "from_data" – any data extraction that is not one of the full fields that can be used as type. 2. "from_data_no_v" – any data extraction without validation. 3. "not_from_data" – extraction from parser result or direct use of default value. 4. "mac_addr" – MAC Address. 5. "tci" – TCI field. 6. "enet_type" – ENET Type. 7. "ppp_session_id" – PPP Session id. 8. "ppp_protocol_id" – PPP Protocol id. 9. "mpls_label" – MPLS Label. 10. "ip_addr" – IP Addr. 11. "protocol_type" – Protocol type. 12. "ip_tos_tc" – TOC or TC. 13. "ipv6_flow_label" – IPV6 flow label. 14. "ipsec_spi" – IPSEC SPI. 15. "l4_port" – L4 Port. 16. "tcp_flag" – TCP Flag
select	required	Default register select. Possible values are: <ol style="list-style-type: none"> 1. "gbl0" – Default selection is KG register 0. 2. "gbl1" – Default selection is KG register 1. 3. "private0" – Default selection is a per scheme register 0. 4. "private1" – Default selection is a per scheme register 1

Here is an example of possible default groups and nonheader definition:

```

<distribution name="Distribution1">
  <queue base="1" count="8"/>
  <key>
    <fieldref name="ipv4.src"/>
    <fieldref name="ipv4.dst"/>
  
```

```

    <fieldref name="ipv4.nextp"/>
    <nonheader source="default" offset="0" size="4"/>
  </key>
  <defaults private0="0xAAAAAAAA">
    <default type="from_data" select="private0"/>
    <default type="from_data_no_v" select="private0"/>
    <default type="not_from_data" select="private0"/>
  </defaults>
  <action type="drop"/>
</distribution>

```

5.2.4.5.1.14.6 The key element

The 'key' element contains a list of 'fieldref' elements. The 'fieldref' elements define the protocol header fields whose values are concatenated to form a hash key. The Key Gen sub block hashes this key and uses a portion of the result in its frame queue ID (FQID) calculation.

5.2.4.5.1.14.6.1 key Attribute Definitions

Table 37. key Attribute Definitions

Attribute	Requirement	Description
shift	optional	Defines the amount by which the concatenation of the fields in the 'key' element are right shifted. The default value is zero. Note: The 'shift' attribute is ignored if the 'key' elements appears within a 'classification' element.
symmetric	optional	Generate the same hash for frames with swapped source and destination fields on all layers. If source is selected, destination must also be selected, and vice versa.

5.2.4.5.1.14.6.2 key Example

```

<key shift="16">
  <fieldref name="ethernet.src"/>
  <fieldref name="ethernet.dst"/>
</key>

```

5.2.4.5.1.14.7 The fieldref element

The 'fieldref' element refers to a protocol header field by its name.

The Standard Protocol file contains the names of the available protocols and their fields. This file is named hxs_pdl_v3.xml and is in the directory /etc/fmc/config/.

5.2.4.5.1.14.7.1 fieldref Attribute Definitions

Table 38. fieldref Attribute Definitions

Attribute	Requirement	Description
name	required	The referenced field name. The field's name should be provided in the form of "protocolname.fieldname".

5.2.4.5.1.14.7.2 fieldref Example

```
<key>
  <fieldref name="ethernet.src"/>
  <fieldref name="ethernet.dst"/>
</key>
```

5.2.4.5.1.14.8 The queue element

The 'queue' element defines the number of queues (default is one) and the base value for the FQIDs for these queues.

When used within a 'distribution' element, the 'queue' element defines a range of queues over which to evenly distribute frames.

When used within other elements, such as a 'classification' element, the 'queue' element defines the single queue on which to place a frame.

5.2.4.5.1.14.8.1 queue Attribute Definitions

Table 39. queue Attribute Definitions

Attribute	Requirement	Description
base	required	The base frame queue ID value.
count	optional	This attribute is only relevant only when a 'queue' element appears within a 'distribution' element. In this case, the 'count' attribute defines the number of frame queues over which to distribute frames. Valid values for 'count' are powers of 2. The default value is 1.

5.2.4.5.1.14.8.2 queue Example

```
<distribution name="eth_dist">
  <queue count="0x400" base="0x810000"/>
  <key>
    <fieldref name="ethernet.src"/>
    <fieldref name="ethernet.dst"/>
  </key>
</distribution>
```

5.2.4.5.1.14.9 The protocols and protocolref elements

The 'protocols' and 'protocolref' elements are used together to extend a 'distribution' element's frame match conditions.

As explained in the 'dist_order' description, a distribution is chosen based on the set of protocols specified in its 'key' element. The 'protocols' and 'protocolref' elements let you extend this set of protocols beyond those listed in the 'key' element.

5.2.4.5.1.14.9.1 protocols and protocolref Attribute Definitions

Table 40. protocols and protocolref Attribute Definitions

Attribute	Requirement	Description
name	required	The name of the protocol.
<i>Table continues on the next page...</i>		

Table 40. protocols and protocolref Attribute Definitions (continued)

Attribute	Requirement	Description
opt	optional	<p>Applicable only for protocolref attribute</p> <p>Use it in a scheme for detecting protocols with the chosen options (e.g. to detect ETHERNET with BROADCAST or MULTICAST option)</p> <p>Table 2 contains all possible values. The values are grouped, each group being separated by a blank row. Values from different groups can be ORed</p>

Table 41. Protocol options. Groups are separated by empty rows.

Value	Description
0x800000 00	Ethernet Broadcast
0x400000 00	Ethernet Multicast
0x200000 00	Stacked VLAN
0x100000 00	Stacked MPLS
0x080000 00	IPv4 Broadcast
0x040000 00	IPv4 Multicast
0x020000 00	Tunneled IPv4 - Unicast
0x010000 00	Tunneled IPv4 - Broadcast/Multicast
0x000000 08	IPV4 reassembly option. When using this option, the IPV4 Reassembly manipulation requires network environment with IPV4 header
0x008000 00	IPv6 Multicast
0x004000 00	Tunneled IPv6 - Unicast
<i>Table continues on the next page...</i>	

Table 41. Protocol options. Groups are separated by empty rows. (continued)

Value	Description
0x002000 00	Tunneled IPv6 - Multicast
0x000000 04	IPV6 reassembly option. When using this option, the IPV6 Reassembly manipulation requires network environment with IPV6 header. In case where fragment found, the fragment-extension offset may be found at 'shim2' (in parser-result).
0x000000 08	CAPWAP reassembly option. When using this option, the CAPWAP Reassembly manipulation requires network environment with CAPWAP header. In case where fragment found, the fragment-extension offset may be found at 'shim2' (in parser-result).

5.2.4.5.1.14.9.2 protocols and protocolref Example

```
<!-- The example demonstrates the case in which -->
<!-- frame queue ID calculation is done using Ethernet header fields, -->
<!-- but the condition for matching a frame to this distribution is -->
<!-- extended by also requiring the presence of a UDP protocol header -->
<distribution name="eth_dist">
  <protocols>
    <protocolref name="udp" opt="0x00000008"/>
  </protocols>

  <queue count="0x400" base="0x810000"/>
  <key>
    <fieldref name="ethernet.src"/>
    <fieldref name="ethernet.dst"/>
  </key>
</distribution>
```

5.2.4.5.1.14.10 The combine element

The 'combine' element (like the 'key' element) is used in a 'distribution' element's frame queue ID calculation. The value built by the 'key' element is hashed, but the value of the 'combine' element is directly bitwised OR'd with the previous 24-bit FQID result.

A single 'combine' element identifies just one byte to retrieve and OR. To work around this limitation, you can have multiple 'combine' elements in a 'distribution' element.

5.2.4.5.1.14.10.1 combine Attribute Definitions

Table 42. combine Attribute Definitions

Attribute	Requirement	Description
portid	required (<i>in absence of frame attribute</i>)	Valid values: true or false If true, this attribute indicates that the logical port ID byte specified in the Configuration file should be retrieved and used in the bitwise OR part of a distribution's FQID calculation. <i>Note that portid and frame are mutually exclusive attributes.</i>
frame	required (<i>in absence of portid attribute</i>)	Valid values: numeric string This attribute identifies the byte with the frame header to extract and use in the bitwise OR part of the FQID calculation. The attribute's value indicates the bit offset from the beginning of the frame. The specified value must be divisible by 8, so it references the first bit of a byte. <i>Note that portid and frame are mutually exclusive attributes.</i>
offset	optional	This attribute controls the placement of the extracted data in the result Frame Queue ID. The offset starts at the FQID's most significant bit.
mask	optional	This attribute defines valid bits in the retrieved value. The extracted value is bitwise ANDed with the mask prior to being ORed with the previous Frame Queue ID value.

5.2.4.5.1.14.10.2 combine Example

```
<distribution name="eth_dist">
  <queue count="0x400" base="0x810000"/>
  <key>
    <fieldref name="ethernet.src"/>
    <fieldref name="ethernet.dst"/>
  </key>
  <combine portid="true" offset="10" mask="0xFF"/>
  <combine frame="64" offset="2" mask="0xFF"/>
  <action type="classification" name="eth_dest_cls"/>
</distribution>
```

5.2.4.5.1.14.11 The action element (for use in a policy file)

The 'action' element permits you to establish a topological parse, classify, police, distribute configuration by defining the next processing element within a distribution, classification, or policer profile.

If there is no 'action' element within a distribution, classification, or policer profile, the default behavior is the completion of PCD frame processing, allowing the frame to leave the Frame Manager. Some hardware restrictions apply in the choice of the next processing element.

5.2.4.5.1.14.11.1 action Attribute Definitions

Table 43. action Attribute Definitions

Attribute	Requirement	Description
type	required	The type of the 'action' element defines the next processing element. Valid values are: <ul style="list-style-type: none"> • "distribution" • "classification" • "policer" • "drop" (Permitted only when the 'action' element is inside a 'policer' element.)
name	required	The name of the element of the type defined in the 'type' attribute. This attribute is not relevant if type is "drop".
condition	required (<i>when used within a 'policer' element</i>) optional (<i>when used within a 'distribution' or 'classification' element</i>)	This attribute defines the condition under which the 'action' is to be taken. This attribute is only relevant when used inside a 'policer' or a 'classification' element. Valid values are: <ul style="list-style-type: none"> • "on-green" • "on-yellow" • "on-red" • "on-miss"

5.2.4.5.1.14.11.2 Statistics

Attribute 'statistics' for action element of the classification and classification entries. This tells if statistics are made on that entry or on the on-miss.

Table 44. 'statistics' Element Attributes:

Attribute	Requirement	Description
statistics	optional	Enable statistics for a particular action. Possible values are: <ul style="list-style-type: none"> • enable/yes/true – to enable it. • disable

5.2.4.5.1.14.11.3 action Example

```
<distribution name="special_dist">
  <queue count="1" base="0xABCD"/>
  <action type="policer" name="policer2"/>
</distribution>

<policer name="policer2">
  <algorithm>rfc2698</algorithm>
  <color_mode>color_aware</color_mode>
  <CIR>1000000</CIR>
  <EIR>1400000</EIR>
  <CBS>1000000</CBS>
```

```
<EBS>140000</EBS>
<unit>packet</unit>
<action condition="on-green" type="distribution" name="special2_dist"/>
<action condition="on-yellow" type="drop"/>
<action condition="on-red" type="drop"/>
</policer>
```

5.2.4.5.1.14.12 The classification element

The 'classification' element allows exact match frame processing.

A classification starts with a 'classification' element, which is a container for these child elements:

- A 'key' element that defines the header fields (in protocol.field form) to use in the exact match operation
- One or more 'entry' elements, each of which defines a value to which the specified fields are compared and a 'queue' and/or 'action' element that defines what to do with the frame upon a match
- An optional 'action' element that defines the default action to take if none of the exact match conditions is met

5.2.4.5.1.14.12.1 classification Attribute Definitions

Table 45. classification Attribute Definitions

Attribute	Requirement	Description
name	required	The name of the classification

5.2.4.5.1.14.12.2 classification Statistics

The statistics are enabled on the Classification element. The parameters to setup the statistics are: - the attribute **statistics** of the element **classification**, the attribute **statistics** of the actions on entries/on-miss and the element **framelength** with attributes **index** and **value**.

Attribute 'statistics' for classification – this specifies the type of statistic used in the entire classification

Table 46. 'statistics' Element Attributes:

Attribute	Requirement	Description
statistics	optional	Choose statistic mode for the particular entry. Possible values are: <ul style="list-style-type: none"> • none • frame • byteframe • rmon

5.2.4.5.1.14.12.3 classification Example

```
<classification name="eth_dest_clsfc">
  <key>
    <fieldref name="ethernet.dst"/>
  </key>

  <entry>
    <data>0x1234567890AB1234567890AB</data>
```

```

    <queue base="0x550000"/>
  </entry>

  <entry>
    <data>0xFFFFFFFFFFFFFFFFFFFFFFFF</data>
    <action type="classification" name="eth_dest_2_clsif"/>
  </entry>

  <action condition="on-miss" type="distribution" name="default_dist"/>
</classification>

```

5.2.4.5.14.12.4 Frame Replicators

The element **replicator** is implemented in FMC as a standalone entity.

This element can follow a Classification in the flow, as a target for one of the actions of the entries or on the on-miss. It is similar to Classification but it has no data/mask in entries, on-miss action and key element.

Table 47. 'fragmentation' Element Attributes:

Attribute	Requirement	Description
name	required	Name of the element. The name is used to refer the frame replicator.
max	optional	The maximum number of entries the frame replicator can have (default and minimum is 2). If the value entered is smaller than 2 or the attribute is not set, the value is set to 2.

The element **entry** has the same syntax as the element **classification**, but the data and mask are not needed and thus are ignored. The action targets of the entry are restricted to:

- policer
- enqueue
- direct distribution

replicator example:

```

<replicator name="frep_1" max="32">
  <entry>
    <action type="policer" name="policer_1"/>
  </entry>
  <entry>
    <queue base="0x0"/>
    <action type="distribution" name="dist_1"/>
  </entry>
  <entry>
    <queue base="0x220"/>
    <vsp name="vsp01">
  </entry>
  <entry>
    <queue base="0x240"/>
    <vsp base="2">
  </entry>
</replicator>

```

Using the frame replicator in an action:

```
<classification name="class_1" max="0" masks="yes">
  <key>
    <fieldref name="ethernet.type"/>
  </key>
  <entry>
    <data>0x8870</data>
    <queue base="0x01"/>
    <action type="replicator" name="frep_1"/>
  </entry>
  <action condition="on-miss" type="replicator" name="frep_1"/>
</classification>
```

5.2.4.5.1.14.12.5 framelength Statistics

Element **framelength** attributes (there can be up to 10 values set, in ascending order and last one must be 0xFFFF). The element **framelength** is valid only for RMON statistics.

Table 48. 'framelength' Element Attributes:

Attribute	Requirement	Description
statistics	required	The index for the frame length value specified. Possible values are from 0 to 9.
value	required	The value to be added at the specified index. Maximum value is 0xFFFF and must be added at index 9. (FMC sets it initially by default).

5.2.4.5.1.14.12.6 Statistics Example

Statistics Example

```
<!-- Coarse classification -->
<classification name="classif_1" max="32" masks="yes" statistics="rmon">
  <!-- Key value to be extracted from the packet -->
  <key>
    <fieldref name="ipv4.dst"/>
  </key>

  <framelength index="0" value="0x1100"/>
  <framelength index="1" value="0x1200"/>
  <framelength index="2" value="0x1300"/>
  <framelength index="3" value="0x1400"/>
  <framelength index="4" value="0x1500"/>
  <framelength index="5" value="0x1600"/>
  <framelength index="6" value="0x1700"/>
  <framelength index="7" value="0x1800"/>
  <framelength index="8" value="0x1900"/>
  <framelength index="9" value="0xFFFF"/>

  <!-- Entries in the lookup table -->
  <entry>
    <!-- 192.168.10.10 -->
    <data>0xC0A80A0A</data>
    <queue base="0x1010"/>
  </entry>
</classification>
```

```

    <action statistics="enable"/>
  </entry>
</classification>

```

5.2.4.5.1.14.12.7 Coarse Classification Resource Reservation

FMD API changes allow pre-allocation of MURAM memory for classification tables. This will be reflected in NetPCD XML syntax extension by introducing attributes **max** and **masks** of the element **classification** as shown in the example below. In addition, to allow proper order of PCD elements initialization, and for the condition that not all **entry** elements are known at initialization time, the XML element **may-use** is introduced:

```

<!-- Coarse classification -->
<classification name="classif_1" max="32" masks="yes" statistics="mode">
  <!-- Key value to be extracted from the packet -->
  <key>
    <fieldref name="ipv4.dst"/>
  </key>

  <may-use>
    <action type="classification" name="fman_test_classif_1"/>
    <action type="distribution" name="default_dist"/>
  </may-use>

  <!-- Entries in the lookup table -->
  <entry>
    <!-- 192.168.10.10 -->
    <data>0xC0A80A0A</data>
    <queue base="0x1010"/>
  </entry>
</classification>

```

Resource Allocation Attributes:

Table 49. Resource Reservation Attributes:

Attribute	Requirement	Description
max	optional	<p>If it exists, this parameter defines the maximum number of coarse classification entries allocated for this PCD element.</p> <p style="text-align: center;">NOTE</p> <p>The element classification may still contain pre-initialized entries, or, alternatively, be empty.</p> <p style="text-align: center;">NOTE</p> <p>For the case of empty or partially initialized element classification, usage of the element may-use might be required .</p>
masks	optional	<p>If provided, indicates that MURAM allocation should be done with the assumption that additional memory is required for an elements' masks. Possible values are:</p> <ul style="list-style-type: none"> • no – don't allocate memory for masks (default) • yes – allocate memory for masks.

'may-use' Element Description:

Table 50. 'may-use' Element Attributes:

Attribute	Requirement	Description
may-use	optional	<p>Contains list of 'action' elements that may appear in the 'classification' entries or, be applied dynamically after partial initial configuration.</p> <p style="text-align: center;">NOTE</p> <p>Attention: the use of this element is required if initial 'classification' is empty and dynamic entries, added through FMD API, use those PCD entities</p>

5.2.4.5.1.14.13 The entry element

The 'entry' element defines:

- the value to use in an exact match comparison with the fields specified by the 'key' element in a classification
- the action to be taken upon a match

An 'entry' element contains a 'data' element which, in turn, contains a numeric value written in hexadecimal form (that is, with a "0x" prefix). The data length of this value is determined by length of the set of 'key' fields.

In addition to the 'data' element, each 'entry' element may also contain these elements:

- queue - causes the frame to be placed on the specified queue
- action - passes the frame to the specified element within the Policy file for further processing.
- mask - a value in hexadecimal format that is applied to the data element

5.2.4.5.1.14.13.1 entry Attribute Definitions

Table 51. entry Attribute Definitions

Attribute	Requirement	Description
none	n/a	n/a

5.2.4.5.1.14.13.2 entry Example

```
<classification name="eth_dest_clsif">
  <key>
    <fieldref name="ethernet.dst"/>
  </key>

  <entry>
    <data>0x1234567890AB1234567890AB</data>
    <queue base="0x550000"/>
  </entry>
</classification>
```

5.2.4.5.1.14.14 The policer element

The 'policer' element is a container whose child elements define a policer profile that performs network bandwidth management.

5.2.4.5.1.14.14.1 policer Attribute Definitions

Table 52. policer Attribute Definitions

Attribute	Requirement	Description
name	required	Name of the policer profile.
algorithm	required	Algorithm used for policing. Valid values: "rfc2698", "rfc4115", "pass_through".
color_mode	required	Color mode used for policing. Valid values: "color_aware", "color_blind".
default_color	optional	Use when algorithm is "pass_through" and color_mode is "color_blind". In this mode, the policer re-colors incoming packets with the specified default color. Valid values: "red", "yellow", "green", or "override". If the value is override, the next invoked action is that specified for "green". The default value is "green".
unit	required	The unit to be used for numeric parameters. Valid values: "packet", "byte".
CIR	required	Committed information rate ^[8]
PIR	required	Peak (or excess) information rate ^[8]
CBS	required	Committed burst size ^[9]
PBS	required	Peak (or excess) burst size ^[9]

5.2.4.5.1.14.14.2 policer Example

```
<policer name="policer2">
  <algorithm>rfc2698</algorithm>
  <color_mode>color_aware</color_mode>
  <CIR>1000000</CIR>
  <EIR>1400000</EIR>
  <CBS>1000000</CBS>
  <EBS>1400000</EBS>
  <unit>packet</unit>
  <action condition="on-green" type="distribution" name="default_dist"/>
  <action condition="on-yellow" type="distribution" name="special2_dist"/>
  <action condition="on-red" type="drop"/>
</policer>
```

5.2.4.5.1.14.15 The nonheader element

Use the 'nonheader' element within a 'key' element to select a non-header extraction source.

Note: The 'nonheader' element can appear within a 'classification' element only. Further, the 'nonheader' element cannot be used at the same time as the 'fieldref' element.

5.2.4.5.1.14.15.1 nonheader Attribute Definitions

[8] If "unit" attribute is "packet" specify CIR and PIR in packets/second. If "unit" attribute is "byte" specify CIR and PIR in Kbits/second.

[9] If "unit" attribute is "packet" specify CBS and PBS in packets. If "unit" attribute is "byte" specify CBS and PBS in bytes.

Table 53. nonheader Attribute Definitions

Attribute	Requirement	Description
source	required	<p>Non-header extraction source</p> <p>Valid values are:</p> <ul style="list-style-type: none"> "frame_start" - Extract from beginning of frame. "key" - Extract from key value built by 'distribution' at preceding step (CC only). "hash" - Extract from hash value built by 'distribution' at preceding step (CC only). "parser" - Extract from parse result array. "fqid" - Use enqueue FQID as the key value. "flowid" - Use dequeue FQID as the key value (CC only) "default" - Extract from a default value (distribution only). "endofparse" - Extract from the point where parsing had finished (distribution only).
action	Required if source is "hash", "flowid" or "key". In other cases, this attribute must not be used.	<p>The type of action for the extraction</p> <p>Valid values are:</p> <ul style="list-style-type: none"> "indexed_lookup" (permitted only for "hash" and "flowid" sources). The extracted value is interpreted as an entry index of classification table "exact_match" (permitted only for "key" and "hash" sources). The extracted value is compared with 'key' value of the entry.
offset	required	Byte offset. Offset of key from start of frame, internal frame context or parse result array. Refer "Table 8-398. Table Descriptor (Type = 01)" of DPAA Reference Manual for full description and possible values
size	required	Size of the key in bytes.
ic_index_mask	Optional (Valid only if action is "indexed_lookup")	Internal context index mask. For the full description and possible values, refer "Table 8-399. Operation Code Description" of DPAA Reference Manual

If the action is "indexed_lookup" and the source is "hash" special checks are done in the drivers on the configured entries and maximum number of entries according to the internal context index mask specified. FMC is adjusting automatically the configured entries if they don't match the provided mask: if the entry must be initialized but the user didn't supplied it a default one is created and if the entry must be uninitialized it's deleted by FMC. Also FMC adjusts the maximum number of entries if it's not configured as 0.

5.2.4.5.1.14.15.2 nonheader Example

```
<classification name="ptp_condition_class">
  <key>
    <nonheader source="hash" action="indexed_lookup" offset="2" size="2" ic_index_mask="0x01b0">
```

```
</key>

<entry>
  <data>0x13F</data>
  <queue base="0x01"/>
</entry>
</classification>
```

5.2.4.5.1.14.16 Hash Tables

The element 'hashtable' can be specified inside an element 'key' of a 'classification'. The element 'hashtable' cannot appear in the same time with either elements 'fieldref' or 'nonheader' in the same 'key'. If the element 'hashtable' is used, the 'classification' may have no entries as these are supposed to be filled at runtime.

Table 54. 'fragmentation' Element Attributes:

Attribute	Requirement	Description
mask	required	Mask that will be used on the hash-result; The number-of-sets for this hash will be calculated as $2^{(\text{number of bits set in 'mask'})}$; The 4 lower bits must be cleared.
hashshift	optional	Byte offset from the beginning of the KeyGen hash result to the 2-bytes to be used as hash index.(Default 0)
keysize	required	Size of the exact match keys held by the hash buckets.

Hash table example:

```
<classification name="classif_1" max="2" statistics="none">
  <key>
    <hashtable mask="0x30" hashshift="0" keysize="24"/>
  </key>
</classification>
```

5.2.4.5.1.14.17 Virtual Storage Profiles Element

The element 'vsp' (Virtual Storage Profile) is implemented in FMC as a standalone entity or can be defined directly in the element that uses it. The element 'vsp' can be used inside distributions, classification and entries (both classification and replicator). When used directly in the 'classification' element (not in 'entry') it counts for the on-miss action. If the 'action' of the 'entry' or on-miss goes to another 'classification' or 'replicator' the 'vsp' is ignored.

5.2.4.5.1.14.17.1 vsp Attributes

Table 55. 'vsp' Element Attributes:

Attribute	Requirement	Description
name	required	Name of the element. The name is used to refer the virtual storage profile inside the elements that are using it.

Table continues on the next page...

Table 55. 'vsp' Element Attributes: (continued)

Attribute	Requirement	Description
type	optional	The type of the VSP. Values: <ul style="list-style-type: none"> • direct – (default) the relative profile ID is selected directly by the 'base' attribute. • indirect – the relative profile ID is selected base on the attributes fqshift, vspoffset, and vspcount can be used only in distribution.
base	required for direct.	--
fqshift	required for indirect.	Shift of KeyGen results without the FQID base.
vspoffset	optional for indirect	OR of KeyGen results without the FQID base; should indicate the storage profile offset within the port's storage profiles window.
vspcount	optional for indirect	Range of profiles starting at base.

5.2.4.5.1.14.17.2 vsp Examples

VSP examples (standalone, defined in element, direct/indirect): The action targets of the entry are restricted to:

```

<vsp name = "storage01" base = "6"/>
<vsp name = "storage02" type = "indirect" fqshift="2" vspoffset="3"          vspcount="8"/>
<vsp name = "storage03" type = "direct" base = "7"/>

Usage:

...

<entry>
    <queue base="0x220"/>
    <vsp name="storage01">
</entry>

...

<distribution name="dist1">
    ...
    <queue count="8" base="0x230"/>
    <vsp type="indirect" fqshift="2" vspoffset="0" vspcount="4"/>
    ...
</distribution>

...

<classification name="eth_dest_clsfc">
    <key>
        <fieldref name="ethernet.dst"/>
    </key>
    ...
    <vsp name="storage03">
    <action condition="on-miss" type="distribution" name="garbage"/>
</classification>

```

5.2.4.5.1.14.18 Manipulation Parameters

Frame Manager accelerator (FMan) attaches manipulation actions as an extension to ethernet port and coarse classification 'next engine' dispatch activity.

To reflect the frame data processing and manipulation capabilities of the hardware, which are propagated through Frame Manager Driver (FMD) API, Frame Manager Configuration (FMC) Tool extends the syntax of the NetPCD configuration language by introducing XML entities described in this document.

Manipulation entities are diverse in their purpose and configuration parameters sets. The same manipulation entity can be referred, or attached, from/to several port or classification actions. That is why they are separated from their usage into a separate group called **manipulations**. At the moment of use, an action refers to the corresponding manipulation entity. For example:

```
<netpcd>
  <manipulations>
    <reassembly name="name1">
      .....
    </reassembly>
    <reassembly name="name2">
      .....
    </reassembly>
    <fragmentation name="defrag1">
      .....
    </fragmentation>
  </manipulations>

  <classification name="clsf1">
    .....
    <!-- 192.168.30.30 -->
    <data>0xC0A81E1E</data>
    <fragmentation name="defrag1"/>
    .....
  </classification>

</netpcd>
```

Formal Definition:

XML element **manipulation** is a container for all types of manipulation algorithms. Configuration for each algorithm has its own XML element name.

Currently three manipulations algorithms are available:

1. IP reassembly
2. IP fragmentation
3. header manipulation

Parameters for these entities are described next.

5.2.4.5.1.14.18.1 IP Fragmentation

XML element **fragmentation** is a container for parameters necessary for configuration of the corresponding action modification. The element, if exists, can be used as a child of element **classification**.

Attention: If element **fragmentation** is present together with other 'action' of 'classification' element, the element **fragmentation** is ignored. This is a subject of FMan firmware capabilities and may change in future.

Table 56. 'fragmentation' Element Attributes:

Attribute	Requirement	Description
name	required	Name of the element. The name is used to refer the manipulation algorithm.

Table 57. 'fragmentation' Child Elements:

Attribute	Requirement	Description
size	required	IP fragmentation will be executed for frames with length greater than this value.
dontFragAction	optional	If an IP packet is larger than MTU and its DF bit is set, then this field will determine the action to be taken. Possible values are: <ul style="list-style-type: none"> • discard - the packet (default action) • fragment – fragment the packet and continue normal processing • continue - continue normal processing without fragmenting the packet
scratchBpid	required for existing HW platforms, but not for 9164	Absolute buffer pool id according to BM configuration (DPAA 1.0 only)
sgBpid	optional	Scatter/Gather buffer pool id. If used sgBpidEn will be set to TRUE.
optionsCounterEn	optional	Enables the counter if the value is set to 'yes', 'true' or 'enable'. Disabled for other values. Default is disabled.

Here is an example of possible IP fragmentation definition:

```

<manipulations>
  <fragmentation name="frag1">
    <size>256</size>
    <dontFragAction>continue</dontFragAction>
  </fragmentation>
</manipulations>

<classification name="clsf1">
  . . . . .
  <!-- 192.168.30.30 -->
  <data>0xC0A81E1E</data>
  <fragmentation name="frag1"/>
  . . . . .
</classification>

```

5.2.4.5.1.14.18.2 IP Reassembly

XML element **reassemble** is a container for parameters necessary for configuration of the corresponding action modification. The element, if it exists, can be used as a child of the element **policy**.

Attention: Up to 2 additional KeyGen schemes will be constructed when using this manipulation action. Custom protocol **shim2** is reserved when element **reassemble** participates in a configuration.

Table 58. 'reassemble' Element Attributes:

Attribute	Requirement	Description
Name	required	Name of the element. The name is used to refer the manipulation algorithm

Table 59. 'reassemble' Child Elements:

Attribute	Requirement	Description
sgBpid	required	Absolute buffer pool id according to BM configuration for scatter-gather (DPAA 1.0 only)
maxInProcess	required	Number of frames which can be processed by reassembly at the same time. It has to be power of 2
dataLiodnOffset	optional	Offset of LIODN. Default value is 0
dataMemId	optional	Memory partition ID for data buffers
ipv4minFragSize	required	Minimum fragmentation size for IPv4
ipv6minFragSize	required	EMinimum fragmentation size for IPv6. The value must be equal or higher than 256
timeOutMode	optional	Expiration delay initialized by Reassembly process. Possible values are: <ul style="list-style-type: none"> • frame - limits the time of the reassembly process from the first fragment to the last (default) • fragment - limits the time of receiving the fragment
fqidForTimeOutFrames	required	FQID to assign for frames enqueued during Time Out Process.
numOfFramesPerHashEntry (numOfFramesPerHashEntry1)	required	Number of frames per hash entry needed for reassembly process – for ipv4. Possible values are: numeric values from 1 to 8.
numOfFramesPerHashEntry2	optional	Number of frames per hash entry needed for reassembly process – for ipv6. Possible values are: numeric values from 1 to 6.
timeoutThreshold	required	Represents the time interval in microseconds which defines if opened frame (at least one fragment was processed but not all the fragments) is found as too old
nonConsistentSpFqid	optional	Handles the case when other fragments of the frame corresponds to a different storage profile than the opening fragment. (DPAA >= 1.1 only). Default is 0

Here is an example of possible IP reassembly definition:

```
<manipulations>
  <reassemble name="reasml">
    <sgBpid>2</sgBpid>
    <maxInProcess>1024</maxInProcess>
    <timeOutMode>fragment</timeOutMode>
    <fqidForTimeOutFrames>1024</fqidForTimeOutFrames>
```

```

    <numOfFramesPerHashEntry>8</numOfFramesPerHashEntry>
    <timeoutThreshold>1000000</timeoutThreshold>
    <ipv4minFragSize>0</ipv4minFragSize>
    <ipv6minFragSize>256</ipv6minFragSize>
  </reassemble>
</manipulations>

<policy name="udp_port">
  <dist_order>
    <distributionref name="custom_dist"/>
    <distributionref name="udp_port_dist"/>
    <distributionref name="default_dist"/>
  </dist_order>

  <reassemble name="reasm1"/>
</policy>

```

5.2.4.5.1.14.18.3 Header Manipulation

XML element **header** is a container for parameters necessary for configuration of the corresponding action modification. The element, if it exists, can be used as parameter to the distribution action going to a classification or inside a classification element **entry**.

The XML element **header** may contain:

- **insert**
- **remove**
- **insert_header**
- **remove_header**
- **update**
- **custom**

Certain combinations between them are possible, for example you can have a **remove** and an **insert_header** in the same manipulation.

The header manipulation can be used inside the PCD by inserting an element **header** in the classification entry that specifies the name of the header manipulation defined in the section **manipulations**. This makes sense in a entry that goes to a policer, distribution or PCD done:

```

<entry>
  <data>0x9100</data>
  <queue base="0x01"/>
  <action type="policer" name="plcr_01"/>
  <header name="upd_hdr"/>
</entry>

```

Table 60. 'header' Element Attributes:

Attribute	Requirement	Description
name	required	Name of the element. The name is used to refer the manipulation algorithm
parse	optional	Activate the parser a second time after completing the manipulation of the frame (if 'yes')

Table continues on the next page...

Table 60. 'header' Element Attributes: (continued)

Attribute	Requirement	Description
duplicate	optional	Will duplicate the header manipulation with the same setting a the specified number of times. The names of the nodes will have “_x” added at the end where x is the index of the node. For example <header name=“upd_ipv4” duplicate=“3”> will create the nodes: upd_ipv4_1, upd_ipv4_2 and upd_ipv4_3. This is only a simple tool to duplicate a header manipulation, it does not allow defining chaining between the elements created by duplication.

5.2.4.5.1.14.18.3.1 Header Manipulation - Insert

XML element **insert** is a container for parameters necessary to configure a header insert manipulation operation. The element, if it exists, can be used as a child of element **header**. There can be only one element **insert** in a header manipulation.

Table 61. 'insert' Child Elements:

Element	Requirement	Description
size	required	Size of inserted section
offset	required	Offset from beginning of header to the start location of the insertion.
replace	optional	If provided, specifies to override (replace) existing data at 'offset' (if 'yes'), 'no' to insert. Possible values: <ul style="list-style-type: none"> no - insert (default) yes - replace
data	required	Data to insert

5.2.4.5.1.14.18.3.2 Header Manipulation - Remove

XML element **remove** is a container for parameters necessary to configure a header remove manipulation operation. The element, if it exists can be used as a child of element **header**. There can only be one element **remove** in a header manipulation.

Table 62. 'remove' Child Elements:

Element	Requirement	Description
size	required	Size of removed section
offset	required	Offset from beginning of header to the start location of the removal.

5.2.4.5.1.14.18.3.3 Header Manipulation - Insert-Header

XML element **insert_header** is a container for parameters necessary to configure a header insert manipulation operation of an entire header (different than generic element **insert**). The element **insert_header**, if it exists, can be used as a child of element **header**. With some restrictions, there can be more than one element **insert_header** in one header manipulation

Table 63. 'insert_header' Element Attributes

Element	Requirement	Description
type	required	The type of the header inserted. Only 'mpls' is valid at this time.
header_index	optional	The header index of the header has possible values "1" and "2". The restrictions on this attribute are: <ul style="list-style-type: none"> • if the value is '2' an 'insert_header' with 'header_index' 1 must be present in the header manipulation. • a value of header_index can be used only once per header manipulation

Table 64. 'insert_header' Child Elements

Element	Requirement	Description
data	optional	The data of the header to be inserted.
replace	optional	If provided, specifies to override (replace) existing data (if 'yes'), 'no' to insert.

insert_header example:

```
<header name="insert_2_l2">
  <insert_header type="mpls" header_index="1">
    <data>0x00000048</data>
  </insert_header>
  <insert_header type="mpls" header_index="2">
    <data>0x00000048</data>
  </insert_header>
</header>
```

5.2.4.5.1.14.18.3.4 Header Manipulation - Remove_Header

XML element **remove_header** is a container for parameters necessary to configure a header remove manipulation operation of an entire header (different then element **remove** that is a generic one). The element, if it exists, can be used as a child of element **header'**. There can be only one instance of element **remove_header** in a manipulation and it cannot appear in the same time with the generic **remove**.

Table 65. 'remove_header' Child Elements

Element	Requirement	Description
type	required	The type of the header remove. Possible values: <ul style="list-style-type: none"> • "qtags" • "mpls" • "ethmpls (or "ethernet_mpls")" • "eth" (or "ethernet")"

remove_header example:

```
<header name="remove_l2">
  <remove_header type="qtags"/>
</header>
```

5.2.4.5.1.14.18.3.5 Header Manipulation - Update

XML element **update** is a container for parameters necessary to configure a header update manipulation. The element if exists can be used as a child of element **header**. There can be only one update in a header manipulation.

update Element Attributes:

Table 66. 'remove_header' Child Elements

Element	Requirement	Description
type	required	The type of the update. Possible values: <ul style="list-style-type: none"> "vlan" "ipv4" "ipv6" "tcpudp"

update Child Elements:

Table 67. 'remove_header' Child Elements

Element	Requirement	Description
field	required	Specifies the field to be updated. There must be atleast one inside an update. For some types of updates the field element can appear multiple times.

Field Element Attributes:

Table 68. 'remove_header' Child Elements

Element	Requirement	Description
type	required	<p>The type of the header remove. Possible values:</p> <ul style="list-style-type: none"> for 'vlan' <ul style="list-style-type: none"> dscp - DSCP to VLAN priority bits translation. vpri - Replace VPri of outer most VLAN tag . for 'ipv4' <ul style="list-style-type: none"> tos - update TOS with the given value. id - update IP ID with the new 16 bit given value. ttl - Decrement TTL by 1. src - update IP source address with the given value. dst - update IP destination address with the given value. for 'ipv6' <ul style="list-style-type: none"> tc - update Traffic Class address with the given value. hl - Decrement Hop Limit by 1. src - update IP source address with the given value. dst - update IP destination address with the given value. for 'tcpudp' <ul style="list-style-type: none"> checksum - update TCP/UDP checksum. src - update TCP/UDP source address with the given value. dst - update TCP/UDP destination address with the given value.
value	optional	<p>The value used for the update. It is not valid for:</p> <ul style="list-style-type: none"> hl tll checksum
fill	optional	<p>Only valid for dscp - fills the entire array with the given value. The fill is performed before the other dscp operations.</p>
index	optional	<p>Only valid for dscp. Specifies the index in the array where that value is set. The index starts from 0.</p>

'update' Example:

```

<header name="upd_checksum">
  <update type = "tcpudp">
    <field type="checksum"/>
  </update>
</header>

<header name="upd_ipv4src">
  <update type = "ipv4">
    <field type="src" value="0xC0A80101"/>
  </update>

```

```

</header>

<header name="upd_vpri">
  <update type = "vlan">
    <field type="dscp" fill="yes" value="4"/>
    <field type="dscp" index="20" value="2"/>
    <!--...-->
    <field type="dscp" index="30" value="2"/>
  </update>
</header>

```

5.2.4.5.1.14.18.3.6 Header Manipulation - Custom

XML element **custom** is a container for parameters necessary to configure custom header manipulation. The custom header manipulation supported by the drivers is now custom IP replace, and allows changing between ipv4 and ipv6.

'custom' Element Attributes

Table 69. 'custom' Element Attributes:

Element	Requirement	Description
type	required	The type of the custom header manipulation. Possible values are: <ul style="list-style-type: none"> • "ipv4byipv6" (or just "ipv4") – Replaces ipv4 by ipv6. • -"ipv6byipv4" (or just "ipv6") – Replaces ipv6 by ipv4.

'custom' Child Elements

Table 70. nextmanip Element Attributes:

Element	Requirement	Description
size	required	Size of the header to be inserted. (max is 256)
data	required	The header data to be inserted.
decttl	optional	Decrement TTL by 1 (ipv4). Possible values: <ul style="list-style-type: none"> • "yes" • "no"
dechl	optional	Decrement Hop Limit by 1 (ipv6). Possible values: <ul style="list-style-type: none"> • "yes" • "no"
ip (or 'ipid')	optional	16 bit New IP ID (ipv4)

'custom' Example:

```

<header name="custom_ex">
  <custom type="ipv6byipv4">
    <decttl>yes</decttl>
    <id>1</id>
    <size>0x20</size>
    <data>0x4500000012340000000100001011121314151617</data>
  </custom>
</header>

```

5.2.4.5.1.14.18.3.7 Header Manipulation - Nextmanip

XML element **nextmanip** Can be used to setup cascading header manipulations. It relates to the header manipulation element and not sub-elements (insert, remove and update).

Table 71. Nextmanip element attributes

Element	Requirement	Description
name	required	The name of the next header manipulation

5.2.4.5.1.14.18.3.8 Header Manipulation - Example

Here is a general example of possible header manipulation definition:

```

<manipulations>
  <header name="ins_rmv" parse="yes">
    <insert>
      <size>14</size>
      <offset>0</offset>
      <data>0x0102030405061112131415168100</data>
    </insert>
    <remove>
      <size>14</size>
      <offset>0</offset>
    </remove>
  </header>

  <header name="vpri_update">
    <update type="vlan">
      <field type="vpri" fill="yes" value="0"/>
    </update>
  </header>

  <header name="ins_vlan" parse="no">
    <insert>
      <size>4</size>
      <offset>12</offset>
      <data>0x81004416</data>
    </insert>
    <nextmanip name="vpri_update"/>
  </header>
</manipulations>

<classification name="clsf_1" max="0" masks="yes" statistics="none">
  <key>
    <fieldref name="ethernet.type"/>
  </key>
  <entry>
    <data>0x8847</data>
    <queue base="0x01"/>
    <action type="policer" name="plcr_1"/>
    <header name="ins_vlan"/>
  </entry>
  <entry>
    <data>0x8848</data>
    <queue base="0x02"/>
    <header name="ins_rmv"/>
  </entry>
</classification>

```

```
</entry>
</classification>
```

5.2.4.5.15 Standard Protocol File - Excerpt

The DPAA SDK includes a file called the Standard Protocol file. This file uses the NetPDL (Network Protocol Description Language) XML dialect to define the fields in each standard protocol header that the FMan can parse with its Hard Parser. In addition, for each protocol, the NetPDL statement define the actions the Hard Parser should take upon encountering this protocol header in the frame window.

For this reason, the SDK includes a copy of the Standard Protocol file here: `/etc/fmc/config/hxs_pdl_v3.xml`. In addition, to give you an idea what the file is like, a small portion is shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<netpdl name="nbee.org NetPDL Database"
  version="0.2" creator="nbee.org" date="28-05-2008">
  <!-- This file is for reference only. -->
  <!-- It describes the protocols and fields supported by the FMan's Hard Parser-->

  <!--
  NetPDL description of the Ethernet Protocol
  -->
  <protocol name="ethernet" longname="Ethernet 802.3"
    comment="Ethernet DIX has been included in 802.3" showsumtemplate="ethernet">

    <execute-code>
      <!-- If we're on Ethernet IEEE 802.3, update the packet length -->
      <after when="buf2int(type) le 1500">
        <assign-variable name="$packetlength" value="buf2int(type) + 14"/>
        <!-- 14 is the size of the ethernet header -->
      </after>
    </execute-code>

    <format>
      <fields>
        <field type="fixed" name="dst" longname="MAC Destination" size="6"
          showtemplate="MACaddressEth"/>
        <field type="fixed" name="src" longname="MAC Source" size="6"
          showtemplate="MACaddressEth"/>
        <field type="fixed" name="type" longname="Ethertype - Length" size="2"
        </fields>
      </format>

    <encapsulation>
      <!-- We have four possible encapsulations for IPX:
      - Ethernet version II
        ==> type= 0x8137
      - Novell-specific framing (raw 802.3)
        ==> directly in Ethernet; check that IPX checksum is == 0xFFFF
      - Ethernet 802.3/802.2 without SNAP
        ==> directly in SNAP; check that IPX checksum is == 0xFFFF (after SNAP hdr)
      - Ethernet 802.3/802.2 with SNAP
        ==> type= 0x8137 (in SNAP)
      See the "IPX Ethernet and FDDI Encapsulation Methods" Cisco doc, at:
      http://www.cisco.com/en/US/tech/tk389/tk224/
      technologies_q_and_a_item09186a0080093d2e.shtml
      -->
      <if expr="buf2int($packet[$currentoffset:2]) == 0xFFFF">
        <if-true>
```

```

    <nextproto proto="#ipx"/>
  </if-true>
</if>
<switch expr="buf2int(type)">
  <case value="0" maxvalue="1500"> <nextproto proto="#llc"/> </case>
  <case value="0x800"> <nextproto proto="#ip"/> </case>
  <case value="0x806"> <nextproto proto="#arp"/> </case>
  <case value="0x8863"> <nextproto proto="#pppoed"/> </case>
  <case value="0x8864"> <nextproto proto="#pppoe"/> </case>
  <case value="0x86DD"> <nextproto proto="#ipv6"/> </case>
  <case value="0x8100"> <nextproto proto="#vlan"/> </case>
  <case value="0x8137"> <nextproto proto="#ipx"/> </case>
  <case value="0x81FD"> <nextproto proto="#ismp"/> </case>
  <case value="0x8847" comment="mpls-unicast">
    <nextproto proto="#mpls"/>
  </case>
  <case value="0x8848" comment="mpls-multicast">
    <nextproto proto="#mpls"/>
  </case>
</switch>
</encapsulation>

<visualization>
  <showsumtemplate name="ethernet">
    <section name="next"/>
    <text value="Eth: "/>
    <protofield name="src" showdata="showvalue"/>
    <text value=" => "/>
    <protofield name="dst" showdata="showvalue"/>
  </showsumtemplate>
</visualization>

</protocol> <!-- End Ethernet protocol definition -->

<!--
NetPDL description of the VLAN Protocol
-->
<protocol name="vlan" longname="Virtual LAN (802.3ac)" showsumtemplate="vlan">
  <format>
    <fields>
      <block name="vlan" size="2" longname="Tag Control Information">
        <field type="bit" name="pri" longname="User Priority"
          mask="0xE000" size="2" showtemplate="FieldHex"/>
        <field type="bit" name="cfi" longname="CFI"
          mask="0x1000" size="2" showtemplate="FieldDec"/>
        <field type="bit" name="vlanid" longname="VLAN ID"
          mask="0x0FFF" size="2" showtemplate="FieldDec"/>
      </block>
      <field type="fixed" name="type" longname="Ethertype - Length"
        size="2" showtemplate="eth.type.length"/>
    </fields>
  </format>

  <encapsulation>
    <switch expr="buf2int(type)">
      <case value="0" maxvalue="1500"> <nextproto proto="#llc"/> </case>
      <case value="0x800"> <nextproto proto="#ip"/> </case>
      <case value="0x806"> <nextproto proto="#arp"/> </case>
      <case value="0x8863"> <nextproto proto="#pppoed"/> </case>
      <case value="0x8864"> <nextproto proto="#pppoe"/> </case>
    </switch>
  </encapsulation>

```

```

        <case value="0x86DD"> <nextproto proto="#ipv6"/> </case>
    </switch>
</encapsulation>

<visualization>
    <showsumtemplate name="vlan">
        <text value=" (VLAN-ID "/>
            <protofield name="vlanid" showdata="showvalue"/>
        <text value=")"/>
    </showsumtemplate>
</visualization>

</protocol> <!-- End VLAN protocol definition -->

<!-- snip - code removed ... -->

<!--
NetPDL description of the IPv6 Protocol
-->
<protocol name="ipv6" longname="IPv6 (Internet Protocol version 6)
showsumtemplate="ipv6">
    <!-- We should check that 'version' is equal to '6' -->
    <execute-code>
        <after>
            <!-- Store ipsrc and ipdst in a couple of variables for the sake of speed -->
            <!-- Hids differences between IPv4 and IPv6 for session tracking -->
            <assign-variable name="$ipsrc" value="src"/>
            <assign-variable name="$ipdst" value="dst"/>
            <if expr="$ipsrc lt $ipdst" >
                <if-true>
                    <assign-variable name="$firstip" value="src"/>
                    <assign-variable name="$secondip" value="dst"/>
                </if-true>
                <if-false>
                    <assign-variable name="$firstip" value="dst"/>
                    <assign-variable name="$secondip" value="src"/>
                </if-false>
            </if>
        </after>
    </execute-code>

    <format>
        <fields>
            <field type="bit" name="ver" longname="Version"
                mask="0xF0000000" size="4" showtemplate="FieldDec"/>
            <field type="bit" name="tos" longname="Type of service"
                mask="0x0F000000" size="4" showtemplate="FieldHex"/>
            <field type="bit" name="flabel" longname="Flow label"
                mask="0x00FFFFFF" size="4" showtemplate="FieldHex"/>
            <field type="fixed" name="plen" longname="Payload Length"
                size="2" showtemplate="FieldDec"/>
            <field type="fixed" name="nexthdr" longname="Next Header"
                size="1" showtemplate="ipv6.nexthdr"/>
            <field type="fixed" name="hop" longname="Hop limit"
                size="1" showtemplate="FieldDec"/>
            <field type="fixed" name="src" longname="Source address"
                size="16" showtemplate="ip6addr"/>
            <field type="fixed" name="dst" longname="Destination address"
                size="16" showtemplate="ip6addr"/>

```

```
<loop type="while" expr="1">
  <!-- Loop until we find a 'break' -->
  <switch expr="buf2int(nexthdr)">
    <case value="0">
      <includeblk name="HBH"/>
    </case>
    <case value="43">
      <includeblk name="RH"/>
    </case>
    <case value="44">
      <includeblk name="FH"/>
    </case>
    <case value="51">
      <includeblk name="AH"/>
    </case>
    <case value="60">
      <includeblk name="DOH"/>
    </case>
    <default>
      <loopctrl type="break"/>
    </default>
  </switch>
</loop>
</fields>

<block name="HBH" longname="Hop By Hop Option">
  <field type="fixed" name="nexthdr" longname="Next Header"
    size="1" showtemplate="ipv6.nexthdr"/>
  <field type="fixed" name="helen"
    longname="Length (multiple of 8 bytes, not including first 8)"
    size="1" showtemplate="ipv6.hbhlen"/>
  <loop type="size" expr="(buf2int(helen) * 8) + 6">
    <!-- '6' because the first two bytes are nexthdr and helen -->
    <includeblk name="Option"/>
  </loop>
</block>

<block name="FH" longname="Fragment Header">
  <field type="fixed" name="nexthdr" longname="Next Header"
    size="1" showtemplate="ipv6.nexthdr"/>
  <field type="fixed" name="reserved"
    longname="Reserved (multiple of 8 bytes)"
    comment="This is in multiple of 8 bytes"
    size="1" showtemplate="FieldDec"/>
  <field type="bit" name="fragment offset" longname="Fragment Offset"
    mask="0xFFF0" size="2" showtemplate="FieldDec"/>
  <field type="bit" name="res" longname="Res"
    mask="0x0004" size="2" showtemplate="FieldHex"/>
  <field type="bit" name="m" longname="M"
    mask="0x0001" size="2" showtemplate="FieldBin"/>
  <field type="fixed" name="identification"
    longname="Identification" size="4" showtemplate="FieldDec"/>
</block>

<block name="AH" longname="Authentication Header">
  <field type="fixed" name="nexthdr" longname="Next Header"
    size="1" showtemplate="ipv6.nexthdr"/>
  <field type="fixed" name="payload len" longname="Payload Len"
    size="1" showtemplate="FieldDec"/>
  <field type="fixed" name="reserved" longname="Reserved"
```

```

    size="2" showtemplate="FieldDec"/>
<field type="fixed" name="spi" longname="Security Parameters Index"
    size="4" showtemplate="FieldDec"/>
<field type="fixed" name="snf" longname="Sequence Number Field"
    size="4" showtemplate="FieldDec"/>
</block>

<block name="DOH" longname="Destination Option Header">
<field type="fixed" name="nexthdr" longname="Next Header"
    size="1" showtemplate="ipv6.nexthdr"/>
<field type="fixed" name="hlen"
    longname="Length (multiple of 8 bytes, not including first 8)"
    size="1" showtemplate="ipv6.hbhlen"/>
<loop type="size" expr="(buf2int(helen) * 8)+6">
    <!-- '6' because the first two bytes are nexthdr and helen -->
    <includeblk name="Option"/>
</loop>
</block>

<block name="RH" longname="Routing Header">
<field type="fixed" name="nexthdr" longname="Next Header"
    size="1" showtemplate="ipv6.nexthdr"/>
<field type="fixed" name="hlen"
    longname="Length (multiple of 8 bytes)"
    comment="This is in multiple of 8 bytes"
    size="1" showtemplate="FieldDec"/>
<field type="fixed" name="rtype" longname="Routing Type"
    size="1" showtemplate="FieldDec"/>
<field type="fixed" name="segment left" longname="Segment Left"
    size="1" showtemplate="FieldDec"/>
<field type="variable" name="tsd" longname="Type Specific Data"
    expr="buf2int(hlen)" showtemplate="Field4BytesHex"/>
</block>

<block name="Option" longname="Option">
<field type="fixed" name="opttype" longname="Option Type"
    size="1" showtemplate="ipv6.opttype">
<field type="bit" name="act"
    longname="Action (action if Option Type is unrecognized)" mask="0xC0"
    size="1" showtemplate="ipv6.optact"/>
<field type="bit" name="chg"
    longname="Change(whether or not option data can change while packet en-route)"
    mask="0x20" size="1" showtemplate="ipv6.optchg"/>
<field type="bit" name="res" longname="Option Code" mask="0x1F"
    size="1" showtemplate="FieldDec"/>
</field>

<switch expr="buf2int(opttype)">
    <case value="0">
        <!-- No fields are present if the option is not 'Pad1'-->
    </case>
    <case value="5"><!-- Router Alert -->
        <field type="fixed" name="optlen" longname="Option Length"
            size="1" showtemplate="FieldDec"/>
        <field type="fixed" name="value" size="2" longname="Option Value"
            showtemplate="ipv6.optroutalert"/>
    </case>
    <default>
        <field type="fixed" name="optlen" longname="Option Length"
            size="1" showtemplate="FieldDec"/>

```

```
        <field type="variable" name="optval" longname="Option Value"
            expr="buf2int(optlen)" showtemplate="Field4BytesHex"/>
    </default>
</switch>
</block>
</format>

<encapsulation>
    <switch expr="buf2int(nextthdr)">
        <case value="4"> <nextproto proto="#ip"/> </case>
        <case value="6"> <nextproto proto="#tcp"/> </case>
        <case value="17"> <nextproto proto="#udp"/> </case>
        <!-- <case value="29"> <nextproto proto="#TP4"/> </case> -->
        <!-- <case value="45"> <nextproto proto="#IDRP"/> </case> -->
        <case value="50"> <nextproto proto="#ipsec_esp"/> </case>
        <case value="51"> <nextproto proto="#ipsec_ah"/> </case>
        <case value="58"> <nextproto proto="#icmp6"/> </case>
        <case value="89"> <nextproto proto="#ospf6"/> </case>
        <case value="103"> <nextproto proto="#pim6"/> </case>
    </switch>
</encapsulation>

<visualization>
    <showtemplate name="ipv6.nextthdr" showtype="dec">
        <showmap>
            <switch expr="buf2int(this)">
                <case value="0" how="Hop By Hop Option Header"/>
                <case value="43" show="Fragment Header"/>
                <case value="44" show="Authentication Header"/>
                <case value="51" show="Destination Option Header"/>
                <case value="60" show="Routing Header"/>
                <case value="50" show="Encapsulating Security Payload"/>
                <case value="58" show="Internet Control Message Protocol (ICMPv6)"/>
                <case value="59" show="No next Header"/>
                <default show="Upper Layer Header"/>
            </switch>
        </showmap>
    </showtemplate>

    <showtemplate name="ipv6.opttype" showtype="hex">
        <showmap>
            <switch expr="buf2int(this)">
                <case value="0" show="Pad1 Option"/>
                <case value="1" show="PadN Option"/>
                <case value="5" show="Router Alert Option"/>
                <default show="Error in IPv6 Option Type lookup"/>
            </switch>
        </showmap>
    </showtemplate>

    <showtemplate name="ipv6.optact" showtype="bin">
        <showmap>
            <switch expr="buf2int(this)">
                <case value="0" show="Skip over option"/>
                <case value="1" show="Discard packet silently"/>
                <case value="2" show="Discard packet-send ICMP"/>
                <case value="3" show="Discard packet-send ICMP if packet was unicast"/>
                <default show="Error in IPv6 Option Action lookup"/>
            </switch>
        </showmap>
    </showtemplate>
</visualization>
```

```

</showtemplate>

<showtemplate name="ipv6.optchg" showtype="bin">
  <showmap>
    <switch expr="buf2int(this)">
      <case value="0" show="Option data does not change en-route"/>
      <case value="1" show="Option data may change en-route"/>
      <default show="Error in IPv6 Option Change lookup"/>
    </switch>
  </showmap>
</showtemplate>

<showtemplate name="ipv6.optroutalert" showtype="dec">
  <showmap>
    <switch expr="buf2int(this)">
      <case value="0" show="Datagram contains Multicast Listener Disc msg"/>
      <case value="1" show="Datagram contains RSVP message"/>
      <case value="2" show="Datagram contains an Active Networks msg"/>
      <default show="Error in IPv6 Router Alert Option lookup"/>
    </switch>
  </showmap>
</showtemplate>

<!-- Length of the hop by hop option header -->
<showtemplate name="ipv6.hbhlen" showtype="dec">
  <showdtl>
    <text expr="(buf2int(this) * 8) + 8"/>
    <text value=" (field value = "/>
    <protofield showdata="showvalue"/>
    <text value=")"/>
  </showdtl>
</showtemplate>

<showsumtemplate name="ipv6">
  <if expr="($prevproto == #ip) or ($prevproto == #ipv6) or
    ($prevproto == #ppp) or ($prevproto == #pppoe) or
    ($prevproto == #gre)">
    <if-true>
      <text value=" - "/>
    </if-true>
    <if-false>
      <section name="next"/>
    </if-false>
  </if>

  <text value="IPv6: "/>
  <protofield name="src" showdata="showvalue"/>
  <text value=" => "/>
  <protofield name="dst" showdata="showvalue"/>
  <text value=" (Len " expr="buf2int(plen) + 40"/>
  <text value=")"/>
</showsumtemplate>
</visualization>
</protocol> <!-- End IPv6 definition -->

<!-- snip - code removed ... -->

</netpdl>

```

```
<!-- End of Standard Protocol file -->
```

5.2.4.5.1.16 Custom protocol file - examples

5.2.4.5.1.16.1 Custom Protocol File - GTP Protocol

The following "GTP_example.xml" file describes the custom GTP protocol.

```
<?xml version="1.0" encoding="utf-8"?>
<netpdl name="GTP" description="GTP-U Example">
  <!-- Gtpu program is an extension to the udp hard shell -->
  <protocol name="gtpu" longname="GTP-U" prevproto="udp">
    <!-- fields in GTP header used for validation and calculating length -->
    <format>
      <fields>
        <field type="bit"      name="flags"      mask="0xE0" size="1" />
        <field type="bit"      name="pt"         mask="0x80" size="1" />
        <field type="bit"      name="version"    mask="0x07" size="1" />
        <field type="fixed"    name="mtype"      size="1" longname="message type"/>
        <field type="fixed"    name="length"     size="2" />
        <field type="fixed"    name="teid"      size="4" />
        <field type="fixed"    name="snum "     size="2" longname="sequence number"/>
        <field type="fixed"    name="npdunum"   size="1" longname="N-PDU number"/>
        <field type="fixed"    name="next"      size="1" longname="Next ext header type"/>
      </fields>
    </format>

    <execute-code>
      <!-- Check that UDP port is 2152 -->
      <before confirm="yes">
        <if expr="udp.dport == 2152">
          <if-true>
            </if-true>
          <if-false>
            <!-- Confirms UDP layer and exits-->
            <action type="exit" confirm="yes" advance="no" nextproto="return"/>
          </if-false>
        </if>
      </before>

      <!-- Done after UDP layer is confirmed-->
      <!-- Check version and calculate length-->
      <after confirm="no">
        <if expr="version == 1">
          <if-true>
            <assign-variable name="$shimoffset_1" value="$NxtHdrOffset"/>
          </if-true>
          <if-false>
            <assign-variable name="$ShimR" value="0x23"/>
            <action type="exit" confirm="no" confirmcustom="no" nextproto="none"/>
          </if-false>
        </if>

        <if expr="flags != 0">
          <if-true>
            <assign-variable name="$NxtHdrOffset" value="$shimoffset_1+12"/>
          </if-true>
          <if-false>
```

```

    <assign-variable name="$NxtHdrOffset" value="$shimoffset_1+8"/>
  </if-false>
</if>
  <action type="exit" confirm="no" confirmcustom="shim1" nextproto="none"/>
</after>
</execute-code>
</protocol>
</netpdl>

```

5.2.5 Security Engine (SEC)

5.2.5.1 SEC Device Driver User Manual

Introduction and Terminology

The Linux kernel contains a Scatterlist CryptoAPI driver for the NXP SEC v4.x, v5.x, v6.x security hardware blocks.

It integrates seamlessly with in-kernel crypto users, such as IPSec, such that any IPSec suite that configures IPSec tunnels with the kernel will automatically use the hardware to do the crypto.

SEC v5.x is backward compatible with SEC v4.x hardware, so one can assume that subsequent SEC v4.x references include SEC v5.x hardware, unless explicitly mentioned otherwise.

The name of the software driver module for SEC v4.x hardware is 'caam', after its internal block name: Cryptographic Accelerator and Assurance Module.

Table 72.

SEC hardware version	driver name
v4.x, v5.x, v6.x	caam

Module Loading

CAAM NXP Security Engine device drivers support either kernel built-in or module.

Kernel Configuration

The designated driver should be configured in the kernel by default for the target platform. If unsure, check CONFIG_CRYPTO_DEV_FSL_CAAM is set under "Cryptographic API" -> "Hardware crypto devices" in the kernel configuration:

Kernel Configure Tree View Options	Description
<pre> Cryptographic API ---> [*] Hardware crypto devices ---> <*> Freescale CAAM-Multicore driver backend (SEC) <*> Freescale CAAM Job Ring driver backend (SEC) (9) Job Ring size [] Job Ring interrupt coalescing <*> Register algorithm implementations with the Crypto API <*> Queue Interface as Crypto API backend <*> Public Key Cryptography Support in CAAM driver <*> Register hash algorithm implementations with the Crypto API </pre>	Enable CAAM device driver

Table continues on the next page...

Table continued from the previous page...

Kernel Configure Tree View Options	Description
<pre><*> Register caam device for hwrng API [] Enable debug output in CAAM driver</pre>	
<pre>Network support ---> Network option ---> <*> PF_KEY sockets <*> IP: AH transformation <*> IP: ESP transformation <*> IP: IPComp transformation <*> IP: IPsec transport mode <*> IP: IPsec tunnel mode</pre>	<p>IPsec support, of course the TCP/IP networking option should be enabled</p>

CAAM Driver specifics

The CAAM driver module implements and utilizes two interfaces:

- a job ring interface (JRI) for all crypto API service requests
- (on DPAA 1.x platforms) a queue interface (QI) for AEAD algorithms-based crypto API service requests

The Linux driver automatically sets the enable bit for the SEC hardware's Queue Interface (QI), depending on QI feature availability in the hardware. This enables the hardware to also operate as a DPAA component for use by e.g., USDPAA apps. This behaviour does not conflict with normal in-kernel job ring operation, other than the potential performance-observable effects of internal SEC hardware resource contention, and vice-versa.

Note the CAAM driver has sub-configuration settings, most notably hardware job ring size and interrupt coalescing. They can be used to fine-tune performance for a particular application.

The first item enables the basic Controller Driver, and the Job Ring backend. All suboptions are dependent on this.

The second item ("Job Ring Size") allows the user to select the size of the hardware job rings; if requests arrive at the driver enqueue entry point in a bursty nature, the bursts' maximum length can be approximated etc. One can set the greatest burst length to save performance and memory consumption.

The third item ("Job Ring interrupt coalescing") allows the user to select the use of the hardware's interrupt coalescing feature. Note that the driver software already performs IRQ coalescing, and zero-loss benchmarks have in fact produced better results with this option turned off.

If selected, two additional options become effective:

- Job Ring interrupt coalescing count threshold (CRYPTO_DEV_FSL_CAAM_INTC_THLD)

Selects the value of the descriptor completion threshold, in the range 1-256. A selection of 1 effectively defeats the coalescing feature, and any selection equal or greater than the selected ring size will force timeouts for each interrupt.
- Job Ring interrupt coalescing timer threshold (CRYPTO_DEV_FSL_CAAM_INTC_TIME_THLD)

Selects the value of the completion timeout threshold in multiples of 64 SEC interface clocks, to which, if no new descriptor completions occur within this window (and at least one completed job is pending), then an interrupt will occur. This is selectable in the range 1-65535.

The options to register to Crypto API, hwrng API respectively, allow the driver to register its algorithm capabilities with the kernel's crypto API. Deselect them only if you do not want crypto API requests to be performed on the SEC; they will be done in software (on the processor core).

Hash algorithms may be individually turned off, since the nature of the application may be such that it prefers software (core) crypto latency due to many small-sized requests.

Random Number Generation (RNG) may be manually turned off in case there is an alternate source of entropy available to the kernel.

Device Tree Binding

Property	Type	Status	Description
compatible	String	Required	fsl,sec-vX.Y (preferred) OR fsl,secX.Y

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/crypto/caam/	CAAM NXP Security Engine Driver

Corresponding Device Tree node

```
crypto@30000 {
    compatible = "fsl,sec-v4.0";
    fsl,sec-era = <2>;
    #address-cells = <1>;
    #size-cells = <1>;
    reg = <0x300000 0x10000>;
    ranges = <0 0x300000 0x10000>;
    interrupt-parent = <&mpic>;
    interrupts = <92 2>;
    clocks = <&clks IMX6QDL_CLK_CAAM_MEM>,
            <&clks IMX6QDL_CLK_CAAM_ACLK>,
            <&clks IMX6QDL_CLK_CAAM_IPG>,
            <&clks IMX6QDL_CLK_EIM_SLOW>;
    clock-names = "mem", "aclk", "ipg", "emi_slow";
};
```

NOTE

See [linux/Documentation/devicetree/bindings/crypto/fsl-sec{4,6}.txt](#) for more info.

How to test the driver

To test the driver, in the kernel configuration menu, under "Cryptographic API" -> "Cryptographic algorithm manager", ensure that run-time self-tests are not disabled, i.e. the "Disable run-time self tests" (CONFIG_CRYPTO_MANAGER_DISABLE_TESTS) entry is not set. This will run standard test vectors against the driver after the driver registers its supported algorithms with the kernel crypto API, usually at boot-time. Then run test on the target system. Below is a snippet extracted from the boot log.

```
[...]
platform caam_qi.0: Linux CAAM Queue I/F driver initialised
caam 1700000.crypto: Entropy delay = 3200
caam 1700000.crypto: Instantiated RNG4 SH0
caam 1700000.crypto: Instantiated RNG4 SH1
caam 1700000.crypto: device ID = 0x0a12060000000000 (Era 8)
caam 1700000.crypto: job rings = 4, qi = 1
alg: No test for authenc(hmac(sha224),ecb(cipher_null)) (authenc-hmac-sha224-ecb-cipher_null-caam)
alg: No test for authenc(hmac(sha256),ecb(cipher_null)) (authenc-hmac-sha256-ecb-cipher_null-caam)
alg: No test for authenc(hmac(sha384),ecb(cipher_null)) (authenc-hmac-sha384-ecb-cipher_null-caam)
```

```
alg: No test for authenc(hmac(sha512),ecb(cipher_null)) (authenc-hmac-sha512-ecb-cipher_null-caam)
alg: No test for authenc(hmac(md5),cbc(aes)) (authenc-hmac-md5-cbc-aes-caam)
alg: No test for authenc(hmac(sha224),cbc(aes)) (authenc-hmac-sha224-cbc-aes-caam)
alg: No test for authenc(hmac(sha384),cbc(aes)) (authenc-hmac-sha384-cbc-aes-caam)
alg: No test for authenc(hmac(md5),cbc(des3_ede)) (authenc-hmac-md5-cbc-des3_ede-caam)
alg: No test for authenc(hmac(md5),cbc(des)) (authenc-hmac-md5-cbc-des-caam)
alg: No test for authenc(hmac(md5),rfc3686(ctr(aes))) (authenc-hmac-md5-rfc3686-ctr-aes-caam)
alg: No test for authenc(hmac(sha1),rfc3686(ctr(aes))) (authenc-hmac-sha1-rfc3686-ctr-aes-caam)
alg: No test for authenc(hmac(sha224),rfc3686(ctr(aes))) (authenc-hmac-sha224-rfc3686-ctr-aes-caam)
alg: No test for authenc(hmac(sha256),rfc3686(ctr(aes))) (authenc-hmac-sha256-rfc3686-ctr-aes-caam)
alg: No test for authenc(hmac(sha384),rfc3686(ctr(aes))) (authenc-hmac-sha384-rfc3686-ctr-aes-caam)
alg: No test for authenc(hmac(sha512),rfc3686(ctr(aes))) (authenc-hmac-sha512-rfc3686-ctr-aes-caam)
caam algorithms registered in /proc/crypto
alg: No test for authenc(hmac(md5),cbc(aes)) (authenc-hmac-md5-cbc-aes-caam-qi)
alg: No test for authenc(hmac(sha224),cbc(aes)) (authenc-hmac-sha224-cbc-aes-caam-qi)
alg: No test for authenc(hmac(sha384),cbc(aes)) (authenc-hmac-sha384-cbc-aes-caam-qi)
alg: No test for authenc(hmac(md5),cbc(des3_ede)) (authenc-hmac-md5-cbc-des3_ede-caam-qi)
alg: No test for authenc(hmac(md5),cbc(des)) (authenc-hmac-md5-cbc-des-caam-qi)
platform caam_qi.0: fsl,sec-v4.4 algorithms registered in /proc/crypto
caam_jr 1710000.jr: registering rng-caam
alg: No test for pkc(rsa) (pkc-rsa-caam)
alg: No test for pkc(dsa) (pkc-dsa-caam)
alg: No test for pkc(dh) (pkc-dh-caam)
caam 1700000.crypto: fsl,sec-v4.4 algorithms registered in /proc/crypto
[...]
```

Verifying driver operation and correctness

Other than noting the performance advantages due to the crypto offload, one can also ensure the hardware is doing the crypto by looking for driver messages in dmesg

The driver emits console messages at initialization time:

```
platform caam_qi.0: fsl,sec-v4.4 algorithms registered in /proc/crypto
caam 1700000.crypto: fsl,sec-v4.4 algorithms registered in /proc/crypto
```

If the messages are not present in the logs, either the driver is not configured in the kernel, or no SEC compatible device tree node is present in the device tree.

Incrementing IRQs in /proc/interrupts

Given a time period when crypto requests are being made, the SEC hardware will fire completion notification interrupts - either on the corresponding Job Ring or on the QMan (Queue Manager) portal IRQ (depending on what CAAM interface is the algorithm using):

```
cat /proc/interrupts | grep jr
CPU0          CPU1          CPU2          CPU3          CPU4          CPU5          CPU6
CPU7
[...]
```

IRQ	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6
88:	0	0	0	941	0	0	
0	0	OpenPIC	88 Level	ffe301000.jr			
89:	0	0	0	0	0	0	
0	0	OpenPIC	89 Level	ffe302000.jr			
90:	0	0	0	0	0	0	
0	0	OpenPIC	90 Level	ffe303000.jr			
91:	0	0	0	0	0	0	
0	0	OpenPIC	91 Level	ffe304000.jr			

```
cat /proc/interrupts | grep QMan
```

	CPU0	CPU1	CPU2	CPU3	CPU4	CPU5	CPU6
CPU7							
[...]							
108:	0	0	0	0	0	0	
0	0	OpenPIC	108 Level	QMan portal	7		
110:	0	0	0	0	0	0	
0	0	OpenPIC	110 Level	QMan portal	6		
112:	0	0	0	0	0	0	
0	0	OpenPIC	112 Level	QMan portal	5		
114:	0	0	0	0	0	0	
0	0	OpenPIC	114 Level	QMan portal	4		
116:	0	0	0	285	0	0	
0	0	OpenPIC	116 Level	QMan portal	3		
118:	0	0	0	0	0	0	
0	0	OpenPIC	118 Level	QMan portal	2		
120:	0	0	0	0	0	0	
0	0	OpenPIC	120 Level	QMan portal	1		
122:	0	0	0	0	0	0	
0	0	OpenPIC	122 Level	QMan portal	0		

If the number of interrupts fired increment, then the hardware is being used to do the crypto.

If the numbers do not increment, then first check the algorithm being exercised is supported by the driver. If the algorithm is supported, there is a possibility that the driver is in polling mode (NAPI mechanism) and the hardware statistics in debugfs (inbound / outbound bytes encrypted / protected - see below) should be monitored.

Note: When QI is used, CAAM driver might be sharing the QMan portal with other drivers in the system; meaning that the interrupt counters shown in /proc/interrupts are for all drivers sharing the portal.

Verifying the 'selftest' fields say 'passed' in /proc/crypto

An entry such as this:

```

name      : cbc(aes)
driver    : cbc-aes-caam
module    : kernel
priority  : 3000
refcnt    : 1
selftest  : passed
type      : ablkcipher
async     : yes
blocksize : 16
min keysize : 16
max keysize : 32
ivsize    : 16
geniv     : eseqiv

```

means the driver has successfully registered support for the algorithm with the kernel crypto API.

Note that although a test vector may not exist for a particular algorithm supported by the driver, the kernel will emit messages saying which algorithms weren't tested, and mark them as 'passed' anyway:

```

alg: No test for authenc(hmac(sha224),ecb(cipher_null)) (authenc-hmac-sha224-ecb-cipher_null-caam)
alg: No test for authenc(hmac(sha256),ecb(cipher_null)) (authenc-hmac-sha256-ecb-cipher_null-caam)
[...]
alg: No test for authenc(hmac(md5),cbc(aes)) (authenc-hmac-md5-cbc-aes-caam)
alg: No test for authenc(hmac(sha224),cbc(aes)) (authenc-hmac-sha224-cbc-aes-caam)
[...]
alg: No test for authenc(hmac(md5),rfc3686(ctr(aes))) (authenc-hmac-md5-rfc3686-ctr-aes-caam)
[...]

```

Examining the hardware statistics registers in debugfs

The controller driver enables a user level view of performance monitor registers located within the controller's register partition. To enable this view, CONFIG_DEBUG_FS must be enabled in the kernel's configuration. If there is no mount of debugfs performed at bootup time, then a manual mount must be performed in order to view these registers. This normally can be done with a superuser shell command of:

```
mount -t debugfs none /sys/kernel/debug
```

Once done, the user can read controller registers in /sys/kernel/debug/caam/ctl. It should be noted that debugfs will provide a decimal integer view of most accessible registers provided, with the exception of the KEK/TDSK/TKEK registers; those registers are long binary arrays, and should be filtered through a binary dump utility such as hexdump.

Specifically, the CAAM hardware statistics registers available are:

fault_addr, or FAR (Fault Address Register): - holds the value of the physical address where a read or write error occurred.

fault_detail, or FADR (Fault Address Detail Register): - holds details regarding the bus transaction where the error occurred.

fault_status, or CSTA (CAAM Status Register): - holds status information relevant to the entire CAAM block.

ib_bytes_decrypted: - holds contents of PC_IB_DECRYPT (Performance Counter Inbound Bytes Decrypted Register)

ib_bytes_validated: - holds contents of PC_IB_VALIDATED (Performance Counter Inbound Bytes Validated Register)

ib_rq_decrypted: - holds contents of PC_IB_DEC_REQ (Performance Counter Inbound Decrypt Requests Register)

kek: - holds contents of JDKEKR (Job Descriptor Key Encryption Key Register)

ob_bytes_encrypted: - holds contents of PC_OB_ENCRYPT (Performance Counter Outbound Bytes Encrypted Register)

ob_bytes_protected: - holds contents of PC_OB_PROTECT (Performance Counter Outbound Bytes Protected Register)

ob_rq_encrypted: - holds contents of PC_OB_ENC_REQ (Performance Counter Outbound Encrypt Requests Register)

rq_dequeued: - holds contents of PC_REQ_DEQ (Performance Counter Requests Dequeued Register)

tdsk: - holds contents of TDKEKR (Trusted Descriptor Key Encryption Key Register)

tkek: - holds contents of TDSKR (Trusted Descriptor Signing Key Register)

See the hardware documentation section "Performance Counter, Fault and Version ID Registers" for more information.

For extended testing process please refer to the Linux IPsec benchmark reproducibility guide.

Kernel configuration to support caam device driver

Algorithms Supported in the linux kernel scatterlist Crypto API

The linux kernel contains various users of the Scatterlist CryptoAPI, including its IPSec implementation, sometimes referred to as the NETKEY stack. The driver, after registering algorithm services with the CryptoAPI, is therefore used to process per-packet symmetric crypto requests and forward them to the SEC hardware.

Since all SEC version hardware processes requests asynchronous to the processor core, the driver registers asynchronous algorithm implementations with the crypto API: ahash, ablkcipher, and aead with CRYPTO_ALG_ASYNC set in .cra_flags.

Different combinations of hardware and driver software version support different sets of algorithms, so searching for the driver name in /proc/crypto on the desired target system will ensure the correct report of what algorithms it supports.

Authenticated Encryption with Associated Data (AEAD) Algorithms

These algorithms are used in applications where the data to be encrypted overlaps, or partially overlaps, the data to be authenticated, as is the case with the IPSec protocol.

These algorithms are implemented in the driver such that the hardware makes a single pass over the input data, and both encryption and authentication data are written out simultaneously.

The AEAD algorithms are mainly for use with IPSec ESP (however there is also support for TLS 1.0 record layer encryption).

At the time of writing, the CAAM driver currently supports offloading the following AEAD algorithms:

- "stitched" AEAD: all combinations of NULL, CBC-AES/DES/3DES-EDE, RFC3686-CTR-AES with MD-5, SHA-1,-224,-256,-384, and -512
- "true" AEAD: GCM-AES, GCM used in IPsec: RFC4543-GCM-AES and RFC4106-GCM-AES
- TLS 1.0 record layer with "stitched" CBC-AES-HMAC-SHA1

Note: Some of the algorithms register twice - one instance for running over the job ring interface and the other for running over the queue interface. The algorithms registered to run over queue interface will be preferred by the Crypto API, since they have a higher priority (CAAM_CRA_PRIORITY is 4000 for QI and 3000 for JRI).

An exhaustive list of algorithms follows:

```

authenc(hmac(md5),cbc(aes))
authenc(hmac(sha1),cbc(aes))
authenc(hmac(sha224),cbc(aes))
authenc(hmac(sha256),cbc(aes))
authenc(hmac(sha384),cbc(aes))
authenc(hmac(sha512),cbc(aes))
authenc(hmac(md5),cbc(des3_ede))
authenc(hmac(sha1),cbc(des3_ede))
authenc(hmac(sha224),cbc(des3_ede))
authenc(hmac(sha256),cbc(des3_ede))
authenc(hmac(sha384),cbc(des3_ede))
authenc(hmac(sha512),cbc(des3_ede))
authenc(hmac(md5),cbc(des))
authenc(hmac(sha1),cbc(des))
authenc(hmac(sha224),cbc(des))
authenc(hmac(sha256),cbc(des))
authenc(hmac(sha384),cbc(des))
authenc(hmac(sha512),cbc(des))
authenc(hmac(md5),rfc3686(ctr(aes)))
authenc(hmac(sha1),rfc3686(ctr(aes)))
authenc(hmac(sha224),rfc3686(ctr(aes)))
authenc(hmac(sha256),rfc3686(ctr(aes)))
authenc(hmac(sha384),rfc3686(ctr(aes)))
authenc(hmac(sha512),rfc3686(ctr(aes)))
authenc(hmac(md5),ecb(cipher_null))
authenc(hmac(sha1),ecb(cipher_null))
authenc(hmac(sha224),ecb(cipher_null))
authenc(hmac(sha256),ecb(cipher_null))
authenc(hmac(sha384),ecb(cipher_null))
authenc(hmac(sha512),ecb(cipher_null))

```

gcm(aes)
rfc4543(gcm(aes))
rfc4106(gcm(aes))
tls10(hmac(sha1),cbc(aes))

Cipher Encryption Algorithms

The CAAM driver currently supports offloading the following encryption algorithms:

cbc(aes)
cbc(des3_ede)
cbc(des)
ctr(aes)
rfc3686(ctr(aes))
xts(aes)

Authentication Algorithms

The CAAM driver's ahash support includes HMAC variants:

hmac(md5)
hmac(sha1)
hmac(sha224)
hmac(sha256)
hmac(sha384)
hmac(sha512)
md5
sha1
sha224
sha256
sha384
sha512

Asymmetric (public key) Algorithms

pkc(dh)
pkc(dsa)
pkc(rsa)

Random Number Generation

caam driver supports random number generation services via the kernel's built-in hwrng interface when implemented in hardware. To enable:

1. verify that the hardware random device file, e.g., /dev/hwrng or /dev/hwrandom exists. If it doesn't exist, make it with:

```
mknod /dev/hwrng c 10 183
```

2. verify /dev/hwrng doesn't block indefinitely and produces random data:

```
rngtest -C 1000 < /dev/hwrng
```

3. verify the kernel gets entropy:

```
rngtest -C 1000 < /dev/random
```

If it blocks, a kernel entropy supplier daemon, such as rngd, may need to be run. See `linux/Documentation/hw_random.txt` for more info.

Using the driver

Once enabled, the driver will forward kernel crypto API requests to the SEC hardware for processing.

Running IPsec

The IPsec stack built-in to the kernel (usually called NETKEY) will automatically use crypto drivers to offload the crypto to the SEC hardware. Documentation regarding how to set up an IPsec tunnel can be found in the respective open source IPsec suite packages, e.g. `strongswan.org`, `openswan`, `setkey`, etc.

Running OpenSSL

TODO: cross-reference to OpenSSL chapter/section and update/remove text below.

While not officially supported in the SDK, there are userspace interface implementations that enable offloading OpenSSL requests to the built-in kernel crypto API, and thus the SEC hardware via its respective driver. While the kernel officially supports the `AF_ALG` socket interface, various third-party `cryptodev` implementations are also available.

Here are some links to a couple of starting points:

```
http://carnivore.it/2011/04/23/openssl_-_af_alg
http://home.gna.org/cryptodev-linux/
http://ocf-linux.sourceforge.net/
```

Executing Custom Descriptors

`caam` has public descriptor submission interfaces, `drivers/crypto/caam/jr.c:caam_jr_enqueue()` and `drivers/crypto/caam/qi.c:caam_qi_enqueue()`.

`caam_jr_enqueue()`

Name

`caam_jr_enqueue` — Enqueue a job descriptor head. Returns 0 if OK, `-EBUSY` if the ring is full, `-EIO` if it cannot map the caller's descriptor.

Synopsis

```
int caam_jr_enqueue (struct device *dev, u32 *desc,
    void (*cbk) (struct device *dev, u32 *desc, u32 status, void *areq),
    void *areq);
```

Arguments

`dev`: contains the job ring device that is to process this request.

`desc`: descriptor that initiated the request, same as “`desc`” being argued to `caam_jr_enqueue`.

`cbk`: pointer to a callback function to be invoked upon completion of this request. This has the form: `callback(struct device *dev, u32 *desc, u32 stat, void *arg)`

`areq`: optional pointer to a user argument for use at callback time.

`caam_qi_enqueue()`

Name

`caam_qi_enqueue` — Enqueue a frame descriptor (FD) into a QMan frame queue. Returns 0 if OK, `-EIO` if it cannot map the caller's S/G array, `-EBUSY` if QMan driver fails to enqueue the FD for some reason.

Synopsis

```
int caam_qi_enqueue(struct device *qidev, struct caam_drv_req *req);
```

Arguments

qidev: contains the queue interface device that is to process this request.

req: pointer to the request structure the driver application should fill while submitting a job to driver, containing a callback function and its parameter, Queue Manager S/Gs for input and output, a per-context structure containing the CAAM shared descriptor, the etc.

Please refer to the source code for example usage.

Supporting Documentation

For more information see the *Linux IPSec Benchmark Reproducibility Guide* located in the following SDK directory: `sdk_documentation/pdf/Linux_IPSec_benchmark_reproducibility_guide.pdf`

The link below leads to a supplemental directory. Download the file you need from this set.

[Linux IPSec Benchmark Reproducibility Guide](#)

5.2.6 Pattern Matching Engine (PME)

5.2.6.1 PME Driver Release Notes

Description

This document describes PME software for the PME hardware block that is part of the QorIQ data path. The PME software runs on version 2.0, 2.1 and 2.2 of the PME hardware. The PME software includes the following components:

- Linux and USDPAA drivers
- Regular Expression compiler for Linux
- Stateful Rule compiler for Linux
- Pattern Matcher Manager (Linux)
- Pattern Matcher Configuration API (Linux)
- Sample Application – Scan Demo (Linux)
- Regular Expression Analyzer Tool (Linux Host only)

Linux and USDPAA Drivers

The PME driver software includes a Linux kernel driver and a PME driver library for USDPAA. The Linux driver provides an ioctl-based Linux user-space interface. The drivers' provide interfaces in Linux kernel and user-space for PME configuration, database setup, and data scanning. For USDPAA, the drivers provide a PME data scanning interface.

The drivers target the Linux and USDPAA environments. The majority of the code is shared between all environments.

The driver includes the following functionality.

PME Configuration interface

The PME configuration interface is an encapsulation of the PME CCSR register space and the global/error interrupt source. This is expected to be managed only by (and visible to) a control-plane operating system,

PME User-space Interface

The user-space interface provides `pme_scan` and `pme_db` devices for sending data for scanning and PME database setup respectively to the PME. There is also a `sysfs` interface to control PME global configuration parameters and to access PME statistics.

PME Low-level Driver Interface

The low-level API is a hardware abstraction layer (HAL) where the users have complete control over the data structures and interfaces used to communicate with the PME.

PME High-level Driver Interface

The PME high-level APIs provide a call-back based interface to the PME. The driver provides APIs to manage flow context and issue PMTCC and scan commands. The high-level driver internally co-ordinates commands to the PME and corresponding results from the PME.

PME Pattern Management Software

The Pattern Management software is used to configure the Pattern Matcher by enabling the addition, deletion and querying of patterns and stateful rules. The Pattern Management software components are summarized below.

Regex Compiler

Regular expressions (regexes) provide a powerful way of representing complex patterns. The regex compiler accepts a regex string with options or a file containing regexes, and converts these regexes into patterns stored in Pattern Matcher hardware-specific format, referred to as “test lines”. The regex compiler is provided as source code and as an executable program with command line interfaces. It is also available as a “C” library. The regex compiler is provided for both the x86 and PowerPC targets.

Stateful Rule Compiler

The stateful rule compiler accepts a file containing one or more stateful rules and converts the rules into Pattern Matcher hardware-specific format, referred to as “stateful rule reactions.” If stateful rule reactions are loaded for a pattern, these reactions are executed when that pattern matches. The regex compiler is provided as source code and as an executable program with command line interfaces. It is also available as a “C” library. The regex compiler is provided for both the x86 and PowerPC targets.

Linker-Loader

The linker-loader accepts patterns encoded in the hardware-specific format (test lines) and stateful rules converted into the hardware-specific format (stateful rule reactions) and generates a hardware-optimal pattern and stateful rule database (referred as the Pattern Matcher database). The linker-loader is available as source code and as a C library.

Pattern Matcher Manager

The Pattern Matcher manager (PMM) is an application that manages the creation and distribution of patterns. It uses NXP-provided libraries, namely the regex compiler, the stateful rule compiler and the linker-loader in order to compile, link and load patterns into the Pattern Matcher. PMM maintains a shadow database that allows it to dynamically add or delete discrete patterns from the PME database while the PME remains in-service. The PMM application is implemented as a Linux user-space process that runs on the QorIQ processor containing the Pattern Matcher.

Pattern Matcher Configuration API

The PMC provides a programming interface to Pattern Matcher Management applications for adding, removing, querying and committing regular expressions and stateful rules into Pattern Matcher hardware.

Regular Expression Analyzer Tool (Linux Host only)

The `regex_analyzer` tool is designed to assist users in getting the best performance out of the Pattern Matcher Engine (PME). It does this by analyzing regular expressions meant for PME hardware use and flagging the ones that may cause performance issues. This tool is available only on the Linux Host.

Software configuration, build information

The PME software is built when supporting platform is selected. The build is via the standard Yocto build process. The user-space tools and applications are available in the `bin_powerpc` directory after a build. The Linux driver is statically linked into the kernel or can be a compiled as a kernel module. A portion of the driver is always built into the kernel - this part allocates a contiguous chunk of memory at boot time that is needed by the PME.

PME Configuration Options

PME Kernel Configure Options	Description
CONFIG_FSL_PME2	Required to build the PME driver
CONFIG_FSL_PME2_CTRL	Compiles device support for the NXP PME2 pattern matching part contained in datapath-enabled SoCs (i.e. accessed via Qman and Bman portal functionality). At least one guest operating system must have this driver support, together with the appropriate device-tree entry, for PME2 functionality to be available. It is responsible for allocating system memory to the device and configuring it for operation. For this reason, it must be built into the kernel and will initialise during early kernel boot.
CONFIG_FSL_PME2_PDSRSIZE	Select the default size of the Pattern Description and Stateful Rule table as a number of 128 byte entries. This only takes effect if the device tree node doesn't have the 'fsl,pme-pdsr' property.
CONFIG_FSL_PME2_SRESIZE	Select the default size of the SRE Context Table as the number of 32 byte entries. This only takes effect if the device tree node doesn't have the 'fsl,pme-sre' property.
CONFIG_FSL_SRE_AIM	Select the alternate inconclusive match mode treatment.
CONFIG_PME2_SRE_ESR	Select if an End of SUI will produce a Simple End of SUI report.
CONFIG_FSL_PME2_SRE_CTX_SIZE_PER_SESSION	Select the default SRE Context Size per Session
CONFIG_FSL_PME2_SRE_CNR	Configured the number of stateful rules as a multiple of 256
CONFIG_FSL_SRE_MAX_INSTRUCTION_LIMIT	Select the maximum number of SRE instructions to be executed per reaction.
CONFIG_FSL_PME2_SRE_MAX_BLOCK_NUMBER	Select the maximum number of reaction head blocks to be traversed per pattern match event (e.g. a matched pattern or an End of SUI event).
CONFIG_FSL_PME2_PORTAL	This compiles I/O support for the NXP PME2 pattern matching part contained in datapath-enabled SoCs (i.e. accessed via Qman and Bman portal functionality).
CONFIG_FSL_PME2_HIGH	This compiles the high-level driver for PME2.
CONFIG_FSL_PME2_TEST_HIGH	This uses the high-level Qman driver (and the cpu-affine portals it manages) to perform high-level PME2 API testing with it.

Table continues on the next page...

Table continued from the previous page...

PME Kernel Configure Options	Description
CONFIG_FSL_PME2_TEST_SCAN	This uses the high-level Qman driver (and the cpu-affine portals it manages) to perform PME2 scan API testing with it.
CONFIG_FSL_PME2_TEST_SCAN_WITH_BPID	This performs the PME scan API test using buffers from the specified buffer pool.
CONFIG_FSL_PME2_TEST_SCAN_WITH_BPID_SIZE	This performs the PME scan API test using a buffer pool with the specified size buffers.
CONFIG_FSL_PME2_DB	This compiles the database driver software for PME2. This provides APIs to update the PME2 database.
CONFIG_FSL_PME2_DB_QOSOUT_PRIORITY	The PME DB has a scheduled output frame queue. The QoS priority level for the FQ is configurable via this option.
CONFIG_FSL_PME2_SCAN	This compiles the scan driver software for PME2. This provides APIs for sending scan data to the PME and receiving scan results.
CONFIG_FSL_PME2_SCAN_DEBUG	Trace the PME2 scan driver software with more verbosity using this option.
PME2_STAT_ACCUMULATOR_UPDATE_INTERVAL	The PME statistics accumulator periodically reads current device statistics and adds them to running counters. The frequency of these updates can be controlled by this option.
CONFIG_FSL_PME_BUG_4K_SCAN_REV_2_1_4	Workaround for errata in PME version 2.1.4. Prevents scans of SUIs greater than 4095 - 127 bytes when this revision of HW is detected.

Source Files and Tools Binaries

PME software exists in Linux user and kernel space. All PME software, including tools (like the regex compiler) and Linux driver, are provided in source form. The list below shows the location of all PME files in the release.

Linux User-space

Source Files	Description
pme_tools/include/*.h	The PME Tools header files. example.h is a sample file to demonstrate the build flow for PME tools. See pme_tools/example for details.
pme_tools/applications/*.c	Source code for PME tools like pmm, regex and stateful rule compilers.
pme_tools/pmConfiguration/*	Pattern Matcher Configuration (PMC) api source code
pme_tools/controlInterface/*	The control interface source and compiled files.
pme_tools/loaderAgent/*	The loader agent source and compiled files.
pme_tools/bin_powerpc/*	Compiled PME tools, Scan demo and Test executables.
pme_tools/lib_powerpc/*	Compiled PME tools libraries.
pme_tools/ltib_supp/*	Files that are installed in the ramdisk by Yocto

Table continues on the next page...

Table continued from the previous page...

Source Files	Description
pme_tools/common/*	Common software utilities used by PME Tools.
pme_tools/example/*	Sample use of PME Tools build and include file hierarchy.

Linux Kernel-space

Source Files	Description
drivers/staging/fsl_pme2/pme2_*.*	The PME2 drivers including user-space PME2 database and scan drivers, and kernel-space high and low-level drivers (except pme2_sample*, pme2_test* which are PME kernel test files).
include/linux/fsl_pme.h	The PME driver APIs

USDPAA

Source Files	Description
include/usdpaa/fsl_pme.h	The PME driver APIs
drivers/pme/*	The PME driver
lib_powerpc	USDPAA static libraries

Test Procedure

The PME software includes tests that are run from the Linux User-space as well as tests that run in the Linux kernel. The kernel tests can be configured via Kconfig as noted in the configuration options described above. These tests complement the PME unit tests.

PME Kernel Tests

The output of the PME tests is shown in the following excerpts. The tests are shown below as being statically linked into the Linux kernel. However, these tests could also have been built and run as loadable kernel modules.

PME High Level Test output:

```
PME2: high-level test starting
PME2: pme_ctx_init done
PME2: pme_ctx_enable done
PME2: pme_ctx_ctrl_update_flow done
pme2_test_high: ctrl_cb() invoked, fd;!
f10791d0: 0100 0000 2f98 93a0 0000 0020 0004 0000
PME2: pme_ctx_ctrl_read_flow done
pme2_test_high: ctrl_cb() invoked, fd;!
f1079210: 0100 0000 2f98 93a0 0000 0020 0005 0000
Default Flow Context Read OK
PME2: pme_ctx_ctrl_nop done
pme2_test_high: ctrl_cb() invoked, fd;!
f1079250: 0100 0000 ea07 fd9c 0000 0000 0007 0000
PME2: pme_ctx_ctrl_update_flow done
pme2_test_high: ctrl_cb() invoked, fd;!
f1079290: 0100 0000 2f98 93a0 0000 0020 0004 0000
pme2_test_high: ctrl_cb() invoked, fd;!
```

```
f10792d0: 0100 0000 2f98 93a0 0000 0020 0004 0000
PME2: pme_ctx_ctrl_read_flow done
pme2_test_high: ctrl_cb() invoked, fd;!
f1079310: 0100 0000 2f98 93a0 0000 0020 0005 0000
PME2: pme_ctx_ctrl_nop done
pme2_test_high: ctrl_cb() invoked, fd;!
f1079350: 0100 0000 ea07 fdfc 0000 0000 0007 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f1079390: 0100 0000
```

PME Scan Test Output:

```
st: About to allocate bpool
st: Allocate buffer pool id 39
st: Allocate buffer of size 256
st: virt address ef980980
st: physical address 0x2f980980
st: Allocate buffer of size 0x100
st: Released to bman
st: Config bpid 39 with size 3
pme2_test_high: ctrl_cb() invoked, fd;!
f1079290: 0100 0000 2f98 93a0 0000 0020 0004 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f10792d0: 0100 0000 2f98 93a0 0000 0020 0005 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f1079350: 0100 0000 2f98 93a0 0000 0020 0005 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f10793d0: 0100 0000 2f98 93a0 0000 0020 0005 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f1079010: 0100 0000 2f98 93a0 0000 0020 0004 0000
st: Scan Test Passed
pme2_test_high: ctrl_cb() invoked, fd;!
f1079110: 0100 0000 2f98 93a0 0000 0020 0004 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f1079150: 0100 0000 2f98 93a0 0000 0020 0005 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f10791d0: 0100 0000 2f98 93a0 0000 0020 0005 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f1079250: 0100 0000 2f98 93a0 0000 0020 0005 0000
pme2_test_high: ctrl_cb() invoked, fd;!
f1079290: 0100 0000 2f98 93a0 0000 0020 0004 0000
st: Scan Test Passed
```

Linux user-space Tests

```
Run the pmm application to load the sample regexs:
Setup the pme database:
$ cd /sample_rules/
$ pmm
[ Enter text at the "pmm> " prompt]
pmm> add regex file source sample_regexs
REC: WARNING: sample_regexs: Line 22 Compiled pattern has only 1
byte fingerprint. Performance may be impacted.
The "sample_regexs" file was compiled with warnings.
Successfully added 27 regexes to the PM DB with handle 0.
```

```
Command execution time: 00:00:00 [hour:min:sec].
pmm> commit
Successfully committed changes made to the data base of expressions.
Command execution time: 00:00:00 [hour:min:sec].
pmm> quit
Terminating the PMM application.
Run the pm_scan_demo application:
$ pm_scan_demo
[ Enter text at the "> " prompt]
> example1 /FTF Americas 2006/
#00: Scanning 31 bytes.
match(0x01): len=0x11 offset=0x000000000000:0000001e tag=0x00000000
>quit
Number of successful scan: 1
Number of full match: 1
Number of inconclusive match: 0
Number of rule report: 0
```

Known Bugs, Limitations, or Technical Issues

1. BMan buffer pools for PME output not supported for Linux user space use.
2. In case of any PME error detection, the software PME Context transitions to DEAD state. This prevents use of a PME Context after an operational error.

Supporting Documentation

- P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual
- Pattern Matcher 2.0 Software User's Guide
- Pattern Matcher 2.0, 2.1, 2.2 Software API Reference Manual
- USDPAA "pme_loopback" User Guide

5.2.6.2 Pattern Matcher 2.0 Software User's Guide

5.2.6.2.1 Preface

5.2.6.2.1.1 About This Book

This user's guide is written for programmers developing software for a PowerQUICC processor with Pattern Matcher 2.0 capabilities. It describes the supplied pattern matcher software components, how to use various application programming interfaces, and how to integrate the resulting customized software. The supplied sample Pattern Matcher applications are also described in this document.

5.2.6.2.1.2 Audience

This manual supports system software and application programmers who want to use the Pattern Matcher 2.0 in their product.

5.2.6.2.1.3 Organization

This book contains the following topics.

- Overview - The Pattern Matcher hardware and software architecture.
- Regular Expressions -The regex operation and the compiler.
- Stateful Rules -The stateful rule concepts and the stateful rule language.

- Pattern Management Software -The functionality of the main software modules for pattern management, including the linker-loader, the pattern matcher manager, the PM statistics manager, and the distributed pattern management software .
- Pattern Matcher Driver -The Pattern Matcher driver software for Linux is the layer of software that provides applications an interface into the Pattern Matcher. This chapter discusses the application interfaces, initializing and configuring the drivers, memory structure, scanning data, third-party interfaces, and error handling.
- Data Scan Sample Application -Three main example applications, including a scan demo and a layer-7 packet classifier for Linux.
- Software Components - Software modules and a listing of sample code modules.
- Pattern Matcher FAQ - Frequently asked Pattern Matcher questions.

5.2.6.2.1.4 Conventions

This document uses the following notational conventions:

- `Courier`
monospaced type indicates commands, command parameters, code examples, and file and directory names.
- *Italic* type indicates replaceable command or function parameters, that is, variables.
- **Bold** type indicates function names.

5.2.6.2.1.5 Suggested Reading

The following documents contain information that supplements this guide:

- Applicable PowerQUICC processor (contains Pattern Matcher 2.0) reference manual
- *Pattern Matcher 2.0 Software API Reference Manual (PMAPIRM)*

5.2.6.2.2 Overview

The Pattern Matcher 2.0 provides the following capabilities:

- High-performance, hardware pattern-matching of compressed and uncompressed data.
- On-chip hash tables for low system memory utilization.
- Patterns expressed in regex with capabilities beyond that provided by the regex language.
- Pattern-matching across data scan units (for example, can match patterns split across packets).
- Improvements over other Pattern Matcher technologies are as follows:
 - No pattern "explosion" to support "wildcarding"
 - Fast compilation of pattern database
 - Fast incremental additions to pattern database
 - Patterns stored in main DDR DRAM, not SRAM or FCRAM

5.2.6.2.2.1 Pattern Matcher Hardware

The Pattern Matcher 2.0 hardware details can be found in the applicable PowerQUICC processor reference manual. A general overview is provided below:

The Pattern Matcher 2.0 (henceforth referred to as Pattern Matcher) contains high-performance hardware capable of performing unanchored searches for up to 32,000 patterns. It offers support for built-in case insensitivity, repetition, and nested rules, and both predefined and user-defined character classes. Additionally, the Pattern Matcher provides the ability to create stateful relationships between data examination events, such as matched expressions detected by the Pattern Matcher so

more complex regular expressions, as well as certain other complex multi-pattern rules, can be implemented directly in hardware.

The Pattern Matcher is implemented as a series of functional units pipelined together to achieve desired functionality and scan performance as shown in [Figure 75. Pattern Matcher High Level Block Diagram](#) on page 352:

- Pattern Matcher Frame Agent (PMFA)
- Key element scanner (KES)
- Data examination engine (DXE)
- Stateful rule engine (SRE)

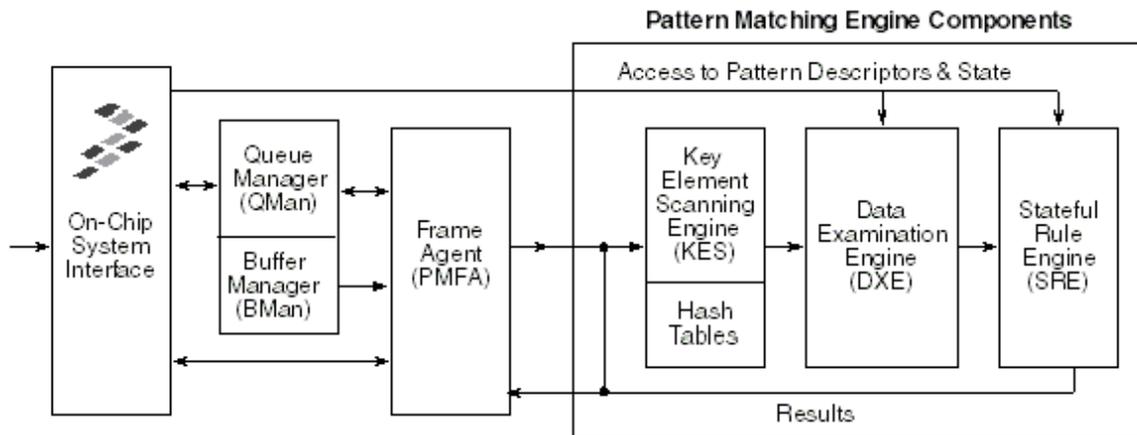


Figure 75. Pattern Matcher High Level Block Diagram

Up to eight work-units can be in flight in the pipeline at any given time. A work-unit represents an atomic work request operation to the Pattern Matcher. Work request operations are conveyed to the Pattern Matcher through the Pattern Matcher Frame Agent (PMFA) via Queue Manager (QMan) Direct Connect Portal. The Pattern Matcher implements a single pipeline, which means that processing of work-units are completed in the same order as they were initially selected.

The first stage of the pipeline is the PMFA. It provides an interface to the Frame Queue datapath environment (Queue Manager and Buffer Manager) to receive work requests (scan data and control messages) and to send notifications/reports (if required) of completed work requests.

The core pattern-matching functionality is implemented as a three stage pipeline. The first stage is the key element scanner (KES), which serves as a preliminary filter by detecting the possibility of matches of up to 32,000 patterns simultaneously through a single pass through the data. The KES implements a multi-stage, hash-based, proprietary algorithm using on-chip memory to determine the likelihood of the data being a match with a pattern. Matches found at this stage are delivered to the data examination engine (DXE), which performs a more stringent comparison to determine if the match actually exists or not. The DXE implements a non-deterministic finite automaton (NFA) capable of implementing a significant subset of the regular expression (regex) pattern definition language, as well as many constructs that cannot be expressed in regex. Successfully matched patterns are passed to the next stage called the stateful rule engine (SRE). The SRE's main function is to execute stateful rules against pattern match events. Stateful rules provide the means to track state and context information between matches. The SRE also formats the reports of matches including pattern match events that don't trigger the execution of stateful rules. The reports are passed to the PMFA, which in turn produces an entry in the output frame queue that contains a pointer to an output data descriptor containing the pattern reports associated for a given work-unit.

5.2.6.2.2 Pattern Matcher Software

The Pattern Matcher software provides the ability to use the Pattern Matcher hardware assists to recognize various complex patterns in a stream of either data or packets. The application may be informed of every single pattern match or a summary of several pattern matches. Based on the pattern match results, the application can take appropriate actions, such as logging the pattern match events, dropping some packets, or forwarding the data onwards. The Pattern Matcher allows for data

scanning with minimum data copy (for example, any data received from the network and stored in the system memory buffers may be sent to the Pattern Matcher without any need for data copy). Also, the Pattern Matcher is capable of detecting patterns spread across successive presentation of bytes in multiple scan units.

The figure below shows the supplied Pattern Matcher software components and their interactions.

- Pattern Management software
 - Regex Compiler
 - Stateful rule Compiler
 - Linker-loader
 - Pattern Matcher (PM) Manager application (Linux implementation)
- Pattern Matcher Driver
 - Pattern Matcher (PM) Driver (Linux implementation)
 - Pattern Matcher (PM) Control Interface (Linux implementation)
- Pattern Matcher (PM) Data Scan Applications

The section, Software Components, contains a list of supplied Pattern Matcher software modules.

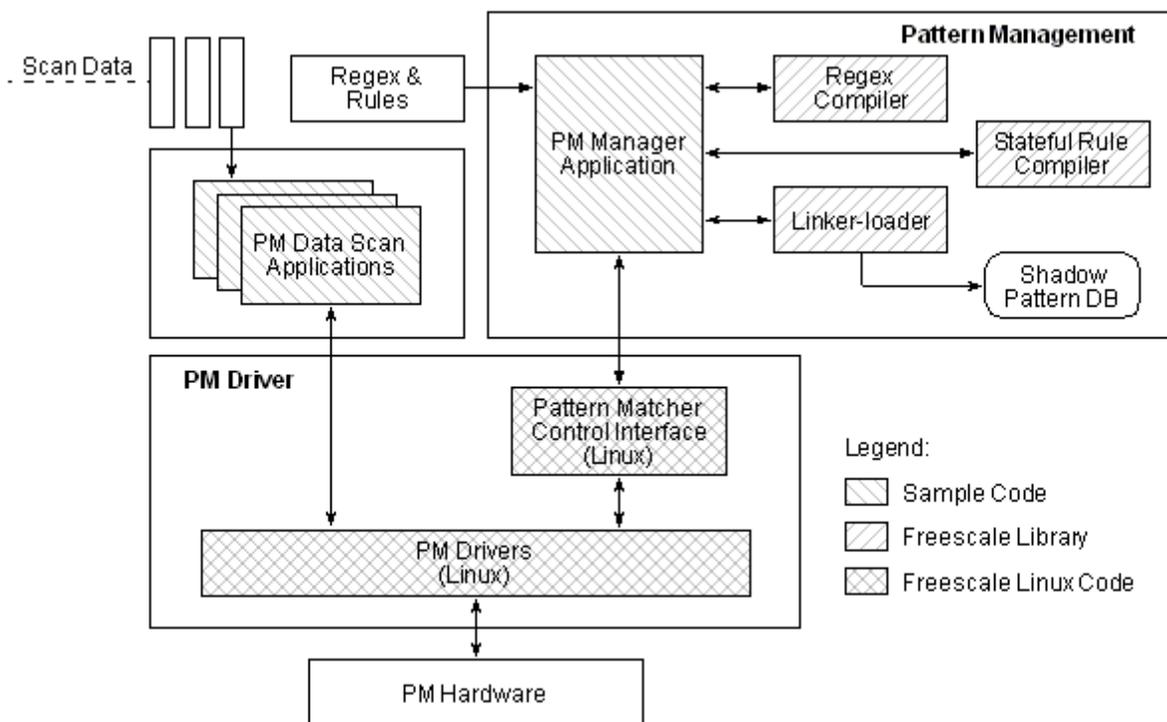


Figure 76. Pattern Matcher Software Overview

5.2.6.2.2.2.1 Pattern Management Software for Pattern Matcher

The Pattern Management software is used to configure the Pattern Matcher by enabling adding, deleting and querying patterns (arrangement of symbols to be matched) and stateful rules (means to track state and context information between pattern matches). The Pattern Management software components are explained in the following sections.

NOTE

Both the regex and stateful rule compilers are available as an executable program with command line interface and also as a "C" library with API.

5.2.6.2.2.2.1.1 Regex Compiler

Regular expressions (regexes) provide a powerful way of representing complex patterns. The regex compiler accepts a regex string with options or a file containing regexes, and converts these regexes into patterns stored in Pattern Matcher hardware-specific format, referred to as "test lines".

5.2.6.2.2.2.1.2 Stateful Rule Compiler

The stateful rule compiler accepts a file containing one or more stateful rules and converts the rules into Pattern Matcher hardware-specific format, referred to as "stateful rule reactions." If stateful rule reactions are loaded for a pattern, these reactions are executed when that pattern matches.

5.2.6.2.2.2.1.3 Linker-Loader

The linker-loader accepts patterns encoded in the hardware-specific format (test lines) and stateful rules converted into the hardware-specific format (stateful rule reactions) and generates a hardware-optimal pattern and stateful rule database (referred as the Pattern Matcher database). A copy of this database (referred as the shadow pattern database) is maintained by the linker-loader for reload and update purposes.

There are architectural advantages to having the separate step to link patterns using linker-loader from compiling patterns; these advantages are as follows:

- Different pattern compilers can coexist with single linker-loader. For example, the linker-loader can be used with the regex compiler from NXP and a proprietary pattern compiler for the customer's own signature definitions.
- Compilers can run on a number of different remote host systems.

The linker-loader is available as a "C" library.

5.2.6.2.2.2.1.4 Pattern Matcher Manager Application (PMM)

The Pattern Matcher manager application (PMM) is an example of an application that manages the creation and distribution of patterns. It uses NXP-provided libraries, namely the regex compiler, the stateful rule compiler and the linker-loader in order to compile, link and load patterns into the Pattern Matcher.

The PMM application is implemented as a Linux user-space process that runs on the PowerQUICC processor containing the Pattern Matcher.

Take note that the Pattern Management software can be implemented as a distributed application with only a small part running on directly on the PowerQUICC processor containing the Pattern Matcher.

5.2.6.2.2.2.2 *Pattern Matcher Driver*

The Pattern Matcher driver provides a means of sending commands (for example, pattern search request or pattern configuration messages) into the Pattern Matcher and delivering the responses from the Pattern Matcher to the software. The pattern search is performed under full application control via the Pattern Matcher Driver. After a unit of data (for example, data work-unit) has been searched, the application is presented with the pattern search report if configured to do so.

5.2.6.2.2.2.2.1 Pattern Matcher Control Interface

The PM Control Interface (PMCI) module is provided as a Linux user-space library. It contains C functional Interface to send and receive Pattern Matcher control commands to the Pattern Matcher via Pattern Matcher driver software. The PM control commands initialize PM driver and configure the PM hardware tables. The PMCI module converts complex PM control commands into PM driver primitives. The PMCI API functions and the PM Control command structures are defined in the Pattern Matcher 2.0 Software API Reference Manual.

5.2.6.2.2.2.3 *Pattern Matcher Data Scan Applications*

The PM Data Scan Applications are customer applications that use the Pattern Matcher hardware for accelerating pattern searches. These applications may receive data from network or other sources. When it comes time to search for patterns, applications use the PM Driver APIs to send a copy of the data to the Pattern Matcher hardware. The NXP Linux

implementation of the PM Driver APIs provides mechanisms for zero-copy DMA operations. The pattern match events are reported back to the application as results of the scan operations. Linux-based Pattern Matcher Drivers provide a variety of scan APIs to the PM data scan applications, which are mainly:

- User-space blocking and non-blocking APIs
- Kernel-space blocking and non-blocking APIs

The expressions and stateful rules are added via the PMM before PM data scan applications can search data for these expressions and rules. Also, the expression and rules may be incrementally added or deleted dynamically without completing stopping the scan applications.

The source code for few data scan applications is provided as an example use of the PM Driver APIs.

5.2.6.2.3 Regular Expressions

The NXP regex compiler accepts search patterns using syntax similar to that in software-based regex engines, such as Perl or the open source Perl-compatible regular expression engine (PCRE). Despite the similar syntax, different software-based regex engines work differently and sometimes produce different match results. Although the NXP Pattern Matcher hardware combined with the NXP regex compiler behaves like other engines in many respects, it has important differences and unique features. The NXP regex engine described in this chapter is the combination of the Pattern Matcher hardware and the regex compiler.

NOTE

For details on the regex compiler, refer to *Pattern Matcher Compiler Software User's Guide*.

5.2.6.2.4 Application interface

The regex compiler is provided as a C library and also as a binary executable program under Linux. The PM Manager system software can compile single regex or multiple regexes stored in a file using either the library function or by invoking the regex compiler program.

5.2.6.2.4.1 Compiler API

The regex compiler API functions are located in following header file:

```
<freescale pm source code path>/pm/user/include/pmrec.h
```

For details on the regex compiler API, refer to *Pattern Matcher 2.0 Software API Reference Manual*.

5.2.6.2.4.2 Command Line

The pmrec is the executable program to compile regexes from stdin or an input file, as shown in the following command help:

```
pmrec --help
Description:
    The regex compiler takes input from a file or stdin and
    converts the regex to patterns in an internal format.
    The output must be passed to the Linker-loader software in order
    to install the patterns on the hardware.
Options:
    -h, --help           This help.
    -i, --input          The name of the file containing the user's regular expressions.
                        Defaults to STDIN.
    -o, --output         The name of the file where the output will be placed.
                        Defaults to 'regex.compiled'.
    -W, --Werror         Warnings are errors.
```

```

-w, --w          Suppress warning messages. Note: Ignored if --Werror used.
-n, --nostrings Do not include expression strings in output binary file.
-b, --8572rev1.0 Compile for 8572 rev 1.0 silicon.
Example:
  pmrec --input my_expressions --output my_expressions.out
  
```

The following example shows the regex input file.

Regex Input File

```

#
# Regular expression file
#
# Look for login attempts
login /login/set=1 subset=0xffff tag=0x01
# Look for logouts
logout /^logout/m tag=0x02
  
```

5.2.6.2.4.3 Scanning API

When the patterns are compiled, they must be added in the Pattern Matcher hardware using the linker-loader. After patterns are configured into the PM hardware, the PM driver pattern scanning interfaces are used to search for patterns.

5.2.6.2.4.4 Scan Result

The data scan operation generates a scan result notification from the Pattern Matcher. The scan result contains zero or more match reports. The match report contains information, such as the tag of the regex that matched and where in the data the match was found.

NOTE

If the noreport option is specified in the regex, no match report is generated for that regex.

The type of report, simple or verbose, is configurable on a per-scan stream basis. For a reference, the simple match report is described in [Table 73. Simple Match Report](#) on page 356. Refer to applicable PowerQUICC processor reference manual for the description of the verbose reports.

Table 73. Simple Match Report

Byte Offset	Bits	Description
0	0	Set to 0
0	1-3	Indication of match type as either full match or inconclusive match. The values varies based on the configuration of the inconclusive mode selection. For the default mode the values are: 000 Complete match within a search window 001 Inconclusive match on the right side of the search window 010 inconclusive match on the left side of the search window 011 inconclusive match on both left and right side of the window
0	4-7	Set to 0x1
1	0-7	Number of bytes matched by the pattern (max value of 128)
<i>Table continues on the next page...</i>		

Table 73. Simple Match Report (continued)

Byte Offset	Bits	Description
2	0-47	The number of bytes scanned initially prior to the current work-unit.
8	0-31	The position of rightmost byte of the match relative to the work-unit scanned
12	0-31	A 32-bit tag assigned to the regex that matched

At the end of the match reports, there is an optional end-of-SUI report. The PM driver configuration parameter is available to enable or disable the end-of-SUI reports per system. It is enabled by default. The end of SUI report is 5 bytes in length as described in [Table 74. End of SUI Report](#) on page 357.

Table 74. End of SUI Report

Byte Offset	Bits	Description
0	0-7	Set to 0x80
1	0-31	Total new bytes scanned for this report. This is equal to bytes in the work-unit.

The scan result notification presented by the PM driver also indicates whether or not the result is truncated due to insufficient buffer space. This can occur for two reasons, as follows:

- The software supplied output scatter/gather buffer is not large enough to contain all of the produced output.
- The DMA Engine channel's free buffer list is exhausted while the DMA Engine is buffering output data.

NOTE

The scan application behavior should be considered carefully in handling the scan result that may be truncated. The truncated results imply that not all of the matches are reported in the result. Additionally, there may be a partial match report or a partial end of SUI report in the result due to truncation.

The scan result notification also indicates whether or not there are any exceptions encountered by the Pattern Matcher. The Pattern Matcher exception codes are described in the application PowerQUICC Processor Reference Manual. The "DXE instruction limit error" exception is worthy of a note. This error indicates that the Pattern Matcher has exceeded the maximum allowed test line executions per pattern. Almost no real regexes can exceed such limit, but it is theoretically possible. If such limit is reached, either rewrite or delete the regex causing this error. Alternatively, using PM driver software, it is possible to increase or disable the maximum allowed test line executions limit.

5.2.6.2.5 Stateful rules

The stateful rule engine (SRE), a component of the Pattern Matcher, detects and reports the occurrence of specified events within the data content of individual sessions. A stateful rule defines a state machine that executes actions when specific events occur. The SRE does not directly examine the content but instead reacts to events from the data examination engine (DXE). Typical DXE events are notifications of successfully matched patterns. The DXE supplies additional information on each event, such as match position and the captured data from the scanned data content. The stateful rule can use this information in its reactions to events.

As shown in the example state machine diagram in the [Figure 77. Stateful Rule State Machine](#) on page 358, a stateful rule can be used to detect and report occurrences of patterns in a particular sequence. For example, the detection of pattern P3 is reported, but only after an occurrence of pattern P1 or an occurrence of P1 followed by P2. The complex scenarios, such

as the example in [Figure 77. Stateful Rule State Machine](#) on page 358, can be written using a combination of the regex and stateful rules.

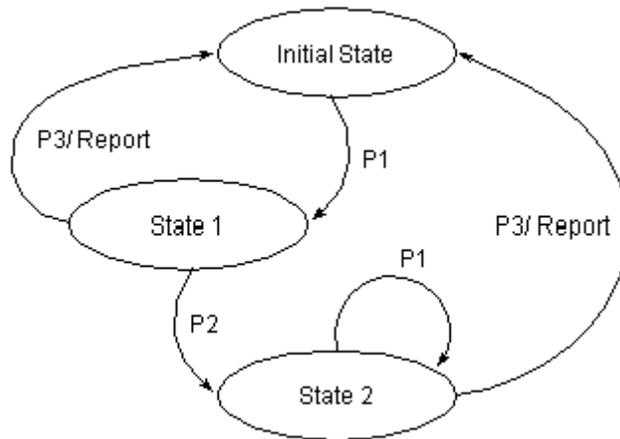


Figure 77. Stateful Rule State Machine

From a hardware perspective, a stateful rule is a set of reactions that is applied to a single meta-state record. Each reaction executes as a result of a specific data examination event, such as the confirmed detection of a pattern. The stateful rule compiler accepts the stateful rule and outputs binary data that represents SRE reactions. The binary output is used in an API call to the linker-loader to link the rule with appropriate patterns and to load the rule to the Pattern Matcher hardware. The expressions used in a stateful rule must be added through the linker-loader before the stateful rule is linked.

NOTE

For details on the stateful rule compiler, refer to *Pattern Matcher Compiler Software User's Guide*.

5.2.6.2.5.1 Stateful Rule Application Interface

The stateful rule compiler is provided as a C library and a binary executable program under Linux. The PM manager software can compile single or multiple rules stored in a file using either the library function or by invoking the compiler program.

5.2.6.2.5.1.1 Stateful Rule Compiler API

The stateful rule compiler header file is:

```
pm/user/include/pmsrc.h
```

For details on the compiler API, refer to the *Pattern Matcher 2.0 Software API Reference Manual*.

5.2.6.2.5.1.2 Command Line Options

The pmsrc is the executable program to compile rules from stdin or input file. See command help, as follows:

```
stateful_rule_compiler
Description:
  The stateful rule compiler takes input from a file or STDIN and
  converts the user code to low level stateful rule instructions.
  The output must be passed to the linker/loader software in order
  to install the rules on the hardware.
Options:
-h, --help      This help.
-i, --input <file>  The name of the file containing the users
                    stateful rules.
                    Defaults to STDIN.
-o, --output <file> The name of the file where the output will
```

```

be placed.
Defaults to 'stateful_rule.compiled'.
-r, --report_pad <size> Pad reports to this byte boundary.
                        Default is none.
                        Allowed values: 0 or 4
-p, --string_pad <size> Pad strings to this byte size.
                        Default is 80.
                        Allowed values: 0 < value <= 16384 (multiple of 2)
-c, --report_constant_size <size> Constants within reports will be
                        aligned to this byte boundary.
                        Default is 4.
                        Allowed values: 2, 4, 6, or 8
-a, --allow_inconclusive Allow inconclusive matching.
                        Default is to disallow inconclusive matches.
-W, --Werror Warnings are errors.
-w, --w Suppress warning messages. Ignored if --Werror used.
Example:
stateful_rule_compiler --input my_rules -output my_rules.out

```

The following example shows the stateful rule input file.

Stateful Rule Input File

```

STATEFUL_RULE: HTTP_Recognizer
RESET_STATE:
EVENT "http_request"
next_state AWAIT_response
STATE AWAIT_response:
EVENT "http_response"
# report HTTP traffic observed
report {0x00000001}
next_state RESET_STATE

```

5.2.6.2.5.13 Scan Report

The scanning interface for stateful rules is no different than the interface for scanning regex. When data is scanned to search for regex, the stateful rules, if configured, are also executed for any matched regex. The stateful rule match report content is customized by the stateful rule reactions. It can be any number of bytes and values, depending on the reaction. Typically, rule writers select a certain stateful rule report header so that the scan applications can interpret the report consistently.

The report action discussed in "Actions" (Chapter 3, "Stateful Rules") found in *Pattern Matcher Compiler Software User's Guide* describes the optional header appended by the stateful rule compiler. Alternatively, stateful rule writers can use a report format that is identical to the Pattern Matcher simple report format.

5.2.6.2.6 Pattern management software

The pattern management software is built using the Pattern Matcher software libraries provided by NXP. The sample Pattern Matcher manager application (PMM), described in [Pattern matcher manager \(PMM\)](#) on page 362, is supplied by NXP as a Linux user-space application. The user can write their own PMM software or add PMM functionality to their existing management application. The high-level PMM application functions are as follows:

- Providing the user interface for adding, deleting, and querying source regexes and stateful rules
- Linking compiled regexes and compiled rules to create a PM hardware database
- Committing the PM hardware database to the PM hardware

- Monitoring PM hardware for any exceptions

The figure below shows the PMM application with NXP-supplied software components.

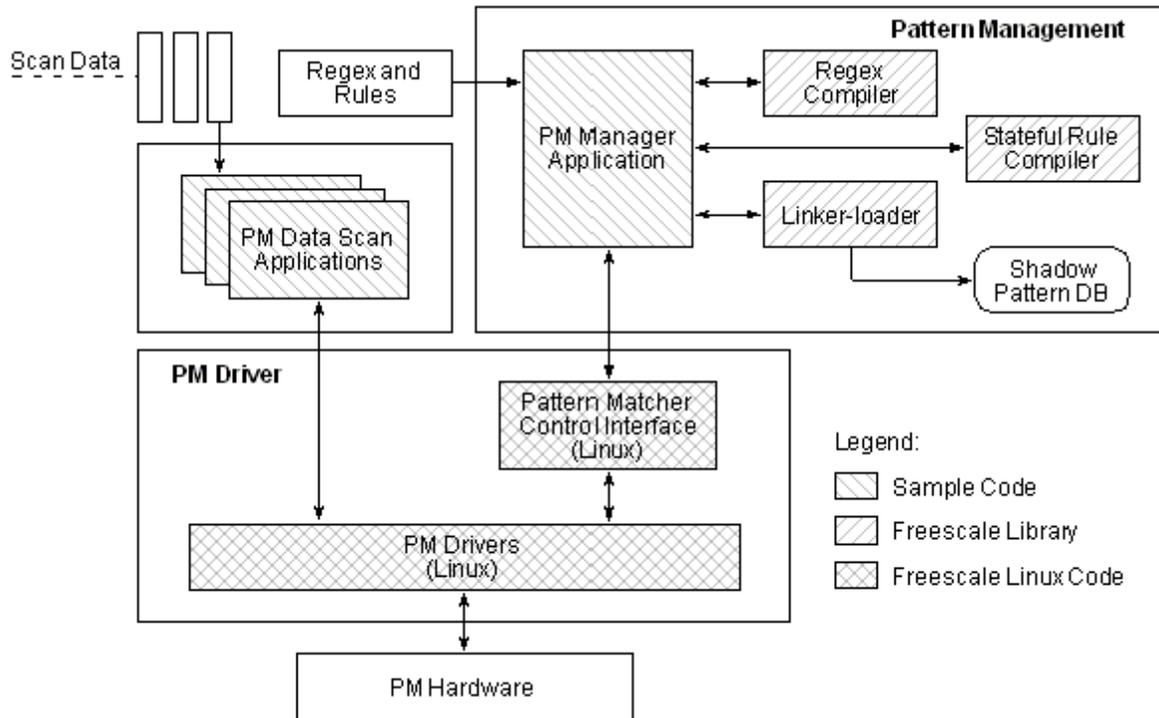


Figure 78. Locally-Managed PM Hardware

5.2.6.2.6.1 Compilers

5.2.6.2.6.2 Linker-loader for pattern management

The linker-loader, a sub-component of the pattern management software system, allows a user of the Pattern Matcher module to configure the hardware. You can use the linker-loader to configure new search patterns in the monitored data or to query the state of the Pattern Matcher hardware. The linker-loader manages the PM hardware tables, which are the trigger, confidence, and confirmation tables. These tables are described in the PowerQUICC processor reference manual for your product. The linker-loader is implemented as a library of functions performing different actions on the Pattern Matcher hardware. These functions are accessible through a well-defined API. In the Linux implementation, the linker-loader library is used by a user space application. [Figure 79. Pattern matcher control data path](#) on page 361 illustrates the PM control data path at a high level.

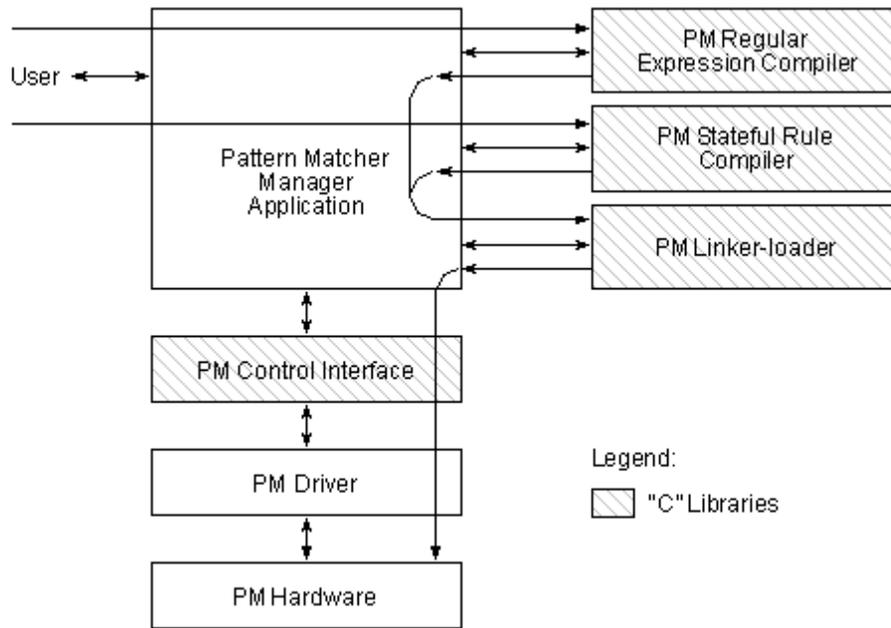


Figure 79. Pattern matcher control data path

The linker-loader library maintains a shadow database of the hardware tables that contains the Pattern Matcher hardware database plus additional information necessary for the software, such as all currently configured expressions and stateful rules. The hardware tables are highly compressed to save PM hardware internal memory usage. The linker-loader compresses the database as part of the commit phase.

The user interacting with the shadow database refers to expressions and rules by their names. The names do not have to be stored in the hardware database, and expressions are added or removed from the shadow database. Whenever the changes made to the shadow database are loaded to the PM hardware, it is reconfigured to reflect the state of the shadow database. The expressions and stateful rules can be incrementally deleted from or added to the Pattern Matcher hardware through the linker-loader library. When incremental changes are committed, the linker-loader does not load the entire shadow database to the hardware. Only incremental changes are loaded, thus limiting any outage while making incremental changes.

NOTE

Because the shadow database is embedded in the linker-loader library, take care to ensure data integrity through the Pattern Matcher manager (PMM) application. The memory used for the shadow database is not persistent. It is allocated in the context of the PMM application.

The API to the linker-loader library are described in the Pattern Matcher 2.0 Software API Reference Manual.

5.2.6.2.6.2.1 Linker-Loader API

The linker-loader is provided as a C library. The pattern management software can include the linker-loader library to link and load patterns and rules into the Pattern Matcher hardware. The linker-loader API is a functional API, and each API function runs to completion. In other words, the requested operation is completed and the results of the operation are available when the invoked function returns. The API functions are divided into the following categories, all of which are described in the *Pattern Matcher 2.0 Software API Reference Manual*:

- Initialization primitives
- Expression related primitives
- Stateful rule related primitives
- Debug primitives

Note that the linker-loader library does not interact directly with the Pattern Matcher control interface. Instead, the library sends Pattern Matcher control messages through functions provided by the PMM application. The PMM registers functions

(Refer to the PM loader agent functions in the Pattern Matcher 2.0 Software API Reference Manual). That is, it sends PM control messages to the Pattern Matcher control interface indirectly using callback functions registered by the PMM. The callback functions provide you the flexibility to implement the communication pipe between the linker-loader and the PM control interface that is most appropriate for the pattern management system. The PM software architecture allows the PMM application to be split into three parts, one that compiles regex and rules, another that interacts with the linker-loader, and yet another that interacts with the Pattern Matcher control interface.

5.2.6.2.6.3 Pattern matcher manager (PMM)

The sample Pattern matcher manager (PMM) is a NXP application implemented as a Linux user-space process. The PMM application provides a simple command-based interface to compile, link, and load expressions and stateful rules. It runs under Linux 2.6 within the same processor system (host) on which the PM hardware resides. This section describes the key commands available through the PMM command-line interface. For more information, refer to the PMM help text.

The PMCC/PMCD application is an alternative application that provide the same features of PMM. PMCD runs as a daemon(memory resident program), and PMCC runs as a client with a command-line interface identical to PMM. To use PMCC/PMCD, run “pmcd” first, then run “pmcc”.

The difference between them is PMM is a standalone application without any persistent datapath that spans multiple invocations of PMM. It starts with an empty expression and rule database, so a new session of PMM cannot add or delete items already committed to PME by a previous PMM session. With PMCC/PMCD, PMCD will keep the database in memory. As long as PMCD is running, the PME shadow database remains persistent. With PMCD running, any new session of PMCC can add or delete items added by previous PMCC sessions.

It is not recommended to run PMM and PMCD, or multiple session of them at the same time. Since each session of PMM or PMCD assume the sole controller role of the PME, the PME database corruption can occur with multiple controllers.

5.2.6.2.6.3.1 Adding Regexes and Rules

The add command is used to add regex or rules to the linker-loader database, as shown in the Adding Regexes and Rules example that follows. The regex(es) or rule(s) to be added can be in the source format (yet to be compiled), or they can be in the binary format (already compiled). If the regex is presented in the source format, it is first compiled. The compilation must finish successfully before the regex is added to the linker-loader database. If the regex or rule is presented in the binary format, it is added to the linker-loader database without preprocessing. Clearly, adding regexes or rules in the binary format is faster.

When source formatted regexes or rules are added from a file, the compiled results can be stored in a binary formatted file. Specify the optional binary keyword followed by a file name.

Adding Regexes and Rules

```
add regex file source myexpressions.src
add regex name e1 exp /matchme/tag=0x01
add regex file binary regexes.bin
add rule file source /tmp/rules.src binary /tmp/rules.bin
```

5.2.6.2.6.3.2 Committing Added Regexes and Rules

The commit command is used to configure the Pattern Matcher hardware with the regexes and rules that are added to the system. The first, or initial, invocation of the commit command results in an optimal distribution of the pattern records in the PM hardware. Such optimization boosts the performance of the PM hardware. The second and subsequent, or incremental, invocations of the commit command do not perform such optimization. Depending on a particular set of regexes in the database, the regexes that are added during incremental commits, and number of incremental commits, this may not pose

any performance issue. However, in general the pattern management system design should use the first commit command with a regex and rule database that is as close to its final state as possible. With many incremental commits, over time the database may not be as optimum as it can be with an initial commit.

NOTE

As part of the initial commit, the PM hardware database is reset. That is, all previously committed patterns and rules are invalidated.

5.2.6.2.6.3.3 *Deleting Regexes and Rules*

The delete command is used to delete previously added expressions or rules. One or more expressions or rules can be deleted with one command, as shown in the following example:

```
delete regex all

delete regex name p1
```

The number of expression or rule names accepted by the command is limited by the number of arguments allowed in a CLI command. When a request is made to delete all items of a given kind and the operation completes with an error, only the items that can be deleted are deleted. For example, when all the expressions are deleted, the expressions that are part of rules are not deleted.

5.2.6.2.6.3.4 *Showing Regexes and Rules*

The show command can be used to display the fields of the records added to the system—for example, the expression or rule records. The command can also be used to display version information for some software components, such as the linker-loader. An example of showing regexes and rules is as follows:

```
show regex all
```

5.2.6.2.6.3.5 *Other Commands*

There are other useful commands within PMM to set, read, and reset various Pattern Matcher attributes.

5.2.6.2.6.3.6 *PM Statistics*

The Pattern Matcher hardware offers a variety of statistics counters for debugging and analyzing pattern scanning functions. The statistics counters are read reset; that is, they are reset to zero when software reads them. Also, if software does not read the counters frequently, they risk rollover. The statistics offers are from all three PME engines, KES, DXE, and SRE.

The PM statistics collection is done via PME kernel drivers. The counters are queried from the driver interface. The PMM sample application provides means to read statistics from driver.

The following shows the output of the statistics query command from the sample PMM application.

Output of Statistics Query Command

pmm> The PM H/W Statistics:		Current	Previous	Delta
PM Input Bytes	(KES) :	50000	0	50000
PM Output Report Bytes	(SRE) :	283155	0	283155
PM Trigger 1B Hits	(KES) :	0	0	0
PM Trigger 2B Hits	(KES) :	0	0	0
PM Trigger Variable Hits	(KES) :	0	0	0

PM Trigger Special Hits	(KES) :	10786	0	10786
PM Confidence Stage Hits	(KES) :	10786	0	10786
PM Matches	(DXE) :	5345	0	5345
PM SR Execution by DXE	(SRE) :	5345	0	5345
PM SR Execution by SUI	(SRE) :	10659	0	10659
PM SUI With Matches	(DXE) :	5285	0	5285
PM SUI With Reports	(SRE) :	10664	0	10664
PM Input SUIs	(KES) :	10664	0	10664
PM Matches with DRCC	(DXE) :	10786	0	10786

5.2.6.2.6.4 Distributed Pattern Management Software

The PMM application can run directly on the host processor with the PM hardware (as a local PMM) or it can be implemented as a distributed application with only a small part running directly on the host processor. [Pattern management software](#) on page 359 depicts the PMM application implemented as a local PMM with a single user-space process and all user interface functionality linked together with the compile, linker-loader, and PMCI libraries. In this local case, the PMM application and PM hardware are collocated on the same system. However, a more distributed PMM application can be designed; that is, expressions and rules can be compiled and linked on a different system from where the PM hardware resides. Running the PMM in a distributed environment can be attractive in low-cost embedded systems without sufficient resources to run the compilers and the linker-loader. Also, it may be desirable to centralize the compilation and linking functions to one location or system and then distribute the compiled and linked configuration to a possibly larger number of pattern scanning systems. There are several ways the PMM functions can be distributed between two or more hosts. [Figure 80. Distributed PMM](#) on page 365 shows a design where expressions and stateful rules are compiled on one host, linked on another host, and finally loaded to PM hardware from the processor on the target PM hardware.

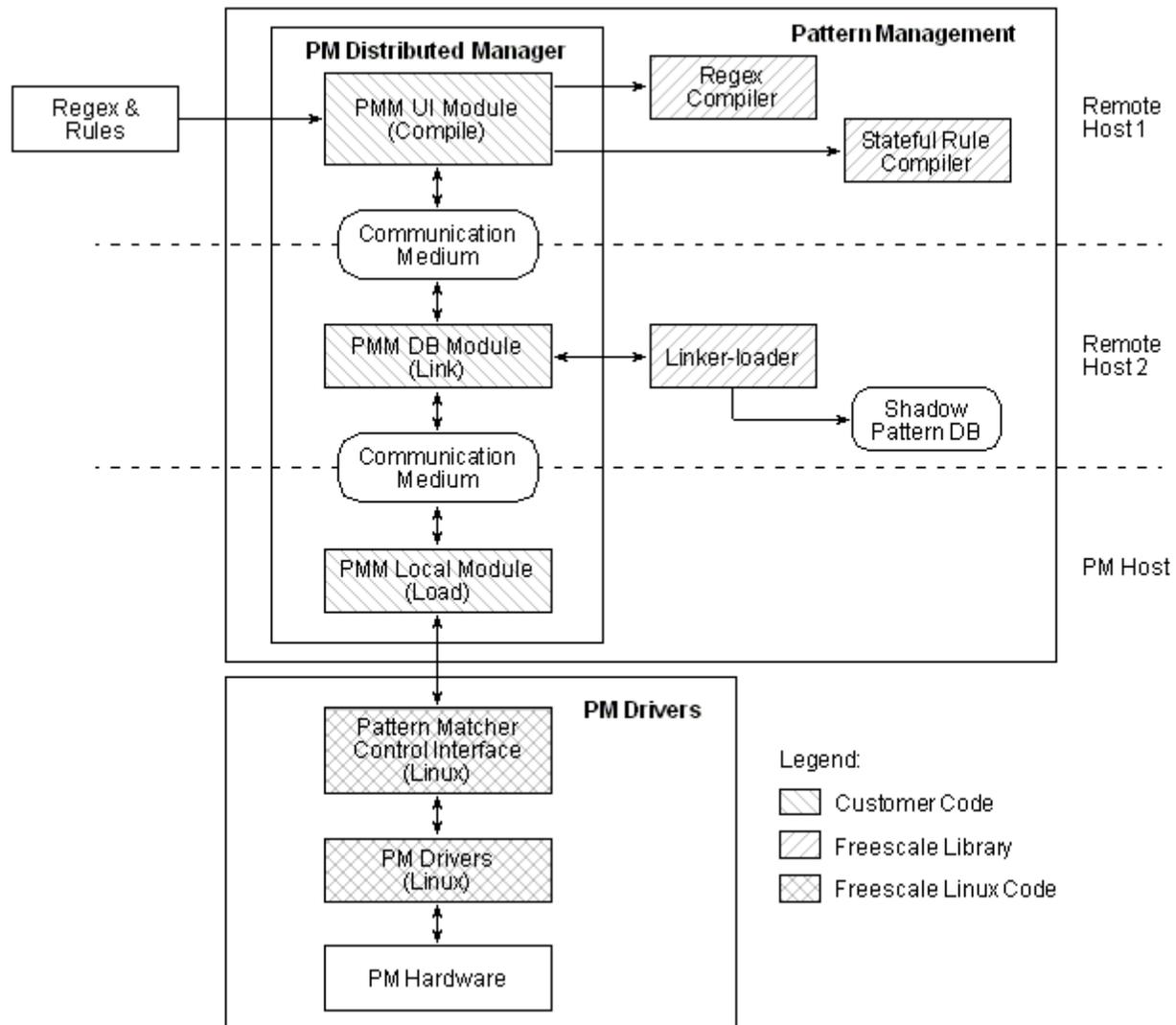


Figure 80. Distributed PMM

In a distributed PMM application, the different parts of the PMM application must communicate with each other using communication mechanisms specific to each PMM application design. The networking and communication details are abstracted away from the Pattern Matcher software libraries through appropriate API functions. This abstraction gives the PMM application the flexibility to implement the communication mechanisms appropriate for it, with the following constraints:

- The PM control messages from linker-loader must be reliably delivered to the PMCI, and the sequence of the sent messages must be preserved.
- Communication failures must be reported through appropriate error return codes from the send and receive functions.

5.2.6.2.7 Pattern Matcher driver software

The Pattern Matcher driver for Linux provides applications an interface for configuring and using the Pattern Matcher.

The PME driver software includes a Linux kernel driver and a PME driver library for USDPA. The driver also provides an ioctl-based Linux user-space interface. The drivers' provide interfaces in Linux kernel and user-space for PME configuration, database setup, and data scanning. For USDPA, the drivers provide a PME data scanning interface only.

Typical data scanning applications will only use the PME scanning and statistics interfaces. The configuration and PME database setup interfaces would typically be used during device initialization by Pattern Management software like the user-space Pattern Matcher Manager (PMM).

The following section describes the high level functionality of the PME driver software. Details of the driver APIs are available in the PME 2.0 Software API Reference Manual.

The PME2.0 Driver software components are shown in the following diagram.

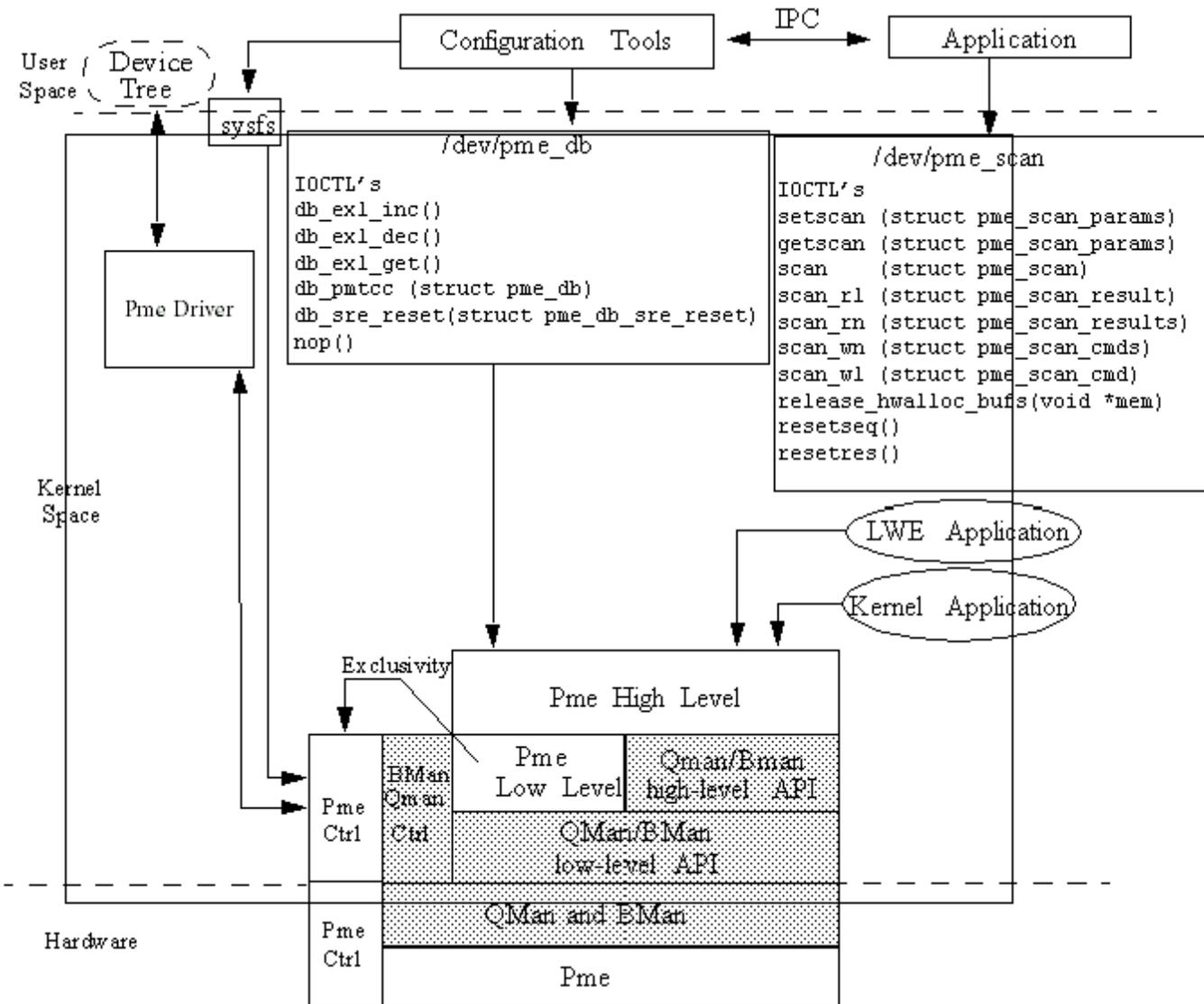


Figure 81. PME driver components

5.2.6.2.7.1 PME configuration interface

The PME configuration interface is an encapsulation of the PME CCSR register space and the global/error interrupt source. This is available to applications running on a control-plane operating system, the hypervisor and/or a boot manager. Non control-plane applications do not have access to the configuration interface.

5.2.6.2.7.1.1 Externally Configured Settings

The U-Boot software is responsible for configuring the PID to LIODN mapping registers as well as the PME's LIODNR register which is used when the PME accesses its private memory.

5.2.6.2.7.1.2 PME device-tree node

The PME device tree node represents the PME device and its CCSR configuration space. When a Linux kernel has PME support built in, it will react to this device tree node by configuring and managing the PME device.

The device-tree node sits within the CCSR node ("soc") and is of the following form:

```
soc@fe000000 {
    [...]
    pme: pme@316000 {
        compatible = "fsl,p4080-pme", "fsl,pme";
        reg = <0x316000 0x10000>;
        fsl,pme-pdsr = <0x0 0x20000000 0x0 0x01000000>
        fsl,pme-sre = <0x0 0x30000000 0x0 0x01000000>
    };
    [...]
};
```

5.2.6.2.7.1.2.1 fsl,pme-pdsr

This property specifies the start location and size of the Pattern Description and Stateful Rule Table in system memory. The table base address must be aligned to a natural 128-byte address boundary. The maximum size of the table is 128 Mbytes. The current driver implementation allows this memory resource to be specified via the 'fsl,pme-pdsr' device-tree property, or by resorting to a default allocation of contiguous memory early during kernel boot. The 'fsl,pme-pdsr' property specifies a 2-tuple of address and size (address is optional), specifying the physical address range. These elements are expressed as 64-bit values, so take two cells each;

```
fsl,pme-pdsr = <0x0 0x20000000 0x0 0x01000000>;
```

Optionally, only the size can be specified and the kernel will try to allocate the contiguous memory during boot time.

```
fsl,pme-pdsr = <0x0 0x100000>
```

If the hypervisor is in use, this address range is "guest physical". If the given memory range falls within the range used by the Linux OS, it will attempt to reserve the range against use by the OS.

5.2.6.2.7.1.2.2 fsl,pme-sre

This property specifies the start location and size of the SRE Context Table in system memory. The table's base address must be aligned to a natural 32-byte address boundary. The maximum size of the table is 4 Gbytes. The current driver implementation allows this memory resource to be specified via the 'fsl,pme-sre' device-tree property, or by resorting to a default allocation of contiguous memory early during kernel boot. The 'fsl,pme-sre' property specifies a 2-tuple of address and size (address is optional), specifying the physical address range. These elements are expressed as 64-bit values, so take two cells each;

```
fsl,pme-sre = <0x0 0x20000000 0x0 0x01000000>;
```

Optionally, only the size can be specified and the kernel will try to allocate the contiguous memory during boot time.

```
fsl,pme-sre = <0x0 0x300000>;
```

If the hypervisor is in use, this address range is "guest physical". If the given memory range falls within the range used by the Linux OS, it will attempt to reserve the range against use by the OS.

5.2.6.2.7.2 PME user space interface

The PME user space interfaces comprises the following:

- `sysfs`
- `/dev/pme_db`
- `/dev/pme_scan`

5.2.6.2.7.2.1 *sysfs*

The *sysfs* interface provides access to PME CCSR space. It is used to control global configuration of PME device parameters and provides an interface for accessing PME statistics. The path to the driver attributes is:

```
/sys/bus/cf_platform/drivers/cf-fsl-pme
```

The `/dev/pme_db` device is used to send Pme pattern matcher database configuration requests via the Pme's Exclusive Frame Queue Control (EFQC) mechanism. This device requires root permissions. The EFQC exclusivity is referenced counted, so by default it is asserted on-demand and released when the processing is done for the context. However, exclusivity can be maintained by using the `db_exl_inc`, and `db_exl_dec` ioctls, which provide supplementary increments and decrements of the reference count. These operations are performed from user space using the following ioctl system calls.

- increment reference count
- decrement reference count
- get the current reference count
- send database request and receive response (synchronous)
- send a nop Pme command

The `/dev/pme_scan` device is used to interface with the PME device via the QMan interface. This interface can only be used for scanning operations. The device can be used to perform synchronous (the calling thread is blocked until an operation's completion) scans or asynchronous (once an operation has begun, the calling thread is able to perform other processing but needs to query the completion of the operation later) scans.

5.2.6.2.7.2.2 *Pme Scan*

The `/dev/pme_scan` device is used to interface with the Pme device via the QMan interface. This interface can only be used for scanning operations. The device can be used to perform synchronous (the calling thread is blocked until an operation's completion) scans via the `scan()` api or asynchronous (once an operation has begun, the calling thread is able to perform other processing but needs to query the completion of the operation later) scans via the `scan_w[1n]` and `scan_r[1n]` APIs. Only flow mode scanning requests are supported by this device. These operations are performed from user space using the ioctl system calls specified below.

- set parameters for scanning operations
- get (retrieve) currently set scanning operation parameters
- reset sequence number
- reset residue
- do single synchronous scan
- send single asynchronous scan command
- send multiple asynchronous scan commands
- get single synchronous scan response
- get multiple synchronous scan responses
- release bman acquired buffers to bman: `release_hwalloc_bufs`

The settable and retrievable scan parameters are grouped as follows:

- Residue attribute
 - residue enable/disable
 - current number of byte in the residue (read only)
- SRE attributes
 - session_id
 - report verbosity
 - end of sui report enabled/disabled
- DXE attributes
 - compare limit
 - match limit
- Pattern attributes
 - pattern set
 - pattern subset

5.2.6.2.7.3 PME Linux kernel, USDPAA driver interface

The PME kernel/USDPAA driver provides a high and low level interface for PME users.

5.2.6.2.7.3.1 PME High-level Driver API

The PME high-level APIs provide a call-back based interface to the PME. The driver provides APIs to manage flow context and issue PMTCC and scan commands. The high-level driver internally co-ordinates the commands to PME and corresponding results. The user-specified call-back functions are called with results from PME. The driver also returns the user-specified context information provided by the application when invoking the call-back function. This saves the applications from maintaining a mapping of active scan requests and their related contexts.

The high-level interface provides the following:

- pme context management. Management operations include: initialize, destroy, disable, enable, query_state, reconfigure. The pme context is comprised of:
 - input/output frame queues
 - flow context record (when in flow mode)
 - frame queues scheduling levels
 - exclusivity state
 - stashing control of frame queue context
 - residue (optional when in flow mode)
- manage mux/demux of scans and pmtcc commands.
- pme input/output qman frame queue setup, modification and teardown.
- registration of user provided scan and pmtcc result callbacks
- statistics management

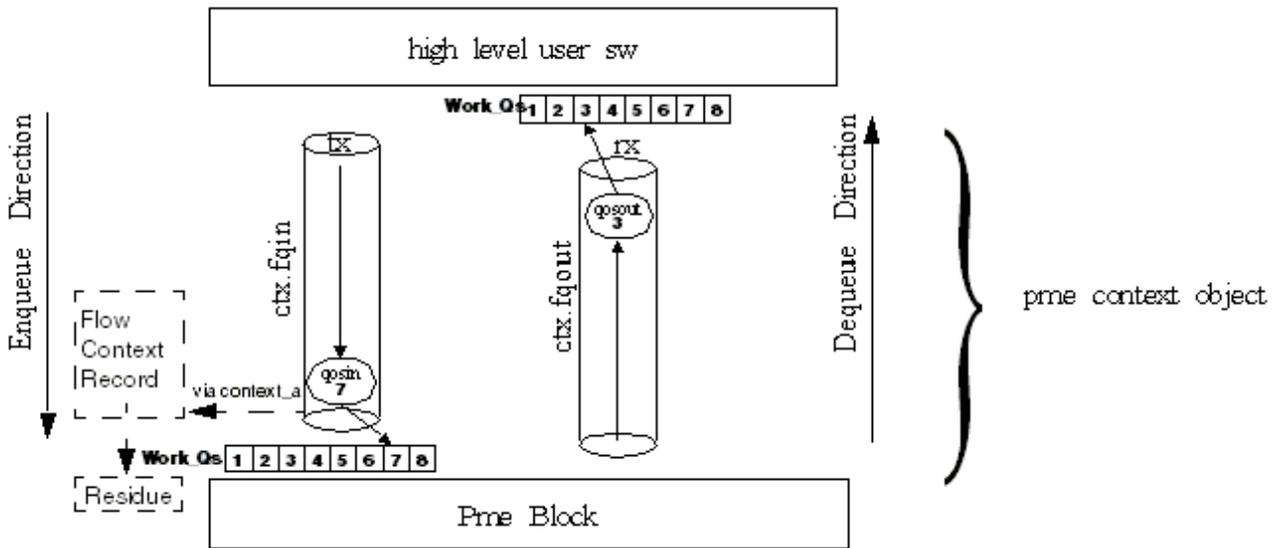


Figure 82. Pme Context Object

5.2.6.2.7.3.1.1 PME Context

The Pme high level layer is represented by a pme_context object. This object allows a user to communicate with the Pme device using the encapsulated QMan interface via two driver allocated frame queues.

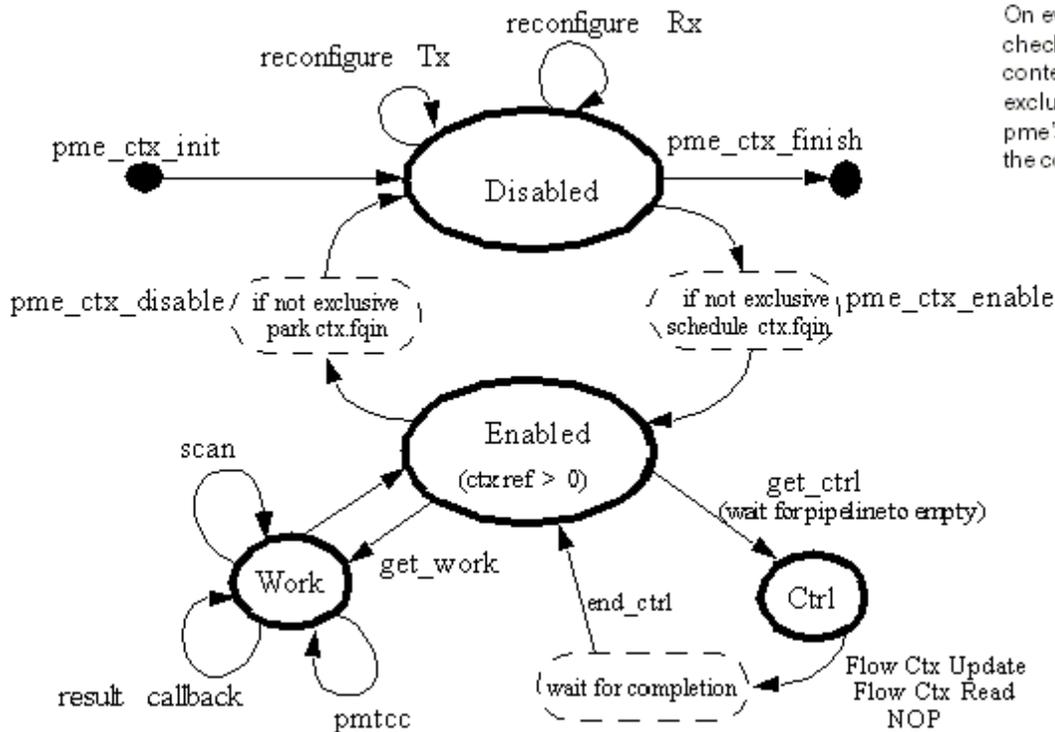
The PME context is a modal object which can be initialized in one of the following modes:

- scan_flow: This mode permits the user to send scan, nop, flow context write and flow context read PME requests. Upon enabling the context the input frame queue is placed in a scheduled state
- scan_direct: This mode permits the user to send scan and nop PME requests. Upon enabling the context the input frame queue is placed in a scheduled state

The following PME Context modes exist for use by the user-space Pattern Matcher Management software. They can also be used by applications making dynamic changes to the PME database or for exclusive scan operations..

- pmtcc: In this mode the context is placed in exclusive mode. The input (tx) frame queue is left in an unscheduled (parked) state. Only pmtcc and nop pme requests are permitted. This mode is only available if the api is built with control functionality and if the operating system has access to the PME's CCSR map.
- scan_exclusive_direct: This mode permits the user to send scan and nop PME requests. Upon enabling the context the input frame queue is left in an unscheduled (parked) state. This mode is only available if the api is built with control functionality and if the operating system has access to the PME's CCSR map.
- scan_exclusive_flow: This mode permits the user to send scan, nop, flow context write and flow context read PME requests. Upon enabling the context the input frame queue is left in an unscheduled (parked) state. This mode is only available if the api is built with control functionality and if the operating system has access to the PME's CCSR map

Once initialized, a pme_context object cannot change modes. If a different mode is required, a new context must be allocated/initialized. The following diagram depicts the pme_context state diagram. Depending on the mode, not all possible states are attainable.



On every pme command, check if exclusive mode context. If yes, acquire exclusivity and set the pme's EFQC register with the corresponding ctx.fqin.

Figure 83. PME Context State Diagram

The PME high-level APIs use the QMan high-level APIs. As such, the driver maintains control over certain fields, like the "contextB" field in the frame descriptor, and these are not available for direct application use. Applications can specify their own per-command context information. This is done by embedding the per-command token field in a larger structure which is returned back in the command result.

5.2.6.2.7.3.1.2 Typical Scan Operation Flow

Typical steps required to setup and perform scanning operations:

- Initialize a pme context via `pme_ctx_init()`.
- Enable the ctx via `pme_ctx_enable()`
- If operating in flow pme, setup the flow context using `pme_sw_flow_new()`, setting the appropriate fields and then invoking `pme_ctx_ctrl_update_flow()`. The flow is released on teardown via `pme_sw_flow_free()`.
- The context is ready for scanning operations. Build a frame description object and invoke the scan operation via `pme_ctx_scan()`.
- Upon receiving the result from the PME, the callback function specified in the pme context object will be invoked in interrupt context in the Linux kernel. For USDPAA, the callback is invoked in the application context.
- Process the resulting output frame description object passed into the callback function.
- Once scanning is done, disable the context via `pme_ctx_disable()`
- Release all resources in the context via `pme_ctx_finish()`.

5.2.6.2.7.3.1.3 Statistics Management

The high-level driver also provides an interface to read PME statistics counters. The values returned are software accumulated versions of the counter. The driver maintains the statistic counters by periodic (user-programmable) accesses to the read-

reset hardware counters. The access have to be frequent enough to prevent the counters from rolling over. The accumulated statistics are available to the user space application via the sysfs interface.

5.2.6.2.7.3.2 PME Low-level Driver API

The low level API is a hardware abstraction layer (HAL) where users' have complete control over structures used to communicate with the PME. This includes initialization of the input and output frame queues and the ownership for setting up the frame descriptor including command codes and the "contextB" field. The low level API is used by the high level driver for communicating with the PME.

The low-level interface provides the following;

- allocation/deallocation of residue data
- allocation/deallocation of hw flow context data
- allocation/deallocation of sw flow context data
- pme frame queue command/status abstraction
- frame descriptor setup for following pme commands: *nop*, *flow context write*, *flow context read*, *pmtcc* and *scan*.

The low level PME driver APIs do not encapsulate QMan portal interactions. Users are assumed to be using the low level PME APIs to have control on their interaction with the devices - including the use of QMan and BMan low level APIs.

5.2.6.2.7.4 Driver Build Configuration

The PME driver is built using the standard build process. The build configuration options and default values are specified in: `<linux directory>/drivers/staging/fsl_pme2/Kconfig`.

5.2.6.2.8 Data Scan Sample Application

The data scan sample application covered in this chapter are as follows:

- *Pattern matcher scan demo application*. A command line Linux user-space application that scans data strings given on a command line parameter or data from a file.
- *Snort*. An open-source network intrusion prevention and detection system using a rule-driven language, which combines the benefits of signature, protocol, and anomaly-based inspection methods.

5.2.6.2.8.1 Pattern Matcher Scan Demo Application (pm_scan_demo)

```
pm_scan_demo
```

is a sample command line Linux user-space application that scans data strings given on a command line parameter or data from a file. The data is scanned for patterns that are preloaded to the Pattern Matcher hardware database by the Pattern Matcher manager application (PMM). The pattern match results are displayed to standard output. This application demonstrates the use of Pattern Matcher hardware assist to scan for complex regexes and stateful rules.

5.2.6.2.8.1.1 Usage

The `pm_scan_demo` command line parameters are described in the help text shown here. The subsections provide simple examples to demonstrate the use of these parameters and how they apply to the Pattern Matcher features.

```
# pm_scan_demo --help
Usage:
    pm_scan_demo [<options>]
Description:
    A sample application that scans user entered string or file, and
    prints the reported matches.
Options:
    -s, --str <string>
        Scan a string, passthru mode only.
```

```

    May use \xNN to represent a byte in hexadecimal.
-f, --file <filename>
    Scan the content of a file.
    Multiple strings and files may be scanned sequentially as
    separate scan units by repeating the -f and -s options.
-r, --residue
    Enable residue feature.
--set <num>
    Expression set to scan against. Default = 0.
--subset <num>
    Expression subset-mask to scan against. Default = 0xffff.
--session <num>
    Stateful rule session id number. Default = 1.
--verbose
    Enable verbose report. HW will report more data per match.
--silent
    Disable printing of match report.
--detail
    Shows scan data covered by matches. Requires ANSI/VT100 terminal.
Example:
    pm_scan_demo -s abcd -s defg --residue --detail

```

5.2.6.2.8.1.1.1 Simple Literal String Scan

The 'pm_scan_demo: Simple Scan of Literal Pattern' example that follows uses the PMM application to add a simple regex to scan for the string

```
abcd
```

and uses the

```
pm_scan_demo
```

application to demonstrate the match result. Notice that the match result, highlighted in bold, shows an offset of 0x0:8, which is

```
workunit_offset_within_stream:match_offset_within_workunit
```

Also, notice that the

```
match_offset
```

is at the end of the match (not the start of the match), which is an offset of 8 in this example.

pm_scan_demo: Simple Scan of Literal Pattern

```

# cd <path of the pattern matcher applications>
# <path>pmm
Successfully created the PMM DB.
pmm> add regex name p1 exp /abcd/i tag=0x100
Successfully compiled the expression.
Successfully added one regex to the PM DB with handle 0.
Command execution time: 00:00:00 [hour:min:sec].
pmm>
pmm> show exp all
name=p1          expression="/abcd/"  options="i = 1, m = 0, s = 0, report = 1, tag
= 0x00000100"
total number of expressions = 1
total number of rules      = 0
pmm> commit

```

```
Successfully committed changes made to the data base of expressions.  
Command execution time: 00:00:00 [hour:min:sec].  
# <path>pm_scan_demo -s "xyz abcd xyz"  
#00: Scanning 12 bytes.  
match(0x01): len=0x04 offset=0x000000000000:00000008 tag=0x00000100  
Number of successful scan:      1  
Number of full match:          1  
Number of inconclusive match:  0  
Number of rule report:         0
```

5.2.6.2.8.1.1.2 Use of Residue

The 'pm_scan_demo: Scan of Literal Pattern that Span Multiple Work-Units' example that follows uses the PMM application to add a simple regex to scan for the string abcd and uses the pm_scan_demo application to demonstrate detection of patterns that cross work units. Notice that the second match result, highlighted in bold letters, shows two matches. The first match crosses between the two work units.

pm_scan_demo: Scan of Literal Pattern that Span Multiple Work-Units

```
# pmm  
Successfully created the PMM DB.  
pmm> add regex name p1 exp /abcd/i set=0 tag=0x100  
Successfully compiled the expression.  
Successfully added one regex to the PM DB with handle 0.  
Command execution time: 00:00:00 [hour:min:sec].  
pmm> commit  
Successfully committed changes made to the data base of expressions.  
Command execution time: 00:00:00 [hour:min:sec].  
pmm> quit  
Terminating the PMM application.  
# <path>pm_scan_demo --set=0 -s "abxyz abcd abxyz abcd xyz"  
#00: Scanning 25 bytes.  
match(0x01): len=0x04 offset=0x000000000000:0000000a tag=0x00000100  
match(0x01): len=0x04 offset=0x000000000000:00000015 tag=0x00000100  
Number of successful scan:      1  
Number of full match:          2  
Number of inconclusive match:  0  
Number of rule report:         0  
#  
# pm_scan_demo --set=0 --residue -s "abxyz ab" -s "cd abxyz abcd xyz"  
#00: Scanning 8 bytes.  
( No match or rule reported. )  
#01: Scanning 17 bytes.  
match(0x01): len=0x04 offset=0x000000000008:00000002 tag=0x00000100  
match(0x01): len=0x04 offset=0x000000000008:0000000d tag=0x00000100  
Number of successful scan:      2  
Number of full match:          2  
Number of inconclusive match:  0  
Number of rule report:         0  
#
```

5.2.6.2.8.1.1.3 Use of Pattern Set and Subsets

The 'pm_scan_demo: Scan of Patterns in Different Sets' example that follows uses the PMM application to add regexes in different sets and subsets, and the 'pm_scan_demo: Scan of Patterns in Subsets' example that follows uses the pm_scan_demo application to demonstrate scanning for patterns in sets and subsets. Notice that the match results, highlighted in bold letters, show matches appropriate to the pattern set or subsets.

pm_scan_demo: Scan of Patterns in Different Sets

```

# pmm
Successfully created the PMM DB.
pmm>
pmm> add regex name p1 exp /abcd/set=1 tag=0x100
Successfully compiled the expression.
Successfully added one regex to the PM DB with handle 0.
Command execution time: 00:00:00 [hour:min:sec].
pmm>
pmm> add regex name p2 exp /pqrst/set=2 tag=0x200
Successfully compiled the expression.
Successfully added one regex to the PM DB with handle 0.
Command execution time: 00:00:00 [hour:min:sec].
pmm>
pmm> commit
Successfully committed changes made to the data base of expressions.
Command execution time: 00:00:00 [hour:min:sec].
pmm>
pmm> quit
Terminating the PMM application.
# pm_scan_demo --set 1 -s "xyz abcd xyz pqrst xyz"
#00: Scanning 22 bytes.
match(0x01): len=0x04 offset=0x000000000000:00000008 tag=0x00000100
Number of successful scan:    1
Number of full match:        1
Number of inconclusive match: 0
Number of rule report:       0
# pm_scan_demo --set 0 -s "xyz abcd xyz pqrst xyz"
#00: Scanning 22 bytes.
( No match or rule reported. )
Number of successful scan:    1
Number of full match:        0
Number of inconclusive match: 0
Number of rule report:       0
# pm_scan_demo --set 2 -s "xyz abcd xyz pqrst xyz"
#00: Scanning 22 bytes.
match(0x01): len=0x05 offset=0x000000000000:00000012 tag=0x00000200
Number of successful scan:    1
Number of full match:        1
Number of inconclusive match: 0
Number of rule report:       0
#

```

pm_scan_demo: Scan of Patterns in Subsets (continued)

```

# pmm
Successfully created the PMM DB.
pmm>
pmm> add regex name p1 exp /abcd/set=1 subsets=0x1 tag=0x100
Successfully compiled the expression.
Successfully added one regex to the PM DB with handle 0.
Command execution time: 00:00:00 [hour:min:sec].
pmm>
pmm> add regex name p2 exp /pqrst/set=1 subsets=0x2 tag=0x200
Successfully compiled the expression.
Successfully added one regex to the PM DB with handle 0.
Command execution time: 00:00:00 [hour:min:sec].
pmm>

```

```
pmm> commit
Successfully committed changes made to the data base of expressions.
Command execution time: 00:00:00 [hour:min:sec].
pmm>
pmm> quit
Terminating the PMM application.
# pm_scan_demo --set 1 --subset 0x1 -s "xyz abcd xyz pqrst xyz"
#00: Scanning 22 bytes.
match(0x01): len=0x04 offset=0x000000000000:00000008 tag=0x00000100
Number of successful scan: 1
Number of full match: 1
Number of inconclusive match: 0
Number of rule report: 0
# pm_scan_demo --set 1 --subset 0x2 -s "xyz abcd xyz pqrst xyz"
#00: Scanning 22 bytes.
match(0x01): len=0x05 offset=0x000000000000:00000012 tag=0x00000200
Number of successful scan: 1
Number of full match: 1
Number of inconclusive match: 0
Number of rule report: 0
# pm_scan_demo --set 1 --subset 0x3 -s "xyz abcd xyz pqrst xyz"
#00: Scanning 22 bytes.
match(0x01): len=0x04 offset=0x000000000000:00000008 tag=0x00000100
match(0x01): len=0x05 offset=0x000000000000:00000012 tag=0x00000200
Number of successful scan: 1
Number of full match: 2
Number of inconclusive match: 0
Number of rule report: 0
```

5.2.6.2.8.1.2 Source Code Description

The demo application source code is in the `pm_scan_demo.c` file. The basic operations of the application are as follows:

1. Read the input string from the command line or the content of the specified file into memory.

The entire file is read into memory, so the file size is limited by the available memory.

2. Open the Pattern Matcher scanner device using the PM device driver APIs:

```
scan_fd = open(PME_DEV_SCAN_PATH, O_RDONLY)
```

3. Initialize the scanner device parameters such as `set`, `subset`, session ID in a local `scanner_params` structure and then configure the scanner device with these parameters.

```
ioctl(scan_fd, PMEIO_SETSCAN, &scanner_params)
```

4. Set up a structure to pass scan data, size of scan data, and other required parameters for the scan operation. The structure type is `struct pme_scan`. Then call the driver to execute the scan operation.

```
ioctl(scan_fd, PMEIO_SCAN, &pme_oper)
```

5. After the scan operation, the result of the scan such as match report buffer length and error flags are also returned in members of the `struct pme_scan` structure.

6. This demo uses BMan buffers to receive match reports. These free buffers should be freed with this call:

```
ioctl(scan_fd, PMEIO_RELEASE_BUFS, &pme_oper.result.output.data)
```

5.2.6.2.9 Software components

The software modules that accompany the Pattern Matcher 2.0 hardware are both production quality and sample software modules. The production and sample quality code is defined as follows.

Production-quality code:

- Completeness; includes necessary functionality to make it usable in a production environment
- Robustness; includes all necessary recovery functionality
- Fully validated/verified
- Supported

Sample code:

- Software example (or template) to help customers/partners to implement their own versions
- May or may not be of production quality but not often viewed as production quality
- Sometimes viewed as bare-bones (does the minimum functions)

5.2.6.2.9.1 Software Modules

The PM Software Modules table lists software version 2.0 software modules with packaging information and the supported platforms for each module.

PM Software Modules

Module	Packaging	OS	Platform Supported
Regex Compiler	source code	Linux	QorIQ™, x86
Stateful Rule Compiler	source code	Linux	QorIQ™ x86
Linker-Loader	source code	Linux	QorIQ™
PM Control Interface	source code	Linux	QorIQ™
PM Driver	Kernel patches (source code) USDPAALibrary (source code)	Linux	QorIQ™

5.2.6.2.9.2 Sample Code

Below, the Pattern Matcher Sample Code table lists software version 2.0 sample code modules.

Pattern Matcher Sample Code

Module	Packaging	OS	Platform Supported
PM Manager (PMM)	Executable and source code	Linux	QorIQ™
Pattern scanner application (PM Scan Demo)	Executable and source code	Linux	QorIQ™
pmcd	Executable and source code	Linux	QorIQ™
pmcc	Executable and source code	Linux	QorIQ™

5.2.6.2.10 Pattern Matcher FAQ

5.2.6.2.10.1 What is it and why is it needed?

The Pattern Matcher Engine (PME, or sometimes just PM) offloads processing from the PowerQUICC core(s) by performing, in hardware, regular expression pattern matching of data against patterns stored in the pattern database. This type of unanchored searching of patterns is processing-intensive and is difficult to perform in software at multi-gigabit rates. Even in sub-gigabit rate deployments, the offload provided by the PME can free the core(s) to perform more complex, content-oriented tasks.

5.2.6.2.10.2 What is a regular expression?

A regular expression (or regex) is a description of a set of strings, written using a powerful and flexible notation/syntax. The expression may contain literal characters and meta-characters, supporting wildcards, grouping, quantification and alternation. Matching a regular expression refers to comparing the arrangement of symbols described by the regular expression against an input string, usually incoming network data.

5.2.6.2.10.3 What types of regular expressions are supported by the supplied compiler?

The first supplied compiler supports a major subset of the PERL regular expression syntax, as well as capabilities beyond that provided by PERL.

5.2.6.2.10.4 What is a pattern?

A pattern is an arrangement of symbols and instructions used by the PME to evaluate against an input string or SUI. It is the hardware representation of a regular expression.

5.2.6.2.10.5 What is an SUI?

A string under inspection (SUI) refers to the string of bytes to be searched by the PME. The PME evaluates this input string against patterns in the pattern database. Often, but not necessarily, the SUI refers to incoming network data.

5.2.6.2.10.6 Why is regular expression matching difficult to perform at multi-gigabit rates in software?

In order to match expressions against the incoming data, the incoming data must be scanned byte by byte and evaluated against the thousands of expressions in the database. The expressions not only contain literal characters, but may contain wildcards, repeats, alternations, and other regular expression constructs, where each expression may represent an almost infinite number of strings. Regular expression matching, therefore, requires not only matching of literal characters, but also what is known as regular expression evaluation.

5.2.6.2.10.7 What applications require pattern matching?

Pattern matching of application layer protocol signatures forms the basis for high-performance application-aware networking. It is used in various content processing and security applications such as intrusion detection/prevention, anti-virus, anti-spam, and other applications where the content of the packet or stream needs to be examined. Regular expressions are gaining widespread adoption over literal string matching for these types of applications because of their expressive power and flexibility to describe complex patterns.

5.2.6.2.10.8 Can the PME scan packet headers as well as packet content?

Not easily (nor is this usually required to be performed in hardware). The PME was designed for the more difficult task of scanning character-based content for unanchored matches. Scanning packet headers requires bit-level granularity, making it impractical for the byte-based PME to perform. However, since packet headers are typically anchored, it is often reasonable to delegate this task to a software pre-processing stage.

5.2.6.2.10.9 Can the PME detect patterns across packet boundaries?

Yes. The PME uses the residue method to detect cross-packet patterns. The residue method is stateless and therefore can scale as the number of patterns grow. There is no hard limit on how many packets the pattern may span (other than the limit imposed by the maximum length of the pattern), or whether the content in the first packet matches the start of one or the start of many patterns.

5.2.6.2.10.10 What is the maximum length of a pattern?

128 bytes long. Patterns in excess of 128 bytes can be supported through stateful rules functionality.

5.2.6.2.10.11 How many patterns can be searched for simultaneously?

32000.

5.2.6.2.10.12 What is an NFA?

Non-deterministic finite automaton (NFA) is one of two generally accepted methods for performing pattern matching. The NFA evaluates each alternative at every byte position of the incoming data. Unlike DFA, an NFA may have multiple transitions out of a state with the same input. As the NFA evaluates each transition, it must remember and "backtrack" to execute alternate transitions if the evaluation fails.

5.2.6.2.10.13 What is a DFA?

Deterministic finite Automaton (DFA) is one of two generally accepted methods for performing pattern matching. A DFA tracks all possible matches as each byte of data is consumed. Each possible input character of the alphabet set is associated with at most one transition in any given state.

5.2.6.2.10.14 Which method is better: NFA or DFA?

There is no single correct answer. An NFA typically supports more regex features but requires many passes over the data as each alternative is evaluated, whereas a DFA only passes over the data once. Part of the reason a DFA only needs to examine the data once is that a lot of work has been performed up front during the compilation of the expressions, such that the compiled DFA state graph contains all the interdependencies between the different expressions. Because of these interdependencies, an incremental change to the expression database may require a recompilation of the entire DFA state graph. For these reasons, a software-based DFA has less flexibility, requires more memory, and has a longer expression compile time, but tends to perform faster than a software-based NFA.

5.2.6.2.10.15 Is the PME an NFA or a DFA?

Since the drawbacks of a DFA, such as the longer compile time, are inherent and cannot be easily overcome, the PME was modeled after an NFA, augmented with specific techniques to improve performance. Part of the reason a traditional NFA is slower is because the NFA needs to evaluate each expression at every byte position of the incoming data. When there are a large number of expressions or alternatives, performance drops. To overcome this, the PME incorporates a hardware pre-processing stage called the Key Element Scanner, which works by eliminating almost all candidate expressions prior to the regex evaluation such that the NFA only needs to perform the regex evaluation for one or a very small number of expressions at any given byte position. In fact, under normal operation when there is no potential match, the regex evaluation does not even need to be performed.

5.2.6.2.10.16 Where is the pattern database stored?

The pattern database is stored in a combination of on-chip memory and the main SDRAM.

5.2.6.2.10.17 Why is SRAM not required?

One key design objective of the PME is to minimize any reliance on expensive low-latency memories. The solution the design team undertook to achieve this objective makes use of a combination of on-chip and external memory. On-chip memory contains just enough pattern information to filter out all but the most likely matches. Only when a likely match has occurred does the PME require an external SDRAM memory access. Pipeline stalls due to memory accesses are minimized with the presence of FIFOs in between the different internal components of the PME. This relative infrequency of external memory access, in addition to the interconnecting FIFOs, leads to performance with a lower dependency on memory latency.

5.2.6.2.10.18 Does the PME support case-insensitivity?

Yes. Support for case-insensitivity is inherent in the design, and does not lead to pattern explosion.

5.2.6.2.10.19 Can contextual searches be performed?

The situation where interest for a specified pattern is predicated upon the detection of another pattern can be performed using Stateful rules.

5.2.6.2.10.20 What is a stateful rule?

A Stateful rule is a set of user-defined actions that are executed by the PME when specified pattern match events occur. The actions include changing state, assignments, bitwise operations, addition, subtraction, and relational operations.

5.2.6.2.10.21 What are the benefits of stateful rules?

Stateful rules enable the PME to go beyond basic pattern matching. Rather than just informing software whenever a pattern is matched, stateful rules allow the PM to act on pattern match events such that a match is declared to software only when user-specified conditions are met. For example, a stateful rule can be written to allow the PME to discriminate between matches found in the header portion versus the body portion of the application message. A stateful rule can allow the PM to declare a match only when all constituent parts of a signature match (such as in an anti-virus application). A Stateful rule can allow the PM to track a normal protocol exchange and glean information between a client and a server (such as in a SIP-aware application retrieving information on the media channel). In all these examples, Stateful rules postpone the need for application software to be invoked until concrete actions need to be taken.

5.2.6.2.10.22 How many stateful rules are supported?

32,768.

5.2.6.2.10.23 What information does the pattern match and stateful rule reports contain?

The pattern match report contains information such as the byte offset of the detected pattern (right edge of the match) and the pattern identifier. The Stateful rule report is customizable, and can include more detailed information on the match, any captured fields, and any context previously saved.

5.2.6.2.10.24 What's the performance of the PME?

The raw hardware performance of the PME is 2.4 Gbps in the MPC8572 PowerQUICC processor. However, the actual system performance may vary due to software and memory access overhead.

5.2.6.2.10.25 What software is included and supported for the PME?

All the software needed to compile, link, and load expressions into the hardware are provided. This includes the regular expression compiler, the stateful rule compiler, the linker-loader, and the low-level drivers for interacting with the hardware.

5.2.6.2.10.26 What operating systems are supported?

In general, the supplied software has been designed to be agnostic of the operating system where possible. The regular expression compiler and the stateful rule compiler are supported on Linux; the linker-loader is supported for Linux; the low-level drivers for interacting with the PM are supported for Linux. Porting of the supplied software to other operating systems is not expected to be difficult.

5.2.6.2.10.27 How can existing pattern management application software use the PME?

The PME software is provided as a library of functions with clearly-defined APIs and capabilities to compile, link, and load the patterns onto the hardware. Both single expression and multi-expression inputs are supported. Some application-specific pre-processing may be required to convert existing expressions into the PERL-like format understood by the supplied compiler.

5.2.6.2.10.28 How does software invoke the PME?

The PME provides a DMA-based "look-aside" interface with four independent DMA channels, which the supplied driver software uses to command pattern matching/decompression work, and to receive notifications of completed pattern matching/decompression work. For each DMA channel, a ring structure (array of entry locations) of programmable depth is used to dispatch commands to the PME. Similarly, a ring structure (array of entry locations) of programmable depth is used by the PME to inform the software of events such as the availability of a pattern scan result. The PME supports "gather buffer lists" for specifying input data and "scatter buffer lists" for specifying the location of output data.

5.2.6.2.10.29 Can software perform other tasks while the PME is scanning the data?

Yes, the supplied driver supports both blocking and non-blocking APIs.

5.2.6.2.10.30 How does software obtain the match results?

Completion of a command can be signaled through an interrupt or through polling. Application software can then read the results from the notification FIFO, including any results in a scatter list.

5.2.6.2.11 Revision History

Document Revision History

Rev. Number	Date	Substantive Change(s)
2	8/2009	Added support for PME2.0 (supporting P4080).
1	4/2008	Removed sections 2.1, 2.2, 2.3, 3.1, and 3.2.
0	11/2007	Initial public release.

5.2.6.3 Pattern Matcher 2.0, 2.1, 2.2 Software API Reference Manual

5.2.6.3.1 Introduction

This reference manual provides detailed information about Pattern Matcher software application programming interfaces (APIs).

This manual is intended for designers developing software for the NXP Pattern Matcher. Readers are expected to be familiar with Pattern Matcher hardware functionality and the information provided in the Pattern Matcher 2.0 Software User's Guide prior to using this document.

All programming interfaces available for use with the Pattern Matcher software are listed in this document. The interfaces are grouped according to software components within the document. Each interface includes a functional description as well as information about syntax and return values. Each interface description also provides information about libraries and kernel modules as well as header files required to use that programming interface.

The Pattern Matcher software is provided as linkable C libraries and kernel drivers, as well as executable binaries where appropriate. This document describes the programming interface for the software. Software executables are described in the Pattern Matcher Software User Guide.

5.2.6.3.1.1 Conventions

The following notational conventions are used in this guide:

Courier monospaced type indicates code examples and file or directory names

Italic type used within interface descriptions indicates interface parameters

5.2.6.3.2 Software Component Overview

The NXP Pattern Matcher provides high-performance hardware pattern matching of data. Input data is searched for user specified patterns by the hardware. The Pattern Matcher software enables users to efficiently configure, use, and manage the Pattern Matcher hardware. The software comprises the following components:

- **Regex Compiler**-The regex compiler converts user defined patterns, specified as regular expressions, into patterns in a NXP hardware-specific format.
- **Stateful Rule Compiler**-Stateful rules define stateful relationships between pattern matching results and also specify the actions to be taken in the defined states. The stateful rule compiler converts user specified stateful rules into a NXP hardware-specific format.
- **Pattern Matcher Configuration** - Pattern Matcher Configuration software provides users with an interface to configure Pattern Matcher hardware. The simplest programmatic interface to configure the pattern matcher hardware is with the PMC API. The following two APIs are available for backward compatibility with existing pattern management applications.
- **Linker-Loader**-The linker-loader accepts patterns and stateful rules encoded in the NXP hardware-specific format and generates a hardware-optimal pattern and reaction database. The linker-loader also provides interfaces to configure and monitor the Pattern Matcher database used by hardware.
- **Control Interface**-The PME control interface module (PMCI) provides C functional interfaces to send and receive Pattern Matcher control commands to the Pattern Matcher through Pattern Matcher driver software. The PME control commands initialize PME driver and configure the PME hardware tables. The PMCI module converts complex PME control commands into PME driver primitives.
- **Driver**-The Pattern Matcher driver software provides users with an interface to the Pattern Matcher. The driver software is targeted to the linux kernel and linux user space (USDPAAs) targets. For Linux, the driver provides user-level and kernel-level interfaces to efficiently transfer, control, and scan data to the hardware, and provide pattern matching results to applications. All PME database updates are expected to be done on a control core running Linux. For USDPAAs, the driver provides an interface to scan data and get match results. In all cases, the PME driver encapsulates direct interaction with the underlying QMan and BMan functionality. There is also a platform specific considerations to bear in mind when working with the interfaces described here, please see

NOTE

The APIs in this document are grouped according to above software components. Additional information about each software component is provided in the appropriate section.

5.2.6.3.3 Regular expression compiler

This chapter describes the application programming interface (API) for the Pattern Matcher regular expression compiler (PMREC). The PMREC allows the user to compile regular expression patterns into a format that is used internally within the NXP Pattern Matcher hardware. All the PMREC functions run to completion, meaning a PMREC function returns the operation performed by that function has been completed.

PMREC is delivered as a linkable C library called libregex.a.

PMREC Required Files

File	Description
libregex.a	Contains all the PMREC objects
pmrec.h	Defines the interface to the PMREC module
generic_types.h	Contains generic type definitions and is included by pmrec.h
pm_defs.h	Defines the interface that is common to PMLL, PMREC, PMSRC, and so on, and is included by pmrec.h
inttypes.h	Defines the basic integer types and is included by generic_types.h
stdbool.h	Defines the bool type and is included by generic_types.h

5.2.6.3.3.1 pmrec_compile

pmrec_compile-Compile a file of regular expressions.

Synopsis

```
#include <pmrec.h>

pmrec_error_codes_t pmrec_compile (

char          *input_file_name_p,

char          *output_file_name_p,

pmrec_module_options_t *options_p,

char          **compile_msg_p);
```

Description

Compile regular expression(s) into NXP hardware format. The user must supply valid input and output file names. The user must also supply two optional parameters (either set to a valid value or set to NULL). The first is an object that contains compile options. The second is a pointer to a character pointer that allows the compiler to pass back a detailed compiler message. Both of the last two parameters may be set to NULL if default options and no compiler message are desired. If a pointer is supplied and no compiler message is generated, the string will be set to NULL. Note that if a pointer is supplied for the compiler message, the user is responsible for freeing the string that is returned by the compiler library.

pmrec_module_options_t is defined in pmrec.h. It is listed below for easy reference.

typedef struct pmrec_module_options

```
{
    pmrec_debug_levels_t      debug_level;
    pmrec_group_definition_t  group_definition;
    char                      group_def_filename_p;
    pmrec_equivalence_definition_t equivalence_definition;
    char                      *equiv_def_filename_p;
    bool                      warnings_are_errors;
    bool                      suppress_warnings;
    bool                      hide_strings;
    bool                      one_byte_triggers;
    bool                      silicon_8572_rev_1_0;
} pmrec_module_options_t;
```

group_def_filename_p and equiv_def_filename_p are for future use and should be set to NULL.

A typical setting for the options would be:

```
pmrec_module_options_t options = { pmrec_debug_none_e,
    pmrec_groups_default0_e,
    NULL,
    pmrec_equivalence_default0_e,
    NULL,
    false,
    false};
```

Return Value

This function returns a code of type *pmrec_error_codes_t*. This type is defined in *pmrec.h*. The possible return codes are as follows:

pmrec_ok_e

Success. Generally expect to see this code.

pmrec_warning_e

Compile warning found.

`pmrec_internal_error_e`

Should not see this error code. Indicates an unexpected condition.

`pmrec_no_trigger_found_e`

No triggerable symbols found in expression

`pmrec_missing_end_block_e`

Should not see this error code. Indicates an internal issue with the data structures.

`pmrec_missing_end_class_e`

Should not see this error code. Indicates an internal issue with the data structures.

`pmrec_no_output_file_e`

No output file was specified.

`pmrec_nested_repeat_found_e`

A nested repeat in the expression was found. Hardware does not support this.

`pmrec_expression_too_large_e`

Expression is too large to fit within the allowable instructions.

`pmrec_nested_capture_found_e`

A nested capture in the expression was found. Hardware does not support this.

`pmrec_compile_failed_e`

Generic compile failure. Specific details will be in the compile message.

`pmrec_input_file_open_error_e`

Error opening supplied input file.

`pmrec_invalid_element_size_e`

Should not see this error. Indicates an internal issue with data structures.

`pmrec_output_file_open_error_e`

Error opening supplied output file.

`pmrec_output_file_write_error_e`

Error writing to output file.

`pmrec_max_test_lines_exceeded_e`

Expression generates too many test lines to fit in hardware.

`pmrec_no_start_position_found_e`

Should not see this error. Indicates an issue finding start position for test lines.

`pmrec_code_string_mismatch_e`

Should not see this error. Indicates erroneous number of error code strings.

`pmrec_must_expand_grouped_duplication_e`

Indicates a condition the compiler cannot deal with currently. Specifically expressions like `(ab(cldle))+`. This should be expanded to `(abclabdlabe)+`

`pmrec_malloc_failure_e`

Failure while calling malloc. System memory likely exhausted.

`pmrec_invalid_option_e`

Invalid option was passed through compiler options.

pmrec_invalid_group_def_e

Invalid group definition was passed through compiler options.

pmrec_invalid_equiv_def_e

Invalid equivalence definition was passed through compiler options.

pmrec_syntax_error_e

Regex format or syntax was invalid

Related information

[pmrec_get_error_string](#) on page 386

5.2.6.3.3.2 pmrec_get_error_string

pmrec_get_error_string-Get the string associated with a compiler error.

Synopsis

```
#include <pmrec.h>

const char *pmrec_get_error_string (pmrec_error_codes_t, code);
```

Description

The function returns a brief string describing the supplied error code.

Return Value

Pointer to the error string.

Related information

[pmrec_compile](#) on page 383

5.2.6.3.4 Stateful rule compiler

This chapter provides a description of the API for the Pattern Matcher stateful rule compiler (PMSRC). The PMSRC compiles stateful rules written in a NXP stateful rule source code language into a format that is used internally within the NXP pattern matcher hardware. All the PMSRC functions run to completion, meaning a PMSRC function returns the operation performed by the function has been completed.

The PMSRC is delivered as a linkable C library called libstatefulRules.a.

PMSRC Required Files

libstatefulRules.a	Contains all the PMSRC objects.
pmsrc.h	Defines the interface to the PMSRC module
generic_types.h	Contains the generic type definitions and is included by pmsrc.h
pm_defs.h	Defines the interface that is common to PMLL, PMREC, PMSRC, etc., and is included by pmsrc.h
inttypes.h	Defines the basic integer types and is included by generic_types.h

Table continues on the next page...

Table continued from the previous page...

stdbool.h	Defines the bool type and is included by generic_types.h
-----------	--

5.2.6.3.4.1 pmsrc_compile

pmsrc_compile-Compile a file of stateful rules.

Synopsis

```
#include <pmsrc.h>

pmsrc_ErrorCodes_t pmsrc_compile (

    char    *input_file_name_p,

    char    *output_file_name_p,

    pmsrc_module_options_t *options_p,

    char **compile_msg_p);
```

Description

Compile stateful rule(s) into NXP hardware instructions. The user must supply valid input and output file names. The user must also supply two optional parameters (set to a valid value or set to NULL). The first is an object that contains compile options. The second is a pointer to a character pointer that allows the compiler to pass back a detailed compiler message. Both of the last two parameters may be set to NULL if default options and no compiler messages are desired. If a pointer is supplied and no compiler message is generated, the string will be set to NULL. Note that if a pointer is supplied for the compiler message, the user is responsible for freeing the string that is returned by the compiler library.

pmsrc_module_options_t is defined in pmsrc.h, listed below for easy reference.

typedef struct pmsrc_module_options

```
{

    pmsrc_debug_levels_t    debug_level;

    pmsrc_report_pad_t     report_pad;

    uint32_t                string_pad;

    pmsrc_report_cnst_sz_t  report_cnst_sz;

    pmsrc_match_type_t     allow_inconclusive;

    bool                    warnings_are_errors;

    bool                    suppress_warnings;

} pmsrc_module_options_t;
```

A typical setting for the options is as follows:

```
pmsrc_module_options_t options = {  
  
    pmsrc_debug_off_e,  
  
    pmsrc_report_pad4_e,  
  
    PMSRC_DEFAULT_STRING_PAD,  
  
    pmsrc_report_cnst_sz4_e,  
  
    pmsrc_conclusive_only_matches_e,  
  
    false,  
  
    false};
```

Return Value

This function returns a code of type *pmsrc_error_codes_t*. This is defined in *pmsrc.h*. The possible return codes are as follows:

pmsrc_ok_e

Success. Generally expect to see this code.

pmsrc_warnings_e

Compile warning found. See compiler message for details.

pmsrc_input_file_open_e

Trouble opening supplied input file.

pmsrc_output_file_open_e

Trouble opening supplied output file.

pmsrc_no_output_file_e

No output file supplied.

pmsrc_compile_failed_e

Compile failed. See compiler message for details.

pmsrc_null_list_pointer_e

Should not see this error. Indicates internal issues with data structures.

pmsrc_no_output_written_e

No output was written to output file.

pmsrc_code_string_mismatch_e

Should not see this error. Indicates incorrect number of error strings.

pmsrc_undefined_state_name_e

A state name was referred to in the stateful rule that does not exist.

pmsrc_internal_error_e

Should not see this error. Indicates an unexpected internal condition.

`pmsrc_malloc_failure_e`

Failure while calling malloc. System memory likely exhausted.

`pmsrc_empty_stack_e`

Should not see this error. Indicates issue with internal data structures.

`pmsrc_name_exists_e`

Rule name used more than once in file.

`pmsrc_empty_input_e`

Empty input file.

`pmsrc_unrecognized_debug_option_e`

Unrecognized debug option. Allowed options are as follows:

`pmsrc_debug_off_e`

`pmsrc_debug_summary_e`

`pmsrc_debug_test_e`

`pmsrc_unrecognized_report_pad_option_e`

Unrecognized report pad option. Allowed options are:

`pmsrc_report_pad_none_e`

`pmsrc_report_pad4_e`

`pmsrc_unrecognized_string_pad_option_e`

Unrecognized string pad option. Allowed values are from 0 to `PMSRC_MAX_STRING_PAD_SIZE`.

`pmsrc_invalid_option_e`

Unrecognized option.

`pmsrc_string_pad_option_too_large_e`

String pad option too large. Maximum defined as `PMSRC_MAX_STRING_PAD_SIZE`.

`pmsrc_unrecognized_report_constant_size_option_e`

Unrecognized report constant size. Allowed options are as follows:

`pmsrc_report_cnst_sz2_e`

`pmsrc_report_cnst_sz4_e`

`pmsrc_report_cnst_sz6_e`

`pmsrc_report_cnst_sz8_e`

`pmsrc_unrecognized_allow_conclusive_option_e`

Unrecognized value for allow conclusive option. Allowed values are as follows:

`pmsrc_conclusive_only_matches_e`

`pmsrc_conclusive_and_inconclusive_matches_e`

Related information

[pmsrc_get_error_string](#) on page 389

5.2.6.3.4.2 pmsrc_get_error_string

`pmsrc_get_error_string`-Get the string associated with a compiler error.

Synopsis

```
#include <pmsrc.h>

const char *pmsrc_get_error_string(pmsrc_error_codes_t code);
```

Description

The function returns a brief string describing the supplied error code.

Return Value

Pointer to the error string.

Related information

[pmsrc_compile](#) on page 387

5.2.6.3.5 Pattern Matcher configuration (PMC) API

This chapter provides a description of the Pattern Matcher configuration (PMC) API.

The PMC provides a programming interface to Pattern Matcher management applications for adding, removing, querying and committing regular expressions and stateful rules into Pattern Matcher hardware.

Pattern Matcher Configuration API supports:

- multiple, independant applications adding/deleting patterns and rules
- a persistent Pattern database
- Handle statistics

PMC is implemented as a user-space header file and a library. The steps involved in configuring patterns in the Pattern Matcher hardware are as follows:

instantiate PMC Daemon (PMCD) process. The PMCD is provided as a Linux executable binary for the target SoC which includes Pattern Matcher functionality (e.g. P4080 or MPC8572).

add regular expressions from a file via a call to `pmc_add_expr_file`

optionally add stateful rules from a file via call to `pmc_add_rule_file`

commit changes to hardware via call to `pmc_commit`

All the PMC functions run to completion, meaning that a PMC function returns the operation performed by the function has been completed.

PMC required files

libpmc.a	Contains all the PMC objects
pmc.h	Defines the interface to the PMC module
pmcd	PMC Deamon

5.2.6.3.5.1 Status codes

```
typedef struct pmc_status {

    pmc_ok_e                = 0,
```

```

pmc_init_fail_e      = 1,

    pmc_connect_fail_e      = 2,

    pmc_send_fail_e        = 3,

    pmc_recv_fail_e        = 4,

    pmc_commit_fail_e      = 5,

    pmc_open_file_fail_e   = 6,

    pmc_record_read_fail_e = 7,

    pmc_unsupported_pme_ver_e = 8,

    pmc_unsupported_si_ver_e = 9,

    pmc_incompatible_si_ver_e = 10,

    pmc_no_memory_e        = 11,

    pmc_record_add_fail_e   = 12,

    pmc_delete_fail_e      = 13,

    pmc_query_fail_e       = 14,

    pmc_set_var_trig_fail_e = 15,

pmc_analysis_fail_e = 16,

} pmc_status_t;

```

5.2.6.3.5.2 Data structures

```

/* Expressions */
typedef struct pmc_expr {
    char          *name_p;
    char          *expr_str_p;
    char          *option_str_p;
    uint32_t      db_index;
    struct pmc_expr *next_p;
} pmc_expr_t;
/* Rules */
typedef struct pmc_rule {
    char          *name_p;
    uint32_t      num_reactions;
    char          **expr_names;
    uint32_t      db_index;
} pmc_rule_t;
/* Statistics */
typedef struct pmc_stats {
/* =====
 *
 *          Pattern Matcher hardware stats
 *
 * =====

```

```

* =====*/
/* The number of input bytes reported by KES */
uint64_t pm_input_bytes;
/* The number of output report bytes reported by SRE */
uint64_t pm_output_bytes;
/* The number of one byte trigger hits reported by KES */
uint64_t pm_trigger_one_byte_hits;
/* The number of two byte trigger hits reported by KES */
uint64_t pm_trigger_two_byte_hits;
/* The number of variable byte trigger hits reported by KES */
uint64_t pm_trigger_variable_hits;
/* The number of special trigger hits reported by KES */
uint64_t pm_trigger_special_hits;
/* The number of confidence stage hits reported by KES */
uint64_t pm_confidence_hits;
/* The number of matches reported by DXE */
uint64_t pm_matches;
/* The number of SR executions triggered by DXE reported by SRE */
uint64_t pm_dxe_executions;
/* The number of SR executions triggered by End of SUI reported by SRE */
uint64_t pm_end_of_sui_executions;
/* The number of SUIs with matches reported by DXE */
uint64_t pm_sui_matching_patterns;
/* The number of SUIs that generate reports reported by SRE */
uint64_t pm_sui_generating_reports;
/* The number of input SUIs reported by KES */
uint64_t pm_input_suis;
/* The number of matches with DRCC reported by DXE */
uint64_t pm_selected_matches;
/* =====
*
* Deflate hardware stats
* =====*/
/* The number of deflate input bytes reported by DFL */
uint64_t df_input_bytes;
/* The number of deflate output bytes reported by DFL */
uint64_t df_output_bytes;
/* The number of deflate work units reported by DFL */
uint64_t df_decompressions;
/* =====
*
* Linker Loader software stats
* =====*/
/* -----
*
* DXE/SRE table statistics.
* -----*/
/* The total number of the DXE/SRE entries present in the DXE/SRE
* table, i.e., this is the DXE/SRE table size. This value is
* specified in a call to the pml1_db_create() function. */
uint32_t dxeSreEntryNum;

/* The number of the base entries in the DXE/SRE table. For a given
* release of PMLL this number is constant. This number is also the
* minimum size of the DXE/SRE table. */
uint32_t dxeSreBaseEntryNum;

/* The number of the DXE/SRE extension entries, i.e., the number of
* the "non-base" entries in the DXE/SRE table. */
uint32_t dxeSreExtensionEntryNum;

/* The number of the currently allocated extension entries. */
uint32_t dxeSreAllocatedExtensionEntryNum;

```

```

/* The number of the currently available extension entries. */
uint32_t  dxoSreAvailableExtensionEntryNum;
/* -----
 *           SRE session statistics.
 * -----*/
/* The number of the SRE sessions. This value is specified in a
 * call to the pml1_db_create() function. */
uint32_t  sreSessionCtxNum;

/* The size (in bytes) of each of the SRE sessions. This value is
 * specified in a call to the pml1_db_create() function. */
uint32_t  sreSessionCtxSize;

/* The size (in bytes) of the session digest area. This value
 * depends on the number of SRE rules as specified by the user in
 * a call to the pml1_db_create() function. */
uint32_t  sreSessionDigestSize;

/* The size (in bytes) of the session flags area. This value is
 * constant for a given release of PMLL. */
uint32_t  sreSessionFlagsSize;

/* The size (in bytes) of the session context areas. */
uint32_t  sreSessionCtxAreaSize;

/* The size (in bytes) of the currently allocated session context areas. */
uint32_t  sreAllocatedSessionCtxAreaSize;

/* The size (in bytes) of the currently available session context areas. */
uint32_t  sreAvailableSessionCtxAreaSize;
/* -----
 *           Expression and pattern statistics.
 * -----*/
/* The maximum number of patterns that can be configured. This value is
 * constant for a given release of PMLL. */
uint32_t  patternMaxNum;

/* The number of the currently configured expressions. */
uint32_t  expNum;

/* The number of the currently configured "special" patterns. */
uint32_t  specialPatternNum;

/* The number of the currently configured "one-byte" patterns. */
uint32_t  oneBytePatternNum;

/* The number of the currently configured "two-byte" patterns. */
uint32_t  twoBytePatternNum;

/* The number of the currently configured "variable" patterns. */
uint32_t  variablePatternNum;

/* The total number of the currently configured patterns. */
uint32_t  totalPatternNum;

/* The currently configured variable trigger size. */
uint32_t  variableTriggerSize;
/* -----

```

```

*           Rule and reaction statistics.
* -----*/
/* The maximum number of stateful rules. This value is specified in
* a call to the pml1_db_create() function. */
uint32_t  statefulRuleMaxNum;

/* The maximum number of stateless rules. This value is constant
* for a given release of PMLL. */
uint32_t  statelessRuleMaxNum;

/* The maximum number of all rules. This value is constant for a
* given release of PMLL. */
uint32_t  totalRuleMaxNum;

/* The number of the currently configured stateless rules. */
uint32_t  statelessRuleNum;

/* The number of the currently configured stateful rules. */
uint32_t  statefulRuleNum;

/* The number of the currently configured rules. */
uint32_t  totalRuleNum;

/* The number of the currently configured end-of-SUI reactions. */
uint32_t  endOfSuiReactionNum;
}

```

5.2.6.3.5.2.1 Add regular expressions from compiled bin file

Adds all regular expressions in the precompiled binary file. Option to allow auto commit.

```

pmc_status_t pmc_add_expr_file(
                                char *filename_p,
                                bool auto_commit,
                                char **info_msg_p)

```

5.2.6.3.5.2.2 Add stateful rules from compiled bin file

Adds all stateful rules in the precompiled binary file. Option to allow auto commit.

```

pmc_status_t pmc_add_rule_file(
                                char *filename_p,
                                bool auto_commit,
                                char **info_msg_p)

```

5.2.6.3.5.2.3 Delete regular expression by name or set and subset

Deletes the regular expression referenced by name or set and subset. Option to allow auto commit.

```

pmc_status_t pmc_del_expr_name(
                                char *name_p,
                                bool auto_commit,
                                char **info_msg_p)

pmc_status_t pmc_del_expr_set(
                                int set,
                                int subset,
                                bool auto_commit,
                                char **info_msg_p)

```

5.2.6.3.5.2.4 Delete regular expressions by compiled bin file

Deletes all regular expressions in the precompiled binary file. Option to allow auto commit.

```

pmc_status_t pmc_del_expr_file(
                                                    char *filename_p,
                                                    bool auto_commit,
                                                    char **info_msg_p)

```

5.2.6.3.5.2.5 Delete stateful rule by name

Deletes the stateful rule referenced by name. Option to allow auto commit.

```

pmc_status_t pmc_del_rule_name(
                                                    char *name_p,
                                                    bool auto_commit,
                                                    char **info_msg_p)

```

5.2.6.3.5.2.6 Delete stateful rule by compiled bin file

Deletes all stateful rules in the precompiled binary file. Option to allow auto commit.

```

pmc_status_t pmc_del_rule_file(
                                                    char *filename_p,
                                                    bool auto_commit,
                                                    char **info_msg_p)

```

5.2.6.3.5.2.7 Commit changes to hardware

Commit all changes to hardware.

```

pmc_status_t pmc_commit(char **info_msg_p)

```

5.2.6.3.5.2.8 Delete all regular expressions and rules

Delete all expressions or all rules or both. Option to allow auto commit.

```

pmc_status_t pmc_del_all_expr(
                                                    bool auto_commit,
                                                    char **info_msg_p)

pmc_status_t pmc_del_all_rules(
                                                    bool auto_commit,
                                                    char **info_msg_p)

pmc_status_t pmc_del_all(
                                                    bool auto_commit,
                                                    char **info_msg_p)

```

5.2.6.3.5.2.9 Query regular expression by name or set and subset

Query information on a single named regular expression or a list of expressions based on set and subset. The information is returned in a single `pmc_expr_t` record or a linked list of `pmc_expr_t` records. See `pmc_free_expr` to cleanup.

```

pmc_status_t pmc_query_expr_name(
                                                    char *name_p,
                                                    pmc_expr_t **expr_p,
                                                    char **info_msg_p)

pmc_status_t pmc_query_expr_set(
                                                    int set,
                                                    int subset,
                                                    pmc_expr_t **expr_p,
                                                    char **info_msg_p)

```

5.2.6.3.5.2.10 Query stateful rule by name

Query information on a single named stateful rule. The information is returned in a `pmc_rule_t` record. See `pmc_free_rule` to cleanup.

```
pmc_status_t pmc_query_rule_name(
                                char *name_p,
                                pmc_rule_t **rule_p,
                                char **info_msg_p)
```

5.2.6.3.5.2.11 Query first regular expression in the database

Query the first regular expression in the database. The index of the expression is stored in the expression object. See `pmc_free_expr` to cleanup.

```
pmc_status_t pmc_query_expr_first( pmc_expr_t **expr_p,
                                   char **info_msg_p)
```

5.2.6.3.5.2.12 Query next regular expression in the database

Query the next regular expression in the database. This function will return the next expression based on the index of the current expression that is passed in. See `pmc_free_expr` to cleanup.

```
pmc_status_t pmc_query_expr_next (
                                uint32_t      index,
                                pmc_expr_t **next_expr_p,
                                char **info_msg_p)
```

5.2.6.3.5.2.13 Query first stateful rule in the database

Query the first stateful rule in the database. The index of the rule is stored in the rule object. See `pmc_free_rule` to cleanup.

```
pmc_status_t
pmc_query_rule_first(
                                pmc_rule_t **rule_p,
                                char
**info_msg_p)
```

5.2.6.3.5.2.14 Query next stateful rule in the database

Query the next stateful rule in the database. This function will return the next rule based on the index of the current rule that is passed in. See `pmc_free_rule` to cleanup.

```
pmc_status_t pmc_query_rule_next(
                                uint32_t index,
                                pmc_rule_t **next_rule_p,
                                char **info_msg_p)
```

5.2.6.3.5.2.15 Free expression and rule objects after queries

Free the expression and rule objects returned by the various queries.

```
void pmc_free_expr(pmc_expr_t *expr_p);
void pmc_free_rule(pmc_rule_t *rule_p);
```

5.2.6.3.5.2.16 Query statistics

Query PM statistics. The information is returned in a `pmc_stats_t` record.

```

pmc_status_t pmc_query_stats(
                                                    pmc_stats_t *stats_p,
                                                    char **info_msg_p)

```

5.2.6.3.5.2.17 Reset statistics

Reset PM statistics.

```

pmc_status_t pmc_reset_stats()

```

5.2.6.3.5.2.18 Set variable length trigger

Set the size of the variable length trigger. This may only be used when the database is empty.

```

pmc_status_t pmc_set_var_trig_size(
                                                    uint32_t size,
                                                    char **info_msg_p)

```

5.2.6.3.5.2.19 Expression analysis

Run the expression analysis tool. Be advised this can take in the order of minutes to complete depending on the number of expressions in the database.

```

pmc_status_t pmc_analysis(
                                                    char **log_str_p,
                                                    char **info_msg_p);

```

5.2.6.3.6 Linker-loader

This chapter provides a description of the API of the Pattern Matcher linker-loader (PMLL). The linker-loader accepts expression and rule records encoded in the NXP hardware specific format, links these records into a hardware-optimal image-also called the shadow database (shadow DB)-and loads this image into the PM hardware.

PMLL is implemented as a user-space library called `libpml.a`. The generic steps involved in configuring the Pattern Matcher hardware are as follows:

- Initialize the PMLL module with a call to the `pml_module_init()` function
- Call the `pml_db_create()` function to create a PMLL shadow DB and register the PMLA functions with the new shadow DB
- Use the `pml_connection_handle_set()` function to associate a PMLA channel with the created PMLL shadow DB
- Optionally reconfigure the equivalence table and the variable trigger length attributes using the `pml_equivalence_table_set()` and `pml_variable_trigger_size_set()` functions
- Use the PMLL expression functions to add expressions to the shadow DB
- Use the PMLL rule functions to add rules to the shadow DB
- Call the `pml_commit()` function to get PMLL to link the expression and rule records and to load the configuration on the PM hardware

All the PMLL functions run to completion, meaning that a PMLL function returns the operation performed by the function has been completed.

PMLL Files

File	Description
libpml.a	Contains all the PMLL objects
pml.h	Defines the interface to the PMLL moduleT
generic_types.h	Contains the generic type definitions and is included by pml.h
pm_defs.h	Defines the interface that is common to the PMLL, PMREC, PMSRC, etc., and is included by pml.h
pmp.h	Contains the definition of the Pattern Matcher Protocol and is included by pml.h
libpthread.a	Defines the pthread objects; used by some of the PMLL objects
byteswap.h, endian.h	Provide support for little and big-endian byte swapping and are included by pmp.h
inttypes.h	Define the basic integer types and is included by generic_types.h
stdbool.h	Defines the bool type and is included by generic_types.h
netinit/in.h	Provides support for the hton and ntohs macros and is included by pmp.h

5.2.6.3.6.1 pml return codes

Most of the PMLL functions return status codes. The different return codes are defined in the `pml_status_t` enumerated type and are listed below.

```
pml_ok_e
```

This is the generic success code.

```
pml_error_e
```

This is the generic error code. It is often used as an initial value for the return status variable.

```
pml_too_few_error_strings_e
```

More error codes than strings are defined.

```
pml_too_many_error_strings_e
```

More error strings than codes are defined.

```
pml_unsupported_record_version_e
```

The record version passed in is not supported.

```
pml_null_pointer_parameter_e
```

Unexpected NULL pointer passed in as a parameter.

```
pml_out_of_memory_e
```

Memory allocation failed - no more memory.

```
pml1_invalid_name_e
```

The passed in expression or rule name is invalid, for example, has zero length.

```
pml1_exp_name_is_in_use_e
```

An expression with the specified expression name exists.

```
pml1_exp_is_not_in_name_db_e
```

An expression with the specified expression name does not exist.

```
pml1_no_trigger_type_e
```

Could not determine the trigger type for a pattern.

```
pml1_unsupported_trigger_type_e
```

The trigger type found in the expression record is not supported.

```
pml1_too_few_trig_row_format_defs_e
```

The number of the row trigger formats is too small.

```
pml1_too_many_trig_row_format_defs_e
```

The number of the row trigger formats is too big.

```
pml1_failed_to_commit_pattern_e
```

Failed to commit a pattern.

```
pml1_entry_write_failed_e
```

Failed to write a table entry to the PM H/W.

```
pml1_bad_confirmation_rec_size_e
```

The size of the confirmation record structure is not as expected.

```
pml1_module_not_initialized_e
```

The PMLL module has not been initialized.

```
pml1_bad_trigger_rec_size_e
```

The size of the trigger record structure is not as expected.

```
pml1_invalid_parameter_value_e
```

The value of a parameter passed to the function is invalid.

```
pml1_too_many_one_byte_patterns_e
```

Adding the pattern would exceed the "1-byte" pattern limit.

```
pml1_too_many_two_byte_patterns_e
```

Adding the pattern would exceed the "2-byte" pattern limit.

```
pml1_too_many_variable_patterns_e
```

Adding the pattern would exceed the "variable" pattern limit.

```
pml1_too_many_special_patterns_e
```

Adding the pattern would exceed the "special" pattern limit.

```
pml1_too_many_patterns_e
```

Adding the pattern would exceed the global pattern limit.

```
pml1_invalid_db_handle_e
```

The passed in PMLL DB handle is invalid.

```
pml1_table_index_too_big_e
```

The index table passed is too big.

```
pml1_pattern_not_in_confid_table_e
```

A pattern could not be found in the confidence table.

```
pml1_bad_logic_e
```

Bad logic in function.

```
pml1_too_many_reactions_in_rule_e
```

Found too many reactions in a rule.

```
pml1_no_reactions_in_rule_e
```

Found no reactions in a rule.

```
pml1_reaction_too_big_e
```

Reaction size is too big.

```
pml1_null_reaction_pointer_in_rule_e
```

Found a NULL reaction pointer in a rule.

```
pml1_rule_name_is_in_use_e
```

A rule with the specified rule name exists.

```
pml1_too_many_rules_e
```

Adding the rule would exceed the rule limit.

```
pml1_rule_is_not_in_name_db_e
```

A rule with the specified rule name does not exist.

```
pml1_exp_is_used_in_rule_e
```

An expression is used in a rule.

```
pml1_failed_to_init_mutex_e
```

Initialization of a mutex failed.

```
pml1_failed_to_destroy_mutex_e
```

Destruction of a mutex failed.

```
pml1_no_more_records_e
```

There are no more records in PMLL DB.

```
pml1_failed_to_create_name_db_e
```

Failed to create a name DB.

```
pml1_failed_to_add_exp_to_name_db_e
```

Failed to add an expression record to the PMLL expression name DB.

```
pml1_failed_to_add_rule_to_name_db_e
```

Failed to add a rule record to the PMLL rule name DB.

```
pml1_index_is_not_in_use_e
```

The index is not in use.

```
pml1_failed_to_create_reset_table_e
```

Failed to create the PMLL DB reset table.

```
pml1_failed_to_create_block_table_e
```

Failed to create the PMLL DB block table.

```
pml1_failed_to_create_cli_menu_e
```

Failed to create a CLI menu.

```
pml1_failed_to_register_cli_command_e
```

Failed to register a CLI command.

```
pml1_too_few_confirmation_blocks_e
```

The number of the confirmation blocks is too small.

```
pml1_out_of_extension_blocks_e
```

Run out of the extension blocks.

```
pml1_failed_to_create_rule_id_table_e
```

Failed to create the PMLL DB rule ID table.

```
pml1_out_of_rule_ids_e
```

Run out of rule IDs.

```
pml1_reaction_too_small_e
```

Size of a reaction is too small.

```
pml1_misaligned_reaction_e
```

Instructions in the reaction are misaligned.

```
pml1_failed_to_create_rule_ctx_table_e
```

Failed to create the PMLL DB rule context table.

```
pml1_session_ctx_area_too_small_e
```

Session context area size is smaller than the minimum size required by PMLL.

```
pml1_failed_to_alloc_rule_ctx_area_e
```

Failed to allocate a rule context area.

```
pml1_vtrigger_size_out_of_range_e
```

Variable trigger size is out of range.

```
pml1_exps_are_present_in_db_e
```

Expressions are present in the PMLL DB.

```
pml1_bad_reaction_event_type_e
```

Unsupported reaction event type.

```
pml1_bad_session_ctx_area_size_e
```

Session context area size is not a multiple of a session context entry.

```
pml1_table_reset_failed_e
```

Failed to reset entries in PM H/W tables.

```
pml1_pmhi_init_failed_e
```

Failed to initialize the PMLL PM H/W I/O module.

```
pml1_failed_to_set_atomic_attr_e
```

Failed to set the PMP atomic attribute.

```
pml1_failed_to_set_vtrig_size_attr_e
```

Failed to set the PMP variable trigger size attribute.

```
pml1_failed_to_set_end_of_sui_attr_e
```

Failed to set the PMP end-of-SUI attribute.

```
pml1_flush_failed_e
```

The PMLA flush operation failed.

```
pml1_failedto_reset_rule_ctxs_ee
```

Failed to reset a rule context.

```
pml1_failed_to_set_batch_attr_e
```

Failed to set the batch attribute.

```
pml1_pmla_channel_is_down_e
```

PMLA channel is down.

```
pml1_pmla_channel_is_not_set_e
```

PMLA channel is not set.

```
pml1_handle_table_create_failure_e
```

Failed to create the PMLL handle table.

```
pml1_handle_table_destroy_failure_e
```

Failed to destroy the PMLL handle table.

```
pml1_failed_to_allocate_handle_e
```

Failed to allocate a PMLL handle.

```
pml1_too_many_stateful_rules_req_e
```

The requested maximum number of stateful rules is too big.

```
pml1_cannot_register_null_pmla_func_e
```

A NULL pointer cannot be registered as a PMLA function pointer.

5.2.6.3.6.2 pml1_api_version_get

pml1_api_version_get-Retrieve the version of the PMLL API.

Synopsis

```
#include <pml1.h>
uint32_t pml1_api_version_get(void);
```

Description

The `pml1_api_version_get()` function retrieves the version of the API of the PMLL library being used.

Return Value

The `pml1_api_version_get()` function returns the version of the API of the PMLL library being used.

5.2.6.3.6.3 pml1_commit

`pml1_commit` - Commit the PMLL shadow DB to the PM hardware.

Synopsis

```
#include <pml1.h>
pml1_status_t pml1_commit(
    unsigned int pml1DbHandle,
    char          *expName_p);
```

Description

The `pml1_commit()` function triggers all the PMLL commit operations. The current version of the `pml1_commit()` function implements both the linking and the loading functions of PMLL. Once the `pml1_commit()` function returns, the optimized image of the PMLL shadow DB with the `pml1DbHandle` handle has been written to the PM hardware and the PM hardware is ready to scan data for the configured expressions and rules.

When the `pml1_commit()` function fails to link the patterns present in the PMLL shadow DB and returns with the `pml1_failed_to_commit_pattern_e` error code, the name of the expression containing the pattern that caused the PMLL linking failure is returned through the `expName_p` parameter.

Return Value

The `pml1_commit()` function returns `pml1_ok_e` upon success or an error code upon a failure. The `pml1_commit()` function can return the following error codes:

```
pml1_bad_logic_e
pml1_entry_write_failed_e
pml1_entry_write_failed_e
pml1_failed_to_commit_pattern_e
pml1_failed_to_reset_rule_ctxs_e
pml1_failed_to_set_atomic_attr_e
pml1_failed_to_set_batch_attr_e
pml1_failed_to_set_end_of_sui_attr_e
pml1_failed_to_set_vtrig_size_attr_e
pml1_flush_failed_e
pml1_invalid_db_handle_e
pml1_module_not_initialized_e
pml1_null_pointer_parameter_e
pml1_out_of_memory_e
pml1_pmla_channel_is_down_e
pml1_pmla_channel_is_not_set_e
pml1_table_reset_failed_e
```

When the `pml1_failed_to_commit_pattern_e` error code is returned the name of the expression containing the pattern that caused the PMLL linking failure is returned through the `expName_p` parameter. At present this failure is critical. There is no

recovery mechanisms built into the `pmll_commit()` function and most of the time this failure results in corrupting the PMLL shadow DB. The user must rebuild the PMLL shadow DB from scratch.

5.2.6.3.6.4 `pmll_connection_handle_set`, `pmll_connection_handle_get`

`pmll_connection_handle_set`, `pmll_connection_handle_get`-Set, get the PMLA connection handle.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_connection_handle_set(
    unsigned int pmllDbHandle,
    handle_t pmlaHandle);
```

`pmll_status_t pmll_connection_handle_get(`

```
    unsigned int pmllDbHandle,
    handle_t *pmlaHandle_p);
```

Description

`pmll_connection_handle_set()` associates a PMLA connection with the PMLL shadow DB with the `pmllDbHandle` handle. The user specified `pmlaHandle` is used by PMLL to refer to the set of PMLA functions registered previously with PMLL when the PM shadow DB was created (using `pmll_db_create`). The `pmlaHandle` is needed as an input parameter when calling PMLA functions.

`pmll_connection_handle_get()` retrieves the PMLA connection handle that is currently associated with the PMLL shadow DB with the `pmllDbHandle` handle. The retrieved PMLA connection handle is returned through the `pmlaHandle_p` parameter.

Return Value

The `pmll_connection_handle_set()` and `pmll_connection_handle_get()` functions return `pmll_ok_e` upon success or an error code upon a failure.

The `pmll_connection_handle_set()` function can return the following error codes:

```
pmll_invalid_db_handle_e
pmll_module_not_initialized_e
```

The `pmll_connection_handle_get()` function can return the following error codes:

```
pmll_invalid_db_handle_e
pmll_module_not_initialized_e
pmll_null_pointer_parameter_e
```

5.2.6.3.6.5 `pmll_db_create`, `pmll_db_destroy`

`pmll_db_create`, `pmll_db_destroy`-Create, destroy a PMLL shadow DB.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_db_create(
    pmll_db_params_t
    *pmllDbParams_p,
    unsigned int
```

```
*pmlldbHandle_p);  
pmlldb_status_t pmlldb_destroy(unsigned int pmlldbHandle);
```

Description

`pmlldb_create()` creates a new PMLL shadow DB. Upon successful completion the function returns a non-negative integer handle that was assigned to the created PMLL shadow DB. PMLL API functions that operate on a specific shadow DB take this handle as an input parameter. The handle is returned through the `pmlldbHandle_p` parameter.

The `pmlldb_params_t` type of the `pmlldbParams_p` parameter is defined as follows:

```
typedef int pmlldb_pmla_bulk_begin_function_t(  
    handle_t                                pmlaHandle);  
typedef int pmlldb_pmla_bulk_end_function_t(  
    handle_t                                pmlaHandle);  
typedef int pmlldb_pmla_flush_function_t(  
    handle_t                                pmlaHandle);  
typedef int pmlldb_pmla_read_function_t(  
    handle_t                                pmlaHandle,  
    pmlldb_msg_t                            *msg_p);  
typedef int pmlldb_pmla_write_function_t(  
    handle_t                                pmlaHandle,  
    pmlldb_msg_t                            *msg_p);  
typedef const char *pmlldb_pmla_error_string_get_t(  
    int                                      errorCode);  
typedef struct {  
    pmlldb_pmla_bulk_begin_function_t  
        *pmlaBulkBeginFunction_p;  
    pmlldb_pmla_bulk_end_function_t  
        *pmlaBulkEndFunction_p;  
    pmlldb_pmla_flush_function_t  
        *pmlaFlushFunction_p;  
    pmlldb_pmla_read_function_t  
        *pmlaReadFunction_p;  
    pmlldb_pmla_write_function_t  
        *pmlaWriteFunction_p;  
    pmlldb_pmla_error_string_get_t  
        *pmlaErrorStringGetFunction_p;  
} pmlldb_pmla_functions_t;  
typedef struct {  
    pmlldb_pmla_functions_t  
pmlaFunctions;  
    pmlldb_extension_block_num_attr_t  
dxeSreTableSize;  
    pmlldb_context_area_size_attr_t  
sreSessionCtxSize;  
    pmlldb_context_max_num_attr_t  
sreSessionCtxNum;  
    pmlldb_max_stateful_rule_num_attr_t  
sreRuleNum;  
} pmlldb_params_t;
```

The `pmlaFunctions` parameter contains pointers to the PMLA functions that must be registered with the PMLL shadow DB. All the functions must be specified. If any of the PMLA function pointers are NULL, then the `pmlldb_create()` function will fail with the `pmlldb_cannot_register_null_pmla_func_e` error code. Refer to the "PMLL PMLA Functions" section later in this chapter for more details on the PMLA functions.

The `dxeSreTableSize`, `sreSessionCtxSize`, `sreSessionCtxNum` and `sreRuleNum` parameters indicate PM hardware settings. These settings are specified when loading PM hardware drivers and can be queried. The `dxeSreTableSize` parameter

indicates the total number of confirmation entries, including both the pattern confirmation entries and the reaction extension entries available to the PM hardware. The minimum value of entries is 74240 and the maximum value is 1048576. The `sreSessionCtxSize` parameter indicates the size of the PM hardware context table entry associated with a single session. The value of `sreSessionCtxSize` must be between 64 and 131072. The `sreSessionCtxNum` parameter indicates the number of PM sessions supported by the PM hardware. The `sreRuleNum` parameter indicates the maximum number of stateful rules that can be configured in the PM hardware. The value of `sreRuleNum` must be between 0 and 32768. PMLL relies on the settings provided by the user to match the PM hardware settings. It does not query the hardware to sanitize the values.

Each PMLL shadow DB has its own mutex to protect accesses to the PMLL shadow DB. This means that the developers using the PMLL library do not have to create external mutual exclusion mechanisms to protect the integrity of the PMLL shadow DB.

`pmlldb_destroy()` destroys the PMLL shadow DB with the `pmlldbHandle` handle. The function removes all the rules and expressions from the DB and frees all the memory that was used by the PMLL shadow DB.

Return Value

Both the `pmlldb_create()` and `pmlldb_destroy` functions return `pmlldb_ok_e` upon success or an error code upon a failure.

The `pmlldb_create()` function can return the following error codes:

```
pmlldb_bad_session_ctx_area_size_e
pmlldb_cannot_register_null_pmla_func_e
pmlldb_failed_to_allocate_handle_e
pmlldb_failed_to_create_block_table_e
pmlldb_failed_to_create_name_db_e
pmlldb_failed_to_create_reset_table_e
pmlldb_failed_to_create_rule_ctx_table_e
pmlldb_failed_to_create_rule_id_table_e
pmlldb_failed_to_init_mutex_e
pmlldb_module_not_initialized_e
pmlldb_null_pointer_parameter_e
pmlldb_out_of_memory_e
pmlldb_session_ctx_area_too_small_e
pmlldb_too_few_confirmation_blocks_e
pmlldb_too_many_stateful_rules_req_e
```

The `pmlldb_destroy()` function can return the following error codes:

```
pmlldb_failed_to_destroy_mutex_e
pmlldb_invalid_db_handle_e
pmlldb_module_not_initialized_e
```

5.2.6.3.6.6 pmlldb_equivalence_table_set, pmlldb_equivalence_table_get

`pmlldb_equivalence_table_set`, `EquivalenceTableGet-Set`, get the equivalence table.

Synopsis

```
#include <pmlldb.h>
pmlldb_status_t pmlldb_equivalence_table_set(
    unsigned int
    pmlldbHandle,
    const uint8_t
    *equivalenceTable_p);
```

`pmlldb_status_t pmlldb_equivalence_table_get(`

```
    unsigned int
    pmlldbHandle,
```

```
uint8_t  
*equivalenceTable_p);
```

Description

`pmll_equivalence_table_set()` can be used to configure new values in the equivalence table associated with the PMLL shadow DB with the `pmllDbHandle` handle. The new values of the equivalence table are passed to the function through the `equivalenceTable_p` parameter. A default equivalence table is defined in `pm_defs.h`. Note that the `pmll_equivalence_table_set()` function will fail if it is called when there are expressions present in the PMLL shadow DB.

`pmll_equivalence_table_get()` retrieves the values in the equivalence table associated with the PMLL shadow DB with handle `pmllDbHandle`.

The size of the buffer pointed to by `equivalenceTable_p` must be at least `PMP_EQUIVALENCE_ENTRY_SIZE` bytes.

Return Value

The `pmll_equivalence_table_set()` and `pmll_equivalence_table_get()` functions return `pmll_ok_e` upon success or an error code upon a failure.

The `pmll_equivalence_table_set()` function can return the following error codes:

```
pmll_exps_are_present_in_db_e  
pmll_invalid_db_handle_e  
pmll_module_not_initialized_e  
pmll_null_pointer_parameter_e
```

The `pmll_equivalence_table_get()` function can return the following error codes:

```
pmll_invalid_db_handle_e  
pmll_module_not_initialized_e  
pmll_null_pointer_parameter_e
```

5.2.6.3.6.7 pmll_error_string_get

`pmll_error_string_get`-Get the error string associated with a PMLL error code.

Synopsis

```
#include <pmll.h>  
const char *pmll_error_string_get(pmll_status_t errorCode);
```

Description

The `pmll_error_string_get()` function returns a concise string describing the error condition associated with `errorCode`.

Return Value

The `pmll_error_string_get()` function returns a concise string describing the error associated with `errorCode`.

5.2.6.3.6.8 pmll expression functions

`pmll_exp_add`-Add an expression to a PMLL shadow DB.

`pmll_exp_delete`-Delete an expression from a PMLL shadow DB.

`pmll_exp_all_delete`-Delete all expressions from a PMLL shadow DB.

`pmll_exp_index_next_get`-Get the index of the next PMLL expression.

`pmll_exp_name_in_use`-Check if an expression name is in use.

Synopsis

```
#include <pml1.h>
pml1_status_t pml1_exp_add(
    unsigned int
    pml1DbHandle,
    uint32_t
    recordVersion,
    const
    pm_exp_record_t
    *expRecord_p,
    uint32_t
    *index_p);
```

pml1_status_t pml1_exp_delete(

```
    unsigned int
    pml1DbHandle,
    uint32_t
    idx);
```

pml1_status_t pml1_exp_all_delete(

```
    unsigned int
    pml1DbHandle);
```

pml1_status_t pml1_exp_index_next_get(

```
    unsigned int
    pml1DbHandle,
    uint32_t
    uint32_t
    *nextIndex_p);
    idx,
```

pml1_status_t pml1_exp_name_in_use(

```
    unsigned int
    pml1DbHandle,
    const char
    *name_p,
    bool
    *nameIsInUse_p,
    uint32_t
    *idx_p);
```

Description

`pml1_exp_add()` adds a single PMLL expression to a PMLL shadow DB with the `pml1DbHandle` handle. The function uses the `index_p` parameter to return an index that is assigned to the expression by PMLL. Other PMLL expression functions take this index as an input parameter that identifies the expression. The `recordVersion` parameter indicates the version of the expression record specified by `expRecord_p`. The supported version for Pattern Matcher 1.1 is `PMLL_EXP_RECORD_V_1_0_0`, which is defined in `pml1.h`.

The `expRecord_p` parameter is a pointer to the `pm_exp_record_t` structure, which is defined in `pm_defs.h`. The `pm_exp_record_v_1_0_0_t` type below defines the version 1.0.0 of the PMLL expression record. The user needs to create this record and cast it to the generic PMLL expression record type (`pm_exp_record_t`) and use this generic type in the PMLL functions.

```
typedef struct pm_pattern_record_t {
```

```
    uint8_t
        triggerEntry[PM_TRIGGER_ENTRY_SIZE];
    uint8_t
        keyElementEntry[PM_KEY_ELEMENT_ENTRY_SIZE];
    uint8_t
        confirmationEntry[PM_CONFIRMATION_ENTRY_SIZE];
    struct pm_pattern_record_t
        *nextPatternRecord_p;
} pm_pattern_record_t;
```

```
typedef struct {
```

```
    char
    name_s[PM_NAME_MAX_LENGTH];
    pm_pattern_record_t
    patterns;
} pm_exp_record_v_1_0_0_t;
```

```
typedef uint8_t *pm_exp_record_t;
```

`pmll_exp_delete()` deletes the expression with `idx` index from the PMLL shadow DB specified by `pmllDbHandle`.

`pmll_exp_all_delete()` deletes all the expressions from the PMLL shadow DB specified by `pmllDbHandle`.

`pmll_exp_index_next_get()` retrieves the next valid expression index after the `idx` index in the PMLL shadow DB specified by `pmllDbHandle`. The retrieved index is returned through the `nextIndex_p` parameter. To get the first valid index the `idx` parameter must be set to the `PMLL_NULL_INDEX` value. The `pmll_exp_index_next_get()` function returns `pmll_no_more_records_e` when there are no valid expression records with index values greater than the passed in `idx` index value. The `pmll_exp_index_next_get()` function allows the user to "walk" through all the expressions in the PMLL shadow DB in an efficient way.

`pmll_exp_name_in_use()` checks if the `name_p` expression name is already in use in the PMLL shadow DB specified by `pmllDbHandle`. This is used to verify that a name is not already in use before attempting to add it to the PMLL shadow DB. If the `name_p` expression name is in use, the `nameIsInUse_p` parameter is set to true and the `idx_p` parameter is set to the index of the expression that uses `name_p`. If the `name_p` is not in use, then `nameIsInUse_p` is set to false.

Return Value

All the PMLL expression functions returns `pmll_ok_e` upon success or an error code upon a failure.

The `pmll_exp_add()` function can return the following error codes:

```
pmll_exp_name_is_in_use_e
pmll_failed_to_add_exp_to_name_db_e
pmll_invalid_db_handle_e
pmll_invalid_name_e
pmll_module_not_initialized_e
pmll_no_trigger_type_e
pmll_null_pointer_parameter_e
pmll_out_of_extension_blocks_e
pmll_out_of_memory_e
pmll_too_many_one_byte_patterns_e
pmll_too_many_patterns_e
pmll_too_many_patterns_e
pmll_too_many_special_patterns_e
pmll_too_many_two_byte_patterns_e
pmll_too_many_variable_patterns_e
pmll_unsupported_record_version_e
pmll_unsupported_trigger_type_e
```

The `pml_exp_delete()` function can return the following error codes:

```
pml_exp_is_not_in_name_db_e
pml_exp_is_used_in_rule_e
pml_index_is_not_in_use_e
pml_invalid_db_handle_e
pml_module_not_initialized_e
```

The `pml_exp_all_delete()` function can return the following error codes:

```
pml_exp_is_not_in_name_db_e
pml_exp_is_used_in_rule_e
pml_invalid_db_handle_e
pml_module_not_initialized_e
```

The `pml_exp_index_next_get()` function can return the following error codes:

```
pml_invalid_db_handle_e
pml_module_not_initialized_e
pml_no_more_records_e
pml_null_pointer_parameter_e
```

The `pml_exp_name_in_use()` function can return the following error codes:

```
pml_invalid_db_handle_e
pml_module_not_initialized_e
pml_null_pointer_parameter_e
```

See Also

`pml_supported_exp_record_versions_get()`, `pml_supported_exp_record_versions_num_get()`

5.2.6.3.6.9 pml_module_init, pml_module_shutdown

`pml_module_init`-Initialize the PMLL module.

`pml_module_shutdown`-Shuts down the PMLL module.

Synopsis

```
#include <pml.h>
pml_status_t pml_module_init(
    unsigned int
    initialDbHandleNum,
    bool
    expandDbHandleNumFlag);
pml_status_t pml_module_shutdown(void);
```

Description

`pml_module_init()` initializes the PMLL module. No other PMLL API function will work properly until the `pml_module_init()` function has been called. The `initialDbHandleNum` parameter indicates the initial number of the PMLL shadow DB handles to be created. The `expandDbHandleNumFlag` flag indicates if the pool of PMLL shadow DB handles can be expanded or not when there are no more handles available in the pool. In order to allow the dynamic expansion of the PMLL shadow DB handle pool the `expandDbHandleNumFlag` flag must be set to true. If it is set to false, dynamic expansion of the PMLL shadow DB handle pool will not be allowed.

`pml_module_shutdown()` destroys the global resources used by the PMLL module and shuts down the PMLL module.

Return Value

The `pmll_module_init()` and `pmll_module_shutdown()` functions return `pmll_ok_e` upon success or an error code upon failure.

The `pmll_module_init()` function can return the following error codes:

```
pmll_bad_confirmation_rec_size_e
pmll_bad_trigger_rec_size_e
pmll_handle_table_create_failure_e
pmll_pmhi_init_failed_e
pmll_too_few_error_strings_e
pmll_too_few_trig_row_format_defs_e
pmll_too_many_error_strings_e
pmll_too_many_trig_row_format_defs_e
```

The `pmll_module_shutdown()` function can return the following error codes:

```
pmll_handle_table_destroy_failure_e
```

5.2.6.3.6.10 pmll pmla functions

`pmll_pmla_functions_set`-Set the PMLA functions.

`pmll_pmla_functions_get`-Retrieve the PMLA functions.

`pmll_pmla_bulk_begin`-Indicate the beginning of a bulk operation.

`pmll_pmla_bulk_end`-Indicate the end of a bulk operation.

`pmll_pmla_error_string_get`-Get error string describing an error code.

`pmll_pmla_flush`-Flush messages in the PMLA connection.

`pmll_pmla_read`-Read a PMP message.

`pmll_pmla_write`-Write a PMP message.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_pmla_functions_get(
    unsigned int
    pmllDbHandle,

    pmll_pmla_functions_t
    *functions_p);
```

`pmll_status_t pmll_pmla_functions_set(`

```
    unsigned int
    pmllDbHandle,

    pmll_pmla_functions_t
    *functions_p);
```

Description

PMLA is responsible for implementing a communication channel to and from the PMCI module and for conveying PMP control messages across that communication channel. Users can develop and use their own versions of PMLA functions. This allows the communication channel to the PMCI module to be defined by users according to their requirements. Sample PMLA functions are delivered with the PM software. They are implemented in `pmla_slm_api.c`.

The PMLL module uses the following PMLA functions:

pmll_pmla_bulk_begin(), pmll_pmla_bulk_end(), pmll_pmla_error_string_get(), pmll_pmla_flush(), pmll_pmla_read() and pmll_pmla_write() functions.

The PMLA functions called by PMLL are registered with PMLL through a call to the pmll_db_create() function. The PMLA functions associated with a PMLL shadow DB can subsequently be changed using the pmll_pmla_functions_set() and pmll_pmla_functions_get() functions.

In pmll_pmla_functions_set() and pmll_pmla_functions_get() the pointers to the PMLA functions are passed through the functions_p parameter and the functions are registered against the PMLL shadow DB with the pmllDbHandle handle. The registered PMLA functions are used by PMLL to implement the commit and a number of other operations. The return value of zero in all the registered PMLA functions, except for the pmll_pmla_error_string_get() function, indicates that the operation was successful; a non-zero return value indicates an error. PMLL does not interpret the PMLA error codes. The signatures of the PMLA functions that must be registered and the pmll_pmla_functions_t structure are given below.

```
typedef int pmll_pmla_bulk_begin_function_t (handle_t pmlaHandle);
typedef int pmll_pmla_bulk_end_function_t (handle_t pmlaHandle);
typedef const char *pmll_pmla_error_string_getFunction_t (int errorCode);
typedef int pmll_pmla_flush_function_t (handle_t pmlaHandle);
typedef int pmll_pmla_read_function_t (handle_t pmlaHandle, pmp_msg_t *msg_p);
typedef int pmll_pmla_write_function_t (handle_t pmlaHandle, pmp_msg_t *msg_p);
```

```
typedef struct {
    pmll_pmla_bulk_begin_function_t

    *pmlaBulkBeginFunction_p;
    pmll_pmla_bulk_end_function_t

    *pmlaBulkEndFunction_p;
    pmll_pmla_flush_function_t

    *pmlaFlushFunction_p;
    pmll_pmla_read_function_t

    *pmlaReadFunction_p;
    pmll_pmla_write_function_t

    *pmlaWriteFunction_p;
    pmll_pmla_error_string_get_t

    *pmlaErrorStringGetFunction_p;
} pmll_pmla_functions_t;
```

When registering the PMLA functions with pmll_pmla_functions_set() function all the functions in the functions_p structure must be defined. A NULL function pointer in the functions_p structure will cause pmll_pmla_functions_set() to fail with the pmll_cannot_register_null_pmla_func_e error code. None of the functions in the functions_p structure are registered in this case.

pmll_pmla_functions_get() retrieves the pointers to the PMLA functions currently registered with the PMLL shadow DB with the pmllDbHandle handle. The retrieved function pointers are returned through the functions_p parameter.

The pmll_pmla_bulk_begin() and pmll_pmla_bulk_end() functions are invoked by the pmll_commit() function at the beginning and the end of loading of the PMLL shadow DB image to the PM hardware. PMLL has no requirements and imposes no restrictions on how these functions are implemented. A failure of either of these functions is logged but otherwise ignored by PMLL. The functions are intended to indicate the start and end of multiple related messages sent by the PMLL to the PMCI.

The pmll_pmla_error_string_get() is expected to return a string describing the specified error code. This function is used by PMLL while generating logs.

The `pmll_pmla_flush()` is invoked from the `pmll_commit()` function and it is expected to call the `pmci_flush()` function. The `pmci_flush()` function blocks until all the outstanding requests on the PMCI channel are processed.

The `pmll_pmla_read()` function is expected to call the `pmci_read()` function, which implements reading of a PMP control message from the PM hardware.

The `pmll_pmla_write()` function is expected to call the `pmci_write()` function, which implements writing of a PMP control message to the PM hardware.

Return Value

`pmll_pmla_functions_get()` and `pmll_pmla_functions_set()` functions return `pmll_ok_e` upon success or an error code upon failure.

The `pmll_pmla_functions_get()` function can return the following error codes:

```
pmll_invalid_db_handle_e
pmll_module_not_initialized_e
pmll_null_pointer_parameter_e
```

The `pmll_pmla_functions_set()` function can return the following error codes:

```
pmll_cannot_register_null_pmla_func_e
pmll_invalid_db_handle_e
pmll_module_not_initialized_e
pmll_null_pointer_parameter_e
```

5.2.6.3.6.11 pmll rule functions

`pmll_rule_add`-Add a rule to a PMLL shadow DB.

`pmll_rule_delete`-Delete a rule from a PMLL shadow DB.

`pmll_rule_all_delete`-Delete all rules from a PMLL shadow DB.

`pmll_rule_index_next_get`-Get the index of the next PMLL rule.

`pmll_rule_name_in_use`-Check if a rule name is in use.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_rule_add(
    unsigned int
    pmllDbHandle,
    uint32_t
    recordVersion,
    const
    pm_rule_record_t
    *ruleRecord_p,
    uint32_t
    *index_p);
pmll_status_t pmll_rule_delete(
    unsigned int
    pmllDbHandle,
    uint32_t
    idx);
pmll_status_t pmll_rule_all_delete(
    unsigned int
    pmllDbHandle);
pmll_status_t pmll_rule_index_next_get(
    unsigned int
```

```

pmlldbHandle,
    uint32_t
    uint32_t
    idx,
*nextIndex_p);
pmlldbHandle,
    const char
*name_p,
    bool
*nameIsInUse_p,
    uint32_t
*idx_p);

```

Description

`pmlldb_rule_add()` adds a single PMLL rule to a PMLL shadow DB with the `pmlldbHandle` handle. The function uses the `index_p` parameter to return an index that is assigned to the rule by PMLL. Other PMLL rule functions take this index as an input parameter that identifies the rule. The `recordVersion` parameter indicates the version of the rule record to be added and passed in using the `ruleRecord_p` parameter. The supported version for Pattern Matcher 1.1 is `PMLL_RULE_RECORD_V_1_0_0`, which is defined in `pmlldb.h`.

The `ruleRecord_p` parameter is a pointer to the `pm_rule_record_t` structure, which is defined in `pm_defs.h`. The `pm_rule_record_v_1_0_0_t` type below defines the version 1.0.0 of the PMLL rule record. The user needs to create this record and cast it to the generic PMLL rule record type (`pm_rule_record_t`) and use this generic type in the PMLL functions.

typedef enum {

```

    pm_null_reaction_event_type_e
= 0,
    pm_pattern_reaction_event_type_e
= 1,
    pm_end_of_sui_reaction_event_type_e
= 2,
    pm_last_reaction_event_type_e
= pm_end_of_sui_reaction_event_type_e
} pm_reaction_event_type_t;

```

typedef struct pm_reaction_entry_t {

```

    uint32_t
reactionEventType;
    char
expName_s[PM_NAME_MAX_LENGTH];
    struct pm_reaction_entry_t

*nextReactionEntry_p;
    uint32_t
reactionSize;
    uint8_t
reactionData[PM_MAX_REACTION_SIZE];
} pm_reaction_entry_t;

```

typedef struct ruleRecord_t {

```

    char
name_s[PM_NAME_MAX_LENGTH];
    uint32_t
reactionNum;
    pm_reaction_entry_t

```

```
*reactionEntry_p;  
} pm_rule_record_v_1_0_0_t;
```

typedef uint8_t *pm_rule_record_t;

pmll_rule_delete() deletes the rule with the idx index from the PMLL shadow DB with the pmllDbHandle handle.

pmll_rule_all_delete() deletes all the rules from the PMLL shadow DB with the pmllDbHandle handle.

pmll_rule_index_next_get() retrieves the next valid rule index after the idx index in the PMLL shadow DB with the pmllDbHandle handle. The retrieved index is returned through the nextIndex_p parameter. To get the first valid index the idx parameter must be set to the PMLL_NULL_INDEX value. The pmll_rule_index_next_get() function returns pmll_no_more_records_e when there are no valid rule records with index values greater than the passed in idx index value. The pmll_rule_index_next_get() function allows the user to "walk" through all the rules in the PMLL shadow DB in an efficient way.

pmll_rule_name_in_use() checks if the name_p rule name is already in use in the PMLL shadow DB with the pmllDbHandle handle. If the name_p rule name is in use, the function sets the nameInUse_p parameter to true and the idx_p parameter to the index of the rule that uses the name_p rule name. If the name_p rule name is not in use, the function sets the nameInUse_p parameter to false.

Return Value

All the PMLL rule functions returns pmll_ok_e upon success or an error code upon failure.

The pmll_rule_add() function can return the following error codes:

```
pmll_bad_reaction_event_type_e  
pmll_exp_is_not_in_name_db_e  
pmll_failed_to_add_rule_to_name_db_e  
pmll_failed_to_alloc_rule_ctx_area_e  
pmll_failed_to_alloc_rule_ctx_area_e  
pmll_invalid_db_handle_e  
pmll_invalid_name_e  
pmll_misaligned_reaction_e  
pmll_module_not_initialized_e  
pmll_no_reactions_in_rule_e  
pmll_no_reactions_in_rule_e  
pmll_null_pointer_parameter_e  
pmll_Aout_of_memory_e  
pmll_out_of_rule_ids_e  
pmll_reaction_too_big_e  
pmll_reaction_too_small_e  
pmll_rule_name_is_in_use_e  
pmll_too_many_reactions_in_rule_e  
pmll_too_many_rules_e  
pmll_unsupported_record_version_e
```

The pmll_rule_delete() function can return the following error codes:

```
pmll_index_is_not_in_use_e  
pmll_invalid_db_handle_e  
pmll_module_not_initialized_e
```

The pmll_rule_all_delete() function can return the following error codes:

```
pmll_invalid_db_handle_e  
pmll_module_not_initialized_e  
pmll_rule_is_not_in_name_db_e
```

The `pml1_rule_index_next_get()` function can return the following error codes:

```
pml1_invalid_db_handle_e
pml1_module_not_initialized_e
pml1_no_more_records_e
pml1_null_pointer_parameter_e
```

The `pml1_rule_name_in_use()` function can return the following error codes:

```
pml1_invalid_db_handle_e
pml1_module_not_initialized_e
pml1_null_pointer_parameter_e
```

See Also

`pml1_supported_rule_record_versions_get()`, `pml1_supported_rule_record_versions_num_get()`

5.2.6.3.6.12 pml1_stats_get

`pml1_stats_get`-Retrieve the PMLL statistics.

Synopsis

```
#include <pml1.h>
pml1_status_t pml1_stats_get(
    unsigned int
    pml1DbHandle,
    pml1_stats_t
    *llStats_p);
```

Description

The `pml1_stats_get()` function retrieves the PMLL statistics for the PMLL shadow DB with the `pml1DbHandle` handle. The retrieved statistics are returned to the caller through the `llStats_p` parameter. The `pml1_stats_t` structure is defined as follows:

typedef struct {

```
    struct {
        uint32_t variableChainNum;           // uncompressed trigger stats.
        uint32_t twoByteChainNum;           // # of variable trigger chains
        uint32_t oneByteChainNum;           // # of 2-byte trigger chains
        uint32_t specialChainNum;           // # of 1-byte trigger chains
        // # of special trigger chains
    };
    struct {
        // post-cluster-compres. trigger stats.
        // # of variable resident trigger chains
        uint32_t variableResidentChainNum;
        // # of variable guest trigger chains
        uint32_t variableGuestChainNum;
        // # of 2-byte resident trigger chains
        uint32_t twoByteResidentChainNum;
        // # of 2-byte guest trigger chains
        uint32_t twoByteGuestChainNum;
        // # of 1-byte resident trigger chains
        uint32_t oneByteResidentChainNum;
        // # of 1-byte guest trigger chains
        uint32_t oneByteGuestChainNum;
        // # of special resident trigger chains
        uint32_t specialResidentChainNum;
        // # of special guest trigger chains
        uint32_t specialGuestChainNum;
```

```
};  
struct { // post-foldback-compression trigger  
stats.  
    // number of primary entries  
    uint32_t primaryEntryNum;  
    // number of secondary entries  
    uint32_t secondaryEntryNum;  
};  
} pml1_stats_t;
```

Return Value

The `pml1_stats_get()` function returns `pml1_ok_e` upon success or an error code upon a failure. The `pml1_stats_get()` function can return the following error codes:

```
pml1_invalid_db_handle_e  
pml1_module_not_initialized_e  
pml1_null_pointer_parameter_e
```

5.2.6.3.6.13 pml1 version functions

`pml1_supported_exp_record_versions_get`-Get the supported expression record versions.

`pml1_supported_exp_record_versions_num_get`-Get the number of the supported expression versions.

`pml1_supported_rule_record_versions_get`-Get the supported rule record versions.

`pml1_supported_rule_record_versions_num_get`-Get the number of the supported rule versions.

`pml1_version_str_get`-Get a version description string.

Synopsis

```
#include <pml1.h>  
pml1_status_t pml1_supported_exp_record_versions_get(  
    uint32_t *versionTable_p,  
    uint32_t tableSize);  
uint32_t pml1_supported_exp_record_versions_num_get(void);  
pml1_status_t pml1_supported_rule_record_versions_get(  
    uint32_t *versionTable_p,  
    uint32_t tableSize);  
uint32_t pml1_supported_rule_record_versions_num_get(void);  
char * pml1_version_str_get(  
    uint32_t version,  
    char *versionStr_p);
```

Description

`pml1_supported_exp_record_versions_get()` allows the user to retrieve the numerical values of the supported expression record versions. The number of the supported versions can be retrieved with `pml1_supported_exp_record_versions_num_get()`. The supported versions are returned to the caller through the `versionTable_p` table which should have `tableSize` entries in it.

`pml1_supported_exp_record_versions_num_get()` retrieves the number of the supported expression record versions.

`pml1_supported_rule_record_versions_get()` allows the user to retrieve the numerical values of the supported rule record versions. The number of the supported versions can be retrieved with `pml1_supported_rule_record_versions_num_get()`. The supported versions are returned to the caller through the `versionTable_p` table which should have `tableSize` entries in it.

`pml1_version_str_get()` returns a string describing the numerical version passed in the `version` parameter. The returned version string is stored in the `versionStr_p` buffer that should be at least `PMLL_VERSION_STR_LENGTH` bytes long.

Return Value

The `pmll_supported_exp_record_versions_get()` and `pmll_supported_rule_record_versions_get()` functions return `pmll_ok_e` upon success or an error code upon failure.

The `pmll_supported_exp_record_versions_get()` function can return the following error codes:

```
pmll_null_pointer_parameter_e
```

The `pmll_supported_rule_record_versions_get()` function can return the following error codes:

```
pmll_null_pointer_parameter_e
```

The `pmll_supported_exp_record_versions_num_get()` and `pmll_supported_rule_record_versions_num_get()` return the numbers of the supported expression or rule record versions, respectively.

The `pmll_version_str_get()` function returns the pointer passed in the `versionStr_p` parameter.

5.2.6.3.6.14 `pmll_variable_trigger_size_get`, `pmll_variable_trigger_size_set`

`pmll_variable_trigger_size_get`-Get the size of the variable trigger.

`pmll_variable_trigger_size_set`-Set the size of the variable trigger.

Synopsis

```
#include <pmll.h>
pmll_status_t pmll_variable_trigger_size_get(
    unsigned int pmllDbHandle,
    uint32_t *variableTriggerSize_p);
```

`pmll_status_t pmll_variable_trigger_size_set(`

```
    unsigned int pmllDbHandle,
    uint32_t variableTriggerSize);
```

Description

`pmll_variable_trigger_size_get()` retrieves the currently configured value of the variable trigger size in the PMLL shadow DB with the `pmllDbHandle` handle. The retrieved value is returned to the caller through the `variableTriggerSize_p` parameter.

`pmll_variable_trigger_size_set()` sets the value of the variable trigger size in the PMLL shadow DB with the `pmllDbHandle` handle to `variableTriggerSize`. The function should be called prior to adding any expressions to the PMLL shadow DB. The function will return a failure status code if it is invoked when there are expressions present in the PMLL shadow DB.

Return Value

The `pmll_variable_trigger_size_get()` and `pmll_variable_trigger_size_set` functions return `pmll_ok_e` upon success or an error code upon a failure.

The `pmll_variable_trigger_size_get()` function can return the following error codes:

```
pmll_invalid_db_handle_e
pmll_module_not_initialized_e
pmll_null_pointer_parameter_e
```

The `pmll_variable_trigger_size_set()` function can return the following error codes:

```
pmll_exps_are_present_in_db_e
pmll_invalid_db_handle_e
pmll_module_not_initialized_e
pmll_vtrigger_size_out_of_range_e
```

5.2.6.3.7 Pattern Matcher driver

This chapter describes the interfaces provided by the Pattern Matcher driver. The Pattern Matcher driver can be built into the kernel statically or as a set of dynamically-loaded kernel modules. In both cases, a small piece of platform support is built into the kernel statically to handle platform-initialisation, early-boot contiguous memory allocation, etc. The driver provides both user space and kernel space APIs that are used to configure the hardware as well as perform pattern matching operations.

The user space API is described in the following sections:

[PME Configuration](#) on page 420 provides the details on the various PME configuration requirements and sysfs interface. The configuration attribute may reside in the device-tree, as Kconfig options and as sysfs entries. There are also statistics which are collected on a continuous basis and made available either via a kernel api and sysfs.

[PME Scan](#) on page 431 provides an interface to perform pattern matching operations. This interface operates on the /dev/pme_scan device using various ioctls.

[PME Database](#) on page 427 provides an interface for configuring and modifying the PME database in hardware. This operates via the /dev/pme_db device and would not typically be used directly but instead user-space interfaces like PMCI and PMM would be used that incorporate the proprietary tools necessary to compile, link, and load patterns and rules.

The various user space ioctl() calls require an appropriate device descriptor parameter. The devices are created via a call to open(). The table, Devices for IOCTL System Calls, summarizes the devices required for each type of ioctl system call.

Devices for IOCTL System Calls

Device	IOCTL Type
/dev/pme_scan	PME Scan
/dev/pme_db	PME Database

[PME Kernel API](#) on page 440 allows kernel programmers to create pattern matcher contexts that can be used for scanning operations.

The PME Driver is a patch-set to the kernel providing a driver (static or module) and a header file that defines the driver interfaces. If built as a module, the driver names are pme.ko, pme_scan.ko and pme_db.ko. The header file is fsl_pme.h, located in the linux include directory-included via "#include <linux/fsl_pme.h>".

5.2.6.3.7.1 PME Configuration

Various items must be configured in order to utilize the PME device. These include device-tree entries, Kconfig parameters, module parameters (when using loadable modules) and also sysfs entries.

5.2.6.3.7.1.1 Device Tree Node Entry

Synopsis

```
pme: pme@316000 {
```

```
    compatible = "fsl,pme"
    reg = <0x316000 0x10000>;
    /* fsl,pme-pdsr = <0x0 0x23000000 0x0 0x01000000>; */
    /* fsl,pme-sre = <0x0 0x24000000 0x0 0x00a00000>; */
    interrupts = <16 2 1 5>;
};
```

Description

The Pme device tree node represents the Pme device and its CCSR configuration space. When a linux kernel has Pme support built in, it will react to this device tree node by configuring and managing the Pme device. It uses the Pme configuration interface to implement this. The device-tree node sits within the CCSR node ("soc").

"*fsl,pme-pdsr*" property specifies the start location and size of the Pattern Description and Stateful Rule Table in system memory. The table base address must be aligned to a natural 128-byte address boundary. The maximum size of the table is 128 Mbytes for PME v.2.0, 64 Mbytes for PME v2.1 and 32 Mbytes for PME v2.2. The current driver implementation allows this memory resource to be specified via the "*fsl,pme-pdsr*" device-tree property, or by resorting to a default allocation of contiguous memory early during kernel boot. This property specifies a 2-tuple of address and size (address is optional), specifying the physical address range. These elements are expressed as 64-bit values, so take two cells each;

```
fsl,pme-pdsr = <0x0 0x20000000 0x0 0x01000000>;
```

Optionally, only the size can be specified and the kernel will try to allocate the contiguous memory during boot time.

```
fsl,pme-pdsr = <0x0 0x100000>
```

If the hypervisor is in use, this address range is "guest physical". If the given memory range falls within the range used by the Linux OS, it will attempt to reserve the range against use by the OS.

"*fsl,pme-sre*" property specifies the start location and size of the SRE Context Table in system memory. The table base address must be aligned to a natural 32-byte address boundary. The maximum size of the table is 4 Gbytes. The current driver implementation allows this memory resource to be specified via this device-tree property, or by resorting to a default allocation of contiguous memory early during kernel boot. This node property specifies a 2-tuple of address and size (address is optional), specifying the physical address range. These elements are expressed as 64-bit values, so take two cells each;

```
fsl,pme-sre = <0x0 0x20000000 0x0 0x01000000>;
```

Optionally, only the size can be specified and the kernel will try to allocate the contiguous memory during boot time.

```
fsl,pme-sre = <0x0 0x300000>;
```

If the hypervisor is in use, this address range is "guest physical". If the given memory range falls within the range used by the Linux OS, it will attempt to reserve the range against use by the OS.

5.2.6.3.7.1.2 KCONFIG Options

Description

The Pme device supports various kernel configuration options.

FSL_PME2_PDSRSIZE

Pattern Description and Stateful Rule default table size defined as the number of 128 byte entries. This only takes effect if the device tree node doesn't have the 'fsl,pme-pdsr' property present. Valid range is 74240-1048573 (9.5MB-128MB). The default is 131072 (16MB)

FSL_PME2_SRESIZE

SRE Session Context Entries table default table size defined as the number of 32 byte entries. This only takes effect if the device tree node doesn't have the 'fsl,pme-sre' property present. Valid range is 0-134217727 (0-4GB) and the default is 327680 (10MB)

FSL_PME2_SRE_AIM

Select the inconclusive match mode treatment. When true the alternate inconclusive mode is used. When false the default inconclusive mode is used. The default is off.

FSL_PME2_SRE_ESR

Configures if an End of SUI event will produce a Simple End of SUI report.

FSL_PME2_SRE_CTX_SIZE_PER_SESSION

Configures the default SRE Context Size per Session as a power of 2 (e.g. a value of 5 translated to 32 bytes, whereas a value of 15 translated to 32 kbytes. Valid range is 5 to 17 with a default of 17.

FSL_PME2_SRE_CNR

Configured Number of Stateful Rules as a multiple of 256. Valid range is 0 to 128 with a default of 128 (i.e. 32kbytes).

FSL_PME2_SRE_MAX_INSTRUCTION_LIMIT

Configured maximum number of SRE instructions to be executed per reaction. Valid range is 0 to 65535 with a default of 65535.

FSL_PME2_SRE_MAX_BLOCK_NUMBER

Configured the maximum number of reaction head blocks to be traversed perpattern match event (e.g. a matched pattern or an End of SUI event). Valid range is 0 to 32767 with a default of 32767.

FSL_PME2_DB_QOSOUT_PRIORITY

Configures the /dev/pme_db qos priority level for its output frame queue. Valid range is 0 to 7 with a default of 2.

FSL_PME2_STAT_ACCUMULATOR_UPDATE_INTERVAL

The pme accumulator kernel thread reads the current device statistics and add it to a running counter. The frequency (expressed in milliseconds) of these updates may be controlled. If 0 is specified, no automatic updates are done. Valid range is 0 to 10000 with a default of 3400 (e.g. 3400 milliseconds).

5.2.6.3.7.1.3 SYSFS Attributes

Description

The sysfs interface is used to control global pme configuration Pme device parameters and provides an interface for pme statistics. The path to the device attributes is /dev/fsl-pme-dev. The following attributes are defined:

aim

This read-write field specifies the inconclusive match mode treatment. When this field is set to 1, the "alternate" inconclusive mode is used. When this field is set to 0 the "default" inconclusive mode is used. The default is 0.

bsc/[0-63]

This read-write field specifies the buffer pool size for the specified buffer pool id (0-63). When PME h/w acquires a buffer pool it determines the size of these buffers based on this configuration. A value of 0 causes the PME not to attempt any acquires. Valid range is 0 to 11 encoded as follows: 0 - 0 1 - 64 bytes 2 - 128 bytes 3 - 256 bytes 4 - 512 bytes 5 - 1024 bytes 6 - 2048 bytes 7 - 4096 bytes 8 - 8192 bytes 9 - 16,384 bytes 10 - 32,768 bytes 11 - 65,536 bytes

cdcr

This read-write entry disables the specified Pme caching. This value should generally always be zero (i.e. caching always enabled).

dmcr

This read-write entry is the DXE memory control settings.

ecc1bes

This read-write entry contains status bits indicating which internal SRAM instances have accumulated more single bit errors than their programmed threshold.

ecc2bes

This read-write entry contains status bits indicating which internal SRAM instances have had at least one double bit error detected.

eccaddr

This read-only entry, when a bit becomes set in the ecc1bes or ecc2bes registers, the Pme internal memory, and the address within that memory, on which the error was detected are recorded in this entry.

ecccode

This read-only entry, when a bit becomes set in the ecc1bes or ecc2bes entry, the corresponding ECC code and syndrome are recorded in this entry.

ecr0

This read-only entry captures debug information that is specific to the most recent Status code of the highest severity error that the Pme has seen but not necessarily communicated to the CPU via a report.

ecr1

This read-only entry captures debug information that is specific to the most recent Status code of the highest severity error that the Pme has seen but not necessarily communicated to the CPU via a report.

efqc_int

This read-write entry specifies the delay (an interval) between consecutive PMFA dequeue requests. Valid value are: 0, 64, 128, 256 (default) and 512.

end_of_simple_sui_report

This read-write entry specifies the delay (an interval) between consecutive PMFA dequeue requests. Valid value are: 0, 64, 128, 256 (default) and 512.

end_of_sui_reaction_ptr

This read-write entry specifies the head pointer of the reaction linked list that is to be traversed by the SRE for every End of SUI event. The pointer is an index to a 128-byte block relative to the start of the PDSR table. Valid rang is 0x0 to 0xFFFFF for PME ver 2.0, 0x0 to 0x7FFFF for PME ver 2.1 and 0x0 to 0x3FFFF for PME ver 2.2. The default is 0x0.

esr

This read-write attribute contains the Status code of the highest severity error that the Pme has seen but not necessarily communicated to the CPU via a report.t

faconf

This read-only attribute is the frame agent configuration.

famcr

This read-write attribute configures he frame agent memory control.

isr

This read-only attribute displays the current interrupt status register value. A non zero value indicates that a serious error was detected and consequently the PME device stops processing input until informed otherwise.

kvlt

This read-write entry specifies the size of the KES Variable Length Trigger from 2 to 16 bytes. Valid range is 0 to 16 with 2 being the default.

liodnr

This read-only entry specifies the LIODN value to use when PM accesses its private data structures.

max_allowed_test_line_per_pattern

This read-write entry specifies maximum allowed test line executions per pattern (scaled by a factor of 8; the actual maximum allowed test line executions is equal to MTE*8). The maximum allowed value for this field is 0x3FFF which translates into a maximum allowed test line execution value of 131,064. Valid range is 0x0 to 0x3FFF with 0x3FFF being the default.

max_chain_length

This read-write entry specifies the maximum chain length. This is the maximum number of allowed entries in confidence collision chains. Valid range is 0x0 to 0x7FFF with 0x7FFF being the default.

max_pattern_evaluations_per_sui

This read-write entry specifies maximum pattern evaluations per SUI (scaled by a factor of 8; the actual maximum allowed pattern evaluations is equal to value*8). A value of 0xFFFF will be treated as an infinite limit. Valid range is 0x0 to 0xFFFF with 0xFFFF being the default.

max_pattern_matches_per_sui

This read-write entry specifies maximum pattern matches per SUI (scaled by a factor of 8; the actual maximum allowed pattern matches is equal to value*8). A value of 0xFFFF will be treated as an infinite limit. Valid range is 0x0 to 0xFFFF with 0xFFFF being the default.

max_pdsr_index

This read-write entry specifies the maximum allocated index into the Pattern Description and Stateful Rule table available to the DXE. Valid range is 0x0 to 0xFFFFC for PME ver 2.0, 0x0 to 0x7FFFC for PME ver 2.1 and 0x0 to 0x3FFFC for PME ver 2.2. The index is 128-byte based.

miace

This read-write attribute is used specify the client read and write ports that are included in the miablc and miabyc counts. It is also used to specify the size of the aligned blocks counted by miablc

miacr

This read-write attribute is used to configure various client read and write port attributes.

pattern_range_counter_idx

This read-write entry specifies the index value targeted for counting pattern matches. Note that this field is only wide enough to allow specification of indices within the PDSR table that the DXE will access. Table entries beyond 0x1FFFF are only accessible by the SRE. Valid range is 0x0 to 0x1FFFF for PME ver 2.0, 0x0 to 0xFFFF for PME ver 2.1 and 0x0 to 0x7FFF for PME ver 2.2. The default is 0x0.

pattern_range_counter_mask

This read-write entry specifies the index mask for pattern_range_counter_index, allowing ranges of addresses to be targeted. A 0 in any bit means a "don't care" in the corresponding pattern_range_counter_index bit. Valid range is 0x0 to 0x1FFFF for PME ver 2.0, 0x0 to 0xFFFF for PME ver 2.1 and 0x0 to 0x7FFF for PME ver 2.2, with 0x0 being the default.

pdsrbah

This read-only attribute is used to provision the start location (upper bit 32 bits) of the PRSR table space in system memory.

pdsrbal

This read-only attribute is used to provision the start location (lower 32 bits) of the PRSR table space in system memory.

pehd

This read-write attribute disables selected Pme (KES, DXE or SRE) error condition detection.

pmtr

This read-write attribute allows programming of a time based latency threshold such that memory read transactions whose latency exceeds the threshold can be measured using the integrated performance monitor.

report_length_limit

This read-write entry specifies the maximum number of bman buffers that can be allocated for a single output report frame. A value of zero disables length limiting. Valid range is 0x0 to 0xFFFF with 0x0 being the default.

rev1

This read-only attribute provides the unique IP block ID for the Pattern Matcher block, as well as major and minor revision numbers. It is displayed as a hexadecimal value.

rev2

This read-only attribute provides the Pattern Matcher block integration and configuration options.

sbarh

This read-only attribute is used to provision the start location (upper bit 32 bits) of the SRE table space in system memory.

scbarl

This read-only attribute is used to provision the start location (lower 32 bits) of the SRE table space in system memory.

smcr

This read-write attribute is used to provide software with control over system memory transaction attributes for the SRE.

src_id

This read-only attribute specifies the source id that the PME uses in system bus transactions.

sre_context_size

This read-only entry specifies the context size per session. This represents the FSL_PME2_SRE_CTX_SIZE_PER_SESSION Kconfig value. Valid range is 0 to 13 encoded as follows: 0 - 0 1 - 32 bytes 2 - 64 bytes ... 13 - 128 bytes

sre_max_block_num

This read-write entry specifies the maximum number of reaction head blocks to be traversed per pattern match event. Valid range is 0x0 to 0x3FFF with 0x3FFF being the default.

sre_max_index_size

This read-only entry specifies the maximum allocated 128-bytes based index in the Pattern Description and Stateful Rule table that is to be accessed by the SRE. Valid range is 0x0 to 0xFFFFC.

sre_max_instruction_limit

This read-write entry specifies the maximum number of SRE instructions to be executed per reaction. Valid range is 0x0 to 0xFFFF with 0xFFFF being the default.

sre_max_offset_ctrl

This read-only entry specifies maximum 32-byte index offset for the SRE context table.

sre_pscl

This read-only entry specifies the prescaler value for the SRE Instruction Operand register \$C free running counter

sre_rule_num

This read-only entry specifies the configured number of Stateful Rules as a multiple of 256. The default is 128 (i.e. 32kbytes). This represents the FSL_PME2_SRE_CNR Kconfig value.

sre_session_ctx_num

This read-only entry specifies the maximum number of session contexts. This is derived by dividing the size of the SRE table size by per session configured size.

sw_db

This read-write entry can be used by software to hold a version number for the pattern match database. This attribute is currently not used.

The PME h/w maintains various statistics. The PME drivers poll these statistics at a configurable interval. The h/w statistics are read-reset based whereas the driver simply keeps a running 64 bit counter. Writing a zero to the sysfs entry will clear both the driver and h/w counter. The sysfs interface to these statistics are located in the *stats* subdirectory. The following statistics are defined:

cmec1ec

This read-write attribute contains a count of the number of single bit ECC errors that have occurred.

dxcmec1ec

This read-write attribute contains a count of the number of single bit ECC errors that have occurred.

dxemec1ec

This read-write attribute contains a count of the number of single bit ECC errors that have occurred.

`mia_blc`

This read-write attribute represents the number of bus aligned memory block accesses for all memory transactions that reach the system bus.

`mia_byc`

This read-write attribute represents the number of bytes transferred for all memory transactions that reach the system bus.

`rbc`

This read-write attribute indicates the number of read bytes.

`stnch`

This read-write attribute indicates the number of confidence check passed by the confidence mechanism.

`stndsr`

This read-write attribute indicates the number of Stateful Rule executions caused by DXE generated events.

`stnesr`

This read-write attribute indicates the number of Stateful Rule executions caused by End of SUI events.

`stnib`

This read-write attribute indicates the number of input bytes scanned by the Pattern Matcher.

`stnis`

This read-write attribute indicates the number of SUIs scanned by the Pattern Matcher.

`stnob`

This read-write attribute indicates the number of output bytes produced in reports by the Pattern Matcher.

`stnpm`

This read-write attribute indicates the number of pattern matches found by the Data Examination Engine.

`stnprm`

This read-write attribute indicates the number of pattern match attempts within the index range defined by the Pattern Range Counter Index Configuration and Pattern Range Counter Mask Configuration attributes.

`stns1m`

This read-write attribute indicates the number of SUIs with at least one pattern match found within.

`stns1r`

This read-write attribute indicates the number of SUIs scanned that produced at least one report.

`stnth1`

This read-write attribute indicates the number of Triggers found by the 1-Byte Trigger mechanism.

`stnth2`

This read-write attribute indicates the number of Triggers found by the 2-Byte Trigger mechanism.

`stnth3`

This read-write attribute indicates the number of Triggers found by the Special Triggers mechanism.

`stnthv`

This read-write attribute indicates the number of Triggers found by the Variable Length Trigger mechanism.

`tbt0ecc1ec`

This read-write attribute indicates the number of single bit ECC errors detected in the 2-byte Trigger 0 Table.

tbt1ecc1ec

This read-write attribute indicates the number of single bit ECC errors detected in the 2-byte Trigger 1 Table.

trunci

This read-write attribute indicates the count for every scan report that is produced where the report data has been truncated.

vlt0ecc1ec

This read-write attribute the number of single bit ECC errors detected in the Variable Length Trigger 0 Table.

vlt1ecc1ec

This read-write attribute indicates the number of single bit ECC errors detected in the Variable Length Trigger 1Table.

The Pme driver which polls these statistics has a configurable polling interval.

stats_ctrl/update_interval

This read-write attribute indicates the frequency (expressed in milliseconds) of the pme stats kernel thread updates. If 0 is specified, no automatic updates are done. Valid range is 0 to 10000 with a default of 4000 (e.g. 4 seconds).

The following attribute configure statistic related attributes:

stats_ctrl/cmec1th

This read-write attribute contains a threshold that controls reporting of 1-bit ECC errors.

stats_ctrl/dxcmec1th

This read-write attribute contains a threshold that controls reporting of 1-bit ECC errors.

stats_ctrl/dxemec1th

This read-write attribute contains contains a threshold that controls reporting of 1-bit ECC errors.

stats_ctrl/tbt0ecc1th

This read-write attribute contains contains a threshold that controls reporting of 1-bit ECC errors.

stats_ctrl/tbt1ecc1th

This read-write attribute contains contains a threshold that controls reporting of 1-bit ECC errors.

stats_ctrl/vlt0ecc1th

This read-write attribute contains contains a threshold that controls reporting of 1-bit ECC errors.

stats_ctrl/vlt1ecc1th

This read-write attribute contains contains a threshold that controls reporting of 1-bit ECC errors.

5.2.6.3.7.2 PME Database

The /dev/pme_db devices device is used to send Pme pattern matcher database configuration requests. This device requires root permissions and only exists on the control plane. A user may acquire exclusive access to the Pattern Matcher device.

5.2.6.3.7.2.1 PMEIO_EXL_INC

PMEIO_EXL_INC-Acquire and increment exclusivity.

Synopsis

```
ioctl(fd, PMEIO_EXL_INC);
```

Description

This `ioctl()` increments the exclusivity "counter". This enabled the user to acquire and keep exclusivity. This is a synchronous and interruptible API.

Return Value

The API returns 0 on success, -1 with `errno` set on error.

Errors

EINVAL

The parameters specified are not valid

EINTR

The process was interrupted by a signal.

5.2.6.3.7.2.2 *PMEIO_EXL_DEC*

`PMEIO_EXL_DEC`-Decrements reference count and possible release exclusivity.

Synopsis

```
ioctl(fd, PMEIO_EXL_DEC);
```

Description

This `ioctl()` decrement the exclusivity reference counter. If the reference count reaches zero than exclusivity is also released.

Return Value

The API returns 0 on success, -1 with `errno` set on error.

Errors

EINVAL

The parameters specified are not valid

EINTR

The process was interrupted by a signal.

5.2.6.3.7.2.3 *PMEIO_EXL_GET*

`PMEIO_EXL_GET`-Retrieve exclusivity reference count

Synopsis

```
ioctl(fd, PMEIO_EXL_GET, int *count);
```

Description

This `ioctl()` returns the current exclusivity counter and writes the result to `count`.

Return Value

The API returns 0 on success, -1 with `errno` set on error.

Errors

EINVAL

The parameters specified are not valid

EINTR

The process was interrupted by a signal.

5.2.6.3.7.2.4 PMEIO_PMTCC

PMEIO_PMTCC-Send a synchronous pattern matching database request

Synopsis

```
ioctl(fd, PMEIO_PMTCC, struct pme_db *p);

struct pme_db {

    struct pme_buffer input;

    struct pme_buffer output;

    __u8 flags;

    enum pme_status status;

};

struct pme_buffer {

    void *data;

    size_t size;

};
```

Description

This ioctl() is used to send a synchronous pattern matching database request.

p contains the pme pmtcc command and response. Various input and output formats are supported depending on the input flag settings.

p->input is the contiguous input data buffer. Where *p->input.data* is the pointer to the contiguous user-space buffer and *p->input.size* indicates the number of bytes in this buffer.

p->output is where the resulting output/result is stored and is populated by the ioctl(). The *p->output.size* is updated to reflect the number of bytes written.

p->flags attribute is populated by the ioctl and is bit wise mask indicating operation results:

PME_DB_RESULT_UNRELIABLE-When set, indicates that this input was processed following detection and reporting of a serious Pattern Matcher error. Any report data carried within this input that was produced by Pattern Matcher cannot be considered reliable. Output tagged as Unreliable do not have a valid Status code.

PME_DB_RESULT_TRUNCATED-When set, indicates that the contents of the output data were truncated and are incomplete.

p->status attribute is the status code of the operation. This value is invalid if the **PME_DB_RESULT_UNRELIABLE** bit mask was set in *p->flags*. Refer to `fs_l_pme.h` for complete list of possible values:

`pme_status_ok`-operation was successful

all other values indicate an exception occurred.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

ENODEV

EINVAL

The parameters specified are not valid

See Also

5.2.6.3.72.5 *PMEIO_SRE_RESET*

PMEIO_SRE_RESET-Performs an SRE rule reset operation

Synopsis

```
ioctl(fd, PMEIO_SRE_RESET, struct pme_db_sre_reset *sre_reset);  
struct pme_db_sre_reset {  
    __u32 rule_vector[PME_SRE_RULE_VECTOR_SIZE];  
    __u32 rule_index;  
    __u16 rule_increment  
    __u32 rule_repetitions;  
    __u16 rule_reset_interval;  
    __u8 rule_reset_priority;  
};  
#define PME_SRE_RULE_VECTOR_SIZE 8
```

Description

This ioctl() performs a sre rule reset operation. Exclusivity should be held during this operation in order to perform the operation atomically.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

ENODEV

EINVAL

The parameters specified are not valid

See Also

5.2.6.3.72.6 *PMEIO_NOP*

PMEIO_NOP-Send a Pme nop command

Synopsis

```
ioctl(fd, PMEIO_NOP);
```

Description

This ioctl() send a Pme nop command. This has no effect on the Pme device. This is a blocking api. Upon return, this indicates that the response to the nop command has been received.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

ENODEV

EINVAL

The parameters specified are not valid

See Also

5.2.6.3.7.3 PME Scan

The /dev/pme_scan device is used to execute patterning matching from user space. The device can be used to perform synchronous (the calling thread is blocked until an operations completion) or asynchronous (once an operation has begun, the calling thread is able to perform other processing but needs to query the completion of the operation later). This section specifies the ioctl() values supported by the pme_scan device.

5.2.6.3.7.3.1 PMEIO_SCAN

PMEIO_SCAN-Perform a pattern matching request

Synopsis

```
ioctl(fd, PMEIO_SCAN, struct pme_scan *p);
```

```
struct pme_scan {
```

```
    struct pme_scan_cmd cmd;

    struct pme_scan_result result;

};

struct pme_scan_cmd {

    __u32 flags;

    void *opaque;

    struct pme_buffer input;
```

```
    struct pme_buffer output;

};

struct pme_scan_result {

    __u8 flags;

    enum pme_status status;

    struct pme_buffer output;

    void *opaque;

};

struct pme_buffer {

    void *data;

    size_t size;

};
```

Description

This ioctl() is used to perform a synchronous pattern matching request. The API will block until the device has completed the request.

p contains the pme scan input command and response. Various input and output formats are supported depending on the input flag settings.

p->cmd.flags attribute defines input and output formats. It's a bit mask of the following values:

PME_SCAN_CMD_RES_BMAN-use Bman for output

PME_SCAN_CMD_STARTRESET- If residue is disabled indicates a Start of Flow. The first data byte of the input will be treated as the Start of Flow for anchored pattern matching purposes. If residue is enabled indicates a Flow Context Reset. The Start of Flow, Sequence Number and Residue Length fields in the Flow Context Record will be reset to 0x0 prior to scanning the input. Empty input data is accepted and may be used as a mechanism to reset Flow Context without scanning a input data.

PME_SCAN_CMD_END-Indicates that the last byte of input data will be considered as the end of Flow. As well, the Flow Context's sequence number and residue length context are reset to zero such that the next data byte on the Flow will be considered a Start of Flow. In order to accommodate protocols such as TCP, where it may not be known until later that the last processed byte was in fact the End of Flow byte, input data with zero length but with the End of Flow indicator set are allowed. In these cases, the session's residue will be recycled through the Pattern Matcher with the End of Flow indication such that any anchored patterns present won't be missed.

The *p->cmd.opaque* is carried through into *p->result.opaque*.

p->cmd.input is the contiguous input data buffer. Where *p->cmd.input.data* is the pointer to the contiguous user-space buffer and *p->cmd.input.size* indicates the number of bytes in this buffer.

p->cmd.output is where the resulting output/result is stored. The caller indicates where to write the result of the scan via this attribute.

p->result is the response of the scan operation and is updated upon return.

p->result.flags attribute is populated by the ioctl and is bit wise mask indicating operation results:

PME_SCAN_RESULT_UNRELIABLE-When set, indicates that this input was processed following detection and reporting of a serious Pattern Matcher error. Any report data carried within this input that was produced by Pattern Matcher cannot be considered reliable. Output tagged as Unreliable do not have a valid Status code.

PME_SCAN_RESULT_TRUNCATED-When set, indicates that the contents of the output data were truncated and are incomplete.

PME_SCAN_RESULT_BMAN-The output was generated from BMan buffers.

p->result.status attribute is the status code of the operation. This value is invalid if the **PME_SCAN_RESULT_UNRELIABLE** bit mask was set in *p->result.flags*. Refer to *fsl_pme.h* for complete list of possible values:

pme_status_ok-operation was successful

all other values indicate an exception occurred.

p->result.output is updated with the output generated from the Pattern Matcher device. *p->result.output.data* is *p->cmd.output.data* and *p->result.output.size* <= *p->cmd.output.size*. In other words the size of the output from the Pattern Matcher may be less than the size that the caller has supplied.

p->result.opaque is set to the corresponding *p->cmd.opaque* value.

Return Value

The API returns 0 on success, -1 with *errno* set on error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

ENODEV

EINVAL

The parameters specified are not valid

See Also

5.2.6.3.7.3.2 PMEIO_SCAN_W1

PMEIO_SCAN_W1-Perform a single asynchronous pattern matching request

Synopsis

```

        ioctl(fd, PMEIO_SCAN_W1, struct pme_scan_cmd *cmd);
struct pme_scan_cmd {
    __u32 flags;
    void *opaque;
    struct pme_buffer input;
    struct pme_buffer output;
};
struct pme_buffer {
    void *data;
    size_t size;
};

```

Description

This *ioctl()* is used to perform a single asynchronous pattern matching request. The API will return once the request has been dispatched to the Pattern Matcher device. The resulting response must be retrieved via **PMEIO_SCAN_R1** or **PMEIO_SCAN_Rn** *ioctl*s. The input memory should not be modified by the caller until the corresponding response has been retrieved.

cmd contains the pme scan input command. Refer to PMEIO_SCAN ioctl for a detail description.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

ENODEV

EINVAL

The parameters specified are not valid

See Also

5.2.6.3.7.3.3 PMEIO_SCAN_R1

PMEIO_SCAN_R1-Retrieve a single Pattern Matcher response.

Synopsis

```
ioctl(fd, PMEIO_SCAN_R1, struct pme_scan_result *result);
struct pme_scan_result {
    __u8 flags;
    enum pme_status status;
    struct pme_buffer output;
    void *opaque;
};
struct pme_buffer {
    void *data;
    size_t size;
};
```

Description

This ioctl() is used to retrieve a single Pattern Matcher response from a corresponding PMEIO_SCAN_W1 or PMEIO_SCAN_Wn ioctl request. This is a non-blocking api. User should wait for response with select() or poll(), then collect the response(s) with either PMEIO_SCAN_R1 or PMEIO_SCAN_Rn ioctl().

result contains the response from a corresponding PMEIO_SCAN_W1 or PMEIO_SCAN_Wn ioctl(). Refer to PMEIO_SCAN ioctl for a detail description.

Return Value

The API returns 0 on success, -1 with errno set on error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

ENODEV

EINVAL

The parameters specified are not valid

See Also

5.2.6.3.7.3.4 *PMEIO_SCAN_Wn*

PMEIO_SCAN_Wn-Performs multiple asynchronous pattern matching request

Synopsis

```

        ioctl(fd, PMEIO_SCAN_Wn, struct pme_scan_cmds *cmds);
struct pme_scan_cmds {
    unsigned num;
    struct pme_scan_cmd *cmds;
};

```

Description

This `ioctl()` is a more generalized version of the `PMEIO_SCAN_W1` `ioctl()`. Permits sending multiple request asynchronously. The results must be retrieved using the `SCAN_R1` or `SCAN_Rn` `ioctl()`. User's should first call `select()` or `poll()` to indicate that responses are ready to be retrieved. The `ioctl` may not have sent all scan requests: in this case `-EINT` is returns and `cmds->num` is updated with the number of commands sent.

`cmds->cmds` is a pointer to an array of `pme_scan_cmd` structures

`cmds->num` is the number of elements in the above array.

Return Value

The API returns 0 on success, -1 with `errno` set on error.

Errors

`ENOMEM`

Not enough memory could be allocated to satisfy the request.

`ENODEV`

`EINVAL`

The parameters specified are not valid

See Also

5.2.6.3.7.3.5 *PMEIO_SCAN_Rn*

PMEIO_SCAN_Rn-Retrieve multiple Pattern Matcher responses synchronously.

Synopsis

```

        ioctl(fd, PMEIO_SCAN_Rn, struct pme_scan_results *results);
struct pme_scan_results {
    unsigned num;
    struct pme_scan_result *result;
};

```

Description

This `ioctl()` is a more generalized version of the `PMEIO_SCAN_R1` `ioctl()`. Permits retrieving multiple request synchronously.

`results->result` is a pointer to an array of `pme_scan_result` structures

`result->num` is the number of elements in the above array. This value is updated upon return to indicate the actual number of responses retrieved.

Return Value

The API returns 0 on success, -1 with `errno` set on error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

ENODEV

EINVAL

The parameters specified are not valid

See Also

5.2.6.3.7.3.6 *PMEIO_RELEASE_BUFS*

PMEIO_RELEASE_BUFS-Release the allocated buffer back to hardware

Synopsis

```
ioctl(fd, PMEIO_RELEASE_BUFS, void *user_mem);
```

Description

This ioctl() release the allocated buffer back to hardware.

user_mem is a pointer to data which has been zero copied mmap'ed. This memory will be released to BMan. For intance this would be *&result.output.data*.

Return Value

Returns 0 on success, -1 with errno set upon error.

Errors

See Also

5.2.6.3.7.3.7 *PMEIO_RESETSEQ*

PMEIO_RESETSEQ-Resets the flow context sequence number to zero

Synopsis

```
ioctl(fd, PMEIO_RESETSEQ);
```

Description

This ioctl() resets the flow context sequence number to zero. The Start of Flow indicator is also set.

Return Value

Returns 0 on success, -1 with errno set upon error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

EINVAL

The parameters specified are not valid

See Also

5.2.6.3.73.8 *PMEIO_RESETRES*

PMEIO_RESETRES-Resets the flow context residue

Synopsis

```
ioctl(fd, PMEIO_RESETRES);
```

Description

This ioctl() resets the flow context residue such that no bytes remain in residue.

Return Value

Returns 0 on success, -1 with errno set upon error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

EINVAL

The parameters specified are not valid

See Also

5.2.6.3.73.9 *PMEIO_SETSCAN*

PMEIO_SETSCAN-Configures a pme context parameters

Synopsis

```
ioctl(fd, PMEIO_SETSCAN, struct pme_scan_params *p);
```

```
struct pme_scan_params {
```

```
    __u32 flags;

    struct pme_scan_params_residue {
        __u8 enable;
        __u8 length;
    } residue;

    struct pme_scan_params_sre {
        __u32 sessionid;
        __u8 verbose;
        __u8 esee;
    } sre;
};
```

```
    } sre;

    struct pme_scan_params_dxe {
        __u16 clim;

        __u16 mlim;
    } dxe;

    struct pme_scan_params_pattern {
        __u8 set;

        __u16 subset;
    } pattern;
};
```

Description

This `ioctl()` sets the flow context and scan related attributes. This is a blocking API.

p->flags attribute is a bit wise mask which indicate which group of attributes are being set. The flags can be OR'ed together.

PME_SCAN_PARAMS_RESIDUE-Residue attributes are being set.

PME_SCAN_PARAMS_SRE-SRE attributes are being set.

PME_SCAN_PARAMS_DXE-DXE attributes are being set.

PME_SCAN_PARAMS_PATTERN-Pattern attributes are being set.

p->residue.enable attribute turn residue on or off. If non-zero, residue is on, otherwise residue is off.

p->residue.length attribute value is ignored by this `ioctl()` but the underlining number of bytes in residue is reset to zero if the *p->residue.enable* to set.

p->sre.sessionid attribute is the index where the per-session context for this Flow will be accessed by SRE in the SRE table. Range: 0x0 ... 0x7FFFFFFF.

p->sre.verbose attribute is the PME report verbosity mode. There are 4 levels: 0,1,2 and 3.

p->sre.esee attribute turns on or off the enables the End of SUI event. If turned on the End of SUI Reaction pointer programmed in ESRP (End of SUI Reaction Pointer Register) is to be executed upon End of SUI. A value of zero will turn off this attribute, otherwise it will be on.

p->dxe.clim attribute is the compare limit. Valid range is 0x0 to 0xFFFF. 0xFFFF is treated as infinite.

p->dxe.mlim attribute is the match limit. Valid range is 0x0 to 0xFFFF. 0xFFFF is treated as infinite.

p->pattern.set attribute defines an exclusive grouping of patterns (i.e. no set overlap) that are to be searched simultaneously by the Pattern Matcher. The Pattern Matcher supports 256 (mutually exclusive) pattern sets. When scanning for patterns in the data, the search is restricted to a particular pattern set. Valid range is 0x0 to 0xFF.

p->pattern.subset attribute defines An non-exclusive grouping of patterns (subset overlap permitted) within a given pattern set that are to be searched simultaneously by the Pattern Matcher with or without other subsets. Unlike a pattern set, patterns may be assigned to multiple pattern subsets. The Pattern Matcher supports 16 subsets per pattern set. When scanning for patterns, the search may use a list of subsets. Valid range is 0x0 to 0xFFFF.

Return Value

Returns 0 on success, -1 with errno set upon error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

EINVAL

The parameters specified are not valid

See Also

5.2.6.3.73.10 *PMEIO_GETSCAN*

PMEIO_GETSCAN-Retrieves the pme context and scanning attributes.

Synopsis

```
ioctl(fd, PMEIO_GETSCAN, struct pme_scan_params *p);
```

```
struct pme_scan_params {
```

```
    __u32 flags;

    struct pme_scan_params_residue {
        __u8 enable;
        __u8 length;
    } residue;

    struct pme_scan_params_sre {
        __u32 sessionid;
        __u8 verbose;
        __u8 esee;
    } sre;

    struct pme_scan_params_dxe {
        __u16 clim;
        __u16 mlim;
    } dxe;

    struct pme_scan_params_pattern {
        __u8 set;
        __u16 subset;
    } pattern;
};
```

```
    } pattern;  
};
```

Description

This `ioctl()` gets the flow context and scan related attributes. This is a blocking API.

`p->flags` is ignored by the `ioctl()`.

`p->residue.enable` attribute indicates if residue is turn on or off. A value of zero indicates off. A value of non-zero indicates on.

`p->residue.length` attribute indicates the current number of bytes in residue. This attribute has no meaning when residue is disabled.

`p->sre.sessionid` attribute is the index where the per-session context for this Flow will be accessed by SRE in the SRE table.

`p->sre.verbose` attribute is the PME report verbosity mode.

`p->sre.esee` attribute indicates the state of the End of SUI event. If turned on (non zero) the End of SUI Reaction pointer programmed in ESRP (End of SUI Reaction Pointer Register) is to be executed upon End of SUI. A value of zero indicates off.

`p->dxe.clim` attribute is the compare limit.

`p->dxe.mlim` attribute is the match limit.

`p->pattern.set` attribute defines an exclusive grouping of patterns (i.e. no set overlap) that are to be searched simultaneously by the Pattern Matcher. The Pattern Matcher supports 256 (mutually exclusive) pattern sets. When scanning for patterns in the data, the search is restricted to a particular pattern set.

`p->pattern.subset` attribute defines An non-exclusive grouping of patterns (subset overlap permitted) within a given pattern set that are to be searched simultaneously by the Pattern Matcher with or without other subsets. Unlike a pattern set, patterns may be assigned to multiple pattern subsets. The Pattern Matcher supports 16 subsets per pattern set. When scanning for patterns, the search may use a list of subsets.

Return Value

Returns 0 on success and `*p` is updated accordingly, -1 with `errno` set upon error.

Errors

ENOMEM

Not enough memory could be allocated to satisfy the request.

EINVAL

The parameters specified are not valid

See Also

5.2.6.3.7.4 PME Kernel API

This describes the kernel API of the PME driver.

5.2.6.3.7.4.1 `pme_ctx_init`

`pme_ctx_init`-Initializes a pme context for performing scan or control operations

Synopsis

```
int pme_ctx_init(  

```

```

struct pme_ctx *ctx, u32 flags, u32 bpid, u8 qosin, u8 qosout,
enum qm_channel dest, const struct qm_fqd_stashing *stashing);

```

Description

The `pme_ctx_init` function is used to initialize a new pme context that can be used for performing pattern matching or control operations on the pattern matching device.

The `ctx` structure will be initialized. The `ctx->cb` field is required to be set prior to invoking this api.

`flags` defines context attributes and is a bit mask of the following values:

PME_CTX_FLAG_LOCKED-When either frame queues state are updated, this flag indicates if a `spin_lock` is used during the update. If this flag is specified, the `spin_lock` is acquired. This is relevant if different cores will be updating the frame queues state.

PME_CTX_FLAG_EXCLUSIVE-When set, causes the input frame queue to remain in a parked state when the context is enabled. Furthermore, when a frame is enqueued on the input frame queue, the PME's exclusive frame queue control will be enabled. Hence, the context input frame queue is the PME's exclusive frame queue for this enqueue. This mode is only available if the driver is built with control functionality and if the OS has access to the PME's CCSR map.

PME_CTX_FLAG_PMTCC-A pme_context operates in two modes: scan and pmtcc mode. If this flag is specified then pmtcc mode is selected, otherwise scan mode. The mode determines which operations on the pme_context are not permitted. In scan mode `pme_ctx_pmtcc()` operations are not permitted. In pmtcc mode, `pme_ctx_scan()`, `pme_ctx_ctrl_update_flow()` and `pme_ctx_ctrl_read_flow()` are not permitted.

PME_CTX_FLAG_DIRECT-When in scan mode, a pme_context may have a flow context allocated. If this flag is specified, no flow context is allocated, otherwise one is. When in direct mode, the following operations are not permitted: `pme_ctx_ctrl_update_flow()`, `pme_ctx_ctrl_read_flow()`.

PME_CTX_FLAG_LOCAL-Use the current core's dedicated portal channel. The `dest` argument will be ignored.

The `bpid` value specifies the buffer pool id to be used for any bman generated output.

The `qosin` value specifies the workqueue priority on the PME channel (0-7). Only applies if **PME_CTX_FLAG_EXCLUSIVE** is not set.

0, 1-High priority

2, 3, 4-Medium Priority

5, 6, 7-Low Priority

The `qosout` value specifies the workqueue priority on the software portal (0-7). Same priorities as described above.

The `stashing` argument configures the desired dequeue stashing behavior. If NULL, no stashing will occur. Refer to the `qman` api document if non NULL behavior is required.

Return Value

The `pme_ctx_init` API returns 0 on success, and may return the following error codes:

-ENOMEM if there was not enough memory available to satisfy an allocation request

-EIO

-EBUSY

See Also

`pme_ctx_finish()`

5.2.6.3.74.2 *pme_ctx_finish*

pme_ctx_finish-Releases all previously allocated resources (e.g. frame queues, memory) from a disabled *pme_ctx*.

Synopsis

```
void pme_ctx_finish(struct pme_ctx *ctx);
```

Description

This API is used to indicate a *pme_ctx* object is no longer required. Once a *pme* context has been disabled and is no longer required, invoking this api will release all previously allocated resources, such as frame queues, flow context context memory, etc.

See Also

pme_ctx_init()

5.2.6.3.74.3 *pme_ctx_enable*

pme_ctx_enable-Enable a *pme_ctx*

Synopsis

```
int pme_ctx_enable(struct pme_ctx *ctx);
```

Description

This API is used to change the state of a *pme_ctx* to the enabled state. The following operations are not permitted while in the enabled state:

pme_ctx_enable()

pme_ctx_reconfigure_tx()

pme_ctx_reconfigure_rx()

pme_ctx_finish()

Return Value

The API will return 0 on success, and may return the following error codes:

-EBUSY

-EINVAL

-EIO

See Also

pme_ctx_disable()

5.2.6.3.74.4 *pme_ctx_disable*

pme_ctx_disable-Disable a *pme_context* object

Synopsis

```
int pme_ctx_disable(struct pme_ctx *ctx, u32 flags, struct pme_ctx_ctrl_token *token);
```

Description

This API disables a previously enabled `pme_context`.

If the return value is 0, the context is disabled.

If the return value is +1, the context is disabling and the token's completion callback will be invoked when disabling is complete.

Otherwise an error is returned and the context remains enabled.

`flags` affects the behavior of this API and is a bit mask of the following values:

`PME_CTX_OP_WAIT`-Indicates that the api can sleep.

`PME_CTX_OP_WAIT_INT`-This qualifies the `PME_CTX_OP_WAIT` flag. Indicates that the sleep is interruptible. Therefore, `PME_CTX_OP_WAIT_INT` on it's own is undefined.

`token` parameter is "owned" by the driver. The driver will write command specific data to this structure. This parameter is "returned" (i.e. driver relinquishes ownership) to the user via the callback function specified in `token` object. The `cb` (callback) member must be set (i.e. non NULL) when a callback is expected.

Return Value

The API will return 0 on success, +1 when the callback will be invoked or one of the following errors on failure:

-EBUSY

-EINTR

-EIO

See Also

`pme_ctx_enable()`

`pme_ctx_is_disable()`

5.2.6.3.7.4.5 *pme_ctx_is_disable*

`pme_ctx_is_disable`-query whether a pme context is disabled

Synopsis

```
int pme_ctx_is_disabled(struct pme_ctx *ctx);
```

Description

This API queries whether a pme context is disabled.

Return Value

The API will return >0 if the ctx is disabled. Otherwise 0 is returned is ctx is not disabled.

See Also

`pme_ctx_disable()`

5.2.6.3.7.4.6 *pme_ctx_is_dead*

`pme_ctx_is_dead`-query whether a pme context is dead

Synopsis

```
int pme_ctx_is_dead(struct pme_ctx *ctx);
```

Description

This API queries whether a pme context is dead.

Return Value

The API will return >0 if the ctx is dead. Otherwise 0 is returned if ctx is not dead.

See Also

`pme_ctx_is_disable()`

5.2.6.3.7.4.7 *pme_ctx_reconfigure_tx*

`pme_ctx_reconfigure_tx`-reconfigure transmit frame queue

Synopsis

```
int pme_ctx_reconfigure_tx(struct pme_ctx *ctx, u32 bpid, u8 qosin);
```

Description

This API reconfigure the tx qman frame queue. The `pme_ctx` must be in a disabled state.

bpid specifies the report buffer pool id that the Pme device shall use when BMan output is generated.

qosin indicates which of the Pme's 8 prioritized workqueues the frame queue should schedule to. This is only applicable if the `PME_CTX_FLAG_EXCLUSIVE` context flag is not set.

Return Value

The `pme_ctx_reconfigure_tx` API returns 0 on success, and may return the following error codes:

- EBUSY
- EIO
- EINVAL

See Also

`pme_ctx_disable()`

5.2.6.3.7.4.8 *pme_ctx_reconfigure_rx*

`pme_ctx_reconfigure_rx`-reconfigure receive frame queue

Synopsis

```
int pme_ctx_reconfigure_rx(struct pme_ctx *ctx, u8 qosout, enum qm_channel dest,  
const struct qm_fqd_stashing *stashing);
```

Description

This API reconfigure the rx qman frame queue. The `pme_ctx` must be in a disabled state.

qosout indicates which of the 8 prioritized workqueues the frame queue should be scheduled to on the software portal.

dest specifies which pool channel *or* dedicated channel to use. Ignored if the `pme_ctx` was initialized with the `PME_CTX_FLAG_LOCAL` flag.

stashing specifies a stashing object. Refer to `qman api` for more detail on this object.

Return Value

The `pme_ctx_reconfigure_rx` API returns 0 on success, and may return the following error codes:

- EBUSY
- EIO
- EINVAL

See Also

`pme_ctx_disable()`

5.2.6.3.74.9 *pme_ctx_ctrl_update_flow*

`pme_ctx_ctrl_update_flow`-Updates the associated pme flow context record

Synopsis

```
int pme_ctx_ctrl_update_flow(struct pme_ctx *ctx, u32 flags, struct pme_flow
*params, struct pme_ctx_ctrl_token *token);
```

Description

This API updates the associated pme flow context record in the Pme device according to the values specified in *params*. For instance to enable residue, set the `PME_CMD_FCW_RES` flags parameter as well as the *params->ren* and *params->rlen* values. To disable residue, set the `PME_CMD_FCW_RES` flags parameter and set the *params->ren* to 0. The `pme_ctx` must be in the following state: enabled, flow mode and scan mode (e.g. `PME_CTX_FLAG_DIRECT` and `PME_CTX_FLAG_PMTCC` were not specified during initialization).

flags indicates which fields in the flow context are to be updated. These values can be or'ed together with the exception of `PME_CTX_OP_RESETRESLEN`.

`PME_CMD_FCW_RES`-Residue related attributes can be updated. These attributes are: *params->ren*, *params->rlen*

`PME_CMD_FCW_SEQ`-Sequence related attributes can be updated. These attributes are: *params->sos*, *params->seqnum*

`PME_CMD_FCW_SRE`-Stateful rule related attributes can be updated. These attributes are: *params->svrm*, *params->esee* and *params->sessionid*

`PME_CMD_FCW_DXE`-Data Examination related attributes can be updated. These attributes are: *params->clim* and *params->mlim*

`PME_CMD_FCW_ALL`-All of the above attributes can be updated.

`PME_CTX_OP_RESETRESLEN`-Applies only to a `pme_ctx` which has residue enabled. The following attribute is updated: *params->rlen*. This flag should only be used on it's own.

`PME_CTX_OP_WAIT`-Indicates that the api can sleep.

`PME_CTX_OP_WAIT_INT`-This qualifies the `PME_CTX_OP_WAIT` flag. Indicates that the sleep is interruptible. Therefore, `PME_CTX_OP_WAIT_INT` on it's own is undefined.

params is the pme flow structure. The contents of this structure get copied by this api.

token parameter is "owned" by the driver. The driver will write command specific data to this structure. This parameter is "returned" (i.e. driver relinquishes ownership) to the user via the callback function specified in *token* object.

Return Value

The `pme_ctx_ctrl_update_flow` API returns 0 on success, and may return the following error codes:

- EBUSY
- EIO

-EINVAL

-ENOMEM

See Also

5.2.6.3.74.10 *pme_ctx_ctrl_read_flow*

pme_ctx_ctrl_read_flow-read the flow record in the Pme device.

Synopsis

```
int pme_ctx_ctrl_read_flow(struct pme_ctx *ctx, u32 flags, struct pme_flow
*params, struct pme_ctx_ctrl_token *token);
```

Description

This API sends a ctrl operation request to read the flow record in the Pme device into the *params* argument. The *ctx* must be in the following state: enabled, flow mode and scan mode(e.g. PME_CTX_FLAG_DIRECT and PME_CTX_FLAG_PMTCC were not specified during initialization).

flags affects the behavior of this API and is a bit mask of the following values:

PME_CTX_OP_WAIT-Indicates that the api can sleep.

PME_CTX_OP_WAIT_INT-This qualifies the PME_CTX_OP_WAIT flag. Indicates that the sleep is interruptible. Therefore, PME_CTX_OP_WAIT_INT on it's own is undefined.

params is a pointer to a flow context object. The *params* will be updated with upon invocation of the callback specified in the token parameter.

token parameter is "owned" by the driver. The driver will write command specific data to this structure. This parameter is "returned" (i.e. driver relinquishes ownership) to the user via the callback function specified in *token* object.

Return Value

The *pme_ctx_ctrl_read_flow* API returns 0 on success, and may return the following error codes:

-EINTR

-EBUSY

-EINVAL

See Also

5.2.6.3.74.11 *pme_ctx_ctrl_nop*

pme_ctx_ctrl_nop-sends a Pme nop command

Synopsis

```
int pme_ctx_ctrl_nop(struct pme_ctx *ctx, u32 flags, struct pme_ctx_ctrl_token
*token);
```

Description

This API sends a pme nop command. The *ctx* must be enabled.

flags affects the behavior of this API and is a bit mask of the following values:

PME_CTX_OP_WAIT-Indicates that the api can sleep.

PME_CTX_OP_WAIT_INT-This qualifies the PME_CTX_OP_WAIT flag. Indicates that the sleep is interruptible. Therefore, PME_CTX_OP_WAIT_INT on it's own is undefined.

token parameter is "owned" by the driver. The driver will write command specific data to this structure. This parameter is "returned" (i.e. driver relinquishes ownership) to the user via the callback function specified in *token* object.

Return Value

The `pme_ctx_ctrl_nop` API returns 0 on success, and may return the following error codes:

- EINTR
- EBUSY

See Also

5.2.6.3.7.4.12 *cb*

cb-Function to invoke when response to a `ctrl` api (`update_flow`, `read_flow`, `nop`) operation has been returned from Pme hardware.

Synopsis

```
void (*cb)(struct pme_ctx *ctx, const struct qm_fd *fd, struct pme_ctx_ctrl_token
*token);
void (*ern_cb)(struct pme_ctx *, const struct qm_mr_entry *, struct
pme_ctx_ctrl_token *);
```

Description

This *cb* function is specified in the `pme_ctx_ctrl_token` parameter specified in the `ctrl` apis. When the response to a `pme_ctx_ctrl_update_flow()`, `pme_ctx_ctrl_read_flow`, `pme_ctx_ctrl_nop()` or `pme_ctx_disable()` is received this callback is invoked (usually in interrupt context) and appropriate action must be taken by the user. In the case of `pme_ctx_disable()`, the callback is not invoked if the api returns 0. Since the callback may be in interrupt context user must ensure that proper requirements are met (e.g. sleeping is not permitted, etc).

The `ern_cb` function is invoked when an enqueue rejection occurs. This may occur before order-restoration or after (eg. if due to congestion or tail-drop). Use the `rc` code of the `mr_entry` to determine.

ctx is a `pme` context structure.

fd is a frame description which has been dequeued from the output frame queue. In other words this is the response from the Pme device.

token is the parameter originally passed in the corresponding `pme_ctx_ctrl` api. The user is now given ownership of this parameter.

See Also

- `pme_ctx_ctrl_update_flow()`
- `pme_ctx_ctrl_read_flow()`
- `pme_ctx_ctrl_nop()`
- `pme_ctx_disable()`

5.2.6.3.7.4.13 *pme_ctx_exclusive_inc*

`pme_ctx_exclusive_inc`-Increment `pme` exclusivity reference count

Synopsis

```
int pme_ctx_exclusive_inc(struct pme_ctx *ctx, u32 flags);
```

Description

This API acquires Pme exclusivity (if not already held) and increments the reference count in *ctx* . This API can only be invoked on the control plane and only on a *ctx* that was initialized with the `PME_CTX_FLAG_EXCLUSIVE` flag. If exclusivity cannot be immediately acquired the *flags* argument determines whether the API sleeps or not.

flags affects the behavior of this API and is a bit mask of the following values:

`PME_CTX_OP_WAIT`-Indicates that the api can sleep.

`PME_CTX_OP_WAIT_INT`-This qualifies the `PME_CTX_OP_WAIT` flag. Indicates that the sleep is interruptible. Therefore, `PME_CTX_OP_WAIT_INT` on it's own is undefined.

Return Value

The `pme_ctx_exclusive_inc` API returns 0 on success, and may return the following error codes:

-ENODEV

-EBUSY

See Also

`pme_ctx_exclusive_dec()`

5.2.6.3.75 pme_ctx_exclusive_dec

`pme_ctx_exclusive_dec`-Decrement pme exclusivity reference count

Synopsis

```
void pme_ctx_exclusive_dec(struct pme_ctx *ctx);
```

Description

This API decrements the exclusivity reference count and if this is the last reference also releases Pme exclusivity. This API can only be invoked on the control plane and only on a *ctx* that was initialized with the `PME_CTX_FLAG_EXCLUSIVE` flag.

Return Value

No return value.

See Also

`pme_ctx_exclusive_inc()`

5.2.6.3.76 pme_scan_cb

`pme_scan_cb`-Function to invoke when response to scan or pmtcc operation has been returned from Pme hardware.

Synopsis

```
typedef void (*pme_scan_cb)(struct pme_ctx *ctx, const struct qm_fd *fd, struct  
pme_ctx_token *token);
```

Description

This callback function is specified prior to the `pme_ctx_init()` API as part of the `ctx->cb` parameter. When the response to a `pme_ctx_scan()` or `pme_ctx_pmtcc()` is received this callback is invoked (usually in interrupt context) and appropriate action must be taken by the user. Since the callback maybe in interrupt context user must ensure that proper requirements are met (e.g. sleeping is not permitted, etc).

ctx is a pme context structure.

fd is a frame description which has been dequeued from the output frame queue. In other words this is the response from the Pme device.

token is the parameter originally passed in the corresponding `pme_ctx_scan()` or `pme_ctx_pmtcc()` commands. The user is now given ownership of this parameter. Typically a user may "outcast" this pointer to a larger object (i.e. this token is embedded within a user defined structure which contains additional context data).

See Also

`pme_ctx_init()`
`pme_ctx_scan()`
`pme_ctx_pmtcc()`

5.2.6.3.77 pme_scan_ern_cb

`pme_scan_ern_cb`-Function to invoke when enqueue rejection occurs during a scan or pmtcc operation.

Synopsis

```
typedef void (*pme_scan_ern_cb)(struct pme_ctx *ctx, const struct qm_mr_entry
*mr, struct pme_ctx_token *token);
```

Description

This error rejection notification callback function is specified prior to the `pme_ctx_init()` API as part of the `ctx->ern_cb` parameter. When an error rejection notification is received this callback is invoked (usually in interrupt context) and appropriate action must be taken by the user. Since the callback may be in interrupt context user must ensure that proper requirements are met (e.g. sleeping is not permitted, etc).

mr is an ern maessage response structure. The contained ern structure contains the rejection code (*rc*) and the associated frame descriptor (*fd*) which was rejected.

token is the parameter originally passed in the corresponding `pme_ctx_scan()` or `pme_ctx_pmtcc()` commands. The user is given ownership of this parameter. Typically a user may "outcast" this pointer to a larger object (i.e. this token is embedded within a user defined structure which contains additional context data).

See Also

`pme_ctx_init()`
`pme_ctx_scan()`
`pme_ctx_pmtcc`

5.2.6.3.78 PME_SCAN_ARGS

`PME_SCAN_ARGS`-Macro that modifies the frame description. Used in the `pme_ctx_scan()` API

Synopsis

```
#define PME_SCAN_ARGS(flags, set, subset)
```

Description

Modify a frame descriptor for a `pme_ctx_scan` api (only modifies `fd->cmd` field).

flags is a bit mask of the following values:

PME_CMD_SCAN_SRVM(n)-Scan Report Verbosity Mode. Where $n = \{0..3\}$. This flag is ignored by the Pme device when the context is configured for direct mode.

PME_CMD_SCAN_FLUSH-Instructs the Pme device to flush any cached Flow Context and Residue data to system memory after the Frame is processed. This flag is ignored by the Pme device when the context is configured for direct mode.

PME_CMD_SCAN_SR-Depends on the pme ctx mode and possibly on the state of the flow context. Direct mode-Start of Flow indication. The first data byte of the Frame will be treated as the Start of Flow for anchored pattern matching purposes. Flow mode-Depends if residue is enabled or not. residue disabled: Start of Flow indication. The first data byte of the Frame will be treated as the Start of Flow for anchored pattern matching purposes. residue enabled: Flow Context Reset. The Start of Flow, Sequence Number and Residue Length fields in the Flow Context Record will be reset to 0x0 prior to scanning the Frame. Empty input Frames are accepted and may be used as a mechanism to reset Flow Context without scanning a Frame

PME_CMD_SCAN_E-End of Flow. Depends on the ctx mode. Direct mode: Indicates that the last symbol of the scanned work unit is to be considered an End of Flow. Flow mode: Indicates that the last byte of input data will be considered as the end of Flow. As well, the Flow Context's sequence number and residue length context are reset to zero such that the next data byte on the Flow will be considered a Start of Flow. In order to accommodate protocols such as TCP, where it may not be known until later that the last processed byte was in fact the End of Flow byte, Frames with zero length input data but with the End of Flow indicator set are allowed. In these cases, the session's residue will be recycled through the Pattern Matcher with the End of Flow indication such that any anchored patterns present won't be missed.

set is the pattern set value. An exclusive grouping of patterns (i.e. no set overlap) that are to be searched simultaneously by the Pattern Matcher. The Pattern Matcher supports 256 (mutually exclusive) pattern sets. When scanning for patterns in the data, the search is restricted to a particular pattern set.

subset is the pattern subset value. A non-exclusive grouping of patterns (subset overlap permitted) within a given pattern set that are to be searched simultaneously by the Pattern Matcher with or without other subsets. Unlike a pattern set, patterns may be assigned to multiple pattern subsets. The Pattern Matcher supports 16 subsets per pattern set. When scanning for patterns, the search may use a list of subsets.

See Also

`pme_ctx_scan()`

5.2.6.3.7.9 pme_ctx_scan

`pme_ctx_scan`-sends a Pme scan command

Synopsis

```
int pme_ctx_scan(struct pme_ctx *ctx, u32 flags, struct qm_fd *fd, u32 args,
struct pme_ctx_token *token);
```

Description

Send a pme scan command. The *ctx* must be enabled and not in the pmtcc mode. If *ctx.flags* indicate exclusivity then Pme exclusivity is acquired before the request is sent. The API returns zero upon successful enqueue of the *fd*. The corresponding *ctx.cb* function is invoked when the response to the scan request has been dequeued. The *ctx.cb* function may be invoked before this api has completed. The *ctx.cb* is invoked in interrupt context (when applicable).

ctx is an enabled pme context that is in scan mode (e.g. The *ctx* was not initialized with `PME_CTX_FLAG_PMTCC`).

flags affects the behavior of this API and is a bit mask of the following values:

`PME_CTX_OP_WAIT`-Indicates that the api can sleep.

`PME_CTX_OP_WAIT_INT`-This qualifies the `PME_CTX_OP_WAIT` flag. Indicates that the sleep is interruptible. Therefore, `PME_CTX_OP_WAIT_INT` on it's own is undefined.

fd is a frame description as defined in the Pattern Matcher Block Guide.

args parameter is produced by the `PME_SCAN_ARGS()` macro.

token parameter is "owned" by the driver. The driver will write command specific data to this structure. This parameter is "returned" (i.e. driver relinquishes ownership) to the user via the callback function specified in pme ctx object. A caller will typically embed this token object as part of a larger object so as to maintain his own per command data.

Return Value

The pme_ctx_scan API returns 0 on success, and may return the following error codes:

-EINTR
-EBUSY

See Also

pme_ctx_enable()
PME_SCAN_ARGS()

5.2.6.3.7.10 pme_ctx_scan_orp

pme_ctx_scan_orp-sends a Pme scan command with order restoration

Synopsis

```
int pme_ctx_scan_orp(struct pme_ctx *ctx, u32 flags, struct qm_fd *fd, u32 args,
struct pme_ctx_token *token, struct qman_fq *orp_fq, u16 seqnum);
```

Description

This extends the pme_ctx_scan() API to provide order restoration support. The *orp_fq* represents the frame queue descriptor that is to be used as the order restoration point and the *seqnum* is the sequence number to use for order restoration.

ctx is an enabled pme context that is in scan mode (e.g. The *ctx* was not initialized with PME_CTX_FLAG_PMTCC).

flags affects the behavior of this API and is a bit mask of the following values:

PME_CTX_OP_WAIT-Indicates that the api can sleep.

PME_CTX_OP_WAIT_INT-This qualifies the PME_CTX_OP_WAIT flag. Indicates that the sleep is interruptible. Therefore, PME_CTX_OP_WAIT_INT on it's own is undefined.

fd is a frame description as defined in the Pattern Matcher Block Guide.

args parameter is produced by the PME_SCAN_ARGS() macro.

token parameter is "owned" by the driver. The driver will write command specific data to this structure. This parameter is "returned" (i.e. driver relinquishes ownership) to the user via the callback function specified in pme ctx object. A caller will typically embed this token object as part of a larger object so as to maintain his own per command data.

orp_fq is the order restoration point frame queue to be used.

seqnum is the sequence number to be used for order restoration.

Return Value

The pme_ctx_scan_orp API returns 0 on success, and may return the following error codes:

-EINTR
-EBUSY

See Also

pme_ctx_enable()
PME_SCAN_ARGS()

5.2.6.3.711 pme_ctx_pmtcc

pme_ctx_pmtcc-sends a Pme pmtcc command

Synopsis

```
int pme_ctx_pmtcc(struct pme_ctx *ctx, u32 flags, struct qm_fd *fd, struct  
pme_ctx_token *token);
```

Description

Send a Pme pmtcc command. The *ctx* must be enabled and have been initialized with the PME_CTX_FLAG_PMTCC flag. The corresponding *ctx->cb* function is invoked when the response to the pmtcc request has been received. The *ctx->cb* function may be invoked before this api has completed. The *ctx->cb* is invoked in interrupt context (when applicable). The token parameter is returned via the *ctx->cb* function.

This API will attempt to acquire exclusivity if the *ctx* doesn't already have it. Exclusivity can only be acquired on the control plane.

ctx is an enabled pme context that is in pmtcc mode (e.g. The *ctx* was initialized with PME_CTX_FLAG_PMTCC).

flags affects the behavior of this API and is a bit mask of the following values:

PME_CTX_OP_WAIT-Indicates that the api can sleep.

PME_CTX_OP_WAIT_INT-This qualifies the PME_CTX_OP_WAIT flag. Indicates that the sleep is interruptible. Therefore, PME_CTX_OP_WAIT_INT on it's own is undefined.

fd is a frame description as defined in the Pattern Matcher Block Guide.

token parameter is "owned" by the driver. The driver will write command specific data to this structure. This parameter is "returned" (i.e. driver relinquishes ownership) to the user via the callback function specified in pme ctx object. A caller will typically embed this token object as part of a larger object so as to maintain his own per command data.

Return Value

The pme_ctx_pmtcc API returns 0 on success, and may return the following error codes:

-EINTR

-EBUSY

See Also

pme_ctx_enable()

pme_ctx_init()

5.2.6.3.712 pme_attr_set

pme_attr_set-Write to a Pme attribute

Synopsis

```
int pme_attr_set(enum pme_attr attr, u32 val);
```

Description

This API permits the setting of certain Pme attributes as defines by the enum pme_attr set. This API is only available on the control plane.

attr is the Pme attribute to write to.

val is the value to be written

Return Value

The pme_attr_set API returns 0 on success, and may return the following error codes:

-ENODEV

See Also

pme_attr_get()

5.2.6.3.7.13 pme_attr_get

pme_attr_get-Reads from a Pme attribute

Synopsis

```
int pme_attr_get(enum pme_attr attr, u32 *val);
```

Description

This API permits the reading of certain Pme attributes as defines by the enum pme_attr set. This API is only available on the control plane.

attr is the Pme attribute to read.

val is the value read

Return Value

The pme_attr_get API returns 0 on success, and may return the following error codes:

-ENODEV

See Also

pme_attr_set()

5.2.6.3.7.14 pme2_have_control

pme2_have_control-Query if there is access to the Pme CCSR register space

Synopsis

```
int pme2_have_control(void);
```

Description

This API return >0 is there is access to the Pme CCSR register space (i.e. caller is on the control plane). Otherwise 0 is returned. Some apis require the caller to have access to CCSR space (such as pme_attr_get()), this api permits a caller to determine is CCSR space is accessible.

Return Value

The pme2_have _control API returns 0 the caller does not have CCSR access. Otherwise a non zero value is returned.

See Also

5.2.6.3.7.15 pme_stat_get

pme_stat_get-Query a Pme statistics attribute

Synopsis

```
int pme_stat_get(enum pme_attr attr, u64 *value, int reset);
```

Description

This api returns an accumulated version of the related attribute. At a configured interval, the Pme driver will query all statistic attributes and maintain an accumulated counter (i.e. the value read is added to the current count). This api also permits resetting the accumulated counter. This API is only available on the control plane.

attr is one of

pme_attr_trunci
pme_attr_rbc
pme_attr_tbt0ecc1ec
pme_attr_tbt1ecc1ec
pme_attr_vlt0ecc1ec
pme_attr_vlt1ecc1ec
pme_attr_cmecc1ec
pme_attr_dxcmecc1ec
pme_attr_dxemecc1ec
pme_attr_stnib
pme_attr_stnis
pme_attr_stnth1
pme_attr_stnth2
pme_attr_stnthv
pme_attr_stnthS
pme_attr_stnch
pme_attr_stnpm
pme_attr_stns1m
pme_attr_stnpr
pme_attr_stndsr
pme_attr_stnesr
pme_attr_stns1r
pme_attr_stnob
pme_attr_mia_byc
pme_attr_mia_blc

value is updated to the current accumulated count.

reset indicates is the accumulated count is to be reset after updating value. A value of 0 will not reset the counter. A value of non-zero will reset the counter. In either case the current accumulated count is returned.

This API returns 0 in success.

Return Value

-ENODEV
-EINVAL

See Also

pme2_have_control()

5.2.6.3.8 Control interface

This chapter provides a description of the API of the Pattern Matcher control interface (PMCI).

PMCI module is a Linux user space library that provides C functional interface to send and receive Pattern Matcher control commands to the Pattern Matcher through Pattern Matcher driver software. These commands allow users to program and monitor the Pattern Matcher database on the hardware. The PMCI converts complex PME control commands into PME driver primitives.

In general, PME control interface is used in the following order:

Initialize a PMCI object using `pmci_open()`.

Use `pmci_write()` to write single or multiple PME control commands. These commands follow the Pattern Matcher protocol (PMP) format.

Use `pmci_read()` to read responses from commands that generate responses.

Use `pmci_close()` to close the PMCI object.

The table below lists the files that constitute the PMCI package.

PMCI files

<code>libpmci.a</code>	PMCI functionality
<code>pmci.h</code>	Defines the interface to the PMCI module
<code>pmci.c</code>	Implementation of the PMCI module
<code>genTypes.h</code>	Contains the generic type definitions and is included by <code>pmci.h</code>
<code>pmDefs.h</code>	Defines the interface that is common to PMLL, PMREC, PMSRC, etc., and is included by <code>pmci.h</code>
<code>pmp.h</code>	Contains the definition of the Pattern Matcher Protocol and is included by <code>pmci.h</code>

5.2.6.3.8.1 `pmci_open`

Initialize a new instance of PME Control Interface channel, and return the handle.

Synopsis

```
#include <pmci.h>
pmci_error_t pmci_open(int channel, handle_t *handle)
```

Description

This function opens a new PME Control Interface channel and returns its handle. Such a handle is required on any subsequent PME Control Interface operations.

`channel` specifies the DMA channel to use for communication with hardware. The possible values are from between 0 and 3.

`handle` specifies the variable where the new handle value is returned.

Return Value

`handle` points to the new PMCI handle if open was successful.

`pmci_success_e` is returned for a successful open, otherwise one of the following errors is returned:

`pmci_unavailable_driver_e`

Part of required driver modules not loaded.

pmci_invalid_channel_e

DMA channel was not configured.

5.2.6.3.8.2 pmci_set_option

Apply option to PMCI handle to modify its behaviour.

Synopsis

```
#include <pmci.h>
```

```
pmci_error_t pmci_set_option(handle_t pmci_handle, pmci_option_id_t optionId, void  
*option, int optionSize)
```

Description

This function provides the ability to change the behavior of the PMCI handle by applying options. The supported options are defined by the `pmci_option_id_t` type. Currently, the only available option are `pmci_option_timeout_e` which changes the timeout behaviour for `pmci_read()`, and `pmci_option_batch_buffer_threshold_e` which adjust how much buffer memory can be used when batch attribute is enabled.

pmci_handle is the PMCI handle to be modified.

optionId is the behaviour to be changed.

option is the new value of the option

optionSize is the size of *option*, in number of bytes.

Return Value

Returns `pmci_success_e` for successful operation, otherwise one of the following error codes is returned:

`pmci_invalid_handle_e`

PMCI handle is invalid.

`pmci_invalid_option_code_e`

Option id is invalid.

`pmci_unavailable_option_e`

Option id is invalid.

`pmci_invalid_parameters_e`

Option value is invalid.

`pmci_invalid_option_size_e`

Option size is invalid.

5.2.6.3.8.3 pmci_get_option

Retrieve the option value previously set by `pmci_set_option`.

Synopsis

```
#include <pmci.h>
```

```
pmci_error_t pmci_get_option(handle_t pmci_handle, pmci_option_id_t optionId, void  
*option, int *optionSize)
```

Description

This function provides the ability to retrieve the current values of options used by the PMCI *handle*. These are either values set by the user or default values set upon initialization.

pmci_handle is the PMCI handle

optionId is the identifier id of the option being queried.

option is the returned value of the option being queried.

optionSize is the returned size of *option*, in number of bytes.

Return Value

Returns `pmci_success_e` upon successful completion, otherwise returns one of the following errors. Upon return, `*option` will contain the retrieved option value, and `*optionSize` will contain the size of the value in bytes.

`pmci_invalid_handle_e`

PMCI handle is invalid.

`pmci_invalid_option_code_e`

Option id is invalid.

`pmci_unavailable_option_e`

Option id is invalid.

`pmci_invalid_parameters_e`

option or optionSize pointer is invalid.

5.2.6.3.8.4 pmci_close

Shut down a PME Control Interface channel.

Synopsis

```
#include <pmci.h>
```

```
pmci_error_t pmci_close (handle_t pmci_handle)
```

Description

This function shuts down a PME Control Interface. During the close all resources related to *pmci_handle* will be released.

pmci_handle is the handle of the control interface to be shut down.

Return Value

Returns `pmci_success_e` for a successful close, otherwise returns one of the following error codes:

`pmci_invalid_handle_e`

Invalid PMCI handle.

`pmci_failure_e`

Failed to release some resources.

5.2.6.3.8.5 pmci_write

Write one or more commands to the PME Control Interface.

Synopsis

```
#include <pmci.h>
```

```
pmci_error_t pmci_write(handle_t pmci_handle, void *cmds, int cmdsSize)
```

Description

This function is used to write one or more commands to the PME control device. These commands follow the PME command protocol (PMP) format. The PMCI writes these commands into control descriptors created by the PME driver for communicating to the hardware.

Some PME control commands are sent directly to hardware, and some will be handled by PMCI software. A software command may be expanded into multiple hardware commands inside PMCI.

pmci_handle is the PMCI handle in use.

cmd is a pointer to a buffer containing one or more PME commands (in PMP format)..

cmdSize is the total size of all PME commands, in bytes, being sent to the PMCI.

Return Value

Returns `pmci_success_e` upon successful completion, otherwise returns one of the following error codes:

`pmci_invalid_handle_e`

PMCI handle is invalid.

`pmci_invalid_attribute_id_e`

Invalid PM control command.

`pmci_failure_e`

Invalid PM control command or hardware failure.

`pmci_invalid_parameters_e`

command buffer is set to NULL, or *cmdsSize* is 0

5.2.6.3.8.6 pmci_read

Read a notification from PME Control Interface.

Synopsis

```
#include <pmci.h>
```

```
pmci_error_t pmci_read(handle_t pmci_handle, pmp_msg_t *notif)
```

Description

This function is used to read command notifications from the PME Control Interface. PME command protocol contains commands that generate a notification. These notifications can be read by calling this function.

pmci_handle is the PMCI handle in use.

notif points to the user defined notification buffer where notifications will be returned. The buffer size should be the size of `pmp_msg_t` which is defined in `pmp.h`

Return Value

Returns `pmci_success_e` upon successful completion, otherwise it returns on of the following error codes:

`pmci_invalid_handle_e`

PMCI handle is invalid.

`pmci_empty_read_e`

Read timed out and no notification was read.

`pmci_failure_e`

Invalid PM control command or hardware failure.

`pmci_invalid_parameters_e`
notification buffer pointer is set to NULL

5.2.6.3.8.7 `pmci_flush`

Flush the PME Control Interface and ensure no more data is in flight.

Synopsis

```
#include <pmci.h>
```

```
pmci_error_t pmci_flush(handle_t pmci_handle)
```

Description

This function provides the ability to flush the PME driver pipe and ensure no more data is in flight. The call blocks until the PME driver has ensured its command pipeline is empty and that all preceding commands have not only reached the hardware but has also completed execution.

pmci_handle is the PMCI handle in use.

Return Value

Returns `pmci_success_e` upon successful completion, otherwise returns one of the following errors:

`pmci_invalid_handle_e`

PMCI handle is invalid.

`pmci_lost_driver_e`

PM driver did not perform the operation.

`pmci_failure_e`

Hardware failure.

5.2.6.3.8.8 `pmci_context_clear_by_session_id`

Clear the stateful rule context associated to a specific session Id.

Synopsis

```
#include <pmci.h>
```

```
pmci_error_t pmci_context_clear_by_session_id(uint32_t sessionId)
```

Description

This utility function provides the ability to clear the stateful rule context associated to a specific session Id. The PME hardware provides the functionality to clear its stateful rule context for a particular session. This function provides users the means to invoke that functionality. It opens a PMCI handle, sends the hardware command to clear the context, and then close the PMCI handle.

session_id is the session id whose associated context information is to be cleared.

Return Value

Returns `pmci_success_e` upon successful completion, otherwise returns one of the following errors:

`pmci_unavailable_driver_e`

Driver not loaded or configured.

`pmci_failure_e`

Driver not configured properly, or HW failure.

pmci_invalid_parameters_e

Bad session id.

5.2.6.3.8.9 pmci_error_string

Returns a string describing the PMCI error code.

Synopsis

```
#include <pmci.h>
```

```
const char *pmci_error_string(pmci_error_t code)
```

Description

This utility function returns a printable string corresponding to the pmci error code.

code is the error code returned from other PMCI functions.

Return Value

Returns a string pointer to an appropriate error message.

5.2.6.3.9 PMP message format

The Pattern Matcher protocol (PMP) defines the format of messages that are used to setup and manage the PME database. The following sections describe in detail the various PMP message formats.

5.2.6.3.9.1 PME Message Syntax

This section summarizes the PMP message syntax. Each PMP message has the following format:

Version (8 bits)-version number of the PME control protocol

Type (8 bits)-command type code

Reserved (16 bits)-for future use

Length (32 bits)-total length in bytes, including header bytes

Message ID (64 bits)-command message sequence number, helps to co-relate the response message with the command message

Command (variable)-control command information, specific to each type of message

Commands and Notifications Types

5.2.6.3.9.2 PMP Message Types

There are various types of PMP messages. These messages vary in format and behavior based on their type. Some messages are one-way only and represent commands to the PME whereas other messages expect a response message or notification from the PME.

The following table lists the various types of messages.

Summary of Numerical Assignment of Type Field

Category	Type	Description ^[10]
<i>Table continues on the next page...</i>		

[10] The descriptions that contain a code in parenthesis have the following legend: C = Command, N = Notification, and V = Virtualized (not directly supported by the hardware). Also note that read and get notifications uses the same type as the commands except the most significant bit is set to differentiate between the request and the reply.

Table continued from the previous page...

Table Manipulation	0x00	Read Table Entry (CN) ^[11]
	0x01	Write Table Entry (C)
	0x02	Reset All Table Entries (V)
Session Context Manipulation	0x08	Clear Session Contexts by Session ID (C)
	0x09	Clear Session Contexts by Rule ID (V)
	0x0c	Clear All Session Contexts (V)
Attribute Manipulation	0x10	Get Attribute (VN)
	0x11	Set Attribute (V)
Debugging	0x1f	Error Indication (N)
Reply Types	0x80-0xff	Reserved for replies (most-significant bit)

5.2.6.3.9.3 TID Information

The read, write and reset messages of PMP require the specification of table identifier (TID), index, and specific entry data sizes. The table below specifies the details of these values.

Table Identifier, Index, and Data Sizes

Table Id	Table Id Name	Start Index	End Index	Entry Data Size
0	pmp_one_byte_trigger_table_id_e	0	0	32
1	pmp_two_byte_trigger_table_id_e	0	511	8
2	pmp_variable_trigger_table_id_e	0	4095	8
3	pmp_confidence_table_id_e	0	18944	4
4	pmp_confirmation_table_id_e	0	65535	128
5	pmp_userDefinedGroupTableId_e	0	0	256
6	pmp_equivalence_table_id_e	0	0	256
7	pmp_session_context_table_id_e	0	1073741823 (B)	32
8	pmp_special_trigger_table_id_e	0	0	32
(A) The number of confirmation entries is configurable.				
(B) The number of session context entries is configurable and relative to the number of sessions and context size.				

[11] Designates commands that map one to one with the PMI commands supported by PME hardware.

5.2.7.1 DCE Drivers Release Notes

Description

This document describes DCE software for the DCE hardware block that is part of the QorIQ family of SoCs. This is a preliminary release of the DCE software. It is expected that the APIs will be updated for future releases.

Linux

The DCE driver software includes a Linux kernel driver. The driver provides a set of kernel level APIs.

The driver includes the following functionality:

DCE Kernel Driver Interface

The DCE kernel driver APIs provide a callback based interface to the DCE. The driver provides APIs to perform either stateless (chunk) based (de)compression or stateful (stream) based (de)compression. The driver internally co-ordinates commands to the DCE and corresponding results from the DCE. The chunk interface is meant for inline (de)compression where each DCE operation is on a complete and independent piece of information. The stream interface is designed to (de)compress many related pieces of information (e.g. a file).

DCE FLIB interface

The DCE FLIB interface provides a consistent interface to the CCSR registers, the memory defined DMA structures and to the `dce_flow` software object.

DCE Configuration interface

The DCE configuration interface is an encapsulation of the DCE CCSR register space and the global/error interrupt source. This is expected to be managed only by (and visible to) a control-plane operating system,

DCE User-space Interface

There is a debugfs interface available for device debugging. No other userspace interface is available. Debugfs provides easy access to DCE memory map registers space. See the *DPAA Reference Manual* for the “DCE Individual Register Memory Map” e.g.

```
0x000 DCE_CFG - DCE configuration
0x03C DCE_IDLE- DCE Idle status Register
0x3F8 DCE_IP_REV_1 - DCE IP Block Revision 1 register
```

Mount debugfs to explore DCE status:

```
mount -t debugfs none /sys/kernel/debug
root@t4240qds:/dev/shm# cat /sys/kernel/debug/dce/ccsrmem_addr
DCE register offset = 0x0
root@t4240qds:/dev/shm# cat /sys/kernel/debug/dce/ccsrmem_rw
DCE register offset = 0x0
value = 0x00000003 <-DCE configuration, x03= Enable. Block is operational, Frame Queues are
consumed.
root@t4240qds:/dev/shm# echo 0x03c > /sys/kernel/debug/dce/ccsrmem_addr
root@t4240qds:/dev/shm# cat /sys/kernel/debug/dce/ccsrmem_rw
DCE register offset = 0x3c
value = 0x00000001 <- DCE Idle status Register, 1 = idle
root@t4240qds:/dev/shm# echo 0x3f8 > /sys/kernel/debug/dce/ccsrmem_addr
root@t4240qds:/dev/shm# cat /sys/kernel/debug/dce/ccsrmem_rw
DCE register offset = 0x3f8
value = 0x0af00101 <-match default value of "0x0AF0_0101"
```

Functionality

Configuration

The DCE device is configured via device-tree nodes and by some compile-time options controlled via Linux's Kconfig system. See the "DCE Kernel Configure Options" section for more info.

Debugfs Interface

The DCE has a debugfs interface available to assist in device debugging. The code can be built either as a loadable module or statically.

Module Loading

The driver can be statically built or as a dynamically loadable module.

DCE Kernel Configure Options

Common Kernel Configure Options	Description
CONFIG_STAGING	Required in order to make "staging" drivers such as DCE available.
CONFIG_FSL_DCE	Required to build DCE support.
CONFIG_FSL_DCE_CONFIG	Compiles in dce device driver support.
CONFIG_FSL_DCE_DEBUGFS	Compiles in support for debugfs interface for the DCE.
CONFIG_FSL_DCE_TESTS	Compiles DCE test code.

Compile-time Configuration Options

The "Kernel Configure Options" above describe the compile-time configuration options for the kernel.

Source Files

Linux

Source Files	Description
drivers/staging/fsl_dce/fsl_dce_chunk.h	The DCE driver APIs for chunk based (de)compression
drivers/staging/fsl_dce/fsl_dce_stream.h	The DCE driver APIs for stream based (de)compression
drivers/staging/fsl_dce/flib/*.*	The DCE flib interface
drivers/staging/fsl_dce/flib/dce_regs.h	The DCE CCSR register macros. Used in conjunction with bitfield_macros.h macros.
drivers/staging/fsl_dce/flib/dce_defs.h	The DCE dma defined memory structures.
drivers/staging/fsl_dce/flib/dce_flow.h	Object which defines the transport mechanism with the DCE engine. This object encompasses the QMan frame queues required to communicate with the DCE. The chunk and stream object use the flow object as a base.
drivers/staging/fsl_dce/dce_debugfs.*	The DCE debugfs interface
drivers/staging/fsl_dce/tests/performance_simple/*.*	Test which demonstrates the DCE throughput performance using single input files. Refer to local README file for more details.

Build Procedure

The procedure is a standard SDK build.

Test Procedure

Refer to drivers/staging/fsl_dce/tests/performance_simple/README for detailed descriptions of sample DCE throughput performance test.

Known Bugs, Limitations, or Technical Issues

- The APIs have been tested in the context of the performance test applications.
- It is possible that in future releases additions and or modification to APIs may occur.

Supporting Documentation

1. T4240 QorIQ Advanced Multiprocessing Processor Reference Manual
2. Queue Manager, Buffer Manager API reference Manual

5.3 Enhanced Secured Digital Host Controller (eSDHC)

5.3.1 eSDHC Driver User Manual

Description

The enhanced SD Host Controller(eSDHC) provides an interface between the host system and MMC/SD cards. The eSDHC supports 1/4-bit data modes and bus clock frequency up to 50MHz

Module Loading

The eSDHC device driver support either kernel built-in or module.

U-boot Configuration

Runtime options

Env Variable	Env Description	Sub option	Option Description
hwconfig	Hardware configuration for u-boot	setenv hwconfig sdhc	Enable esdhc for the kernel

Kernel Configure Options

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
<pre> Device Drivers ---> <*> MMC/SD/SDIO card support ---> <*> MMC block device driver (8) Number of minors per block device [*] Use bounce buffer for simple hosts </pre>	Enables SD/MMC block device driver support

Table continues on the next page...

Table continued from the previous page...

Kernel Configure Tree View Options	Description
<pre> *** MMC/SD/SDIO Host Controller Drivers *** <*> Secure Digital Host Controller Interface support <*> SDHCI platform and OF driver helper [*] SDHCI OF support for the NXP eSDHC controller </pre>	Enables NXP eSDHC driver support

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_MMC	y/n	n	Enable SD/MMC bus protocol
CONFIG_MMC_BLOCK	y/n	y	Enable SD/MMC block device driver support
CONFIG_MMC_BLOCK_MINORS	integer	8	Number of minors per block device
CONFIG_MMC_BLOCK_BOUNCE	y/n	y	Enable continuous physical memory for transmit
CONFIG_MMC_SDHCI	y/n	y	Enable generic sdhc interface
CONFIG_MMC_SDHCI_PLTFM	y/n	y	Enable common helper function support for sdhci platform and OF drivers
CONFIG_MMC_SDHCI_OF_ESDHC	y/n	y	Enable NXP eSDHC support

Device Tree Binding

Property	Type	Status	Description
compatible	String	Required	Should be 'fsl,esdhc'
reg	integer	Required	Register map

Default node:

```

qoriq-esdhc-0.dtsi:
sdhc: sdhc@114000 {
    compatible = "fsl,esdhc";
    reg = <0x114000 0x1000>;
    interrupts = <48 2 0 0>;
    clock-frequency = <0>;
};

For special platform (T1040 as example):
#include/ "qoriq-esdhc-0.dtsi"
sdhc@114000 {
    compatible = "fsl,t1040-esdhc", "fsl,esdhc";
    fsl,iommu-parent = <&pamu0>;
    fsl,liodn-reg = <&guts 0x530>; /* eSDHCLIODNR */
    sdhci,auto-cmd12;
    rcpm-wakeup = <&rcpm 0x00000080>;
};

```

NOTE: For different platform, the compatilbe can be different.
And the property "sdhci, auto-cmd12" is option.

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/mmc/host/sdhci.c	Linux SDHCI driver support
drivers/mmc/host/sdhci-pltfm.c	Linux SDHCI platform devices support driver
drivers/mmc/host/sdhci-of-esdhc.c	Linux eSDHC driver

User Space Application

The following applications will be used during functional or performance testing. Please refer to the SDK UM document for the detailed build procedure.

Command Name	Description	Package Name
iozone	IOzone is a filesystem benchmark tool. The benchmark generates and measures a variety of file operations. The benchmark tests file I/O performance for the following operations: Read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read, pread ,mmap, aio_read, aio_write.	iozone

Verification in U-boot

The u-boot log:

```
=> mmcinfo
Device: FSL_ESDHC
Manufacturer ID: 3
OEM: 5344
Name: SD02G
Tran Speed: 25000000
Rd Block Len: 512
SD version 2.0
High Capacity: No
Capacity: 2032664576
Bus Width: 4-bit
=> mmc read 0 10000 0 1
MMC read: dev # 0, block # 0, count 1 ... 1 blocks read: OK
=> mmc part 0
Partition Map for MMC device 0 -- Partition Type: DOS
Partition      Start Sector      Num Sectors      Type
-----
1                16                3970032          b
```

Verification in Linux

Add environment value

```
=> setenv hwconfig sdhc
```

The booting log

```
.....  
sdhci: Secure Digital Host Controller Interface driver  
sdhci: Copyright(c) Pierre Ossman  
mmc0: SDHCI controller on ffe2e000.sdhci-of [ffe2e000.sdhci-of] using PIO  
.....  
mmc0: new SD card at address 87e2  
mmcblk0: mmc0:87e2 SD02G 1.89 GiB  
mmcblk0: p1
```

Check the disk

```
~ # fdisk -l /dev/mmcblk0  
  
disk /dev/mmcblk0: 2032 MB, 2032664576 bytes  
  
63 heads, 62 sectors/track, 1016 cylinders  
  
Units = cylinders of 3906 * 512 = 1999872 bytes  
  
Device Boot      Start         End      Blocks   Id System  
  
/dev/mmcblk0p1    1           1016     1984217    b Win95 FAT32  ~ #
```

Mount the file system and operate the card.

```
~ #  
  
~ # mkdir /mnt/sd  
  
~ # mount -t vfat /dev/mmcblk0p1 /mnt/sd  
  
~ # ls /mnt/sd/  
  
vim  
  
~ # cp /bin/busybox /mnt/sd  
  
~ # ls /mnt/sd  
  
busybox vim  
  
~ # umount /mnt/sd  
  
~ # mount -t vfat /dev/mmcblk0p1 /mnt/sd
```

```
~ # ls /mnt/sd  
  
busybox  vim  
  
~ #
```

Benchmarking

```
~ #  
  
~ # # iozone -Rab ./iosdresult/result -i 0 -i 1 -f test -n  
  
512M -g 1G -r 64K  
  
Iozone: Performance Test of File I/O  
  
Version $Revision: 3.263 $  
  
Compiled for 32 bit mode.  
  
Build: linux-arm  
  
  
Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins  
Al Slater, Scott Rhine, Mike Wisner, Ken Goss  
Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,  
Randy Dunlap, Mark Montague, Dan Million,  
Jean-Marc Zucconi, Jeff Blomberg,  
Erik Habbinga, Kris Strecker, Walter Wong.  
  
Run began: Wed Feb 16 20:33:04 2011  
  
Excel chart generation enabled  
  
Auto Mode  
  
Using minimum file size of 524288 kilobytes.  
  
Using maximum file size of 1048576 kilobytes.  
  
Record Size 64 KB  
  
Command line used: iozone -Rab ./iosdresult/result -i 0 -i 1 -f test -n 512M -g 1G -r 64K  
  
Output is in Kbytes/sec
```

```

Time Resolution = 0.000005 seconds.

Processor cache size set to 1024 Kbytes.

Processor cache line size set to 32 bytes.

File stride size set to 17 * record size.

random random bkwd record stride

KB reclen write rewrite read reread read write read rewrite read fwrite frewrite fread freread

524288 64 7040 7253 371022 372079

1048576 64 6537 6566 9857 10203

```

Known Bugs, Limitations, or Technical Issues

1. Call trace when run "iozone" to test SDCARD performace on some platforms

workaround:increase the timeout value (in kernel configuration) and decrease the dirty_ratio in proc file system.

1) menuconfig:

Kernel hacking

(xxx) Default timeout for hung task detection (in seconds)

Note: the xxx may be 400 seconds or greater

2) modify 'proce file system':

```
echo xx > /proc/sys/vm/dirty_ratio
```

```
echo xx > /proc/sys/vm/dirty_background_ratio
```

Note: the xx may be 10 or 5, which meas 10% or 5%, the default is 20%.

2. The platform whose card is required to work on a special transfer mode which is not FS or HS mode needs a special rcw. (e.g. rcw_66_15_1800MHz_emmc_ddr.rcw is for t2080qds eMMC DDR mode. Because of pin multiplexing with SPI, SPI would not work when eMMC card works on DDR mode)

5.4 Ethernet

5.4.1 Linux Ethernet Driver for DPAA 1.x Family

5.4.1.1 Linux DPAA 1.x Ethernet Primer

Understanding the high-level concepts of the Linux driver

5.4.1.1.1 Introduction

An overview of the DPAA-Ethernet network driver, in the more generic context of Linux device drivers.

The primary concepts of the DPAA-Ethernet driver architecture are presented without going into such intricate details as device-tree configuration or code structure. The current document is neither a Linux Device Drivers tutorial, nor a replacement to the **Linux Ethernet Driver** document in the SDK, but a quick start guide which provides context for users.

The DPAA-Ethernet driver software shipped with the standard QorIQ Linux SDK is described.

5.4.1.1.2 Intended Use Cases

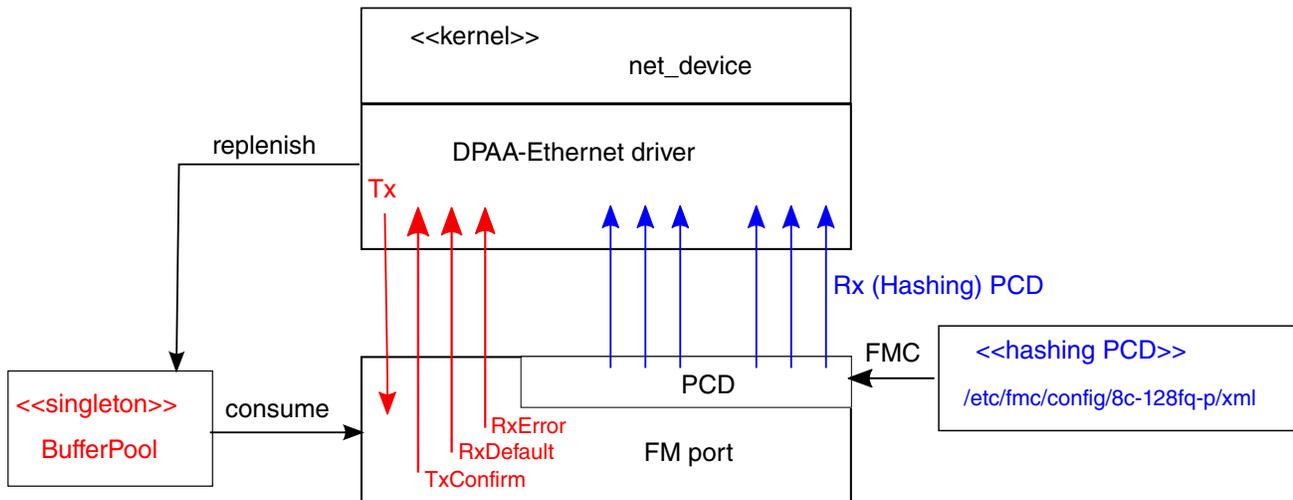
This chapter presents a high-level view of the standard use-cases (based on the QorIQ Linux SDK requirements) that the DPAA-Ethernet driver currently supports.

5.4.1.1.2.1 Private Net Devices

This is the primary driver and the one which currently delivers the best performance. Characteristics of this driver are:

- The private driver is a multiqueue driver - it uses 1 TX queue per CPU
- All private interfaces use a single BPID - usually dynamically allocated
- The FQIDs for the common types of queues - RX, TX, RX Error, TX Error, TX Confirm - are dynamically allocated
- The Hashing/PCD frame queues are hardcoded in the device tree. The private driver imports the PCD configuration from device tree at startup
- The above resources are allocated and visible only to the private driver

In the case of private interfaces, all network traffic takes place between the Linux kernel and the physical FMan port private to that partition.



There is one Buffer Pool used by all driver instances from this Linux partition. The buffer lifecycle is entirely between the DPA-Ethernet driver and the FMan port and all buffers in the pool are dynamically allocated by the driver. The BPID itself can be static, although this is not encouraged.

In the standard configuration, each driver instance dynamically allocates a private set of default Rx and Tx FQs (in red).

Additionally, there are 128 "hashing PCD FQs" (in blue), statically allocated for user's convenience. A standard FMC configuration file is shipped with the SDK enabling the "hashing PCD FQ's".

Figure 84. Network Traffic Between the Linux Kernel and the Physical FMan Port

5.4.1.1.2.2 Shared-MAC Net Devices

A shared-MAC device is one that can be used from two (or potentially more) Linux and/or USDPAA partitions. A shared-MAC encompasses one of the following partitioning scenarios:

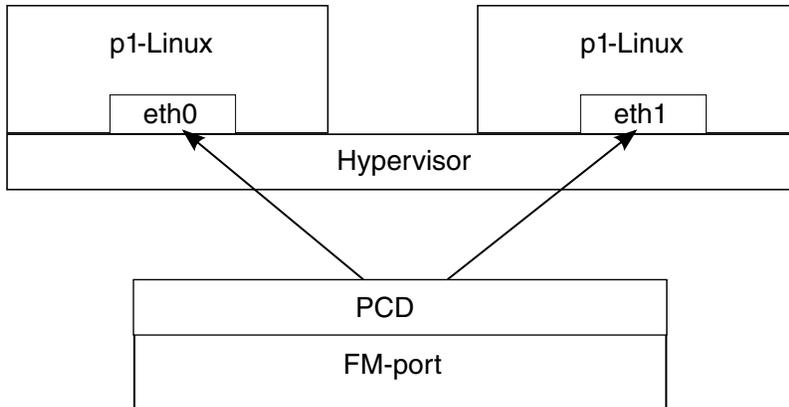


Figure 85. Two (or more) Linux separate partitions, under control of the TOPAZ hypervisor.

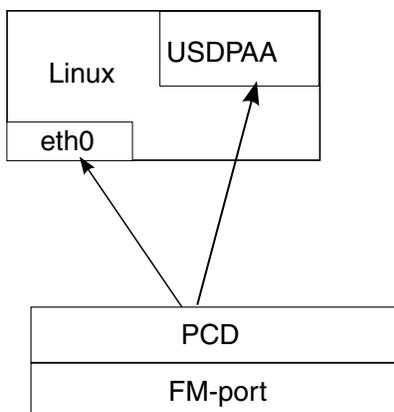


Figure 86. One Linux and one USDPAAs running in the same partition

Points to emphasize

- In all cases, there is exactly one physical MAC. Linux always “owns” the MAC, which it initializes.
- A shared-MAC between two Linux interfaces (say, eth0 and eth1) from the same partition is **not supported** unless in a hypervised scenario (and has never been intended as a use-case, anyway).

Configuration and ownership of a shared-MAC is asymmetric and driven by a number of hardware and software constraints.

The following constraints and design assumptions apply to Buffer Pools in a shared-MAC device:

- FMan v2 (not supporting virtual storage profiles) picks the buffer to store the Rx frame based solely on the ingress frame size, regardless of the result of PCD.
- In shared-MAC devices used by Linux and USDPAAs, to avoid the need for the USDPAAs fastpath to remap the ingress buffers, the same Buffer Pool is shared between Linux and USDPAAs.
- In shared-MAC devices used by two Linux partitions (no USDPAAs involved), it is necessary that the Linux guests have the same true-physical to guest-physical mappings, in order for FMan-DMA to work seamlessly regardless of the destination partition. Moreover, that presumes that the buffer space seen by the two partitions is identical, which comes down to the Buffer Pools being physically shared between the two Linux partitions.
- Sharing a Buffer Pool between two or several Linux and/or USDPAAs partitions requires that the BPID be statically defined (identically hard-coded) in the `.dts` configurations of all partitions.
- The convention between Linux and USDPAAs is that USDPAAs initializes and seeds the shared Buffer Pool. Linux dynamically remaps ingress buffers received on a shared-MAC, copies them into a buffer dynamically allocated in its own memory space, then releases the ingress buffer back into the shared Buffer Pool.

- In the case of a shared-MAC between two Linux partitions (in a hypervised scenario), only one partition will initialize and seed the shared Buffer Pool. That partition is determined by means of a special property in the `hv.dts` partition configuration. Refer to **Linux Ethernet Driver - Buffer Pools** for details.

The following constraints and design assumptions apply to Frame Queues in a shared-MAC device as seen by the Linux DPAA-Ethernet (i.e. not USDPAA) driver:

- Rx traffic is driven to either partition via a PCD configuration applied on the shared physical port. This means that, for Linux partitions using a shared-MAC, at least one Rx Frame Queue has to be statically declared in the `.dts`. That may be the Rx Default Frame Queue and/or the (automatically initialized) 128 core-affine **Hashing PCD Frame Queues**.
- In the Linux-Linux shared-MAC scenario, all Tx Frame Queues must be statically specified and be the same in both partitions. The reasoning is explained in the **Linux Ethernet Driver - Virtual/Shared Controller** document. .
- In the Linux-USDPAA shared-MAC scenarios, the Linux partition will always initialize its own Tx Frame Queues, be they dynamically or statically allocated. It is up to the USDPAA application to choose its own Tx FQIDs.
- The Tx Confirm Frame Queues are always used in shared-MAC scenarios; all egress buffers are confirmed. See **Shared MAC: Tx** packet lifecycle for details.

5.4.1.1.2.3 MAC-less Net Devices

A MAC-less device, also called a Virtual Controller in the **Linux Ethernet Driver - Virtual/Shared Controller**, appears to Linux as a regular net device (Ethernet interface). It is “virtual” in the sense that it does not have a physical FMan port (be it an online or an OH port) underlying, but that is transparent to the Linux kernel and userspace applications – only the DPAA-Ethernet driver is aware of the difference.

The figure below shows the typical MAC-less use-case is as a communication device between two Linux partitions:

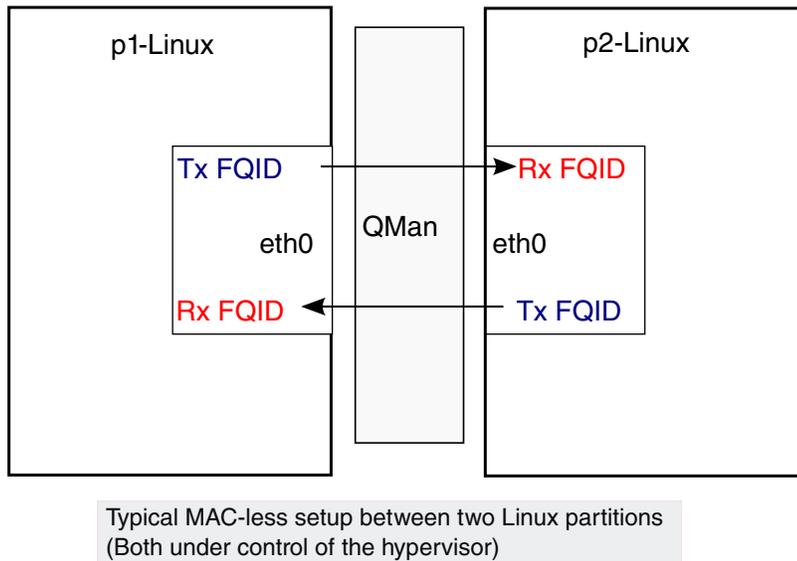


Figure 87. Communication device between two Linux partitions

... or between Linux and USDPAA:

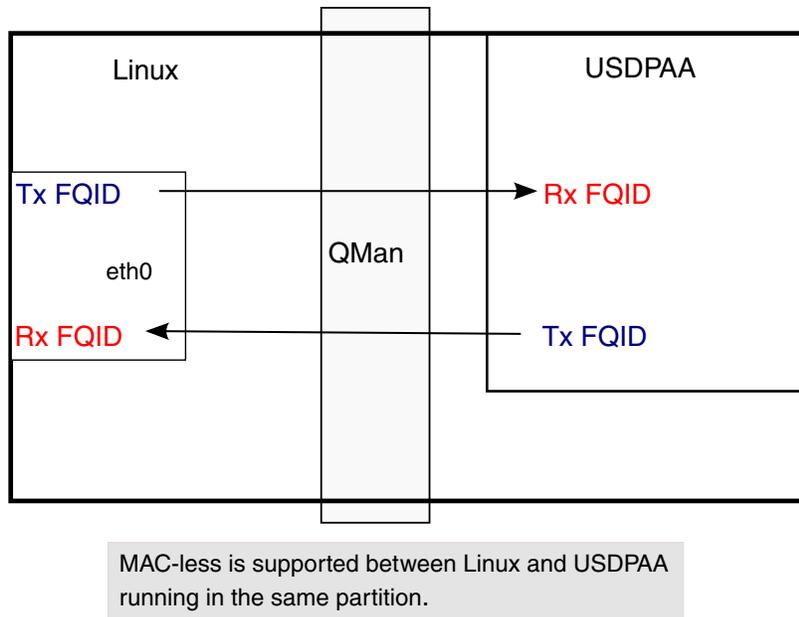


Figure 88. Communication device between one Linux and one USDPAA in the same partition

From the DPAA resource configuration standpoint, a MAC-less net device is very similar to a **Shared MAC: Tx**. The same constraints apply to Buffer Pools and Frame Queues as in the case of a shared-MAC, with one notable difference:

- Because the Tx FQs of a MAC-less device always sink into another partition (or USDPAA) instead of a physical FMan port, there is the convention that the DPAA-Ethernet driver of a MAC-less node **only initializes its Rx FQs**. In other words, each partition initializes its own Rx FQs, because it has to bind local dequeue callbacks to them.

In a MAC-less setup, one endpoint's Tx FQIDs are the other endpoint's Rx FQIDs and vice versa.

While the previous diagrams have shown typical MAC-less use-cases, one can design more complex scenarios by interposing various processing blocks between the two MAC-less endpoints. Refer to the **Extended Use Cases** chapter for a discussion.

5.4.1.1.2.4 Choosing the Current Use Case

The following activity diagram describes the configuration selection logic in the DPAA-Ethernet driver, based on properties found in the `.dts` specification:

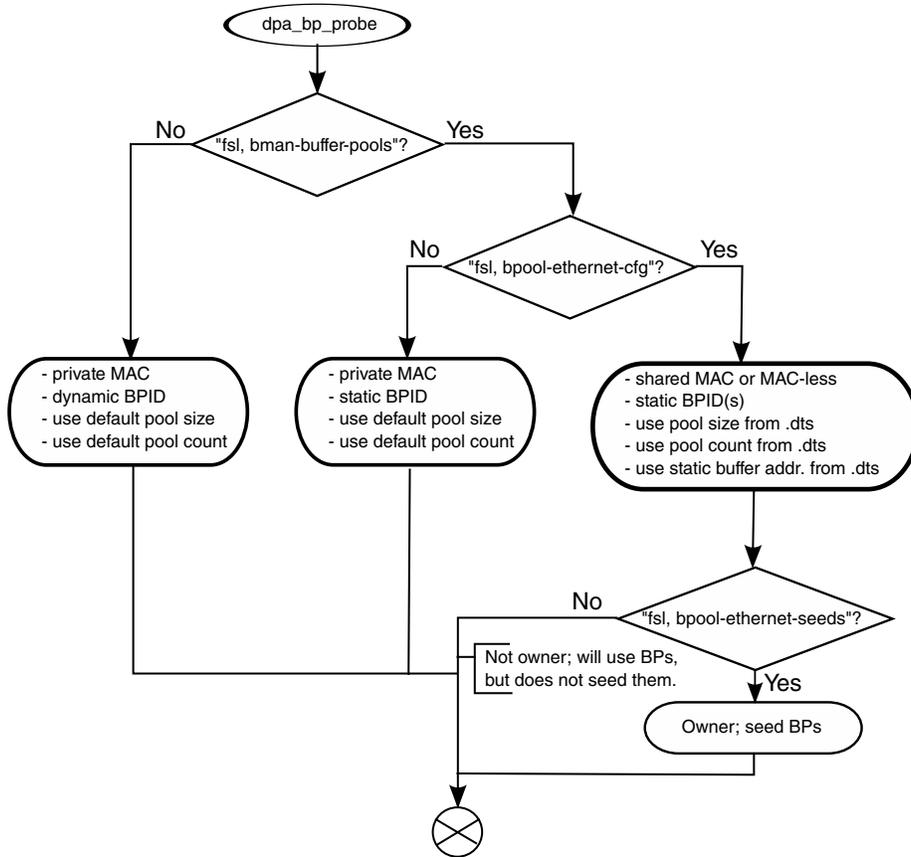


Figure 89. Configuration Selection Logic

5.4.1.1.3 The DPAA-Eth View of the World

This section presents the primary concepts behind the DPAA-Ethernet driver design.

As a Linux driver, one of DPAA-Ethernet driver's main goals is proper integration with the Linux kernel ecosystem. As a hardware device driver, the DPAA-Ethernet driver integrates functions of several DPAA IP blocks, within the scope of the defined/supported use cases.

5.4.1.1.3.1 The Linux Kernel API's

The DPAA-Ethernet drivers interface with the Linux kernel via the latter's networking stack APIs. This is a strong requirement, mandated by the integration with the Linux kernel.

Another type of interaction with the kernel code is at boot-time, via the Open-Firmware API. That API is used to parse the PowerPC platform device tree and discover the hardware modules that need to be configured. In particular, the DPAA-Ethernet drivers use the platform device tree to discover:

- What net devices to probe and what type of hardware is underlying those devices;
- Which DPAA resources are involved: FQIDs, BPIDs, CGRIDs, FMan port IDs.

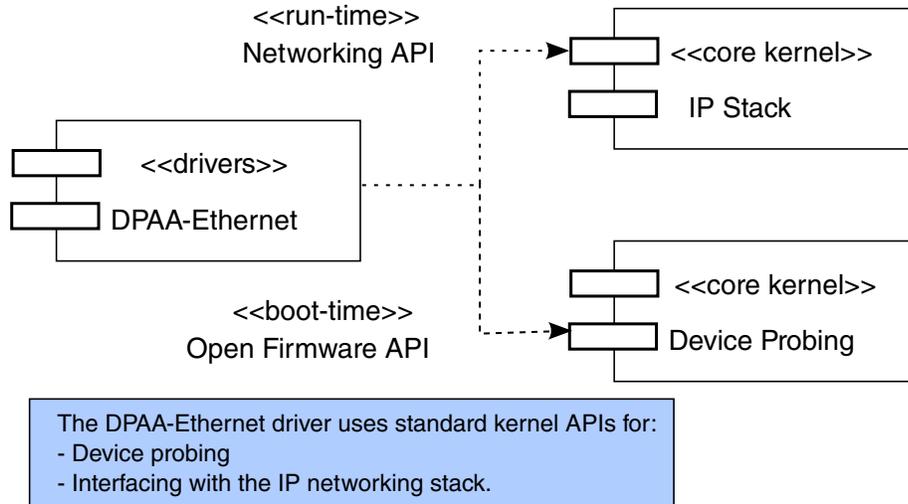


Figure 90. Platform Device Tree

Generally, we prefer drivers configurations to be dynamic and transparent to the rest of the system. Among the benefits of dynamic resource allocations, we count:

- Portability of the drivers across multiple QorIQ platforms;
- Seamless support of platform changes (e.g. via booting with different RCWs);
- Seamless support of multiple partitions under the control of a hypervisor;
- Cohabitation with other DPAA drivers (e.g. a SEC driver) in the SDK.

In certain scenarios, however, configurations are statically defined and are extracted directly from the .dts specification. This is for instance the case of shared MAC and MAC-less devices.

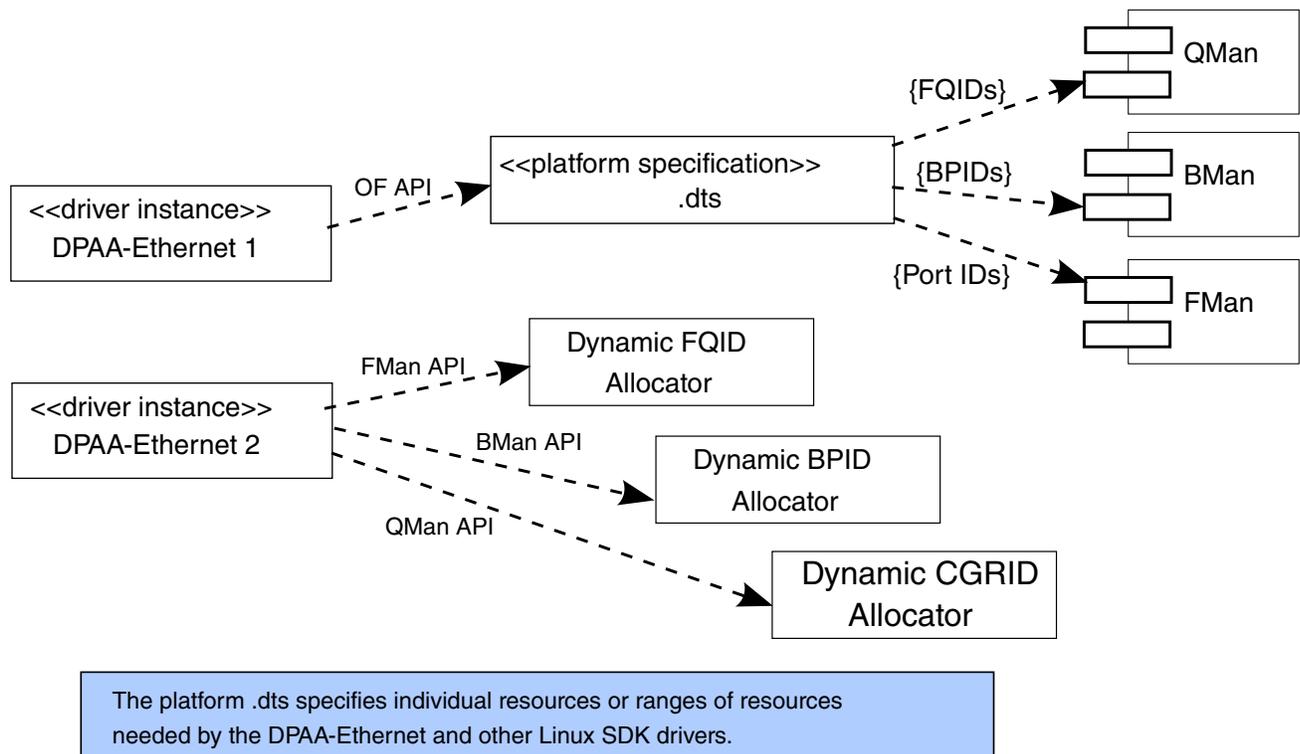


Figure 91. Shared MAC and MAC-less Devices

5.4.1.1.3.2 The Driver's Building Blocks

This chapter presents the main structures and data entities with which the DPAA-Ethernet driver operates.

The driver's building blocks are part of the interfaces of the driver with the relating components, i.e.:

- The kernel's IP stack;
- The DPAA hardware blocks and their drivers.

The DPAA Ethernet driver is actually a series of drivers, each tailored for a specific use case:

- Private driver - maps Linux kernel network interfaces to physical ports
- Shared driver - gives simultaneous control of a physical port to the Linux kernel and user space applications
- Macless driver - although it appears to the Linux kernel as a regular net device, it does not control a physical port. It is typically used as a virtual communication device between two Linux partitions
- Proxy driver - an interface through which the Linux kernel configures a physical port, only to pass its control to user space applications
- Offline port driver - controls Offline Parsing / Host Command port (OH)

Each of the above drivers has its own code base and implements the Linux kernel API by providing its own callback functions.

5.4.1.1.3.2.1 Net Devices

A net device (`struct net_device` in C representation) is the fundamental structure of any Linux network device driver.

A net device describes a (physical or virtual) device capable of sending and receiving packets over a (virtual or physical) network. All incoming and outgoing traffic is accounted and processed on behalf of the net device it comes or goes on.

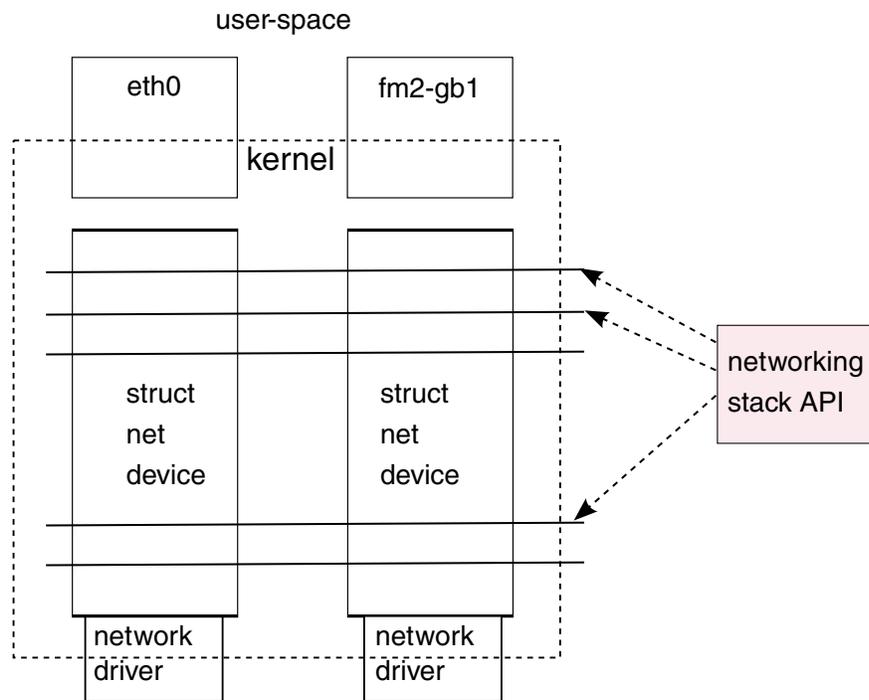
Each supported type of net device has its own kernel driver. If there are several such devices present in a system, there will be as many device driver instances.

A net device is accessible to the Linux user via the standard tools, such as 'ifconfig' or 'ethtool'.

Not all net devices have real underlying hardware; tunnel endpoints, for examples, are represented by net devices but are not directly backed by hardware. Same holds for drivers such as "bonding" or "dummy".

It is worth emphasizing, however, that **every** Linux interface is represented by a net device. This is a fundamental design aspect of all Linux networking drivers, including DPAA-Ethernet. One can describe the Linux IP stack as being a **netdev-centric** construction. Nearly all of the kernel networking APIs receive a `struct net_device` as a parameter. The `net_device` structure is the handle through which the driver and the network stack communicate.

The following diagram illustrates what has just been described:



The kernel networking APIs are generally netdevice-centric.
A network driver interfaces with the IP stack on behalf of a net device

Figure 92. Every Linux Interface is Represented by a Net Device

5.4.1.1.3.2.2 Frame Queues

The Frame Queue is one of the fundamental concepts of DPAA. In the case of DPAA-Ethernet, it is the main interface between the network driver and the hardware blocks.

Ingress frames received by the DPAA-Ethernet driver on one of the Frame Queues it is servicing are sent to the IP stack on behalf of the net device structure that the driver is associated with. Conversely, outgoing frames coming from the IP stack into the driver are enqueued to one of the egress Frame Queues.

NOTE

Depending on its configuration (see usecase), the DPAA-Ethernet driver may initialize some of its Frame Queues and make assumptions over what entity initializes the others.

5.4.1.1.3.2.3 Buffer Pools

Buffer pool configuration is another fundamental part of the DPAA-Ethernet driver design.

Unlike the Frame Queue utilization – which is more flexible – the Buffer Pool utilization is conditioned by several design assumptions:

- The source and ownership of the ingress frame buffers are presumed by the DPAA-Ethernet driver. The exact allocation logic depends on the driver configuration (see usecase), but in all cases the driver makes hard assumptions on how the buffers were allocated.

For instance, the “private MAC” driver configuration seeds the Buffer Pools at predefined checkpoints on the Rx path. There are also buffer utilization counters maintained by the driver, which influence the buffer allocation logic.

The “shared MAC” and “MAC-less” driver configurations only work with the Buffer Pools hard-coded in the platform .dts.

- The layout of incoming frames is also presumed by the driver. Depending on its configuration (see usecase), the driver expects the frame layout to conform to its own allocation logic. The actual buffer layout is outside the scope of this document and should not be assumed upon by driver users.
- The existence of a static Buffer Pool configuration in the platform `.dts` determines the driver configuration. The DPAA-Ethernet driver currently assumes that, if a static Buffer Pool is configured in its device tree node (as number/size/address of buffers), then it is describing a “shared MAC” or a “MAC-less” configuration.

5.4.1.1.4 DPAA Resources Initialization

The rationale behind the “what”s, “why”s and “how”s of DPAA resource initializations made by the DPAA-Ethernet driver are presented. This description does not go into the full detail of driver configuration (please refer to the **Linux Ethernet Driver** for that level of detail).

5.4.1.1.4.1 What, Why and How Resources are Initialized

DPAA resources initialized by the various configurations of the DPAA-Ethernet driver are:

- FQs and FQIDs (where static config applies);
- BPs and BPIDs (where static config applies);
- Buffers (not quite “DPAA” resources, rather “system” resources);;
- CGRs (CGRIDs are always dynamic);
- FMan’s online ports (note: the offline ports are configured by a different driver than DPAA-Ethernet).

Frame Queues and Buffer Pools have been covered at length in the previous chapters of this document. CGRs are of lesser interest from the initialization viewpoint.

FMan’s online ports are initially probed by the FMan Driver (FMD) and later in the boot process they are configured by the DPAA-Ethernet driver instances according to the specifications in the `.dts`.

5.4.1.1.4.2 Hashing/PCD Frame Queues

Among the Frame Queues initialized by the DPAA-Ethernet driver, there is a predefined set of 128 core-affined Rx FQs, automatically initialized by the driver for user’s convenience. They are there because most performance-enhanced setups must use a PCD configuration; to that end, the standard QorIQ Linux SDK provides a “hashing PCDs” configuration that can be applied by the user via the FMC tool. Since FMC does not support dynamic FQID specification in its `.xml` configuration files, the “hashing PCD” Frame Queues also have static, hard-coded FQIDs.

Furthermore, apart from the core-affined Rx FQs, there is another set of 128 core-affined Rx FQs, which have a higher priority than the former. They are named throughout this documentation “Rx PCD High Priority Frame Queues”. Likewise, the queues in this set are also core-affined and have static, hard-coded FQIDs.

For details about the “hashing PCD” Frame Queues and the Rx PCD High Priority Frame Queues, refer to the **Linux Ethernet Driver - Core Affined Queues**.

5.4.1.1.5 The (Simplified) Life of a Packet

This chapter presents a packet’s lifecycle in various configurations of the DPAA-Ethernet driver.

5.4.1.1.5.1 Private Net Device: Tx

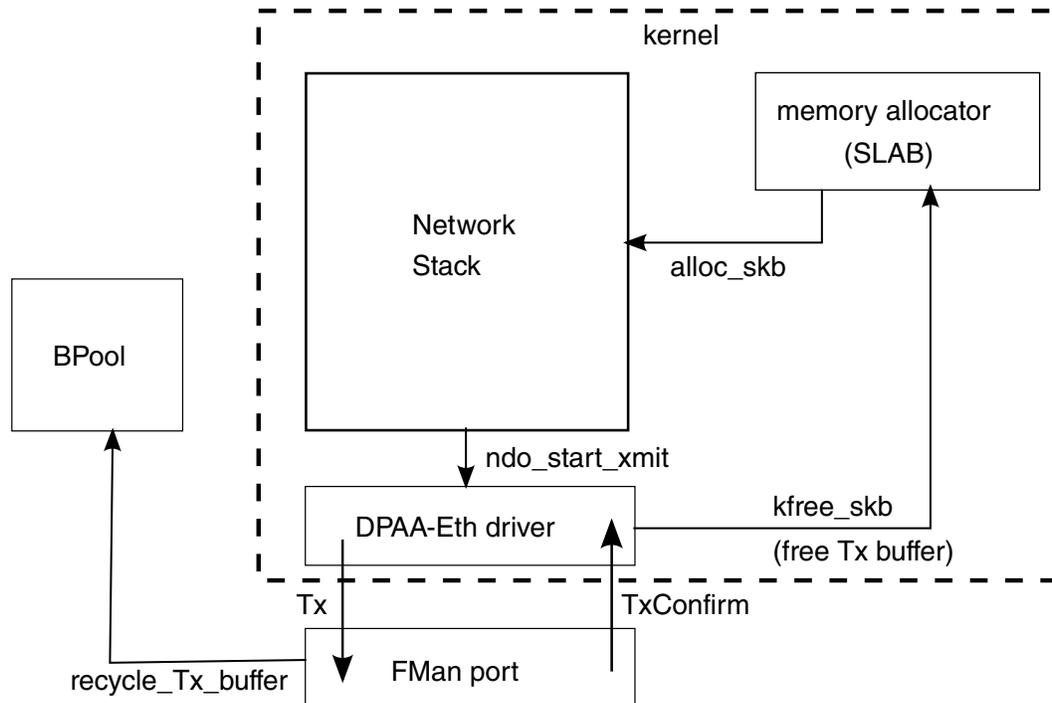


Figure 93. DPAA-Ethernet driver enqueues the packet to the FMan port

Arrows in the diagram above represent the direction of the buffer/packet flow.

A packet on the egress path is allocated by the network stack using the kernel's standard memory allocator. The DPAA-Ethernet driver enqueues the packet to the FMan port with an indication to recycle the buffer if possible. If recycling is not possible, the DPAA-Ethernet driver itself frees the buffer memory back to the kernel's allocator, when Tx delivery is confirmed by FMan.

5.4.1.1.5.2 Private Net Device: Rx

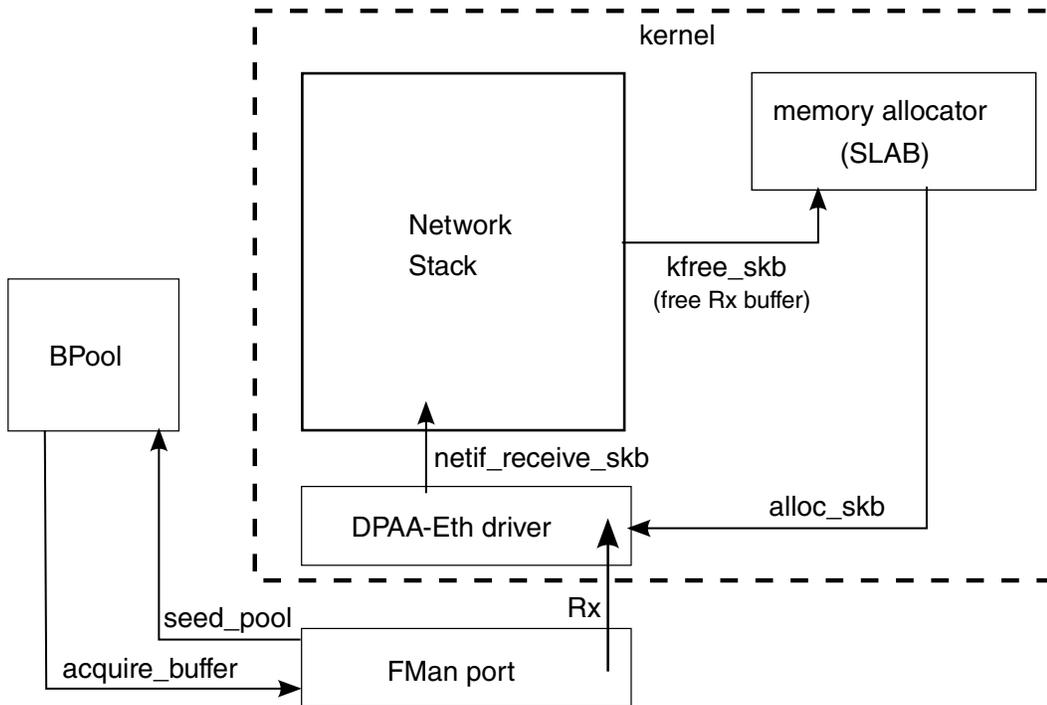


Figure 94. Buffers on the Ingress Path

Buffers on the ingress path are acquired by FMan directly from a Buffer Pool which was seeded by the DPAA-Ethernet driver. Buffer layout is important to the driver, which assumes ownership on the BP. Arrows in the diagram above represent the direction of the buffer/packet flow.

5.4.1.1.5.3 Shared MAC: Tx

The following diagram presents the lifecycle of an egress buffer/packet in the case of a shared-MAC device used by two Linux partitions (under the control of the Hypervisor, not shown in the picture):

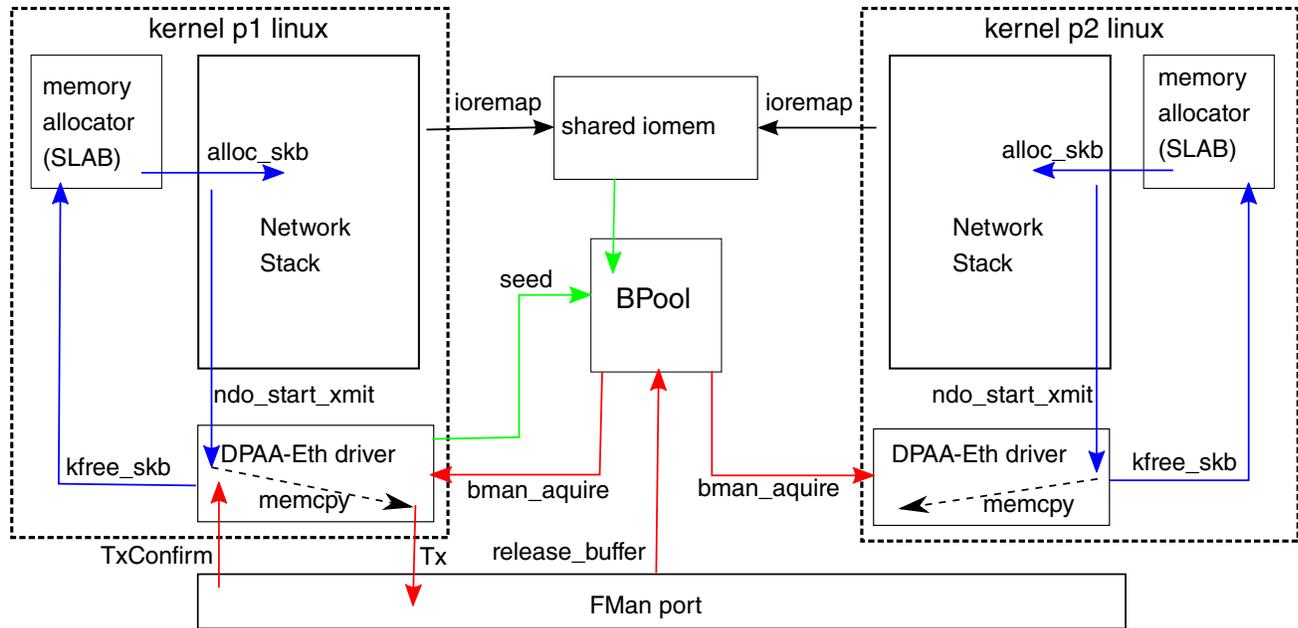


Figure 95. A Shared-MAC Device Used by Two Linux Partitions

There are essentially two buffer circuits in this scenario:

- The in-stack socket buffer lifecycle (colored in blue);
- The bpool buffer lifecycle (colored in red).

Socket buffers are dynamically allocated from the kernel memory. The DPAA-Ethernet driver memcopies them in buffers acquired from the Buffer Pool, then releases the socket buffers back into the kernel memory.

The shared Buffer Pool is seeded by one (and only one) of the Linux partitions, and is never again replenished by the software. Memory used for seeding the shared Buffer Pool is statically defined in the guest `.dts` configuration and is presented to the kernel as device memory (not as physical memory). Both DPAA-Ethernet drivers statically ioremap the entire shared Buffer Pool space (as per the static configuration in the `.dts`) at probe time. No run-time mapping/unmapping is effected afterwards.

After transmission, FMan confirms the frame and the Linux DPAA-Ethernet driver releases the buffers back into the Shared Buffer Pool.

Similarly to the above, the next diagram the lifecycle of an egress buffer/packet in the case of a shared-MAC device used by Linux and USDPAA:

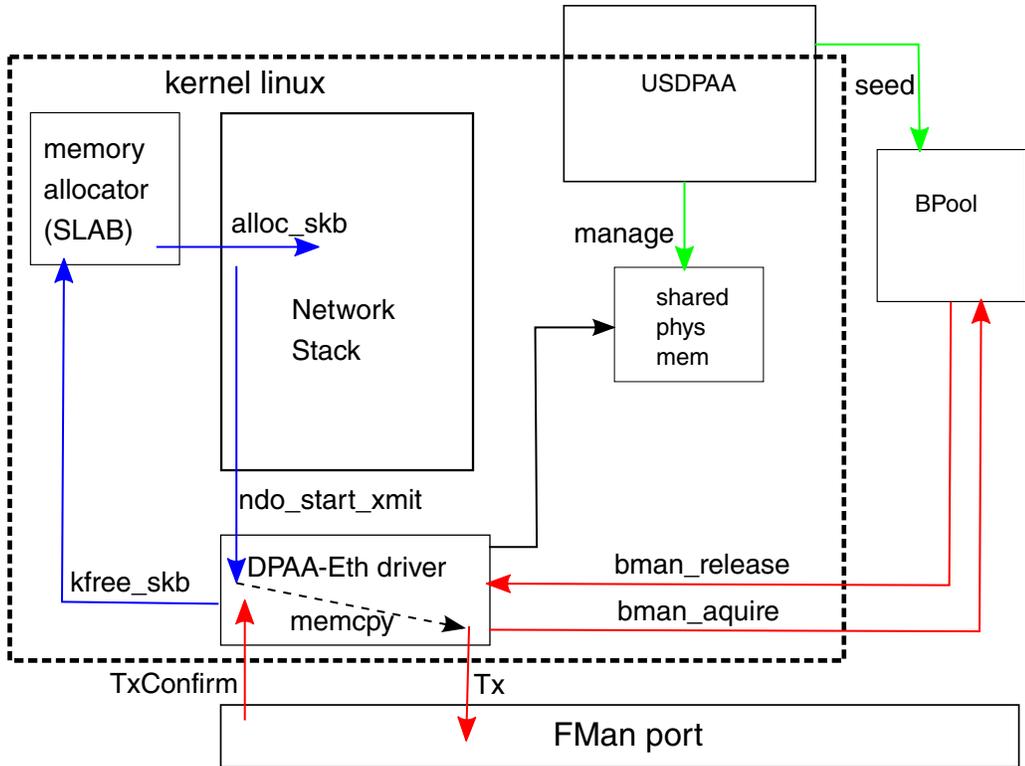


Figure 96. Shared-MAC Device Used by Linux and USDPAAs

5.4.1.1.5.3.1 Shared MAC: Rx

The diagram below presents the lifecycle of a buffer/packet in the case of a shared-MAC device used by Linux and USDPAAs running in the same partition.

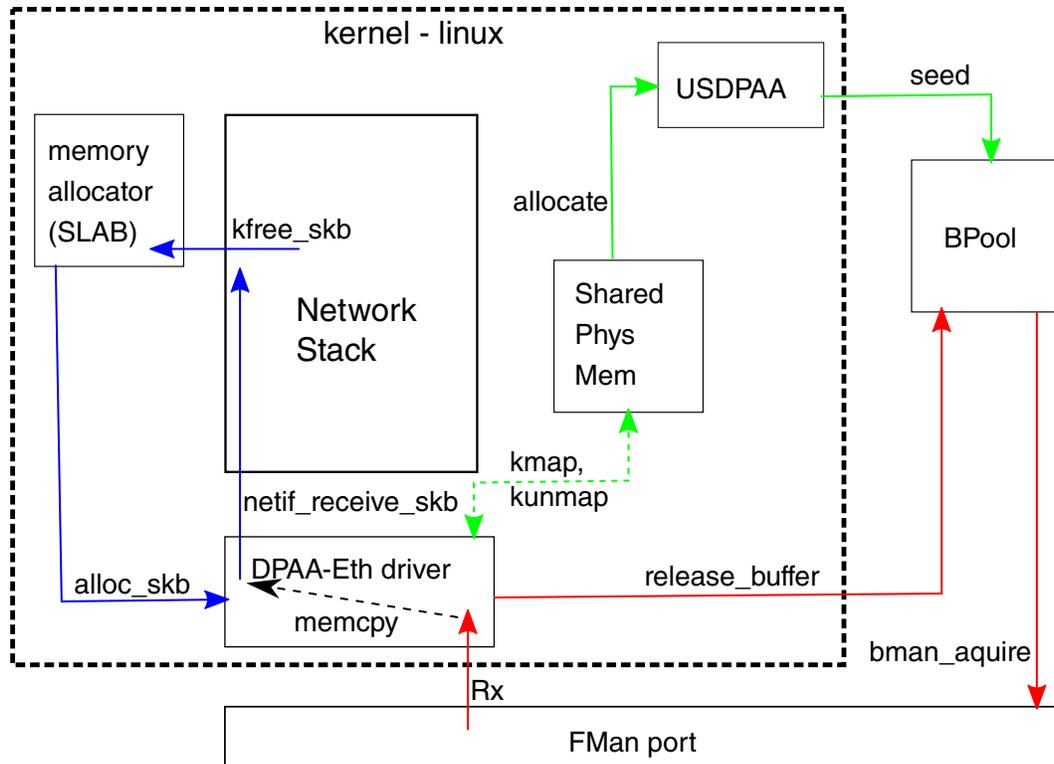


Figure 97. Shared-MAC Device Used by Linux and USDPAA

The Linux driver is the one to initialize the shared FMan port, but it is the USDPAA partition that allocates and seeds the buffers in the Shared Buffer Pool. To that end, USDPAA uses a block of physical memory reserved at boot time, such that the kernel can still map it, but not allocate from it (via either the page allocator or the object cache allocator).

Each incoming frame is dynamically mapped by the DPAA-Ethernet driver. As in the case of shared-MAC Tx, a memcopy is involved in the driver, from each incoming frame into a newly allocated socket buffer (`sk_buff`). The DPAA-Ethernet driver subsequently unmaps the buffer and releases it back into the shared Buffer Pool, for later reuse.

One should note the invariant that, as in all MAC-less and shared-MAC scenarios, the Shared Buffer Pool is initialized and seeded exactly once, at init-time; at run-time, the total number of in-flight buffers and buffers available in the pool is constant. One must preallocate a large enough number of buffers in the Shared Buffer Pool, such that to prevent its run-time depletion.

NOTE

A similar diagram can be obtained in the case of a shared-MAC between two different Linux partitions, with the principal change that the shared memory from which the Buffer Pool is seeded is seen as `iomem` (rather than physical memory), and is statically `ioremaped` all at once, at boot-time.

5.4.1.1.5.3.2 MAC-less Net Devices: Tx

The following diagram presents the lifecycle of a buffer/packet in the case of a MAC-less device used for communication from Linux to USDPAA:

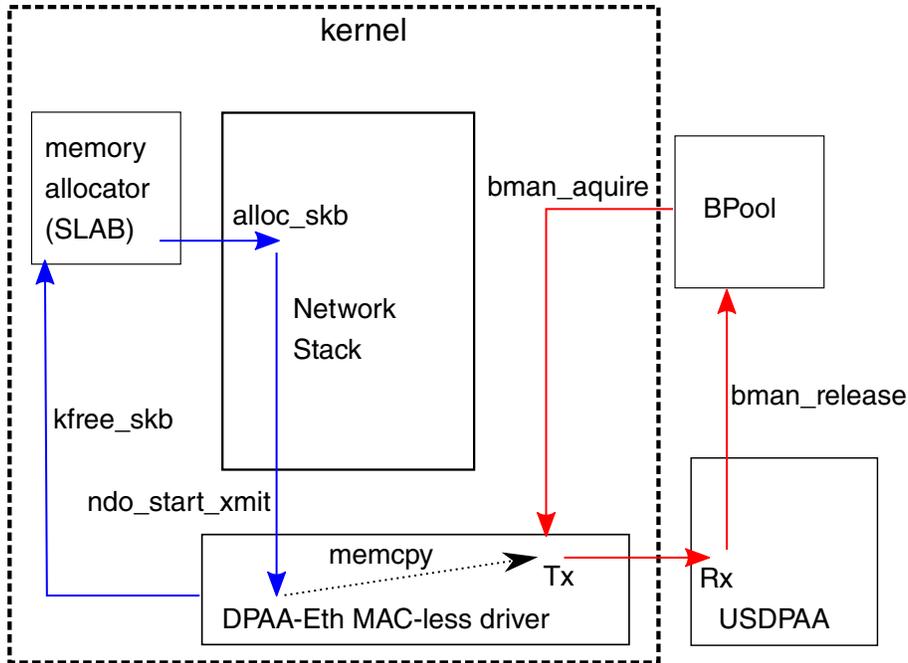


Figure 98. MAC-less Device Used for Communication from Linux to USDPAA

5.4.1.1.5.3.3 MAC-less Net Devices: Rx

The following diagram presents the lifecycle of a buffer/packet in the case of a MAC-less device used for communication from USDPAA to Linux:

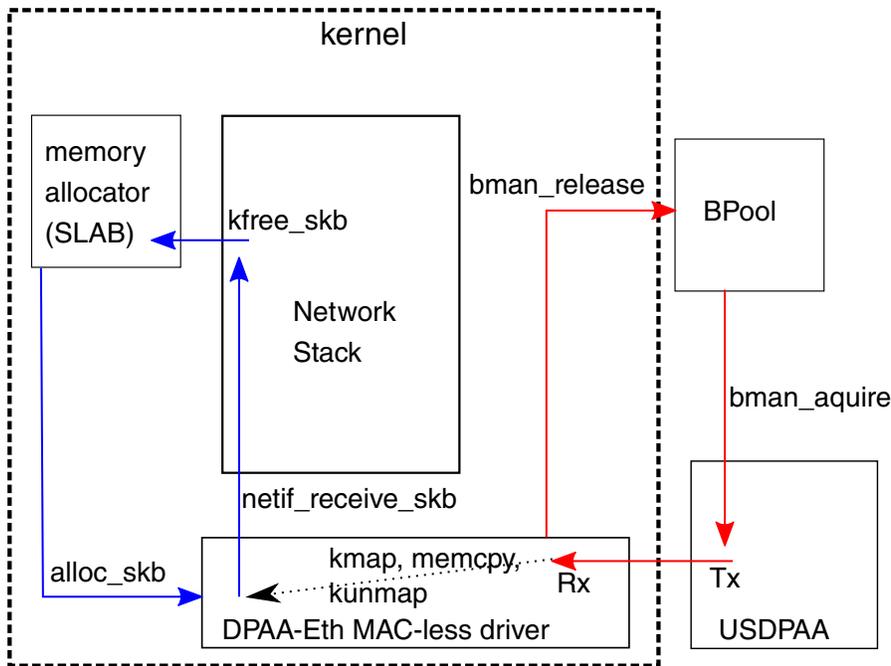


Figure 99. a MAC-less Device Used for Communication from USDPAA to Linux

5.4.1.1.6 Advanced Drivers Use Cases

This chapter is meant as an initial guideline toward building more complex use-cases, based on the predefined built-in capabilities of the DPAA-Ethernet driver that have been explained so far. (Note: Such use-cases are not currently part of the standard Linux QorIQ SDK.)

5.4.1.1.6.1 MAC-less Over OH (Linux-USDPAA)

One can interpose an OH port between a Linux MAC-less net device and a USDPAA MAC-less interface, where the Linux-to-USDPAA path goes through the OH port, while the USDPAA-to-Linux path is direct:

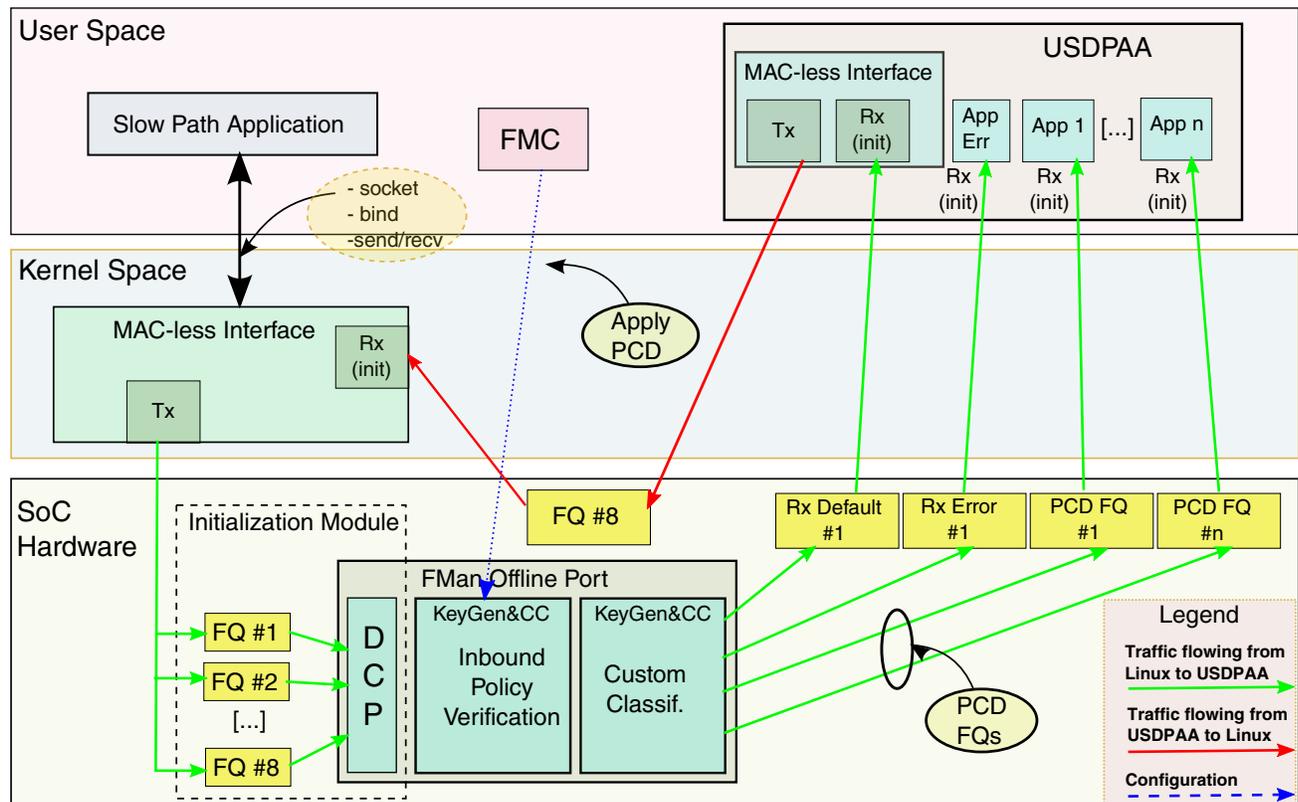


Figure 100. Linux MAC-less net device and a USDPAA MAC-less interface

In the current implementation, the DPAA Ethernet Driver initializes Linux MAC-less Rx FQs (= USDPAA's MAC-less Tx FQs).

Because there is no symmetric MAC-less endpoint to initialize the Linux endpoint's Tx FQs (as in the case of a standard MAC-less Linux-Linux setup), a separate kernel module is necessary in order to:

- Initialize the Linux MAC-less Tx FQs; and
- Add them to the DCP of the Offline Port.

The diagram shows:

Table 75. DPAA Ethernet Driver initializes FQs

#	Description	Detail
1	A MAC-less Linux interface with:	<ul style="list-style-type: none"> • 8 Tx FQs – initialized by the Offline Port Initialization Module and placed in the OH port’s DCP; • 8 Rx FQs – initialized by the MAC-less interface and connected to the Tx FQs of the USDPAA MAC-less interface.
2	A MAC-less USDPAA interface with:	<ul style="list-style-type: none"> • 8 Tx FQs – initialized by the Linux MAC-less interface. (Note: USDPAA currently asserts that there are exactly 8 Tx FQs. This can be changed if a different number of frame queues are requested). • 8 Rx FQs - initialized by the MAC-less interface. Unlike in the standard Linux-Linux MAC-less scenario, these are different from the Tx FQs of the Linux MAC-less interface, because of the interposing OH port. The Offline Port’s Rx Default and Rx Error FQs (and possibly the PCD FQs) use some of these FQIDs.
3	An Offline Port Initialization Module.	Responsible with initializing the 8 Tx FQs of the Linux MAC-less interface and connecting them to the DCP of the OH port.

The common idea of this design is that the entity which uses the queues for Rx is responsible with their initialization and (for software portals) specifying their dequeue callbacks. On the kernel side, this is the responsibility of the MAC-less interface (DPAA-Ethernet driver) and on USDPAA the responsibility is of the application that uses the FQs.

5.4.1.1.6.2 MAC-less Over OH (Linux-Linux)

Similarly to the previous use-case, one can use an Offline Port’s interface with the QMan for interposing an OH between two MAC-less endpoints and effectively build a MAC-less-over-OH net device.

5.4.1.1.6.3 ARP Handling in Shared MAC

1. Linux-Linux: Currently (FMan-v2), ARP packets arriving on a shared-MAC interface are (in absence of relevant PCD configuration) routed to the Linux partition.
2. Linux-USDPAA: This is entirely handled by USDPAA infrastructure. Please refer to the USDPAA documentation.

5.4.1.1.6.4 Multicast Support in Shared MAC

This is not a supported feature in the FMan-v2- and FMan-v3-based DPAA-Ethernet drivers.

5.4.1.1.7 Appendix A: Infrequently Asked Questions

Table 76. Q and A

#	Question	Answer
1	How do I send a frame up the network stack?	The frame-processing network stack only exists in the context of a net device. So, “sending a frame into the stack” is an inaccurate statement: the frame must first be associated to a net device, and then the respective instance of the Ethernet driver will deliver the frame to the stack, on behalf of that net device. To achieve that, the frame must arrive via the physical device that underlies the driver (or via a MAC-less device’s ingress FQs).
2	Can I allocate a buffer and inject it as a frame into a private interface’s ingress queues?	This is probably a mistake. The DPAA-Ethernet driver makes hard assumptions on buffer ownership, allocation and layout. In addition, the driver expects FMan Parse Results to be placed in the frame preamble, at an offset which is implementation-dependent. In short, while a carefully crafted code might work, it would make for <i>*very*</i> brittle design, and hard to maintain, too. <p style="text-align: center;">NOTE</p> <p style="text-align: center;">A MAC-less interface (potentially modified to interpose an OH port), however, will support this case, as long as: a) the buffer originates from the interface’s shared Buffer Pool; and b) the frame is a valid Ethernet frame.</p>
3	But can I acquire a buffer directly from a private interface’s Buffer Pool, and inject it as such into the private interface’s Rx FQs?	While this works on a MAC-less or shared-MAC interface, it is not an intended use-case for private interfaces.
4	What format must an ingress frame have, from the standpoint of the DPAA-Ethernet driver and the Linux kernel stack?	The DPAA-Ethernet driver is expected to perform an initial validation of the ingress frame, but does not look at the Layer-2 fields directly. The current kernel networking code does make a check on the MAC addresses of the frame and the protocol (Ethertype) field. One should not make assumptions on such details of frame processing, because the kernel stack implementation is not bound by any contract.

5.4.1.1.8 Appendix B: Frequently Asked Questions

The right place to look at the driver FAQs is the **Linux Ethernet Driver** document in the SDK.

5.4.1.2 Linux DPAA 1.x Ethernet Drivers

5.4.1.2.1 Introduction

This document describes the Linux drivers which enable support for Ethernet on processors with the Datapath Acceleration Architecture (DPAA). The focus is on the theory and operation behind using Ethernet. It provides only limited discussion of the BMan, QMan, and FMan, describing instead the layer of software which allows all of these to interoperate. For the purposes of this document, all these drivers will be referred to as the DPAA Ethernet Drivers. Enablement, configuration and debugging for all DPAA Ethernet Drivers are described in this document.

Purpose

The DPAA Ethernet Driver is meant to manage the use of the Datapath hardware for communication via the Ethernet protocol. This includes facilities for:

- Allocating buffer pools and buffers
- Allocating frame queues
- Assigning frame queues and buffer pools to specified FMan ports
- Transferring packets between frame queues and the Linux stack
- Controlling Link Management features

Overview

Ethernet in the Datapath is realized by interconnecting BMan, QMan, and FMan. The primary interaction for the DPAA Ethernet Driver is between the kernel and the QMan. Ethernet frames are delivered to the driver from the frame queue via the QMan portal, and the driver delivers Ethernet frames to the outgoing frame queue via the QMan portal. For some use cases, that is the only interaction.

Usually, the frame queues are connected to a FMan port. Each FMan port has two queues which must be assigned to them: a default queue and an error queue. This assignment can be specified in the device tree, or created dynamically by the driver on initialization.

The Ethernet frames are often stored in buffers which are associated with a BMan buffer pool. The driver sets up this pool, and either seeds it with buffers, or maps the buffers which are put into the pool. Depending on the pool, the buffers may be allocated and freed by the kernel during network activity, or they may be allocated once, and return to the pool when not in use by the Datapath hardware.

DPAA Ethernet Driver types

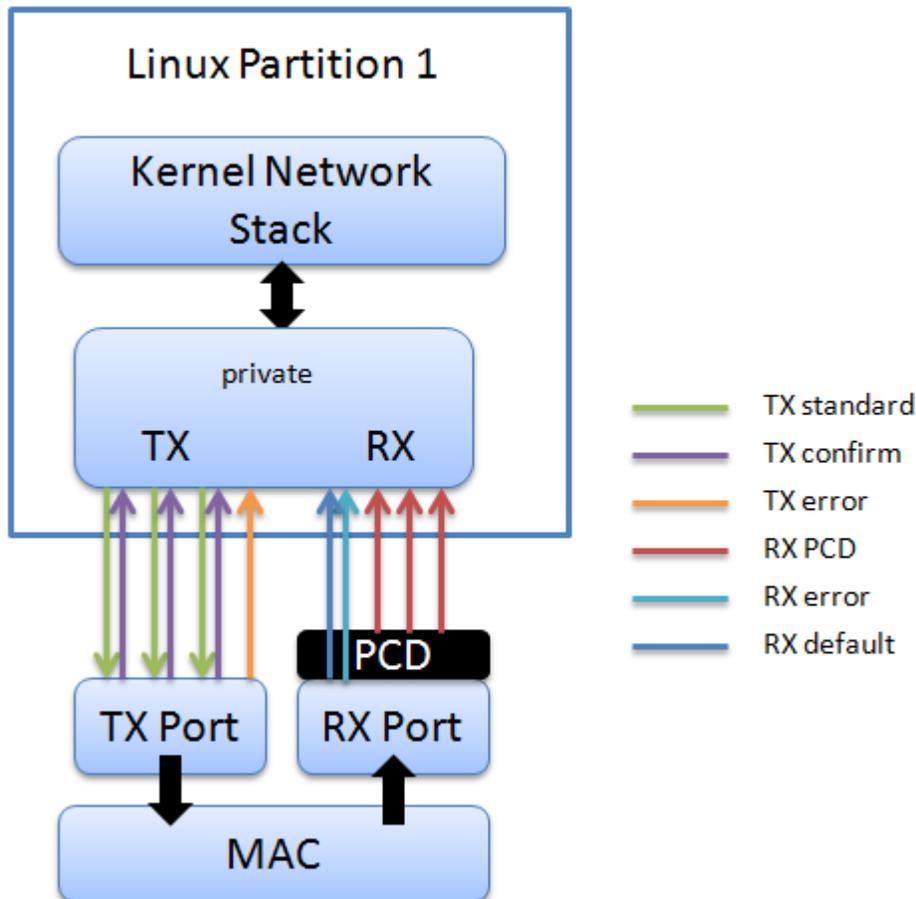
The complexity of DPAA allows a variety of possible use cases. Although speed is the key factor for performance in most use cases, customization and usability are preferred in others. Building a single Ethernet driver to cover all these tasks was, in the ever-changing world of the Linux kernel, a task that was becoming harder day-by-day. Instead we divided the single basic block into several specialized and simpler Ethernet drivers which can be combined in complex configurations:

- The Private DPAA Ethernet Driver resembles the common Linux Ethernet driver. It is highly improved for performance and uses all the features that DPAA offers;
- The Macless DPAA Ethernet Driver and the Shared DPAA Ethernet Driver flavors are used in virtualized multi-partition scenarios (with the Topaz Hypervisor) or in custom user-space traffic analyzer use cases;
- The Proxy DPAA Ethernet Driver will do the entire preliminary work in scenarios where all the control is passed to user-space, bypassing the standard Linux kernel standard.
- The Offload NIC Ethernet Driver is similar to the Macless Driver and can be used in the same scenarios. In addition, it has offloading capabilities that the Macless Driver lacks, and uses Offline Parsing/Host Command Ports.
- The MACsec Driver is used to configure the FMan MACsec hardware block that is capable of offloading the IEEE 802.1AE's protocol features.

5.4.1.2.2 Private DPAA Ethernet Driver

The Private DPAA Ethernet Driver manages the network interfaces which are fully owned by the Linux partition who runs them. Therefore, it is possible to take advantage of the DPAA facilities in order to increase the performance in both termination and forwarding scenarios.

The Private DPAA Ethernet Driver will be further referenced as the Private Driver.



5.4.1.2.2.1 Configuration

This chapter present the configuration options for the Private DPAA Ethernet Driver.

5.4.1.2.2.1.1 Device Tree Configuration

The compatible string used to define a private interface in device tree is „fsl,dpa-ethernet“. The default structure for the device tree node that specifies a Private interface should be similar to the below snippet of a B4860QDS device tree node:

```
ethernet@4 {
    compatible = "fsl,b4860-dpa-ethernet", "fsl,dpa-ethernet";
    fsl,fman-mac = <&fmlmac5>;
};
```

“fsl,fman-mac” is the reference to the MAC device connected to this interface. This property is used to determine which RX and TX ports are connected to this interface.

Buffer pools

A single buffer pool is currently defined and used by all the Private interfaces. The buffer pool ID is dynamically allocated and provided by the Buffer Manager. The number and size of the buffers in the pool is decided internally by the Private driver therefore no device tree configuration is accepted.

Frame queues

The frame queues are allocated by the Private driver with IDs dynamically allocated and provided by the Queue Manager. The frame queues can also be statically defined using two additional device tree properties:

```
ethernet@0 {
    compatible = "fsl,b4860-dpa-ethernet", "fsl,dpa-ethernet";
    fsl,fman-mac = <&fmlmac5>;
    fsl,qman-frame-queues-rx = <0x100 1 0x101 1 0x180 128>;
    fsl,qman-frame-queues-tx = <0x200 1 0x201 1 0x300 8>;
};
```

Within the example above, a value of 0x100 was assigned to the RX error frame queue ID and 0x101 to the RX default frame queue ID. In addition, 128 PCD frame queues ranging between 0x180-0x1ff are defined and assigned to the core-affined portals in a round-robin fashion.

There is exactly one RX error and one RX default queue hence a value of "1" for the frame count. Optionally, one can specify a value of "0" for the base to instruct the driver to dynamically allocate the frame queue IDs.

Within the example above, a value of 0x200 was assigned to the TX error queue ID and 0x201 to the TX confirmation queue ID. The third entry specifies the queues used for transmission.

If the qman-frame-queues-rx and qman-frame-queues-tx are not present in the device tree, the number of dynamically allocated TX queues is equal to the number of cores available in the partition.

5.4.1.2.2.1.2 Bootargs

Two bootarg parameters are defined for the Frame Manager driver but they also influence the behavior of the Private driver:

- fsl_fm_max_frm
- fsl_fm_rx_extra_headroom

fsl_fm_max_frm

The Frame Manager discards both Rx and Tx frames that are larger than a specific Layer2 MAXFRM value. The DPAA Ethernet driver won't allow one to set an interface's MTU too high such that it would produce Ethernet frames larger than MAXFRM. The maximum value one can use as the MTU for any interface is (MAXFRM - 22) bytes, where 22 is the size of an Eth+VLAN header (18 bytes), plus the Layer2 FCS (4 bytes).

Currently, the value of MAXFRM is set at boot-time and cannot be changed without rebooting the system.

The default MAXFRM is 1522, allowing for MTUs up to 1500. If a larger MTU is desired, one would have to reboot and reconfigure the system as described next. The maximum MAXFRM is 9600

The MAXFRM can be set in two ways:

- as a Kconfig option (CONFIG_FSL_FM_MAX_FRAME_SIZE):

```
Device Drivers
+--> Network device support
    +--> Ethernet driver support
        +--> Freescale devices
            +--> Frame Manager support
                +--> Freescale Frame Manager (datapath) support
                    +--> Maximum L2 frame size
```

- as a bootarg:

- In no-HV scenarios: In the u-boot environment, add "fsl_fm_max_frm=<your_MAXFRM>" directly to the "bootargs" variable.
- In Hypervisor-based scenarios: Add or modify the "chosen" node in Hypervisor device tree having a "bootargs" property specifying "fsl_fm_max_frm=<your_MAXFRM>;".

Note that any value set directly in the kernel bootargs will override the Kconfig default. If not explicitly set in the bootargs, the Kconfig value will be used.

Symptoms of Misconfigured MAXFRM

MAXFRM directly influences the partitioning of FMan's internal MURAM among the available Ethernet ports, because it determines the value of an FMan internal parameter called "FIFO Size". Depending on the value of MAXFRM and the number of ports being probed, some of these may not be probed because there is not enough MURAM for all of them. In such cases, one will see an error message in the boot console.

fsl_fm_rx_extra_headroom

Configure this to tell the Frame Manager to reserve some extra space at the beginning of a data buffer on the receive path, before Internal Context fields are copied. This is in addition to the private data area already reserved for driver internal use. The option does not affect in any way the layout of transmitted buffers. The default value (64bytes) offers best performance for the case when forwarded frames are being encapsulated (e.g. IPSec).

The RX extra headroom can be set in two ways:

- as a Kconfig option (CONFIG_FSL_FM_RX_EXTRA_HEADROOM):

```
Device Drivers
+--> Network device support
    +--> Ethernet driver support
        +--> Freescale devices
            +--> Frame Manager support
                +--> Freescale Frame Manager (datapath) support
                    +--> Add extra headroom at beginning of data buffers
```

- as a bootarg:
 - In no-HV scenarios: In the u-boot environment, add "fsl_fm_rx_extra_headroom=< your_rx_extra_headroom>" directly to the "bootargs" variable.
 - In Hypervisor-based scenarios: Add or modify the "chosen" node in Hypervisor device tree having a "bootargs" property specifying " fsl_fm_rx_extra_headroom=<your_rx_extra_headroom>;".

5.4.1.2.2.1.3 Kconfig Options

The private driver has a number of parameters which can be tuned at compile time from menuconfig. These can be found in:

```
Device Drivers
+- Network Device Support
    +- Ethernet Driver Support
        +- Freescale Devices
            +- DPAA Ethernet
```

FSL_DPAA_ETH_JUMBO_FRAME - "Optimize for jumbo frames"

Optimizes the DPAA Ethernet driver throughput for large frames termination traffic (e.g. 4K and above).

On FMan v2 platforms this option requires modifications to the device tree, otherwise the ping will not work. Include 'qoriq-fman-0-chosen-fifo-resize.dtsi' in 'qoriq-fman-0.dtsi' and 'qoriq-fman-1-chosen-fifo-resize.dtsi' in 'qoriq-fman-1.dtsi'

```
/include/ "qoriq-fman-0-chosen-fifo-resize.dtsi"
```

Using this option in combination with small frames increases significantly the driver's memory footprint and may even deplete the system memory. Also, the skb truesize is altered and messages from the stack that warn against this are bypassed.

This option is not available on LS1043A platforms.

FSL_DPAA_1588 - "IEEE 1588-compliant timestamping"

Enables IEEE1588 support code.

FSL_DPAA_TS - "Linux compliant timestamping"

Enables Linux API compliant timestamping support.

FSL_DPAA_ETH_USE_NDO_SELECT_QUEUE - "Use driver's Tx queue selection mechanism"

The DPAA-Ethernet driver defines a `ndo_select_queue()` callback for optimal selection of the egress FQ. That will override the XPS support for this netdevice. If for whatever reason you want to be in control of the egress FQ-to-CPU selection and mapping, or simply don't want to use the driver's `ndo_select_queue()` callback, then unselect this and use the standard XPS support instead.

FSL_DPAA_ETH_MAX_BUF_COUNT - "Maximum number of buffers in private bpool"

Defaults to "128". The maximum number of buffers to be by default allocated in the DPAA-Ethernet private port's buffer pool. One needn't normally modify this, as it has probably been tuned for performance already. This cannot be lower than `DPAA_ETH_REFILL_THRESHOLD`.

FSL_DPAA_ETH_REFILL_THRESHOLD - "Private bpool refill threshold"

Defaults to "128". The maximum number of buffers to be by default allocated in the DPAA-Ethernet private port's buffer pool. One needn't normally modify this, as it has probably been tuned for performance already. This cannot be lower than `DPAA_ETH_REFILL_THRESHOLD`.

FSL_DPAA_CS_THRESHOLD_1G - "Egress congestion threshold on 1G ports"

The size in bytes of the egress Congestion State notification threshold on 1G ports. Ranges from 0x1000 to 0x10000000. Defaults to 0x06000000. This option can help when:

- the device stays congested for a prolonged time (risking the netdev watchdog to fire - see also the `tx_timeout` module param)
- preventing the Tx cores from tightly-looping (as if the congestion threshold was too low to be effective)

This might also implies some risks:

- affecting performance of protocols such as TCP, which otherwise behave well under the congestion notification mechanism
- running out of memory if the CS threshold is set too high

FSL_DPAA_CS_THRESHOLD_10G - "Egress congestion threshold on 10G ports"

The size in bytes of the egress Congestion State notification threshold on 10G ports. Ranges from 0x1000 to 0x20000000. Defaults to 0x10000000.

FSL_DPAA_INGRESS_CS_THRESHOLD - "Ingress congestion threshold on FMan ports"

The size in bytes of the ingress tail-drop threshold on FMan ports. Defaults to 0x10000000. Traffic piling up above this value will be rejected by QMan and discarded by FMan.

FSL_DPAA_ETH_DEBUGFS - "DPAA Ethernet debugfs interface"

This option compiles debugfs code for the DPAA Ethernet driver.

FSL_DPAA_ETH_DEBUG - "DPAA Ethernet Debug Support"

This option compiles debug code for the DPAA Ethernet driver.

5.4.1.2.2.1.4 ethtool Options

The private driver implements the following ethtool operations

```
-a --show-pause
    Queries the specified Ethernet device for pause parameter information.
-A --pause
    Changes the pause parameters of the specified private devices.
    rx on|off
        Specifies whether RX pause should be enabled.
    tx on|off
        Specifies whether TX pause should be enabled.
-k --show-features
    Lists the offloadable DPAA driver features. Specifies which features can be changed.
-K --features
    Changes a driver feature.
    feature on|off
        Specifies whether a certain feature should be enabled.
-s --change
    msglvl N
    msglvl type on|off ...
        Sets the driver message type flags by name or number. type names the type of message to
        enable or disable; N specifies the new flags numerically.
-S --statistics
    Shows driver statistics and counters: interrupt counter, packet counters, error counters,
    congestion state, and more.
--show-eee
    Shows the Energy-Efficient Ethernet configurations.
--set-eee
    Configures the EEE behavior.
```

5.4.1.2.2.2 Features

This chapter presents the Private DPAA Ethernet Driver features.

5.4.1.2.2.2.1 Congestion Management

QMan offers 3 methods of managing congestion:

- WRED
- congestion state tail drop (CSTD)
- FQ tail drop (FQTD)

The Private driver implements CSTD both on TX and on RX. When the number of bytes residing in a TX FQ congestion group reaches a congestion threshold (high watermark), the QMan rejects any further incoming frames, until the sum of all the frames contained in the congestion groups drops under a low watermark, which is 7/8 of the high watermark. The high watermark can be configured from menuconfig. See section "Kconfig options" for more details.

5.4.1.2.2.2.1.1 Bootargs

Two bootarg parameters are defined for the Frame Manager driver but they also influence the behavior of the Private driver:

- fsl_fm_max_frm

- `fsl_fm_rx_extra_headroom`

fsl_fm_max_frm

The Frame Manager discards both Rx and Tx frames that are larger than a specific Layer2 MAXFRM value. The DPAA Ethernet driver won't allow one to set an interface's MTU too high such that it would produce Ethernet frames larger than MAXFRM. The maximum value one can use as the MTU for any interface is (MAXFRM - 22) bytes, where 22 is the size of an Eth+VLAN header (18 bytes), plus the Layer2 FCS (4 bytes).

Currently, the value of MAXFRM is set at boot-time and cannot be changed without rebooting the system.

The default MAXFRM is 1522, allowing for MTUs up to 1500. If a larger MTU is desired, one would have to reboot and reconfigure the system as described next. The maximum MAXFRM is 9600

The MAXFRM can be set in two ways:

- as a Kconfig option (`CONFIG_FSL_FM_MAX_FRAME_SIZE`):

```
Device Drivers
+--> Network device support
    +--> Ethernet driver support
        +--> Freescale devices
            +--> Frame Manager support
                +--> Freescale Frame Manager (datapath) support
                    +--> Maximum L2 frame size
```

- as a bootarg:
 - In no-HV scenarios: In the u-boot environment, add "`fsl_fm_max_frm=<your_MAXFRM>`" directly to the "bootargs" variable.
 - In Hypervisor-based scenarios: Add or modify the "chosen" node in Hypervisor device tree having a "bootargs" property specifying "`fsl_fm_max_frm=<your_MAXFRM>;`".

Note that any value set directly in the kernel bootargs will override the Kconfig default. If not explicitly set in the bootargs, the Kconfig value will be used.

Symptoms of Misconfigured MAXFRM

MAXFRM directly influences the partitioning of FMan's internal MURAM among the available Ethernet ports, because it determines the value of an FMan internal parameter called "FIFO Size". Depending on the value of MAXFRM and the number of ports being probed, some of these may not be probed because there is not enough MURAM for all of them. In such cases, one will see an error message in the boot console.

fsl_fm_rx_extra_headroom

Configure this to tell the Frame Manager to reserve some extra space at the beginning of a data buffer on the receive path, before Internal Context fields are copied. This is in addition to the private data area already reserved for driver internal use. The option does not affect in any way the layout of transmitted buffers. The default value (64bytes) offers best performance for the case when forwarded frames are being encapsulated (e.g. IPSec).

The RX extra headroom can be set in two ways:

- as a Kconfig option (`CONFIG_FSL_FM_RX_EXTRA_HEADROOM`):

```
Device Drivers
+--> Network device support
    +--> Ethernet driver support
        +--> Freescale devices
            +--> Frame Manager support
                +--> Freescale Frame Manager (datapath) support
                    +--> Add extra headroom at beginning of data buffers
```

- as a bootarg:

- In no-HV scenarios: In the u-boot environment, add "fsl_fm_rx_extra_headroom=< your_rx_extra_headroom>" directly to the "bootargs" variable.
- In Hypervisor-based scenarios: Add or modify the "chosen" node in Hypervisor device tree having a "bootargs" property specifying " fsl_fm_rx_extra_headroom=<your_rx_extra_headroom>";".

5.4.1.2.2.2 Scatter/Gather Support

On the Rx path, the first S/G entry is used to build the skb linear part and the other entries are used as fragments.

The Private driver can access the egress skbufs allocated in high memory (e.g. mapped directly from user-space, as is the case of the sendfile() system call). This eliminates the kernel need to copy such skbufs into newly-allocated low memory buffers, allowing zero-copy on the egress path.

5.4.1.2.2.3 Jumbo Frames Support

Termination traffic with large frames performs better if only linear skbs (and single buffer frames) are used. The driver has the option to allocate Rx buffers large enough to accommodate the entire frame (of max 9.6K).

This option needs to be used with caution, as the memory footprint can be a real problem when small frames are used.

The option can be enabled from the menuconfig option:

```
Device Drivers
+-> Network Device Support
    +-> Ethernet Driver Support
        +-> Freescale Devices
            +-> DPAA Ethernet
                +-> Optimize for jumbo frames
```

In addition to enabling this feature from menuconfig, the user is required to set the L2 maximum frame size to 9600, otherwise the configuration is not valid. This can be achieved by either setting fsl_fm_max_frm=9600 in the bootargs, or configuring CONFIG_FSL_FM_MAX_FRAME_SIZE from menuconfig. For more details see [Bootargs](#) on page 496.

5.4.1.2.2.4 GRO/GSO Support

Generic Receive Offload (GRO) is tied to NAPI support and works by keeping a list of GRO flows per each NAPI instance. These flows can then "merge" incoming packets, until some termination condition is met or the current NAPI cycle ends, at which point the flows are flushed up the protocol stack. Flows merging several packets share the protocol headers and coalesce the payload (without memcopying it). This results in a CPU load decrease and/or network throughput increase. Packets which don't match any of the stored flows (in the current NAPI cycle) are sent up the stack via the normal, non-GRO path.

GRO is commonly supported in hardware as a set of "GRO assists", rather than full packet coalescing. The following features count as GRO assists:

- RX hardware checksum validation
- Receive Traffic Distribution (RTD)
- Multiple RX/TX queues
- Receive Traffic Hashing
- Header prefetching
- Header separation
- Core affinity
- Interrupt affinity

Note: With the exception of header separation, the DPAA platforms feature all other hardware assists. Most notably, they are implicitly achieved through the mechanisms that accompany PCDs.

Generic Segmentation Offload (GSO) is also a well-established feature in the Linux kernel. Normally, a TCP segment is composed in the Layer 4 of the Linux stack, based on the current MSS (Maximum Segment Size) connection setting. It has been observed, though, that delaying segmentation is a better approach in terms of CPU load, because fewer headers are processed. Linux has taken an optimization approach, called GRO, whereby the L4 segments are only composed just before they are handed over to the L2 driver.

GRO and GSO support are available by default in the Private driver and can be independently switched on and off at runtime, via *ethtool -k*.

Note: Older versions of ethtool don't support this. Ethtool version 3.0 does - and possibly others before it, too.

Generic optimizations that enhance the driver's performance in the general case also apply to the GRO/GSO-enabled driver. PCD support is therefore recommended in this regard. We have found that these optimizations yield the best results on 10Gbps traffic, and to a lesser extent (if any) on 1Gbps traffic. TCP tests, especially, can benefit from GRO by shedding CPU load and upping the network throughput. The improvements are the more visible with smaller network MTU - with MTU=1500 and below, the benefits are higher, while starting from MTU=4k they are no longer observable.

One optimization that boosts GSO performance is the zero-copy egress path. That is available thanks to the *sendfile()* system call, which may be used instead of the plain *send()* syscall, and which certain benchmark applications know about. Netperf for instance has *sendfile* support in its *TCP_SENDFILE* tests.

GRO and GSO are no panacea, one-button-fix-all kind of optimization. While under most circumstances they should be transparent (this being why GRO is by default enabled in the Linux kernel), there are scenarios and configurations where they may in fact under-perform. Traffic on 1Gbps ports sees little benefit from GRO/GSO. Also, if the Private Driver detects that PCDs are not in place, GRO is automatically by-passed.

5.4.1.2.2.2.5 Transmit Packet Steering

The Private driver exposes to the Linux networking stack a TX-multiqueue interface. This provides the stack with better control of the transmission queues and reduces the need for locking. The user may also control the mapping of egress FQs to the CPUs via a standard Linux feature called Transmit Packet Steering (XPS) and documented here: <http://lwn.net/Articles/412062/>

NOTE

The kernel transmission queues are different entities than the Private driver Frame Queues.

The Private driver, however, matches the two realms by mapping the DPAA FQs onto kernel's own queue structures. To that end, the Private driver provides a standard callback (net-device operation, or NDO) called *ndo_select_queue()*, which the stack can interrogate to find out the specific queue mapping it needs for transmitting a frame. The existence of that NDO (which is otherwise optional) overrides the kernel queue selection via XPS. This is why the Private driver provides a compile-time choice to disable the *ndo_select_queue()* callback, leaving it to the stack to choose a transmission queue.

To use the Private driver's builtin *ndo_select_queue()* callback, select the Kconfig option **FSL_DPAA_ETH_USE_NDO_SELECT_QUEUE**.

To disable the Private driver's queue selection mechanism and use XPS instead, unselect this Kconfig option. Further on, the users can configure their own txq-to-cpu mapping, as described in the LWN article above.

5.4.1.2.2.2.6 TX and RX Hardware Checksum

Introduction

The FMan block supports calculation of the L3 and/or L4 checksum for certain standard protocols.

This can be used, on the TX path, for calculating the checksum of the outgoing frame, and on the RX path, for validating the L3/L4 checksum of the incoming frame and making classification, or distribution decisions.

TX Checksum Support

On TX, the checksum computation is enabled on a per-frame basis by the Private driver. The TX checksum support for standard protocols is as follows:

Table 77. TX checksum support

Header	IPv4	IPv6	Other
IP header	yes	not available	no
TCP header	yes	yes	no
UDP header	yes	yes	no

NOTE

IP Header checksum capability also exists in SEC block (see IPSEC).

NOTE

Ethernet CRC is calculated on a per frame basis during frame transmission.

NOTE

The main precondition for TX checksum to be enabled in hardware is that IP tunneling must not be present (i.e., not GRE, not MinEnc, not IPIP). Other conditions pertain to the validity and integrity of the frame.

RX Checksum Support

This feature is disabled by default. In order to enable RX checksum computation for supported protocols, a PCD scheme must be applied to the respective RX port. In the current release, L3 and L4 are both enabled if a PCD is applied.

If enabled, L3 and L4 checksum validation is performed for TCP, UDP and IPv4.

NOTE

Controlling this feature via ethtool is not yet supported.

5.4.1.2.2.2.7 Pause Frames Flow Control

FMan supports IEEE 802.3x flow control. Whenever the FMan RX FIFO threshold is exceeded, FMan transmits PAUSE frames to the other peer on the link. In Linux, the transmission and reception of PAUSE frames can be enabled or disabled using ethtool.

To display PAUSE frames settings in use for an interface

```
ethtool -a intf_name
```

Triggering PAUSE frames ON/OFF

PAUSE frames can be enabled/disabled on RX/TX using ethtool -A, like in the following examples

```
ethtool -A intf_name rx on
ethtool -A intf_name tx off
ethtool -A intf_name rx off tx off
```

Autonegotiation

Starting with SDK 1.6, the DPAA Private driver supports PAUSE frame autonegotiation.

When autonegotiation is enabled and the user enables/disables PAUSE frames on RX/TX, these will not automatically be triggered on/off. Instead, the local and the peer PAUSE symmetric/asymmetric capabilities will be considered. If the peer does not match the local capabilities, the following commands may have no effect:

```
ethtool -A intf_name rx on
ethtool -A intf_name rx off
ethtool -A intf_name tx on
ethtool -A intf_name tx ff
```

When autonegotiation is disabled, ethtool settings override the result of link negotiation.

PAUSE frame autonegotiation can also be enabled/disabled using ethtool -A

```
ethtool -A intf_name autoneg on
ethtool -A intf_name autoneg off
```

FMAN v3 platforms

On the following platforms: T4, B4, and T1040, 802.1Qbb Priority Flow Control is used instead of 802.3x PAUSE frames. The ethtool controls are the same, but the structure of the frames is different.

Find about DPAA Private support of PFC in the following section:

[Priority Flow Control](#) on page 504

5.4.1.2.2.2.8 Priority Flow Control

Beginning with SDK 1.6, the DPAA Ethernet Driver offers experimental support for IEEE standards 802.1Qbb (Priority Flow Control) and 802.1p.

These standards aim to implement lossless Ethernet, in which the highest-priority classes of traffic benefit from maximum bandwidth and minimum delay. Up to 8 classes of service can be used, but only a minimum of 3 is required.

The terms “Class of Service (CoS)” and “priority” will be used interchangeably in this section.

802.1Qbb PFC frames are available only on platforms with FMan v3, namely T4, B4 and T1040. For the other platforms 802.3x PAUSE frames are used instead for Ethernet flow control.

Enabling PFC Support

To enable PFC support, enable the following options from menuconfig

```
Device Drivers
+ Network device support
  + Ethernet driver support
    + Freescale devices
      + Frame Manager support
        + Freescale Frame Manager (datapath) support
          + FMan PFC support (EXPERIMENTAL)
            + (3)      Number of PFC Classes of Service
            + (65535) The pause quanta for PFC CoS 0
            + (65535) The pause quanta for PFC CoS 1
```

```
+ (65535) The pause quanta for PFC CoS 2
```

The number of Classes of Service can range between 1 and 4. It defines the number of Work Queues used and the number of priorities that are set when a PFC frame is issued. 3 is the default value. Changing this value also changes the number of WQs and priorities.

The pause time can be adjusted for each CoS individually.

Enabling and disabling CoS and their pause time is unavailable at runtime. It is only possible at compile time in this release.

Selecting the Class of Service

When PFC support is enabled, the egress traffic flowing on a DPAA Private interface is distributed on the first 3 Work Queues of a TX port, namely WQ0, WQ1 and WQ2.

These function in strict priority. WQ0 has the highest priority and WQ2 the lowest priority. FMan cannot dequeue frames from WQ1 unless WQ0 is empty and from WQ2 unless WQ1 and WQ0 are empty.

The work queue a frame will be enqueued on is determined from the socket buffer priority. `skb_prio` is just an internal tag that the kernel applies to the frames on the egress path and is not visible to the receiver.

<code>skb_prio</code>	CoS
0	0
1	1
≥ 2	2

The default `skb_prio` is 0, which means all frames will be distributed to WQ0. `skb_prio` can be modified using a number of methods, including traffic control.

To edit a socket buffer's priority using `tc`, one needs to enable the following options from `menuconfig`.

```
Networking support
+ Networking options
  + QoS and/or fair queueing
    + Multi Band Priority Queueing (PRIO)
    + Elementary classification (BASIC)
    + Universal 32bit comparisons w/ hashing (U32)
    + Extended Matches
      + U32 key
    + Actions
      + SKB Editing
```

The following commands assign a `skb_prio` of 1 to traffic destined to TCP and UDP port 5000 and implicitly direct it on WQ1.

```
tc qdisc del dev fm1-mac9.0 root
tc qdisc add dev fm1-mac9.0 root handle 1: prio
tc filter add dev fm1-mac9.0 parent 1: protocol ip u32 match ip dport 5000 action skbedit
priority 1
```

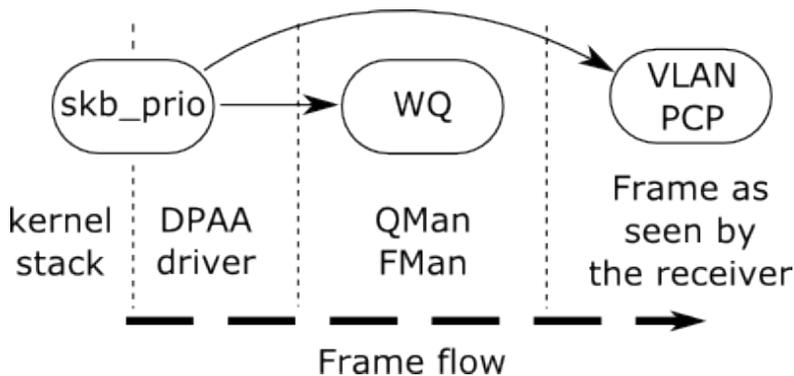
VLAN tagging

In order to be classified by the receiver according to 802.1p the egress traffic must be VLAN tagged, with the Class of Service contained in the PCP field. The PCP priority is also determined from `skb_prio`.

```
# create a subinterface of fm1-mac9, with VLAN ID 0
vconfig add fm1-mac9 0
# all frames tagged with skb_prio 1, will have PCP priority of 1.
vconfig set_egress_map fm1-mac9.0 1 1
```

If no mapping is specified the PCP field will be set to 0 by default.

The dependence between `skb_prio`, work queues and VLAN PCP priority:



Receiving PFC Frames

Unlike ordinary 802.3x PAUSE frames, PFC frames can selectively pause a certain priority/CoS.

WQ0 responds to PFC frames that have priority 0 set. Example: When a PFC frame arrives containing priority 0 and having a 100 pause time for priority 0, WQ0 i.e. all traffic from CoS 0 is ignored for dequeuing for 100 bit times, and dequeuing is done from WQ1 and WQ2.

Generating PFC frames

All DPAA Private interfaces share a single buffer pool which accounts for the buffers in which the frames are stored upon receiving.

When the Buffer Pool reaches the refill/depletion threshold, PFC frames are sent back to the sender in order to pause frames transmission and thus avoid frame loss.

FMan sends PFC frames that pause all Classes of Traffic defined. The only difference between the classes is the pause time.

The pause time can be configured from menuconfig. A pause time of 0 disables that Class of Service.

When the common buffer pool depletes, issued PFC frames look like this.

Class-Enable Vector							
1	1	1	0	0	0	0	0
Pause Quanta Class 0							
Pause Quanta Class 1							
Pause Quanta Class 2							
0							
0							
...							

FMan issues either 802.1Qbb PFC or 802.3x PAUSE frames depending on the platform, but there is no difference in controlling their transmission and reception via ethtool. For more details, see the chapter on PAUSE frames support.

Enabling and disabling PFC using ethtool

FMan issues either 802.1Qbb PFC or 802.3x PAUSE frames depending on the platform, but there is no difference in controlling their transmission and reception via ethtool. For more details, see the chapter on PAUSE frames support.

[Pause Frames Flow Control](#) on page 503

5.4.1.2.2.9 Core Affined Queues

The driver automatically creates 128 core-affined queues, intended to be used as RX PCD frame queues. These frame queues can be used in PCD configuration files to process certain types of frames on particular CPUs. In order to enhance the PCD files creation, the `/etc/fmc/config/` directory from rootfs contains the default configuration and policy files for each platform.

The driver calculates the frame queue IDs based on the address of the MAC registers corresponding to the port using the following formula:

$$((\text{MAC register address}) \& 0x1ffff) \gg 6$$

Following are the values for various QorIQ DPAA platforms:

Table 78. FMAN v2 devices core affined queues

Interface	FQID base	P4080	P5020	P5040	P3041	P2041	T1040 (FMAN v3)
fm1-gb0	0x3800	Y	Y	Y	Y	Y	Y
fm1-gb1	0x3880	Y	Y	Y	Y	Y	Y
fm1-gb2	0x3900	Y	Y	Y	Y	Y	Y
fm1-gb3	0x3980	Y	Y	Y	Y	Y	Y
fm1-gb4	0x3a00		Y	Y	Y		Y
fm1-10g	0x3c00	Y	Y	Y			

Table continues on the next page...

Table 78. FMAN v2 devices core affined queues (continued)

Interface	FQID base	P4080	P5020	P5040	P3041	P2041	T1040 (FMAN v3)
fm2-gb0	0x7800	Y		Y			
fm2-gb1	0x7880	Y		Y			
fm2-gb2	0x7900	Y		Y			
fm2-gb3	0x7980	Y		Y			
fm2-gb4	0x7a00			Y			
fm2-10g	0x7c00	Y		Y			

Table 79. FMAN v3 devices core affined queues

Interface	FQID base	T4240	T4160	B4860	B4420	T2080
fm1-mac1	0x3800	Y	Y	Y	Y	Y
fm1-mac2	0x3880	Y	Y	Y	Y	Y
fm1-mac3	0x3900	Y	Y	Y	Y	Y
fm1-mac4	0x3980	Y	Y	Y	Y	Y
fm1-mac5	0x3a00	Y	Y	Y		
fm1-mac6	0x3a80	Y	Y	Y		
fm1-mac9	0x3c00	Y	Y	Y		Y
fm1-mac10	0x3c80	Y		Y		Y
fm2-mac1	0x7800	Y	Y			
fm2-mac2	0x7880	Y	Y			
fm2-mac3	0x7900	Y	Y			
fm2-mac4	0x7980	Y	Y			
fm2-mac5	0x7a00	Y	Y			
fm2-mac6	0x7a80	Y	Y			
fm2-mac9	0x7c00	Y	Y			
fm2-mac10	0x7c80	Y				

These queues are assigned to cores in a round-robin fashion. For instance, if there are 8 cores, 0x3800 will be serviced by core 0, 0x3801 by core 1, 0x3808 by core 0, etc. Currently, if one specifies extra RX PCD queues in the device tree, these queues will **also** be assigned in this round-robin fashion.

High Priority Core Affined Queues

Starting with SDK 2.0, a new set of RX PCD frame queues has been added, to aid in implementing complex traffic management scenarios. This set of frame queues has a higher priority than the normal RX PCD frame queues, and as such, traffic coming in on these frame queues has a higher precedence than the traffic coming on on the default RX PCD frame queues. One scenario where this is useful is the back-to-back IPsec testing scenario, where the encrypted traffic (RX) is desirable to have a higher priority than the plain text traffic.

The driver calculates the high priority frame queue IDs based on the address of the MAC registers corresponding to the port using the following formula:

$$65536 + ((\text{MAC register address}) \& 0x1ffff) \gg 6$$

Following are the values for various QorIQ DPAA platforms:

Table 80. FMAN v2 devices core affined queues

Interface	FQID base	P4080	P5020	P5040	P3041	P2041	T1040 (FMAN v3)
fm1-gb0	0x13800	Y	Y	Y	Y	Y	Y
fm1-gb1	0x13880	Y	Y	Y	Y	Y	Y
fm1-gb2	0x13900	Y	Y	Y	Y	Y	Y
fm1-gb3	0x13980	Y	Y	Y	Y	Y	Y
fm1-gb4	0x13a00		Y	Y	Y		Y
fm1-10g	0x13c00	Y	Y	Y			
fm2-gb0	0x17800	Y		Y			
fm2-gb1	0x17880	Y		Y			
fm2-gb2	0x17900	Y		Y			
fm2-gb3	0x17980	Y		Y			
fm2-gb4	0x17a00			Y			
fm2-10g	0x17c00	Y		Y			

Table 81. FMAN v3 devices core affined queues

Interface	FQID base	T4240	T4160	B4860	B4420	T2080
fm1-mac1	0x13800	Y	Y	Y	Y	Y

Table continues on the next page...

Table 81. FMAN v3 devices core affined queues (continued)

Interface	FQID base	T4240	T4160	B4860	B4420	T2080
fm1-mac2	0x13880	Y	Y	Y	Y	Y
fm1-mac3	0x13900	Y	Y	Y	Y	Y
fm1-mac4	0x13980	Y	Y	Y	Y	Y
fm1-mac5	0x13a00	Y	Y	Y		
fm1-mac6	0x13a80	Y	Y	Y		
fm1-mac9	0x13c00	Y	Y	Y		Y
fm1-mac10	0x13c80	Y		Y		Y
fm2-mac1	0x17800	Y	Y			
fm2-mac2	0x17880	Y	Y			
fm2-mac3	0x17900	Y	Y			
fm2-mac4	0x17980	Y	Y			
fm2-mac5	0x17a00	Y	Y			
fm2-mac6	0x17a80	Y	Y			
fm2-mac9	0x17c00	Y	Y			
fm2-mac10	0x17c80	Y				

5.4.1.2.3 Ethernet Advanced Drivers

5.4.1.2.3.1 Macless DPAA Ethernet Driver

Macless DPAA Ethernet Driver is a virtual Ethernet driver, hence not using an Ethernet controller to transmit frames. This type of DPAA Ethernet driver is a lightweight version of Private DPAA Ethernet Driver that does not have MAC device control primitives, but maintains the same structure. Macless DPAA Ethernet Driver is used with many different names, “macless;” “MAC-less” or previous SDK name, “Virtual Ethernet Driver.” This DPAA Ethernet Driver is used in simple scenarios, like communication between USDPAA and Linux or communication between two Linux partitions or more complex ones, like Offloading Architecture and Shared MAC scenarios. All these scenarios are presented in the Configuration chapter.

5.4.1.2.3.1.1 Configuration

The main configuration options are offered, like in any other embedded Linux Ethernet driver, by the device tree configuration and the common interface offered by the Linux kernel (ifconfig, ethtool, etc.). All configuration capabilities are presented in this chapter.

5.4.1.2.3.1.1.1 Device Tree Configuration

The Macless DPAA Ethernet Driver has *fsl,dpa-ethernet-macless* as compatible string in the device tree. For example, the standard structure for the device tree node that specifies a Macless interface in a B4860QDS device tree looks like:

```
ethernet@16 {
    compatible = "fsl,b4860-dpa-ethernet-macless", "fsl,dpa-ethernet-macless";
    fsl,bman-buffer-pools = <&bp10>;
    fsl,qman-frame-queues-rx = <0xfa0 0x8>;
    fsl,qman-frame-queues-tx = <0xfa8 0x8>;
    local-mac-address = [00 11 22 33 44 55];
};
```

The properties that a Macless Driver device tree node can have are:

- *fsl,bman-buffer-pools* - a list of buffer pools used by this interface. The Macless DPAA Ethernet Driver will use only static defined buffer pools because the frames are transmitted between different memory spaces. See Note 2 for more details;
- *fsl,qman-frame-queues-rx* - a list of base/count pairs of frame queues that will transmit frames to the Macless Driver. These queues are initialized as PCD queues. By default there are 8 Rx queues because there are 8 CPUs on B4860QDS boards;
- *fsl,qman-frame-queues-tx* - a list of base/count pairs of frame queues that fetch the frames from Macless Driver to the next hardware device (TX Port, O/H Port) or to another Macless Driver. These queues will only be created, but not initialized. It is the role of the next software module in the data path to initialize these queues;
- *local-mac-address* - this is a virtual L2 address used to identify the driver in the Linux kernel network stack.

Note 1: The Macless DPAA Ethernet Driver does not have a MAC device to control. Because of this the “fsl,fman-mac” property is missing from the device tree specification. This property exists for all other DPAA Ethernet Drivers.

Note 2: A static defined buffer pool should be declared as exemplified below for a B4860QDS board:

```
bp7: buffer-pool@7 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <7>;
    fsl,bpool-ethernet-cfg = < 0 256 0 192 0 0x40000000>;
    fsl,bpool-thresholds = <0x400 0xc00 0x0 0x0>;
};
```

Although the above device tree node is a representation for the BMan Driver, *fsl,bpool-ethernet-cfg* property is parsed solely by the DPAA Ethernet Driver that has a reference to this buffer pool. This property has the following meaning:

```
fsl,bpool-ethernet-cfg = <count size base_address>;
```

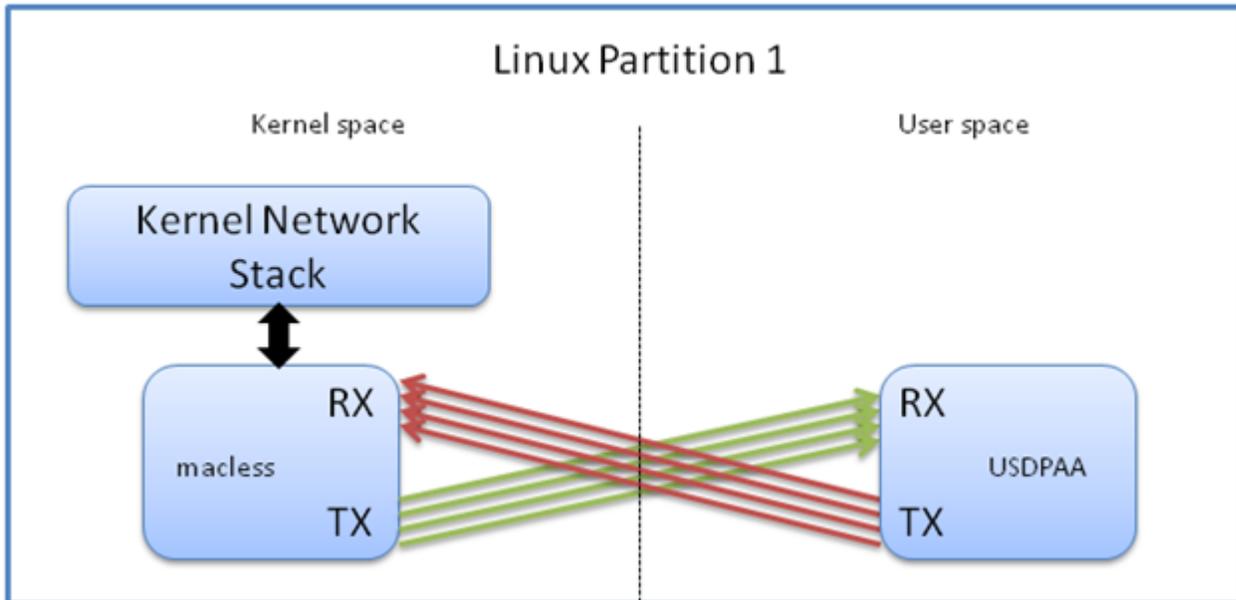
- *count* - represents the number of buffers from the buffer pool;
- *size* - buffer size;
- *base_address* - physical address of the buffer pool. Because two Linux partition have different memory spaces, this physical address will be mapped in both partitions. In scenarios where a single partition is used, this address will be invalid, particularly 0, and a dynamic mapping from user-space to kernel-space will be done.

The example above declares a buffer pool with buffer pool ID 7, and describes a pool with 256 192-byte buffers, occupying the memory region from 0x40000000 to 0x40000000 + 256*192. The reason there are two numbers per each of *count*, *size*, and *base_address* is that we support 36-bit addresses on the P4080 (and 64-bit on P5020). It should be noted that the size of those parameters should be set by the root node's *#address-cells* and *#size-cells* properties. The *fsl,bpool-ethernet-seeds* property is there to tell the driver which is using the buffer pool whether to seed the pool with the declared buffers, or not.

The above generic device tree structure can be modified, depending on the scenario that Macless Ethernet Driver is used into. These scenarios are:

Communication inside a single Linux partition between USDPAA and Linux Network Stack through a Macless DPAA Ethernet Driver.

For some applications, USDPAAs needs the benefits and flexibility of Linux networking capabilities. To connect to the Linux Network Stack, it uses a Macless DPAA Ethernet Driver. This scenario is represented in the following picture:



The device tree configuration should be similar to the one below extracted from a B4860QDS device tree configuration:

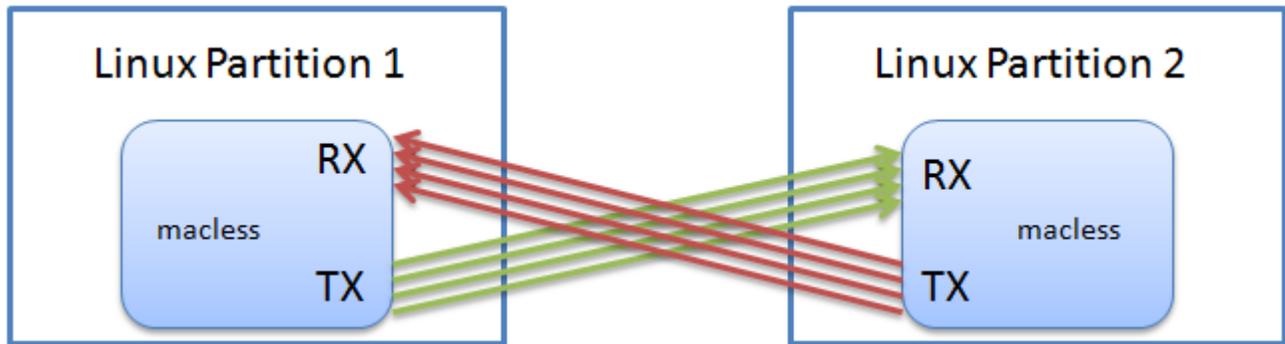
```
ethernet@16 {
    compatible = "fsl,b4860-dpa-ethernet-macless", "fsl,dpa-ethernet-macless";
    fsl,bman-buffer-pools = <&bp16>;
    fsl,qman-frame-queues-rx = <0xfa0 0x8>;
    fsl,qman-frame-queues-tx = <0xfa8 0x8>;
    local-mac-address = [00 11 22 33 44 55];
};

bp16: buffer-pool@16 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <0x10>;
    fsl,bpool-ethernet-cfg = <0x0 0x800 0x0 0x6c0 0x0 0x0>;
    fsl,bpool-thresholds = <0x100 0x300 0x0 0x0>;
};
```

In the above device tree snippet, there are two device tree nodes, one for the Macless DPAA Ethernet Driver and one for the static definition of the buffer pool used in the transmission. USDPAAs will also parse these nodes and it will manage the buffer pool used in the communication. Additionally, because a Macless DPAA Ethernet Driver initializes only its RX frame queues, USDPAAs has the additional role of initializing Macless DPAA Ethernet Driver's TX frame queues also. Another important configuration is the base address of the buffer pool. Because the communication occurs inside a single Linux partition, there is no need for the buffer pool to advertise its physical address, hence the 0 address representing an invalid physical address. The mapping of the Macless kernel memory space to USDPAAs's user-space memory space is done through the kernel *kmap/kunmap* primitives.

Communication between two Linux partitions through a Macless DPAA Ethernet Driver

With the help of the NXP virtualization solution, called Topaz, two or more partitions can run on the same board. Every communication between partitions is supervised by Topaz. Although network communication between partitions is possible through external connections, one direct connection can be created inside the board using Macless DPAA Ethernet Driver and hardware frame queues. To create a fast networking communication between partitions, two or more Macless DPAA Ethernet Drivers should be used as depicted below:



The TX frame queues from one partition should be the RX frame queues from the other partition. The device tree nodes in each partition are represented below.

First partition's device tree representation:

```

dpa-ethernet@10 {
    compatible = "fsl,p4080-dpa-ethernet", "fsl,dpa-ethernet-macless";
    fsl,qman-frame-queues-rx = <0x4000 8>;
    fsl,qman-frame-queues-tx = <0x4008 8>;
    local-mac-address = [02 00 c0 a8 6f fe];
    fsl,bman-buffer-pools = <&bp10>;
};
bp10: buffer-pool@10 {
    compatible = "fsl,b4080-bpool", "fsl,bpool";
    fsl,bpid = <10>;
    fsl,bpool-ethernet-cfg = <0 0x80 0 1728 0 0x80000000>;
    fsl,bpool-ethernet-seeds;
};

```

Device tree from the second partition:

```

dpa-ethernet@10 {
    compatible = "fsl,p4080-dpa-ethernet", "fsl,dpa-ethernet-macless";
    fsl,qman-frame-queues-rx = <0x4008 8>;
    fsl,qman-frame-queues-tx = <0x4000 8>;
    local-mac-address = [02 00 c0 a8 79 fe];
    fsl,bman-buffer-pools = <&bp10>;
};
bp10: buffer-pool@10 {
    compatible = "fsl,b4080-bpool", "fsl,bpool";
    fsl,bpid = <10>;
    fsl,bpool-ethernet-cfg = <0 0x80 0 1728 0 0x80000000>;
};

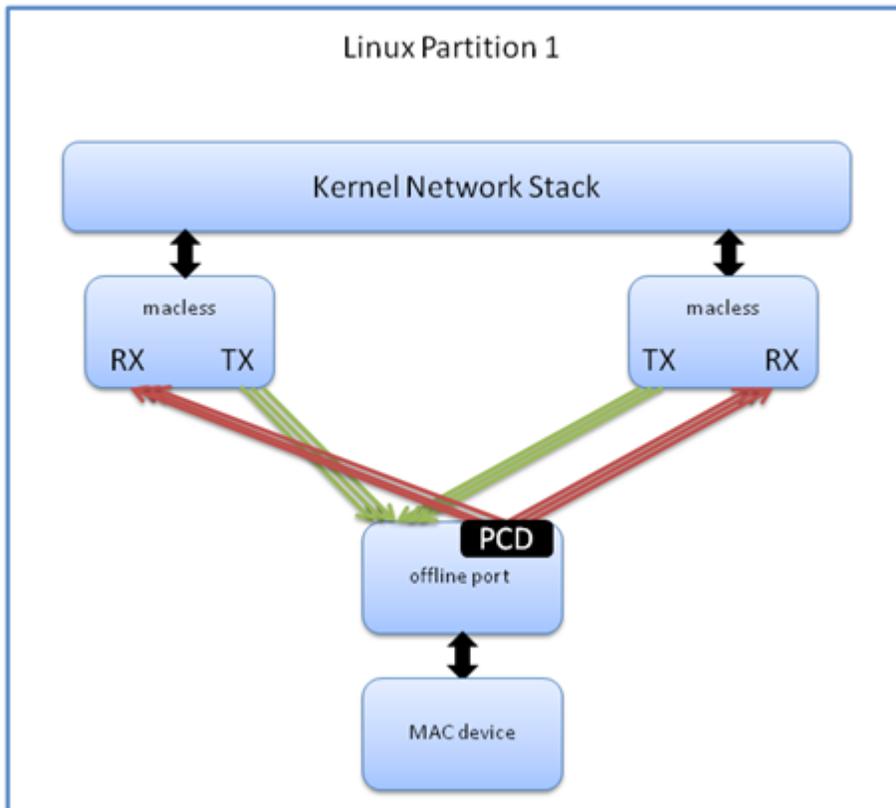
```

In the above configuration there are 8 TX frame queues and 8 RX frame queues used in interchangeable roles in each partition. Both interfaces share one statically defined buffer pool. The same buffer pool definition should be used in both partitions, except for the *fsl,bpool-ethernet-seeds* property. In order to easily map from one memory address space from one partition to another partition a valid physical base address must be configured. The mapping from physical address to each partition's virtual kernel address is done with *ioremap* kernel primitive. The seeding of the buffer pool will be done by the Macless DPAA Ethernet Driver from the partition that has *fsl,bpool-ethernet-seeds* in its device tree node. In the above configuration, the first partition will seed the buffer pool.

Communication inside a Linux partition using Offline Ports and Macless DPAA Ethernet Driver

DPAA has many hardware offload capabilities. One of them is the Offline Parsing/Host Command Port. This hardware portal can fetch traffic from multiple destinations (such as Macless DPAA Ethernet Drivers) and send traffic through many

configurable destinations, like MAC devices. For example, this scenario can be used to offload into DPAA stack functionality, like IPSec, IP fragmentation and reassembly or TCP segmentation and reassembly.



Because the Macless DPAA Ethernet Driver does not have an Ethernet controller attached, it can be used in many hardware configuration scenarios similar to the one used in the figure above. In order to do that, special attention should be paid to Frame Queues configuration and the Buffer Pool configuration.

Communication of different Linux partitions through a single MAC device, using Macless DPAA Ethernet Driver and Shared DPAA Ethernet Driver

In this scenario, a new type of DPAA Ethernet Driver is needed. This is called Shared DPAA Ethernet Driver and its configuration is described in Shared DPAA Ethernet Driver configuration chapter. Shared DPAA Ethernet Driver together with Macless DPAA Ethernet Driver are used in a typical shared MAC scenario, described in the Shared DPAA Ethernet Driver chapter.

5.4.1.2.3.1.1.2 Configuration available through Linux interfaces

Because Macless driver does not manage a MAC device, some *ethtool* and *ifconfig* options will not be available. All Layer 3 configurations, like setting an IP address, are generally available.

5.4.1.2.3.1.2 Features

This chapter describes the features of the Macless DPAA Ethernet Driver, their configuration options, fine-tuning and known limitations.

MAC device control capability

As of SDK 1.5, the Macless DPAA Ethernet Driver is capable of controlling a MAC device on behalf of an user-space application, like USDPAA. This will add an additional control mechanism to the amount of customization available through Macless DPAA Ethernet Driver. For example, in the scenario depicted in *Communication inside a Linux partition using Offline*

Ports and Macless DPAA Ethernet Driver scenario, it is desired that one of the Macless DPAA Ethernet Drivers should control the MAC device.

Some of the MAC devices are initialized by the Proxy DPAA Ethernet Driver to be used by USDPAA. By setting the following attribute in the device tree specification of the Macless DPAA Ethernet Driver

```
proxy = <&proxy1>;
```

a reference to the MAC device initialized by the Proxy driver can be obtained in the Macless driver. This will allow basic operations regarding L2 address of the MAC device using Macless driver Linux interface. Currently, the supported operations are enablement and disablement of the MAC device and setting multicast or unicast addresses.

Scatter/Gather

The current implementation supports DPAA Scatter/Gather only on the RX path. Therefore, the Macless DPAA Ethernet Driver will know how to handle Scatter/Gather frames, but the driver does not advertise TX Scatter/Gather capability to the Linux Network Core and only linear socket buffers will be accepted on the TX path.

Hardware checksum

As previously presented in the Private DPAA Ethernet Driver's Features chapter, the FMan supports computation of the L3 and/or L4 checksum for certain protocols (mainly TCP/IP and UDP/IP). Because the same transmission procedure is used to activate hardware checksum in both Private and Macless drivers, the same limitations presented in the *Private DPAA Ethernet Driver* chapter apply to Macless driver. Currently, at frame reception, the Macless Driver is not able to fetch the checksum computed by FMan.

5.4.1.2.3.2 Shared DPAA Ethernet Driver

Shared DPAA Ethernet Driver is similar to Macless Driver, only it has a MAC device in its control. Although it has similar structures with Private DPAA Ethernet Driver, it does not have the same optimization and feature set available. This is because it can be used in pair with a Macless Driver from another Linux partition or it can be used in pair with USDPAA. Therefore, it needs portability and easiness, qualities that Private DPAA Ethernet Driver sacrifices in exchange for higher performance. Shared DPAA Ethernet Driver is also known as *shared* or *Shared Controller* in the previous SDK release. In this document Shared DPAA Ethernet Driver will be referred as *Shared Driver*. This flavor of DPAA Ethernet Driver is used in two scenarios. First, when a MAC device is shared between different partitions under a Hypervisor and second, when a MAC device is shared with a User Space DPAA application in a single Linux partition. These use cases are presented in the Configuration chapter below.

5.4.1.2.3.2.1 Configuration

The main configuration options are offered by the device tree configuration and common Linux interfaces (*ifconfig*, *ethtool*, etc.). All configuration capabilities are presented in this chapter.

5.4.1.2.3.2.1.1 Device Tree Configuration

The Shared Driver has the *fsl,dpa-ethernet-shared* string as compatible string in the device tree. Therefore, the standard structure for the device tree node that specifies a Shared interface should be similar to the below snippet of a B4860QDS device tree node:

```
ethernet@9 {
    compatible = "fsl,b4860-dpa-ethernet-shared", "fsl,dpa-ethernet-shared";
    fsl,fman-mac = <&fmlmac10>;
    fsl,bman-buffer-pools = <&bp17>;
    fsl,qman-frame-queues-rx = <0x5e 1 0x5f 1 0x2000 3>;
    fsl,qman-frame-queues-tx = <0 1 0 1 0x3000 8>;
};
```

Following are the properties of a Shared Driver's device tree node:

- *fsl,fman-mac* - this is the MAC device reference that is in Shared Driver's control;

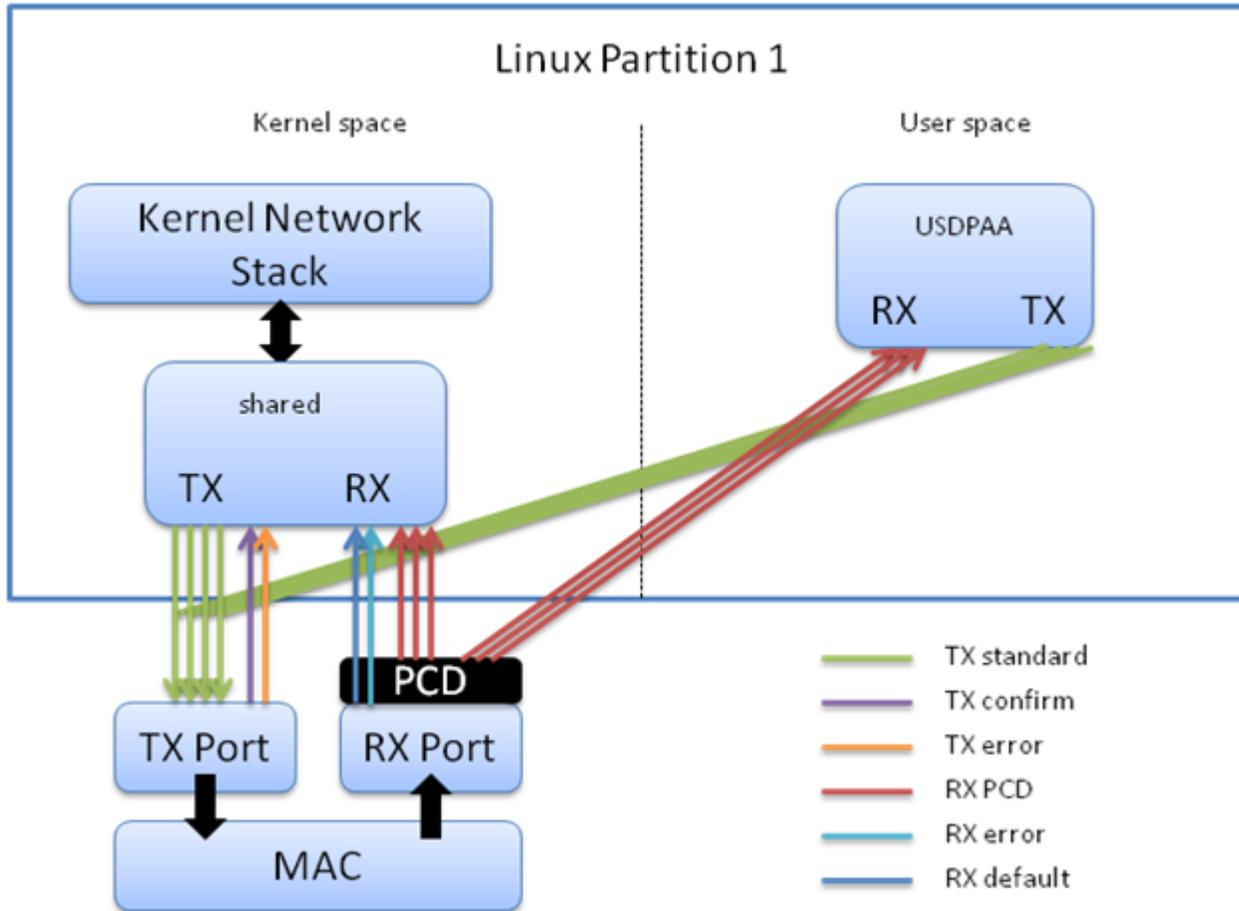
- *fsl,bman-buffer-pools* - as in Macless Driver specification this is a list of buffer pools used by this interface. Since the frames are transmitted between different memory spaces, Shared Driver will use only static defined buffer pools in both use cases;
- *fsl,qman-frame-queues-rx* - a list of base/count pairs of frame queues from which the Shared Driver dequeues frames. Because the Shared Driver has a MAC device in its control, the device tree format of the frame queues is similar to the Private Driver's device tree format. The MAC device must have at least two types of frame queues specified at initialization. These are the error frame queue which is the frame queue on which received erroneous frames will be placed and the default frame queue which is the frame queue where the frame will be placed if no other frame queue is selected for transmission. The next batch represents the PCD queues. These queues will be used by the PCD rules configured by the user. The above device tree example defines one error queue with ID 0x5e, one default queue with ID 0x5f and 3 configurable PCD queues, 0x2000, 0x2001, 0x2002;
- *fsl,qman-frame-queues-tx* - a list of base/count pairs of frame queues used by the Shared Driver send the frames to the MAC device. As in Private DPAA Ethernet Driver, these are the TX error frame queue, TX confirmation frame queue and the standard TX frame queues. In the above example, a value of 0 for TX error and TX confirmation queues enables the dynamic allocation of values for queues' IDs, letting the QMan assign available values. Besides one dynamic TX error and one dynamic TX confirmation queue, 8 standard TX queues will be created, with IDs between 0x3000 and 0x3007. It is recommended to specify up to NR_CPUs frame queues and have a direct mapping between frame queue and CPU. In the above example, B4860QDS has 8 CPUs.

Note 1: The static defined buffer pool has the same representation as defined in the Macless Driver Configuration chapter. As stated above, the Shared Driver will be used in two different scenarios. Each scenario involves entities that have different memory address spaces, but must share the buffer pools' configuration. This is the reason for not using dynamic buffer pools, like in Private DPAA Ethernet Driver.

Following are the scenarios in which Shared Driver can be used:

USDPAAs and Linux stack communicating through a single MAC device, using Shared Driver inside a single Linux partition

USDPAAs applications handle only a certain type of traffic, the rest of the packets being handled by Linux Network Stack. The simplest solution is to have a Shared Driver that knows about the traffic division. In order to split the traffic, PCD rules must be applied on the incoming port. This scenario is presented below.



The device tree configuration should be similar to the one below extracted from a B4860QDS device tree configuration.

```

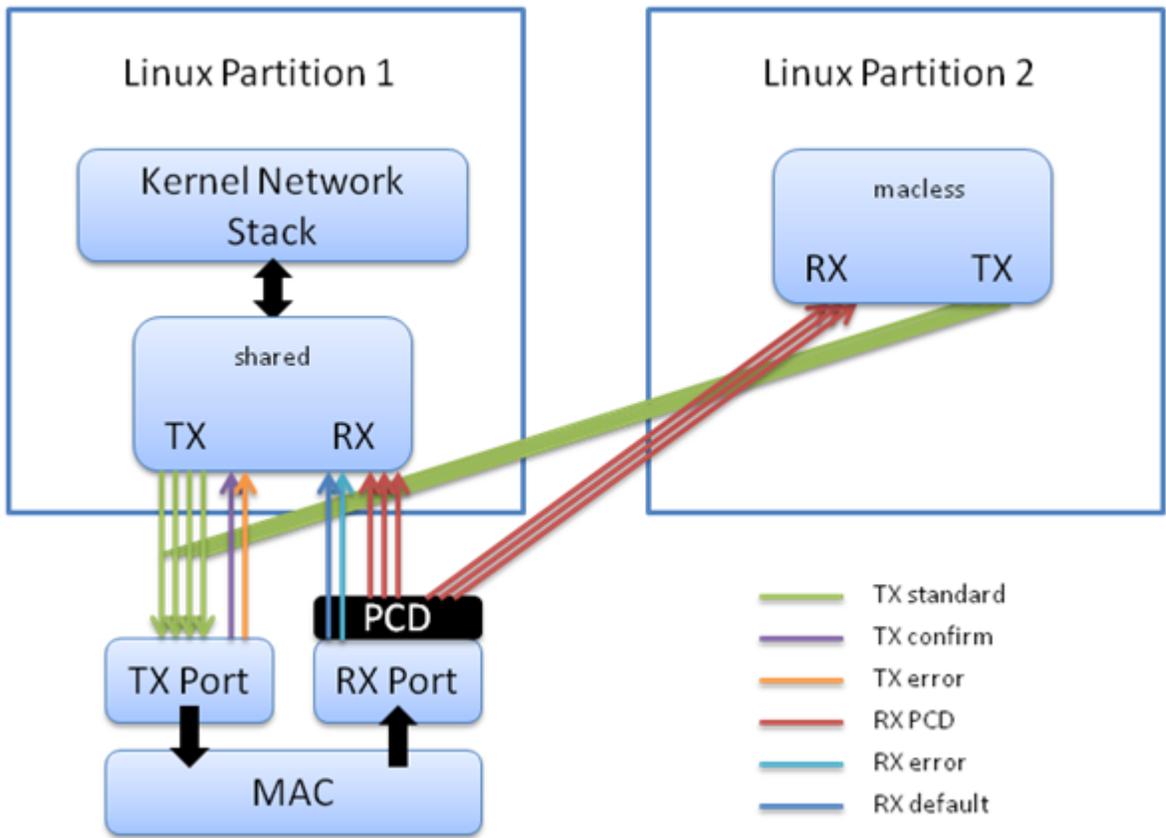
ethernet@9 {
    compatible = "fsl,b4860-dpa-ethernet-shared", "fsl,dpa-ethernet-shared";
    fsl,bman-buffer-pools = <&bp17>;
    fsl,qman-frame-queues-rx = <0x5e 1 0x5f 1 0x2000 3>;
    fsl,qman-frame-queues-tx = <0 1 0 1 0x3000 8>;
    fsl,fman-mac = <&fmlmac10>;
};
bp17: buffer-pool@17 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <17>;
    fsl,bpool-ethernet-cfg = <0 2048 0 1728 0 0>;
    fsl,bpool-thresholds = <0x100 0x300 0x0 0x0>;
};

```

In the above device tree snippet, there are two device tree nodes, one for the Shared Driver and one for the static definition of the buffer pool used for transmission. In this scenario, USDPAA allocates the buffer pool used in the communication, therefore the above device tree nodes will be parsed by USDPAA also. One important configuration is the base address of the buffer pool. Because the communication occurs inside a single Linux partition, there is no need for the buffer pool to advertise its physical address, hence the 0 address representing an invalid physical address. Because the buffers are allocated by USDPAA, these are in the virtual address space of USDPAA. The mapping of the USDPAA's user-space memory space to kernel space is done through kernel *kmap/kunmap* primitives.

Shared communication between two Linux partitions through a single MAC device using a Shared Driver and Macless Driver

In scenarios with multiple Linux partitions managed by a Hypervisor, it is sometimes desired to split one MAC device's traffic between partitions. To achieve this, one partition should use the Shared Driver which will manage the MAC device and the other Linux partition should use the Macless Driver as depicted below.



In scenarios with multiple Linux partitions managed by a Hypervisor, it is sometimes desired to split one MAC device's traffic between partitions. To achieve this, one partition should use the Shared Driver which will manage the MAC device and the other Linux partition should use the Macless Driver as depicted below.

First partition:

```

dpa-ethernet@4 {
    compatible = "fsl,b4860-dpa-ethernet", "fsl,dpa-ethernet-shared";
    fsl,qman-frame-queues-rx = <0x340 1 0x341 1 0x320 8>;
    fsl,qman-frame-queues-tx = <0x342 1 0x343 1 0x300 8>;
    fsl,bman-buffer-pools = <&part1_bp11>;
};
part1_bp11: buffer-pool@11 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <11>;
    fsl,bpool-ethernet-cfg = <0 0x80 0 1728 0 0x80036000>;
    fsl,bpool-ethernet-seeds;
};
    
```

Second partition:

```

dpa-ethernet@20 {
    compatible = "fsl,b4860-dpa-ethernet", "fsl,dpa-ethernet-macless";
    fsl,qman-frame-queues-rx = <0x350 8>;
    fsl,qman-frame-queues-tx = <0x300 8>;
    local-mac-address = [02 00 c0 a8 a1 fe];
};
    
```

```

    fsl,bman-buffer-pools = <&part2_bp11>;
};
part2_bp11: buffer-pool@11 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <11>;
    fsl,bpool-ethernet-cfg = <0 0x80 0 1728 0 0x80036000>;
};

```

In the above configuration the standard 8 TX queues from the first partition are the same as the standard 8 TX queues from the second partition. Both interfaces share one static buffer pool and the same buffer pool definition is used in both partitions. As in other multi-partition scenarios, the buffers are allocated from the physical memory, indicated by the base address parameter. In the above configuration, the base address is 0x80036000. When a frame arrives to one partition, its address should be mapped to partition's kernel address space. The mapping from physical address to each partition's virtual kernel address is done with *ioremap* kernel primitive. The seeding of the buffer pool will be done by the Shared Driver from the first partition as it has *fsl,bpool-ethernet-seeds* in its device tree node.

5.4.1.2.3.2.1.2 Configuration available through Linux interfaces

The Shared Driver has the same data structures as the Private Driver, therefore it has the same configuration capabilities available through standard Linux interfaces (*ethtool*, *ifconfig*, etc.) as the Private Driver.

5.4.1.2.3.2.2 Features

The same transmission primitives are used both by the Shared and the Macless driver. Therefore, as stated in the Macless Driver's *Features* chapter, the Shared Driver is capable of receiving Scatter/Gather frames and is able to offload checksum computation on transmission. On reception, Shared Driver is not able to fetch the checksum computed by FMan.

5.4.1.2.3.3 Proxy DPAA Ethernet Driver

USDPAA applications achieve high speeds for particular types of traffic, bypassing the processing done by the Linux Network Stack. These applications reside in user-space and need kernel-space configuration in order to initialize the necessary hardware modules used along the data path. The configuration of buffer pools and frame queues and initialization of MAC devices used by the USDPAA are delegated to the Proxy DPAA Ethernet Driver. This type of DPAA Ethernet Driver runs once at system startup and does not have a data structure at run-time like the other DPAA Ethernet Drivers. Proxy DPAA Ethernet Driver is used with many different names, *proxy* or *Initialization Manager*, as it was known in the previous SDK release. In this document, it will be referred to as *Proxy Driver*.

5.4.1.2.3.3.1 Configuration

The only configuration for this type of DPAA Ethernet Driver can be made through device tree node configuration.

5.4.1.2.3.3.1.1 Device Tree Configuration

The Proxy Driver has *fsl,dpa-ethernet-init* as compatible string in the device tree. For example, the standard structure for the device tree node that specifies a Proxy interface in a B4860QDS device tree is depicted below:

```

ethernet@8 {
    compatible = "fsl,b4860-dpa-ethernet-init", "fsl,dpa-ethernet-init";
    fsl,fman-mac = <&fmlmac10>;
    fsl,bman-buffer-pools = <&bp7 &bp8 &bp9>;
    fsl,qman-frame-queues-rx = <0x5c 0x1 0x5d 0x1>;
    fsl,qman-frame-queues-tx = <0x7c 0x1 0x7d 0x1>;
};

```

All the above properties are the same as those used by Shared Driver that has a MAC device reference:

- *fsl,fman-mac* - this is the MAC device reference that will be initialized by the Proxy Driver;

- *fsl,bman-buffer-pools* - each port needs a buffer pool to get buffers from. Since it has a MAC device reference, Proxy Driver will need a statically defined buffer pool. This buffer cannot be dynamically created because the Proxy Driver does not know the memory address space of the software entity that will use the initialized infrastructure.
- *fsl,qman-frame-queues-rx* - a list of base/count pairs of frame queues. The device tree format of this property is the same as the format used in Private and Shared Drivers. The MAC device must have at least two queues specified at initialization. These are:
 - error queue which is the queue on which received erroneous frames will be placed;
 - default queue which is the queue where the frame will be placed if none of the PCD queues is selected for transmission;
 - PCD queues are optional. These queues will be used by the PCD rules configured by the user. In the above device tree example no PCD queues are defined.
- *fsl,qman-frame-queues-tx* - a list of base/count pairs of frame queues used by the software entity to send the frames to the MAC device. As in Private or Shared Drivers, these are TX error queue, TX confirmation queue and standard TX queues. In the above example, there are no standard TX queues.

Note 1: The statically defined buffer pools have the same device tree structure as that used for other DPAA Ethernet Drivers.

The classical scenario where a Proxy Driver is used is the one in which it initializes the MAC device on behalf of USDPAA inside a single Linux partition. Another scenario is where the initialization is made on behalf of another software entity or even another Linux partition.

5.4.1.2.3.3.2 Features

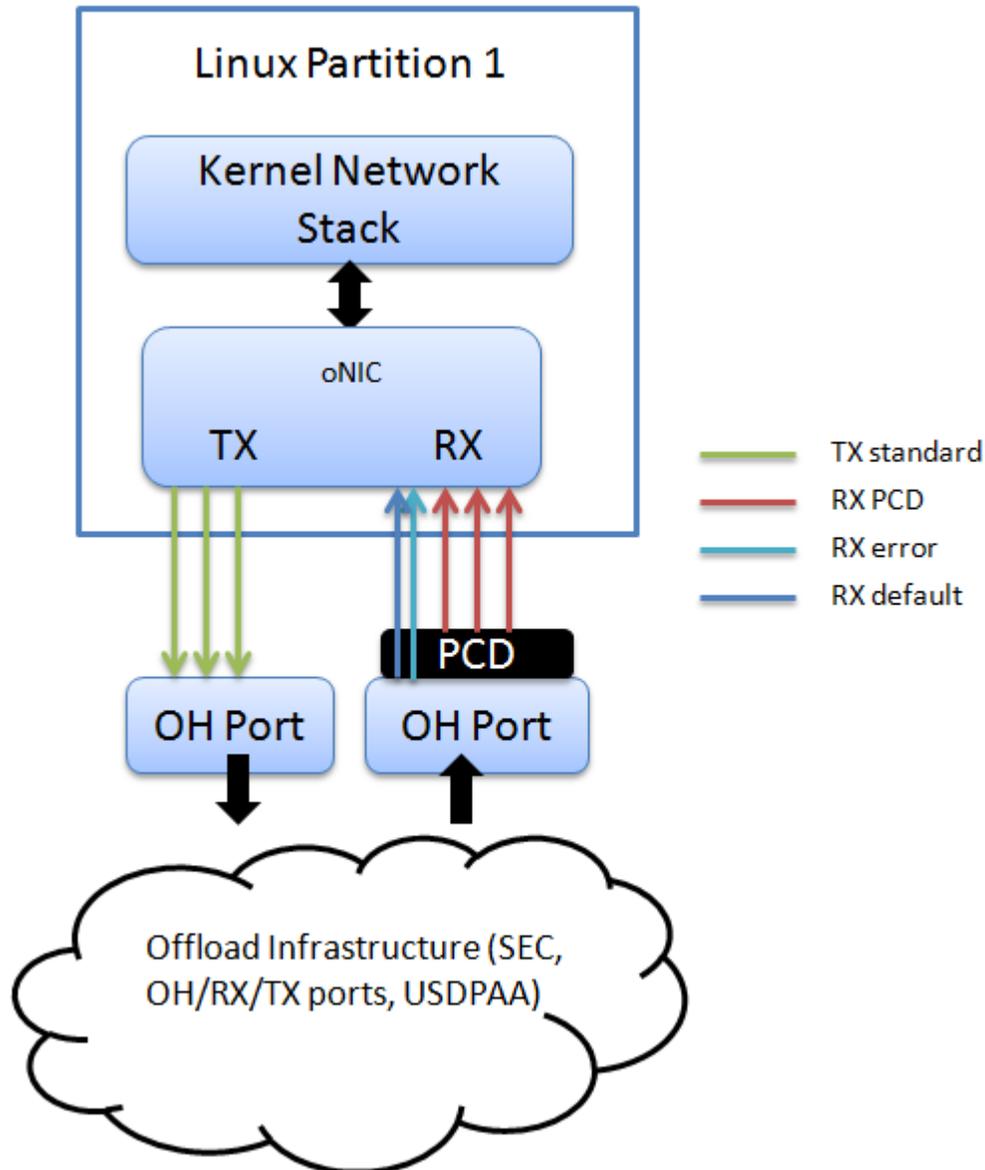
Proxy Driver is used only to initialize some of the hardware modules and does not have advanced features like other DPAA Ethernet Drivers.

5.4.1.2.3.4 Offload NIC Ethernet Driver

Offload NIC Ethernet Driver is a virtual Ethernet driver, similar to Macless. This DPAA Ethernet Driver has CSUM offload and zero-copy, features that cannot be achieved with Macless. This DPAA Ethernet Driver is used in complex scenarios, like IPSec offload or communication with USDPAA where memory space isolation between kernel space and underlying offload architecture (OH ports, SEC, USDPAA) space is needed. Some of these scenarios are presented in the Configuration chapter.

This type of DPAA Ethernet Driver needs two OH ports with VSP capabilities enabled in order to achieve zero-copy.

Offload NIC Ethernet Driver will be referred in this documentation as "oNIC".



5.4.1.2.3.4.1 Configuration

The main configuration options are offered, like in any other embedded Linux Ethernet driver, by the device tree configuration and the common interface offered by the Linux kernel (ifconfig, ethtool, etc.). All configuration capabilities are presented in this chapter.

5.4.1.2.3.4.1.1 Device tree configuration

oNIC Ethernet Driver has *fsl,dpa-ethernet-generic* as compatible string in the device tree. For example, the standard structure for the device tree node that specifies an oNIC interface in a B4860QDS device tree looks like:

```

ethernet@16 {
    compatible = "fsl,b4860-dpa-ethernet-generic", "fsl,dpa-ethernet-generic";
    fsl,qman-frame-queues-tx = <4000 8x8>;
    fsl,qman-frame-queues-rx = <4008 8x8>;
    fsl,oh-ports = <&oh1 &oh2>;
    fsl,disable_buff_dealloc;
    local-mac-address = [00 11 22 33 44 55];
};

```

The properties that the oNIC device tree node can have are:

- *fsl,qman-frame-queues-tx* - a list of base/count pairs of frame queues that forwards the frame buffers generated by oNIC to the Tx O/H port; it is recommended to have NR_CPUS Tx frame queues;
- *fsl,qman-frame-queues-rx* - a list of base/count pairs of frame queues called PCD queues that fetch the frames from the Rx O/H port to oNIC; besides these queues, oNIC uses the default and the error queue of the Rx O/H port; the PCD queues can be used in PCD schemes applied on the Rx O/H port.
- *fsl,oh-ports* – links to O/H port device tree representation. The first value is a reference to the Rx O/H port used on ingress path, the second value is a reference to the Tx O/H port used on egress path. The device tree representation for the O/H port is described below;
- *fsl,disable_buff_dealloc* - in some scenarios the TX OH port cannot enable VSP because of the OH port limitation (O/H port cannot have VSP capability and IP Fragmentation enabled in the same time), therefore another hardware module along the TX path should release the buffers generated by oNIC into the draining buffer pool. With this flag, oNIC will configure the TX OH port to NOT release the buffers into the draining buffer pool. If other hardware modules do not release the buffers into the draining pool the device will leak available memory. This is an optional attribute.
- *local-mac-address* - this is a virtual L2 address used to identify the driver in the Linux kernel network stack.

Note 1: Like Macless DPAA Ethernet Driver, oNIC does not have a MAC device to control, therefore the “fsl,fman-mac” property is missing from the device tree specification.

oNIC does not have a buffer pool specified in its device tree representation. It will use the default buffer pools specified in the O/H port device tree node. The O/H port device tree representation looks like:

```
oh2: dpa-fman0-oh@2 {
    compatible = "fsl,dpa-oh";
    fsl,bman-buffer-pools = <&bp1 &bp2 &bp3 &bp4>;
    fsl,qman-frame-queues-oh = <110 1 111 1>;
    fsl,fman-oh-port = <&fman0_oh2>;
};
```

The properties of the O/H port device tree node can have are:

- *fsl,bman-buffer-pools* - the default buffer pools managed by oNIC; up to four static buffer pools can be specified. If this O/H port has VSPs, these buffer pools will be used in the default VSP. This property is parsed by oNIC that will initialize the default buffer pools. No other entities should try to initialize these buffer pools;
- *fsl,qman-frame-queues-oh* - the default egress queues. The default queues are represented by one default frame queue and one error frame queue. This property will be parsed by oNIC that will initialize both queues and use them in case PCD frame queues are not defined;
- *fsl,fman-oh-port* - reference to FMan port device tree representation; this field is mandatory for Offline Port Driver, but it is not used by oNIC;

To activate VSP capability on a port, the user will have to configure the chosen node in the device tree. One valid entry looks like:

```
fman0_oh2-extd-args {
    cell-index = <0x1>;
    compatible = "fsl,fman-port-op-extended-args";
    vsp-window = <0x8 0x0>;
};
```

The most important property is:

- *vsp-window* - the number of VSPs that this port can have and the default VSP (starting VSP index). In the above example 8 VSPs can be added to the O/H port (with ids 0, 1, ..., 8).

The device tree node of the O/H port use statically defined buffer pools. oNIC will parse the device tree representation of the default buffer pools of the Rx O/H port and will seed them. The device tree representation of the buffer pool is:

```
bp7: buffer-pool@7 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <7>;
    fsl,bpool-ethernet-cfg = <0 256 0 192 0 0>;
    fsl,bpool-thresholds = <0x400 0xc00 0x0 0x0>;
};
```

oNIC parses *fsl,bpid* and *fsl,bpool-ethernet-cfg* properties to initialize its internal buffer pools structures. As in other DPAA Ethernet Drivers, *fsl,bpool-ethernet-cfg* property has the following meaning:

```
fsl,bpool-ethernet-cfg = <count size base_address>;
```

- *count* - represents the number of buffers from the buffer pool;
- *size* - buffer size;
- *base_address* - this value is ignored by oNIC.

Note 2: oNIC will seed its configured buffer pools when the Ethernet interface is raised up. Because of this the *fsl,bpool-ethernet-seeds* property is ignored.

Note 3: All the buffers are allocated in the kernel memory space, therefore the *base_address* value is ignored.

The example above declares a buffer pool with buffer pool ID 7, and describes a pool with 256 192-byte buffers, occupying the memory in the kernel space. The *base_address* with <0 0> value is ignored. It should be noted that the size of those parameters should be set by the root node's *#address-cells* and *#size-cells* properties.

5.4.1.2.3.4.1.2 Configuration available through Linux interfaces

Because oNIC does not manage a MAC device, some *ethtool* and *ifconfig* options will not be available. All Layer 3 configurations, like setting an IP address, are generally available.

5.4.1.2.3.4.2 Features

This chapter describes the features of the oNIC Ethernet Driver, their configuration options, fine-tuning and known limitations.

Hardware Checksum

The OH ports are able to validate and compute L3 and/or L4 checksums for certain protocols (mainly TCP/IP and UDP/IP). Because oNIC uses OH ports as communication points with the lower architecture, CSUM validation and computation are supported.

Zero Copy

oNIC, representing kernel space, works in memory "isolation". This capability is achieved through OH ports with VSP capability enabled, which are able to copy the frames from one memory space to another. oNIC code and the underlying offload architecture code are not aware of the existence of another memory space, therefore simplifying the communication procedures.

Scatter/Gather

The current implementation supports DPAA Scatter/Gather only on the TX path.

5.4.1.2.4 Offline Parsing Port Driver

Offline Parsing/Host Command Port, also known as *O/H Port* or simply *O/H*, is an FMan hardware module that is capable of multiplexing and de-multiplexing network traffic inside DPAA. Its behavior and programming model is similar to an "online" FMan port, supporting Parse-Classify-Distribute (PCD) function and buffer copy from one buffer pool to another. It is useful in complex DPAA scenarios, where plenty of hardware offload is desired, like IPsec or TCP fragmentation and reassembly.

Similar to most DPAA hardware modules, a Linux kernel driver for O/H Ports is needed for initialization and configuration. In this document, the Offline Parsing Port Driver will be referred to as *Offline Port Driver*.

5.4.1.2.4.1 Configuration

Like other DPAA Ethernet Drivers, the Offline Port Driver is a standard Linux platform device driver, whose configuration is available through device tree. Also, the support for Offline Port Driver is enabled through a kernel Kconfig option.

5.4.1.2.4.1.1 Kconfig Option

The Offline Port Driver is enabled in the Kbuild Linux configuration by default. It can be toggled via the following menuconfig option:

```
Device Drivers
---> Network device support
    ---> Ethernet driver support
        ---> Freescale devices
            ---> DPAA Ethernet
                ---> Offline Ports support
```

5.4.1.2.4.1.2 Device Tree Configuration

The Offline Port Driver has *fsl,dpa-oh* as compatible string in the device tree. For example, the standard structure for the device tree node that specifies an Offline Port Driver in a B4860QDS device tree is depicted below:

```
dpa-fman0-oh@2 {
    compatible = "fsl,dpa-oh";
    fsl,bman-buffer-pools = <&bp10>;
    fsl,qman-frame-queues-oh = <0x6e 0x1 0x6f 0x1>;
    fsl,qman-frame-queues-tx = <0x70 0x5>;
    fsl,qman-frame-queues-ingress = <base_id1 count1 ... base_idn countn>;
    fsl,qman-frame-queues-egress = <base_id1 count1 ... base_idn countn>;
    fsl,qman-channel-ids-egress = <channel_id1 ... channel_idn>;
    fsl,fman-oh-port = <&fman0_oh2>;
};
```

This device tree node resides inside *fsl,dpaa* device tree node, near DPAA Ethernet Driver device tree node specification.

Historically the offline port driver did not initialize the frame queues that entered and exited the OH port and relied on software components (Ethernet driver, USDPAA, other kernel modules) to initialize those queues. The offline port driver added capabilities to initialize ingress queues (queues that enter the Offline Port) simplifying the work for the Ethernet driver, but maintaining the same complexity in USDPAA and/or other kernel modules. Full capability for initializing both ingress and egress queues has been added, eliminating the dependency on USDPAA and/or other kernel modules. With it, complex offload architectures that have OH ports can be largely initialized by the offline port driver.

The additional device tree attributes, *qman-frame-queues-ingress*, *qman-frame-queues-egress* and *qman-channel-ids-egress* are optional and do not interfere with existing code. Therefore, backward compatibility is maintained.

Following are the properties that could be used within Offline Port Driver's device tree node:

- *fsl,bman-buffer-pools* - a list of buffer pools used by this O/H Port. The O/H Port will use these statically defined buffer pools in case it will do IP Fragmentation or buffer copy from one buffer pool to another. For IP Fragmentation, one incoming frame is divided into multiple frames. In order to copy data from a buffer pool to another, VSP (Virtual Storage Profile) properties must be configured on this port and the destination buffer pool is initialized and managed by the destination software entity. Therefore, these buffer pools are not initialized or managed by the Offline Port Driver. In some scenarios, this can be done by one of DPAA Ethernet Drivers. If the O/H Port does not use IP Fragmentation or VSP capabilities, this attribute will not be used;

- *fsl,qman-frame-queues-oh* - a list of base/count pairs of frame queues through which the frames are transmitted from the O/H Port to the next hardware or software module. These queues are not initialized, but are solely used as ID references for O/H Port initialization. This is necessary because the initialization of an FMan Port (online or offline) requires a TX error and a TX default frame queues. These frame queues must be initialized by other software module. This device tree property is mandatory;
- *fsl,qman-frame-queues-tx* - deprecated
- *fsl,qman-frame-queues-ingress* - a base/count pair of frame queues to be initialized by the Offline Port Driver that will fetch the frames from the hardware or software entity to the O/H Port.
- *fsl,qman-frame-queues-egress* - a list of base/count pairs of frame queues through which the frames are transmitted from the O/H Port to the next hardware or software module. These queues are connected only to DC portals therefore the property below (*qman-channel-ids-egress*) is mandatory. Frame queues connected to SW portals should be created from another software entity. Also, if a frame queue uses different initialization parameters it should be created by another software entity.
- *fsl,qman-channel-ids-egress* - a list of channel ids used together with "*fsl,qman-frame-queues-egress*" option to configure the egress frame queues.
- *fsl,fman-oh-port* - this is the reference to the O/H Port device tree node and has the same meaning as *fsl,fman-mac* from DPAA Ethernet Driver device tree specification. This attribute is mandatory.

Note 1: It should be noted that the Offline Port Driver does not initialize frame queues specified through *fsl,qman-frame-queues-oh* attribute. When the port is probed, the O/H Port is enabled and configured with the values from the device tree, but no action is taken on the declared frame queues.

Note 2: The statically defined buffer pool should be declared in a similar way as for the DPAA Ethernet Drivers.

The O/H Ports can be used in multiple scenarios as nodes that fetch traffic from multiple sources or as intermediate nodes that multiplex incoming traffic to multiple destinations.

5.4.1.2.4.2 Features

The Offline Port Driver is an initialization driver for the O/H Ports. This chapter presents only the Offline Port Driver initialization capabilities, not the hardware features offered by the O/H Ports.

TX queues initialization

Before SDK 1.5, no frame queue referenced by the Offline Port Driver's device tree node was initialized. It was other driver's job to initialize the frame queues used by the O/H Port. As of SDK 1.5, *fsl,qman-frame-queues-tx* and *fsl,qman-channel-id* attributes were introduced. The frame queues referenced by the above mentioned *fsl,qman-frame-queues-tx* attribute are managed by the Offline Port Driver, decreasing thus the complexity of other drivers.

5.4.1.2.5 Link Management

The FMan has two types of ethernet controllers: 1G and 10G. The two types of ethernet conform to different standards (802.3 Clause 22 and Clause 45, respectively) for link management, so there are two corresponding types of MDIO controller for those standards. Each ethernet controller has its own MDIO device, however only one of each type are pinned out on currently shipping parts (10/2011). On P4080, for example, only the MDIO on FM1's first 1G MAC is pinned out, as well as the MDIO on FM1's 10G MAC. This is true even if FM1's first 1G MAC or 10G MAC is disabled.

On the 1G MACs, the MDIO controllers serve a second purpose -- configuring the SERDES link for SGMII between the MAC and the external PHY. This is done via management commands to an on-chip "TBI" PHY. This PHY is configured to sit at the address written to the TBIPA register in the MAC, and all MDIO transactions from that MAC to that address will be intercepted by the TBI PHY. This means that the setting of this register on the first 1G MAC is very relevant to system architects, as the address must not conflict with the address of an external PHY, or that PHY will not be reachable by MDIO management commands.

5.4.1.2.5.1 Device Tree

The MDIO nodes are described in the device tree, and look like this:

```
mdio0: mdio@e1120 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl,fman-mdio";
    reg = <0xe1120 0xee0>;
    interrupts = <100 1 0 0>;
    ethernet-phy@0 {
        reg = <0x0>;
    };
    tbi0: tbi-phy@8 {
        reg = <0x8>;
        device_type = "tbi-phy";
    };
};
```

This is one of the 1G MDIO nodes for a P4080 (Not the DS, but more on that later). It specifies that there is a PHY at address 0, and the TBI PHY will sit at address 8. Note that it is the first MAC, which is the only one whose pins are connected outside the chip. The other 7 MACs would have a similar node, but with only a TBI PHY in each. This is how one would set up PHYs under a standard MDIO bus topology; the first MAC connects to all of the PHYs (up to 31 of them), and also has its assigned internal TBI PHY.

In order to associate an ethernet controller with a PHY, one uses the phy-handle property

5.4.1.2.5.2 Bootargs

```
fsl_fm_max_frm
```

Configure this to set the L2 Maximum Frame Size. This influences the Frame Manager FIFO size resources.

```
fsl_fm_rx_extra_headroom
```

Configure this to tell the Frame Manager to reserve some extra space at the beginning of a data buffer on the receive path, before Internal Context fields are copied. This is in addition to the private data area already reserved for driver internal use. The option does not affect in any way the layout of transmitted buffers. The default value (64bytes) offers best performance for the case when forwarded frames are being encapsulated (e.g. IPSec). For plain forwarding or termination cases, a value of zero is recommended for optimum performance.

The current size of the buffers in USDPAA dts files is 1728bytes ($\text{odd_cache_line} * \text{cache_line_size}$) which is calculated according to the default value (64bytes) of the `dpa_extra_headroom`. If the `dpa_extra_headroom` is set to 0 then the buffer size will be 1600bytes (also multiple of odd cache line). For any extra headroom values ranging between 1 and 128bytes the buffer size is 1728bytes. For values ranging between 129bytes and 256bytes, the buffer size is 1856bytes and so on.

5.4.1.2.5.3 Muxed MDIO

The Development Systems for the QorIQ product lines do not have a standard MDIO topology. In order to support the variety of configurations that are possible under the QorIQ product line, the PHYs often reside on riser cards, many of which use the same addresses; this means that it is not possible to keep all of the PHYs on the same bus. To work around this problem, the MDIO buses have been muxed so that, at any one time, only a subset of the PHYs will be connected to the MDIO buses. In order to send an MDIO transaction to a PHY, it is therefore necessary to first modify the MUX. This is done via a set of

"virtual" MDIO interfaces. Each MUX setting is considered a separate bus, each with its own PHYs. That means that the P4080DS's MDIO node looks like this:

```
mdio0: mdio@e1120 {
    gpios = <&gpio0 0 0
           &gpio0 1 0>;
    tbi0: tbi-phy@8 {
        reg = <0x8>;
        device_type = "tbi-phy";
    };
    p4080mdio0: p4080ds-mdio0 {
        #address-cells = <1>;
        #size-cells = <0>;
        compatible = "fsl,p4080ds-mdio";
        fsl,mdio-handle = <&mdio0>;
        fsl,muxval = <0>;
        phyrgmii: ethernet-phy@0 {
            reg = <0x0>;
        };
    };
};
```

The bus now has a subnode, "p4080ds-mdio0", which represents the bus topology when the MUX is set to 0. That bus connects to the RGMII PHY at address 0. Other buses connect to the SGMII PHYs in the 3 riser card slots. It's important to note that it is not at all required to use a muxed MDIO bus on QorIQ products; this is only a solution to the problem of trying to access PHYs which can be in multiple places. The other important thing to note is that U-Boot modifies the device tree. In the example of the P4080DS, U-Boot uses the RCW to determine which interfaces are enabled, and which PHYs those interfaces are connected to. Therefore the connections in the dts file may not be the final arrangements of the PHYs.

5.4.1.2.6 Debugging

This chapter describes the debugging capabilities of the DPAA Ethernet driver.

5.4.1.2.6.1 Ethtool support

Various counters and statistics are exported through ethtool such as the number of interrupts per core, the number of frames per core, the number of available buffers, congestion detection, etc.

Following is an example of an ethtool output:

```
root@ls1043ardb:~# ethtool -S fml-mac1
NIC statistics:
    interrupts [CPU 0]: 1
    interrupts [CPU 1]: 1
    interrupts [CPU 2]: 2
    interrupts [CPU 3]: 2
    interrupts [TOTAL]: 6
    rx packets [CPU 0]: 0
    rx packets [CPU 1]: 0
    rx packets [CPU 2]: 0
    rx packets [CPU 3]: 0
    rx packets [TOTAL]: 0
    tx packets [CPU 0]: 0
    tx packets [CPU 1]: 0
    tx packets [CPU 2]: 6
    tx packets [CPU 3]: 0
    tx packets [TOTAL]: 6
    tx recycled [CPU 0]: 0
```

```

tx recycled [CPU 1]: 0
tx recycled [CPU 2]: 0
tx recycled [CPU 3]: 0
tx recycled [TOTAL]: 0
tx confirm [CPU 0]: 1
tx confirm [CPU 1]: 1
tx confirm [CPU 2]: 2
tx confirm [CPU 3]: 2
tx confirm [TOTAL]: 6
tx S/G [CPU 0]: 0
tx S/G [CPU 1]: 0
tx S/G [CPU 2]: 0
tx S/G [CPU 3]: 0
tx S/G [TOTAL]: 0
rx S/G [CPU 0]: 0
rx S/G [CPU 1]: 0
rx S/G [CPU 2]: 0
rx S/G [CPU 3]: 0
rx S/G [TOTAL]: 0
tx error [CPU 0]: 0
tx error [CPU 1]: 0
tx error [CPU 2]: 0
tx error [CPU 3]: 0
tx error [TOTAL]: 0
rx error [CPU 0]: 0
rx error [CPU 1]: 0
rx error [CPU 2]: 0
rx error [CPU 3]: 0
rx error [TOTAL]: 0
bp count [CPU 0]: 128
bp count [CPU 1]: 128
bp count [CPU 2]: 128
bp count [CPU 3]: 128
bp count [TOTAL]: 512
rx dma error: 0
rx frame physical error: 0
rx frame size error: 0
rx header error: 0
rx csum error: 0
qman cg_tdrop: 0
qman wred: 0
qman error cond: 0
qman early window: 0
qman late window: 0
qman fq tdrop: 0
qman fq retired: 0
qman orp disabled: 0
congestion time (ms): 0
entered congestion: 0
congested (0/1): 0

```

5.4.1.2.6.2 Read/Write of FMan Registers

Most of the FMan configuration registers are mapped into the system memory space. Efficient debugging and testing can be done by making read/write operations on the registers through specialized tools. For example, the number of pause frames received on a particular MAC device can be computed summing the base relative address of every component:

```

0xffe000000 (the address of the SoC) +
0x400000 (FMan 1) +

```

```

0xe8000 (MAC 5) +
  0x234 (RX pause frames) =
-----
0xffe4e8234

```

A memory print of the 0xffe4e8234 address will display the number of pause frames received by the fifth MAC device from the first FMan on a P4080DS platform.

The entire memory map for all mapped registers of the DPAA hardware components can be found in each platform's Reference Manual.

5.4.1.2.6.3 Sysfs support

To enable Sysfs in the Linux kernel one must set the CONFIG_SYSFS option in Kconfig. The DPAA Ethernet Driver exports a series of information in Sysfs such as the buffer pool IDs, the frame queue IDs used by the interface and the MAC registers as shown in the following examples:

```

root@p4080ds:~# cat /sys/devices/fsl,dpaa.16/ethernet.18/net/fm2-gb0/bpids
32
root@p4080ds:~# cat /sys/devices/fsl,dpaa.16/ethernet.17/net/fm1-gb1/fqids
Rx error: 262
Rx default: 263
Rx PCD: 14464 - 14591
Tx confirmation (mq): 264 - 271
Tx error: 272
Tx default confirmation: 273
Tx: 274 - 281

```

5.4.1.2.7 Adding support for DPAA Ethernet in Topaz Hypervisor

We start from the assumption that hv.dts file for the platform has already been created starting from an existing platform. The practice is to place the device tree under the following folder hierarchy: hv-cfg/"platform_name"/"RCW_name". It means that the RCW gives the number and types of ports that will be added in hv.dts. We will consider T1040RDB hv.dts as a reference throughout this guide.

Following are the steps to be followed in order to add DPAA ports (private, shared-mac and macless) in hv.dts:

1. Add bman-portal and qman-portal nodes under "part 1" node. E.g.:

```

bman-portal@0 {
    device = "/bman-portals/bman-portal@0";
};

qman-portal@0 {
    device = "/qman-portals@ff600000/qman-portal@0";

    stash-mem {
        liodn-index = <1>;
        dma-window = <&dw_linux1>;
        operation-mapping = <0>;
        stash-dest = <3>;
    };

    stash-dqrr {
        liodn-index = <0>;
        dma-window = <&dw_dqrr_qportal0>;
    };
};

```

```
        operation-mapping = <0>;
        stash-dest = <3>;
    };
};
```

Make sure dma-window "dw_dqrr_qportal0" exists under "dma-windows" node:

```
// DMA window for stash_dqrr for qman-portal0
dw_dqrr_qportal0: window3 {
    compatible = "dma-window";
    guest-addr = <0xf 0xf6000000>;
    size = <0 0x4000>;
};
```

All bman-portal and qman-portal nodes must be added except at least one which should be added to the second partition. For T1040RDB, bman-portal@1c000 and qman-portal@1c000 were added in the second partition. The list of available bman-portal and qman-portal nodes for T1040RDB can be retrieved from the following dtsi file included by the platform device tree: arch/powerpc/boot/dts/fsl/t1040si-post.dtsi bman-portals are located under bportals node and qman-portals under qportals node.

2. For the current RCW that is used, add the appropriate dpaa ethernet node under "part 1" parent node. For T1040RDB the following ports are added:

```
// FMAN0 RGMII -- ethernet 3 assigned to this partition
dpa-ethernet@3 {
    device = "/fsl,dpaa/ethernet@3";
};

// FMAN0 RGMII -- ethernet 4 assigned to this partition
dpa-ethernet@4 {
    device = "/fsl,dpaa/ethernet@4";
};
```

The full list of available ethernet ports can be retrieved from the platform device tree: arch/powerpc/boot/dts/t1040rdb.dts

3. bman, qman, fman0 and fman1 (in case there are two FMANs) nodes must be also added under "part 1" parent node. See T1040RDB example.
4. fman0 and fman1 (if applicable) must be added under "portal-devices" node. See T1040RDB example.
5. Under "node-update" node make sure you define at least two buffer pools for macless and shared-mac interfaces. While here, make sure gpma1 is defined and guest-addr reserves the inter-partition memory area. Also check that dw_linux1 and dw_linux2 both have sub-window nodes with the same memory area region matching the inter-partition area. See T1040RDB example.
6. Make sure "fsl,dpaa" node exists under "node-update" node and check or create a macless and a shared-mac interface. The compatible string for the "fsl,dpaa" node must be: compatible = "fsl,t1040-dpaa", "fsl,dpaa"; replace t1040 with the name of the platform you want to add. The compatible string for the macless interface must be: compatible = "fsl,t1040-dpa-ethernet", "fsl,dpa-ethernet-macless"; The compatible string for the shared-mac interface must be: compatible = "fsl,t1040-dpa-ethernet", "fsl,dpa-ethernet-shared"; Make sure the macless node includes the local-mac-address identifier. See T1040RDB example.

7. Make sure "fsl,dpaa" node exists under "node-update-phandle" node. Here the buffer pools are associated with macless and shared-mac interfaces. See T1040RDB example.
8. Under "part 1" node check or add the following nodes: bman-bpids, qman-fqids@0, qman-fqids@1, qman-pools, qman-cgrids. The values used by these nodes can be retrieved from arch/powerpc/boot/dts/fsl/qoriq-dpaa-res3.dtsi. For T1040RDB two partitions were created therefore the resources, bpids, fqids, etc were splitted among the two partitions. In case more partitions are created the resources must be splitted accordingly. See T1040RDB example.
9. Repeat steps 3-8 for "part 2" node in case of 2 partitions scenario. The same steps must be followed in case more than 2 partitions are added. Exception for step 6: Two macless nodes must be added under "fsl,dpaa" node. The first node is the "pair" of the macless interface in the first partition and the second is the interface connected to the shared-mac port (the port shared with the first partition). See T1040RDB example.

5.4.1.2.8 MACsec

Introduction

This chapter provides information about the MACsec kernel module and user space API that can be used to configure and control the MACsec hardware block inside the Frame Manager. The Quick Start Guide lists the steps that need to be taken in order to enable MACsec after integrating the API into the hostapd and wpa_supplicant applications. The API Reference section details the user space MACsec API and gives examples on how to use it.

Intended Audience

This chapter is intended for software developers and architects who want to develop software applications that implement the 802.1X and 802.1AE (MACsec) protocols.

5.4.1.2.8.1 MACsec User Space API Reference

The following API is implemented in the hostapd and wpa_supplicant user space applications and can be reached by including the src/drivers/dpaa_utils_macsec.h header.

MACsec API data structures

1. enable_macsec_t

Member's description:

- if_name - char* that holds the name of the interface on which MACsec is to be enabled
- if_name_length - size_t that holds the length of the if_name field, including the terminating null character
- config_unknown_sci_treatment - boolean that selects between the default value and the one in unknown_sci_treatment
- unknown_sci_treatment - enum that selects how frames received with unknown SCI are treated (default is DISCARD_BOTH - discarded on both the controlled and uncontrolled ports)
- config_invalid_tag_treatment - boolean that selects between the default value and the one in deliver_uncontrolled
- deliver_uncontrolled - boolean that selects if frames received with an invalid SecTAG or a zero PN value should be delivered on the uncontrolled port (default is FALSE)
- config_kay_frame_treatment - boolean that selects between the default value and the one in discard_uncontrolled
- discard_uncontrolled - boolean that selects if frames received with the Encryption (E) bit set and the Changed Text (C) bit clear (reserved for use by the KaY) should be discarded on the uncontrolled port (default is FALSE)
- config_untag_treatment - boolean that selects between the default value and the one in untag_treatment
- untag_treatment - enum that selects how frames received without the MAC SecTAG are treated (default is UNTAG_DELIVER_UNCTRL_DISCARD_CTRL)
- config_pn_exhaustion_threshold - boolean that selects between the default value and the one in pn_threshold
- pn_threshold - u32 that sets the TX PN exhaustion threshold (default is 0xffffffff)

- `config_keys_unreadable` - boolean that selects if the RX and TX keys and hash values should be unreadable (default is FALSE)
- `config_sectag_without_sci` - boolean that selects if the maximum frame length should not consider the SCI's size in the SecTAG (default is FALSE)
- `config_exception` - boolean that selects between the default values and the ones in `enable_exception` and `exception`
- `enable_exception` - boolean that selects if the exception mask set by the exception field should be enabled or not
- `exception` - enum that defines the mask of the exception that is to be enabled or disabled (default is `SINGLE_BIT_ECC` | `MULTI_BIT_ECC` - both masks are enabled)

2. `enable_secy_t`

Member's description:

- `macsec_id` - int value returned by the `dpa_macsec_enable()` call that identifies one MACsec instance per interface
- `sci` - 64bit value made from the MAC address of the interface on which MACsec has been enabled, concatenated with a Port Identifier
- `config_insertion_mode` - boolean that selects between the default value and the one in `sci_insertion_mode`
- `sci_insertion_mode` - enum that selects if the SCI will be included in the SecTag or not (default is `SCI_INSERTION_MODE_EXPLICIT_SECTAG` - the SCI is always included)
- `config_protect_frames` - boolean that selects between the default value and the one in `protect_frames`
- `protect_frames` - boolean that selects if the packet will be encapsulated with MACsec header (default is TRUE)
- `config_replay_window` - boolean that selects between the default values and the ones in `replay_window` and `replay_protect`
- `replay_protect` - boolean that selects if the replay protection algorithm is enabled or not (default is FALSE)
- `replay_window` - unsigned integer that represents the replay window dimension (default is 0)
- `config_validation_mode` - boolean that selects between the default value and the one in `validate_frames`
- `validate_frames` - enum that selects the behaviour of MACsec from frames validation point of view (default is `VALID_FRAME_BEHAVIOR_STRICT` - strictly filter out invalid frames)
- `config_confidentiality` - boolean that selects between the default values and the ones in `confidentiality_enable` and `confidentiality_offset`
- `confidentiality_enable` - boolean that selects if the packages will be encrypted or not (default is FALSE)
- `confidentiality_offset` - unsigned integer that represents the offset from which the data will be encrypted (default is 0)
- `config_point_to_point` - boolean that selects if MACsec must be configured only for point-to-point communication (default is FALSE)
- `config_exception` - boolean that selects between the default values and the ones in `enable_exception` and `exception`
- `enable_exception` - boolean that selects if the exception mask set by the exception field should be enabled or not
- `exception` - enum that defines the mask of the exception that is to be enabled or disabled (default is `FRAME_DISCARDED`)
- `config_event` - boolean that selects between the default values and the ones in `enable_event` and `event`
- `enable_event` - boolean that selects if the event mask set by the event field should be enabled or not
- `event` - enum that defines the mask of the event that is to be enabled or disabled (default is `NEXT_PN`)

MACsec API functions

1. dpa_macsec_enable

```
int dpa_macsec_enable(enable_macsec_t en_macsec)
```

Description:

- used to configure and enable MACsec on a certain interface

Parameters:

- en_macsec - MACsec data structure including the name of the interface to enable MACsec on and several configuration options

Return:

- returns the macsec_id that will be used in further calls of the API's functions
- returns a negative value in case something went wrong

2. dpa_macsec_secy_en

```
int dpa_macsec_secy_en(enable_secy_t enable_secy)
```

Description:

- used for configuring and creating SecY; also creates a corresponding TxSc (secure channel for TX)
- sets the default values for cipher suite, SCI insertion mode, protect frames, replay protection, replay window dimension, confidentiality, confidentiality offset and point-to-point connection, as IEEE802.1AE-2006 states

Parameters:

- enable_secy - SecY data structure

Return:

- returns the sc_id allocated by MACsec (a value from 15 to 0, or simply 0, if the setup is point-to-point) that uniquely identifies the SecY
- returns a negative value in case something went wrong

3. dpa_macsec_secy_create_tx_sa

```
int dpa_macsec_secy_create_tx_sa(int macsec_id, u8 an, u8 *sak, u32 sak_len)
```

Description:

- used for creating a TxSa (secure association for TX)

Parameters:

- macsec_id - MACsec interface identifier
- an - association number
- sak - secure association key (the key used for encryption)
- sak_len - length of the key (must be at most 32)

Return:

- returns 0 on success or a negative value in case something went wrong

4. dpa_macsec_secy_activate_tx_sa

```
int dpa_macsec_secy_activate_tx_sa(int macsec_id, u8 an)
```

Description:

- used for activating the secure association for TX channel

Parameters:

- macsec_id - MACsec interface identifier
- an - association number

Return:

- returns 0 on success or a negative value in case something went wrong

5. dpa_macsec_secy_create_rx_sc

```
int dpa_macsec_secy_create_rx_sc(int macsec_id, u64 sci)
```

Description:

- creates a RxSc (secure channel for RX)

Parameters:

- macsec_id - MACsec interface identifier
- sci - a 64 bit value made of the MAC address of the destination(48 bits) and a port number(16 bits)

Return:

- returns the rx_sc_id allocated by MACsec (a value from 15 to 0, or simply 0, if the setup is point-to-point) that uniquely identifies the peer to whom we are discussing MACsec
- returns a negative value in case something went wrong

6. dpa_macsec_secy_create_rx_sa

```
int dpa_macsec_secy_create_rx_sa(int macsec_id, u32 rx_sc_id, u8 an, u32 lpn, u8 *sak, u32 sak_len)
```

Description:

- creates RxSa (secure association for RX)

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number
- lpn - lowest packet number, value used in anti-replay algorithm
- sak - secure association key (the key used for decryption)
- sak_len - length of the key (must be at most 32)

Return:

- returns 0 on success or a negative value in case something went wrong

7. dpa_macsec_secy_activate_rx_sa

```
int dpa_macsec_secy_activate_rx_sa(int macsec_id, u32 rx_sc_id, u8 an)
```

Description:

- used for activating the secure association for RX channel

Parameters:

- macsec_id - MACsec interface identifier

- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number

Return:

- returns 0 on success or a negative value in case something went wrong

8. dpa_macsec_secy_disable_rx_sa

```
int dpa_macsec_secy_disable_rx_sa(int macsec_id, u32 rx_sc_id, u8 an)
```

Description:

- used for disabling the secure association for RX channel

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number

Return:

- returns 0 on success or a negative value in case something went wrong

9. dpa_macsec_secy_delete_rx_sa

```
int dpa_macsec_secy_delete_rx_sa(int macsec_id, u32 rx_sc_id, u8 an)
```

Description:

- used for deleting the secure association for RX channel

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number

Return:

- returns 0 on success or a negative value in case something went wrong

10. dpa_macsec_secy_delete_rx_sc

```
int dpa_macsec_secy_delete_rx_sc(int macsec_id, u32 rx_sc_id)
```

Description:

- used for deleting the secure channel for RX

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call

Return:

- returns 0 on success or a negative value in case something went wrong

11. dpa_macsec_secy_delete_tx_sa

```
int dpa_macsec_secy_delete_tx_sa(int macsec_id, u8 an)
```

Description:

- used for deleting the secure association for TX channel

Parameters:

- macsec_id - MACsec interface identifier
- an - association number

Return:

- returns 0 on success or a negative value in case something went wrong

12. dpa_macsec_secy_disable

```
int dpa_macsec_secy_disable(int macsec_id)
```

Description:

- used for deleting the SecY and disabling the associated TxSc

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns 0 on success or a negative value in case something went wrong

13. dpa_macsec_disable

```
int dpa_macsec_disable(int macsec_id)
```

Description:

- used for disabling MACsec

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns 0 on success or a negative value in case something went wrong

14. dpa_macsec_disable_all

```
int dpa_macsec_disable_all(int macsec_id)
```

Description:

- used as a single call for disabling all that was enabled for MACsec to work
- is the equivalent for calling dpa_macsec_secy_disable_rx_sa, dpa_macsec_secy_delete_rx_sa, dpa_macsec_secy_delete_rx_sc, dpa_macsec_secy_delete_tx_sa, dpa_macsec_secy_disable and dpa_macsec_disable one by one

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns 0 on success or a negative value in case something went wrong

15. dpa_macsec_secy_get_txsc_phys_id

```
int dpa_macsec_secy_get_txsc_phys_id(int macsec_id)
```

Description:

- obtain the ID associated with the TX SC by the Fman

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns the tx_sc_id corresponding to the SecY from Fman
- returns a negative value in case something went wrong

16. dpa_macsec_secy_modify_txsa_key

```
int dpa_macsec_secy_modify_txsa_key(int macsec_id, u8 an, u8 *sak, u32 sak_len)
```

Description:

- used to change the encryption key

Parameters:

- macsec_id - MACsec interface identifier
- an - association number
- sak - new secure association key (the key used for encryption)
- sak_len - length of the new key (must be at most 32)

Return:

- returns 0 on success or a negative value in case something went wrong

17. dpa_macsec_secy_modify_rxsa_key

```
int dpa_macsec_secy_modify_rxsa_key(int macsec_id, u32 rx_sc_id, u8 an, u8 *sak, u32 sak_len)
```

Description:

- used to change the decryption key

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number
- sak - new secure association key (the key used for decryption)
- sak_len - length of the new key (must be at most 32)

Return:

- returns 0 on success or a negative value in case something went wrong

18. dpa_macsec_secy_get_tx_sa_active_an

```
int dpa_macsec_secy_get_tx_sa_active_an(int macsec_id)
```

Description:

- used to get the active association number for this TxSc

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns the association number on success or a negative value in case something went wrong

19. `dpa_macsec_secy_get_rxsc_phys_id`

```
int dpa_macsec_secy_get_rxsc_phys_id(int macsec_id, u32 rx_sc_id)
```

Description:

- obtain the ID associated with the RX SC by the Fman

Parameters:

- `macsec_id` - MACsec interface identifier
- `rx_sc_id` - the one received from MACsec after `dpa_macsec_secy_create_rx_sc` call

Return:

- returns the `rx_sc_id` corresponding to the peer's SecY from Fman
- returns a negative value in case something went wrong

20. `dpa_macsec_secy_rx_sa_update_npn`

```
int dpa_macsec_secy_rx_sa_update_npn(int macsec_id, u32 rx_sc_id, u8 an, u32 pn)
```

Description:

- used to set the next packet number that MACsec should handle on RX

Parameters:

- `macsec_id` - MACsec interface identifier
- `rx_sc_id` - the one received from MACsec after `dpa_macsec_secy_create_rx_sc` call
- `an` - association number
- `pn` - packet number

Return:

- returns 0 on success or a negative value in case something went wrong

21. `dpa_macsec_secy_rx_sa_update_lpn`

```
int dpa_macsec_secy_rx_sa_update_lpn(int macsec_id, u32 rx_sc_id, u8 an, u32 pn)
```

Description:

- used to set the lowest packet number that MACsec should handle on RX

Parameters:

- `macsec_id` - MACsec interface identifier
- `rx_sc_id` - the one received from MACsec after `dpa_macsec_secy_create_rx_sc` call
- `an` - association number
- `pn` - packet number

Return:

- returns 0 on success or a negative value in case something went wrong

22. dpa_macsec_get_revision

```
int dpa_macsec_get_revision(int macsec_id)
```

Description:

- used to find the current revision of MACsec

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns the revision corresponding to the macsec_id
- returns a negative value in case something went wrong

23. dpa_macsec_set_exception

```
int dpa_macsec_set_exception(int macsec_id, bool enable_ex, macsec_exception ex)
```

Description:

- used to enable/disable a trigger for a certain exception

Parameters:

- macsec_id - MACsec interface identifier
- enable_ex - bool that will select if the exception given by the third argument will be enabled or not
- ex - the exception for which a trigger will be activated/deactivated

Return:

- returns 0 on success or a negative value in case something went wrong

MACsec API usage examples

1. Basic usage

```
enable_macsec_t en_macsec;
enable_secy_t enable_secy;
int i, rx_sc_id;
u8 an = 0;
u32 lpn = 0;
u32 sa_key_len = 32;
u16 port_src = 1;
u16 port_dst = 1;
u64 sci_src; //MAC + port - 64b
u64 sci_dest; //MAC + port - 64b

u8 *sa_key = malloc(sa_key_len * sizeof(u8));
for (i = 0; i < sa_key_len; i++)
    sa_key[i] = 0x12;

sci_src = (mac_src << 16) | port_src;
sci_dest = (mac_dst << 16) | port_dst;

en_macsec.if_name = ifname;
en_macsec.if_name_length = strlen(ifname) + 1;

en_macsec.config_unknown_sci_treatment = FALSE;
en_macsec.config_invalid_tag_treatment = FALSE;
```

```
en_macsec.config_key_frame_treatment = FALSE;
en_macsec.config_untag_treatment = FALSE;
en_macsec.config_pn_exhaustion_threshold = FALSE;
en_macsec.config_keys_unreadable = FALSE;
en_macsec.config_sectag_without_sci = FALSE;
en_macsec.config_exception = FALSE;
en_macsec.enable_exception = FALSE;

macsec_id = dpa_macsec_enable(en_macsec);

enable_secy.macsec_id = macsec_id;
enable_secy.sci = sci_src;

enable_secy.config_insertion_mode = FALSE;
enable_secy.config_protect_frames = FALSE;
enable_secy.config_replay_window = FALSE;
enable_secy.config_validation_mode = FALSE;
enable_secy.config_confidentiality = FALSE;
enable_secy.config_point_to_point = FALSE;
enable_secy.config_exception = FALSE;
enable_secy.config_event = FALSE;

dpa_macsec_secy_en(enable_secy);
dpa_macsec_secy_create_tx_sa(macsec_id, an, sa_key, sa_key_len);
dpa_macsec_secy_activate_tx_sa(macsec_id, an);

rx_sc_id = dpa_macsec_secy_create_rx_sc(macsec_id, sci_dest);
dpa_macsec_secy_create_rx_sa(macsec_id, rx_sc_id, an, lpn, sa_key, sa_key_len);
dpa_macsec_secy_activate_rx_sa(macsec_id, rx_sc_id, an);

/* ... */

free(sa_key);

dpa_macsec_disable_all(macsec_id);

/* Or:
 * dpa_macsec_secy_disable_rx_sa(macsec_id, rx_sc_id, an);
 * dpa_macsec_secy_delete_rx_sa(macsec_id, rx_sc_id, an);
 * dpa_macsec_secy_delete_rx_sc(macsec_id, rx_sc_id);
 * dpa_macsec_secy_delete_tx_sa(macsec_id, an);
 * dpa_macsec_secy_disable(macsec_id);
 * dpa_macsec_disable(macsec_id);
 */
```

2. Enable confidentiality

Enable the confidentiality flag and set the confidentiality offset in the `enable_secy_t` structure.

```
enable_secy.macsec_id = macsec_id;
enable_secy.sci = sci_src;

enable_secy.config_insertion_mode = FALSE;
enable_secy.config_protect_frames = FALSE;
enable_secy.config_replay_window = FALSE;
enable_secy.config_validation_mode = FALSE;

enable_secy.config_confidentiality = TRUE;
```

```

enable_secy.confidentiality_enable = TRUE;
enable_secy.confidentiality_offset = 0;

enable_secy.config_point_to_point = FALSE;
enable_secy.config_exception = FALSE;
enable_secy.config_event = FALSE;

```

3. Modify the TxSa and the RxSa keys

Generate a new set of keys and call the appropriate API functions.

```

for(i = 0; i < sa_key_len; ++i) {
    txsa_key[i] = 0x23;
}

dpa_macsec_secy_modify_txsa_key(macsec_id, an, txsa_key, sa_key_len);

for(i = 0; i < sa_key_len; ++i) {
    rxsa_key[i] = 0x57;
}

dpa_macsec_secy_modify_rxsa_key(macsec_id, rx_sc_id, an, rxsa_key, sa_key_len);

```

5.4.1.2.8.2 MACsec Quick Start Guide

After integrating the MACsec API with hostapd and wpa_supplicant, as exemplified in the [MACsec User Space API Reference](#) on page 531 chapter, follow these steps to run the applications.

Software Installation and Build

- Compile MACsec as a module:

```

Device Drivers
+--> Network device support (NETDEVICES [=y])
+--> Ethernet driver support (ETHERNET [=y])
+--> Freescale devices (NET_VENDOR_FREESCALE [=y])
+--> DPAA Ethernet (FSL_SDK_DPAA_ETH [=y])
+--> DPAA Macsec [=m]

```

- Increase the FMan maximum frame size to 1554 bytes. This is needed in order to accommodate the MACsec header without decreasing the MTU:

```

Device Drivers
+--> Network device support (NETDEVICES [=y])
+--> Ethernet driver support (ETHERNET [=y])
+--> Freescale devices (NET_VENDOR_FREESCALE [=y])
+--> Frame Manager Support
+--> Freescale Frame Manager (datapath) support - SDK driver
(FSL_SDK_FMAN [=y])
+--> Maximum L2 frame size [=1554]

```

- Set the appropriate FMan version (needed by B-series and T-series platforms):

```
Device Drivers
+--> Network device support (NETDEVICES [=y])
    +--> Ethernet driver support (ETHERNET [=y])
        +--> Freescale devices (NET_VENDOR_FREESCALE [=y])
            +--> Frame Manager Support
                +--> Freescale Frame Manager (datapath) support - SDK driver
(FSL_SDK_FMAN [=y])
                    +--> FMAN_V3L like T1040, T1042, T1020, T1022 [=y]
```

- Build the hostapd and wpa_supplicant

Add the recipes to your Yocto build by adding the following lines to your `conf/local.conf` file:

```
IMAGE_INSTALL_append = " hostapd"
IMAGE_INSTALL_append = " wpa-supPLICANT"
```

In order to integrate the MACsec API with the hostapd application, apply the `0001-Add-DPAA-MACsec-API.patch` patch from the recipe directory (`meta-freescale/recipes-connectivity/hostapd/hostapd-2.4`) on top of the hostapd version 2.4 repository. Build independently after committing the necessary changes.

Execution setup

- Generate a set of OpenSSL keys and certificates for the 802.1X authentication.
- Create a configuration file for the HostAP.

A separate configuration file is needed for each interface on which the hostapd will run. `hostapd.conf` example:

```
ctrl_interface=/var/run/hostapd
ctrl_interface_group=0
use_pae_group_addr=1

# Replace the following with the appropriate interface name
interface=fm1-gb3

driver=wired
logger_syslog=-1
logger_syslog_level=2
logger_stdout=-1
logger_stdout_level=2
dump_file=/tmp/hostapd.dump

eapol_version=2
ieee8021x=1
eap_server=1

# Replace the following with the path to the eap_user_file
eap_user_file=/etc/hostapd.eap_user

# Replace the following with your CA certificate path
ca_cert=/etc/server.crt
server_cert=/etc/server.crt
private_key=/etc/server.key
```

- Create an `eap_user_file` for the HostAP.

`hostapd.eap_user` example:

```
# Phase 1 users
* PEAP
# Replace the following with your desired credentials
"test" MSCHAPV2,MD5,PEAP "password" [2]
```

- Create a configuration file for the `wpa_supplicant`.

`wpa_supplicant.conf` example:

```
ctrl_interface=/var/run/wpa_supplicant
ctrl_interface_group=0
ap_scan=0
fast_reauth=1
eapol_version=2

network={
    ssid=""
    key_mgmt=IEEE8021X
    scan_ssid=0
    eap=PEAP
    priority=100
    phase2="auth=MSCHAPV2"

    # Replace the following with the credentials from the HostAP's eap_user_file
    identity="test"
    password="password"

    # Replace the following with your CA certificate path
    ca_cert="/etc/server.crt"
}
```

- Boot the board according to the platform's documentation.
- Insert the `fsl-dpa-macsec` module.

Specify in a comma separated list the interfaces on which MACsec will be enabled:

```
modprobe fsl-dpa-macsec ifs="fm1-gb3,fm1-gb4"
```

- Start the HostAP.

```
# Replace with the path to the configuration file
/usr/sbin/hostapd /etc/hostapd.conf &
```

- Start the `wpa_supplicant`.

```
# Replace with the path to the configuration file and the appropriate interface name
```

```
wpa_supplicant -Dwired -ifm1-gb3 -c/etc/wpa_supplicant.conf &
```

- Inspect MACsec traffic statistics with `ethtool`.

```
ethtool -S fm1-gb3
```

Limitations and known issues

- Only one SecY with one corresponding TX SC can be enabled on each interface.
- Once MACsec is enabled, all traffic passes through the SecY.
- MACsec is not compatible with ASF.
- MACsec statistics for RX path are not supported in `ethtool`.
- MACsec is not functional on ports connected to the internal L2switch on T1040 platforms. Consult your platform's specifications to determine which ports have MACsec support.

5.4.1.2.9 Changes from previous versions

• SDK2.0

- *Debugging with ethtool*

Removed the DPAA Ethernet counters from debugfs and exported them through ethtool statistics.

- *CEETM*

Integrated the CEETM qdisc with the DPAA Ethernet driver.

• SDK1.8

- *MACsec*

Added MACsec kernel driver and a set of User-space Linux APIs to allow developers to configure the MACsec hardware block.

- *DPAA Ethernet as loadable modules*

Added loadable module support for the DPAA Ethernet driver.

- *Adding support for DPAA Ethernet in Topaz Hypervisor*

This is a documentation section enumerating the steps to follow in order to enable DPAA Ethernet in Topaz Hypervisor.

• SDK1.7

- *PCD configuration files*

Included in the SDK package PCD configuration and policy files (.xml files) for each platform supported by the SDK. The configuration files include all the ports enabled by the RCWs associated with each platform.

• SDK1.6

- *oNIC device driver - DPAA hardware offloading aware Ethernet net device*

Added oNIC netdevice, based on OH FMAN ports in order to offer advanced DPAA offloading capabilities like: IPsec offload, zero-copy frames between USDPAA and kernel stack, OH CSUM offload.

- *DPAA OH port driver update*

Previous versions of the offline port driver did not initialize the frame queues that entered and exited the OH port and relied on other software components (Ethernet driver, USDPAA, or other kernel modules) to initialize those queues. This update adds full capability for initializing both ingress and egress queues, eliminating the dependency on USDPAA and/

or other kernel modules. With it, complex offload architectures that have OH ports can be largely initialized by the offline port driver.

- *Priority Flow Control (PFC) - 802.1Qbb*

Experimental feature added on T4240/T2080 for traffic priority classes and MAC support for 802.1Qbb. PFC provides the ability to issue and respond to Pause Frames on a priority-flow basis and prevents a single flow from consuming the entire port's bandwidth. This is distinct from standard flow control which turns on or off the Ethernet port, stopping all flows.

- *Sleep/DeepSleep support and Wake on LAN (WoL) support*

Implement the suspend/resume features that allow stopping the Ethernet port before system enters the sleep state and resuming the port to normal state when the system is waken up. Updated the Ethernet driver to work together with the Wakeup-on-LAN options of ethtool utility. More information about this can be found at Documentation/power/devices.txt in the kernel source tree.

- **SDK1.5**

- *Single driver for termination and forwarding*

There is only one Ethernet driver codebase now, based on the "optimized for termination" version which implements the support for S/G frames. But the driver has undergone an optimization process such that its "IP forwarding" performance is similar now to that of the removed "optimized for forwarding" driver. In addition, the "termination" performance has been improved.

All source code related to the "forwarding" driver is removed, along with the following Kconfig options: FSL_DPAA_ETH_OPTIMIZE_FOR_IPFWD, FSL_DPAA_ETH_OPTIMIZE_FOR_TERM and CONFIG_FSL_DPAA_ETH_SG_SUPPORT. The "termination" driver support for S/G frames requires a different socket buffer layout. The **ASF** and **IEEE1588** modules has been adapted to the new buffer layout, so the existing feature set has been preserved.

- *Buffer recycling algorithm updates*

The most important part of the private driver performance improvement strategy consist in data buffer fragments recycling and skb structure recycling. These techniques, formerly deployed in "forwarding" driver, has been adapted for the S/G support and the new skb layout.

- *CPU hotplug support*

In order to support CPU Hotplug, the DPAA Ethernet driver has been adapted to a new QMAN API and implemented a new NAPI logic which maps the portals to each available CPU. When a CPU goes offline the portal interrupts are seamlessly migrated and processed by the boot-CPU (CPU #0)..

- *Control MAC multicast group using socket API on MAC-less netdevice.*

The MAC-less interface is now able to configure a MAC device by using the Proxy DPAA Ethernet Driver . The proxy interface offers a simple API to MAC-less interface for enablement and disablement of the MAC device and for adding and removing mac unicast and multicast addresses. This MAC-less feature is optional and can be activated by setting "proxy" attribute in the device tree node of the MAC-less interfaces.

- *ASF is the default SDK configuration*

For specific use cases, ASF can provide superior performance than the upstream IP stack, bypassing the normal Ethernet driver and stack processing. Normal stack features may become unavailable in ASF configuration. But, if one needs to compile out the ASF support, the variable KERNEL_DELTA_DEFCONFIG must be set to empty.

- **SDK1.4**

- Added pause frame control support through ethtool
- Added netpoll support
- Moved "QDisc bypass for performance reasons" option to ASF
- Added Linux standard API for hardware timestamping (IEEE1588)

- Moved kernel config options for DPAA Ethernet driver from "Device Drivers -> Network device support -> Ethernet (10000 Mbit) -> NXP Data Path Frame Manager Ethernet" to "Device Drivers->Network device support->Ethernet driver support->NXP devices -> DPAA Ethernet "
- **SDK1.3**
 - The bootarg which controls the maximum frame size (previously "fsl_fman_phy_max_frm") has been renamed as "fsl_fm_max_frm" and is now available in the menuconfig via: Device Drivers --> Frame Manager support --> NXP Frame Manager (datapath) support --> Maximum L2 frame size (CONFIG_FSL_FM_MAX_FRAME_SIZE)
 - The bootarg which controls the extra headroom ("fsl_fm_rx_extra_headroom") has been moved in the Kconfig and is now available via: Device Drivers --> Frame Manager support --> NXP Frame Manager (datapath) support --> Add extra headroom at beginning of data buffers (CONFIG_FSL_FM_RX_EXTRA_HEADROOM)
 - Scatter/Gather support is activated by the driver's "Optimize for termination" compile-time choice. It is no longer explicitly accessible via the Kconfig.

5.4.1.2.9.1 Known Issues

- The MTU currently defaults to a maximum of 1522. If you want a higher MTU, it is necessary to pass `fsl_fm_max_frm=N` on the kernel bootargs, where "N" is the desired maximum MTU + 22 .

5.4.1.2.9.2 Good Questions

1. What channel are the FQs assigned to?

A: Each interface uses by default one pool channel across all Software Portals and also the dedicated channels of each CPU. Note that any of these channels may be shared with other DPAA Eth net devices, and even with other DPAA drivers such as SEC. The *default* and *error* FQs are assigned to the pool channel. The TX queues are assigned to the (direct connect) channel linked to the TX port associated with the interface. Any other statically-defined queues will be assigned in a round-robin fashion to the core-affine portals.

2. What work queue are the FQs assigned to?

A:

- Tx Confirmation FQs go to WQ1
- Rx Error and Tx Error FQs go to WQ2
- Rx Default, Tx and PCD FQs go to WQ3

3. How do I use the core-affined queues?

The anticipated way of using the core-affined queues is to use one of the default FMC policy files:

```
/etc/fmc/config/private/common/policy_ipv4.xml
```

```
/etc/fmc/config/private/common/policy_ipv6.xml
```

Default FMC configuration files are provided for each reference board:

```
/etc/fmc/config/private/<name of reference board>/<RCW directory>/<name of configuration file>
```

Here are two examples showing FMC commands using the default configuration and policy files:

```
(1) fmc -c /etc/fmc/config/private/t2080rdb/RRFFXX_P_66_15/config.xml -p /etc/fmc/config/private/t2080rdb/RRFFXX_P_66_15/policy_ipv4.xml -a
```

Note that `/etc/fmc/config/private/t2080rdb/RRFFXX_P_66_15/policy_ipv4.xml` is a soft link to `/etc/fmc/config/private/common/policy_ipv4.xml`.

```
(2) fmc -c /etc/fmc/config/private/t4240rdb/SSFPPH_27_55_1_9/config_20g.xml -p /etc/fmc/config/private/common/policy_ipv4.xml -a
```

Note that `/etc/fmc/config/private/t4240rdb/RRFFXX_P_66_15/policy_ipv4.xml` is a soft link to `/etc/fmc/config/private/common/policy_ipv4.xml`.

If you create a configuration file instead of using one of the default configuration files, be sure to use the appropriate policies found in the default policy files:

```
/etc/fmc/config/private/common/policy_ipv4.xml
```

```
/etc/fmc/config/private/common/policy_ipv6.xml.
```

5.4.1.3 Quality of Service

5.4.1.3.1 Features

DPAA platforms can offload QoS functions such as policing, shaping, scheduling and prioritization to dedicated hardware blocks.

Traffic policing is achieved on ingress through the FMan. A two rate three color marker algorithm can be configured through the `fmc` tool.

Traffic scheduling, shaping, and prioritization is executed on the egress path in the QMan. Multiple algorithms, such as dual rate shaping and strict prioritization, are implemented and can be configured through queuing disciplines.

5.4.1.3.2 Policing

The FMan's Policer sub block implements a two rate, three color marker (trTCM) traffic policing algorithm. The algorithm has two configurable flavors: RFC2698 and RFC4115.

The `fmc` tool, described in detail in [Frame Manager Configuration Tool User's Guide](#), is used to enable the Policer and set up its parameters.

For more information regarding the FMan Policer and how it can be configured, see the [FMan Policer](#) and the [Policer Section](#) chapters.

5.4.1.3.3 Scheduling and Shaping

5.4.1.3.3.1 Description

Specific DPAA 1.x platforms offer scheduling, shaping and prioritization capabilities through CEETM (Customer Edge Egress Traffic Management). The CEETM hardware block is a member of the QMan on the NXP QorIQ T-series and B-series platforms. Its purpose is to enhance the performances of DPAA platforms by moving the egress QoS logic from software to hardware.

This chapter briefly describes the CEETM block and its capabilities. Furthermore, it presents how it can be configured through the Linux traffic control tool (`tc`) by using a custom queuing discipline.

5.4.1.3.3.1.1 The CEETM architecture

CEETM is a sub block of the QMan and is an alternative to the regular *frame queue - work queue - channel* scheduling mode. For more information regarding this workflow, or on DCPs and sub-portals, please refer to the [QMan Overview](#) chapter.

A CEETM block, pictured in [Figure 101. CEETM block](#) on page 548, is available for each FMan and it is intended to be used by FMan sub-portals linked to Ethernet interfaces.

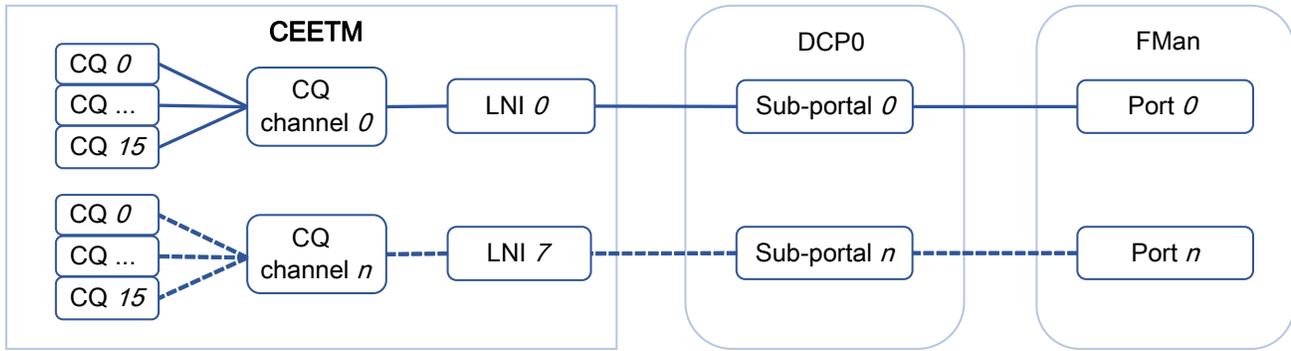


Figure 101. CEETM block

CEETM uses 8 Logical Network Interfaces (LNIs) that can be mapped to the FMan's DCP sub-portals. Depending on the platform used, there are 8 or 32 class queue channels (or CQ channels) that can be mapped to the LNIs. Multiple CQ channels can be mapped to the same LNI.

Each CQ channel contains 16 class queues. 8 CQs are independent while the other 8 can be grouped into 1 class group or 2 class groups of 4 queues each. The first group is called *group A* and the second is called *group B*.

5.4.1.3.1.2 Features

CEETM implements the following algorithms:

- Strict Priority scheduling
- Weighted Bandwidth Fair Scheduling (WBFS)
- dual-rate shaping with committed and excess rates (CR/ER)
- shaped and unshaped Fair Queueing scheduling (shFQ, uFQ)

These algorithms are used together in specific combinations based on the CEETM's architecture described previously and pictured in [Figure 102. CEETM architecture](#) on page 548 .

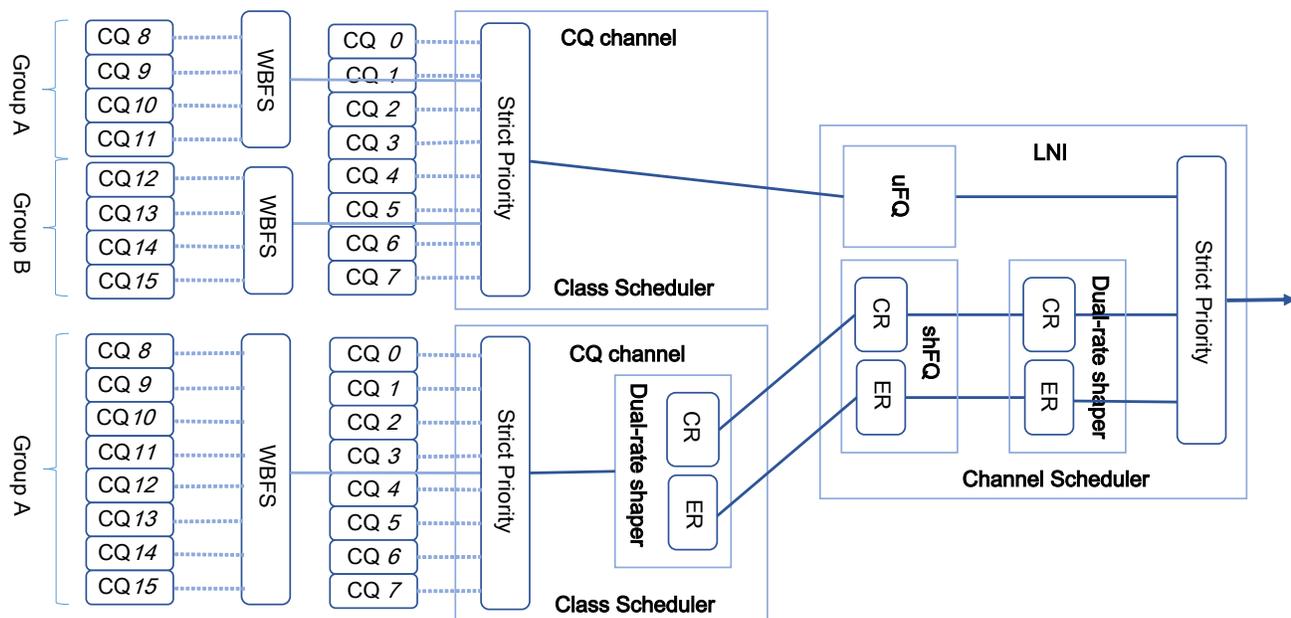


Figure 102. CEETM architecture

All the CQs connected to a CQ channel pass through a Strict Priority scheduler. The lower the CQ's ID, the higher the CQ's priority (e.g. CQ#3 has a higher priority than CQ#4, thus, as long as there are frames queued to CQ#3, CQ#4 will not be dequeued).

The priority of the CQ groups is configurable. All frames coming from the grouped CQs pass through the WBFS algorithm. Each CQ belonging to a group is assigned a weight portion of the bandwidth available to the group. The weight is a value from 1 to 248 in pseudo logarithmic steps of 1.5%. A list of available weights can be found in the platform's QorIQ DPAA Reference Manual.

The CQ channels can be shaped or unshaped. For CQs leading to a shaped channel, all frames will pass through a dual-rate shaper before entering the LNI. The independent CQs, as well as the class groups, can be configured to lead their frames through the CR shaper, the ER shaper, or both.

Each LNI aggregates frames from the CQ channels linked to it. All the unshaped frames from the unshaped CQ channels mapped to the LNI pass through the uFQ algorithm. The CR/ER frames from the shaped CQ channels pass through the shFQ algorithm and through another dual-rate shaper. Lastly, all frames pass through the LNI's Strict Priority module that schedules the unshaped frame (with high priority), the CR frames (with medium priority) and the ER frames (with low priority).

The shFQ algorithm schedules a channel for transmitting if the channel's shaper is time eligible (the shaper has a positive number of tokens in its bucket). When a channel finished its tokens, it is added to a waiting queue where it must wait for any other time eligible channels ahead of it finish transmitting.

The uFQ algorithm is similar to the shFQ. In the uFQ algorithm, all channels are time eligible. After finishing to transmit all their available data, they are added to the back of the time eligible waiting queue where their bucket is instantly refilled. The token bucket limit of the unshaped channels is configurable.

For more information regarding the CEETM's capabilities and detailed descriptions of the mentioned algorithms, take a look at your platform's QorIQ DPAA Reference Manual.

5.4.13.3.13 Integration with queuing disciplines

The CEETM block can be configured through the *ceetm* queuing discipline. A comparison between the hardware block and the traffic control's terminology is drawn in [Figure 103. Comparison between CEETM and tc terminology](#) on page 549.

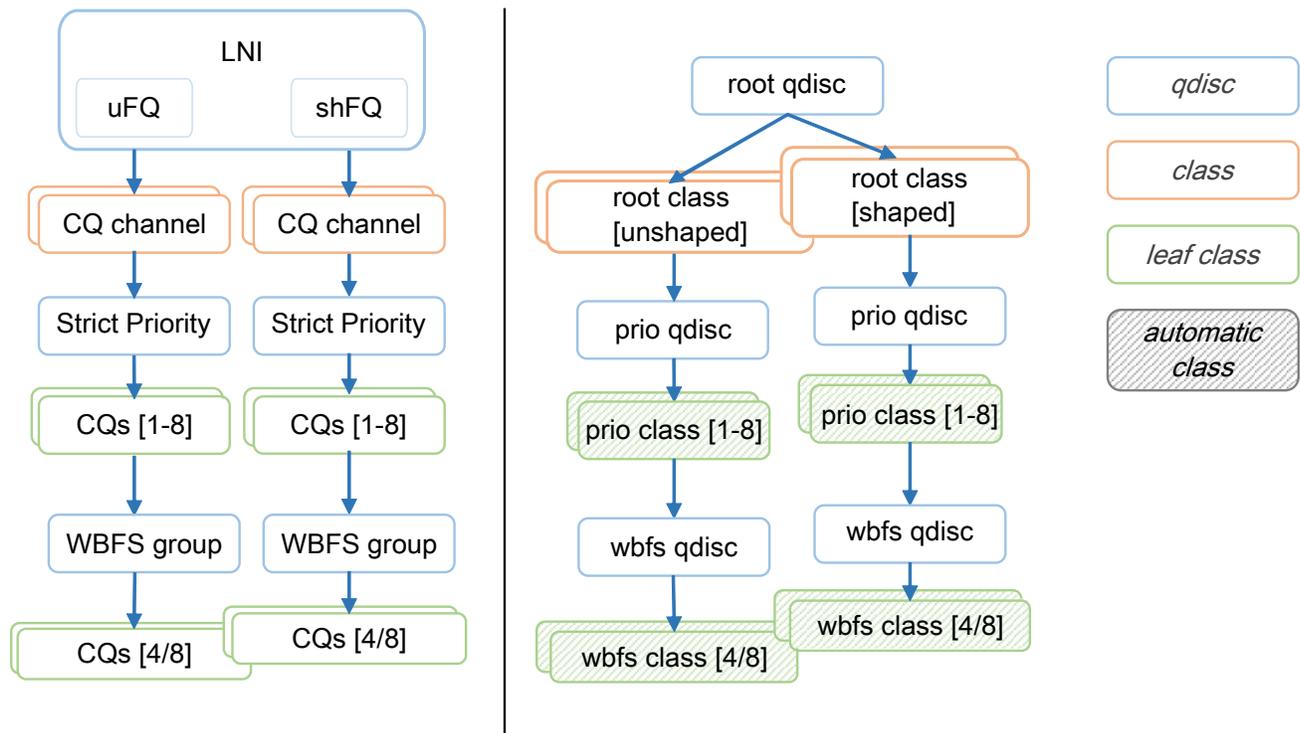


Figure 103. Comparison between CEETM and tc terminology

A LNI can be mapped to a FMan port by adding a `root ceetm` qdisc to a network interface. The LNI shaper's CR and ER are configured by setting a `rate`, and optional `ceil` and `overhead`, on the qdisc.

A CQ channel can be linked to a LNI by creating a `ceetm root` class mapped to the `root` qdisc. For an unshaped channel, the uFQ's token bucket limit (`tbl`) needs to be configured. For a shaped channel, the `rate`, and optional `ceil`, set the CR and ER.

Note: Shaped CQ channels can be linked to the LNI only if the LNI's shaper is enabled.

A channel's independent CQs are configured when a `prio` qdisc is linked to a `root` class. Between 1 and 8 `prio` classes are generated, each class corresponding to a CQ linked to the channel's Strict Priority scheduler. The `qcount` parameter indicates the number of child classes. If the channel is shaped, all generated classes participate by default in both CR and ER shaping. In order to disable one or the other, the CQ's corresponding `prio` class's `cr` and `er` parameters can be changed.

Note: CQs linked to a shaped CQ channel can not have both CR and ER shaping disabled.

In order to configure the CQ groups, a `wbfs` qdisc is linked to one of the `prio` classes. Either 4 or 8 `wbfs` classes are generated, depending on the number of CQs in the group indicated by the `qcount` parameter. The group is placed right after its parent in the channel's Strict Priority list (e.g. if the `wbfs` qdisc is linked to the `prio` class #2, the priority list becomes: class #1, class #2, group, class #3, class #4, etc). The CQ weights are configured through the `qweight` parameter and can be changed for each CQ individually. For groups linked to shaped CQ channels, the CR and ER shaping are enabled by the `cr` and `er` parameters.

Note: Groups linked to a shaped CQ channel can not have both CR and ER shaping disabled.

For more details on the `ceetm` qdisc's parameters and configuration, see the [Usage](#) on page 551 chapter.

5.4.1.3.3.2 User guide

5.4.1.3.3.2.1 Supported platforms

The CEETM block is present and configurable through the `ceetm` qdisc on all T-series, B-series and LS1043A platforms.

5.4.1.3.3.2.2 Getting started

1. Enable the `ceetm` qdisc support in the kernel, along with any classifiers or other features that might be needed.

```
-> Device Drivers
  -> Network device support (NETDEVICES [=y])
    -> Ethernet driver support (ETHERNET [=y])
      -> Freescale devices (NET_VENDOR_FREESCALE [=y])
        -> DPAA Ethernet (FSL_SDK_DPAA_ETH [=y])
          -> DPAA CEETM QoS (FSL_DPAA_CEETM [=y])

-> Networking support (NET [=y])
  -> Networking options
    -> QoS and/or fair queueing (NET_SCHED [=y])
      -> Universal 32bit comparisons w/ hashing (u32) (NET_CLS_U32 [=y])
```

2. Add the `ceetm` recipe to your Yocto build by adding the following line to your `conf/local.conf` file.

```
IMAGE_INSTALL_append = " ceetm"
```

5.4.1.3.3.2.3 Limitations

- CEETM is supported on DPAA Private Ethernet interfaces only.
- CEETM isn't supported on top of Linux bonding interfaces.

5.4.1.3.3.2.4 Usage

You can see the `ceetm qdisc`'s help message by running the following command:

```
~# tc qdisc add ceetm help
Usage:
... qdisc add ... ceetm type root [rate R [ceil C] [overhead O]]
... class add ... ceetm type root (tbl T | rate R [ceil C])
... qdisc add ... ceetm type prio qcount Q
... qdisc add ... ceetm type wbfs qcount Q qweight W1 ... Wn [cr CR] [er ER]

Update configurations:
... qdisc change ... ceetm type root [rate R [ceil C] [overhead O]]
... class change ... ceetm type root (tbl T | rate R [ceil C])
... class change ... ceetm type prio [cr CR] [er ER]
... qdisc change ... ceetm type wbfs [cr CR] [er ER]
... class change ... ceetm type wbfs qweight W

Qdisc types:
root - configure a LNI linked to a FMan port
prio - configure a channel's Priority Scheduler with up to eight classes
wbfs - configure a Weighted Bandwidth Fair Scheduler with four or eight classes

Class types:
root - configure a shaped or unshaped channel
prio - configure an independent class queue

Options:
R - the CR of the LNI's or channel's dual-rate shaper (required for shaping scenarios)
C - the ER of the LNI's or channel's dual-rate shaper (optional for shaping scenarios, defaults to 0)
O - per-packet size overhead used in rate computations (required for shaping scenarios, recommended value is 24 i.e. 12 bytes IFG + 8 bytes Preamble + 4 bytes FCS)
T - the token bucket limit of an unshaped channel used as fair queuing weight (required for unshaped channels)
CR/ER - boolean marking if the class group or prio class queue contributes to CR/ER shaping (1) or not (0) (optional, at least one needs to be enabled for shaping scenarios, both default to 1 for prio class queues)
Q - the number of class queues connected to the channel (from 1 to 8) or in a class group (either 4 or 8)
W - the weights of each class in the class group measured in a log scale with values from 1 to 248 (when adding a wbfs qdisc, either four or eight, depending on the size of the class group; when updating a wbfs class, only one)
```

Filters need to be added on each `qdisc` layer in order to allow packets to reach the leaf classes. Likewise, all filters need to be removed from each `qdisc` layer when no longer used.

5.4.1.3.3.3 Examples

5.4.1.3.3.3.1 Rate limit two streams

5.4.1.3.3.3.1.1 Setup

In the following example a platform with CEETM support (T1040RDB - Client) is connected to another board (T1024RDB - Server) through a 1G link. The described setup is pictured in [Figure 104. Rate example setup](#) on page 552.

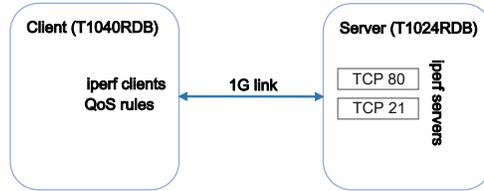


Figure 104. Rate example setup

The `iperf` clients run on the Client while the `iperf` servers run on the Server. The Server listens on 2 TCP ports (21 and 80).

```
root@t1024rdb:~# iperf -s -p 21 &
root@t1024rdb:~# iperf -s -p 80 &
```

PCDs are applied on both platforms in advance.

```
root@t1024rdb:~# fmc -c /etc/fmc/config/private/t1024rdb/RRX_PPP_95/config.xml -p /etc/fmc/config/private/t1024rdb/RRX_PPP_95/policy_ipv4.xml -a
root@t1040rdb:~# fmc -c /etc/fmc/config/private/t1040rdb/RR_P_66/config.xml -p /etc/fmc/config/private/t1040rdb/RR_P_66/policy_ipv4.xml -a
```

In order to keep this example minimal, ARP frames aren't filtered and classified. Thus, MAC addresses need to be exchanged and saved in advance as well.

```
root@t1040rdb:~# arp -s <server IP address> <server HW address>
root@t1024rdb:~# arp -s <client IP address> <client HW address>
```

After adding the qdiscs, the Client runs the `iperf` clients.

```
root@t1040rdb:~# iperf -c <server IP address> -p 21 &
root@t1040rdb:~# iperf -c <server IP address> -p 80 &
```

5.4.1.3.3.1.2 Execution

This example's corresponding qdisc and class hierarchy is pictured in [Figure 105. Rate example class hierarchy](#) on page 552.

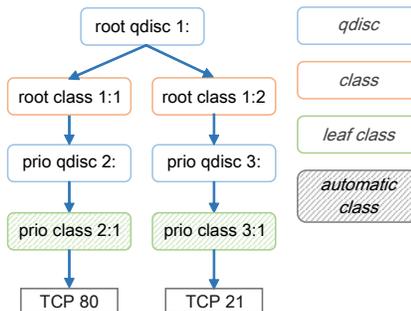


Figure 105. Rate example class hierarchy

Add a `ceetm` qdisc to the interface and configure the LNI's dual-rate shaper with a CR of 1Gbps.

```
root@t1040rdb:~# tc qdisc add dev fm1-gb0 root handle 1: ceetm type root rate 1000mbit overhead 24
```

Add a shaped channel to the LNI and configure its dual-rate shaper with a CR of 150mbps.

```
root@t1040rdb:~# tc class add dev fm1-gb0 parent 1: classid 1:1 ceetm type root rate 150mbit
```

Add another shaped channel to the LNI and configure its dual-rate shaper with a CR of 850mbps.

```
root@t1040rdb:~# tc class add dev fm1-gb0 parent 1: classid 1:2 ceetm type root rate 850mbit
```

Configure one of the first channel's priority classes (marked by default as both CR and ER eligible).

```
root@t1040rdb:~# tc qdisc add dev fm1-gb0 parent 1:1 handle 2: ceetm type prio qcount 1
```

Configure one of the second channel's priority classes (marked by default as both CR and ER eligible).

```
root@t1040rdb:~# tc qdisc add dev fm1-gb0 parent 1:2 handle 3: ceetm type prio qcount 1
```

Add filters that will classify all packets with the destination port equal to 80 and lead them through the priority class of the first channel.

```
root@t1040rdb:~# tc filter add dev fm1-gb0 parent 1: prio 1 protocol ip u32 match ip dport 80 0xffff flowid 1:1
root@t1040rdb:~# tc filter add dev fm1-gb0 parent 2: prio 1 protocol ip u32 match ip dport 80 0xffff flowid 2:1
```

Add filters that will classify all packets with the destination port equal to 21 and lead them through the priority class of the second channel.

```
root@t1040rdb:~# tc filter add dev fm1-gb0 parent 1: prio 1 protocol ip u32 match ip dport 21 0xffff flowid 1:2
root@t1040rdb:~# tc filter add dev fm1-gb0 parent 3: prio 1 protocol ip u32 match ip dport 21 0xffff flowid 3:1
```

5.4.1.3.3.3.2 Prioritization of two streams

5.4.1.3.3.3.2.1 Setup

The same setup is used as for the [rate limit](#) example.

5.4.1.3.3.3.2.2 Execution

This example's corresponding qdisc and class hierarchy is pictured in [Figure 106. Prioritization example class hierarchy](#) on page 553.

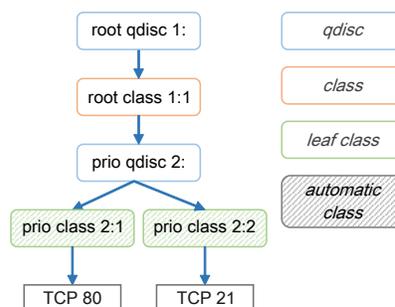


Figure 106. Prioritization example class hierarchy

Add a ceetm qdisc to the interface and configure the LNI's dual-rate shaper with a CR of 1Gbps.

```
root@t1040rdb:~# tc qdisc add dev fm1-gb0 root handle 1: ceetm type root rate 1000mbit overhead 24
```

Add a shaped channel to the LNI and configure its dual-rate shaper with a CR of 1Gbps.

```
root@t1040rdb:~# tc class add dev fm1-gb0 parent 1: classid 1:1 ceetm type root rate 1000mbit
```

Configure two of the channel's priority classes (marked by default as both CR and ER eligible).

```
root@t1040rdb:~# tc qdisc add dev fm1-gb0 parent 1:1 handle 2: ceetm type prio qcount 2
```

Add filters that will classify all packets with the destination port equal to 80 and lead them through the highest priority class of the channel.

```
root@t1040rdb:~# tc filter add dev fm1-gb0 parent 1: prio 1 protocol ip u32 match ip dport 80 0xffff flowid 1:1
root@t1040rdb:~# tc filter add dev fm1-gb0 parent 2: prio 1 protocol ip u32 match ip dport 80 0xffff flowid 2:1
```

Add filters that will classify all packets with the destination port equal to 21 and lead them through the second (lowest) priority class of the channel.

```
root@t1040rdb:~# tc filter add dev fm1-gb0 parent 1: prio 1 protocol ip u32 match ip dport 8000 0xffff flowid 1:1
root@t1040rdb:~# tc filter add dev fm1-gb0 parent 2: prio 1 protocol ip u32 match ip dport 8000 0xffff flowid 2:2
```

5.4.1.3.3.3.3 Assigning weights to two streams

5.4.1.3.3.3.3.1 Setup

The same setup is used as for the [rate limit](#) example.

5.4.1.3.3.3.3.2 Execution

This example's corresponding qdisc and class hierarchy is pictured in [Figure 107. WBFS example class hierarchy](#) on page 554.

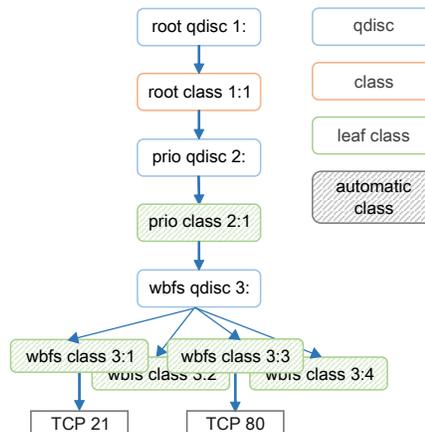


Figure 107. WBFS example class hierarchy

Add a ceetm qdisc to the interface and configure the LNI's dual-rate shaper with a CR of 1Gbps.

```
root@t1040rdb:~# tc qdisc add dev fm1-gb0 root handle 1: ceetm type root rate 1000mbit overhead 24
```

Add a shaped channel to the LNI and configure its dual-rate shaper with a CR of 1Gbps.

```
root@t1040rdb:~# tc class add dev fm1-gb0 parent 1: classid 1:1 ceetm type root rate 1000mbit
```

Configure one of the channel's priority classes (marked by default as both CR and ER eligible).

```
root@t1040rdb:~# tc qdisc add dev fm1-gb0 parent 1:1 handle 2: ceetm type prio qcount 1
```

Configure a class group of four classes, place it after the 2:1 class in the priority list, and assign different weights to each class (10, 50, 120 and 200).

```
root@t1040rdb:~# tc qdisc add dev fm1-gb0 parent 2:1 handle 3: ceetm type wbfq qcount 4 qweight 10 50 120 200 cr 1 er 1
```

Add filters that will classify all packets with the destination port equal to 21 and lead them through the class with the highest weight of the group.

```
root@t1040rdb:~# tc filter add dev fm1-gb0 parent 1: prio 1 protocol ip u32 match ip dport 21 0xffff flowid 1:1
root@t1040rdb:~# tc filter add dev fm1-gb0 parent 2: prio 1 protocol ip u32 match ip dport 21 0xffff flowid 2:1
root@t1040rdb:~# tc filter add dev fm1-gb0 parent 3: prio 1 protocol ip u32 match ip dport 21 0xffff flowid 3:1
```

Add filters that will classify all packets with the destination port equal to 80 and lead them through another classes of the group.

```
root@t1040rdb:~# tc filter add dev fm1-gb0 parent 1: prio 1 protocol ip u32 match ip dport 80 0xffff flowid 1:1
root@t1040rdb:~# tc filter add dev fm1-gb0 parent 2: prio 1 protocol ip u32 match ip dport 80 0xffff flowid 2:1
root@t1040rdb:~# tc filter add dev fm1-gb0 parent 3: prio 1 protocol ip u32 match ip dport 80 0xffff flowid 3:3
```

5.4.1.3.3.4 Unshaped Fair Queuing of two streams

5.4.1.3.3.4.1 Setup

In the following example a platform with CEETM support (T1024RDB - Main) is connected to two other boards: a T1024RDB (Client) through a 10G link and a T1040RDB (Server) through a 1G link. The described setup is pictured in [Figure 108. Unshaped Fair Queuing example setup](#) on page 555.

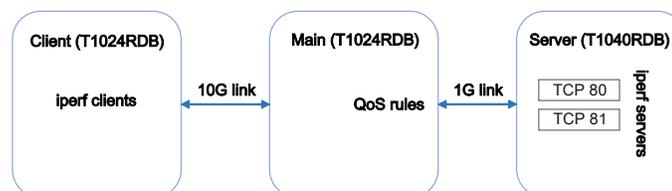


Figure 108. Unshaped Fair Queuing example setup

The `iperf` clients run on the Client while the `iperf` servers run on the Server. The Server listens on two TCP ports (80 and 81).

```
root@t1040rdb:~# iperf -s -p 80 &
root@t1040rdb:~# iperf -s -p 81 &
```

PCDs are applied on all platforms in advance.

```
root@t1024rdb:~# fmc -c /etc/fmc/config/private/t1024rdb/RRX_PPP_95/config.xml -p /etc/fmc/config/private/t1024rdb/RRX_PPP_95/policy_ipv4.xml -a
root@t1040rdb:~# fmc -c /etc/fmc/config/private/t1040rdb/RR_P_66/config.xml -p /etc/fmc/config/private/t1040rdb/RR_P_66/policy_ipv4.xml -a
```

In order to keep this example minimal, ARP frames aren't filtered and classified. Thus, MAC addresses need to be exchanged and saved in advance as well.

```
# Server:
root@t1040rdb:~# arp -s <main IP address> <main HW address>
# Main:
root@t1024rdb:~# arp -s <client IP address> <client HW address>
root@t1024rdb:~# arp -s <server IP address> <server HW address>
# Client:
root@t1024rdb:~# arp -s <main IP address> <main HW address>
```

IP forwarding is enabled on the Main board. Routes are added on the Server and Client boards as well.

```
# Main:
root@t1024rdb:~# echo 1 > /proc/sys/net/ipv4/ip_forward
# Client:
root@t1024rdb:~# route add -net <server network address> <server network mask> gw <main IP address>
# Server:
root@t1040rdb:~# route add -net <client network address> <client network mask> gw <main IP address>
```

After adding the qdiscs, the Client runs the `iperf` clients.

```
root@t1024rdb:~# iperf -c <server IP address> -p 80 &
root@t1024rdb:~# iperf -c <server IP address> -p 81 &
```

5.4.1.3.3.3.4.2 Execution

This example's corresponding qdisc and class hierarchy is pictured in [Figure 109. Unshaped Fair Queuing example class hierarchy](#) on page 556.

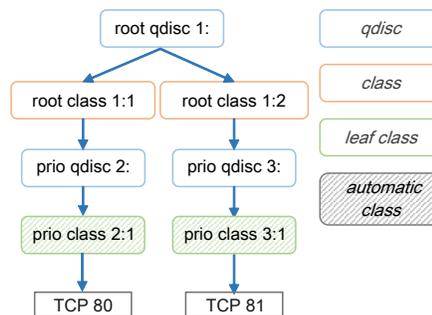


Figure 109. Unshaped Fair Queuing example class hierarchy

Add a ceetm qdisc to the interface and don't configure the LNI's dual-rate shaper.

```
root@t1024rdb:~# tc qdisc add dev fm1-mac3 root handle 1: ceetm type root
```

Add an unshaped channel to the LNI and configure its CR's token bucket limit to 1000 bytes.

```
root@t1024rdb:~# tc class add dev fm1-mac3 parent 1: classid 1:1 ceetm type root tbl 1000
```

Add another unshaped channel to the LNI and configure its CR's token bucket limit to 500 bytes.

```
root@t1024rdb:~# tc class add dev fm1-mac3 parent 1: classid 1:2 ceetm type root tbl 500
```

Configure one of the first channel's priority classes.

```
root@t1024rdb:~# tc qdisc add dev fm1-mac3 parent 1:1 handle 2: ceetm type prio qcount 1
```

Configure one of the second channel's priority classes.

```
root@t1024rdb:~# tc qdisc add dev fm1-mac3 parent 1:2 handle 3: ceetm type prio qcount 1
```

Add filters that will classify all packets with the destination port equal to 80 and lead them through the priority class of the first channel.

```
root@t1024rdb:~# tc filter add dev fm1-mac3 parent 1: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 1:1
root@t1024rdb:~# tc filter add dev fm1-mac3 parent 2: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 2:1
```

Add filters that will classify all packets with the destination port equal to 81 and lead them through the priority class of the second channel.

```
root@t1024rdb:~# tc filter add dev fm1-mac3 parent 1: prio 1 protocol ip u32 match ip dport 81
0xffff flowid 1:2
root@t1024rdb:~# tc filter add dev fm1-mac3 parent 3: prio 1 protocol ip u32 match ip dport 81
0xffff flowid 3:1
```

5.5 Flash Memory

5.5.1 JFFS2 on NOR Flash Device Driver User Manual

Linux SDK for QorIQ Processors

Description

Flash Controller in the SoC includes a NOR flash Machine. NOR flash machine support various type of NOR flash.

NOTE

This User Manual is also valid for Integrated Flash Controller (IFC). Please refer to the ["IFC NOR Flash User Manual"](#) to enable IFC NOR.

Module Loading

The Flash device driver supports kernel built-in and module.

U-Boot Configuration

Env Variable	Env Description	Sub Option	Option Description
bootargs	Kernel command line argument passed to kernel	setenv bootargs root=/dev/mtdblockx rootfstype=jffs2 rw console=ttyS0,115200	Uses NOR jffs2 filesystem as rootfs filesystem , and x should be the NOR jffs2 partition's mtdblock number.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
<pre>Device Drivers ---> <*> Memory Technology Device (MTD) support ---> --- Memory Technology Device (MTD) support [*] Command line partition table parsing <*> Flash partition map based on of description <*> Direct char device access to MTD devices <*> Caching block device access to MTD devices RAM/ROM/Flash chip drivers ---> <*> Detect flash chips by Common Flash Interface (CFI) probe <*> Support for AMD/Fujitsu flash chips Mapping Drivers for chip access ---> <*> Flash device in physical memory map based on OF description</pre>	MTD and AMD flash support
<pre>File systems ---> Miscellaneous filesystems ---> <*> Journaling Flash File System v2 (JFFS2) support (0) JFFS2 debugging verbosity [*] JFFS2 write-buffering support</pre>	JFFS2 FS support

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_MTD	y/m/n	y	Enable Memory Technology Devices support
CONFIG_MTD_CMDLINE_PARTS	y/n	y	Enable parsing partition via kernel command line
CONFIG_MTD_BLOCK	y/m/n	y	Enables block driver for MTD device
CONFIG_MTD_OF_PARTS	y/m/n	y	Enables partition parsing of partition map from the children of the flash node
ONFIG_MTD_CFI_AMDSTD	y/m/n	y	Enables support for AMD/Fujitsu flash chips
ONFIG_MTD_PHYSMAP_OF	y/m/n	y	Enables physical memory map based on OF description
CONFIG_JFFS2_FS	y/m/n	y	Enable Journal Flash FS v2 support
CONFIG_JFFS2_FS_WRITEBUFFER	y/n	y	Enable JFFS2 write-buffering support

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
/linux/drivers/mtd	Linux MTD subsystem
/linux/drivers/mtd/chips	Linux MTD NOR device support
/linux/fs/jffs2	Linux JFFS2 FS system

In U-Boot, flash the JFFS2 FS to the NOR flash

Note: the erase block size for the NOR flash device (as example) is 128KB. Generate the JFFS2 FS accordingly.

```
=> tftp 1000000 $nor_jffs2_fs
=> erase $nor_jffs_start_address $nor_jffs_end_address /* erase address and size is decided by
your partition setting */
=> cp.b 1000000 $nor_jffs_start_address $filesize /* flash JFFS2 FS to the jffs2 partition on NOR
device */
```

The `nor_jffs_start_address` and `nor_jffs_end_address` can get from `(board)_Quick_Strat_Guide` or `(board)_user_manual`.

Special MTD partition DTS

Note: The `mtdblock` number for JFFS2 partitions on NOR device is decided by your system configuration and your partition setting.

Use `mtdparts=fe800000.nor:1m(uboot),8m(kernel),512k(dtb),11m(jffs2),-(fs)` in the `bootargs` for NOR partitions on B4860QDS.

Boot up the board

Note: Different boards have different partition settings. The following example is B4860QDS board's partition setting.

The `mtdblock3` is NOR JFFS2 partition.

```
fe800000.nor: Found 1 x16 devices at 0x0 in 16-bit bank. Manufacturer ID 0x000001 Chip ID
0x002801
Amd/Fujitsu Extended Query Table at 0x0040
Amd/Fujitsu Extended Query version 1.5.
number of CFI chips: 1
5 cmdlinepart partitions found on MTD device fe800000.nor
Creating 5 MTD partitions on "fe800000.nor":
0x0000000000000-0x0000000100000 : "uboot"
0x0000000100000-0x0000000900000 : "kernel"
0x0000000900000-0x0000000980000 : "dtb"
0x0000000980000-0x0000001480000 : "jffs2"
0x0000001480000-0x0000008000000 : "fs"
....
....
/* log in kernel */
root@b4860qds:~# cat /proc/mtd
dev: size erasesize name
mtd0: 00100000 00020000 "uboot"
mtd1: 00800000 00020000 "kernel"
mtd2: 00080000 00020000 "dtb"
mtd3: 00b00000 00020000 "jffs2"
mtd4: 06b80000 00020000 "fs"
mtd5: 00100000 00020000 "NAND U-Boot Image"
```

Linux Kernel Drivers

Flash Memory

```
mtd6: 00100000 00020000 "NAND DTB Image"
mtd7: 00a00000 00020000 "NAND Linux Kernel Image"
mtd8: 1f400000 00020000 "NAND RFS Image"
mtd9: 00080000 00001000 "spife110000.0"

[root@b4860qds root]# ls /dev/mtdblock3 -l /* the block number should be the NOR jffs2
partition's mtdblock number*/
brw-r----- 1 root disk 31, 4 Mar  5 14:13 /dev/mtdblock3
[root@b4860ds root]# mount -t jffs2 /dev/mtdblock3 /mnt/
[root@b4860ds root]# ls /mnt/
bin  etc  lib      log  opt   root  sys  usr
dev  home  linuxrc  mnt  proc  sbin  tmp  var
[root@b4860qds root]# umount /mnt/
[root@b4860qds root]# ls /mnt/
cdrom  floppy  nfs  rwfs  src
[root@b4860qds root]#
```

Another way to test NOR JFFS2 support is to boot up kernel and mount JFFS2 filesystem

Note: the block number should be the NOR JFFS2 partition's mtdblock number

Under U-Boot prompt:

```
setenv bootargs root=/dev/mtdblock3 rootfstype=jffs2 rw console=ttyS0,115200
tftp 1000000 uImage
tftp c00000 dtb
bootm 1000000 - c00000
```

5.5.2 JFFS2 on NAND Flash Device Driver User Manual

Linux SDK for QorIQ Processors

Description

The Enhanced Local Bus Controller in the platform includes a special NAND flash Control Machine (FCM). FCM provides the hardware support for using small or large page NAND flash.

NOTE

This User Manual is written for Enhanced Local Bus Controller (eLBC). It is also valid for Integrated Flash Controller (IFC). Please refer to "[IFC NAND Flash User Manual](#)" for enabling IFC NAND.

Module Loading

The eLBC device driver supports kernel built-in and module.

U-Boot Configuration

Env Variable	Env Description	Sub Option	Option Description
bootargs	Kernel command line argument passed to kernel	setenv bootargs root=/dev/mtdblockx rootfstype=jffs2 rw console=ttyS0,115200	Uses NAND JFFS2 filesystem as rootfs filesystem, and x should be the NAND JFFS2 partition's mtdblock number.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
<pre> Device Drivers ---> <*> Memory Technology Device (MTD) support ---> --- Memory Technology Device (MTD) support [*] Command line partition table parsing <*> Direct char device access to MTD devices <*> Caching block device access to MTD devices Mapping Drivers for chip access ---> <*> Flash device in physical memory map based on OF description <*> NAND Device support ---> [*] NAND support for Freescale eLBC controllers </pre>	MTD and eLBC NAND support
<pre> File systems ---> Miscellaneous filesystems ---> <*> Journaling Flash File System v2 (JFFS2) support (0) JFFS2 debugging verbosity [*] JFFS2 write-buffering support </pre>	JFFS2 FS support

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_MTD	y/m/n	y	Enable Memory Technology Devices support
CONFIG_MTD_CMDLINE_PARTS	y/n	y	Enable parsing partition via kernel command line
CONFIG_MTD_BLOCK	y/m/n	y	Enables block driver for MTD device
CONFIG_MTD_OF_PARTS	y/m/n	y	Enables partition parsing of partition map from the children of the flash node
CONFIG_MTD_NAND	y/m/n	y	Enables MTD NAND subsystem
CONFIG_MTD_NAND_FSL_ELBC	y/n	y	Enables NXP ELBC NAND device driver
CONFIG_JFFS2_FS	y/m/n	y	Enable Journal Flash FS v2 support
CONFIG_JFFS2_FS_WRITEBUFFER	y/n	y	Enable JFFS2 write-buffering support

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
/linux/drivers/mtd	Linux MTD subsystem
/linux/drivers/mtd/nand	Linux MTD NAND device support
/linux/fs/jffs2	Linux JFFS2 FS system

In U-boot, flash the JFFS2 FS to the NAND flash

Note: the erase block size for the NAND flash device (as example) is 128KB. Generate the JFFS2 FS accordingly.

```
=> tftp 1000000 $nand_jffs2_fs
=> nand erase $nand_jffs2_start_address $nand_jffs2_size /*erase address and size is decided by
your partition setting */
=> nand write 1000000 $nand_jffs2_start_address $filesize /* flash JFFS2 FS to the jffs2-nand
partition of NAND device */
The nand_jffs2_start_address and nand_jffs2_size can get from (board)_Quick_Strat_Guide or
(board)_user_manual.
```

Special MTD partition DTS

Note: The mtdblock number for JFFS2 partitions on NAND device is decided by your system configuration and your partition setting.

Using 'mknod' command to create the corresponding mtdblock device node if it does not exist in the file system.

```
nand@2,0 {
    #address-cells = "<1>";
    #size-cells = "<1>";
    compatible = "fsl,mpc8572-fcm-nand", "fsl,elbc-fcm-nand";
    reg = <0x2 0x0 0x40000>;
    u-boot@0 {
        reg = <0x0 0x02000000>;
        label = "NAND - U-Boot Image";
        read-only;
    };
    jffs2@2000000 {
        reg = <0x02000000 0x10000000>;
        label = "NAND - JFFS2 Root File System";
    };
    ramdisk@12000000 {
        reg = <0x12000000 0x08000000>;
        label = "NAND - RAMDISK File System";
        read-only;
    };
    kernel@1a000000 {
        reg = <0x1a000000 0x04000000>;
        label = "NAND - Linux Kernel Image";
    };
    dtb@1e000000 {
        reg = <0x1e000000 0x01000000>;
        label = "NAND - DTB Image";
        read-only;
    };
    empty@1f000000 {
        reg = <0x1f000000 0x21000000>;
        label = "NAND - Empty";
    };
};
nand@4,0 {
    compatible = "fsl,mpc8572-fcm-nand", "fsl,elbc-fcm-nand";
    reg = <0x4 0x0 0x40000>;
};
nand@5,0 {
    compatible = "fsl,mpc8572-fcm-nand", "fsl,elbc-fcm-nand";
    reg = <0x5 0x0 0x40000>;
};
```

```
nand@6,0 {
    compatible = "fsl,mpc8572-fcm-nand","fsl,elbc-fcm-nand";
    reg = <0x6 0x0 0x40000>;
```

Boot up the board

Note: The following example assumes that only NAND is enabled, NOR is not enabled.

Otherwise, the mtdblock number for NAND will be different, e.g. mtdblock1 will be NAND JFFS2 partition

```
....
NAND device: Manufacturer ID: 0xec, Chip ID: 0xd3 (Samsung NAND 1GiB 3,3V 8-bit)

nand_bbt: ECC error while reading bad block table

RedBoot partition parsing not available

Creating 6 MTD partitions on "ffa00000.flash":

0x00000000-0x02000000 : "NAND - U-Boot Image"

0x02000000-0x12000000 : "NAND - JFFS2 Root File System"

0x12000000-0x1a000000 : "NAND - RAMDISK File System"

0x1a000000-0x1e000000 : "NAND - Linux Kernel Image"

0x1e000000-0x1f000000 : "NAND - DTB Image"

0x1f000000-0x40000000 : "NAND - Empty"

eLBC NAND device at 0xffa00000, bank 2

NAND device: Manufacturer ID: 0xec, Chip ID: 0xd3 (Samsung NAND 1GiB 3,3V 8-bit)

nand_bbt: ECC error while reading bad block table

RedBoot partition parsing not available

eLBC NAND device at 0xffa40000, bank 4

NAND device: Manufacturer ID: 0xec, Chip ID: 0xd3 (Samsung NAND 1GiB 3,3V 8-bit)

nand_bbt: ECC error while reading bad block table

RedBoot partition parsing not available

eLBC NAND device at 0xffa80000, bank 5

NAND device: Manufacturer ID: 0xec, Chip ID: 0xd3 (Samsung NAND 1GiB 3,3V 8-bit)

nand_bbt: ECC error while reading bad block table

RedBoot partition parsing not available

eLBC NAND device at 0xffac0000, bank 6

sh-2.05b# cat /proc/mtd

dev:   size  erasesize  name
```

```
mtdd0: 02000000 00020000 "NAND - U-Boot Image"
mtdd1: 10000000 00020000 "NAND - JFFS2 Root File System"
mtdd2: 08000000 00020000 "NAND - RAMDISK File System"
mtdd3: 04000000 00020000 "NAND - Linux Kernel Image"
mtdd4: 01000000 00020000 "NAND - DTB Image"
mtdd5: 21000000 00020000 "NAND - Empty"
mtdd6: 40000000 00020000 "ffa40000.flash"
mtdd7: 40000000 00020000 "ffa80000.flash"
mtdd8: 40000000 00020000 "ffac0000.flash"
...

/* log in kernel */
-sh-2.05b #
-sh-2.05b # mount -t jffs2 /dev/mtdblock2 /mnt/cdrom/
-sh-2.05b # ls /mnt/cdrom
bin  dev  home  linuxrc  opt   root  sys  usr

boot  etc  lib   mnt      proc  sbin  tmp  var
-sh-2.05b # umount /mnt/cdrom
```

Another way to test NAND JFFS2 support is to boot up kernel and mount JFFS2 filesystem

Note: the block number should be the NAND JFFS2 partition's mtdblock number

Under U-Boot prompt:

```
setenv bootargs root=/dev/mtdblock2 rootfstype=jffs2 rw console=ttyS0,115200
tftp 1000000 uImage
tftp c00000 dtb
bootm 1000000 - c00000
```

5.5.3 Integrated Flash Controller NOR Flash User Manual

Description

NXP's Integrated Flash Controller can be used to connect various types of flashes e.g. NOR/NAND on board for boot functionality as well as data storage.

U-Boot Configuration

Compile time options

Below are major u-boot configuration options related to this feature defined in platform specific config files under include/configs/ directory.

Option Identifier	Description
CONFIG_FSL_IFC	Enable IFC support
CONFIG_FLASH_CFI_DRIVER CONFIG_SYS_FLASH_CFI CONFIG_SYS_FLASH_EMPTY_INFO	Enable CFI Driver for NOR Flash devices

Source Files

The following source files are related to this feature in u-boot.

Source File	Description
./drivers/misc/fsl_ifc.c	Set up the different chip select parameters from board header file
drivers/mtd/cfi_flash.c	CFI driver support for NOR flash devices

Kernel Configure Options

Tree View

Below are the configure options need to be set/unset while doing "make menuconfig" for kernel

Kernel Configure Tree View Options	Description
<pre> Device Drivers ---> <*> Memory Technology Device (MTD) support ---> [*] MTD partitioning support [*] Command line partition table parsing <*> Flash partition map based on OF description <*> Direct char device access to MTD devices -*- Common interface to block layer for MTD 'translation layers' <*> Caching block device access to MTD devices < > FTL (Flash Translation Layer) support RAM/ROM/Flash chip drivers ---> <*> Detect flash chips by Common Flash Interface (CFI) probe <*> Support for Intel/Sharp flash chips </pre>	<p>These options enable CFI support for NOR Flash under MTD subsystem and Integrated Flash Controller support on Linux</p>
<i>Table continues on the next page...</i>	

Table continued from the previous page...

Kernel Configure Tree View Options	Description
<pre> <*> Support for AMD/Fujitsu/Spansion flash chips Mapping drivers for chip access ---> <*> Flash device in physical memory map based on OF description </pre>	
<pre> File systems ---> [*] Miscellaneous filesystems ---> <*> Journalling Flash File System v2 (JFFS2) support </pre>	This option enables JFFS2 file system support for MTD Devices

Identifier

Below are the configure identifiers which are used in kernel source code and default configuration files.

Special Configure needs to be enabled("Y") for LS1021 and LS1043. Please find in below table with default value as "N"

Option	Values	Default Value	Description
CONFIG_FSL_IFC	Y/N	Y	Integrated Flash Controller support
CONFIG_MTD	Y/N	Y	Memory Technology Device (MTD) support
CONFIG_MTD_PARTITIONS	Y/N	Y	MTD partitioning support
CONFIG_MTD_CMDLINE_PARTS	Y/N	Y	Allow generic configuration of the MTD partition tables via the kernel command line.
CONFIG_MTD_OF_PARTS	Y/N	Y	This provides a partition parsing function which derives the partition map from the children of the flash nodes described in Documentation/powerpc/booting-without-of.txt
CONFIG_MTD_CHAR	Y/N	Y	Direct char device access to MTD devices
CONFIG_MTD_BLOCK	Y/N	Y	Caching block device access to MTD devices
CONFIG_MTD_CFI	Y/N	Y	Detect flash chips by Common Flash Interface (CFI) probe
CONFIG_MTD_GEN_PROBE	Y/N	Y	NA
CONFIG_MTD_MAP_BANK_WIDTH_1	Y/N	Y	Support 8-bit buswidth
CONFIG_MTD_MAP_BANK_WIDTH_2	Y/N	Y	Support 16-bit buswidth
CONFIG_MTD_MAP_BANK_WIDTH_4	Y/N	Y	Support 32-bit buswidth
CONFIG_MTD_PHYSMAP_OF	Y/N	Y	Flash device in physical memory map based on OF description
CONFIG_FTL	Y/N	N	FTL (Flash Translation Layer) support

Table continues on the next page...

Table continued from the previous page...

Option	Values	Default Value	Description
CONFIG_MTD_CFI_INTELEXT	Y/N	Y	Support for Intel/Sharp flash chips
CONFIG_MTD_CFI_AMDSTD	Y/N	Y	Support for AMD/Fujitsu/Spansion flash chips
CONFIG_MTD_CFI_ADV_OPTIONS	Y/N	N	Enable only for LS1021 and LS1043
CONFIG_MTD_CFI_BE_BYTE_SWAP	Y/N	N	Enable only for LS1021 and LS1043

Device Tree Binding

Documentation/devicetree/bindings/memory-controllers/fsl/ifc.txt

Flash partitions are specified by platform device tree.

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/memory/fsl_ifc.c	Integrated Flash Controller driver to handle error interrupts
drivers/mtd/mtdpart.c	Simple MTD partitioning layer
drivers/mtd/mtdblock.c	Direct MTD block device access
drivers/mtd/mtdchar.c	Character-device access to raw MTD devices.
drivers/mtd/ofpart.c	Flash partitions described by the OF (or flattened) device tree
drivers/mtd/ftl.c	FTL (Flash Translation Layer) support
drivers/mtd/chips/cfi_probe.c	Common Flash Interface probe
drivers/mtd/chips/cfi_util.c	Common Flash Interface support
drivers/mtd/chips/cfi_cmdset_0001.c	Support for Intel/Sharp flash chips
drivers/mtd/chips/cfi_cmdset_0002.c	Support for AMD/Fujitsu/Spansion flash chips

Verification in U-Boot

Test the Read/Write/Erase functionality of NOR Flash

1. Boot the u-boot with above config options to get NOR Flash access enabled. Check this in boot log,

```
FLASH: * MiB
```

where * is the size of NOR Flash

2. Erase NOR Flash
3. Make test pattern on memory e.g. DDR
4. Write test pattern on NOR Flash
5. Read the test pattern from NOR Flash to memory e.g. DDR
6. Compare the test pattern data to verify functionality.

Test Log :

Test log with initial u-boot log removed

```
--  
--  
  
FLASH: 32 MiB  
L2: 256 KB enabled  
--  
--  
/* u-boot prompt */  
=> mw.b 1000000 0xa5 10000  
=> md 1000000  
01000000: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
01000010: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
01000020: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
01000030: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
01000040: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
01000050: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
01000060: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
01000070: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
01000080: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
01000090: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
010000a0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
010000b0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
010000c0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
010000d0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
010000e0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
010000f0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....  
=> protect off all  
Un-Protect Flash Bank # 1  
=> erase ee000000 ee01ffff  
  
. done  
Erased 1 sectors  
=> cp.b 1000000 ee000000 10000  
Copy to Flash... 9....8....7....6....5....4....3....2....1....done  
=> cmp.b 1000000 ee000000 10000  
Total of 65536 bytes were the same  
=>
```

Verification in Linux

To cross check whether IFC NOR driver has been configured in the kernel or not, see the kernel boot log with following entries. Please note mtd partition number can be changed depending upon device tree.

```
ee000000.nor: Found 1 x16 devices at 0x0 in 16-bit bank  
  
Amd/Fujitsu Extended Query Table at 0x0040  
  
number of CFI chips: 1  
  
RedBoot partition parsing not available  
  
Creating 4 MTD partitions on "ee000000.nor":  
  
0x0000000040000-0x000000080000 : "NOR DTB Image"
```

```
ata1: Signature Update detected @ 0 msecs

0x000000080000-0x000000780000 : "NOR Linux Kernel Image"

0x000000800000-0x000001c00000 : "NOR JFFS2 Root File System"

0x000001f00000-0x000002000000 : "NOR U-Boot Image"
```

To verify NOR flash device accesses see the following test,

```
[root@ root]# cat /proc/mtd

dev:   size  erasesize  name

mtd0: 00040000 00020000 "NOR DTB Image"

mtd1: 00700000 00020000 "NOR Linux Kernel Image"

mtd2: 01400000 00020000 "NOR JFFS2 Root File System"

mtd3: 00100000 00020000 "NOR U-Boot Image"

mtd4: 00100000 00004000 "NAND U-Boot Image"

mtd5: 00100000 00004000 "NAND DTB Image"

mtd6: 00400000 00004000 "NAND Linux Kernel Image"

mtd7: 00400000 00004000 "NAND Compressed RFS Image"

mtd8: 00f00000 00004000 "NAND JFFS2 Root File System"

mtd9: 00700000 00004000 "NAND User area"

mtd10: 00080000 00010000 "SPI (RO) U-Boot Image"

mtd11: 00080000 00010000 "SPI (RO) DTB Image"

mtd12: 00400000 00010000 "SPI (RO) Linux Kernel Image"

mtd13: 00400000 00010000 "SPI (RO) Compressed RFS Image"

mtd14: 00700000 00010000 "SPI (RW) JFFS2 RFS"

[root@ root]# flash_eraseall -j /dev/mtd2

Erasing 128 Kibyte @ 1400000 -- 100% complete. Cleanmarker written at 13e0000.

[root@P1010RDB root]# mount -t jffs2 /dev/mtdblock2 /mnt/

JFFS2 notice: (1202) jffs2_build_xattr_subsystem: complete building xattr subsystem, 0 of xdatum
(0 unchecked, 0 orphan) and 0 of xref (0 dead, 0 orphan) found.

[root@ root]# cd /mnt/

[root@ mnt]# ls -l

[root@ mnt]# touch flash_file
```

```
[root@ root]# umount mnt
//ls must list local_file
[root@ root]# ls mnt
//mount again
[root@ root]# mount -t jffs2 /dev/mtdblock2 /mnt/
JFFS2 notice: (1219) jffs2_build_xattr_subsystem: complete building xattr subsystem, 0 of xdatum
(0 unchecked, 0 orphan) and 0 of xref (0 dead, 0 orphan) found.
//use ls ; it must show the created file
[root@ root]# ls /mnt/
flash_file
//unmount
[root@ root]# umount /mnt/
```

5.5.4 Integrated Flash Controller NAND Flash User Manual

Description

NXP's Integrated Flash Controller can be used to connect various types of flashes (e.g. NOR/NAND) on board for boot functionality as well as data storage.

U-Boot Configuration

Compile time options

Below are major U-Boot configuration options related to this feature defined in platform specific config files under include/configs/ directory.

Option Identifier	Description
CONFIG_FSL_IFC	Enable IFC support
CONFIG_NAND_FSL_IFC	Enable NAND Machine support on IFC
CONFIG_SYS_MAX_NAND_DEVICE	No of NAND Flash chips on platform
CONFIG_MTD_NAND_VERIFY_WRITE	Verify NAND flash writes
CONFIG_CMD_NAND	Enable various commands support for NAND Flash
CONFIG_SYS_NAND_BLOCK_SIZE	Block size of the NAND flash connected on Platform

Source Files

The following source files are related to this feature in u-boot.

Source File	Description
./drivers/misc/fsl_ifc.c	Set up the different chip select parameters from board header file
drivers/mtd/nand/fsl_ifc_nand.c	IFC nand flash machine driver file

Kernel Configure Options

Tree View

Below are the configure options need to be set/unset while doing "make menuconfig" for kernel

Kernel Configure Tree View Options	Description
<pre> Device Drivers ---> <*> Memory Technology Device (MTD) support ---> [*] MTD partitioning support [*] Command line partition table parsing= <*> Flash partition map based on OF description <*> Direct char device access to MTD devices -* - Common interface to block layer for MTD 'translation layers' <*> Caching block device access to MTD devices <*> NAND Device Support ---> <*> NAND support for Freescale IFC controller Enable UBIFS filesystem in linux configuration Device Drivers ---> <*> Memory Technology Device (MTD) support ---> UBI - Unsorted block images ---> <*> Enable UBI (4096) UBI wear-leveling threshold (1) Percentage of reserved eraseblocks for bad eraseblocks handling < > MTD devices emulation driver (gluebi) *** UBI debugging options *** </pre>	<p>These options enable Integrated Flash Controller NAND support to work with MTD subsystem available on Linux.</p> <p>Also UBIFS support needs to be enabled.</p>

Kernel Configure Tree View Options	Description
<pre> [] UBI debugging File systems ---> [*] Miscellaneous filesystems ---> <*> UBIFS file system support [*] Extended attributes support [] Advanced compression options [] Enable debugging </pre>	

Identifier

Below are the configure identifiers which are used in kernel source code and default configuration files.

Option	Values	Default Value	Description
CONFIG_FSL_IFC	y/n	y	Enable Integrated Flash Controller support
CONFIG_MTD_NAND_FSL_IFC	y/n	Y	Enable Integrated Flash Controller NAND Machine support
CONFIG_MTD_PARTITIONS	y/n	Y	MTD partitioning support
CONFIG_MTD_CMDLINE_PARTS	y/n	Y	Allow generic configuration of the MTD partition tables via the kernel command line.
CONFIG_MTD_OF_PARTS	y/n	Y	This provides a partition parsing function which derives the partition map from the children of the flash nodes described in Documentation/powerpc/booting-without-of.txt
CONFIG_MTD_CHAR	y/n	Y	Direct char device access to MTD devices
CONFIG_MTD_BLOCK	y/n	Y	Caching block device access to MTD devices
CONFIG_MTD_GEN_PROBE	y/n	Y	NA
CONFIG_MTD_PHYSMAP_OF	y/n	Y	Flash device in physical memory map based on OF description

Device Tree Binding

Documentation/devicetree/bindings/memory-controllers/fsl/ifc.txt

Flash partitions are specified by platform device tree.

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/memory/fsl_ifc.c	Integrated Flash Controller driver to handle error interrupts
drivers/mtd/nand/fsl_ifc_nand.c	Integrated Flash Controller NAND Machine driver
include/linux/fsl_ifc.h	IFC Memory Mapped Registers

Verification in U-Boot

Test the Read/Write/Erase functionality of NAND Flash

1. Boot the u-boot with above config options to get NAND Flash driver enabled. Check this in boot log,

```
NAND: * MiB
Where * is NAND flash size
```

2. Erase NAND Flash
3. Make test pattern on memory e.g. DDR
4. Write test pattern on NAND Flash
5. Read the test pattern from NAND Flash to memory e.g DDR
6. Compare the test pattern data to verify functionality.

Test Log :

```
...
...

Flash: 32 MiB

L2:    256 KB enabled

NAND:  32 MiB

...
...

/* U-boot prompt */
=> nand erase.chip

NAND erase.chip: device 0 whole chip

Bad block table found at page 65504, version 0x01 Bad block table found at page 65472, version
0x01

Skipping bad block at  0x01ff0000

Skipping bad block at  0x01ff4000

Skipping bad block at  0x01ff8000

Skipping bad block at  0x01ffc000
```

Linux Kernel Drivers
Flash Memory

```
OK

=> mw.b 1000000 0xa5 100000

=> md 1000000

01000000: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....
01000010: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....
01000020: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....
01000030: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....
01000040: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....
01000050: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....
01000060: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....
01000070: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....
01000080: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....
01000090: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....
010000a0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....
010000b0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....
010000c0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....
010000d0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....
010000e0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....
010000f0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5 .....

=> nand write 1000000 0 100000

NAND write: device 0 offset 0x0, size 0x100000

1048576 bytes written: OK

=> nand read 2000000 0 100000

NAND read: device 0 offset 0x0, size 0x100000

1048576 bytes read: OK

=> cmp.b 1000000 2000000 100000
```

```
Total of 1048576 bytes were the same
```

Verification in Linux

To cross check whether IFC NAND driver has been configured in the kernel or not, check the following. Please note mtd partition numbers can be changed depending upon board device tree

```
[root@(none) root]# cat /proc/mtd

dev:   size   erasesize  name

mtd0: 00100000 00004000 "NAND U-Boot Image"

mtd1: 00100000 00004000 "NAND DTB Image"

mtd2: 00400000 00004000 "NAND Linux Kernel Image"

mtd3: 00400000 00004000 "NAND Compressed RFS Image"

mtd4: 00f00000 00004000 "NAND Root File System"

mtd5: 00700000 00004000 "NAND User area"

mtd6: 00080000 00010000 "SPI (RO) U-Boot Image"

mtd7: 00080000 00010000 "SPI (RO) DTB Image"

mtd8: 00400000 00010000 "SPI (RO) Linux Kernel Image"

mtd9: 00400000 00010000 "SPI (RO) Compressed RFS Image"

mtd10: 00700000 00010000 "SPI (RW) JFFS2 RFS"

[root@(none) root]# flash_eraseall /dev/mtd4 Erasing 16 Kibyte @ f00000 -- 100% complete.

[root@(none) root]# ubiattach /dev/ubi_ctrl -m 4

UBI: attaching mtd4 to ubi0

UBI: physical eraseblock size:   16384 bytes (16 KiB)

UBI: logical eraseblock size:    15360 bytes

UBI: smallest flash I/O unit:    512

UBI: VID header offset:         512 (aligned 512)

UBI: data offset:               1024

UBI: empty MTD device detected

UBI: create volume table (copy #1)
```

Linux Kernel Drivers
Flash Memory

```
UBI: create volume table (copy #2)

UBI: attached mtd4 to ubi0

UBI: MTD device name:           "NAND Root File System"

UBI: MTD device size:           15 MiB

UBI: number of good PEBs:       960

UBI: number of bad PEBs:        0

UBI: max. allowed volumes:      89

UBI: wear-leveling threshold:   4096

UBI: number of internal volumes: 1

UBI: number of user volumes:    0

UBI: available PEBs:            947

UBI: total number of reserved PEBs: 13

UBI: number of PEBs reserved for bad PEB handling: 9

UBI: max/mean erase counter: 0/0

UBI: image sequence number: 0

UBI: background thread "ubi_bgt0d" started, PID 7541 UBI device number 0, total 960 LEBs
(14745600 bytes, 14.1 MiB), available 947 LEBs (14545920 bytes, 13.9 MiB), LEB size 15360 bytes
(15.0 KiB)

[root@(none) root]# ubimkvol /dev/ubi0 -N rootfs -s 14205KiB Volume ID 0, size 947 LEBs (14545920
bytes, 13.9 MiB), LEB size 15360 bytes (15.0 KiB), dynamic, name "rootfs", alignment 1

[root@(none) root]# mount -t ubifs /dev/ubi0_0 /mnt/

UBIFS: default file-system created

UBIFS: mounted UBI device 0, volume 0, name "rootfs"

UBIFS: file system size: 14361600 bytes (14025 KiB, 13 MiB, 935 LEBs)
UBIFS: journal size: 721920 bytes (705 KiB, 0 MiB, 47 LEBs)

UBIFS: media format: w4/r0 (latest is w4/r0)

UBIFS: default compressor: lzo

UBIFS: reserved for root: 678333 bytes (662 KiB)

[root@(none) root]# cd /mnt/
```

```
[root@(none) mnt]# ls

[root@(none) mnt]# touch flash_file

[root@(none) mnt]# ls -l

total 0

-rw-r--r-- 1 root root 0 Jul  6 14:45 flash_file

[root@(none) mnt]# cd

[root@(none) root]# umount /mnt/

UBIFS: un-mount UBI device 0, volume 0

[root@(none) root]# mount -t ubifs /dev/ubi0_0 /mnt/

UBIFS: mounted UBI device 0, volume 0, name "rootfs"

UBIFS: file system size: 14361600 bytes (14025 KiB, 13 MiB, 935 LEBs)
UBIFS: journal size: 721920 bytes (705 KiB, 0 MiB, 47 LEBs)

UBIFS: media format: w4/r0 (latest is w4/r0)

UBIFS: default compressor: lzo

UBIFS: reserved for root: 678333 bytes (662 KiB)

[root@(none) root]# ls -l /mnt/

total 0

-rw-r--r-- 1 root root 0 Jul  6 14:45 flash_file
```

Known Bugs, Limitations, or Technical Issues

Boards which have NAND Flash with 512byte page size, JFFS2 cannot be supported using H/W ECC support of IFC , as there is not enough remaining space in the OOB area.

To use JFFS2 use SOFT ECC.

5.6 IEEE 1588 Device Driver User Manual

5.6.1 IEEE 1588 Device Driver User Manual

Description

From IEEE-1588 perspective the components required are:

1. IEEE-1588 extensions to the gianfar driver or DPAA driver.

2. A stack application for IEEE-1588 protocol.

Module Loading

IEEE 1588 device driver support either kernel built-in or module

Kernel Configure Tree View Options

1. eTSEC - Using IXXAT stack

Kernel Configure Tree View Options	Description
<pre>Device Drivers ---> [*]Network device support ---> [*]Ethernet driver support ---> <*> Gianfar Ethernet [*] Gianfar 1588</pre>	Enable 1588 driver for IXXAT stack

2. eTSEC - Using PTPd stack

Kernel Configure Tree View Options	Description
<pre>Device Drivers ---> PTP clock support ---> <*> Freescale eTSEC as PTP clock</pre>	Enable 1588 driver for PTPd stack

3. DPAA - Using IXXAT stack

Kernel Configure Tree View Options	Description
<pre>Device Drivers ---> [*]Network device support ---> [*]Ethernet driver support ---> [*]Freescale devices ---> <*>IEEE 1588-compliant timestamping Optimization choices for the DPAA Ethernet driver ---> (X)Optimize for forwarding</pre>	Enable IEEE 1588-compliant timestamping

4. DPAA - Using PTPd stack

Kernel Configure Tree View Options	Description
<pre>Device Drivers ---> PTP clock support ---> <*> Freescale dtSEC as PTP clock</pre>	Enable 1588 driver for PTPd stack

Compile-time Configuration Options

For eTSEC (IXXAT)

Option	Values	Default Value	Description
CONFIG_GIANFAR	y/n	y	Enable eTSEC driver support
CONFIG_FSL_GIANFAR_1588	y/n	n	Enables 1588 driver support

For eTSEC (PTPd)

Option	Values	Default Value	Description
CONFIG_GIANFAR	y/n	y	Enable eTSEC driver support
CONFIG_PTP_1588_CLOCK_GIANFAR	y/n	y	Enables 1588 driver support

For DPAA (IXXAT)

Option	Values	Default Value	Description
CONFIG_FSL_DPAA_1588	y/n	n	Enable IEEE 1588 support
CONFIG_FSL_SDK_DPAA_ETH	y/n	y	Enables DPAA driver support

For DPAA (PTPd)

Option	Values	Default Value	Description
CONFIG_PTP_1588_CLOCK_DPAA	y/n	n	Enable IEEE 1588 support
CONFIG_FSL_SDK_DPAA_ETH	y/n	y	Enables DPAA driver support

Source Files

The driver source is maintained in the Linux kernel source tree.

1. eTSEC (for IXXAT)

Source File	Description
drivers/net/ethernet/freescale/gianfar.c	IEEE 1588 hooks in the Ethernet driver
drivers/net/ethernet/freescale/gianfar_1588.c	IEEE 1588 driver

2. eTSEC (for PTPd)

Source File	Description
drivers/net/ethernet/freescale/gianfar.c	IEEE 1588 hooks in the Ethernet driver
drivers/net/ethernet/freescale/gianfar_ptp.c	IEEE 1588 driver

3. DPAA (for IXXAT)

Source File	Description
drivers/net/ethernet/freescale/sdk_dpaa/dpaa_1588.c	IEEE 1588 driver support
drivers/net/ethernet/freescale/sdk_dpaa/dpaa_1588.h	IEEE 1588 driver head file
drivers/net/ethernet/freescale/sdk_dpaa/dpaa_eth.c	DPAA Ethdriver support

4. DPAA (for PTPd)

Source File	Description
drivers/net/ethernet/freescale/sdk_dpaa/dpaa_ptp.c	IEEE 1588 driver support
drivers/net/ethernet/freescale/sdk_dpaa/dpaa_eth.c	DPAA Ethdriver support

Device Tree Binding

1. eTSEC (for IXXAT)

Property	Type	Status	Description
compatible	String	Required	Should be 'fsl,gianfar-ptp-timer' for IEEE 1588 driver

```

Default node:
    ptp_timer: ptimer@24e00 {
        compatible = "fsl,gianfar-ptp-timer";
        reg = <0x24e00 0xb0>;
        fsl,ts-to-buffer;
        fsl,tmr-prsc = <0x2>;
        fsl,clock-source-select = <1>;
    };

    enet0: ethernet@24000 {
        .....
        ptimer-handle = <&ptp_timer>;
    };
  
```

2. eTSEC (for PTPd)

Property	Type	Status	Description
compatible	String	Required	Should be 'fsl,etsec-ptp' for IEEE 1588 driver

```

Default node:
    ptp_clock@b0e00 {
        compatible = "fsl,etsec-ptp";
        reg = <0xb0e00 0xb0>;
        interrupts = <68 2 0 0 69 2 0 0>;
        fsl,tclk-period = <10>;
        fsl,tmr-prsc = <2>;
        fsl,tmr-add = <0x80000016>;
        fsl,tmr-fiper1 = <0x3b9ac9f6>;
        fsl,tmr-fiper2 = <0x00018696>;
    };
  
```

```
fsl,max-adj = <199999999>;
};
```

3. DPAA

Property	Type	Status	Description
compatible	String	Required	Should be 'fsl,fman-rtc'
reg	integer	Required	Register map

```
Default node:
fman0: fman@400000 {
    #address-cells = <1>;
    #size-cells = <1>;
    cell-index = <0>;
    compatible = "fsl,p4080-fman", "fsl,fman", "simple-bus";

    enet0: ethernet@e0000 {
        cell-index = <0>;
        compatible = "fsl,p4080-fman-1g-mac", "fsl,fman-1g-mac";
        reg = <0xe0000 0x1000>;
        fsl,port-handles = <&fman0_rx0 &fman0_tx0>;
        tbi-handle = <&tbi0>;
        phy-handle = <&phy0>;
        phy-connection-type = "sgmii";
        ptimer-handle = <&ptp_timer0>;
    };

    ...enet1/2...

    enet3: ethernet@e6000 {
        cell-index = <3>;
        compatible = "fsl,p4080-fman-1g-mac", "fsl,fman-1g-mac";
        reg = <0xe6000 0x1000>;
        fsl,port-handles = <&fman0_rx3 &fman0_tx3>;
        tbi-handle = <&tbi3>;
        phy-handle = <&phy3>;
        phy-connection-type = "sgmii";
        ptimer-handle = <&ptp_timer0>;
    };

    ptp_timer0: rtc@fe000 {
        compatible = "fsl,fman-rtc";
        reg = <0xfe000 0x1000>;
    };
};

fman1: fman@500000 {
    #address-cells = <1>;
    #size-cells = <1>;
    cell-index = <1>;
    compatible = "fsl,p4080-fman", "fsl,fman", "simple-bus";

    enet5: ethernet@e0000 {
        cell-index = <0>;
        compatible = "fsl,p4080-fman-1g-mac", "fsl,fman-1g-mac";
        reg = <0xe0000 0x1000>;
```

```
fsl,port-handles = <&fman1_rx0 &fman1_tx0>;
tbi-handle = <&tbi5>;
phy-handle = <&phy5>;
phy-connection-type = "sgmii";
ptimer-handle = <&ptp_timer1>;
};

...enet6/7...

enet8: ethernet@e6000 {
    cell-index = <3>;
    compatible = "fsl,p4080-fman-1g-mac", "fsl,fman-1g-mac";
    reg = <0xe6000 0x1000>;
    fsl,port-handles = <&fman1_rx3 &fman1_tx3>;
    tbi-handle = <&tbi8>;
    phy-handle = <&phy8>;
    phy-connection-type = "sgmii";
    ptimer-handle = <&ptp_timer1>;
};

ptp_timer1: rtc@fe000 {
    compatible = "fsl,fman-rtc";
    reg = <0xfe000 0x1000>;
};
};
```

Verification in Linux and test procedure

Connect two boards through crossover, ex, eth1 to eth1, and connect the other ethernet port on each board to switch or PC. One board runs as master, and the other as slave.

- **Using the IXXAT stack image**

1. Enable IEEE-1588 support in the gianfar/DPAA driver, rebuild and reload it. You should get the following message during system boot:

```
...
IEEE1588: ptp 1588 is initialized.
...
```

- On the master side:

```
# ifconfig eth1 192.168.1.100 allmulti up
# ./ptp -i 0:eth1 -do //run stack application
```

Note: On a DPAA platform, use fm1-gb1 instead of eth1

- On the slave side :

```
# ifconfig eth1 192.168.1.200 allmulti up
# ./ptp -i 0:eth1 -do //run stack application
```

Note: On a DPAA platform, use the desired interface(e.g. fm1-gb1) instead of eth1

2. You should start getting synchronization messages on the slave side.

NOTE

A detailed configuration of the IEEE-1588 stack application is described in a Quick Start Guide provided with the stack application.

• Using the PTPd stack image

1. Enable IEEE-1588 support in the gianfar driver, rebuild and reload it. You should get the following message during system boot:

```
...  
pps pps0: new PPS source ptp0  
...
```

- On the master side:

```
# ifconfig eth0 up  
  
# ifconfig eth0 192.168.1.100  
  
# ./ptpd -i eth0 -MV //run stack application
```

Note: On a DPAA platform, use the desired interface(e.g. fm1-gb1) instead of eth0.

- On the slave side :

```
# ifconfig eth0 up  
  
# ifconfig eth0 192.168.1.200  
  
# ./ptpd -i eth0 -sV --servo:kp=0.32 --servo:ki=0.05 //run stack application
```

Note: On a DPAA platform, use the desired interface(e.g. fm1-gb1) instead of eth0.

2. You should start getting synchronization messages on the slave side.

Known Bugs, Limitations, or Technical Issues

- For DPAA, the PTPd stack limits to use only one ptp timer, so only the interfaces on the second FMAN(such as fm2-gb2) are available for PTPd if the platform has two FMANs.
- The IEEE 1588 conflicts with SGMII 10/100M mode. 1588 TimeStamp is supported in all Gbps modes(both RGMII and SGMII) and all full-duplex 10/100 modes except 10/100 SGMII mode.
- For eTSEC, the slave PPS signal (available on TSEC_1588_PULSE_OUT1 pin) is not phase aligned with master PPS signal. This is a known limitation.
- For eTSEC, running IEEE-1588 function need disable CONFIG_RX_TX_BUFF_XCHG. In default SDK kernel image, the ASF was enabled, so CONFIG_RX_TX_BUFF_XCHG was enabled too. you should disable CONFIG_RX_TX_BUFF_XCHG and rebuild kernel for 1588 function.
- VLAN feature was disable in default kernel configuration. So, if 1588 over VLAN is used in your application or solution, you need to enable VLAN in Linux kernel by yourself.
- For DPAA, the PTPd stack limits to use only one ptp timer, so only the interfaces on the second FMAN(such as fm2-gb2) are available for PTPd if the platform has two FMAN.

5.7 Low Power UART User Guide

5.7.1 Low Power UART User Guide

Description

Low Power Universal asynchronous receiver/transmitter (LPUART) is a high speed and low power uart. Refer to below table for the NXP soc can support LPUART.

SOC	Num of LPUART module
LS1021A	6
LS1043A	6

U-boot ConfigurationCompile time options

Below are major u-boot configuration options related to this feature defined in platform specific config files under include/configs/ directory.

Option Identifier	Description
CONFIG_LPUART	Enable lpuart support
CONFIG_FSL_LPUART	Enable NXP lpuart support
CONFIG_LPUART_32B_REG	Select 32-bit lpuart register mode

Choosing predefined u-boot board configs:

Please make the defconfig include 'lpuart', like: ls1021atwr_nor_lpuart_defconfig. That's will support lpuart.

Runtime options

Env Variable	Env Description	Sub option	Option Description
bootargs	Kernel command line argument passed to kernel	console=ttyLP0,1152000	select LPUART0 as the system console

Kernel Configure Options

Tree View

Below are the configure options need to be set/unset while doing "make menuconfig" for kernel

Kernel Configure Tree View Options	Description
<pre> Device Drivers ---> Character devices ---> Serial drivers ---> </pre>	LPUART driver and enable console support

Kernel Configure Tree View Options	Description
<pre><*> Freescale lpuart serial port support [*] Console on Freescale lpuart serial port</pre>	

Identifier

Below are the configure identifiers which are used in kernel source code and default configuration files.

Option	Values	Default Value	Description
CONFIG_SERIAL_FSL_LPUART	y/m/n	n	LPUART Driver

Device Tree Binding

Below is an example device tree node required by this feature. Note that it may has differences among platforms.

```
lpuart0: serial@2950000 {
    compatible = "fsl,vf610-lpuart";
    reg = <0x0 0x2950000 0x0 0x1000>;
    interrupts = <GIC_SPI 80 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&sysclk>;
    clock-names = "ipg";
    fsl,lpuart32;
    status = "okay";
}
```

Source Files

The following source file are related the this feature in u-boot.

Source File	Description
drivers/serial/serial_lpuart.c	The LPUART driver file

The following source file are related the this feature in Linux kernel.

Source File	Description
drivers/tty/serial/fsl_lpuart.c	The LPUART driver file

Verification in U-Boot

1. Boot up U-Boot from bank0, and update rcw and u-boot for lpuart support to bank4, first copy the rcw and U-Boot binary to the tftp directory.
2. Please refer to the platform depoly document to update the rcw and uboot.
3. After all is updated, run u-boot command to switch to alt bank, then will bring up the new U-Boot to the lpuart console.

```
CPU: Freescale LayerScape LS1020E, Version: 1.0, (0x87081010)
Clock Configuration:
  CPU0(ARMV7):1000 MHz,
  Bus:300 MHz, DDR:800 MHz (1600 MT/s data rate),
Reset Configuration Word (RCW):
  00000000: 0608000a 00000000 00000000 00000000
```

```
00000010: 60000000 00407900 e0025a00 21046000
00000020: 00000000 00000000 00000000 08038000
00000030: 00000000 001b7200 00000000 00000000
I2C: ready
Board: LS1021ATWR
CPLD: V2.0
PCBA: V1.0
VBank: 0
DRAM: 1 GiB
Using SERDES1 Protocol: 48 (0x30)
Flash: 0 Bytes
MMC: FSL_SDHC: 0
EEPROM: NXID v16777216
PCIe1: Root Complex no link, regs @ 0x3400000
PCIe2: disabled
In: serial
Out: serial
Err: serial
SATA link 0 timeout.
AHCI 0001.0300 1 slots 1 ports ? Gbps 0x1 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc
Found 0 device(s).
SCSI: Net: eTSEC1 is in sgmi mode.
eTSEC2 is in sgmi mode.
eTSEC1, eTSEC2 [PRIME], eTSEC3
=>
```

Verification in Linux

1. After uboot startup, set the command line parameter to pass to the linux kernel including console=ttyLP0,115200 in bootargs. For deploy the ramdisk as rootfs, the bootargs can be set as: "set bootargs root=/dev/ram0 rw console=ttyLP0,115200"

```
=> set bootargs root=/dev/ram0 rw console=ttyLP0,115200

=> dhcp 81000000 <tftpboot dir>/zImage.ls1021a;tftp 88000000 <tftpboot dir>/
initrd.ls1.uboot;tftp 8f000000 <tftpboot dir>/ls1021atwr.dtb;bootz 81000000 88000000 8f000000

[...]

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
Booting Linux on physical CPU 0xf00
Linux version 3.12.0+ (xxx@rock) (gcc version 4.8.3 20131202 (prerelease) (crosstool-NG
linaro-1.13.1-4.8-2013.12 - LinaroGCC 2013.11) ) #664 SMP Tue Jun 24 15:30:45 CST 2014
CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=30c73c7d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine: Freescale Layerscape LS1021A, model: LS1021A TWR Board
Memory policy: ECC disabled, Data cache writealloc
PERCPU: Embedded 7 pages/cpu @8901c000 s7936 r8192 d12544 u32768
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 520720
Kernel command line: root=/dev/ram rw console=ttyLP0,115200
PID hash table entries: 4096 (order: 2, 16384 bytes)

[...]
```

```
ls1021atwr login: root
root@ls1021atwr:~#
```

2. After the kernel boot up to the console, You can type any shell command in the LPUART TERMINAL.

5.8 PCI Express Interface Controller

5.8.1 PCIe Linux Driver

Module Loading

The MPC85xx/Layerscape PCIe host bridge support code is compiled into the kernel. It is not available as a module.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
<pre>Bus support ---> [*] PCI support [*] Message Signaled Interrupts (MSI and MSI-X)</pre>	Enable PCI host bridge and message support
<pre>Bus support ---> PCI host controller drivers ---> [*] Freescale Layerscape PCIe controller</pre>	Enable NXP Layerscape PCIe controller
<pre>Device Drivers ---> [*]Network device support ---> [*]Ethernet device support ---> [*] Intel devices ---> <*> Intel (R) PRO/1000 PCI-Express Gigabit Ethernet support</pre>	Intel PRO/1000 PCI-Express support
<pre>Device Drivers ---> <*> Serial ATA and Parallel ATA drivers (libata) ---> <*> Silicon Image 3124/3132 SATA support</pre>	Enable support for Silicon Image 3124/3132 Serial ATA.

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_PCI	y/n	y	Enable PCI host bridge
CONFIG_PCI_MSI	y/n	y	Message support
CONFIG_PCI_LAYERSCAPE	y/n	y	Enable PCI for Layerscape
CONFIG_E1000E	y/m/n	y	Enable Intel Pro/1000 driver
CONFIG_SATA_SIL	y/m/n	y	Silicon Image SATA support

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
arch/powerpc/sysdev/fsl_pci.c	The MPC85XX platform PCIE host bridge support source
drivers/pci/host/pci-layerscape.c	The Layerscape platform PCIE host bridge support source
drivers/net/ethernet/intel/e1000e/	Intel Pro/1000 driver source code
drivers/ata/sata_sil.c	Silicon Image source code

SATA Card Test Procedure

the user can use command
fdisk, mke2fs mount to operate the ide disk.
After kernel boots up, please follow the log to operate:

```
[root@pX0XX /root]# fdisk -l
```

```
Disk /dev/sda: 85.8 GB, 85899345920 bytes  
255 heads, 63 sectors/track, 10443 cylinders  
Units = cylinders of 16065 * 512 = 8225280 bytes
```

```
Disk /dev/sda doesn't contain a valid partition table
```

```
[root@pX0XX /root]# fdisk /dev/sda
```

```
Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel  
Building a new DOS disklabel. Changes will remain in memory only,  
until you decide to write them. After that the previous content  
won't be recoverable.
```

The number of cylinders for this disk is set to 10443.

There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:

- 1) software that runs at boot time (e.g., old versions of LILO)
- 2) booting and partitioning software from other OSs
(e.g., DOS FDISK, OS/2 FDISK)

```
Command (m for help): n
```

```
Command action
```

```
  e  extended
```

```
  p  primary partition (1-4)
```

```
p
```

```
Partition number (1-4): 1
```

```
First cylinder (1-10443, default 1): Using default value 1
```

```
Last cylinder or +size or +sizeM or +sizeK (1-10443, default 10443): 100
```

```
Command (m for help): w
```

```
The partition table has been altered!
```

```
Calling ioctl() to re-read partition table
```

```
sd 0:0:0:0: [sda] 167772160 512-byte hardware sectors (85899 MB)
```

```
sd 0:0:0:0: [sda] Write Protect is off
```

```
sd 0:0:0:0: [sda] Asking for cache data failed
sd 0:0:0:0: [sda] Assuming drive cache: write through
sda: sda1

[root@pX0XX /root]# mke2fs /dev/sda1
mke2fs 1.34 (25-Jul-2003)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
100576 inodes, 200804 blocks
10040 blocks (5.00%) reserved for the super user
First data block=0
7 block groups
32768 blocks per group, 32768 fragments per group
14368 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 31 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.

[root@pX0XX /root]# mkdir sda1_test
[root@pX0XX /root]# mount /dev/sda1 sda1_test/
[root@pX0XX /root]# cp /bin/tar sda1_test/
[root@pX0XX /root]#
```

Ethernet Card Test Procedure

- plug Intel Pro/1000e network card into standard PCI-E slot on a board. After linux bootup, ifconfig ethx ip address and netmask, then do ping testing.

Tips: x ethernet interface number, an example is as the following for Intel e1000 network card is eth0.

For example:

After kernel boot up, bring up with the pci Ethernet card

```
ifconfig ethx 192.168.20.100
```

ip address should not be conflicted with other Ethernet port.

In Linux window, run ping 192.168.20.101

Known Bugs, Limitations, or Technical Issues

- LSI-SAS card cannot be used on the second PCIe controller when system enables more than one PCIe controller. Use code modification below to workaround this issue:

```
--- a/arch/powerpc/sysdev/fsl_pci.c
+++ b/arch/powerpc/sysdev/fsl_pci.c
@@ -511,7 +511,7 @@ int __init fsl_add_bridge(struct platform_device *pdev, int is_primary)
     printk(KERN_WARNING "Can't get bus-range for %s, assume"
            " bus 0\n", dev->full_name);

-    pci_add_flags(PCI_REASSIGN_ALL_BUS);
```

```

+   pci_add_flags(PCI_ENABLE_PROC_DOMAINS);
   hose = pcibios_alloc_controller(dev);
   if (!hose)
       return -ENOMEM;
@@ -846,7 +846,7 @@ int __init mpc83xx_add_bridge(struct device_node *dev)
       " bus 0\n", dev->full_name);
   }

-   pci_add_flags(PCI_REASSIGN_ALL_BUS);
+   pci_add_flags(PCI_ENABLE_PROC_DOMAINS);
   hose = pcibios_alloc_controller(dev);
   if (!hose)
       return -ENOMEM;
    
```

5.8.2 EDAC Driver User Manual

Description

The EDAC kernel module's goal is to detect and report errors that occur within the computer system running under Linux.

Note

Currently the EDAC wasn't supported on P5040/P5020 64bit.

Module Loading

Linux EDAC Driver supports kernel built-in or module.

Kernel Configure Options

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
<pre> Device Drivers ---> <*> EDAC (Error Detection And Correction) reporting --- > <*> Main Memory EDAC (Error Detection And Correction) reporting <*> Freescale MPC83xx / MPC85xx </pre>	<p>Enables EDAC support for 85xx platform</p>

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_EDAC_MM_EDAC	y/n	y/n	Enables EDAC core support
CONFIG_EDAC_MPC85XX	y/n	y/n	Enables EDAC NXP 85xx support

Device Tree Binding

Property	Type	Status	Description
compatible	String	Required	Should be 'fsl,qoriq-memory-controller', 'fsl,p4080-pcie'
reg	integer	Required	Register map

Default node:

```

ddr1: memory-controller@8000 {
    compatible = "fsl,qoriq-memory-controller-v4.4", "fsl,qoriq-memory-controller";
    reg = <0x8000 0x1000>;
    interrupts = <16 2 1 23>;
};

/* controller at 0x200000 */
pci0 {
    compatible = "fsl,p4080-pcie";
    device_type = "pci";
    #size-cells = <2>;
    #address-cells = <3>;
    bus-range = <0x0 0xff>;
    clock-frequency = <33333333>;
    interrupts = <16 2 1 15>;
};

```

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/edac/edac_core.c	Enables EDAC core support
drivers/edac/mpc85xx_edac.c	Enables EDAC NXP 85xx support

Kernel boot message:

```

.....
.....
EDAC MC: Ver: 2.1.0
Freescale(R) MPC85xx EDAC driver, (C) 2006 Montavista Software
EDAC MC0: Giving out device to 'MPC85xx_edac' 'mpc85xx_mc_err': DEV mpc85xx_mc_err
MPC85xx_edac acquired irq 16 for MC
MPC85xx_edac MC err registered
EDAC MC1: Giving out device to 'MPC85xx_edac' 'mpc85xx_mc_err': DEV mpc85xx_mc_err
MPC85xx_edac acquired irq 16 for MC
MPC85xx_edac MC err registered
EDAC PCI0: Giving out device to module 'MPC85xx_edac' controller 'mpc85xx_pci_err': DEV
'ffe200000.pcie' (INTERRUPT)
MPC85xx_edac acquired irq 16 for PCI Err
MPC85xx_edac PCI err registered
EDAC PCI1: Giving out device to module 'MPC85xx_edac' controller 'mpc85xx_pci_err': DEV
'ffe201000.pcie' (INTERRUPT)
MPC85xx_edac acquired irq 16 for PCI Err

```

```
MPC85xx_edac PCI err registered
EDAC PCI2: Giving out device to module 'MPC85xx_edac' controller 'mpc85xx_pci_err': DEV
'ffe202000.pcie' (INTERRUPT)
MPC85xx_edac acquired irq 16 for PCI Err
MPC85xx_edac PCI err registered
Testing edac driver is start.
PCIE error(s) detected
PCIE ERR_DR register: 0x00020000
PCIE ERR_CAP_STAT register: 0x80000001
PCIE ERR_CAP_R0 register: 0x00000800
PCIE ERR_CAP_R1 register: 0x00000000
PCIE ERR_CAP_R2 register: 0x00000000
PCIE ERR_CAP_R3 register: 0x00000000
.....
.....
.....
p4080 login: root
Password:
[root@p4080 root]#
```

Test Procedure:

```
[root@p4080 root]#
[root@p4080 root]# cat /proc/interrupts |grep EDAC
16:          1          0          0          0          0          0          0          0
OpenPIC Level [EDAC] MC err, [EDAC] MC err, [EDAC] PCI err, [EDAC] PCI err, [EDAC] PCI err
[root@p4080 root]#
[root@p4080 root]#
```

Now, see that whether the total number of interrupt 16 of EDAC is zero or less than twenty. If it is that, EDAC driver is OK.

5.8.3 PCIe Advanced Error Reporting User Manual

Description

How to test the PCI Express Advanced Error Reporting (AER) function.

Testing the PCIe AER error recovery code in actual environment is quite difficult because it is hard to trigger real hardware errors. So we use a software tool based error injection to fake various kinds of PCIe errors.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
<pre>Bus options ---> [*] PCI Express support [*] Root Port Advanced Error Reporting support <*> PCIe AER error injector support</pre>	<p>enable PCI-Express AER and AER-INJECTOR in kernel</p>

Kernel compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_PCIEAER	y/n	y	Enable AER
CONFIG_PCIEAER_INJECT	y/n	n	Enables AER INJECT

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/pci/pcie/aer/*.c	AER driver support

• Prepare aer-inject test tool

```
1, Download aer-inject test utility.

2, Write a test config file
e.g. $ vi aer-cfg
    AER
    DOMAIN 0001
    BUS 1
    DEV 0
    FN 0
    COR_STATUS BAD_TLP
    HEADER_LOG 0 1 2 3

NOTE:
error type can be ["COR_STATUS", "UNCOR_STATUS"]

Corrected error can be:
["BAD_TLP", "RCVR", "BAD_DLLP", "REP_ROLL", "REP_TIMER"]

Uncorrected non-fatal error can be:
["POISON_TLP", "COMP_TIME", "COMP_ABORT", "UNX_COMP", "ECRC", "UNSUP"]

Uncorrected fatal error can be:
["TRAIN", "DLP", "FCP", "RX_OVER", "MALF_TLP"]
```

• Test Steps

```
1, insert a pcie device in PCI slot of board, ensure the pcie device has AER capability, e.g.
e1000e PCIe NIC network card.

2, In u-boot prompt, add "pcie_ports=native" in bootargs command-line.
=> setenv othbootargs pcie_ports=native

3, boot the kernel and filesystem.
=> tftp 1000000 uImage;tftp 2000000 board.dtb; tftp 3000000 rootfs.ext2.gz.uboot; bootm 1000000
3000000 2000000

4, check AER device and config
# zcat /proc/config.gz|grep -i CONFIG_PCIEAER_INJECT
CONFIG_PCIEAER_INJECT=y
```

```
# cat /proc/cmdline
root=/dev/ram rw console=ttyS0,115200 pcie_ports=native
check "pcie_ports=native" has been set.

# ls /dev/aer_inject
Check if the aer injector device is created.

# lspci
00:00.0 Class 0604: 1957:0410
01:00.0 Class 0200: 8086:10d3
e.g. here device "01:00.0" is the PCIe NIC e1000 network card in the test scenario.

5, Download aer-inject and aer-cfg from host to test-board
$ scp aer-inject aer-cfg root@test-board-ip:~

6, ensure the pcie device domain-number/bus-number/device-number/function-number in aer-cfg is
accordant to those in the output of lspci

7, Run aer-inject, corresponding error information will be reported as below and AER will recover
PCIe device according to the type of errors.
# ./aer-inject aer-cfg
example of error report as below:
pcieport 0000:00:00.0: AER: Corrected error received: id=0100
e1000e 0000:01:00.0: PCIe Bus Error: severity=Corrected, type=Data Link Layer, id=0100(Receiver
ID)
e1000e 0000:01:00.0: device [8086:10d3] error status/mask=00000040/00002000
e1000e 0000:01:00.0: [ 6] Bad TLP
root@p1010rdb:~#

8, The pcie device(e1000e PCIe NIC) should still work after AER error recovery.
# ping 192.168.1.1 -c 2 -s 64
PING 192.168.1.1 (192.168.1.1): 64 data bytes
72 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=0.272 ms
72 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=0.210 ms
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.210/0.241/0.272/0.031 ms
```

Note:

On some legacy platforms with legacy PCI controller(e.g. some non-DPAA platforms), hardware doesn't support Fatal error type for AER, just support Non-Fatal error.

Generally, DPAA platforms with new PCIe controller can support both Fatal error and Non-Fatal error.

5.8.4 PCI-e Remove and Rescan User Manual

Description

Describes how to remove and rescan a PCI-e device under runtime Linux system.

U-boot Configuration

Use the default configurations.

Kernel Configure Options

Use the default configurations, make sure the configure option is set while doing "make menuconfig" for kernel.

Kernel Configure Tree View Options	Description
Device Drivers ---> [*] Network device support---> [*] Ethernet (1000 Mbit) ---> [*] Intel(R) PRO/1000 PCI-Express Gigabit Ethernet support	This option enables kernel support for Intel PCI-e e1000e network card

Below are the configure identifiers which are used in kernel source code and default configuration files.

Option	Values	Default Value	Description
CONFIG_E1000E	y/n	y	Intel PCI-e e1000e network card driver

Device Tree Binding

Use the default dtb file.

Verification in Linux

Make sure the PCI-e controller which you add the PCI-e e1000e network card to works as RC mode. Use the kernel, dtb and ramdisk rootfs to boot the board.

```
1. Suppose the PCI-e device under /sys/bus/pci/devices/0001\:03\:00.0 is the Intel PCI-e e1000e network card, recognized as eth0. The /sys/bus/pci/devices/0001\:02\:00.0 is the bus of network card. Configure an ip and ping another host which is in the same subnet, make sure the network card works well.
```

```
# ls /sys/bus/pci/devices/0001\:03\:00.0/net
eth0
# ifconfig eth0 10.193.20.100
# ping -I eth0 10.193.20.31
```

```
2. Remove the PCI-e network card from system.
# echo 1 > /sys/bus/pci/devices/0001\:03\:00.0/remove
e1000e 0001:03:00.0 eth0: removed PHY
```

```
3. Check whether the PCI-e network card still exist in system. All should fail.
# ifconfig eth0
# ls /sys/bus/pci/devices/0001\:03\:00.0
```

```
4. Rescan it from the bus.
# echo 1 > /sys/bus/pci/devices/0001\:02\:00.0/rescan
```

```
5. Check whether the device is rescanned and works well.
# ls /sys/bus/pci/devices/0001\:03\:00.0
# ifconfig eth0 10.193.20.100
# ping -I eth0 10.193.20.31
```

```
6. All the commands of step 5 should success.
```

Known Bugs, Limitations, or Technical Issues

The support of PCI-e device remove rescan on powerpc platform is first added in NXP Linux SDK 1.4(kernel version: 3.8.4). If it fail, the PCI-e device will be rescanned, but the driver of the device will fail to loaded.

5.8.5 PCIE EP

Description

SKMM (Secure Key Management Module) is Linux user space implementation for accelerating the encryption/decryption operations on the NXP processors with SEC engine. SKMM consists of two parts of software, the host kernel driver and the application on EP side. The host driver interfaces with OpenSSL or sysfs to pass the encryption/decryption operations to EP side by abstract request and the EP application parses the abstract request to do the encryption/decryption operations, pass the results to host side. The key point here is the private key used is only kept in EP side and is invisible to host side. Also the private key(s) is stored in NOR flash using blob mechanism for protecting the key across system power cycles.

SKMM has two sub use cases:

- - SKMM with PCIe data path. (For now, only SKMM with PCIe data path is supported.)

For SKMM with PCIe data path, the two boards are connected through the PCIe interface, the requests are sent to EP through PCIe interface.

- - SKMM with Ethernet data path.

Requests are sent to the EP through the Ethernet port.

Platforms Supported

- P4080DS
- T4240QDS

Compiling SKMM code

Refer to SDK Documentation provided with this release for installing and using Yocto for Image compilation and building.

How to configure and build Host images for PowerPC

1. Change directory to Yocto, execute commands to configure kernel

```
> source ./poky/fsl-setup-poky -m <board-type> -j 12 -t 12

#> bitbake -c menuconfig linux-qoriq-sdk

To P4080DS and T4240QDS:
Location:
-> Device Drivers
-> DMA Engine support
->[*] NXP Elo and Elo Plus DMA support (enable the option)
[ ] Network: TCP receive copy offload (disable the option)
```

2. Execute Commands to build ulmage with DMA enabled

```
#> bitbake linux-qoriq-sdk
```

3. Execute Command to generate RFS with kernel modules for host and SKMM application for EP

```
#> bitbake fsl-image-core
```

How to configure and build Host images for X86:

NOTE

When using X86 as Host of SKMM, need a Linux distribution to be installed to X86 PC, suggested using Ubuntu 12.04 64bit.

1. Change directory to Yocto, then extract the SKMM Host source code as following:

```
#> source ./poky/fsl-setup-poky -m <board-type> -j 12 -t 12
#> bitbake -c patch skmmhost
```

2. The source code will be in `tmp/work/<board -type>-fsl_networking-linux/skmmhost/git-r0/git/`. Copy the source directory to your X86 Host machine for building.
3. Change the directory to the top of the SKMM Host's source code, execute the command:

```
#> make ARCH=x86_64 KERNEL_DIR=/lib/modules/$(uname -r)/build
```

4. The driver modules will be built in the current directory, named "fsl_crypto.ko" and "rsa_test.ko"

How to configure and build EP kernel:

1. Change the directory to Yocto, then execute commands to configure the kernel:

```
#> source ./poky/fsl-setup-poky -m <board-type> -j 12 -t 12
#> bitbake -c menuconfig linux-qoriq-sdk
```

- For P4080ds: Location:

```
-> Cryptographic API
-> [*] Hardware crypto devices
-> < >   NXP CAAM-Multicore driver backend (disable the option)
-> Device Drivers
-> <*> Userspace I/O drivers
-> <*>   NXP SEC support(disable the option)
```

- For T4240qds: Location:

```
-> Cryptographic API
-> [*] Hardware crypto devices
-> < >   NXP CAAM-Multicore driver backend (disable the option)
-> Device Drivers
-> <*> Userspace I/O drivers
-> <*>   NXP SEC support(disable the option)
-> <*> VFIO Non-Privileged userspace driver framework(enable the option)
-> <*>   VFIO support for Fresscale PCI Endpoint devices(enable the option)
```

2. Execute command to build the kernel image

```
#> bitbake linux-qoriq-sdk
```

3. Execute command to generate RFS with SKMM application for EP

```
#> bitbake fsl-image-core
```

Configure physical connection

EP

- For P4080ds, route the PCIe cable between slot #1 and the Host.
- For T4240qds, route the PCIe cable between slot #5 and the Host.

Host

- For P4080ds, route the PCIe cable between slot #3 and the EP
- For X86, route the PCIe cable between X86 PCIe slot with the x4 to x1 connector, and the EP
- For T4240qds, route the PCIe cable between slot #7 and the EP

How to operate EP

Store all the images for PowerPC machine on the tftp server. Configure the board IP and tftp server IP on the u-boot environment using following commands, the third step is to reserve memory for SKMM, it couldn't be ignored.

For P4080ds as EP

1.

```
#> setenv ipaddr <board ip >
#> setenv serverip <tftp server ip >
#> setenv bootargs "$bootargs usdpaa_mem=256m"
#> save
```

2. Program RCW with `R_PPPNN_0x5/rcw_ep_1500mhz.bin` to flash.

3. Execute following command at u-boot prompt to boot the board

```
#> tftp 1000000 uImage-<bsp>.bin
#> tftp 2000000 fsl-image-core-<bsp>.ext2.gz.uboot
#> tftp c000000 uImage-<bsp>.dtb
#> bootm 1000000 2000000 c000000
```

4. Once the Linux Image boots, enter username=root and password=root to logon.

5. If the SKMM application is being run for the first time, update private key into Nor flash MTD4.

```
root@p4080:flash_eraseall /dev/mtd4
root@p4080:skmm_$(host) /dev/mtd4 update-key ~/.skmm/RSA_priv3
```

NOTE

skmm_\$(host) is the application for different Host. For x86 its name is "skmm_x86_64", for PowerPC its is "skmm_powerpc", please check the application used is correct to Host.

6. Run SKMM application, then EP will wait for request offloaded

```
root@p4080:skmm_$(host) /dev/mtd4
```

For T4240qds as EP

1.

```
#> setenv ipaddr <board ip >
#> setenv serverip <tftp server ip >
#> setenv bootargs "$bootargs usdpaa_mem=256m"
#> save
```

2. Program RCW with "`RR_XXSSPRPH_1_28_6_12/rcw_1_28_6_12_pexep_1666MHz.bin`"

```
#> tftp 1000000 uImage-<bsp>.bin
#> tftp 2000000 fsl-image-core-<bsp>.ext2.gz.uboot
#> tftp c000000 uImage-<bsp>.dtb
```

```
#> fdt addr $fdtaddr;fdt mknod /localbus/nor partition@7000000;
#> fdt set /localbus/nor/partition@7000000 reg <0x07000000 0x00100000>;
#> fdt set /localbus/nor/partition@7000000 label "blob";
#> bootm $loadaddr - $fdtaddr;
```

3. Once the Linux Image boots, enter username=root and password=root to logon.
4. If the SKMM application is being run for the first time, update private key into Nor flash MTDO

```
root@t4240:flash_eraseall /dev/mtd0
root@t4240:skmm_$(host) /dev/mtd0 update-key ~/.skmm/RSA_priv3
```

NOTE

skmm_\$(host) is the application for different Host. For x86 its name is "skmm_x86_64", for PowerPC its is "skmm_powerpc", please check the application used is correct to Host.

5. Run SKMM application, then EP will wait for request offloaded

```
root@t4240:skmm_$(host) /dev/mtd0
```

How to operate Host

1. boot up PowerPC Host:
2. Configure the board IP and tftp server IP on the u-boot environment using following commands:

```
#> setenv ipaddr <board ip >
#> setenv serverip <tftp server ip >
#> save
```

3. Execute following command at u-boot prompt to boot the board

```
#> tftp 1000000 uImage-<bsp>.bin
#> tftp 2000000 fsl-image-core-<bsp>.ext2.gz.uboot
#> tftp c000000 uImage-<bsp>.dtb
#> bootm 1000000 2000000 c000000
```

4. Once the Linux Image boots, enter username=root and password=root to logon.
5. Insmod the module to start the test process

```
Root #>:insmod /lib/modules/$(uname -r)/extra/fsl_crypto.ko dev_config_file=/etc/skmm/
skmm_crypto.cfg
```

6. boot up X86 Host
7. There is no specific setup for X86, change directory to c293_skmm_host copied from Yocto, and make sure the modules has been generated.
8. Insert the module to start the test process
 - For PowerPC Host:

```
Root #>:insmod /lib/modules/$(uname -r)/extra/fsl_crypto.ko dev_config_file=/etc/skmm/
skmm_crypto.cfg
```

- For X86 Host:

```
root #>:insmod fsl_crypto.ko dev_config_file=crypto.cfg
```

How to test

When you complete one of RSA public key test or private key test, you have to reboot both HOST and EP, and reload the fsl_crypto.ko module above, then move on another function test step.

```
For PowerPC as Host
RSA public key:
Root #>:insmod /lib/modules/$(uname -r)/extra/rsa_test.ko op=rsa_pub
RSA private key:
Root #>:insmod /lib/modules/$(uname -r)/extra/rsa_test.ko op=rsa_priv3

For x86 as Host
RSA public key:
root #>:insmod rsa_test.ko op=rsa_pub
RSA private key:
root #>:insmod rsa_test.ko op=rsa_priv3
```

Because PowerPC haven't supported PCIe hotplug yet, removing fsl_crypto.ko module will cause a Host kernel call trace

Function test result

If there was nothing "ERROR" log printed, it means test result was correct, but if the console prints the words like "!!!! Not matching byte got [xx] original [%0x] at index [xx]"(note: xx is a number of 0 to127), it means test failed.

Performance test

```
RSA public key:
Root #>:echo "RSA_PUB_OP_1K" >/sys/fsl_crypto/fsl_crypto_1/test-i/test_name
RSA private key:
Root #>:echo "RSA_PRIV_OP_1K" >/sys/fsl_crypto/fsl_crypto_1/test-i/test_name
```

After console prints the start time and end time, it means the test process is finished, then please use the formula below to calculate the performance ops/s (The number of crypto operations in a second):

```
Ops/s = Host cpu Frequency / ((end time - start time) / 5000)
```

For example, to P4080ds, Cpu Frequency is 1.5GHz(1.5 x 10⁹). 5000 is the number of crypto operations (it is the default setting for performance test), it has been hard code, so it couldn't be modified.

5.9 SATA Controller

5.9.1 External SATA Controller User Manual

Description

The SATA controller is integrated in ULi1575 south bridge on the board.

Source Files

The driver source is maintained in the U-boot source tree.

Source File	Description
drivers/block/ahci.c	U-Boot sata driver support
common/cmd_scsi.c	U-Boot scsi command support

The driver source is maintained in the Linux source tree.

Source File	Description
drivers/ata/sata_fsl.c	NXP SATA driver

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
<pre>Device Drivers ---> <*> Serial ATA (prod) and Parallel ATA (experimental) drivers ---> [*] SATA Port Multiplier support <*> AHCI SATA support <*> Freescale 3.0Gbps SATA support</pre>	Enable NXP 3.0Gbps SATA SoC driver
<pre>Device Drivers ---> SCSI device support ---> <*> SCSI disk support</pre>	Enable SCSI disk support
<pre>File systems ---> <*> Second extended fs support <*> Ext3 journalling file system support Partition Types ---> [*] Advanced partition selection [*] PC BIOS (MSDOS partition tables) support Native Language Support ---> (iso8859-1) Default NLS Option <*> Codepage 437 (United States, Canada) <*> NLS ISO 8859-1 (Latin 1; Western European Languages)</pre>	Enable filesystem and partition table support

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_ATA	y/n	y	Enables libata support
CONFIG_SATA_FSL	y/m/n	y	Enables NXP 3.0Gbps SATA driver
CONFIG_BLK_DEV_SD	y/m/n	y	Enables SCSI disk support
CONFIG_EXT2_FS	y/m/n	y	Enables ext2 filesystem support
CONFIG_EXT3_FS	y/m/n	y	Enables ext3 filesystem support
CONFIG_PARTITION_ADVANCED	y/n	y	Enables partition table of other OS
CONFIG_MSDOS_PARTITION	y/n	y	Enables MSDOS partition table support
CONFIG_NLS_DEFAULT	y/n	y	Default NLS used when mounting file system
CONFIG_NLS_CODEPAGE_437	y/n	y	Add support to codepage 437

Verification in U-Boot

Make sure sata had been inserted on board before power on.

U-Boot log . Take P2020DS board as an example.

```
SCSI: AHCI 0001.0000 32 slots 4 ports 3 Gbps 0xf impl IDE mode
flags: ncq ilck pm led clo pmp pio slum part
scanning bus for devices...

Device 0: (1:0) Vendor: ATA Prod.: Hitachi HDT72101 Rev: ST10

Type: Hard Disk

Capacity: 131071.9 MB = 127.9 GB (268435455 x 512)
```

Scan the contents of the SATA disk in u-boot

```
=> scsi scan

scanning bus for devices...

Device 0: (1:0) Vendor: ATA Prod.: Hitachi HDT72101 Rev: ST10

Type: Hard Disk

Capacity: 131071.9 MB = 127.9 GB (268435455 x 512)

=>
```

Verification in Linux

Scan the contents of the SATA disk.

```
=> ext2ls scsi 0:3 /  
  
<DIR> 4096 .  
  
<DIR> 4096 ..  
  
<DIR> 16384 lost+found  
  
619548672 rootfs.p2020  
  
<DIR> 4096 bin  
  
<DIR> 4096 boot  
  
....  
  
<DIR> 4096 usr  
  
<DIR> 4096 var  
  
=>
```

Set bootargs.

```
=>setenv scsiboot 'scsiboot=setenv bootargs root=/dev/sda3 rw console=ttyS0,115200;ext2load scsi  
0:3 1000000 /boot/uImage.p2020;ext2load scsi 0:3 c00000 /boot/p2020ds.dtb;bootm 1000000 - c00000'  
  
=>setenv bootcmd 'run scsiboot'  
  
=>saveenv  
  
=>reset
```

Here is the booting log.

```
....  
  
SCSI: AHCI 0001.0000 32 slots 4 ports 3 Gbps 0xf impl IDE mode  
  
flags: ncq ilck pm led clo pmp pio slum part  
  
scanning bus for devices...  
  
Device 0: (2:0) Vendor: ATA Prod.: ST3160815AS Rev: 3.AA  
  
Type: Hard Disk  
  
Capacity: 131071.9 MB = 127.9 GB (268435455 x 512)  
  
Net: eTSEC1, eTSEC2, eTSEC3  
  
Hit any key to stop autoboot: 0
```

Linux Kernel Drivers
SATA Controller

```
2681833 bytes read
12288 bytes read
....
EXT3-fs warning: checktime reached, running e2fsck is recommended
EXT3 FS on sda3, internal journal
EXT3-fs: recovery complete.
EXT3-fs: mounted filesystem with ordered data mode.
VFS: Mounted root (ext3 filesystem).
Freeing unused kernel memory: 212k init
Mounting /proc and /sys
Starting the hotplug events dispatcher udevd
Synthesizing initial hotplug events
Setting the hostname to p2020ds
Running depmod
WARNING: Couldn't open directory /lib/modules/2.6.28.6-00524-g3c7d759-dirty: No such file or
directory
FATAL: Could not open /lib/modules/2.6.28.6-00524-g3c7d759-dirty/modules.dep.temp for writing: No
such file or directory
Mounting filesystems
Starting syslogd and klogd
Running sysctl
Setting up networking on loopback device:
Setting up networking on eth0:
ADDRCONF(NETDEV_UP): eth0: link is not ready
Adding static route for default gateway to 192.168.1.1:
Setting nameserver to 192.168.1.1 in /etc/resolv.conf:
Setting up networking oADDRCONF(NETDEV_UP): eth1: link is not ready
n eth1:
Adding static route for default gateway to 192.168.1.1:
Setting nameserver to 192.168.1.1 in /etc/resolv.conf:
Starting inetd:
```

```

Starting the port mapper:

Setting time from ntp server:

/bin/ntpclient: option requires an argument -- h

Usage: /bin/ntpclient [-c count] [-d] [-g goodness] -h hostname [-i interval]

[-l] [-p port] [-r] [-s]

Starting the watchdog daemon:

rebuilding rpm database

PHY: mdio@24520:00 - Link is Up - 1000/Full

ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready

Welcome to Freescale Semiconductor Embedded Linux Environment

!!!! WARNING !!!!!!!

The default password for the root account is: root

please change this password using the 'passwd' command

and then edit this message (/etc/issue) to remove this message

p2020ds login:

```

Mount sata after bootup.

```

~ #

~ # fdisk -l

Disk /dev/sda: 160.0 GB, 160041885696 bytes

255 heads, 63 sectors/track, 19457 cylinders

Units = cylinders of 16065 * 512 = 8225280 bytes

Device Boot Start End Blocks Id System

/dev/sda1 1 13 104391 83 Linux

/dev/sda2 14 379 2939895 83 Linux

/dev/sda3 380 866 3911827+ 83 Linux

~ # mount /dev/sda3 /mnt/src

~ # ls /mnt/src

bin include mnt sbin

boot lib opt sys

dev linuxrc proc tmp

```

```

etc lost+found root usr

home man rootfs.ext2.gz.uboot var

~ #

~ # umount /dev/sda3

~ # ls /mnt/src

~ #

~ #

```

5.9.2 NXP Native SATA Driver User Manual

Description

The driver supports NXP native SATA controller. There are two types of SATA controller. One is PowerPC-based and the other is ARM-based. There is slight difference between them. This manual will take P5020DS board as example for description.

Module Loading

SATA driver supports either kernel built-in or module.

Kernel Configure Tree View Options

1) For PowerPC-based Socs, like P5020, P5040 etc.

Kernel Configure Tree View Options	Description
<pre> Device Drivers---> <*> Serial ATA and Parallel ATA drivers ---> --- Serial ATA and Parallel ATA drivers <*> Freescale 3.0Gbps SATA support </pre>	Enables SATA controller support on PowerPC-based SoCs

2) For ARM-based Socs, like LS1021, LS2085 etc.

Kernel Configure Tree View Options	Description
<pre> Device Drivers---> <*> Serial ATA and Parallel ATA drivers ---> --- Serial ATA and Parallel ATA drivers <*> AHCI SATA support <*> Freescale QorIQ AHCI SATA support </pre>	Enables SATA controller support on ARM-based SoCs

Compile-time Configuration Options

1) For PowerPC-based Socs, like P5020, P5040 etc.

Option	Values	Default Value	Description
CONFIG_SATA_FSL=y	y/m/n	y	Enables SATA controller

2) For ARM-based Socs, like LS1021, LS2085 etc.

Option	Values	Default Value	Description
CONFIG_SATA_AHCI=y	y/m/n	y	Enables SATA controller
CONFIG_SATA_AHCI_QORIQ=y	y/m/n	y	Enables SATA controller

Source Files

The driver source is maintained in the Linux kernel source tree.

1) For PowerPC-based Socs, like P5020, P5040 etc.

Source File	Description
drivers/ata/sata-fsl.c	SATA controller driver

2) For ARM-based Socs, like LS1021, LS2085 etc.

Source File	Description
drivers/ata/ahci_qoriq.c	Platform AHCI SATA support

P5020DS Test Procedure

```

Please follow the following steps to use USB in Simics
(1) Boot up the kernel
...
fsl-sata ffe18000.sata: Sata FSL Platform/CSB Driver init
scsi0 : sata_fsl
ata1: SATA max UDMA/133 irq 74
fsl-sata ffe19000.sata: Sata FSL Platform/CSB Driver init
scsi1 : sata_fsl
ata2: SATA max UDMA/133 irq 41
...
(2) The disk will be found by kernel.
...
ata1: Signature Update detected @ 504 msecs
ata2: No Device OR PHYRDY change,Hstatus = 0xa0000000
ata2: SATA link down (SStatus 0 SControl 300)
ata1: SATA link up 1.5 Gbps (SStatus 113 SControl 300)
ata1.00: ATA-8: WDC WD1600AAJS-22WAA0, 58.01D58, max UDMA/133
ata1.00: 312581808 sectors, multi 0: LBA48 NCQ (depth 16/32)
ata1.00: configured for UDMA/133
scsi 0:0:0:0: Direct-Access      ATA          WDC WD1600AAJS-2 58.0 PQ: 0 ANSI: 5
sd 0:0:0:0: [sda] 312581808 512-byte logical blocks: (160 GB/149 GiB)
sd 0:0:0:0: Attached scsi generic sg0 type 0
sd 0:0:0:0: [sda] Write Protect is off

```

Linux Kernel Drivers
SATA Controller

```
sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't support DPO
or FUA
sda: sda1 sda2 sda3 sda4 < sda5 sda6 >
sd 0:0:0:0: [sda] Attached SCSI disk
```

(3)play with the disk according to the following log.

```
[root@p5020 root]# fdisk -l /dev/sda
Disk /dev/sda: 160.0 GB, 160041885696 bytes
255 heads, 63 sectors/track, 19457 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
```

Device	Boot	Start	End	Blocks	Id	System
/dev/sda1		1	237	1903671	83	Linux
/dev/sda2		238	480	1951897+	82	Linux swap
/dev/sda3		481	9852	75280590	83	Linux
/dev/sda4		9853	19457	77152162+	f	Win95 Ext'd (LBA)
/dev/sda5		9853	14655	38580066	83	Linux
/dev/sda6		14656	19457	38572033+	83	Linux

```
[root@p5020 root]#
[root@p5020 root]# mke2fs /dev/sda1
mke2fs 1.41.4 (27-Jan-2009)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
65280 inodes, 261048 blocks
13052 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=268435456
8 block groups
32768 blocks per group, 32768 fragments per group
8160 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376
```

```
Writing inode tables: done
Writing superblocks and filesystem accounting information: done
```

This filesystem will be automatically checked every 22 mounts or 180 days, whichever comes first. Use tune2fs -c or -i to override.

```
[root@p5020 root]#
[root@p5020 root]# mkdir sata
[root@p5020 root]# mount /dev/sda1 sata
[root@p5020 root]# ls sata/
lost+found
[root@p5020 root]# cp /bin/busybox sata/
[root@p5020 root]# umount sata/
[root@p5020 root]# mount /dev/sda1 sata/
[root@p5020 root]# ls sata/
busybox    lost+found
[root@p5020 root]# umount sata/
[root@p5020 root]# mount /dev/sda3 /mnt
[root@p5020 root]# df
```

Filesystem	1K-blocks	Used	Available	Use%	Mounted on
rootfs	852019676	794801552	13937948	99%	/
/dev/root	852019676	794801552	13937948	99%	/
tmpfs	1036480	52	1036428	1%	/dev
shm	1036480	0	1036480	0%	/dev/shm

```
/dev/sda3          74098076   4033092   66300956   6% /mnt
```

Known Bugs, Limitations, or Technical Issues

- 1) For PowerPC-based Socs, like P5020, P5040 etc.
- The best value of RX_WATER_MARK for good performance is 0x16, but it is set to 0x10 in driver since some disks cannot work with higher value. The value can be changed at run time like below:

```
echo 22 > /sys/devices/ffe000000.soc/ffe220000.sata/rx_watermark

22 is 0x16, ffe220000 is the register base of first SATA controller (ffe221000 is the second
SATA controller),
after changing it, use below instruction to check:

cat /sys/devices/ffe000000.soc/ffe220000.sata/rx_watermark
```

- The SATA controller has only 32-bit DMA access ability, it cannot access memory space above 4G if there is more than 4G memory in system, then kernel will enable SWIOTLB (which is also know as bounce buffer) to do an extra copy, this will cause performance degradation. So if there is more than 4G memory and need a good performance for SATA, set 'mem=4G' in U-boot bootargs, this will limit the system to use only 4G memory.
 - P5040DS board has issue to support Gen1(1.5Gbps) hard drive, use Gen2(3Gbps) hard drive on P5040DS.
- 2) For ARM-based Socs, like LS1021, LS2085 etc.
 - CDROM is not supported due to the silicon limitation

5.10 Serial Peripheral Interface

5.10.1 QuadSPI Driver for TWR-LS1021A User Manual

5.10.1.1 QuadSPI Driver User Manual

U-Boot Configuration

Make sure your boot mode support QSPI.

Use QSPI boot mode to boot an board, please check the board user manual and boot from QSPI. (or some other boot mode decide by your board.)

Kernel Configure Tree View Options

```
Device Drivers --->
  Memory Technology Device (MTD) support
  RAM/ROM/Flash chip drivers --->
    < > Detect flash chips by Common Flash Interface (CFI) probe
    < > Detect non-CFI AMD/JEDEC-compatible flash chips
    < > Support for RAM chips in bus mapping
    < > Support for ROM chips in bus mapping
    < > Support for absent chips in bus mapping
  Self-contained MTD device drivers --->
    <*> Support most SPI Flash chips (AT26DF, M25P, W25X, ...)
  < > NAND Device Support ----
```

Linux Kernel Drivers
Serial Peripheral Interface

```
[*] the framework for SPI-NOR support  
<*> Freescale Quad SPI controller
```

```
Device Drivers --->  
 [ ] Memory Controller drivers ----
```

Compile-time Configuration Options

Config	Values	Default Value	Description
CONFIG_SPI_FSL_QUADSPI	y/n	y	Enable QSPI module
CONFIG_MTD_SPI_NOR_BASE	y/n	y	Enables the framework for SPI-NOR

Verification in U-Boot

```
=> sf probe 0:0  
SF: Detected N25Q128A13 with page size 256 Bytes, erase size 4 KiB, total 16 MiB  
=> sf erase 0 100000  
SF: 1048576 bytes @ 0x0 Erased: OK  
=> sf write 82000000 0 1000  
SF: 4096 bytes @ 0x0 Written: OK  
=> sf read 81100000 0 1000  
SF: 4096 bytes @ 0x0 Read: OK  
=> cm.b 81100000 82000000 1000  
Total of 4096 byte(s) were the same
```

Verification in Linux:

```
The booting log  
  
.....  
fsl-quadspi 1550000.quadspi: n25q128a13 (16384 Kbytes)  
fsl-quadspi 1550000.quadspi: QuadSPI SPI NOR flash driver  
.....  
  
Erase the QSPI flash  
  
~ # mtd_debug erase /dev/mtd0 0x1100000 1048576  
Erased 1048576 bytes from address 0x00000000 in flash  
  
Write the QSPI flash  
  
~ # dd if=/bin/tempfile.debianutils of=tp bs=4096 count=1  
~ # mtd_debug write /dev/mtd0 0 4096 tp  
Copied 4096 bytes from tp to address 0x00000000 in flash  
  
Read the QSPI flash
```

```

~ # mtd_debug read /dev/mtd0 0 4096 dump_file

Copied 4096 bytes from address 0x00000000 in flash to dump_file

Check Read and Write

Use compare tools(yacto has tools named diff).
~ # diff tp dump_file
~ #
If diff command has no print log, the QSPI verification is passed.
  
```

5.11 Universal Serial Bus Interfaces

5.11.1 USB 2.0 Host Driver

5.11.1.1 USB 2.0 Host Driver User Manual

Description

The driver supports USB controller in host mode

Module Loading

The USB Host driver in linux supports either kernel built-in or module driver. U-boot USB driver is always built-in

U-Boot Compile Time Configuration Options

U-Boot Configure Options	Description
<pre> #define CONFIG_HAS_FSL_DR_USB #ifdef CONFIG_HAS_FSL_DR_USB #define CONFIG_USB_EHCI #ifdef CONFIG_USB_EHCI #define CONFIG_CMD_USB #define CONFIG_USB_STORAGE #define CONFIG_USB_EHCI_FSL #define CONFIG_EHCI_HCD_INIT_AFTER_RESET #define CONFIG_CMD_EXT2 </pre>	<p>Enables USB host Dual Role controller support. Defined inside platform config file: include/configs/<platform.h>.</p>
<pre> CONFIG_SYS_FSL_USB_INTERNAL_UTMI_PHY </pre>	<p>Enable internal UTMI Phy support. Required only for SoCs having internal UTMI PHY. Defined inside file: arch/powerpc/include/asm/config_mpc85xx.h</p>
<pre> CONFIG_USB_MAX_CONTROLLER_COUNT </pre>	<p>Tell maximum no. of USB controllers in this SoC. Defined inside file: arch/powerpc/include/asm/config_mpc85xx.h</p>

U-Boot Source Files

The driver source is maintained in the U-boot source in following files

Source File	Description
drivers/usb/host/ehci-fsl.c	EHCI FSL USB host controller driver
common/cmd_usb.c	Common usb command file
drivers/usb/host/ehci-hcd.c	EHCI USB host controller driver

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
<pre>Device Drivers---> USB support ---> [*] Support for Host-side USB</pre>	Enables USB host controller support
<pre>Device Drivers---> USB support ---> <*> EHCI HCD (USB 2.0) support -- Root Hub Transaction Translators [] Improved Transaction Translator scheduling (EXPERIMENTAL) [*] Support for Freescale on-chip EHCI USB controller [*] EHCI support for PPC USB controller on OF platform bus <*> OHCI HCD support [*] OHCI support for PPC USB controller on OF platform bus [*] Support big endian HC [*] Support little endian HC [*] OHCI support for PCI-bus USB controllers</pre>	Enables EHCI Host Controller Driver and transaction translator.

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_USB	y/m/n	y	Enables USB host controller
CONFIG_USB_EHCI_HCD	y/m/n	y	Enables EHCI HCD
CONFIG_USB_EHCI_ROOT_HUB_TT	y/n	y	Enables EHCI to support USB1.1 device
CONFIG_USB_OHCI_HCD	y/m/n	y	Enables OHCI HCD

Kernel Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/usb/host/ehci-fsl.c	EHCI USB host controller driver
arch/powerpc/sysdev/fsl_soc.c	Hook between OF tree and platform device
drivers/usb/host/ohci_hcd.c	OHCI USB host controller driver

Device Tree Binding

```
usb@22000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl-usb2-<controller-type>-v<controller version>",
                "fsl-usb2-<controller-type>";
    reg = <0x22000 0x1000>;
    interrupt-parent = <&mpic>;
    interrupts = <28 0x2>;
    phy_type = "ulpi";                /* ulpi/utmi/utmi_dual */
    dr_mode = "host"                  /* host, peripheral */
};
```

NOTE

controller-type: dr(dual-role), mph(multi-port-host) controller-version: 1.6, 2.2, or earlier default mode is always host

Verification in U-Boot

U-boot environment to specify usb phy and usb mode type

```
=> setenv hwconfig 'usb<controller-no>:dr_mode=<mode>,phy_type=<phy_type>;<next usb controller>'
```

For example:

```
For socs having single usb controller and ULPI phy  
=> setenv hwconfig 'usb1:dr_mode=host,phy_type=ulpi'
```

```
For socs having single usb controller and UTMI phy  
=> setenv hwconfig 'usb1:dr_mode=host,phy_type=utmi'
```

```
For socs having two usb controllers and ULPI phys only  
=> setenv hwconfig 'usb1:dr_mode=host,phy_type=ulpi;usb2:dr_mode=host,phy_type=ulpi'
```

Then use usb start to start the usb device

```
=> usb start
```

```
(Re)start USB...
```

```
USB: Register 10011 NbrPorts 1
```

```
USB EHCI 1.00
```

```
scanning bus for devices... 2 USB Device(s) found
```

```
scanning bus for storage devices... 1 Storage Device(s) found
```

```
=> usb dev
```

```
USB device 0: Vendor: SanDisk Rev: 8.02 Prod: Cruzer Colors+
```

```
Type: Removable Hard Disk
```

```
Capacity: 7663.9 MB = 7.4 GB (15695871 x 512)
```

```
=> usb tree
```

```
Device Tree:
```

```
 1 Hub (480 Mb/s, 0mA)  
  | u-boot EHCI Host Controller  
  |  
 +-2 Mass Storage (480 Mb/s, 500mA)  
     JetFlash Mass Storage Device 63Z0A5608TZFZ0AC
```

```
=> usb info
```

```
1: Hub, USB Revision 2.0  
- u-boot EHCI Host Controller  
- Class: Hub  
- PacketSize: 64 Configurations: 1  
- Vendor: 0x0000 Product 0x0000 Version 1.0  
Configuration: 1  
- Interfaces: 1 Self Powered 0mA  
Interface: 0
```

```

- Alternate Setting 0, Endpoints: 1
- Class Hub
- Endpoint 1 In Interrupt MaxPacket 2048 Interval 255ms

2: Mass Storage, USB Revision 2.0
- JetFlash Mass Storage Device 63ZOA5608TZFZ0AC
- Class: (from Interface) Mass Storage
- PacketSize: 64 Configurations: 1
- Vendor: 0x8564 Product 0x1000 Version 17.0
Configuration: 1
- Interfaces: 1 Bus Powered 500mA
Interface: 0
- Alternate Setting 0, Endpoints: 2
- Class Mass Storage, Transp. SCSI, Bulk only
- Endpoint 1 In Bulk MaxPacket 512
- Endpoint 2 Out Bulk MaxPacket 512

=> md 2000000
02000000: 02992004 02060002 08462cc0 84990329 .. .....F,....)
02000010: 00c48e24 82181008 06501810 01a80004 ...$.....P.....
02000020: 083d3881 08080270 40a00000 b012a502 .=8....p@.....
02000030: d4000088 28840b45 80028200 40244400 ....(.E....@$D.
02000040: 022b1004 04842482 20610b81 0494d020 .+....$. a.....
02000050: 8012b628 08200100 010c6300 0411b880 ...(. ....c.....
02000060: 42400200 8004a4c8 29802818 904000c0 B@.....).(..@..
02000070: 08210200 2040a8c0 448aae00 a0000000 .!.. @..D.....
02000080: 2800c800 04b62080 60199885 02a62324 (. .... .`.....#$
02000090: 04870a08 a0008000 18020003 0281a232 .....2
020000a0: 50414020 4000850b 02044c00 10013018 PA@ @.....L...0.
020000b0: 00208810 000c2280 081805a8 88800010 . ....".....
020000c0: 000c020a 0012b024 01282c02 00808181 .....$. (,.....
020000d0: 00010824 0160b602 81621008 00828082 ..$. `...b.....
020000e0: 38d0f028 42010e03 1d242290 02000120 8..(B....$"....
020000f0: 6c217230 00920800 20200d40 41c08011 l!r0.... .@A...

=> mw 2000000 ffffaaaa 100
=> md 2000000
02000000: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....
02000010: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....
02000020: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....
02000030: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....
02000040: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....
02000050: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....
02000060: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....
02000070: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....
02000080: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....
02000090: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....
020000a0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....
020000b0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....
020000c0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....
020000d0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....
020000e0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....
020000f0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa .....

=> usb write 2000000 0 100

USB write: device 0 block # 0, count 256 ... 256 blocks write: OK
=> md 1000000
01000000: fb3eae6 2feeffbf dbf7775d ff5bebf7 .... /.....w).[..
01000010: abefefaf 7dbb3e3b bfffb5bb bfb86a7f .... }.>;.....j.
01000020: eff7b68f deaadfff eebf7bff bd7fed1f ..... {.....
01000030: ffef7deb e7bfbbeff dfeff7df 7f3ffcba ..}.....?..
    
```

Linux Kernel Drivers

Universal Serial Bus Interfaces

```
01000040: ab3bfbfe dfdee69b ffe18fd5 ff3e777f .;. . . . .>w.
01000050: da7effef bfabff7f f58ef768 9ffffeff .~. . . . .h...
01000060: cfebf8b0 f1b7dfef e9eefbfe f37bbadb . . . . .{..
01000070: b7f34ea2 da7efbff dfddf7ff f7effde3 .N.~. . . . .
01000080: fffbfbff fe56ff5d 6ffd7ffd ff87efdf . . . .V.]o. . . . .
01000090: b6bfafac ddebfbfb ffacebfd f87bff9f . . . . .{..
010000a0: ffebffff ff7e7ff9 aefefd7f 5f7f5ebf . . . .~. . . ._.^
010000b0: 6fe87e7b fabfdbcf d3faefad 6fbb5e7a o.~{. . . . .o.^z
010000c0: f6af86de ffdb7bbf ff5ff6ba bfa4bfdf . . . .{.._ . . . .
010000d0: ffbfa87f ffe67fcd efefb9a 9b7e6a6f . . . . .~jo
010000e0: fffc776f efbfeebb ffaceab1 5cfbffff . . . . .\...
010000f0: edebffde e29fefff deaeafdb f97bdf5f . . . . .{..
=> usb read 1000000 0 100
```

```
USB read: device 0 block # 0, count 256 ... 256 blocks read: OK
=> md 1000000
```

```
01000000: ffffffff ffffffff ffffffff ffffffff . . . . .
01000010: ffffffff ffffffff ffffffff ffffffff . . . . .
01000020: ffffffff ffffffff ffffffff ffffffff . . . . .
01000030: ffffffff ffffffff ffffffff ffffffff . . . . .
01000040: ffffffff ffffffff ffffffff ffffffff . . . . .
01000050: ffffffff ffffffff ffffffff ffffffff . . . . .
01000060: ffffffff ffffffff ffffffff ffffffff . . . . .
01000070: ffffffff ffffffff ffffffff ffffffff . . . . .
01000080: ffffffff ffffffff ffffffff ffffffff . . . . .
01000090: ffffffff ffffffff ffffffff ffffffff . . . . .
010000a0: ffffffff ffffffff ffffffff ffffffff . . . . .
010000b0: ffffffff ffffffff ffffffff ffffffff . . . . .
010000c0: ffffffff ffffffff ffffffff ffffffff . . . . .
010000d0: ffffffff ffffffff ffffffff ffffffff . . . . .
010000e0: ffffffff ffffffff ffffffff ffffffff . . . . .
010000f0: ffffffff ffffffff ffffffff ffffffff . . . . .
=>
```

Verification in Linux

· Kernel configuration for USB memory stick support

```
* Device Drivers---> SCSI Device Support---> SCSI Device Support ---> <*> SCSI disk support

* Device Drivers---> SCSI Device Support---> SCSI Device Support ---> <*> SCSI generic support

* Device Drivers---> USB Support---> [*] USB Mass Storage Support

(The user can enable it either in kernel mode (set option as True) or as module (set option as
Module)).

* File Systems---> DOS/FAT/NT Filesystems---> <*> MSDOS fs support

* File Systems---> DOS/FAT/NT Filesystems---> <*> VFAT(Window-95)fs support

* File Systems---> DOS/FAT/NT Filesystems---> VFAT(Window-95)fs support---> Default codepage for
FAT - 437

* File Systems---> DOS/FAT/NT Filesystems---> VFAT(Window-95)fs support---> Default iocharset
for FAT - "iso8859-1"

* File Systems---> Partition Types---> [*] Advanced partition selection
```

```
* File Systems---> Partition Types---> [*] PC BIOS (MSDOS partition tables) support

* File Systems---> Native Language Support---> Base native language support---> (iso8859-1)
Default NLS Option

* File Systems---> Native Language Support---> Base native language support---> <*> Codepage 437
(United States, Canada)

* File Systems---> Native Language Support---> Base native language support---> <*> NLS ISO
8859-1 (Latin 1; Western European Languages)
```

· plug in memory stick

```
~ # usb 1-1: new high speed USB device using fsl-ehci and address 2

usb 1-1: configuration #1 chosen from 1 choice

scsi6 : SCSI emulation for USB Mass Storage devices

scsi 6:0:0:0: Direct-Access SanDisk Cruzer 7.01 PQ: 0 ANSI: 0 CCS

sd 6:0:0:0: [sda] 3907583 512-byte hardware sectors: (2.00 GB/1.86 GiB)

sd 6:0:0:0: [sda] Write Protect is off

sd 6:0:0:0: [sda] Assuming drive cache: write through

sd 6:0:0:0: [sda] 3907583 512-byte hardware sectors: (2.00 GB/1.86 GiB)

sd 6:0:0:0: [sda] Write Protect is off

sd 6:0:0:0: [sda] Assuming drive cache: write through

sda: sda1

sd 6:0:0:0: [sda] Attached SCSI removable disk

sd 6:0:0:0: Attached scsi generic sg0 type 0

scsi 6:0:0:1: CD-ROM SanDisk Cruzer 7.01 PQ: 0 ANSI: 0

sr0: scsi3-mmc drive: 48x/48x tray

Uniform CD-ROM driver Revision: 3.20

sr 6:0:0:1: Attached scsi generic sg1 type 5

~ # fdisk -l

Disk /dev/sda: 2000 MB, 2000682496 bytes

64 heads, 63 sectors/track, 969 cylinders

Units = cylinders of 4032 * 512 = 2064384 bytes
```

Linux Kernel Drivers
Universal Serial Bus Interfaces

```
Device Boot Start End Blocks Id System
/dev/sda1 1 969 1953439+ b Win95 FAT32

~ # mount -t vfat /dev/sda1 /mnt/cdrom/

~ # cd /mnt/cdrom/

/mnt/cdrom # ls

/mnt/cdrom # cp /usr/sbin/wd_keepalive .

/mnt/cdrom # ls

wd_keepalive

/mnt/cdrom # cd ..

/mnt # umount /mnt/cdrom/

=====
To create ext2 file-system on USB flash drive, follow below steps:
# fdisk /dev/sda
Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel
e1
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that the previous content
won't be recoverable.

Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-1011, default 1): Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-1011, default 1011): Using default
value 1011

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table
sd 2:0:0:0: [sda] Test WP failed, assume Write Enabled
sd 2:0:0:0: [sda] Assuming drive cache: write through
sda: sda1
[root@p5020 root]# mke2fs /dev/sda1
mke2fs 1.41.4
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
65536 inodes, 262094 blocks
13104 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=268435456
8 block groups
32768 blocks per group, 32768 fragments per group
```

```
8192 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 38 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
[root@p5020 /root]# mount /dev/sda1 /mnt
```

· Kernel configuration for USB Human Input Devices support

```
* Device Drivers--->

  [*] HID Devices --->

    *- Generic HID support

      <*> USB Human Interface Device (full HID) support

    Input device support --->

      *- Generic input layer (needed for keyboard, mouse, ...)

        <*> Mouse interface

          [*] Provide legacy /dev/psaux device

            (1024) Horizontal screen resolution

            (768) Vertical screen resolution

          [*] Keyboards --->

            <*> AT keyboard

          [*] Mice --->

            <*> PS/2 mouse

              [*] ALPS PS/2 mouse protocol extension

              [*] Logitech PS/2++ mouse protocol extension

              [*] Synaptics PS/2 mouse protocol extension

              [*] IBM Trackpoint PS/2 mouse protocol extension
```

· plug in USB keyboard

```
~ # usb 1-1: new full speed USB device using fsl-ehci and address 3

usb 1-1: configuration #1 chosen from 1 choice

hub 1-1:1.0: USB hub found
```

Linux Kernel Drivers
 Universal Serial Bus Interfaces

```

hub 1-1:1.0: 3 ports detected

usb 1-1.1: new full speed USB device using fsl-ehci and address 4

usb 1-1.1: configuration #1 chosen from 1 choice

input: Dell Dell USB Keyboard Hub as /class/input/input1

generic-usb 0003:413C:2002.0002: input: USB HID v1.10 Keyboard [Dell Dell USB Ke yboard Hub] on
usb-fsl-ehci.0-1.1/input0

input: Dell Dell USB Keyboard Hub as /class/input/input2

generic-usb 0003:413C:2002.0003: input: USB HID v1.10 Device [Dell Dell USB Keyb
  
```

· plug in USB mouse

```

~ # usb 1-1: new low speed USB device using fsl-ehci and address 2

usb 1-1: configuration #1 chosen from 1 choice

input: HID 413c:3010 as /class/input/input0

generic-usb 0003:413C:3010.0001: input: USB HID v1.00 Mouse [HID 413c:3010] on u
  
```

Power Management

Following Pwr. Mgmt. features are supported:

- 1) Sleep
- 2) Deep Sleep

Pwr. Mgmt. Kernel Configuration Option(s)

Kernel Configure Tree View Options	Description
<pre> Kernel options---> [*] Suspend to RAM and standby [*] Hibernation (aka 'suspend to disk') () Default resume partition (NEW) </pre>	<p>Enable Power Management feature</p>
<pre> Platform support --> CPU Frequency scaling --> [*] CPU Frequency scaling <*> CPU frequency translation statistics Default CPUFreq governor (userspace) --> *- 'userspace' governor for userspace frequency scaling CPU Frequency drivers --> [*] Support for Freescale MPC85xx CPU freq </pre>	<p>Enable the CPU frequency driver</p>

Verification in Linux

Sleep Capability

A system can be put into Suspend state, and can also be Resumed (woken-up) by USB. For this the following needs to be done:

1. Enable USB remote wake-up capability before putting the system into Suspend state

```
~ # echo enabled >/sys/bus/usb/devices/usb1/power/wakeup
```

2. Insert/Remove a USB flash drive into USB port after the system is put into SUSPEND state. This will bring the system out of the SUSPEND state

```
# echo standby > /sys/power/state
PM: Syncing filesystems ... done.
Freezing user space processes ... (elapsed 0.01 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.01 seconds) done.
Suspending console(s) (use no_console_suspend to debug)
sd 0:0:0:0: [sda] Synchronizing SCSI cache
sd 0:0:0:0: [sda] Stopping disk
PM: suspend of devices complete after 519.108 msecs
PM: late suspend of devices complete after 0.489 msecs
PM: noirq suspend of devices complete after 0.555 msecs
Disabling non-boot CPUs ...

USB Flash drive inserted --->

Enabling non-boot CPUs ...
CPU1 is up
PM: noirq resume of devices complete after 0.513 msecs
PM: early resume of devices complete after 0.349 msecs
fsl-lbc ffe05000.localbus: Chip select error: LTESR 0x00080000
/pcie@ffe09000: PCICSRBAR @ 0xffff0000
/pcie@ffe0a000: PCICSRBAR @ 0x0
/pcie@ffe0a000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the memory map
could reduce unnecessary bounce buffering.
/pcie@ffe0a000: DMA window size is 0x0
/pcie@ffe0b000: PCICSRBAR @ 0x0
/pcie@ffe0b000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the memory map
could reduce unnecessary bounce buffering.
/pcie@ffe0b000: DMA window size is 0x0
pci 0000:00:00.0: enabling device (0106 -> 0107)
pci 0001:02:00.0: enabling device (0106 -> 0107)
pci 0002:04:00.0: enabling device (0106 -> 0107)
ata2: No Device OR PHYRDY change,Hstatus = 0xa0000000
ata2: SATA link down (SStatus 0 SControl 300)
ata1: Signature Update detected @ 504 msecs
ata1: SATA link up 3.0 Gbps (SStatus 123 SControl 300)
ata1.00: configured for UDMA/133
sd 0:0:0:0: [sda] Starting disk
PM: resume of devices complete after 5419.653 msecs
Restarting tasks ... done.
root@p1022ds:~# usb 1-1: new high-speed USB device number 2 using fsl-ehci
scsi2 : usb-storage 1-1:1.0
scsi 2:0:0:0: Direct-Access      SRT          USB          1100 PQ: 0 ANSI: 4
sd 2:0:0:0: Attached scsi generic sgl type 0
sd 2:0:0:0: [sdb] 15568896 512-byte logical blocks: (7.97 GB/7.42 GiB)
sd 2:0:0:0: [sdb] Write Protect is off
```

```
sd 2:0:0:0: [sdb] Mode Sense: 43 00 00 00
sd 2:0:0:0: [sdb] No Caching mode page present
sd 2:0:0:0: [sdb] Assuming drive cache: write through
sd 2:0:0:0: [sdb] No Caching mode page present
sd 2:0:0:0: [sdb] Assuming drive cache: write through
sdb: sdb1
sd 2:0:0:0: [sdb] No Caching mode page present
sd 2:0:0:0: [sdb] Assuming drive cache: write through
sd 2:0:0:0: [sdb] Attached SCSI removable disk
FAT-fs (sdb): error, fat_get_cluster: invalid cluster chain (i_pos 0)
FAT-fs (sdb): Filesystem has been set read-only
```

Deep Sleep Capability

USB working across Deep sleep using Timer Interrupt

System is put into deep sleep using the following command :

```
~# echo 30 > /sys/devices/system/mpic/timer_wakeup;echo mem > /sys/power/state
PM: Syncing filesystems ... done.
mmc0: card e624 removed
Freezing user space processes ... (elapsed 0.001 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
Suspending console(s) (use no_console_suspend to debug)
sd 0:0:0:0: [sda] Synchronizing SCSI cache
sd 0:0:0:0: [sda] Stopping disk
PM: suspend of devices complete after 316.061 msecs
PM: late suspend of devices complete after 0.217 msecs
PM: noirq suspend of devices complete after 31.099 msecs
Disabling non-boot CPUs ...
/pcie@ffe240000: PCICSRBAR @ 0xff000000
/pcie@ffe240000: Setup 64-bit PCI DMA window
/pcie@ffe240000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the memory map
could reduce unnecessary bounce buffering.
/pcie@ffe240000: DMA window size is 0xe0000000
/pcie@ffe250000: PCICSRBAR @ 0xff000000
/pcie@ffe250000: Setup 64-bit PCI DMA window
/pcie@ffe250000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the memory map
could reduce unnecessary bounce buffering.
/pcie@ffe250000: DMA window size is 0xe0000000
/pcie@ffe260000: PCICSRBAR @ 0x0
/pcie@ffe260000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the memory map
could reduce unnecessary bounce buffering.
/pcie@ffe260000: DMA window size is 0x0
/pcie@ffe270000: PCICSRBAR @ 0xff000000
/pcie@ffe270000: Setup 64-bit PCI DMA window
/pcie@ffe270000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the memory map
could reduce unnecessary bounce buffering.
/pcie@ffe270000: DMA window size is 0xe0000000
```

After 30 seconds, system comes out of deep sleep and usb storage device is successfully detected

```
Enabling non-boot CPUs ...
CPU1 is up
CPU2 is up
CPU3 is up
PM: noirq resume of devices complete after 63.844 msecs
PM: early resume of devices complete after 0.166 msecs
caam ffe300000.crypto: Instantiated RNG4 SH0
caam ffe300000.crypto: Instantiated RNG4 SH1
```

```
ata2: No Device OR PHYRDY change,Hstatus = 0x80000000
ata2: SATA link down (SStatus 10 SControl 300)
ata1: Signature Update detected @ 504 msecs
ata1: SATA link up 3.0 Gbps (SStatus 123 SControl 300)
ata1.00: configured for UDMA/133
sd 0:0:0:0: [sda] Starting disk
PM: resume of devices complete after 4461.429 msecs
Restarting tasks ... done.
usb 1-1: USB disconnect, device number 3
root@t1040rdb:~# EXT2-fs (sdb1): previous I/O error to superblock detected

mmc0: new high speed SDHC card at address e624
mmcblk0: mmc0:e624 SU08G 7.40 GiB
  mmcblk0: p1
usb 1-1: new high-speed USB device number 4 using fsl-ehci
usb-storage 1-1:1.0: USB Mass Storage device detected
scsi4 : usb-storage 1-1:1.0
scsi 4:0:0:0: Direct-Access      JetFlash Transcend 4GB      8.07 PQ: 0 ANSI: 4
sd 4:0:0:0: Attached scsi generic sgl type 0
sd 4:0:0:0: [sdb] 7843200 512-byte logical blocks: (4.01 GB/3.73 GiB)
sd 4:0:0:0: [sdb] Write Protect is off
sd 4:0:0:0: [sdb] Write cache: disabled, read cache: enabled, doesn't support DPO or FUA
  sdb: sdb1
sd 4:0:0:0: [sdb] Attached SCSI removable disk
```

Deep Sleep using USB wake-up interrupt

This feature is not yet supported.

Known Bugs, Limitations, or Technical Issues

1. Across system Deep Sleep, if a device is already mounted, it may get umounted automatically. Hence, to use it again, the user needs to re-mount the device
2. USB remote wake-up during system Deep-Sleep is not yet supported
3. On some platforms where USB2 controller is muxed with some other IP, USB2 is disabled in default platform configurations inside both U-boot and Linux. For more details, please refer Platform BSP/Board User Manuals
4. Dual-Utmi Phy HW register restoration requirement for System Deep-Sleep feature: Some SoCs have a new utmi phy version called "dual-utmi" phy (for example: T1040, T1042, T1020, T1022, T2080, T2081: rev1.0 and rev1.1). This dual-phy hw registers need to be saved and restored across system Deep-Sleep. Hence, a code is added in u-boot usb driver that identifies all socs having this dual-utmi phy, and adds "dual_utmi" in phy_type property. This is used to determine if all phy registers are to be saved (during system-suspend) and restored (during system-resume). In absence of restoration of dual-phy hw registers, system restore during deep-sleep is going to fail - system hangs and goes into non-recoverable state.

5.11.2 USB Gadget Memory Driver User Manual

5.11.2.1 USB 2.0 Gadget Memory Driver User Manual

Description

The NXP processor has a High speed Dual-Role(DR) USB controller, which supports device mode.

Module Loading

USB device controller driver can be built in kernel or compiled as a module.

Gadget drivers are recommended to be built as modules, because parameters will be passed as module parameter

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
<pre> Device Drivers ---> USB support ---> <*> Support for Host-side USB <*> EHCI HCD (USB 2.0) support -* Root Hub Transaction Translators [] Use Xilinx usb host EHCI controller core [*] Support for Freescale PPC on-chip EHCI USB controller </pre>	<p>Need to enable CONFIG_USB_FSL_MPH_DR_OF</p>
<pre> Device Drivers ---> USB support ---> USB Gadget Support ---> < M > Support for USB Gadgets USB Peripheral Controller (Freescale Highspeed USB DR Peripheral Controller) ---> < M > Freescale Highspeed USB DR Peripheral Controller </pre>	<p>Enable NXP USB Device Controller support</p>
<pre> Device Drivers ---> USB support ---> USB Gadget Support ---> < M > Support for USB Gadgets USB Gadget Drivers < M > Mass Storage Gadget </pre>	<p>Enable USB Gadget support</p>

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_USB_SUPPORT	y/n/m	y	Enables USB Support
CONFIG_USB_FSL_MPH_DR_OF	y/n/m	y	Enable NXP EHCI USB controller
CONFIG_USB_GADGET	y/n/m	m	Enable USB Gadget modules
CONFIG_USB_GADGET_FSL_USB2	y/n/m	m	Enable NXP USB peripheral controller
CONFIG_USB_MASS_STORAGE	y/n/m	m	Enable File Storage Gadget

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/usb/gadget/fsl_usb2_udc.[ch]	NXP USB peripheral controller driver
drivers/usb/host/fsl-mp-hdr-of.c	Hook between OF tree and platform device
drivers/usb/gadget/mass_storage.c	Memory gadget driver

Device Tree Entry

```
usb@22000 {
    #address-cells = <1>;

    #size-cells = <0>;

    compatible = "fsl-usb2-<controller-type>-v<controller version>",
                "fsl-usb2-<controller-type>";

    reg = <0x22000 0x1000>;          /* specifies register base addr, soc dependent */

    interrupt-parent = <&mpic>;

    interrupts = <28 0x2>;          /* specifies usb interrupt line, soc dependent */

    phy_type = "ulpi";              /* phy can be ulpi(external)/utmi(internal) */

    dr_mode = "peripheral"          /* this entry specifies usb mode */

};
```

NOTE: Controller-type: dr(dual-role), mph(multi-port-host)
 controller-version: 1.6, 2.2, or earlier
 Default mode is always host. It can be either changed to peripheral inside the dts entry like above. In this case re-compilation of dts is required.
 dr_mode can also be changed to peripheral via u-boot command line.
 This won't require dts recompilation, and can work with default dts

```
For USB1 controller, to configure for gadget mode:  
=> setenv hwconfig 'usb1:dr_mode=peripheral,phy_type=<ulpi/utmi>
```

Test Procedure

For board specific changes (required for USB Gadget mode), please refer board BSP user manual

Preparation

1. Bring all USB Gadget modules (driver/usb/gadget/*.ko including fs/configfs/configfs.ko) onto the target board using tftp.
2. Create a (e.g. 16MB) file backed storage as under on the target:

```
bash# dd bs=1M count=16 if=/dev/zero of=/root/bkup  
  
16+0 records in  
  
16+0 records out
```

3. Load device controller driver and test mass-storage gadget

- a. Load FSL USB Gadget driver module

```
bash# insmod udc-core.ko  
bash# insmod fsl_usb2_udc.ko
```

- b. Load Mass storage module

```
bash# insmod configfs.ko  
bash# insmod libcomposite.ko  
bash# insmod usb_f_mass_storage.ko  
bash# insmod g_mass_storage.ko file=/root/bkup  
Number of LUNs=8  
Mass Storage Function, version: 2009/09/11  
LUN: removable file: (no medium)  
Number of LUNs=1  
LUN: file: /root/bkup  
Number of LUNs=1  
g_mass_storage gadget: Mass Storage Gadget, version: 2009/09/11  
g_mass_storage gadget: userspace failed to provide iSerialNumber  
g_mass_storage gadget: g_mass_storage ready
```

4. Connect a USB cable between USB port of target board and the USB port on Windows machine(host). The following message pops up on the target. At the same time XP recognizes the USB storage device and Windows configures and installs its driver.

```
bash# g_mass_storage gadget: high-speed config #1: Linux File-Backed Storage
```

5. To see the driver letter for the newly found USB storage, create a partition in /root/bkup on the target:

```
bash# fdisk /root/bkup  
  
Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklabel  
Building a new DOS disklabel with disk identifier 0x8a12fec0.  
Changes will remain in memory only, until you decide to write them.
```

```

After that, of course, the previous content won't be recoverable.

You must set cylinders.
You can do this from the extra functions menu.
Warning: invalid flag 0x0000 of partition table 4 will be corrected by w(rite)

Command (m for help): x

Expert command (m for help): s
Number of sectors (1-63, default 63): 8
Warning: setting sector offset for DOS compatibility

Expert command (m for help): h
Number of heads (1-256, default 255): 16

Expert command (m for help): c
Number of cylinders (1-1048576): 1024

Expert command (m for help): r

Command (m for help): p

Disk /root/bkup: 0 MB, 0 bytes
16 heads, 8 sectors/track, 1024 cylinders
Units = cylinders of 128 * 512 = 65536 bytes
Disk identifier: 0x8a12fec0

    Device Boot      Start         End      Blocks   Id  System

Command (m for help): n

Command action

    e   extended
    p   primary partition (1-4)

p

Partition number (1-4): 1

First cylinder (1-1024, default 1):
Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-1024, default 1024):
Using default value 1024

Command (m for help): p

Disk /root/bkup: 0 MB, 0 bytes
16 heads, 8 sectors/track, 1024 cylinders
Units = cylinders of 128 * 512 = 65536 bytes
Disk identifier: 0x8a12fec0

    Device Boot      Start         End      Blocks   Id  System
/root/bkup1          1         1024      65532    83  Linux
  
```

```
Command (m for help): w

The partition table has been altered!
Calling ioctl() to re-read partition table.
WARNING: Re-reading the partition table failed with error 25: Inappropriate ioctl for device.
The kernel still uses the old table.
The new table will be used at the next reboot.
Syncing disks.

bash# fdisk -lu /root/bkup

You must set cylinders.
You can do this from the extra functions menu.

Disk /root/bkup: 0 MB, 0 bytes
16 heads, 8 sectors/track, 0 cylinders, total 0 sectors
Units = sectors of 1 * 512 = 512 bytes
Disk identifier: 0x8a12fec0

   Device Boot      Start         End      Blocks   Id  System
 /root/bkup1            8       131071       65532   83  Linux
```

6. Run `losetup` to setup the loopback device with offset ($8*512 = 1024$) and create an `ext2` file system

```
bash# losetup -o 4096 /dev/loop0 /root/bkup
bash# mke2fs /dev/loop0
mke2fs 1.41.4 (27-Jan-2009)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
4096 inodes, 16380 blocks
819 blocks (5.00%) reserved for the super user
First data block=1
Maximum filesystem blocks=16777216
2 block groups
8192 blocks per group, 8192 fragments per group
2048 inodes per group
Superblock backups stored on blocks:
    8193

Writing inode tables: done
Writing superblocks and filesystem accounting information: done
This filesystem will be automatically checked every 24 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

7. Now a drive letter should appear for the USB storage device on Windows XP and should be able to read/write

Supporting Documentation

Linux USB gadget framework - <http://www.linux-usb.org/gadget/>

Further details about Linux USB file backed storage - http://www.linux-usb.org/gadget/file_storage.html

Known Limitations/Issues

If a board/platform is having multiple USB controller, they cannot be simultaneously used in "gadget/peripheral" mode. Please do not set dr_mode as "peripheral" for both the controllers at the same time.

5.11.3 USB Gadget Network Driver User Manual

5.11.3.1 USB 2.0 Gadget Network Driver User Manual

Description

The NXP processor has a High speed Dual-Role(DR) USB controller, which supports device mode

Module Loading

USB device controller driver can be built in kernel or compiled as a module.

Gadget drivers are recommended to be built as modules, because parameters will be passed as module parameter

Table 82. Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
<pre>Device Drivers ---> USB support ---> <*> Support for Host-side USB <*> EHCI HCD (USB 2.0) support -* Root Hub Transaction Translators [] Use Xilinx usb host EHCI controller core [*] Support for Freescale PPC on-chip EHCI USB controller</pre>	<p>Need to enable CONFIG_USB_FSL_MPH_DR_OF</p>
<pre>Device Drivers ---> USB support ---> USB Gadget Support ---> < M > Support for USB Gadgets USB Peripheral Controller (Freescale Highspeed USB DR Peripheral Controller) ---> Freescale Highspeed USB DR Peripheral Controller</pre>	<p>Enable NXP USB Device Controller support</p>

Table continues on the next page...

Table 82. Kernel Configure Tree View Options (continued)

Kernel Configure Tree View Options	Description
<pre> Device Drivers ---> USB support ---> USB Gadget Support ---> < M > Support for USB Gadgets USB Gadget Drivers <M> Ethernet Gadget (with CDC Ethernet support) [*]RNDIS support (NEW) []Ethernet Emulation Model (EEM) support (NEW) </pre>	Enable USB Gadget support

Table 83. Compile-time Configuration Options

Options	Values	Default	Description
CONFIG_USB_SUPPORT	y/n/m	Ym	Enable USB Support
CONFIG_USB_FSL_MPH_DR_OF	y/n/m	Y	Enable NXP EHCI USB controller
CONFIG_USB_GADGET	y/n/m	m	Enable USB Gadget modules
CONFIG_USB_GADGET_FSL_USB2	y/n/m	m	Enable NXP USB peripheral controller
CONFIG_USB_ETH	y/n/m	m	Enable Ethernet Gadget
CONFIG_USB_ETH_RNDIS	y/n	y	Enable Ethernet Gadget

Source Files

The driver source is maintained in the Linux kernel source tree.

Table 84. Source Files

Source File	Description
drivers/usb/gadget/fsl_usb2_udc.c	NXP USB peripheral controller driver
drivers/usb/host/fsl-mph-dr-of.c	Hook between OF tree and platform device
drivers/usb/gadget/ether.c	Ethernet gadget driver
Drivers/usb/gadget/rndis.[ch]	Microsoft's RNDIS support

Device Tree Entry

```
usb@22000 {
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl-usb2-<controller-type>-v<controller version>",
                "fsl-usb2-<controller-type>";
    reg = <0x22000 0x1000>; /* specifies register base addr, soc dependent */
    interrupt-parent = <&mpic>;
    interrupts = <28 0x2>; /* specifies usb interrupt line, soc dependent */
    phy_type = "ulpi"; /* phy can be ulpi(external)/utmi(internal) */
    dr_mode = "peripheral" /* this entry specifies usb mode */
};
```

NOTE

Controller-type: dr(dual-role), mph(multi-port-host) controller-version: 1.6, 2.2, or earlier default mode is always host. It can be either changed to peripheral inside the dts entry like above. In this case re-compilation of dts is required. DR mode can also be changed to peripheral via u-boot command line. This won't require DTS recompilation, and can work with default DTS For USB1 controller.

```
=> setenv hwconfig 'usb1:dr_mode=peripheral,phy_type=<ulpi/utmi>
```

Test Procedure

For board specific changes (required for USB Gadget mode), please refer to the board BSP User Manual.

1. Bring all USB Gadget modules (driver/usb/gadget/*.ko including fs/configfs/configfs.ko) onto the target board.
2. Load device controller driver and test ethernet gadget

Load FSL gadget driver module **udc-core.ko** & **fsl_usb2_udc.ko**

```
bash# insmod udc-core.ko
bash# insmod fsl_usb2_udc.ko
```

Load Ethernet modules

```
bash# insmod configfs.ko
bash# insmod libcomposite.ko
bash# insmod u_ether.ko
bash# insmod u_rndis.ko
bash# insmod usb_f_rndis.ko
bash# insmod usb_f_ecm.ko
bash# insmod usb_f_ecm_subset.ko
bash# insmod g_ether.ko
using random self ethernet address
using random host ethernet address
usb0: HOST MAC 82:14:b4:63:d1:85
usb0: MAC 4a:b1:59:3b:b3:bd
using random self ethernet address
using random host ethernet address
g_ether gadget: Ethernet Gadget, version: Memorial Day 2008
g_ether gadget: g_ether ready
```

```
bash# ifconfig usb0
usb0      Link encap:Ethernet  HWaddr 5e:0c:de:2f:f9:0f
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

3. Assign an IP to usb0

```
bash# ifconfig usb0 10.232.1.11 netmask 255.255.255.0 up
IPv6: ADDRCONF(NETDEV_UP): usb0: link is not ready

bash# ifconfig usb0

usb0      usb0      Link encap:Ethernet  HWaddr 5e:0c:de:2f:f9:0f
          inet addr:10.232.1.11  Bcast:10.232.1.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

4. Connect a USB cable between target board USB port and the USB port on Windows host machine.

As soon as USB cable is plugged into a Windows XP host, the following message displays:

```
http://embedded.seattle.intel-research.net/wiki/index.php?
title=Setting_up_USBnet#Install_the_RNDIS_Driver
```

5. Download linux.inf from either of the following, and install the Windows XP RNDIS driver as mentioned in the previous step:

```
http://www.davehylands.com/linux/gumstix/usbnet/linux.inf
http://embedded.seattle.intel-research.net/wiki/files/linux.inf
```

For Windows 7, driver will automatically install.

6. As soon as driver installed on host, the following message displays on the target:

```
bash# g_ether gadget: high-speed config #2: RNDIS
IPv6: ADDRCONF(NETDEV_CHANGE): usb0: link becomes ready
```

7. Once the RNDIS driver is installed, configured, and loaded, configure the IP address for the new network device.

For example, assign 10.232.1.10 as IP to the RNDIS device and run ipconfig to verify the network configuration.

8. Now run ping both ways to check the connectivity between RNDIS@Windows and usb0@linux

```
D:\Profiles>ping 10.232.1.11

Pinging 10.232.1.11 with 32 bytes of data:

Reply from 10.232.1.11: bytes=32 time<1ms TTL=64
Reply from 10.232.1.11: bytes=32 time<1ms TTL=64
Reply from 10.232.1.11: bytes=32 time<1ms TTL=64

Ping statistics for 10.232.1.11:
    Packets: Sent = 3, Received = 3, Lost = 0 (0% loss),
```

```
Approximate round trip times in milli-seconds:
  Minimum = 0ms, Maximum = 0ms, Average = 0ms

bash# ping 10.232.1.10

PING 10.232.1.10 (10.232.1.10): 56 data bytes

64 bytes from 10.232.1.10: seq=0 ttl=128 time=4.352 ms
64 bytes from 10.232.1.10: seq=1 ttl=128 time=1.015 ms
64 bytes from 10.232.1.10: seq=2 ttl=128 time=0.974 ms
64 bytes from 10.232.1.10: seq=3 ttl=128 time=0.935 ms
64 bytes from 10.232.1.10: seq=4 ttl=128 time=1.021 ms

--- 10.232.1.10 ping statistics ---

5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.935/1.659/4.352 ms
```

Known Bugs, Limitations, or Technical Issues

If a board/platform is having multiple USB controller, they cannot be simultaneously used in "gadget/peripheral" mode. Please do not set `dr_mode` as "peripheral" for both the controllers at the same time.

Supporting Documentation

Linux USB gadget framework - <http://www.linux-usb.org/gadget/>

Please refer to http://embedded.seattle.intel-research.net/wiki/index.php?title=Setting_up_USBnet for setting up the RNDIS on Windows XP.

Linux USBnet @ <http://www.linux-usb.org/usbnet/>

5.11.4 USB 3.0 Host/Peripheral Linux Driver User Manual

5.11.4.1 USB 3.0 Host/Peripheral Linux Driver User Manual

Description

The driver supports xHCI SuperSpeed (SS) Dual-Role-Device (DRD) controller

Main features of xHCI controller

- Supports operation as a standalone USB xHCI host controller
- USB dual-role operation and can be configured as host or device
- Super-speed (5 GT/s), High-speed (480 Mbps), full-speed (12 Mbps), and low-speed (1.5 Mbps) operations
- Supports operation as a standalone single port USB
- Supports eight programmable, bidirectional USB endpoints
- OTG (On-The-Go) 2.0 compliant, which includes both device and host capability.

Modes of Operation

- Host Mode: SS/HS/FS/LS
- Device Mode: SS/HS/FS

- OTG: HS/FS/LS

NOTE

Super-speed operation is not supported when OTG is enabled

NOTE

This document explains working of **HS Host and HS Device** in Linux

Module Loading

The default kernel configuration enables support for USB_DWC3 as built-in kernel module.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
<pre>Device Drivers---> USB support ---> [*] Support for Host-side USB</pre>	Enables USB host controller support
<pre>Device Drivers---> USB support ---> <*> xHCI HCD (USB 3.0) support</pre>	Enables XHCI Host Controller Driver and transaction translator
<pre>Device Drivers---> USB support ---> <*> USB Mass Storage support [] USB Mass Storage verbose debug</pre>	Enable support for USB mass storage devices. This is the driver needed for USB flash devices, and memory sticks
<pre><*> Sound card support ---> <*> Advanced Linux Sound Architecture ---> <*> OSS Mixer API <*> OSS PCM (digital audio) API [*] OSS PCM (digital audio) API - Include plugin system [*] Support old ALSA API [*] USB sound devices ---> <*> USB Audio/MIDI driver</pre>	Enables support for USB Audio devices. This driver is needed for USB microphone.
<pre>Device Drivers---> USB support ---> <*> USB Gadget Support ---></pre>	<p>Note: Required only for USB Gadget/Peripheral Support</p> <ul style="list-style-type: none"> • Enable driver for peripheral/device controller

Table continues on the next page...

Table continued from the previous page...

Kernel Configure Tree View Options	Description
<pre> <M> USB Gadget Drivers < > USB functions configurable through configs < > Gadget Zero (DEVELOPMENT) <M> Ethernet Gadget (with CDC Ethernet support) [*] RNDIS support [] Ethernet Emulation Model (EEM) support < > Network Control Model (NCM) support < > Gadget Filesystem < > Function Filesystem <M> Mass Storage Gadget < > Serial Gadget (with CDC ACM and CDC OBEX support) </pre>	<ul style="list-style-type: none"> • Enable Ethernet Gadget Client driver • Enable Mass Storage Client driver
<pre> Device Drivers----> <*> DesignWare USB3 DRD Core Support </pre>	Enable XHCI DRD Core Support

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_USB	y/m/n	y	Enables USB host controller
CONFIG_USB_XHCI_HCD	y/m/n	y	Enables XHCI HCD
CONFIG_USB_DWC3	y/m/n	y	Enables DWC3 Controller
CONFIG_USB_GADGET	y/m/n	n	Enables USB peripheral device
CONFIG_USB_ETH	y/m/n	n	Enable Ethernet style communication
CONFIG_USB_MASS_STORAGE	m/n	n	Enable USB Mass Storage disk drive
CONFIG_SOUND	y/m/n	y	Enables Sound Card Support
CONFIG_SND	y/m/n	y	Enables ALSA (Advanced Linux Sound Architecture)
CONFIG_SND_MIXER_OSS	y/m/n	y	Enables OSS Mixer API
CONFIG_SND_PCM_OSS	y/m/n	y	Enables OSS PCM (digital audio) API
CONFIG_SND_PCM_OSS_PLUGINS	y/n	y	Enables OSS PCM (digital audio) API - Include plugin system
CONFIG_SND_SUPPORT_OLD_API	y/n	y	Enables old ALSA API
CONFIG_SND_USB	y/n	n	Enables USB sound devices
CONFIG_SND_USB_AUDIO	y/m/n	n	Enables USB Audio/MIDI driver

NOTE: USB Audio configuration options default value is listed for LS1021A platform.

Source Files

The driver source is maintained in the Linux kernel source tree in below files

Table continued from the previous page...

Source File	Description
drivers/usb/host/xhci-*	xhci platform driver
drivers/usb/gadget/mass_storage.c	USB Mass Storage
drivers/usb/gadget/ether.c	Ethernet gadget driver

Device Tree Binding for Host

```
usb@3100000 {
    compatible = "snps,dwc3";
    reg = <0x0 0x3100000 0x0 0x10000>;
    interrupts = <GIC_SPI 93 IRQ_TYPE_LEVEL_HIGH>;
    dr_mode = "host";
};
```

Device Tree Binding for Peripheral

```
usb@3100000 {
    compatible = "snps,dwc3";
    reg = <0x0 0x3100000 0x0 0x10000>;
    interrupts = <GIC_SPI 93 IRQ_TYPE_LEVEL_HIGH>;
    dr_mode = "peripheral";
    maximum-speed = "super-speed";
};
```

Host Testing

Following are serial console logs that appear during bootup if dr_mode set to host in device-tree

```
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb
xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 1
xhci-hcd xhci-hcd.0.auto: irq 125, io mem 0x03100000
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 1 port detected
xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 2
hub 2-0:1.0: USB hub found
hub 2-0:1.0: 1 port detected
usbcore: registered new interface driver usb-storage
```

Following are serial-console logs after connecting a USB flash drive

```
For High-Speed Device attach
usb 1-1.2: new high-speed USB device number 3 using xhci-hcd
usb-storage 1-1.2:1.0: USB Mass Storage device detected
scsi0 : usb-storage 1-1.2:1.0
scsi 0:0:0:0: Direct-Access      SanDisk  Cruzer           7.01 PQ: 0 ANSI: 0 CCS
```

```
sd 0:0:0:0: [sda] 1957887 512-byte logical blocks: (1.00 GB/955 MiB)
sd 0:0:0:0: Attached scsi generic sg0 type 0
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] No Caching mode page found
sd 0:0:0:0: [sda] Assuming drive cache: write through
sd 0:0:0:0: [sda] No Caching mode page found
sd 0:0:0:0: [sda] Assuming drive cache: write through
sda: sda1
sd 0:0:0:0: [sda] No Caching mode page found
sd 0:0:0:0: [sda] Assuming drive cache: write through
sd 0:0:0:0: [sda] Attached SCSI removable disk
```

For Super-Speed Device attach

```
# usb 2-1: new SuperSpeed USB device number 2 using xhci-hcd
usb 2-1: Parent hub missing LPM exit latency info. Power management will be impacted.
usb-storage 2-1:1.0: USB Mass Storage device detected
scsi0 : usb-storage 2-1:1.0
scsi 0:0:0:0: Direct-Access SanDisk Extreme 0001 PQ: 0 ANSI: 6
sd 0:0:0:0: [sda] 31277232 512-byte logical blocks: (16.0 GB/14.9 GiB)
sd 0:0:0:0: Attached scsi generic sg0 type 0
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Write cache: disabled, read cache: enabled, doesn't support DPO or FUA
sda:
sd 0:0:0:0: [sda] Attached SCSI removable disk
FAT-fs (sda): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
```

Make filesystem and mount connected USB flash drive using below commands

```
root@freescale /# fdisk -l

Disk /dev/sda: 16.0 GB, 16013942784 bytes
255 heads, 63 sectors/track, 1946 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1                1         1946    15631213+  83  Linux
root@freescale /#
root@freescale /# df
Filesystem            1K-blocks      Used Available Use% Mounted on
shm                   516684          0    516684   0% /dev/shm
rwfs                   512            0         512   0% /mnt/rwfs
root@freescale /#
root@freescale /#
root@freescale /#
root@freescale /#
root@freescale /# fdisk /dev/sda
```

The number of cylinders for this disk is set to 1946.
 There is nothing wrong with that, but this is larger than 1024,
 and could in certain setups cause problems with:

- 1) software that runs at boot time (e.g., old versions of LILO)
- 2) booting and partitioning software from other OSs
 (e.g., DOS FDISK, OS/2 FDISK)

```
Command (m for help): d
Selected partition 1
```

```
Command (m for help): n
Command action
e   extended
```

Linux Kernel Drivers
Universal Serial Bus Interfaces

```
p primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-1946, default 1): Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-1946, default 1946): Using default value 1946

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table
root@freescale /$
root@freescale /$
root@freescale /$ fdisk -l

Disk /dev/sda: 16.0 GB, 16013942784 bytes
255 heads, 63 sectors/track, 1946 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1            1         1946    15631213+  83  Linux
root@freescale /$ df
Filesystem            1K-blocks      Used Available Use% Mounted on
shm                   516684          0    516684   0% /dev/shm
rwfs                   512            0         512   0% /mnt/rwfs
root@freescale /$ mkdir my_mnt
root@freescale /$
root@freescale /$
root@freescale /$ mkfs.ext2 /dev/sda1
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
977280 inodes, 3907803 blocks
195390 blocks (5%) reserved for the super user
First data block=0
Maximum filesystem blocks=4194304
120 block groups
32768 blocks per group, 32768 fragments per group
8144 inodes per group
Superblock backups stored on blocks:
    32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$ mount /dev/sda1 my_mnt/
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$ df
Filesystem            1K-blocks      Used Available Use% Mounted on
shm                   516684          0    516684   0% /dev/shm
rwfs                   512            0         512   0% /mnt/rwfs
/dev/sda1             15385852        20   14604272   0% /my_mnt
root@freescale /$
```

Test by writing/reading data on mount drive

```
root@freescale /$ dd if=/dev/urandom of=/tmp/123 bs=1M count=100
100+0 records in
100+0 records out
104857600 bytes (100.0MB) copied, 54.535026 seconds, 1.8MB/s
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$ cp /tmp/123 /my_mnt/.
root@freescale /$ sync
root@freescale /$ ls /my_mnt/
123          lost+found
root@freescale /$
```

Peripheral testing with Win7 as Host

NOTE

In gadget mode standard USB cables with micro plug should be used.

Below Message will appear during bootup if dr_mode set as peripheral in device-tree

```
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb

usbcore: registered new interface driver usb-storage
```

Make sure "dr_mode" contains "peripheral" string

```
# cat /proc/device-tree/soc/usb@3100000/dwc3/dr_mode
peripheralroot@ls1021aqds:~#
```

Move all below modules to platform

```
fs/configfs/configfs.ko
driver/usb/gadget/libcomposite.ko
driver/usb/gadget/g_mass_storage.ko
driver/usb/gadget/u_rndis.ko
driver/usb/gadget/u_ether.ko
driver/usb/gadget/usb_f_ecm.ko
driver/usb/gadget/usb_f_ecm_subset.ko
driver/usb/gadget/usb_f_rndis.ko
driver/usb/gadget/g_ether.ko
```

Mass Storage Gadget

To use ramdisk as a backing store use the following

```
root@ls1021aqds:/home# mkdir /mnt/ramdrive
root@ls1021a-# mount -t tmpfs tmpfs /mnt/ramdrive -o size=600M
root-# dd if=/dev/zero of=/mnt/ramdrive/vfat-file bs=1M count=500
root@ls1021aqds:/home# mke2fs -F /mnt/ramdrive/vfat-file
root@ls1021aqds:/home# insmod configfs.ko
root@ls1021aqds:/home# insmod libcomposite.ko
root-# insmod g_mass_storage.ko file=/mnt/ramdrive/vfat-file stall=n
```

We will get below messages

```
[ 39.987594] g_mass_storage gadget: Mass Storage Function, version: 2009/09/11
[ 39.994822] g_mass_storage gadget: Number of LUNs=1
[ 39.989240] lun0: LUN: file: /home/backing_file_20mb
[ 39.994367] g_mass_storage gadget: Mass Storage Gadget, version: 2009/09/11
[ 39.990902] g_mass_storage gadget: userspace failed to provide iSerialNumber
[ 39.987547] g_mass_storage gadget: g_mass_storage ready
```

Attached ***USB3.0 only*** gadget cable to host and you will get below message. Now Storage is ready to use.

```
g_mass_storage gadget: super-speed config #1: Linux File-Backed Storage
```

Speaker and Microphone

1. Aplay utility can be used to list the available sound cards e.g. Here Jabra 410 USB speaker is detected as a second sound card and can be addressed as **-D hw:1,0 OR -c1**:

```
[root@freescale ~]$ aplay -l**** List of PLAYBACK Hardware Devices ****
card 0: FSLVF610TWRBOAR [FSL-VF610-TWR-BOARD], device 0: HiFi sgt15000-0 [ ]
Subdevices: 1/1
Subdevice #0: subdevice #0
card 1: USB [Jabra SPEAK 410 USB], device 0: USB Audio [USB Audio] Subdevices: 1/1 Subdevice
#0: subdevice #0
```

2. Sample wav file can be played using the below command:

```
[root@freescale ~]$ aplay -D hw:1,0 LYNC_fsringing.wav
Playing WAVE 'LYNC_fsringing.wav' : Signed 16 bit Little Endian, Rate 48000 Hz, Stereo
```

3. Sample wav file can be recorded using the below command:

```
[root@freescale ~]$ arecord -f S16_LE -t wav -Dhw:1,0 -r 16000 foobar.wav -d 5
Recording WAVE 'foobar.wav' : Signed 16 bit Little Endian, Rate 16000 Hz, Mono
```

NOTE: If recorded audio is not played, try to use "-D plughw:1,0" in above command.

4. Audio controls can be checked using the below command, control details and name of the controls can be checked from output of "amixer -c1" as below:

```
[root@freescale ~]$ amixer -c1 controls
numid=3,iface=MIXER,name='PCM Playback Switch'
numid=4,iface=MIXER,name='PCM Playback Volume'
numid=5,iface=MIXER,name='Headset Capture Switch'
numid=6,iface=MIXER,name='Headset Capture Volume'
numid=2,iface=PCM,name='Capture Channel Map'
numid=1,iface=PCM,name='Playback Channel Map'

[root@freescale ~]$ amixer -c1
Simple mixer control 'PCM',0 Capabilities: pvolume pvolume-joined pswitch pswitch-joined penum
Playback channels: Mono
Limits: Playback 0 - 11
Mono: Playback 4 [36%] [-20.00dB] [on]

Simple mixer control 'Headset',0 Capabilities: cvolume cvolume-joined cswitch cswitch-joined penum
Capture channels: Mono
```

```
Limits: Capture 0 - 7
Mono: Capture 5 [71%] [0.00dB] [on]
```

For Example, in above output there are two controls named “PCM” and “Headset” for Speaker and microphone respectively.

Sample Audio controls Usage:

a. mute/unmute

```
[root@freescale ~]$ amixer -c1 set PCM mute
Simple mixer control 'PCM',0
  Capabilities: pvolume pvolume-joined pswitch pswitch-joined
  Playback channels: Mono
  Limits: Playback 0 - 11
  Mono: Playback 2 [18%] [-28.00dB] [off]
[root@freescale ~]$ amixer -c1 set PCM unmute
Simple mixer control 'PCM',0
  Capabilities: pvolume pvolume-joined pswitch pswitch-joined
  Playback channels: Mono
  Limits: Playback 0 - 11
  Mono: Playback 2 [18%] [-28.00dB] [on]
```

b. volume up/down – Below commands are trying to set volume to 11 and 2 performing volume up and down respectively.

```
root@freescale ~]$ amixer -c1 set PCM 11
Simple mixer control 'PCM',0
  Capabilities: pvolume pvolume-joined pswitch pswitch-joined
  Playback channels: Mono
  Limits: Playback 0 - 11
  Mono: Playback 11 [100%] [8.00dB] [on]
[root@freescale ~]$ amixer -c1 set PCM 2
Simple mixer control 'PCM',0
  Capabilities: pvolume pvolume-joined pswitch pswitch-joined
  Playback channels: Mono
  Limits: Playback 0 - 11
  Mono: Playback 2 [18%] [-28.00dB] [on]
```

Ethernet Gadget

To use Ethernet gadget use the following

```
root@ls1021aqds:/home# insmod configfs.ko
root@ls1021aqds:/home# insmod libcomposite.ko
root@ls1021aqds:/home# insmod u_ether.ko
root@ls1021aqds:/home# insmod u_rndis.ko
root@ls1021aqds:/home# insmod usb_f_ecm.ko
root@ls1021aqds:/home# insmod usb_f_ecm_subset.ko
root@ls1021aqds:/home# insmod usb_f_rndis.ko
root@ls1021aqds:/home# insmod g_ether.ko
```

We will get below messages

```
[ 28.692611] using random self ethernet address
[ 28.697156] using random host ethernet address
[ 28.694271] usb0: HOST MAC 82:96:69:7e:a5:7d
[ 28.698928] usb0: MAC 72:00:a5:80:2b:e8
[ 28.692586] using random self ethernet address
[ 28.697080] using random host ethernet address
```

Linux Kernel Drivers
Universal Serial Bus Interfaces

```
[ 28.691368] g_ether gadget: Ethernet Gadget, version: Memorial Day 2008  
[ 28.698028] g_ether gadget: g_ether ready
```

Make sure USB0 ethernet interface is available after this

```
root@ls1021aqds:/home# ifconfig -a  
can0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  
          NOARP  MTU:16  Metric:1  
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0  
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0  
          collisions:0 txqueuelen:10  
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)  
          Interrupt:158  
  
can1      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  
          NOARP  MTU:16  Metric:1  
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0  
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0  
          collisions:0 txqueuelen:10  
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)  
          Interrupt:159  
  
eth0      Link encap:Ethernet  HWaddr 00:E0:0C:BC:E5:60  
          BROADCAST MULTICAST  MTU:1500  Metric:1  
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0  
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0  
          collisions:0 txqueuelen:1000  
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)  
  
eth1      Link encap:Ethernet  HWaddr 00:E0:0C:BC:E5:61  
          BROADCAST MULTICAST  MTU:1500  Metric:1  
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0  
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0  
          collisions:0 txqueuelen:1000  
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)  
  
eth2      Link encap:Ethernet  HWaddr 00:E0:0C:BC:E5:62  
          inet addr:10.232.132.212  Bcast:10.232.135.255  Mask:255.255.252.0  
          inet6 addr: fe80::2e0:cff:febc:e562/64  Scope:Link  
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1  
          RX packets:2311 errors:0 dropped:3 overruns:0 frame:0  
          TX packets:66 errors:0 dropped:0 overruns:0 carrier:0  
          collisions:0 txqueuelen:1000  
          RX bytes:290810 (283.9 KiB)  TX bytes:8976 (8.7 KiB)  
  
lo        Link encap:Local Loopback  
          inet addr:127.0.0.1  Mask:255.0.0.0  
          inet6 addr: ::1/128  Scope:Host  
          UP LOOPBACK RUNNING  MTU:65536  Metric:1  
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0  
          TX packets:2 errors:0 dropped:0 overruns:0 carrier:0  
          collisions:0 txqueuelen:0  
          RX bytes:100 (100.0 B)  TX bytes:100 (100.0 B)  
  
sit0      Link encap:IPv6-in-IPv4  
          NOARP  MTU:1480  Metric:1  
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0  
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0  
          collisions:0 txqueuelen:0
```

```

RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

usb0    Link encap:Ethernet  HWaddr 72:00:A5:80:2B:E8
        BROADCAST MULTICAST  MTU:1500  Metric:1
        RX packets:0 errors:0 dropped:0 overruns:0 frame:0
        TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:1000
        RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

```

Attached the cable with Win7 and Configure RNDIS interface in windows under "Control Panel -> Network and Internet -> Network Connections" and set IP Address

Set IP Address in Platform and start Ping

```

root@ls1021aqds:/home# ifconfig usb0 10.232.1.11
root@ls1021aqds:/home#
root@ls1021aqds:/home#
root@ls1021aqds:/home# ping usb 10.232.1.10
PING 10.232.1.10 (10.232.1.10): 56 data bytes
64 bytes from 10.232.1.10: seq=0 ttl=128 time=5.294 ms
64 bytes from 10.232.1.10: seq=1 ttl=128 time=6.101 ms
64 bytes from 10.232.1.10: seq=2 ttl=128 time=4.170 ms
64 bytes from 10.232.1.10: seq=3 ttl=128 time=4.233 ms

```

Known Bugs, Limitations, or Technical Issues

- Some issue with Pen drives from Kingston/Transcend. This have noticed some patches floating in open-source for these issues, and also found that open-source USB community trying to fix.
- Linux allow only one peripheral at one time. Please make sure When DWC3 set as Peripheral the other should not be set in same mode.
- Erratum:A-009116 (Frame length of USB3 controller for USB2.0 and USB3.0 operation is incorrect) impacts some socs like LS1020A/LS1021A because of which some USB2.0 and USB3.0 devices may not work properly, and hence, a sw workaround is needed. This sw workaround involves programing following registers of XHCI controller as: GFLADJ[5:0] = 20H and GFLADJ[7] = 1. This is already done via u-boot and linux codebase.

5.12 Watchdog Timers

5.12.1 Watchdog Device Driver User Manual

Description

Watchdog driver description here.

Module Loading

Watchdog device driver support kernel built-in mode.

U-Boot Configuration

Runtime options

Env Variable	Env Description	Sub option	Option Description
bootargs	Kernel command line argument passed to kernel	setenv othbootargs wdt_period=35	Sets the watchdog timer period timeout

Kernel Configure Options

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
<pre> Device Drivers ---> [*] Watchdog Timer Support ---> [*] Disable watchdog shutdown on close [*] PowerPC Book-E Watchdog Timer </pre>	PowerPC Book-E Watchdog Timer

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_BOOKE_WDT	y/n	y	PowerPC Book-E Watchdog Timer

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/char/watchdog/booke_wdt.c	PowerPC Book-E Watchdog Timer

User Space Application

The following applications will be used during functional or performance testing. Please refer to the SDK UM document for the detailed build procedure.

Command Name	Description	Package Name
watch	watchdog is a daemon for watchdog feeding	watchdog

Verification in Linux

· set nfs rootfs

```
build a rootfs image which includes watchdog daemon.
```

· et booting parameter

on the u-boot prompt, set following parameter

```
set nfsargs "setenv bootargs wdt_period=35 root=/dev/nfs rw nfsroot=$serverip:$rootpath ip=
$ipaddr:$serverip:$gatewayip:$netmask:$hostname:$netdev:off
```

```
console=$consoledev,$baudrate $othbootargs"
```

```
set nfsboot "run nfsargs;tftp $loadaddr $bootfile;tftp $fdtaddr $fdtfile;bootm $loadaddr -
$fdtaddr"
```

```
run nfsboot
```

Note: wdt_period is watchdog timeout period, set it with proper value depending on your board bus frequency.

Also wdt_period is inversely proportional to watchdog expiry time ie. Higher the wdt_period, lower the watchdog expiry time.

So if we increase wdt_period to high, watchdog will expiry early.

· check watchdog feeding operation

after system boots up, check the screen output, if you see

...

```
PowerPC Book-E Watchdog Timer Enabled (wdt_period=35)
```

...

it means watchdog module loads successfully

login in system, run command "watchdog /dev/watchdog"

```
root@p1020rdb:~# watchdog /dev/watchdog
```

```
root@p1020rdb:~# ps -ae | grep watchdog
```

```
3285 ?          00:00:00 watchdog
```

```
root@p1020rdb:~#
```

wait for some minutes, if system is still alive, watchdog feeding is OK

· check watchdog reboot operation

run command "killall"

```
root@p1020rdb:~# killall -9 watchdog
```

```
root@p1020rdb:~#
```

```
root@p1020rdb:~# ps -ae | grep watchdog
```

```
root@p1020rdb:~#
```

```
root@p1020rdb:~# PowerPC Book-E Watchdog Exception
```

```
wait for some seconds, if system reboots, watchdog reboot operation is OK
```

Known Bugs, Limitations, or Technical Issues

- On the T4240RDB board, if you will use watchdog, please disable the following menu configuration in kernel

Location:

```
    |--> Device  
Drivers  
    |--> Hardware Monitoring support (HWMON [=n])  
Or they are conflicting with each other.
```

Chapter 6 Additional Linux Use Cases

6.1 Power Management

6.1.1 Power Management User Manual

Linux SDK for QorIQ Processors

Description

QorIQ Processors have features to minimize power consumption at several different levels. All processors support a sleep mode (LPM20). Some processors, such as T1040, LS1021, LS1046, also support a deep sleep mode (LPM35).

The following power management features are supported on various QorIQ processors:

- Dynamic power management
- Shutting down unused IP blocks
- Cores support low power modes (such as PW15)
- Processors enter low power state (LPM20, LPM35)
 - LPM20 mode: most part of processor clocks are shut down
 - LPM35 mode: power is removed to cores, cache and IP blocks of the processor such as DIU, eLBC, PEX, eTSEC, USB, SATA, eSDHC etc.
- CPU hotplug: If cores are down at runtime, they will enter low power state.

The wake-up event sources caused quitting from low power mode are listed as below:

- Wake on LAN (WoL) using magic packet
- Wake by MPIC timer or FlexTimer
- Wake by Internal and external interrupts

For more information on a specific processor, refer to processor Reference Manual.

Kernel Configure Tree View Options

For ARM platforms

Kernel Configure Tree View Options	Description
<pre>Power management options --> [*] Suspend to RAM and standby</pre>	Enable sleep feature
<pre>Device Drivers ---> SOC (System On Chip) specific Drivers ---></pre>	Enable the FTM alarm (FlexTimer module) driver

Table continues on the next page...

Table continued from the previous page...

Kernel Configure Tree View Options	Description
<pre>[*] Layerscape Soc Drivers [*] FTM alarm driver</pre>	
<pre>CPU Power Management ---> CPU Idle ---> [*] CPU idle PM support [*] Ladder governor (for periodic timer tick) -*- Menu governor (for tickless system) ARM CPU Idle Drivers ---> [*] Generic ARM/ARM64 CPU idle Driver</pre>	Enable the CPU Idle driver
<i>Table continues on the next page...</i>	

Compile-time Configuration Options

Linux Framework	Hardware Feature	Platform	Kernel Config
Suspend	LPM20	LS1012, LS1021, LS1046	CONFIG_SUSPEND
	wake by Flextimer	LS1012, LS1021, LS1046	CONFIG_FTM_ALARM
CPU idle	PW15	LS1012, LS1021, LS1046	CONFIG_ARM_CPUIDLE

Device Tree Binding

Property	Type	Description
fsl,#rcpm-wakeup-cells	unsigned int	the number of cells in "rcpm-wakeup" except the pointer to "rcpm"
rcpm-wakeup	unsigned int	require if the IP block can work as a wakeup source

For processors integrated RCPM

```
rcpm: rcpm@1ee2000 {
    compatible = "fsl,ls1046a-rcpm", "fsl,qoriq-rcpm-2.1";
    reg = <0x0 0x1ee2000 0x0 0x1000>;
    fsl,#rcpm-wakeup-cells = <1>;
};

ftm0: ftm0@29d0000 {
    compatible = "fsl,ftm-alarm";
    reg = <0x0 0x29d0000 0x0 0x10000>;
    interrupts = <0 86 0x4>;
    big-endian;
    rcpm-wakeup = <&rcpm 0x0 0x20000000>;
};
```

```
    status = "okay";  
};
```

Refer to the Linux document: [Documentation/devicetree/bindings/soc/fsl/rcpm.txt](#)

Source Files

The source files are maintained in the Linux kernel source tree.

Source File	Description
drivers/soc/fsl/layercape/rcpm.c	the RCPM driver needed by the sleep feature
drivers/soc/fsl/layercape/ftm_alarm.c	the FTM timer driver worked as a wakeup source
drivers/cpuidle/cpuidle-arm.c	the cpuidle driver for ARM core

Verification in Linux

- Cpuidle Driver

The cpuidle driver can switch CPU state according to the idle policy (governor). For more information, please see "Documentation/cpuidle/sysfs.txt" in kernel source code.

```
/* Check the cpuidle driver which is currently used. */  
# cat /sys/devices/system/cpu/cpuidle/current_driver  
  
/* Check the following directory to see the detailed statistic information of each state on  
each CPU. */  
/sys/devices/system/cpu/cpu0/cpuidle/state0/  
/sys/devices/system/cpu/cpu0/cpuidle/state1/
```

- Sleep and Wake up by FTM timer

```
/* Start a FTM timer. It will trigger an interrupt to wake up the system in 5 seconds. */  
echo 5 > /sys/devices/platform/soc/29d0000.ftm0/ftm_alarm && echo mem > /sys/power/state
```

Supporting Documentation

- QorIQ processor reference manuals

6.1.2 CPU Frequency Switching User Manual

Linux SDK for QorIQ Processors

Abbreviations and Acronyms

DFS: Dynamic Frequency Scaling

P2, T4, B4, ... : stand for P2xxx, T4xxx, B4xxx processors

Description

QorIQ Processors support DFS (Dynamic Frequency Switching) feature, also known as CPU Frequency Switch, which can change the frequency of cores dynamically.

For more information on a specific processor, refer to processor Reference Manual.

U-boot Configuration

None

Kernel Configure Tree View Options

For Powerpc platform

Kernel Configure Tree View Options	Description
<pre>Platform support --> CPU Frequency scaling --> [*] CPU Frequency scaling <*> CPU frequency translation statistics Default CPUFreq governor (userspace) --> *- 'userspace' governor for userspace frequency scaling PowerPC CPU frequency scaling drivers ---> <*> CPU frequency scaling driver for Freescale QorIQ SoCs CPU Frequency drivers --> [*] Support for Freescale MPC85xx CPU freq</pre>	<p>Enable the CPU frequency driver for DFS</p>

For Layerscape platform

Kernel Configure Tree View Options	Description
<pre>CPU Power Management --> CPU Frequency scaling --> [*] CPU Frequency scaling <*> CPU frequency translation statistics Default CPUFreq governor (userspace) --> *- 'userspace' governor for userspace frequency scaling ARM CPU frequency scaling drivers --> <*> CPU frequency scaling driver for Freescale QorIQ SoCs</pre>	<p>Enable the CPU frequency driver</p>

Compile-time Configuration Options

Linux Framework	Hardware Feature	Platform	Kernel Config
cpufreq	DFS	ALL	CONFIG_CPU_FREQ, CONFIG_CPU_FREQ_DEFAULT_GOV_USERSPACE
cpufreq	DFS	P2	CONFIG_MPC85xx_CPUFREQ
cpufreq	DFS	P3, P4, P5, T4, B4	CONFIG_QORIQ_CPUFREQ

User Space Application

Simply using command "cat" and "echo" can verify this feature.

Device Tree Binding

Property	Type	Status	Description
#clock-cells	unsigned int	Required	The number of cells in a clock-specifier
clock-output-names	String	Required	Clock output name
clocks	handle	Required	Clock source handle
compatible	String	Required	Compatible strings
reg	unsigned int	Required	register address range
<i>Table continues on the next page...</i>			

For processors used old clock binding, like ls1021a:

```

clockgen: clocking@1ee1000 {
    #address-cells = <1>;
    #size-cells = <1>;
    ranges = <0x0 0x0 0x1ee1000 0x10000>;

    sysclk: sysclk {
        compatible = "fixed-clock";
        #clock-cells = <0>;
        clock-output-names = "sysclk";
    };

    cga_pll1: pll@800 {
        compatible = "fsl,qoriq-core-pll-2.0";
        #clock-cells = <1>;
        reg = <0x800 0x10>;
        clocks = <&sysclk>;
        clock-output-names = "cga-pll1", "cga-pll1-div2",
            "cga-pll1-div4";
    };

    platform_clk: pll@c00 {
        compatible = "fsl,qoriq-core-pll-2.0";
        #clock-cells = <1>;
        reg = <0xc00 0x10>;
        clocks = <&sysclk>;
        clock-output-names = "platform-clk", "platform-clk-div2";
    };

    cluster1_clk: clk0c0@0 {
        compatible = "fsl,qoriq-core-mux-2.0";
        #clock-cells = <0>;
        reg = <0x0 0x10>;
        clock-names = "pll1cga", "pll1cga-div2", "pll1cga-div4";
        clocks = <&cga_pll1 0>, <&cga_pll1 1>, <&cga_pll1 2>;
        clock-output-names = "cluster1-clk";
    };
};

```

For processors used new clock framework, like ls2085a and ls1043a:

```
clockgen: clocking@1ee1000 {
    compatible = "fsl,ls1043a-clockgen";
    reg = <0x0 0x1ee1000 0x0 0x1000>;
    #clock-cells = <2>;
    clocks = <&sysclk>;
};
```

Source Files

The driver source is maintained in the Linux kernel source tree.

Table continued from the previous page...

Source File	Description
drivers/cpufreq/qoriq_cpufreq.c	CPU frequency scaling driver for qoriq chips

Verification in Linux

- CPU frequency mode

In order to test the CPU frequency scaling feature, we need to enable the CPU frequency feature on the menuconfig and choose the USERSPACE governor. You can learn more about CPU frequency scaling feature by referring to the kernel documents. They all are put under Documentation/cpu-freq/ directory. For example: all the information about governors is put in Documentation/cpu-freq/governors.txt.

Test step:

1. list all the frequencies a core can support (take cpu 0 for example) :

```
# cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_frequencies
1199999 5999999 2999999 7999999 3999999 1999999 1066666 533333 266666
```

2. check the CPU's current frequency

```
# cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq
1199999
```

3. change the CPU's frequency we expect:

```
# echo 799999 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
```

You can check the CPU's current frequency again to confirm if the frequency transition is successful.

Please note that if the frequency you want to change to doesn't support by current CPU, kernel will round up or down to one CPU supports.

Known Bugs, Limitations, or Technical Issues

- On T4 series, when the CCB frequency is 666.667MHz, CPU frequency scaling can't work correctly. When CCB frequency is 600MHz, CPU frequency scaling works fine.

Supporting Documentation

- QorIQ processor reference manuals

6.1.3 System Monitor

6.1.3.1 Power Monitor User Manual

Description

There are two methods currently we can use to measure the power consumption which are called online and offline power monitoring respectively. The difference between them is that offline power monitoring support measuring power consumption during sleep or deep sleep.

The Power Monitor can be supported on P4080DS/P5020DS/P5040DS/T4240QDS board.

This User guide uses the T4240QDS board as an example.

Online Power Monitoring

The Lm-sensors tool (download from <http://dl.lm-sensors.org/lm-sensors/releases>) will be used to read the power/temperature from on-boards sensors. The drivers vary from sensor to sensor. Basically they would be INA220, ZL6100 and ADT7461 etc.

The device driver support either a built-in kernel or module loading.

Kernel Configure Tree View Options

Option	Description
<pre>Device Drivers ---> <*> Hardware Monitoring support ---> <*> Texas Instruments INA219 and compatibles</pre>	Enables INA220
<pre>Device Drivers ---> [*] Enable compatibility bits for old user-space <*> I2C device interface [*] Autoselect pertinent helper modules I2C Hardware Bus support ---> <*> MPC107/824x/85xx/512x/52xx/83xx/86xx</pre>	Enables I2C block device driver support
<pre>Device Drivers ---> <*> I2C bus multiplexing support Multiplexer I2C Chip support ---> <*> Philips PCA954x I2C Mux/switches</pre>	Enables I2C bus multiplexing PCA9547

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_I2C_MPC	y/n	y	Enable I2C bus protocol
SENSORS_INA2XX	y/n	y	Enables INA220
CONFIG_I2C_MUX_PCA954x	y/n	y	Enables I2C multiplexing PCA9547

Device Tree Binding

Property	Type	Status	Description
compatible	String	Required	"Philips,pca9547" for pca9547
reg	integer	Required	reg = <0x77>
compatible	String	Required	"ti,ina220" for ina220
reg	integer	Required	reg = <the i2c address of ina220>

```

Default node:
    i2c@118000 {
        pca9547@77 {
            compatible = "philips,pca9547";
            reg = <0x77>;
            #address-cells = <1>;
            #size-cells = <0>;

            channel@2 {
                #address-cells = <1>;
                #size-cells = <0>;
                reg = <0x2>;

                ina220@40 {
                    compatible = "ti,ina220";
                    reg = <0x40>;
                    shunt-resistor = <1000>;
                };

                ina220@41 {
                    compatible = "ti,ina220";
                    reg = <0x41>;
                    shunt-resistor = <1000>;
                };

                ina220@44 {
                    compatible = "ti,ina220";
                    reg = <0x44>;
                    shunt-resistor = <1000>;
                };

                ina220@45 {
                    compatible = "ti,ina220";
                    reg = <0x45>;
                    shunt-resistor = <1000>;
                };

                ina220@46 {
                    compatible = "ti,ina220";
                    reg = <0x46>;
                    shunt-resistor = <1000>;
                };

                ina220@47 {
                    compatible = "ti,ina220";
                    reg = <0x47>;
                    shunt-resistor = <1000>;
                };
            };
        };
    };

```

```
};  
};  
};
```

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/i2c/muxes/i2c-mux-pca954x.c	PCA9547 driver
drivers/hwmon/ina2xx.c	ina220 driver

Test Procedure

Do the following to validate under the kernel

1. The bootup information is displayed:

```
.....  
i2c /dev entries driver  
mpc-i2c ffe118000.i2c: timeout 1000000 us  
mpc-i2c ffe118100.i2c: timeout 1000000 us  
mpc-i2c ffe119000.i2c: timeout 1000000 us  
mpc-i2c ffe119100.i2c: timeout 1000000 us  
i2c i2c-0: Added multiplexed i2c bus 6  
i2c i2c-0: Added multiplexed i2c bus 7  
i2c i2c-0: Added multiplexed i2c bus 8  
i2c i2c-0: Added multiplexed i2c bus 9  
i2c i2c-0: Added multiplexed i2c bus 10  
i2c i2c-0: Added multiplexed i2c bus 11  
i2c i2c-0: Added multiplexed i2c bus 12  
i2c i2c-0: Added multiplexed i2c bus 13  
pca954x 0-0077: registered 8 multiplexed busses for I2C mux pca9547  
ina2xx 8-0040: power monitor ina220 (Rshunt = 1000 uOhm)  
ina2xx 8-0041: power monitor ina220 (Rshunt = 1000 uOhm)  
ina2xx 8-0045: power monitor ina220 (Rshunt = 1000 uOhm)  
ina2xx 8-0046: power monitor ina220 (Rshunt = 1000 uOhm)  
ina2xx 8-0047: power monitor ina220 (Rshunt = 1000 uOhm)  
ina2xx 8-0044: power monitor ina220 (Rshunt = 1000 uOhm)  
.....
```

- 2.

```
# sensors  
ina220-i2c-8-40  
Adapter: i2c-0-mux (chan_id 2)  
in0:          +0.08 V  
in1:          +1.06 V  
power1:       35.00 W  
curr1:        +32.77 A  
  
ina220-i2c-8-41  
Adapter: i2c-0-mux (chan_id 2)  
in0:          +0.01 V  
in1:          +0.01 V  
power1:       60.00 mW  
curr1:        +6.62 A  
  
ina220-i2c-8-45
```

```

Adapter: i2c-0-mux (chan_id 2)
in0:      +0.01 V
in1:      +0.00 V
power1:   60.00 mW
curr1:    +8.20 A

ina220-i2c-8-46
Adapter: i2c-0-mux (chan_id 2)
in0:      +0.01 V
in1:      +0.01 V
power1:   60.00 mW
curr1:    +6.60 A

ina220-i2c-8-47
Adapter: i2c-0-mux (chan_id 2)
in0:      +0.01 V
in1:      +0.02 V
power1:   140.00 mW
curr1:    +7.40 A

ina220-i2c-8-44
Adapter: i2c-0-mux (chan_id 2)
in0:      +0.00 V
in1:      +1.51 V
power1:   1.86 W
curr1:    +1.23 A

```

NOTE

Please make sure to include the "sensors" command in your rootfs

Offline Power Monitoring

Inside the FPGA of some NXP QorIQ (PowerPC) reference boards is a microprocessor called the General Purpose Processor (GSMA). Running on the GSMA is the Data Collection Manager (DCM), which is used to periodically read and tally voltage, current, and temperature measurements from the on-board sensors. You can use this feature to measure power consumption while running tests, without having the host CPU perform those measurements.

This method support measuring power consumption when kernel is in sleep or deep sleep status. It gets the average power value of period from the time DCM starts to the time it ends.

Module Loading

The device driver support either kernel built-in or module.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
<pre> Device Drivers ---> [*] Misc devices ---> <*> Freescale Data Collection Manager (DCM) driver </pre>	Enables DCM driver

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_FSL_DCM	y/n	y	Enable DCM module

Device Tree Binding

Property	Type	Status	Description
compatible	String	Required	"fsl,t4240qds-fpga", "fsl,fpga-qixis"
reg	Integer	Required	reg = <3 0 0x300>

Default node:

```
ifc: localbus@ffe124000 {
    board-control@3,0 {
        compatible = "fsl,t4240qds-fpga", "fsl,fpga-qixis";
        reg = <3 0 0x300>;
    };
};
```

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/misc/fsl_dcm.c	DCM driver

Test Procedure

Do the following to validate under the Kernel:

1. The bootup information is displayed:

```
.....
Freescale Data Collection Module is installed.
.....
```

2. Start measuring measure power

```
# echo 1 > /sys/devices/platform/fsl-dcm.0/control
```

3. Stop measuring power

```
#echo 0 > /sys/devices/platform/fsl-dcm.0/control
```

4. Display the average power consumption

```
#cat /sys/devices/platform/fsl-dcm.0/result

Name                               Average
=====                             =====
CPU voltage:                        1068   (mV)
CPU current:                         25910  (mA)
DDR voltage:                         1348   (mV)
DDR current:                         740    (mA)
CPU temperature:                     38     (C)
```

6.1.3.2 Thermal Monitor User Manual

Description

The Temperature Monitoring function is provided by the chip ADT7461.

This driver exports the values of Temperature to SYSFS. The user space lm-sensors tools can get and display these values.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description
<pre>Device Drivers ---> [*] Hardware Monitoring support ---> [*] National Semiconductor LM90 and compatibles</pre>	Enable thermal monitor chip driver like ADT7461.
<pre>Device Drivers ---> <*> I2C bus multiplexing support ---> Multiplexer I2C Chip support ---> <*> Philips PCA954x I2C Mux/switches</pre>	Enable I2C PCA954x multiplexer support

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_HWMON	y/m/n	n	Enable Hardware Monitor
CONFIG_SENSORS_LM90	y/m/n	n	Enable ATD7461 driver
CONFIG_I2C_MUX	y/m/n	n	Enable I2C bus multiplexing support
CONFIG_I2C_MUX_PCA954x	y/m/n	n	Enable PCA954x driver

Device Tree Binding

```
adt7461@4c {
    compatible = "adi,adt7461";
    reg = <0x4c>;
};

pca9547@77 {
    compatible = "philips,pca9547";
    reg = <0x77>;
};
```

Source Files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/hwmon/hwmon.c	Linux hwmon subsystem support
<i>Table continues on the next page...</i>	

Table continued from the previous page...

Source File	Description
drivers/hwmon/lm90.c	ADT7461 chip driver
drivers/i2c/i2c-mux.c	I2C bus multiplexing support
drivers/i2c/muxes/pca954x.c	PCA954x chip driver

Verification in Linux

There are two ways to get temperature results.

```
1. You can manually read the thermal interfaces in sysfs:
~$ ls /sys/class/hwmon/hwmon1/devices
alarms          temp1_crit      temp1_min_alarm temp2_max_alarm
driver          temp1_crit_alarm temp2_crit        temp2_min
hwmon           temp1_crit_hyst temp2_crit_alarm  temp2_min_alarm
modalias        temp1_input     temp2_crit_hyst  temp2_offset
name            temp1_max       temp2_fault      uevent
power           temp1_max_alarm temp2_input       update_interval
subsystem       temp1_min       temp2_max
```

```
~$ cat /sys/class/hwmon/hwmon1/devices/temp1_input
29000
```

2. You can use `lm_sensors` tools as follows.

```
~ # sensors

adt7461-i2c-1-4c
Adapter: MPC adapter
temp1:      +34.0 C (low = +0.0 C, high = +85.0 C)
             (crit = +85.0 C, hyst = +75.0 C)
temp2:      +48.5 C (low = +0.0 C, high = +85.0 C)
             (crit = +85.0 C, hyst = +75.0 C)
```

lm_sensors is integrated into Yocto file system by default. If there is no "sensors" command in your rootfs just add `lmsensors-sensors` package and build your own rootfs using Yocto:

```
IMAGE_INSTALL += "lmsensors-sensors"
```

6.1.3.3 Web-based System Monitor User Guide

Monitors the health of a system using a web browser in real time.

Description

Web-based System Monitor is a tool for monitoring the health of your system using a web browser in real time. The following procedures will guide you to setup the system monitor.

Kernel requirements

The raw data of this monitor system is collected from hardware monitor chips. So before you setup this monitor system you should enable the `hwmon` subsystem and drivers of monitor chips in the kernel. Kernel configure details are listed below.

Kernel Configure Tree View Options	Description
<pre>Device Drivers ---> [*] Hardware Monitoring support ---> [*] National Semiconductor LM90 and compatibles [*] Texas Instruments INA219 and compatibles</pre>	Enable monitor chip drivers like ADT7461(ADT7481)/INA220, etc.
<pre>Device Drivers ---> <*> I2C bus multiplexing support ---> Multiplexer I2C Chip support ---> <*> Philips PCA954x I2C Mux/switches</pre>	Enable I2C PCA954x multiplexer support

Some monitor chips may not be included in the device tree. In this case you could add the device manually. Take adt7461 on T4240QDS as an example: the monitor is attached to I2C multiplexer PCA954x channel 3 in address 0x43. T4240QDS has 4 I2C controllers so the channel index of multiplexer start from 4 (represent the channel 0). ADT7461 is connected to channel 3 which indexed as 7. Use the flowing command to add the device to kernel:

```
~$ echo adt7461 0x4c > /sys/bus/i2c/devices/i2c-7/new_device
```

Rootfs requirements

You could use the fsl-image-full rootfs in which all packages needed are included.

Or you can build your own rootfs using Yotco. Please follow the steps below.

1. Add following package group to your rootfs recipes like fsl-image-core.bb:

```
IMAGE_INSTALL += "packagegroup-fsl-monitor"
```

2. If you are using ramdisk boot please add following settings to local.conf to get enough space for monitor systems:

```
IMAGE_ROOTFS_EXTRA_SPACE = "100000"
```

NOTE

This will add 100000KB (100MB) more space to rootfs for monitor database. Each sensor needs about 10MB more space for logging raw data.

Setting up system monitor

The monitor system will be setup automatically. What you need to do is to make sure that the network on board is working. Then you can monitor the system via any web browser by visiting: <http://your.ip.address/senspix/sensors.cgi>. This results page will refresh itself for every 10 seconds.

If you need to re-setup the system you could enter /usr/rrd directory and run:

```
$ make clean
$ make
```

NOTE

The System Monitor only works when system time is right. So you should guarantee that.

How to configure the system monitor

The monitor results you see is based on the configuration file: "monitor.conf". It's automatically generated by scanning the hwmon subsystem. You could manually modify it too. Here is how:

1. Each line of this configuration file represents one monitor curve. It contains four fields formatted as follows:

```
SENSDEV:MONITOR_TERM:DURATION:DESCRIPTION
```

SENSDEV: The sensor data can be monitored from /sys/class/hwmon/ interfaces. Each sensor has a corresponding folder distinguished by hwmon# like hwmon0 or hwmon1. SENSDEV is the folder name.

MONITOR_TERM: This is the item you want to read like temp1 or temp2 etc.

DURATION: This is how long you want to see the results. You can set minute/hour/day here.

DESCRIPTION: This DESCRIPTION will show up on the result picture helping you to understand the contents of the curve.

2. You could add/remove/resort the configuration file. After modifying it, you could enter the /usr/rrd directory and run:

```
$ make config
```

3. Then the monitor results will be updated to what you configured.

Run demo

We also provide scripts to cycle the system through different PM low power states to form a out-of-box demo for PM features. Please enter /usr/pm_demo directory and simply run:

```
$ ./pm_demo.sh
```

The output of the script will state the current PM features. It helps you to understand the system monitor results better.

NOTE

The demo could be terminated by CTRL-C and will apply the default PM features back.

6.2 PCIe DMA Test User Manual

6.2.1 Introduction to PCI DMA Test

Introduction

When NXP PCIE controller working in EP mode, it can be connected to a host such as x86 computer or PowerPC board, and works as a PCI device. The PCI DMA test application is for the scenario and is used to test PCI memory write performance. It contains two parts. One is PCI DMA EP application running on EP side, the other is PCI DMA host driver module running on host side.

Test Environment

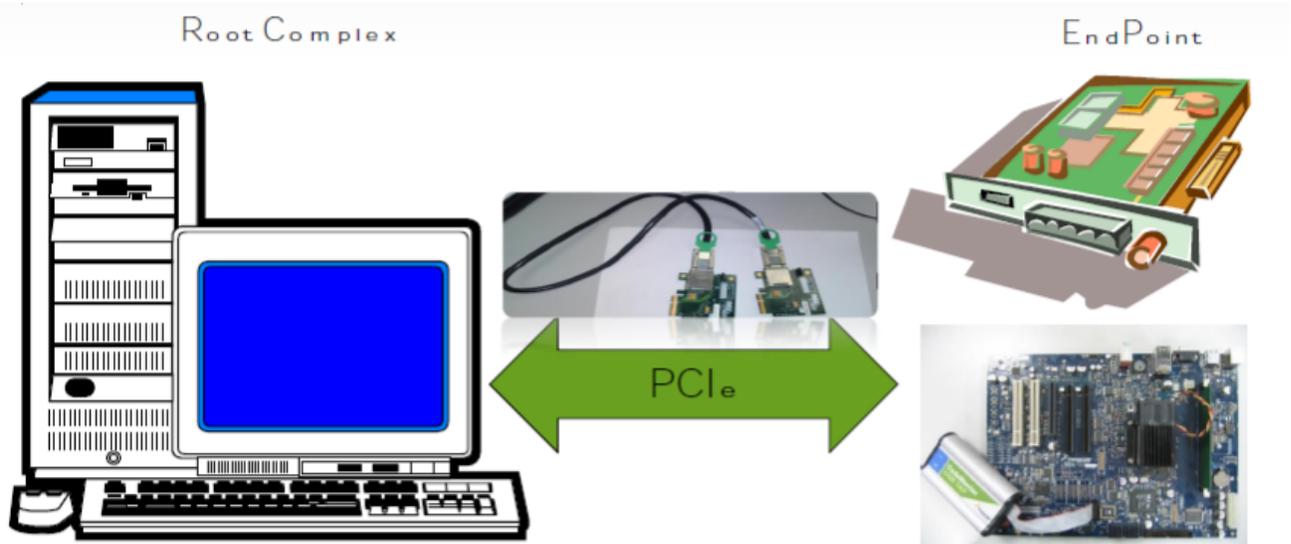


Figure 110. Test environment

As the above figure shown, we can directly insert the PCI card into host or use PCI card and cable to connect host and EP board.

Host side supports x86 and PowerPC boards.

EP side supports P5020DS, T4240QDS.

Data processing

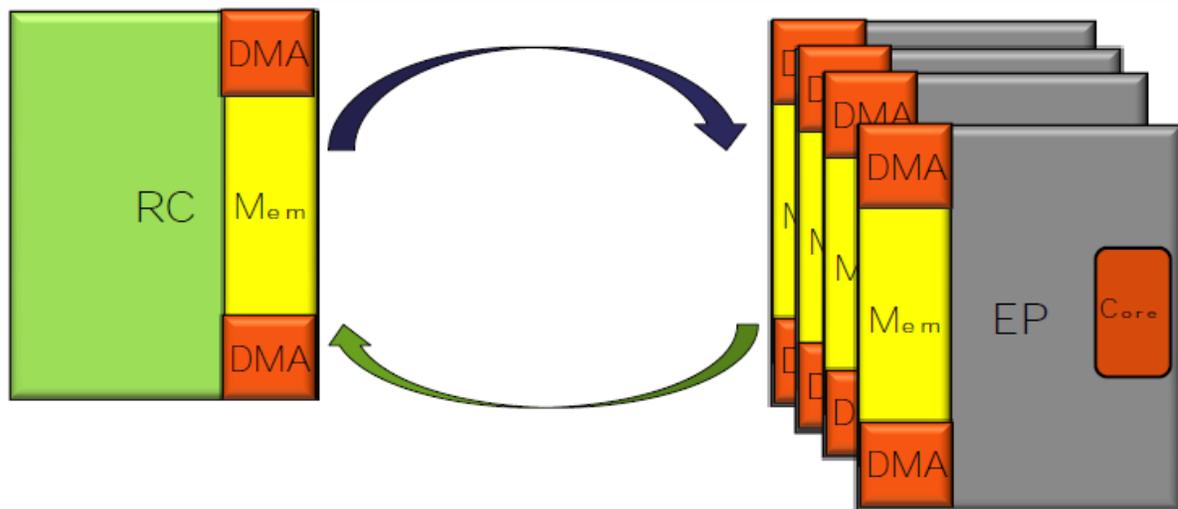


Figure 111. Data processing

As the above figure shown, EP may include multiple functions, each function is looked as a PCI device. EP application will initialize all the functions, allocate memory to store the data from host, and can write data to host. In host side, the driver module also allocates memory to store the data from EP, and can write data to the specified EP.

6.2.2 PCI DMA EP Application

Introduction

PCI DMA EP application is a user space application running on EP side. It initializes all the PCI functions of the specified PCI controller. For T4240QDS, the first PCI controller contains two physical functions and each physical function supports 64 virtual functions. Each function is looked as a PCI DMA device indexed by PF index and VF index, and is created three or four inbound windows.

1. Inbound window 0 is associated with BAR0 mapped to CCSR. Only enabled for Physical function.
2. Inbound window 1 is associated with BAR1, which is used for MSIX.
3. Inbound window 2 is associated with BAR2, which is used to store configuration information including RC command, EP status and test result.
4. Inbound window 3 is associated with BAR4, which is used to receive data from host.

The application can create a test thread on a specified PCI DMA device. Test thread will wait for host command and start performance test.

EP Kernel Image

PCI DMA EP application depends on DMA UIO driver and PCI EP VFIO driver. please make sure the two drivers are selected when compiling the EP kernel image.

```
-> Device Drivers
  ->VFIO Non-Privileged userspace driver framework
    <*>   VFIO support for NXP PCI Endpoint devices

  -> Userspace I/O drivers
<*>   NXP DMA support
```

Compile EP Application

1. Use yacto to compile EP application.

The detailed information can refer to SDK documents.

```
#> bitbake fsl-image-core
```

2. Directly Compile EP application

- a. Get the code and move them to sdk folder.

```
git clone git://git.am.freescale.net/gitolite/sdk/skmm-ep.git
git checkout remotes/origin/pciedma_multi_chan_perf -b pciedma_multi_chan_perf
```

- b. Configure the compile environment

```
lmh@lmh:~/work/sdk-devel-gerrit/skmm-ep$ source standalone-env -m t4240qds
```

- c. Compile code

```
lmh@lmh:~/work/sdk-devel-gerrit/skmm-ep$ make
[CC] common.c          (lib:skmm_common)
[AR] libskmm_common.a
[CC] skmm_pci.c        (lib:skmm_pci)
[AR] libskmm_pci.a
[CC] skmm_sec.c        (lib:skmm_sec)
```

```
[AR] libskmm_sec.a
[CC] skmm_dds.c      (lib:skmm_mem)
[AR] libskmm_mem.a
[CC] skmm_sec_blob.c (lib:skmm_sec_blob)
[AR] libskmm_sec_blob.a
[CC] skmm_memmgr.c  (lib:skmm_memmgr)
[AR] libskmm_memmgr.a
[CC] skmm_uio.c     (lib:skmm_uio)
[AR] libskmm_uio.a
[AR] libskmm_cache.a
[CC] process.c      (lib:skmm_process)
[AR] libskmm_process.a
[CC] setup.c        (lib:skmm_dma_mem)
[CC] allocator.c    (lib:skmm_dma_mem)
[CC] map.c          (lib:skmm_dma_mem)
[AR] libskmm_dma_mem.a
[CC] of.c           (lib:skmm_of)
[AR] libskmm_of.a
[CC] pci.c          (lib:skmm_usdpaa_pci)
[AR] libskmm_usdpaa_pci.a
[CC] pci_ep_vfio.c  (lib:skmm_pci_ep_vfio)
[AR] libskmm_pci_ep_vfio.a
[CC] dma_driver.c   (lib:skmm_dma)
[AR] libskmm_dma.a
[CC] trigger_msi_ep2rc.c (bin:trigger_msi_ep2rc)
[LD] trigger_msi_ep2rc
[CC] skmm.c         (bin:skmm)
[CC] abstract_req.c (bin:skmm)
[CC] rsa.c          (bin:skmm)
[CC] dsa.c          (bin:skmm)
[CC] ecdsa.c        (bin:skmm)
[CC] dh.c           (bin:skmm)
[CC] ecdh.c         (bin:skmm)
[LD] skmm
[CC] pcie_dma.c     (bin:pcie_dma)
[LD] pcie_dma
```

Run EP Application

The EP application is named `pcie_dma` and locates `skmm-ep/bin` folder. We can directly run this application on EP side, for example:

```
root@p5020ds:~# ./pcie_dma 0
Initialized pci0-pf0
pcidma>
```

The following commands are provided to perform the PCI DMA performance test.

1. `help`: list the available command

```
pcidma> help
Available commands:
list
rm
add
dump
info
help
```

2. add : create a test thread on the specified PF/VF waiting the RC command to start the performance test .

The command format:

```
add [pf idx] [vf idx]
```

For example:

```
pcidma> add 0 0  
pcidma> Starting a pci0-pf0's test thread on cpul
```

3. rm : remove the test thread

The command format:

```
rm [pf idx] [vf idx]
```

For example:

```
pcidma> rm 0 0  
Leaving pci0-pf0's test thread on cpul
```

4. list : list all the running test thread

The command format:

```
list
```

For example:

```
pcidma> list  
pci0-pf0's test thread is runing on cpu0
```

5. info : display the inbound/outbound/Registers/configuration windows information.

The command format:

```
info [PF index][VF index]
```

For example:

```
pcidma> info 0 0  
PCI EP device pci0-pf0 info:  
type: PF  
  
Outbound windows:  
Win0: cpu_addr:0x0 pci_addr:0x0 size:0x1000000000 attr:0x80044023 vfio_off:0x1000000000  
Win1: cpu_addr:0xc00000000 pci_addr:0x100f1400000 size:0x400000 attr:0x80044015 vfio_off:  
0x11000000000  
Win2: cpu_addr:0xff8000000 pci_addr:0x0 size:0x10000 attr:0x8008800f vfio_off:0x12000000000  
Win3: cpu_addr:0x0 pci_addr:0x0 size:0x0 attr:0x0 vfio_off:0x13000000000  
Win4: cpu_addr:0x0 pci_addr:0x0 size:0x1000000000 attr:0x44023 vfio_off:0x14000000000  
  
Inbound windows:  
Win0: cpu_addr:0xffe000000 pci_addr:0xe0000000 size:0x1000000 attr:0x80e44017 vfio_off:0x0  
Win1: cpu_addr:0xe0000000 pci_addr:0xe1000000 size:0x2000 attr:0xa0f5500c vfio_off:  
0x1000000000  
Win2: cpu_addr:0xe0002000 pci_addr:0xe1002000 size:0x1000 attr:0xa0f5500b vfio_off:  
0x2000000000  
Win3: cpu_addr:0xe0400000 pci_addr:0xe1400000 size:0x400000 attr:0xa0f55015 vfio_off:
```

```
0x3000000000

Registers window:
cpu_addr:0xffe20000 size:0x1000 vfio_off:0x4000000000

PCI configurations window:
cpu_addr:0x0 size:0x0 vfio_off:0x5000000000
pcidma>
```

6. dump : dump the Hexadecimal values of the registers, EP configuration local buffer and remote buffer on the function
The command format:

```
dump [pf idx] [vf idx] [reg config msix local_buffer remote_buffer] [length]
```

For example:

```
pcidma> dump 0 0 local_buffer
dump local_buffer:
00000000: 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5
00000004: 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5
00000008: 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5
0000000c: 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5 0xa5a5a5a5
pcidma> dump 0 0 reg
dump reg:
00000000: 0x80000024 0x00000000 0x00000000 0x0013ffff
00000004: 0x0400ffff 0x00040028 0x00008000 0x00000000
00000008: 0x00000280 0x00000000 0x000cffff 0x00000000
0000000c: 0x00000000 0x00000000 0x00000000 0x00000000
pcidma> dump 0 0 config
dump config:
00000000: 0x00000000 0x00000000 0x00000000 0x00000000
00000004: 0x00000000 0x00000000 0x00000000 0x00000000
00000008: 0x00000000 0x00000000 0x00000000 0x00000000
0000000c: 0x00000000 0x00000000 0x00000000 0x00000000
```

6.2.3 PCI DMA Host Driver Module

Introduction

PCI DMA host driver module is a PCI driver module running in the host side and is used to measure PCI memory write performance. It supports multiple PCI functions and SR-IOV, it can create a thread on the specified function to measure performance using host DMA or memcpy. It also can control EP device to start EP DMA and calculate the performance. The driver also provides some sysfs interface to change test settings such as packet length loop times and write direction (RC to EP or EP to RC).

Host Kernel Image

PCI DMA Host driver module depends on DMA driver and PCI IOV. Please make sure the two drivers are selected when compiling the host kernel image.

```
-> Device Drivers
-> DMA Engine support
[*] Freescale Elo and Elo Plus DMA support (enable the option)
[ ] Network: TCP receive copy offload (disable the option)
```

```
->Bus options  
PCI IOV support
```

[*]

Compile Host Module

1. Use yacto to compile EP application.

The detailed information can refer to SDK documents.

```
#> bitbake fsl-image-core
```

2. Directly Compile host module

- a. Get the code and move them to sdk folder.

```
git clone git://git.am.freescale.net/gitolite/sdk/skmm-host.git  
git checkout remotes/origin/pciedma_multi_chan_perf -b pciedma_multi_chan_perf
```

- b. Compile code

For x86:

```
lmh@lmh:~/work/sdk-devel-gerrit/skmm-host$ make ARCH=x86  
make -C /lib/modules/3.8.0-31-generic/build SUBDIRS=`pwd` modules  
make[1]: Entering directory `/usr/src/linux-headers-3.8.0-31-generic'  
LD [M] /home/lmh/work/sdk-devel-gerrit/skmm-host/"pci_dma_test".o  
Building modules, stage 2.  
MODPOST 1 modules  
LD [M] /home/lmh/work/sdk-devel-gerrit/skmm-host/pci_dma_test.ko  
make[1]: Leaving directory `/usr/src/linux-headers-3.8.0-31-generic'  
cc -Wall perf/mini_calc/mini_calc.c -o mini_calc  
lmh@lmh:~/work/sdk-devel-gerrit/skmm-host$
```

For PowerPC:

```
lmh@lmh:~/work/sdk-devel-gerrit/skmm-host$ make KERNEL_DIR=<path_to_your_linux_kernel>  
CROSS_COMPILE=<name_of_your_toolchain> ARCH=powerpc  
make -C /home/lmh/work/linux SUBDIRS=`pwd` modules  
make[1]: Entering directory `/home/work/linux'  
CC [M] /home/lmh/work/sdk-devel-gerrit/skmm-host/pci_dma_test/pci_dma_dev.o  
CC [M] /home/lmh/work/sdk-devel-gerrit/skmm-host/pci_dma_test/pci_dma_sys.o  
CC [M] /home/lmh/work/sdk-devel-gerrit/skmm-host/pci_dma_test/pci_dma_test.o  
LD [M] /home/lmh/work/sdk-devel-gerrit/skmm-host/"pci_dma_test".o  
Building modules, stage 2.  
MODPOST 1 modules  
WARNING: ".dma_find_channel" [/home/lmh/work/sdk-devel-gerrit/skmm-host/pci_dma_test.ko]  
undefined!  
CC /home/lmh/work/sdk-devel-gerrit/skmm-host/pci_dma_test.mod.o  
LD [M] /home/lmh/work/sdk-devel-gerrit/skmm-host/pci_dma_test.ko  
make[1]: Leaving directory `/home/work/linux'  
cc -Wall perf/mini_calc/mini_calc.c -o mini_calc  
lmh@lmh:~/work/sdk-devel-gerrit/skmm-host$
```

Run Host Module

The host module is named `pci_dma_test.ko`. We can directly insert the module. For example:

```
root@t4240qds:~# insmod pci_dma_test.ko
FSL PCI DMA Test Driver.
root@t4240qds:~#
```

We can use `'num_vfs'` to make the driver enable virtual function. If we set `'num_vfs=4'`, the module will create 4 virtual functions for PF0 and PF1 respectively. For example:

```
root@t4240qds:~# insmod pci_dma_test.ko num_vfs=4
FSL PCI DMA Test Driver.
root@t4240qds:~#
```

The driver module provides the following sysfs interface to configure test settings. We can use `cat` and `echo` to change the related value.

such as packet length loop times and write direction(RC to EP or EP to RC)

1. bars_info: show the device bar information

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat bars_info
PCI DMA BAR0:
cpu_addr:0x0000000c00000000 size:0x000000001000000
PCI DMA BAR1:
cpu_addr:0x0000000c01000000 size:0x000000000002000
PCI DMA BAR2:
cpu_addr:0x0000000c01002000 size:0x0000000000001000
PCI DMA BAR3:
cpu_addr:0x0000000000000000 size:0x0000000000000000
PCI DMA BAR4:
cpu_addr:0x0000000c01400000 size:0x0000000000400000
```

2. config_info : display PCI EP configuration information

For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat config_info
status:0x00000001 commond:0x00000000
rx config: addr:0x100f1000000, size:0x400, loop:0x3e8
root@t4240qds:/sys/class/pcidma/pcidma0#
```

3. test_dma_enable : enable/disable DMA

For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_dma_enable
test dma status: enable
root@t4240qds:/sys/class/pcidma/pcidma0# echo 0 > test_dma_enable
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_dma_enable
test dma status: disable
root@t4240qds:/sys/class/pcidma/pcidma0#
```

4. test_loop : test loop times

For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_loop
test loop: 500
root@t4240qds:/sys/class/pcidma/pcidma0# echo 1000 > test_loop
```

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_loop  
test loop: 1000
```

5. test_lens: test packet length

For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_lens  
test length: 64B 256B 1024B 4096B 1048576B 2097152B  
root@t4240qds:/sys/class/pcidma/pcidma0# echo 1024 > test_lens  
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_lens  
test length: 1024B
```

6. test_rc2ep: test direction RC writes to EP.

For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_rc2ep  
test rc2ep: true  
root@t4240qds:/sys/class/pcidma/pcidma0# echo 0 > test_rc2ep  
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_rc2ep  
test rc2ep: false  
root@t4240qds:/sys/class/pcidma/pcidma0#
```

7. test_ep2rc: test direction EP writes to RC

For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_ep2rc  
test ep2rc: false  
root@t4240qds:/sys/class/pcidma/pcidma0# echo 1 > test_ep2rc  
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_ep2rc  
test ep2rc: true
```

8. test_start: start to test, when finished the results will be printed.

For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# echo 1 > test_start  
test starting  
root@t4240qds:/sys/class/pcidma/pcidma0#  
test info:  
test0 packet length:1024B loop:1000times  
EP->RC throughput:2570Mbps
```

9. test_info: display the current test results.

For example:

```
root@t4240qds:/sys/class/pcidma/pcidma0# cat test_info  
test info:  
test0 packet length:1024B loop:1000times  
EP->RC throughput:2570Mbps  
root@t4240qds:/sys/class/pcidma/pcidma0#
```

6.2.4 Test Procedure

Start EP Application

1. Configure PCI controller in EP mode via changing RCW
2. Start EP board and login using EP kernel image
3. Run PCI DMA EP application to initialize EP device.

```
root@p5020ds:~# ./pciep_dma 0
Initialized pci0-pf0
pcidma>
```

4. Start test thread on the specified PCI functions.

```
pcidma> add 0 0
pcidma> Starting a pci0-pf0's test thread on cpul
```

Start Host driver module

1. Start host computer or board and login
2. Run PCI DMA host module.

```
root@t4240qds:~# insmod pci_dma_test.ko
FSL PCI DMA Test Driver.
FSL-PCIDMA-Driver 0001:01:00.0: BAR:0 addr:0xc2000000 len:0x16777216
FSL-PCIDMA-Driver 0001:01:00.0: BAR:1 addr:0xc2100000 len:0x8192
FSL-PCIDMA-Driver 0001:01:00.0: BAR:2 addr:0xc2100200 len:0x4096
FSL-PCIDMA-Driver 0001:01:00.0: BAR:3 addr:0x0 len:0x0
FSL-PCIDMA-Driver 0001:01:00.0: BAR:4 addr:0xc2140000 len:0x4194304
```

Test performance

1. Test RC to EP DMA performance

```
root@t4240qds:/sys/class# cd pcidma/pcidma0/
root@t4240qds:/sys/class/pcidma/pcidma0# ls
bars_info  config_infodevicesubsystem  test_dma_enable  test_ep2rcctest_info  test_lens
test_loop  test_rc2ep  test_start  uevent

root@t4240qds:/sys/class/pcidma/pcidma0# echo 1 > test_start
test starting
root@t4240qds:/sys/class/pcidma/pcidma0#
test info:
test0 packet length:64B loop:500times
RC->EP throughput:49Mbps
test1 packet length:256B loop:500times
RC->EP throughput:164Mbps
test2 packet length:1024B loop:500times
RC->EP throughput:658Mbps
test3 packet length:4096B loop:500times
RC->EP throughput:1594Mbps
test4 packet length:1048576B loop:500times
RC->EP throughput:11216Mbps
```

```
test5 packet length:2097152B loop:500times  
RC->EP throughput:11337Mbps
```

2. Test RC to EP memcopy performance

```
root@t4240qds:/sys/class/pcidma/pcidma0# echo 0 > test_dma_enable  
root@t4240qds:/sys/class/pcidma/pcidma0# echo 1 > test_start  
test starting  
root@t4240qds:/sys/class/pcidma/pcidma0#  
test info:  
test0 packet length:64B loop:500times  
RC->EP throughput:767Mbps  
test1 packet length:256B loop:500times  
RC->EP throughput:720Mbps  
test2 packet length:1024B loop:500times  
RC->EP throughput:767Mbps  
test3 packet length:4096B loop:500times  
RC->EP throughput:763Mbps  
test4 packet length:1048576B loop:500times  
RC->EP throughput:762Mbps  
test5 packet length:2097152B loop:500times  
RC->EP throughput:372Mbps
```

3. Test EP to RC DMA performance

```
root@t4240qds:/sys/class/pcidma/pcidma0# echo 1 > test_ep2rc  
root@t4240qds:/sys/class/pcidma/pcidma0# echo 0 > test_rc2ep  
root@t4240qds:/sys/class/pcidma/pcidma0# echo 1 > test_start  
test starting  
root@t4240qds:/sys/class/pcidma/pcidma0#  
test info:  
test0 packet length:64B loop:500times  
EP->RC throughput:353Mbps  
test1 packet length:256B loop:500times  
EP->RC throughput:1270Mbps  
test2 packet length:1024B loop:500times  
EP->RC throughput:3893Mbps  
test3 packet length:4096B loop:500times  
EP->RC throughput:7710Mbps  
test4 packet length:1048576B loop:500times  
EP->RC throughput:12148Mbps  
test5 packet length:2097152B loop:500times  
EP->RC throughput:12162Mbps
```

Note: PCI DMA host module does not support KVM.

Chapter 7

Linux User Space

7.1 Linux HugeTLBFS User Guide

7.1.1 Introduction

Instructions for using large pages for networking application performance improvement with Linux HugeTLBFS.

It is becoming customary for user-level applications to have large memory requirements in embedded systems. Even so, the default page size in Linux is 4KB while the amount of memory mappable in TLB0 with 4KB pages is small. Small TLBs may result in many TLB misses for large applications. This document illustrates, with examples, how to properly configure a QorIQ device for large page sizes.

7.1.2 Objectives

These are results expected for the method outlined:

- Baseline understanding of Linux HugeTLBFS.
- Identify any optimizations that have been discovered and ensure they are implemented.
- Investigate other changes that may improve performance.
- Compare these results with end product performance objectives.

7.1.3 TLBFS Basics

The amount of memory mappable in TLB0 with 4KB pages is small:

Table 85. Mappable Memory of e500 Versions

Processor	No. TLB Entries	Mappable by TLB0
e500 v1	256	1MB
<ul style="list-style-type: none"> • e500 v2 • e500MC • e5500 	512	2MB

7.1.3.1 4KB TLB0 Miss Issues

The salient contributors to TLB0 miss issues:

- TLB0 is shared between multiple processes.
- TLB0 is used for mapping both instructions and data.
- Thrashing - A single MMU-hogging process in the system can have performance impact on smaller processes running in the system.
- A single process may use significantly more than 2M.

For the case of conflict miss, TLB0 is 4-way set associative (2-way on e500v1), which has a higher potential for conflict miss than a fully-associative TLB cache. Another concern is latency issues caused by e500 reloads of TLB entries using software which has a latency of at least 100-200 cycles per TLB.

7.1.3.2 e500 Family TLB1

The e500 processor family provides a second software-visible TLB structure, the TLB1. This is sometimes referred to as the TLBCAM. Because the TLB1 is fully associative, new mappings may be placed in any entry.

TLB1 supports a number of large page sizes but has a small number of entries:

Table 86. e500 Family TLB1 Characteristics

Processor	TLB1 Entries	Page Size
e500 v1	16	4KB, 16KB, 64KB, 256KB, 1MB, 4MB, 16MB, 64MB, 256MB
e500 v2	16	Same as e500 v1, also, 1GB, 4GB
e500MC, e5500	64	Same as e500 v2

7.1.3.3 HugeTLB Basics

Hugetlbfs allows the Linux kernel to use large pages mapped by TLB1 for user processes:

- The kernel itself does not use “hugetlbfs”; it uses a few large TLB1 entries to map some amount of memory using large mappings
- Drivers that run as part of the kernel cannot use hugetlbfs (may be a short-term restriction)

The kernel will not allocate hugepages unless requested:

- This is intentional - the number of TLB1 entries is limited so they should be used with care.
- Hugepages can cause performance problems if misused.

Hugetlb for BookE supports 4MB, 16MB, 64MB, 256MB, and 1GB pages:

- This is a subset of the hardware page sizes supported by TLB1.
- We cannot easily support a page size less than 4MB due to the current page table architecture of the 32-bit Linux kernel.

Hugepages can be used for:

- program
- .text
- .data
- .bss
- `malloc()`
- `mmap()`
- `shmget()`

Using hugepages may or may not require application modification and/or rebuild.

7.1.3.4 Normal vs. Gigantic hugepages

Page sizes larger than 16MB are called **Gigantic** pages. This depends on `CONFIG_FORCE_MAX_ZONEORDER` in Linux and may vary on different kernels; this value is for the FSL BSP. There are differences in how normal hugepages and gigantic hugepages are allocated in the kernel:

Table 87. (Normal) Huge vs. Gigantic Pages

Type	Sizes	Allocation	Deallocation
Normal	4MB, 16MB	At boot, or while system is running	Allowed
Gigantic	64MB, 256MB, 1GB	Only at boot time	Not allowed

The table below lists several limiting cases for Gigantic pages:

Table 88. Gigantic Page Limitations

Limiting Case	Description
Gigantic pages should be allocated with care since they cannot be deallocated.	Has the effect of removing RAM from the system if the hugepages are not used.
Virtual and physical addresses in the TLB must be aligned to the huge page size requested.	<ul style="list-style-type: none"> • May not be possible for the kernel to allocate many gigantic pages because of alignment requirements • example: The user's virtual address space is limited to 3GB total on 32-bit Linux, so at most one or two 1GB pages and eight or so 256MB pages may be used by a single process

7.1.4 Building and Booting Linux with Hugetlb

Linux must be built with `CONFIG_HUGETLBFS` and `CONFIG_HUGETLB_PAGE` set; `CONFIG_FORCE_MAX_ZONEORDER` should be set to 13.

- Upcoming FSL BSPs have these options enabled
- The publicly available Linux trees do not yet contain the Hugetlb support; this update is pending publication.

Specify the default hugepage size to allocate hugepages via the kernel boot command line:

- `default_hugepagesz=16m hugepagesz=16m hugepages=25 hugepagesz=4m hugepages=25`
- Once allocated, hugepages are removed from the available memory pool and will only be used for huge mappings. Avoid allocating hugepages that you don't expect to use.

7.1.4.1 Booting Linux for HugeTLB

Once Linux is booted, follow these steps:

Table 89. Step 1: The hugetlbfs filesystem must be mounted by root.

Description	Action
Check that hugetlb is enabled:	<ul style="list-style-type: none"> • <code>grep hugetlbfs /proc/filesystems</code> • <code>grep HugePages_Total /proc/meminfo</code> (look for a non-zero number)
Mount the default huge page size:	<code>mount -t hugetlbfs none /mnt/hugetlbfs</code>
Mount a secondary page size:	<code>mount -t hugetlbfs none -opagesize=16m /mnt/hugetlbfs-16M</code>
To allow non-root to use hugepages:	mount point perms = 1777; or use <code>hugeadm</code> to create a user or group-specific mount point

Step 2: Build and install libhugetlbfs

- simple `make; make install`
- Will be included in the BSP.
- Specific version from NXP required.

Step 3: Use `hugeadm` utility to query/change huge page parameters:

- Change the number of non-gigantic huge pages available.
- Query the size of the current huge page pool.
- View supported page sizes.
- Create hugepage mount points.

Step 4: Use `HUGETLB_DEBUG=yes` in environment to help debug applications linked with the library when there are issues

7.1.4.2 Setting Up Mount Points with hugeadm

4 commands are provided for creating mount points:

Table 90. Rev. History Table

Command Line	Description
<code>--create-mounts</code>	Creates a mount point for each available huge page size on this system under <code>/var/lib/hugetlbfs</code>
<code>--create-user-mounts [user]</code>	Creates a mount point for each available hugepage size under <code>/var/lib/hugetlbfs/[user]</code> usable by <code>[user] [user]</code>
<code>--create-group-mounts [group]</code>	Creates a mount point for each available huge page size under <code>/var/lib/hugetlbfs/[group]</code> usable by group <code>[group]</code>
<code>--create-global-mounts</code>	Creates a mount point for each available huge page size under <code>/var/lib/hugetlbfs/global</code> usable by anyone

7.1.4.3 Running Hugepage mount/query Example

Each result of each action in the example is shown in the tables

```
-bash-3.2# mount -t hugetlbfs none /mnt/hugetlbfs
```

```
-bash-3.2# hugeadm --pool-list
```

Table 91. Pool List

Size	Minimum	Size	Minimum	Default
4194304	25	25	25	
16777216	25	25	25	*
67108864	0	0	0	
268435456	0	0	0	
1073741824	0	0	0	

```
-bash-3.2# hugeadm -pool-pages-min 4M:100
```

```
-bash-3.2# hugeadm --pool-list
```

Table 92. Pool List

Size	Minimum	Size	Minimum	Default
4194304	100	100	100	
16777216	25	25	25	*
67108864	0	0	0	
....				

7.1.5 Hugepage Allocation in User Programs

User applications can request hugepages several ways:

Table 93. Pool List

Method	Steps
Shared memory	<ul style="list-style-type: none"> • <code>shmget()</code> call specifies <code>MAP_HUGETLB</code> • Link application with <code>libhugetlbfs</code> to override all <code>shmget()</code> calls
Heap-based allocations	Causes all <code>malloc()</code> calls (and C++ <code>new</code> operator) to allocate in a huge page heap region.

Table continues on the next page...

Table 93. Pool List (continued)

Method	Steps
<code>mmap()</code>	<ul style="list-style-type: none"> Anonymous <code>mmap()</code> (<code>MAP_ANONYMOUS</code>) with <code>MAP_HUGETLB</code> <code>mmap()</code> <code>mmap()</code> file on the <code>hugetlbfs</code> mount point.
Back program segments with hugepages	<code>.text</code> , <code>.data</code> , and <code>.bss</code> segments

7.1.5.1 Using Shared Memory Hugepages: SHM_HUGETLB

Specify `SHM_HUGETLB` in `shmget()` flags in application code

- Requires application modification to add `SHM_HUGETLB` to `shmget()` arguments.
- Application must be rebuilt.
- Libhugetlbfs is not required to use this method (no special linking needed).
- Can specify exactly which `shmget()` calls use hugepages.
- Can only use the systemwide default hugepage size.
- The size requested must be a multiple of the huge page size.

7.1.5.2 Running the SHM_HUGETLB Example

The code below illustrates several of the concepts discussed so far:

```
#define LENGTH 4 * 1024 * 1024

/* Create shared memory region */
if ((shmid = shmget(2, LENGTH, SHM_HUGETLB | IPC_CREAT | SHM_R | SHM_W)) < 0)
    exit(1);

    shmaddr = shmat(shmid, 0, 0); /* Attach the region to the process */
if (shmaddr == (char *)-1) {
    shmctl(shmid, IPC_RMID, NULL);
    exit(2);
}

for (i = 0; i < LENGTH; i++)          /* Example write to the region */
    shmaddr[i] = (char)(i);

if (shmdt((const void *)shmaddr) != 0) { /* Detach the region */
    shmctl(shmid, IPC_RMID, NULL);
    exit(3);
}
shmctl(shmid, IPC_RMID, NULL); /* Remove the shmid */
```

7.1.5.3 Using Shared Memory Hugepages: Link with libhugetlbfs

Instead of specifying `SHM_HUGETLB`, link application with `libhugetlbfs` to override all `shmget()` calls

- No application modification required.
- With dynamic linking, can use an existing binary.

- Can only use the systemwide default hugepage size.
- Simply invoke the application as follows (dynamic link):
 - `LD_PRELOAD=libhugetlbfs.so HUGETLB_SHM=yes [your app command line]`
- These variables can be set in the user's shell, but they will then apply to all binaries executed in this shell, which is usually not desired.
- The size requested will automatically be rounded up to the huge page size.

7.1.6 Manipulating Shared Memory System Tunables

Hugepage `shmget()` requests can fail due to limits placed on process shared memory by 4 system tunables:

- **shmmax**: Limit on shared memory size for a single process, in bytes.
- **shmall**: System-wide limit on total shared memory "pages"; in this context a "page" is 4k.
- **shmmin**: Minimum size of a shared memory segment.
- **shmmni**: System-wide maximum number of shared memory segments.

Low `shmmax` usually the cause of `shmget()` failure. It should be adjusted to the max size required by a single process on the system. For example, to allow any process to have 256MB of shared memory:

- `echo 268435456 > /proc/sys/kernel/shmmax`

`shmall` will rarely require change; the others need modification extremely rarely.

7.1.6.1 Allocating hugepages for Heap Memory

Link application with `libhugetlbfs` to use hugepages for all `malloc()` calls and the C++ `new` operator

- Requires no application modification.
- With dynamic linking, can use an existing binary.
- Any available and mounted hugepage size can be used via the `HUGETLB_MORECORE` environment variable.

All `malloc()` calls and the C++ `new` operator will use hugepages provided by kernel.

- C library `malloc()` code is the same as without `libhugepages`; the underlying interface to kernel (known as `morecore()`) is what is changed by `libhugetlbfs`
- Total amount of hugepages is limited by system configuration
- `malloc()` or `new` requests smaller than the hugepage size will be rounded up by the kernel; multiple small requests can end up being mapped by a single hugepage.

7.1.6.2 Invoking Applications with Huge Heap (`malloc()/new`)

To use the default hugepage size:

- `LD_PRELOAD=libhugetlbfs.so`
- `HUGETLB_MORECORE=yes [your app command line]`

To use a different hugepage size:

- `LD_PRELOAD=libhugetlbfs.so`
- `HUGETLB_MORECORE=16M [your app command line]`

If the `libhugetlbfs` library is not installed in the normal system library directory, use `LD_LIBRARY_PATH` to specify the location::

- `LD_PRELOAD=libhugetlbfs.so`
- `LD_LIBRARY_PATH=~/.mylibdir`

- `HUGETLB MORECORE=yes` [your app command line]

7.1.6.3 Moving the Heap for Hugepage Allocations

By default, the heap is located after the program image and can continue until it runs into a mmap. The program image typically resides at `text/data/bss`. Addresses for `mmap` are normally chosen by the kernel, starting from `TASK_UNMAPPED_BASE`. The result is a large portion of address space, sometimes larger than the default heap, unused. The `HUGETLB_MORECORE_HEAPBASE` option can be used to start the heap after the `mmap` area, rather than before it, so as to provide more space for the heap.

There may not be enough address space at the default heap location for hugepage `malloc()` and new requests to succeed.

- The 32-bit kernel uses `0x48000000` as `TASK_UNMAPPED_BASE`, which causes failures when trying to map > about 832MB of memory

The library provides `HUGETLB_MORECORE_HEAPBASE` to allow the user to specify the heap location.

- Use `pmmap` or set `HUGETLB_DEBUG=yes` to get more information on the process' heap allocations
- Find a hole in the map large enough for your allocation and set `HUGETLB_MORECORE_HEAPBASE` to this when you run.

Example:

- `LD_PRELOAD=libhugetlbf.so HUGETLB_MORECORE=yes HUGETLB_MORECORE_HEAPBASE=0x4C000000` [your app command line]

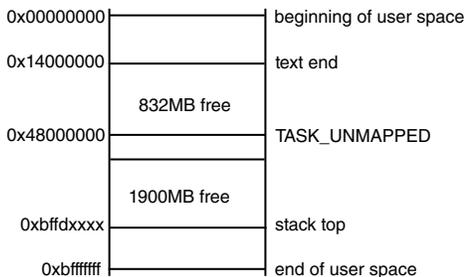


Figure 112. Heap Example:

7.1.7 Using Hugepages for .text, .data, and/or .bss (gnu ld 2.17+)

Specify `-hugetlbf-align` option to GNU ld.

- Requires no application modification.
- Application needs to be relinked.
- `libhugetlbf` wrapper script and linker script must be installed.
- Quite flexible – can specify exactly which segment types should use hugepages.
- Causes segments to be properly aligned for hugepage use.

Hugeedit utility

- Change the default segment-specific behavior without relinking the program.
- Default behavior is no hugepage mapping.
- The settings are used when `HUGETLB_ELFMAP` is not specified.

Run application with `HUGETLB_ELFMAP`.

Table 94. Environment variable that controls which segments use hugepages

Variable	Description
HUGETLB_ELFMAP=R	Read-only segments (text)
HUGETLB_ELFMAP=W	Writable segments (data/BSS)
HUGETLB_ELFMAP=RW	All segments (text/data/BSS)
HUGETLB_ELFMAP=no	No segments

The HUGETLB_ELFMAP application can specify specific page sizes for the different segment types:

- HUGETLB_ELFMAP=[R[=[pagesize]]]:[W[=[pagesize]]]

Example: program invocation (dynamic link):

- LD_PRELOAD=libhugetlbf.so HUGETLB_ELFMAP=R=4M:W=16M [your app command line]

If you are **not** using HUGETLB_ELFMAP:

- The default settings for the binary (set via hugeedit) will be used.
- The system-wide hugepage size will be used.

7.1.8 Performing mmap on a Hugepage-backed File

The option exists for an alternative to hugetlb-backed memory. This option is appropriate for these cases:

- To have some things hugetlb-backed without making the heap huge-tlb backed.
- To Share huge-tlb backed memory.
- To make use of address space above TASK_UNMAPPED_BASE without moving heap and thereby losing use of the address space above TASK_UNMAPPED_BASE.

Create and mmap() files on the hugetlbf mount point.

- Can use any mounted huge page size.
- The huge page size will be the page size supported by the mount point.
- Files on the hugetlb file system are not regular files – they represent memory. You cannot copy a regular file onto the hugetlb file system mount point.
- Do not specify MAP_HUGETLB for this type of mapping – files on the hugetlb mount point are automatically backed by hugepage memory
- Files on a hugetlb mount point should be created as a multiple of the huge page size for that particular mount point; otherwise mmap() fails.

7.1.8.1 Exercising the mmap() File Example

The following example demonstrates a 4M hugetlbf mount point.

```
#define LENGTH 4*1024*1024
#define FILE_NAME "/var/lib/hugetlbf/global/pagesize-4MB/my_hugefile"

/* Open/create the file on the 4m mountpoint */
fd = open(FILE_NAME, O_CREAT | O_RDWR, 1777);
if (fd < 0) exit(1);
```

```
/* map in the file and check that the mmap succeeded */
addr = mmap(NULL, LENGTH, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (addr == MAP_FAILED) {
    perror("mmap");
    unlink(FILE_NAME);
    exit(1);
}
/* unmap file, check for error */
if (munmap(addr, LENGTH) < 0) {
    perror("munmap");
    exit(2);
}
close(fd);
unlink(FILE_NAME);
```

7.1.8.2 Requesting Anonymous mmap() with Hugepages

The user can directly request hugepages with MAP_HUGETLB flag to anonymous (no file descriptor specified) mmap

- Libhugetlbfs is not required.
- This method can only use the default hugepage size
- No mount points required.

Example:

```
mmap(NULL, 4*1024*1024, PROT_READ | PROT_WRITE,
      MAP_ANONYMOUS | MAP_HUGETLB | MAP_SHARED, 0, 0);
```

NOTE

MAP_ANONYMOUS and MAP_HUGETLB are both required

7.1.8.3 Using HugeTLB mmap(): Arguments and Error Checking

If specified, the address argument passed to mmap() must be aligned to the huge page size.

- Address may be NULL; the system will pick one for you.

For non-anonymous mmap() calls, the “offset” should be a multiple of the huge page size, if specified as non-zero.

The length of a mmap() should be a multiple of the huge page size.

- mmap() calls that use a fd (i.e. non-anonymous) must follow this rule.
- This is not required for anonymous mmmaps, but is recommended to avoid munmap() problems.

Always check mmap() return codes.

7.1.8.4 Using HugeTLB munmap(): Arguments and Error Checking

Exercise caution with this process step, munmap() calls must specify a length that is a multiple of the huge page size.

- This is not optional.
- The munmap() will fail if this rule is broken.

Programs should always check the return code of munmap().

- It is common coding practice to check the return code from mmap() but not from munmap(). This is incorrect. Check the return code of both mmap() and munmap() to properly catch errors.

7.1.9 Running the Test

After two boards boot up, login linux and run following commands:

Table 95. sRIO Test Procedure

Memory Allocation Type	Code Mod	Reuse Binary	Huge Page Size	Need Lib?	Notes
heap (malloc()/new)	No	Yes*	Any>	Yes	* If dynamically linked.
anonymous (mmap()/new)	Yes	No	Default Only>	No	
file mmap()	Yes	No	Any*>	Yes	* Page size used is that supported by the mount
shared memory (shmget() with SHM_HUGETLB)	Yes	No	Default Only	No	
shared memory (shmget() without SHM_HUGETLB)	No	Yes*	Default Only	Yes	*If dynamically linked.
.text, .data, .bss	No	No	Any	Yes	

7.1.9.1 When to Use Hugepages

In order to calculate whether a program will benefit from using huge pages, count the L2 MMU miss rate using core performance monitors.

- If (# L2 MMU misses * 200 cycles) represents a large percentage of the program's total number of cycles, it is possibly a good candidate for hugepages

Use “top” or “ps” on a running process to determine its resource usage.

- For a long-running process, `/usr/bin/time -v` can be used to get the “Maximum resident set size,” but it can return values that are 4x the actual usage.

Look at pattern of mmap()calls in a program trace.

- Large calls lend themselves more easily to using hugepages. Programs with lots of small calls can also benefit from code change to consolidate the calls into a multiple of the huge page size.
- Also look at munmap() calls – the program must be able to munmap() in multiples of the huge page size.

Generate an instruction trace of the program and look at load/store addresses

- If the top 20 bits of load/store addresses change often and have more than 512 distinct values, then the program may be using a lot of data and bss.
- Also see if the same is true for instruction addresses

Programs with malloc() or new requests whose sum is large may benefit.

- When the library sees one of these requests it will allocate a large heap region that is the size of the hugepage specified by the HUGETLB_MORECORE environment variable.
- Future malloc() or new requests will allocate from that same region until it is fully utilized.
- The smallest hugepage that encompasses all the requests should be used when possible.

Smaller heap requests can also benefit.

- If the number of hugepages needed in the system is smaller than the number of entries in TLB1, the application may see fewer TLB misses with a hugepage.

7.1.10 Matching Hugetlb Methods with Program Characteristics

Table 96. sRIO Protocol Efficiency

Program Characteristic	HugeTLB usage
Large malloc() or new requests.	HUGETLB_MORECORE = [hugepage size]
Large program segments (.text, hugetlbfs align .data, .bss)	--hugetlbfs-align, HUGETLB_ELFMAP=[specify segments/size]
Large MAP_ANONYMOUS mmap()	mmap() on hugetlb mount point.
Large shared memory	SHM_HUGETLB or link with library and specify HUGETLB_SHM.

7.1.11 Using Multiple Types of Hugepage Allocation Methods

HugeTLB allocation methods are not mutually exclusive.

- Example: HUGETLB_MORECORE + HUGETLB_ELFMAP
- Example: shmget(SHM_HUGETLB) + HUGETLB_MORECORE

All combinations are allowed; but care should be taken.

- Poor choice of options can result in less-than optimal hugepage allocations.
- Can run out of hugepages in the system.
- TLB1 can end up oversubscribed.
- Can waste process virtual address space and have failed program allocations.

7.1.12 Hugetlb Benefits and Limitations

Benefits:

- Fewer TLB misses in large processes.
- Other processes can benefit because their 4k pages aren't getting evicted by memory hog processes.
- Potential for significant performance gains.

Limitations:

- Limited number of TLB1 entries.
- Adds minor latency to critical kernel paths for 4k pages.
- Hugepages are more costly for the kernel to manage.
- Allocating hugepages removes them from the available memory pool, causing less memory to be available.

7.1.13 Understand the Differences Between Applications

Some applications convert easily to hugepages. In many cases, only slight modifications, or none at all, are required. Other applications may benefit, but with more work:

- Some huge page usage models can only operate on multiples of a huge page size.
- Applications that map and unmap large amounts of memory using a lot of small `mmap()` and/or `munmap()` calls will likely require significant modifications in order to use hugepages efficiently.
- Code that doesn't properly check system call return codes may have unexpected fails later in the program execution.

There are programs that will not benefit from huge pages. Either the memory footprint is too small or it may be the case that a low percentage of cycles are spent in TLB miss handling.

7.1.14 HugeTLB Status and Support

The current status of HugeTLB features and future goals for this effort includes these items:

- Some huge page usage models can only operate on multiples of a huge page size.
- Applications that map and unmap large amounts of memory using a lot of small `mmap()` and/or `munmap()` calls will likely require significant modifications in order to use hugepages efficiently.
- Code that doesn't properly check system call return codes may have unexpected fails later in the program execution.

There are programs that will not benefit from huge pages. Either the memory footprint is too small or it may be the case that a low percentage of cycles are spent in TLB miss handling.

7.1.15 HugeTLB Resources

4-part LWN.net series on HugeTLB

Running Applications with Libhugetlbf

Libhugetlbf HOWTO:

NOTE

These supporting documents don't have specifics for FSL Power Architecture such as page size but are included for general understanding.

7.2 OpenSSL Offload User's Guide

7.2.1 Overview

The Secure Socket Layer (SSL) protocol is the most widely deployed application protocol to protect data during transmission by encrypting the data using popular cipher algorithms such as AES, DES and 3DES.

Apart from encryption it also provides message authentication services using popular hash/digest algorithms such as SHA1 and MD5. SSL is widely used in application web servers (HTTP) and other applications such as SMTP POP3, IMAP, Proxy servers etc., where protection of data in transit is essential for data integrity

There are various version of SSL protocol such as TLSv1, SSLv3 that are commonly used. Other newer versions are available, such as TLSv2, TLSv3 and DTLS (Datagram TLS). Of all the SSL protocol versions, TLSv1 and SSLv3 are in common use.

This document introduces NXP SSL acceleration solution on QorIQ platforms using OpenSSL:

1. OpenSSL software architecture
2. Building OpenSSL with hardware offload support
3. Examples of OpenSSL Offloading

7.2.1.1 OpenSSL Software architecture

The SSL protocol is implemented as a library in OpenSSL - the most popular library distribution in Linux and BSD systems. The OpenSSL library has several sub-components such as:

1. SSL protocol library
2. Crypto library (Symmetric and Asymmetric cipher support, digest support etc.)
3. Certificate Management

The following figure presents the general interconnect architecture for OpenSSL. Each relevant layer is represented with a clear separation between Linux User Space and Linux Kernel Space.

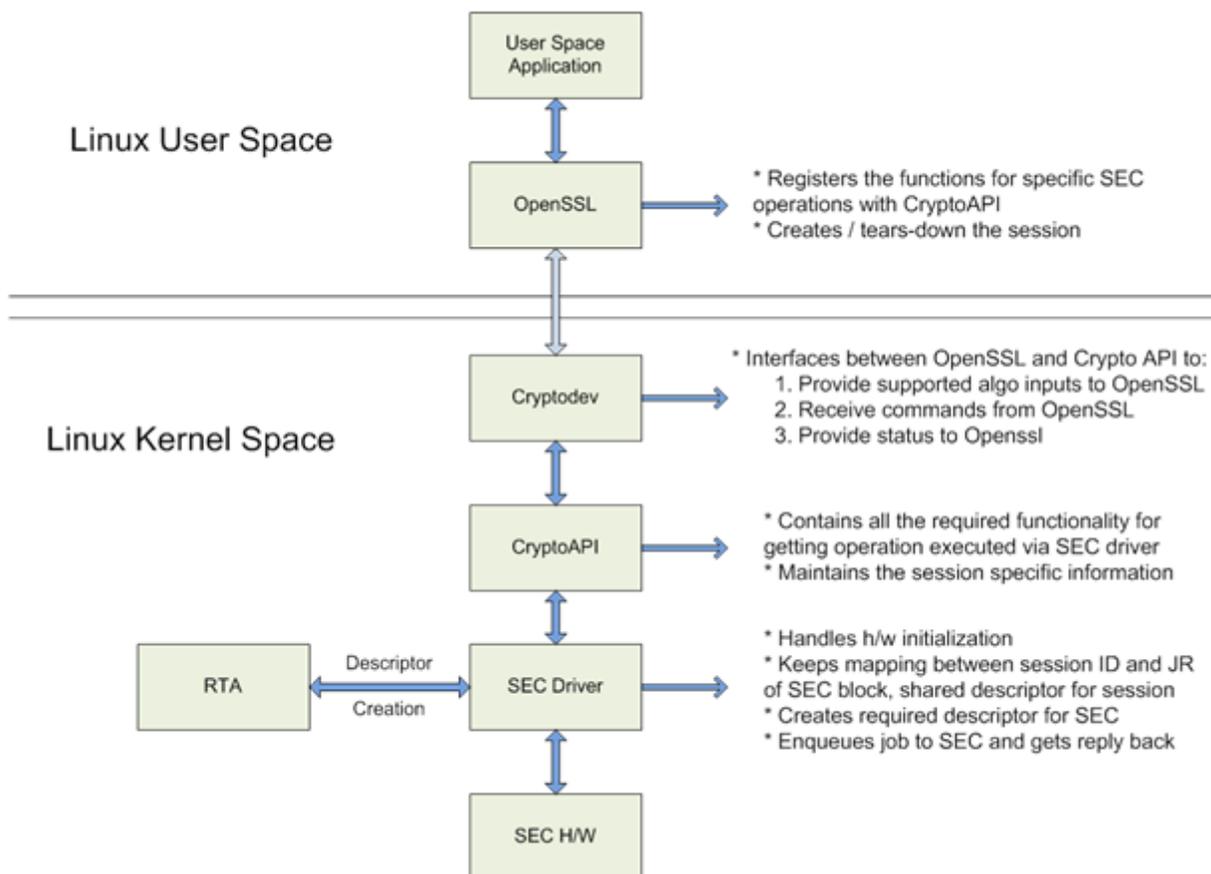


Figure 113. OpenSSL interface with Linux kernel

7.2.1.1 OpenSSL's ENGINE Interface

OpenSSL Crypto library provides Symmetric and Asymmetric (PKI) cipher support that is used in a wide variety of applications such as OpenSSH, OpenVPN, PGP, IKE, XML-SEC etc. The OpenSSL Crypto library provides software support for:

1. Cipher algorithms
2. Digest algorithms
3. Random number generation
4. Public Key Infrastructure

Apart from the software support, the OpenSSL can offload these functions to hardware accelerators via the ENGINE interface. The ENGINE interface provides callback hooks that integrate hardware accelerators with the crypto library. The callback hooks provide the glue logic to interface with the hardware accelerators. Generic offloading of cipher and digests algorithms through Linux kernel is possible with cryptodev engine.

7.2.1.1.2 NXP solution for OpenSSL hardware offloading

The following layers can be observed in NXP's solution for OpenSSL hardware offloading:

- OpenSSL (user space) - implements the SSL protocol
- cryptodev-engine (user space) - implements the OpenSSL ENGINE interface; talks to cryptodev-linux (/dev/crypto) via ioctl, offloading cryptographic operations in kernel
- cryptodev-linux (kernel space) - Linux module that translates ioctl requests from cryptodev-engine into calls to Linux Crypto API
- Linux Crypto API (kernel space) - Linux kernel crypto abstraction layer
- CAAM driver (kernel space) - Linux device driver for the CAAM (Cryptographic Acceleration and Assurance Module) crypto engine

The following are offloaded in hardware in current SDK:

- protocols: TLS v1.0
- cipher modes:
 - one-shot (a single ioctl for both encryption and authentication): AES128-SHA, AES256-SHA
 - two-shot (two ioctls - one for encryption, the other for authentication): all combinations of AES with SHA1, all combinations of DES and 3DES with SHA1

7.2.2 Valid TLS Ciphersuites based on TLS protocol version

In order to avoid compatibility issues cipher suite vs protocol version, the following list (extracted from OpenSSL) represent the compatibility list.

Table 97. OpenSSL CipherSuite Compatibility

CipherSuite	TLS Protocol Version
SSL_RSA_WITH_NULL_MD5	SSL3.0
SSL_RSA_WITH_NULL_SHA	SSL3.0
SSL_RSA_EXPORT_WITH_RC4_40_MD5	SSL3.0
SSL_RSA_WITH_RC4_128_MD5	SSL3.0
<i>Table continues on the next page...</i>	

Table 97. OpenSSL CipherSuite Compatibility (continued)

CipherSuite	TLS Protocol Version
SSL_RSA_WITH_RC4_128_SHA	SSL3.0
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5	SSL3.0
SSL_RSA_WITH_IDEA_CBC_SHA	SSL3.0
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_RSA_WITH_DES_CBC_SHA	SSL3.0
SSL_RSA_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_DH_DSS_WITH_DES_CBC_SHA	SSL3.0
SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_DH_RSA_WITH_DES_CBC_SHA	SSL3.0
SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_DHE_DSS_WITH_DES_CBC_SHA	SSL3.0
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_DHE_RSA_WITH_DES_CBC_SHA	SSL3.0
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5	SSL3.0
SSL_DH_anon_WITH_RC4_128_MD5	SSL3.0
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_DH_anon_WITH_DES_CBC_SHA	SSL3.0
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_FORTEZZA_KEA_WITH_NULL_SHA	SSL3.0
SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA	SSL3.0
SSL_FORTEZZA_KEA_WITH_RC4_128_SHA	SSL3.0
TLS_RSA_WITH_NULL_MD5	TLS1.0
TLS_RSA_WITH_NULL_SHA	TLS1.0
TLS_RSA_EXPORT_WITH_RC4_40_MD5	TLS1.0
TLS_RSA_WITH_RC4_128_MD5	TLS1.0
TLS_RSA_WITH_RC4_128_SHA	TLS1.0
<i>Table continues on the next page...</i>	

Table 97. OpenSSL CipherSuite Compatibility (continued)

CipherSuite	TLS Protocol Version
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5	TLS1.0
TLS_RSA_WITH_IDEA_CBC_SHA	TLS1.0
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_RSA_WITH_DES_CBC_SHA	TLS1.0
TLS_RSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_DES_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_DES_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_DES_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_DES_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_DH_anon_EXPORT_WITH_RC4_40_MD5	TLS1.0
TLS_DH_anon_WITH_RC4_128_MD5	TLS1.0
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_DES_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_RSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_RSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_AES_128_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_AES_256_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_AES_256_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	TLS1.0

Table continues on the next page...

Table 97. OpenSSL CipherSuite Compatibility (continued)

CipherSuite	TLS Protocol Version
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_AES_128_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_AES_256_CBC_SHA	TLS1.0
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_RSA_WITH_SEED_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_SEED_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_SEED_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_SEED_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_SEED_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_SEED_CBC_SHA	TLS1.0
TLS_ECDH_RSA_WITH_NULL_SHA	TLS1.0
TLS_ECDH_RSA_WITH_RC4_128_SHA	TLS1.0
TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_ECDH_ECDSA_WITH_NULL_SHA	TLS1.0
TLS_ECDH_ECDSA_WITH_RC4_128_SHA	TLS1.0
TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA	TLS1.0

Table continues on the next page...

Table 97. OpenSSL CipherSuite Compatibility (continued)

CipherSuite	TLS Protocol Version
TLS_ECDHE_RSA_WITH_NULL_SHA	TLS1.0
TLS_ECDHE_RSA_WITH_RC4_128_SHA	TLS1.0
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_ECDHE_ECDSA_WITH_NULL_SHA	TLS1.0
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA	TLS1.0
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_ECDH_anon_WITH_NULL_SHA	TLS1.0
TLS_ECDH_anon_WITH_RC4_128_SHA	TLS1.0
TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_ECDH_anon_WITH_AES_128_CBC_SHA	TLS1.0
TLS_ECDH_anon_WITH_AES_256_CBC_SHA	TLS1.0
TLS_RSA_WITH_NULL_SHA256	TLS1.2
TLS_RSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_RSA_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_RSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_RSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_DH_RSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_DH_RSA_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_DH_RSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_DH_RSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_DH_DSS_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_DH_DSS_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_DH_DSS_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_DH_DSS_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	TLS1.2

Table continues on the next page...

Table 97. OpenSSL CipherSuite Compatibility (continued)

CipherSuite	TLS Protocol Version
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_DHE_DSS_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_DHE_DSS_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384	TLS1.2
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384	TLS1.2
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	TLS1.2
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	TLS1.2
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_DH_anon_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_DH_anon_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_DH_anon_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_DH_anon_WITH_AES_256_GCM_SHA384	TLS1.2

7.3 USDPAA

7.3.1 USDPAA User Guide

7.3.1.1 Introduction

The NXP Data Path Acceleration Architecture comprises a set of hardware components which are integrated via a hardware queue manager and use a common hardware buffer manager. Software accesses the DPAA via hardware components called "software portals". These directly provide queue and buffer manager operations such as enqueues, dequeues, buffer allocations, and buffer releases and indirectly provide access to all of the other DPAA hardware components via the queue manager.

This document describes the User Space Datapath Acceleration Architecture (USDPA) software. USDPA is a software framework that permits Linux user space applications to directly access the DPAA queue and buffer manager software portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

7.3.1.1.1 Intended audience

This document is intended for software developers and system architects who work with the USDPA framework.

The material is technical in nature. The reader is assumed to be familiar with

- General Linux software development, operation, and configuration-- for Power architecture devices in particular.
- The concept of device drivers and their role in operating systems.
- Linux network administration and use.
- The NXP Linux SDK for DPAA-based SoCs.
- At least broadly, the DPAA hardware blocks.

The P4080 was the first NXP SoC to incorporate the DPAA. As such, it is used in many examples. However, USDPA is intended to apply in the same manner to all DPAA-based SoCs.

7.3.1.1.2 USDPA overview

As mentioned above, USDPA is an environment that allows the development of Linux user space applications that have direct access to software portals for the DPAA buffer manager (BMan) and the DPAA queue manager (QMan).

Software portals are memory mapped hardware components. The items within them (message ring, dequeue ring, etc.) have specific addresses in the SoC's physical address map just like regular memory does. USDPA works by permitting the physical address ranges that correspond to software portals to be mapped into the virtual (technically "effective" in the Power architecture nomenclature) address space of user space processes. Thus, the user space application can use normal load and store instructions to perform operations on the portals.

Portals must be accessed using correct instruction sequences, and also the memory range for the portals must be mapped using the correct cache attributes. For this reason, the USDPA software includes both the user space driver for the portals and also an access and control API packaged as a user space library.

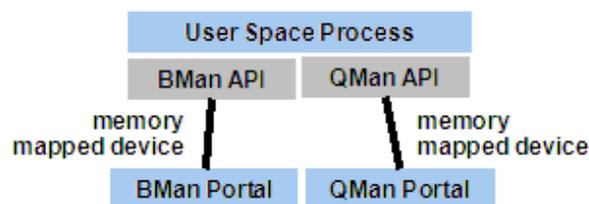


Figure 114. C-Language APIs access memory-mapped devices

The USDPA framework assumes the context of a single SMP Linux instance on an SoC such as the P4080 . There is no dependence on the NXP Embedded Hypervisor, and this document assumes that it is not used.

Specifically, the SMP Linux instance may contain many user spaces processes. These processes can be (and normally are) multi-threaded using the pthreads facility of Linux. Define a "USDPA thread" as a thread that has been allocated a BMan

and a QMan software portal for its use via direct access. A "USDPAA process" is a Linux user space process that contains at least one USDPAA thread.

Clearly, the number of USDPAA threads is limited by the number of software portals that the particular SoC provides. The P4080 provides 10 QMan and 10 BMan software portals. The available portals must be divided into two sets: portals for use by the Linux kernel and portals for use by USDPAA threads.

7.3.1.1.3 USDPAA and legacy Linux software

This section describes the relationship between USDPAA and various types of legacy software.

7.3.1.1.3.1 Legacy user space applications

USDPAA provides user space processes access to DPAA functionality via user space device drivers and device-oriented C-language APIs layered upon them. Legacy applications must be modified in order to use these APIs. USDPAA's primary goal is to provide these APIs.

As an example, it is possible to offload cryptographic operations to the DPAA Security Engine (SEC) using USDPAA drivers, but this does not imply that all applications that use a cryptography-related legacy Linux facility are automatically accelerated. Software someplace (in user space) must explicitly use the USDPAA functions.

It is possible that some legacy user space facilities could be accelerated in a manner that automatically applies to legacy user space applications if means other than or in addition to USDPAA are used. This topic is, however, beyond the scope of the document.

7.3.1.1.3.2 Relationship to conventional kernel-based drivers

Conventional device drivers reside in the Linux kernel and are accessed via system calls such as read and write. These often (but not always) involve copying data between the application's address space and the kernel's address space. These types of drivers are desirable in many situations.

The existence of USDPAA neither implies nor excludes the existence of conventional drivers. It also implies nothing about whether or not conventional drivers copy data. Mostly, the topic of conventional drivers is beyond the scope of this document. Details will vary among DPAA IP blocks. A few points follow:

- SEC has a job queue interface that can be used concurrently with its QMan interface. The DPAA SDK contains kernel code that does this.
- As figure [QMan driver overview](#) on page 696 shows, The DPAA's Buffer Manager and Queue Manager in-kernel portal drivers support access by multiple entities such as the DPAA kernel ethernet driver. It is possible to create custom conventional kernel drivers that layer on top of the in-kernel portal drivers. These can provide traditional kernel-mediated device access.

7.3.1.2 USDPAA assumptions and use cases

The primary purpose of USDPAA is application performance. User space drivers can provide substantial performance improvement over traditional kernel-mediated access to devices.

- No system calls are needed to do I/O.
- This implies no need to switch into and out of the kernel's execution context.
- User space applications directly access data buffers, thus providing zero copy I/O in all cases.

Because the goal is performance, USDPAA provides simple but low-level access to I/O functionality. It does not employ complex and costly abstractions. Applications deal directly with QMan and BMan via their software APIs. These form a thin but helpful layer between the application and the hardware.

7.3.1.2.1 Assumptions

The USDPAA software makes certain assumptions and supports multiple use-cases. These are detailed in the sections that follow.

7.3.1.2.1 General assumptions

- USDPAAs threads should be made affine to a core. This is due to support for stashing to per-core caches. (Nothing breaks if this is not the case, but performance will be sub-optimal.)
- Every USDPAAs thread has its own BMan and QMan software portals. No other thread or entity accesses these portals. Within the drivers, these are stored as thread-local variables and the locking assumptions depend on them being thread-private.
- USDPAAs threads may make use of arbitrary Linux system calls. For example, they can do file I/O.
- USDPAAs threads are compatible with general Linux services such as debuggers like gdb.
- Threading is via standard Linux pthreads and the standard Linux GNU C library.

7.3.1.2.2 Use cases

USDPAAs is intended to support multiple use-cases, and the supported use-case set will grow with time.

7.3.1.2.2.1 Run-to-completion

The full run-to-completion use case is defined by the following characteristics:

- The number of USDPAAs threads per core must not exceed the number of QMan and BMan portals assigned to that core (and reserved for USDPAAs use) within the device-tree. Note however that testing with more than one thread per core has been minimal. Not all cores need have a USDPAAs thread, however.
- Cores hosting USDPAAs threads may be configured to run nothing in user space other than that core's USDPAAs thread. For example the kernel parameter "isolcpus" could be used - see [CPU isolation](#) on page 701 for more on this topic.
- The /proc/irq mechanism may be used to limit interrupts for miscellaneous peripherals to non-isolated cores. Again, USDPAAs does not require this architecturally, but it is often done.
- In run-to-completion, USDPAAs threads poll their portals. Polling generally implies that the USDPAAs threads will always be running or ready to run. It would be unusual to have other threads scheduled on the same core when USDPAAs threads are in this "non-interactive" mode.
- Often, one thinks of the USDPAAs threads as "workers" able to process any form of "work" delivered via QMan messages. In this model, the QMan scheduler can be used as a general "work" scheduler rather than relying on the Linux scheduler for this purpose.

7.3.1.2.2.2 Interrupt-driven

USDPAAs supports an interrupt-driven model that allows benefits such as cooperating with other threads on the same core, potential power-saving by not polling all of the time, and so on. This model introduces more operating system overheads and thus trades performance and latency for these benefits.

- The user space drivers for BMan and QMan software portals are implemented using a Linux character device. It permits USDPAAs threads to await interrupts from software portals by doing file operations (like "select()") using a file descriptor associated with the user space device. See section [QMan and BMan drivers and C API](#) on page 696 for more details.
- USDPAAs threads process dequeued frames from portals (as much as they like) after an interrupt indicates that data are available. When dequeue processing is complete, the thread re-enables the interrupt.
- This interrupt mechanism allows USDPAAs threads to sleep awaiting I/O. Thus, other threads can execute while a particular USDPAAs thread sleeps. In this model, it is normal it have multiple threads, USDPAAs or otherwise, execute on the same core.
- This use-case is compatible with SCHED_FIFO Linux scheduling. USDPAAs is independent of scheduling policy. It is also independent of real-time policy and patches such as PREEMPT_RT.

Application developers may chose the model, run-to-completion or interrupt-driven that best fits their need. The PPAC-based example applications in USDPAAs (eg. reflector) implement a hybrid of both modes, where an interrupt-driven mode is used

whenever the packet-processing has been idle for a short period of time, and switching back to run-to-completion once processing resumes.

7.3.1.3 USDPAA components

USDPAA consists of several components which may be used in the context of standard SMP Linux on the NXP family of DPAA-based SoCs. These components are described below.

7.3.1.3.1 Device-tree handling

Many configuration and resource details are defined within the "device-tree" used to boot Linux. The kernel itself uses this data structure to determine the system's devices and attributes, and USDPAA applications are dependent on the same information. As will be seen, the QMan and BMan portals that are available to USDPAA (and their attributes such as channel IDs) are declared within the device-tree, as are the ethernet interfaces and related attributes from the FMan side of things.

The USDPAA "of" driver^[12] parses the device-tree information from the procfs filesystem exported by the Linux kernel (as found at /proc/device-tree/) and constructs an internal data-structure representation of the tree, including interrelationships between device-nodes (eg. portals and CPUs, pool-channels and portals, etc). This information is then used by driver and application-configuration layers where required to obtain device information.

7.3.1.3.1.1 Device-tree initialization requirements

In the current "of" driver implementation, it is the application's responsibility to initialize this driver layer prior to initializing or using any other driver layers that may be dependent on it. This is achieved by calling the of_init() API. To satisfy leak-checkers or to support the use of persistent/reusable processes, an application can also undo all allocations and driver state by calling of_finish().

The "Queue Manager, Buffer Manager API Reference Manual" covers these APIs in more detail, if the above description (and the <usdpaa/of.h> header) are insufficient.

7.3.1.3.2 QMan and BMan drivers and C API.

The Queue Manager (QMan) and Buffer Manager (BMan) drivers are the heart of USDPAA. The two drivers are structurally similar in the broadest sense. This discussion will focus on QMan, but the material also applies to BMan.

7.3.1.3.2.1 QMan driver overview

The QMan driver serves two roles:

1. Initialization and management of all aspects of QMan that are not per-portal, i.e. are global.
2. Initialization and management of per software portal QMan operations, i.e. operations that are local to the system component to which the portal is dedicated.

It is reasonable to think of the QMan driver as being split into parts: a QMan global driver and QMan software portal driver. There is only one instance of the global driver per SoC. It is always a kernel-level driver. There is an instance of the software portal driver for every QMan software portal that exists physically on the SoC. For example, P4080 provides 10 QMan software portals. Thus, there are 10 instances of the portal driver available.

[12] "OF" refers to Open Firmware, the specification behind device-trees, a subject beyond the scope of this document.

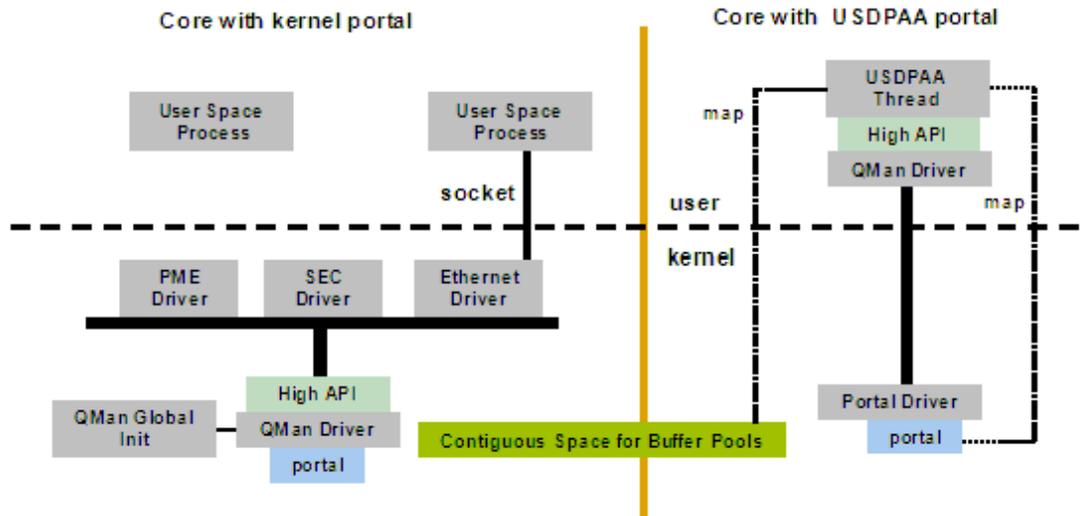


Figure 115. QMan Driver Architecture

As the figure above shows, the software portals (and hence their driver instances) may be dedicated for kernel or user space use. Essentially the same QMan portal API is presented for both the kernel and the user space instances. Indeed, the same software is used in user space and the kernel.

The kernel driver instances have existing "users" in the kernel. For example, the ethernet driver, the PME driver and (though not in the current release) the SEC driver all use the QMan API to share access to the same physical portal on a core. They all enqueue and queue on the same portal using the API associated with the portal's driver instance. Kernel developers are free to add additional users in the form of additional kernel-level components that share the same portal.

As an aside, the SEC is a special case because it has two interfaces that software can use: the job queue interface, and the QMan interface. With current versions of the DPAA SDK, kernel software uses the job queue interface. User space processes can use the SEC via QMan using their portals.

User space processes access QMan using software portals that are dedicated to user space. To a great extent the QMan API abstracts it, but this access is implemented using standard Linux character devices for user space drivers. It provides /dev entries for devices to be accessed from user space. A user space process opens such a device and requests that the device be mapped directly into the process's address space. The assumption is that the physical device is a conventional memory mapped peripheral - as QMan software portals are. At this point, the process can access the device hardware via normal load and store instructions.

QMan's hardware design requires that portals be mapped with the correct caching attributes for each part of the portal. In addition, careful instruction sequences are needed to ensure that portal operations are carried out correctly. This is due to the hardware's cache stashing support, which provides low-latency access to portals and data dequeued from portals.

The QMan API mentioned above abstracts this and provides software with a convenient method to perform portal operations such as enqueues and dequeues. The QMan API in user space is a library that is layered on top of the USDPAAs character device infrastructure. For the user space driver instances, there is

- In user space:
 - The QMan API
 - Direct access to the portal hardware
- In the kernel:
 - The character device code to establish mappings
 - Handle interrupts

This C-callable API is documented in the "Queue Manager, Buffer Manager API Reference Manual" that is distributed with the DPAA SDK software.

One difference in usage of portals in user space versus the kernel is that kernel portals are expected to be persistent. They are brought into service and remain in service so upper layer kernel services such as the ethernet driver can assume that they are there. User space portals are, in contrast, dedicated to an application thread and need to be in service only as long as that thread is running. Thus, the time during which the portal is in service is determined by the application and all uses of the portal are determined by the application.

7.3.1.3.2 QMan portals and the Linux device-tree

Since QMan software portals can be dedicated to either the Linux kernel or user space, there must be a mechanism to indicate how each portal will be used.

Linux device trees describe physical resources that are available to Linux and provide information that allows Linux to select drivers for those devices. As such, there are entries in the device tree for all QMan software portals.

The device tree property "fsl,usdpaa-portal" indicates that a given portal is to be made available to user space; the absence of this property implies that the portal will be used only within the Linux kernel.

A listing of two QMan software portal device tree nodes follows. The first portal is used by the kernel. The second is made available to user space.

```
qportal0: qman-portal@0 {
    cell-index = <0x0>;
    compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";
    reg = <0x0 0x4000 0x100000 0x1000>;
    cpu-handle = <&cpu0>;
    interrupts = <104 0x2 0 0>;
    fsl,qman-channel-id = <0x0>;
    fsl,qman-pool-channels = <&qpool1 &qpool2 &qpool3>;
};

qportal1: qman-portal@4000 {
    cell-index = <0x1>;
    compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";
    fsl,usdpaa-portal;
    reg = <0x4000 0x4000 0x101000 0x1000>;
    cpu-handle = <&cpu1>;
    interrupts = <106 0x2 0 0>;
    fsl,qman-channel-id = <0x1>;
    fsl,qman-pool-channels = <&qpool14 &qpool15 &qpool16
                            &qpool17 &qpool18 &qpool19
                            &qpool110 &qpool111 &qpool112
                            &qpool113 &qpool114 &qpool115>;
};
```

There is no mechanism provided to determine what, if anything, user space software will do with a given user space portal. Initially, these portals are present but not initialized. Each user space portal has a /dev entry. It is a USDPAA driver decision as to which portals a given process or thread will use. A portal is placed into use when a user space process or thread requests it via the QMan API.

7.3.1.3.2.3 Note on the current implementation

In the current implementation, portals are bound to cores via the cpu-handle in the portal device tree node. Most commonly, a thread will make itself affine to its portal's core, because stashing for the portal will be targeting that core's cache. Functionality will not break if the thread executes on a different core, because coherency will be maintained, but at the expense of sub-optimal performance (and increased contention between the cores).

Expect more flexibility in this respect in later releases of the QMan software.

7.3.1.3.2.4 Portal initialization requirements

As described in [Device-tree initialization requirements](#) on page 696, the USDPAAs "of" driver must be initialised prior to trying to initialize or use QMan or BMan. Once this is done, a pthread that wishes to be initialized with access to a QMan portal should call the `qman_thread_init()` API, which is defined in the `<usdpaa/fsl_usd.h>` header along with all other USDPAAs-specific QMan/BMan APIs.

Please consult the "Queue Manager, Buffer Manager API reference" for more information.

7.3.1.3.2.5 Buffer Manager (BMan)

The BMan driver and its software support is very similar to QMan's. BMan software portals may be used in the kernel or in user space just like QMan portals. Just as with QMan, the same BMan software API is available in both the kernel and in user space. This API also is defined in the "Queue Manager, Buffer Manager API Reference Manual."

7.3.1.3.2.6 Raw Portal APIs

Raw portal APIs are available to allow USDPAAs process to allocate QMan and BMan portals on behalf of another processor. The portals allocated by these APIs are not configured, it is the responsibility of the allocator to do any configuration of the portal that may be needed. The APIs are as follows:

- `int qman_allocate_raw_portal(struct usdpaa_raw_portal *portal)` - Allocates an unconfigured QMan Portal
- `int qman_free_raw_portal(struct usdpaa_raw_portal *portal)` - Releases a raw QMan portal that was previously allocated
- `int bman_allocate_raw_portal(struct usdpaa_raw_portal *portal)` - Allocates an unconfigured BMan portal
- `int bman_free_raw_portal(struct usdpaa_raw_portal *portal)` - Releases a raw BMan portal that was previously allocated

For detailed information on then `usdpaa_raw_portal` struct refer to the `fsl_usd.h` header file.

7.3.1.3.3 DMA memory management

The NXP DPAA hardware provides several peripherals such as FMan, SEC, and PME that read and write memory directly using DMA. Buffers used for peripherals' DMA must be allocated from memory with special properties:

- The memory must be physically contiguous since the peripheral does not access memory via the core's MMU (or any page-mode I/O MMU).
- The memory must be addressable by the peripheral.
- The memory must not be swapped out by Linux while the device is accessing it.
- For user space drivers, there must be an efficient mechanism to convert the physical addresses used by the peripheral hardware to and from the effective addresses used in a user space process or thread's address space.
- For BMan and DPAA usage, it is often convenient for software to allocate memory from physically contiguous regions that are quite large.
- It is desirable to use the Power core's TLB1 mechanism to map large physically contiguous memory regions of this sort. This increases performance by reducing the number of MMU-related interrupts that must be processed. A single TLB0 entry can map only a single 4K page. A single TLB1 entry, in contrast, can map a large (but variable-sized) page. One TLB1 entry can do the work of many TLB0 entries in circumstances such as this one.

Memory that meets the criteria above is called DMA memory. Drivers for all DMA-capable devices must use DMA memory for buffers. This is true for both conventional in-kernel drivers and user space drivers. It is a hardware implication of peripherals' relationships to cores and MMUs.

7.3.1.3.3.1 Current USDPAAs solution

The USDPAAs Linux kernel reserves a contiguous region of memory very early in the kernel boot process for use as DMA memory. This is done prior to full initialization of the kernel's memory-management subsystem that subsequently inhibits such allocations. This memory reservation is of a size and alignment that is hard-coded into the kernel via the `Kconfig` option

"CONFIG_FSL_USDPAA_SHMEM". Within the kernel's "make menuconfig" interface, this can be found under "Device Drivers" -> "Misc devices" -> "NXP USDPAA shared memory driver". The default is for a 64MB memory reservation.

This same USDPAA kernel driver exposes the reserved memory range via a "/dev/fsl_usdpaa_shmem" character device, which allows the USDPAA application to mmap() the physically-contiguous memory range to a corresponding contiguous range in its virtual address space, thus facilitating trivial virtual/physical address conversions. Additionally, a hook is placed into the memory-management code to "catch" page faults within this memory range and ensure that they are resolved by a single TLB1 mapping that spans the entire memory reservation (rather than the usual 4KB TLB0 mappings). That is, once the USDPAA application has accessed any address within the DMA memory range, no more page faults should occur for any other access within the entire region. Given that USDPAA datapaths of less than 200 processor cycles per packet have been demonstrated, for traffic rates that can consume over 1MB of buffer space in less than a millisecond, we consider that incurring the overhead of page-fault handlers in the kernel when iterating across thousands of pages of memory would likely be too high a price to pay for high performance applications.

The 'fsl_usdpaa_shmem' driver does not automatically constrain the mmap() alignment, so due to the alignment requirements of TLB1 entries, the USDPAA application has to propose virtual base addresses to the kernel rather than letting it allocate them. This will be fixed in a future release. However all of this is encapsulated within the USDPAA "dma_mem" driver. This driver is initialized via the dma_mem_setup() API, which handles the loading and mapping of the DMA memory region. Furthermore, a hard-coded subset of the DMA memory is reserved for buffer pool usage in the USDPAA demo applications, and the rest of it is exposed for general purpose DMA-able memory allocation via the dma_mem_memalign() and dma_mem_free() APIs.

Implementation Note

The USDPAA QMan and BMan APIs always refer to buffers by physical address, as these are what are passed to and from DMA-enabled devices within the Datapath Architecture. So ultimately, any mechanism can be used to provide "DMA memory" to a USDPAA application so long as it allows the application easy access to the memory and an efficient mechanism for converting to and from physical addresses.

Performance considerations will probably also require that page-faults be minimized or eliminated in performance-critical scenarios. The USDPAA "dma_mem" driver (and underlying character device and TLB1 kernel hook) is just one way to achieve this. In future USDPAA releases, it is likely that this will be deprecated in favour of a HugeTLB-based mechanism. In particular, this would help eliminate a limitation of the current USDPAA release, that prevents more than one application instance running at once-- the DMA memory region can only be safely mapped into one process at a time.

7.3.1.3.2 DMA memory API

See the <usdpaa/dma_mem.h> header for a simple user space DMA memory API.

- dma_mem_memalign - dynamic allocation of DMA memory from within the large physically contiguous region.
- dma_mem_free - free allocated memory.
- dma_mem_ptov - convert physical to virtual (effective) address within user space process address space.
- dma_mem_vtop - convert virtual (effective) address to physical address.

Expect this API to be expanded in future software releases.

7.3.1.3.4 Network configuration

The USDPAA QMan and BMan drivers do not, in and of themselves, dictate which resources such as frame queues or buffer pools are used. In some cases, they can be dynamically allocated. In other cases, the specific resource is determined by a factor that is external to the USDPAA application itself.

The most common examples relate to the frame manager (FMan) and are discussed in section [Relationship to SDK Linux ethernet subsystem](#) on page 702.

The PPAC-based USDPAA applications ("reflector" and "ipfwd") use the usdpaa_netcfg_acquire() API to determine the specific resources needed for a network interface. This configuration information is collected from several external sources:

- FMC policy file

- FMC configuration file
- the device-tree

7.3.1.3.4.1 Network configuration initialisation requirements

As described in [Device-tree initialization requirements](#) on page 696, the USDPA "of" driver must be initialised prior to network configuration being extracted from it. Once this is done, the `usdpaa_netcfg_acquire()` API, as defined in the `<usdpaa/usdpaa_netcfg.h>` header, can be used to extract the network configuration. In addition to initialisation of the USDPA "of" driver layer, this API also requires the path to the two FMC XML files described above (policy and configuration), which are passed as parameters.

To satisfy leak-checkers or to support the use of persistent/reusable processes, an application can also undo all allocations and state by calling `usdpaa_netcfg_release()`, passing the "info" structure previously returned from `usdpaa_netcfg_acquire()` as a parameter.

7.3.1.3.5 CPU isolation

USDPA provides no non-standard Linux features to provide CPU isolation, but the topic is important enough to merit some discussion. Standard Linux features may be use to achieve CPU isolation.

Especially in the run-to-completion use case, it can make sense to dedicate an entire core (which Linux would call a "cpu") to a single USDPA process or thread. In other words, one wishes to reduce or eliminate any other software use of that particular core.

First, consider user space processes and threads. Linux provides a helpful kernel parameter called "isolcpus". This parameter indicates to the Linux kernel which cores should not have any user space process or thread scheduled onto them by default. The only way that a user space process or thread can execute on an isolated core is by explicit request.

For example, booting the kernel with "isolcpus=1-7" will isolate cores 1 through 7 as shown in the figure below. Note that this isolation is not architecturally required for functionality.

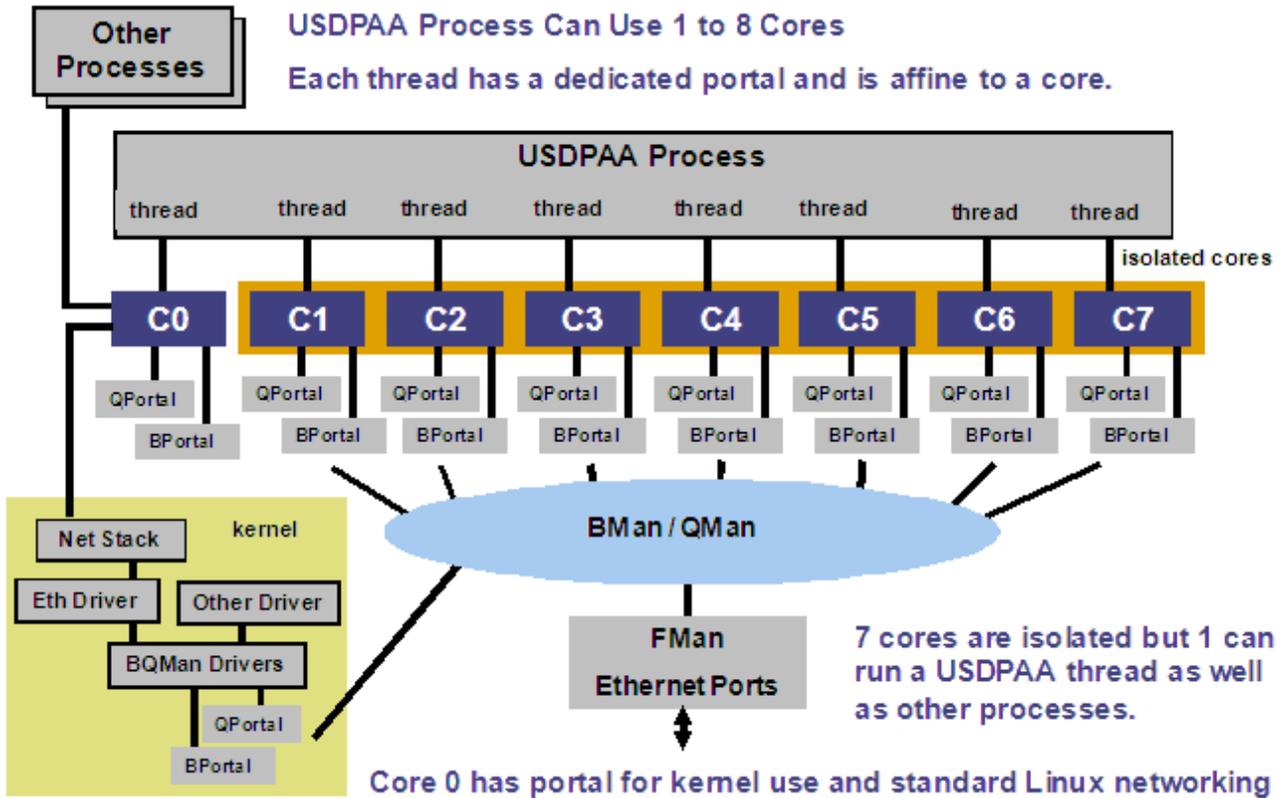


Figure 116. Portal allocation, affinity, and core-isolation

The "isolcpus" kernel parameter is documented (along with others) in the kernel source documentation directory, Documentation/kernel-parameters.txt.

The most convenient way to cause a USDPAAs thread to be scheduled on an isolated to core is to use the function pthread_setaffinity_np(). This is a standard Linux pthreads function and "man pthread_setaffinity_np" should give documentation on it. (The "_np", for "non-portable", simply implies that it is not necessarily available in other pthreads implementations on other operating systems.)

The USDPAAs example applications also provide an example of this function's use. It is used not only to allow the USDPAAs thread to be executed on an isolated core, it makes the thread affine to that core. Recall that affinity is recommended when using the user space QMan and BMan drivers.

Isolation is also a kernel topic. Users who wish to maximize isolation should use the standard Linux interrupt core binding mechanism, /proc/irq to bind interrupts to cores other than those that are isolated. Of course, interrupts associated with portals bound to cores should be bound to that core.

7.3.1.4 Relationship to SDK Linux ethernet subsystem

The DPAA SDK FMan and Linux kernel ethernet driver software is defined as external to the USDPAAs software for this release. However, USDPAAs, the ethernet driver in the kernel, and the FMan software are all interrelated. The figure below shows four use cases involving the three components.

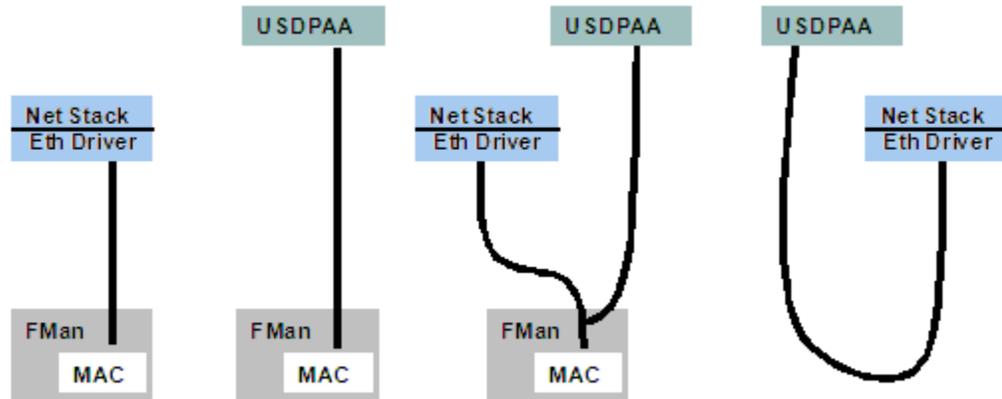


Figure 117. USDPA, FMan, ethernet use cases

1. QMan connects FMan MAC only to the kernel ethernet driver, which exchanges frames with the Linux network stack.
2. QMan connects an FMan MAC only to a USDPA application.
3. QMan connects an FMan MAC to both the kernel driver and to a USDPA application. On ingress, FMan selects the destination for each frame and enqueues it onto a particular frame queue accordingly. Different frame queues make the connections between the MAC and the ethernet driver and the MAC and the USDPA application. This use case can be generalized by assuming multiple USDPA applications, multiple ethernet driver instances, or both.
4. QMan connects an ethernet driver to a USDPA application. FMan is not involved. Note that it is possible to accomplish this via the standard Linux facility TUN/TAP rather than using QMan.

The current release of USDPA demonstrates cases 1 and 2.

7.3.1.4.1 Selecting ethernet interfaces for USDPA

As will be discussed below, the ethernet driver is always involved in configuring FMan. This is true even for the second use case above in which the ethernet driver does not directly use FMan. Instead, the ethernet driver is creating an interface for another entity to use. Historically in the SDK that other entity was the LWE. In the current release, that other entity is a USDPA application. The FMan and ethernet subsystem is actually unchanged from the standard DPAA SDK.

It is ethernet-related Linux device tree entries that determine the use case. This is documented in the "P4080 DPAA Device Bindings" distributed with the DPAA SDK. The sub-topic is "Data-Path Acceleration Assist".

The following device tree snippet shows a Linux private interface and also an interface used privately by USDPA.

```

ethernet@0 {
    compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
    fsl,bman-buffer-pools = < &bp7 &bp8 &bp9>;
    fsl,qman-channel = < &qpool4>;
    fsl,qman-frame-queues-rx = < 0x50 1 0x51 1>;
    fsl,qman-frame-queues-tx = < 0x70 1 0x71 1>;
    fsl,fman-mac = < &enet0>;
};
ethernet@1 {
    compatible = "fsl,p4080-dpa-ethernet", "fsl,dpa-ethernet";
    fsl,qman-channel = < &qpool1>;
    fsl,fman-mac = < &enet1>;
};

```

The first ethernet is used by USDPA. The second is used by the Linux ethernet driver.

Table 98. P4080DS ethernet interfaces

Deice Tree Name	U-boot Name	U-Boot MAC Environment Variable	Linux Name (udev)	SerDes 0xe Physical Position
ethernet@0	FM1@DTSEC1	ethaddr	fm1-gb0	not used
ethernet@1	FM1@DTSEC2	eth1addr	fm1-gb1	Motherboad RGMII
ethernet@2	FM1@DTSEC3	eth2addr	fm1-gb2	not used
ethernet@3	FM1@DTSEC4	eth3addr	fm1-gb3	not used
ethernet@4	FM1@TGEC1	eth4addr	fm1-10g	slot 5 XAUI
ethernet@5	FM2@DTSEC1	eth5addr	fm2-gb0	not used
ethernet@6	FM2@DTSEC2	eth6addr	fm2-gb1	not used
ethernet@7	FM2@DTSEC3	eth7addr	fm2-gb2	slot 3 SGMII 2nd closest to motherboard
ethernet@8	FM2@DTSEC4	eth8addr	fm2-gb3	slot 3 SGMII closest to motherboard
ethernet@9	FM2@TGEC1	eth9addr	fm2-10g	slot 4 XAUI

The DPAA SDK uses different names for different physical ethernet interfaces in different contexts. This is due to long-standing decisions on naming conventions, many not made by NXP. It can be confusing. See table above for a list of all of the ethernet interfaces on the P4080 and their names in all of the important contexts. For P4080, no single SerDes protocol number provides access to all of the ethernets at the same time. This is due to pin multiplexing reasons.

The example defaults assume the use of the SerDes 0xe because it gives a total of 23 Gbps of ethernet connectivity on the P4080DS. The table shows which ethernet interfaces are used in SerDes 0xe.

7.3.1.4.2 FMC, FMD, and the ethernet Driver

The Frame Manager (FMan) software in the DPAA SDK is divided into a user space component (FMC) and a kernel component (FMD). The latter is the FMan driver in the conventional sense. FMD provides a capable API to support FMan configuration.

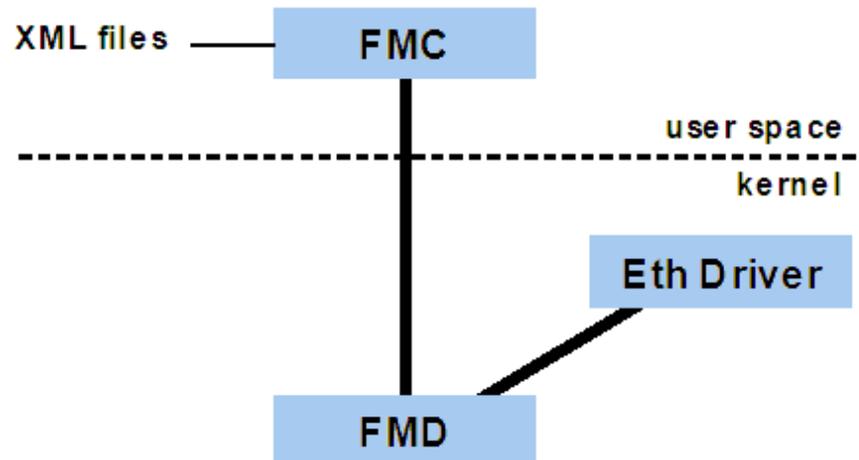


Figure 118. FMC, FMD, and ethernet

By itself, the ethernet driver uses FMD to configure only very simple FMan use cases. Basically, it configures only the default and error frame queues for ingress and egress. Doing more requires running the "fmc" user space application.

This application reads xml files as input and can establish more complex use-cases such as parsing incoming frames and hashing them into a set of frame queues for distribution to multiple cores (meaning USDPAAs threads in the context of this document).

Various USDPAAs example applications provide XML files and assume that fmc will be run. This usage is described in the application user guides.

7.3.1.5 Supported hardware platforms

This USDPAAs release is tested in the following environments:

7.3.1.5.1 P4080DS

- P4080 revision 2 SoC. (Revision 1 is not supported.)
- The main test configuration
 - SGMII card in slot 3 with two interfaces used.
 - XAUI card in slot 4 (but the reflector example will work without it)
 - XAUI card in slot 5 (but the reflector example will work without it)
- SerDes protocol 0xe is the main test case. It provides
 - 1 x 1G ethernet port (via RGMII) used by the Linux kernel network stack.
 - 2 x 1G ethernet ports (via SGMII) used by USDPAAs.
 - 2 x 10G ethernet ports (via XAUI) used by USDPAAs.
- Testing is done with P4080 high bin clocking:
 - 1500 MHz core, 800 MHz platform, 1300 MHz DDR rate, 600 MHz FMan and PME
- Memory, 4 GB (but 8 GB should work also).
- Linux 8-way SMP.
- All caches (core-private L1/L2 and shared L3) are enabled. No locking or cache partitioning is used.

7.3.1.5.2 P3041DS

- SerDes protocol 0x36 (RR_HXAPNSP_0x36) is the main test case. It provides
 - 2 x 1G ethernet port (via RGMII) used by USDPAA.
 - 3 x 1G ethernet ports (via SGMII) used by USDPAA.
 - 2 x 10G ethernet ports (via XAUI) used by USDPAA.
- no interfaces used/available for the Linux kernel network stack.
- Testing is done with the rcw_15g_1500mhz:
 - 1500 MHz core, 750 MHz platform, 1333 MHz DDR rate, 583 MHz FMan and 375 MHz PME

7.3.1.5.3 P5020DS

- SerDes protocol 0x36 (RR_HXAPNSP_0x36) is the main test case. It provides
 - 2 x 1G ethernet ports (via RGMII) used by USDPAA.
 - 3 x 1G ethernet ports (via SGMII) used by USDPAA.
 - 1 x 10G ethernet port (via XAUI) used by USDPAA.
 - no interfaces used/available for the Linux kernel network stack.
- Testing is done with the rcw_15g_2000mhz:
 - 2000 MHz core, 800 MHz platform, 1333 MHz DDR rate, 600 MHz FMan and 400 MHz PME

7.3.1.5.4 P5040DS

- SerDes protocol 0x02 (RR_XXSNSpP_0x02) is the main test case. It provides
 - 2 x 1G ethernet ports (via RGMII) used by USDPAA.
 - 4 x 1G ethernet ports (via SGMII) used by USDPAA.
 - 2 x 10G ethernet port (via XAUI) used by USDPAA.
 - no interfaces used/available for the Linux kernel network stack.
- Testing is done with the rcw_26g_2267mhz.bin:
 - 2267 MHz core, 800 MHz platform, 1600 MHz DDR rate, 600 MHz FMan and 400 MHz

7.3.1.5.5 P2041RDB

- SerDes protocol 0x09 (RR_PX_0x09) is the main test case. It provides
 - 2 x 1G ethernet ports (via RGMII) used by USDPAA.
 - 2 x 1G ethernet ports (via SGMII) used by USDPAA.
 - 1 x 10G ethernet port (via XAUI) used by USDPAA.
 - no interfaces used/available for the Linux kernel network stack.
- Testing is done with the rcw_14g_1500mhz:
 - 1500 MHz core, 750 MHz platform, 1333 MHz DDR rate, 583 MHz FMan and 400 MHz PME

7.3.1.5.6 B4860QDS

- SerDes protocol 2a and 49 (N_NNSS_0x2A_0x49) is the main test case. It provides
 - 4 x 1G ethernet port (via SGMII) used by USDPAA.

- 1 x 1G ethernet port (via SGMII) used/available for the Linux kernel network stack.
- Testing is done with the rcw_5sgmii_1600mhz.bin:
 - 1600 MHz core, 666.667 MHz platform, 800 MHz DDR rate, 666.667 MHz FMan and 333.333 MHz QMan.

7.3.1.5.7 T4240QDS

- SerDes protocol RR_XXXXPRPR_1_1_6_6 is the main test case. It provides
 - 2 x 1G ethernet ports (via SRIO) used by USDPAAs.
 - 4 x 10G ethernet port (via XAUI) used by USDPAAs.
 - 2 x 1G ethernet ports (via PCIe) used/available for the Linux kernel network stack.
- Testing is done with the rcw_1_1_6_6_1666MHz.bin:
 - 1666.667 MHz core, 666.667 MHz platform, 800 MHz DDR rate, 400 MHz FMan and 333.333 MHz PME

7.3.1.6 Example applications

USDPAAs will be distributed with a set of example applications that is expected to grow over time.

The current set is

- A packet reflector application, "reflector".
- An IP forwarding performance demonstration, "ipfwd".
- A cryptographic accelerator example, "simple_crypto".
- A pattern-matching accelerator example, "pme_loopback_test".
- An IPFwd application based upon Longest Prefix Match methodology, "lpm_ipfwd".
- An application to route IPv4 packets after performing encryption/decryption, "IPSecfwd".
- A non-PPAC based stand-alone application, "hello_reflector".

Each example has its own user manual.

7.3.1.7 USDPAAs installation and execution

The USDPAAs software is contained within the SDK release, and should be compiled and installed by default by following the general instructions accompanying the release. Those instructions will not be reproduced here in their entirety. All that should be required to bring up and run the USDPAAs environment, compared to booting the "standard" Linux system from the SDK, is to use the USDPAAs-specific device tree. Eg. on the P4080DS system, one should use "ulmage-p4080ds-usdpaa.dtb" instead of "ulmage-p4080ds.dtb". Other than that, please follow the steps prescribed in the SDK installation notes. The following sections reproduce some of these steps (for the P4080DS case only), but primarily to ensure that USDPAAs-relevant information is correctly identified.

7.3.1.7.1 Files needed to boot Linux on the P4080DS system

To run the USDPAAs software, one must first boot Linux with the correct files;

- **R_PPSXX_0xe/rcw_2sgmii_1500mhz.bin**
Reset configuration word (rcw) that selects the SerDes Protocol, suitable to program into the P4080DS NOR flash.
- **u-boot-P4080DS.bin**
U-Boot bootloader image suitable to program into the P4080DS NOR flash.
- **fsl_fman_ucode.bin**
FMan microcode for appropriate P4080 silicon suitable to program into the P4080DS NOR flash.

Linux User Space
USDPAAs

- **ulmage-p4080ds.bin**

Linux kernel supporting USDPAAs.

- **ulmage-p4080ds-usdpaa.dtb**

Device tree file configured for USDPAAs usage.

- **fsl-image-core-p4080ds.ext2.gz.u-boot**

File system (used in RAM disk) that contains the needed user space files including USDPAAs example application binaries. Linux must be booted using this file system.

All six of the files listed above are needed to run USDPAAs. The first three must be programmed into the P4080DS NOR flash. The last three may be programmed into the NOR flash, but also may be loaded into RAM by u-boot using the tftp protocol.

U-boot is also capable of loading files into RAM via tftp and then programming them into the NOR flash. In all cases, you must have access to a tftp server, ideally on your Linux development host.

Copy the six files listed above to a directory from which they can be accessed via your tftp server. U-boot on the P4080DS must use tftp to access them. Details of installing and configuring a tftp server on your development host are specific to your host Linux distribution.

7.3.1.7.2 About U-Boot and network interfaces

U-boot is a bootloader. It is very flexible. but its main job is to do quite a bit of system configuration and then to load an operating system image (mainly Linux) into RAM and transfer control to it.

One of the simplest ways of transferring images (and other files) to the P4080DS running u-boot is to use tftp. For this to work, you must configure a network interface in u-boot.

Specifically, we assume that the 1 Gbps ethernet interface on the P4080DS motherboard will be used, and that the RCW file used causes this interface to correspond to the 2nd DTSEC in the P4080's first FMan instance.

At this point, we list for reference the text that U-boot should print when it runs. Note that there can be some variation in what U-boot prints.

```
U-Boot 2011.06-rc1-00037-g2bc0243 (May 27 2011 - 11:01:49)

CPU0: P4080E, Version: 2.0, (0x82080020)
Core: E500MC, Version: 2.0, (0x80230020)
Clock Configuration:
  CPU0:1499.985 MHz, CPU1:1499.985 MHz, CPU2:1499.985 MHz, CPU3:1499.985 MHz,
  CPU4:1499.985 MHz, CPU5:1499.985 MHz, CPU6:1499.985 MHz, CPU7:1499.985 MHz,
  CCB:799.992 MHz,
  DDR:649.994 MHz (1299.987 MT/s data rate) (Asynchronous), LBC:99.999 MHz
  FMAN1: 599.994 MHz
  FMAN2: 599.994 MHz
  PME: 599.994 MHz
L1: D-cache 32 kB enabled
  I-cache 32 kB enabled
Board: P4080DS, Sys ID: 0x17, Sys Ver: 0x01, FPGA Ver: 0x0b, vBank: 4
36-bit Addressing
Reset Configuration Word (RCW):
  00000000: 105a0000 00000000 1e1e181e 0000cccc
  00000010: 3842440c 3c3c2000 fe800000 e1000000
  00000020: 00000000 00000000 00000000 008b6000
  00000030: 00000000 00000000 00000000 00000000
SERDES Reference Clocks: Bank1=100MHz Bank2=125MHz Bank3=125MHz
I2C: ready
SPI: ready
DRAM: Initializing...using SPD
Detected UDIMM EBJ21EE8BAFA-DJ-E
```

```

Detected UDIMM EBJ21EE8BAFA-DJ-E
CS2 is disabled.
CS3 is disabled.
CS2 is disabled.
CS3 is disabled.
2 GiB left unmapped
  DDR: 4 GiB (DDR3, 64-bit, CL=9, ECC on)
    DDR Controller Interleaving Mode: cache line
    DDR Chip-Select Interleaving Mode: CS0+CS1
Testing 0x00000000 - 0x7fffffff
Testing 0x80000000 - 0xffffffff
Remap DDR 2 GiB left unmapped

POST memory PASSED
Flash: 128 MiB
L2:    128 KB enabled
Corenet Platform Cache: 2048 KB enabled
SRI01: disabled
SRI02: disabled
MMC:   FSL_ESDHC: 0
EEPROM: Invalid ID (ff ff ff ff)
PCIE1: Root Complex, x1, regs @ 0xfe200000
  01:00.0 - 1095:3132 - Mass storage controller
PCIE1: Bus 00 - 01
PCIE2: disabled
PCIE3: Root Complex, no link, regs @ 0xfe202000
PCIE3: Bus 02 - 02
In:    serial
Out:   serial
Err:   serial
Net:   Fman1: Uploading microcode version 101.8.0
FM1@DTSEC2 connected to Vitesse VSC8244
FM1@TGEC1 connected to Teranetics TN2020
Fman2: Uploading microcode version 101.8.0
FM2@DTSEC3 connected to Vitesse VSC8234
FM2@DTSEC4 connected to Vitesse VSC8234
FM2@TGEC1 connected to Teranetics TN2020
FM1@DTSEC2, FM1@TGEC1, FM2@DTSEC3, FM2@DTSEC4, FM2@TGEC1

```

In the above text, notice the final line;

```
FM1@DTSEC2, FM1@TGEC1, FM2@DTSEC3, FM2@DTSEC4, FM2@TGEC1
```

It lists the available network interfaces, which is determined by the RCW file and the SerDes protocol that it selects. This document assumes that SerDes 0xe is used. It provides the interfaces above. They will be used as follows.

- FM1@DTSEC2, U-boot MAC "eth1addr"

This is the 1 Gbps ethernet interface on the P4080DS motherboard. It will be used by U-Boot to transfer images and also is available in Linux as a standard ethernet interface. The "ifconfig" command will show it as "fm1-gb1" because the "gb's" are counted from zero.

- FM1@TGEC1 (slot 5 XAUJ), U-boot MAC "eth4addr"
- FM2@DTSEC3 (slot 3 SGMII, 2nd closest to motherboard), U-boot MAC "eth7addr"
- FM2@DTSEC4 (slot 3 SGMII, closest to motherboard), U-boot MAC "eth8addr"
- FM2@TGEC1 (slot 4 XAUJ), U-boot MAC "eth9addr"

These other 4 interfaces, 2 x 1 GB and 2 x 10 GB, are dedicated to user space access via USDPA.

This provides the USDPA application with 22 Gbps of full-duplex ethernet capacity, assuming that an SGMII card is in P4080DS slot 3, and 10G XAUI cards are in slots 4 and 5. The USDPA "reflector" application will function if the XAUI cards are not present, but only two GB of capacity will be available.

The P4080 has many ethernet interfaces. See [Selecting ethernet interfaces for USDPA](#) on page 703 for a complete summary of the names of these interfaces in different software contexts and for a statement of their use by the SerDes 0xe protocol on the P4080DS.

Returning to u-boot, selection of the network interface and networking parameters is done via u-boot environment variables. These are set using the "setenv" command that can be entered at the u-boot prompt. The network parameters include MAC addresses for the interfaces. Set them for all ten of the P4080 interfaces even though only a subset will be used.

Here is a complete list of the needed setenv commands. You will have to adjust the IP addresses and netmask to match your network. The addresses shown below are examples only.

```
setenv ethact FM1@DTSEC2
setenv ethaddr 00:04:9F:77:4E:00
setenv eth1addr 00:04:9F:77:4E:01
setenv eth2addr 00:04:9F:77:4E:02
setenv eth3addr 00:04:9F:77:4E:03
setenv eth4addr 00:04:9F:77:4E:04
setenv eth5addr 00:04:9F:77:4E:05
setenv eth6addr 00:04:9F:77:4E:06
setenv eth7addr 00:04:9F:77:4E:07
setenv eth8addr 00:04:9F:77:4E:08
setenv eth9addr 00:04:9F:77:4E:09
setenv ipaddr 10.82.146.151
setenv serverip 10.82.146.150
setenv gatewayip 10.82.146.150
setenv netmask 255.255.255.0
saveenv
```

In summary, U-boot will use the motherboard ethernet and will give it IP address 10.82.146.151. The tftp server has IP address 10.82.146.150. The "saveenv" saves all environment variable values into flash so they are retained after a reboot.

After entering the values above, you can test the U-boot network connection via ping and a trial tftp transfer. If this does not work, check the variables and network cables. This must work. Here is a text capture showing a successful test. You may need to adjust the path usdpaa/u-boot.bin per your tftp server configuration.

```
=> ping $serverip
Using FM1@DTSEC2 device
host 10.82.146.150 is alive
=> tftpboot 01000000 usdpaa/u-boot.bin
Using FM1@DTSEC2 device
TFTP from server 10.82.146.150; our IP address is 10.82.146.151
Filename 'usdpaa/u-boot.bin'.
Load address: 0x1000000
Loading: #####
done
Bytes transferred = 524288 (80000 hex)
```

73.1.73 P4080DS NOR flash banks

The P4080DS board has a feature that uses address swizzling to make it appear that the NOR flash is divided into multiple parts-- this document will assume two. The parts are called "bank 0" and "bank 4".

When you power-on or reset, u-boot will boot from bank 0. U-boot in bank 0 can program images into bank 4. Then, you can enter "pixis altbank" from the bank 0 u-boot prompt to boot into bank 4.

It is very wise to leave the bank 0 images alone and simply use them to program images into bank 4. This is to ensure that you always have working images in bank 0. This document will assume that you have u-boot flashed in bank 0. Use it only to program bank 4.

Look at the u-boot output in [About U-Boot and network interfaces](#) on page 708. The line

```
Board: P4080DS, Sys ID: 0x17, Sys Ver: 0x01, FPGA Ver: 0x0b, vBank: 4
```

shows the bank from which u-boot was booted. In this case it was bank 4 (per vBank).

73.1.7.4 Programming the P4080DS NOR flash bank 4

First, boot from bank 0 by doing a reset or power-on. Check that you see "vBank: 0" since this is very important. Then, set network parameters using the u-boot environment variables described in section [About U-Boot and network interfaces](#) on page 708.

The following U-boot commands will flash all six of the needed files into bank 4. Remember that you may have to adjust the paths in the tftpboot commands per your tftp server.

```
# BE BOOTED FROM BANK 0; WE WILL FLASH THE ALT BANK, WHICH WILL BE BANK 4

# rcw altbank
tftpboot 0x01000000 usdpaa/rcw_2sgmii_1500mhz.bin
protect off 0xec000000 +$filesize && erase 0xec000000 +$filesize && cp.b 0x01000000 0xec000000
$filesize

# u-boot altbank
tftpboot 0x01000000 usdpaa/u-boot.bin
protect off 0xebf80000 +$filesize && erase 0xebf80000 +$filesize && cp.b 0x01000000 0xebf80000
$filesize

# Delete altbank the u-boot env-- use default
protect off 0xebf60000 +0x20000 && erase 0xebf60000 +0x20000

# FMan u-code altbank
tftpboot 0x01000000 usdpaa/fsl_fman_ucose.bin
protect off 0xeb000000 +$filesize && erase 0xeb000000 +$filesize && cp.b 0x01000000 0xeb000000
$filesize

# device tree alt bank
tftpboot 0x01000000 usdpaa/p4080ds-usdpaa.dtb
protect off 0xec800000 +$filesize && erase 0xec800000 +$filesize && cp.b 0x01000000 0xec800000
$filesize

# kernel altbank
tftpboot 0x01000000 usdpaa/uImage
protect off 0xec020000 +$filesize && erase 0xec020000 +$filesize && cp.b 0x01000000 0xec020000
$filesize

# rootfs altbank
tftpboot 0x01000000 usdpaa/initramfs.cpio.gz.u-boot
protect off 0xed300000 +$filesize && erase 0xed300000 +$filesize && cp.b 0x01000000 0xed300000
$filesize
```

73.1.7.5 Boot into bank 4 and set more variables

Next, enter the command "pixis altbank" to boot into bank 4 and press "any key" to stop the boot. Check that u-boot prints "vBank: 4". It is important that it does. If not, there is probably a mistake in the previous steps.

At this point, you must enter all of the networking u-boot environment variables (see section [About U-Boot and network interfaces](#) on page 708) again and also set variable bootcmd. The latter is shown below along with a "saveenv" to save the values.

```
setenv bootcmd setenv bootargs root=/dev/ram rw console=ttyS0,115200 usdpaa_mem=256M
bportals=s0-1 qportals=s0-1 \; bootm 0xe8020000 0xe9300000 0xe8800000
saveenv
```

It is important that you type the backslash (\) because semicolon (;) is a command separator in u-boot. Here is a "printenv bootcmd" in a larger font and with line wrap showing the correct value. Note that the printenv does not show the backslash.

```
=> printenv bootcmd
bootcmd=setenv bootargs root=/dev/ram rw console=ttyS0,115200 usdpaa_mem=256M bportals=s0-1
qportals=s0-1 ; bootm 0xe8020000 0xe9300000 0xe8800000
```

U-boot has an environment variable "bootdelay" that controls the number of seconds u-boot counts down before automatically running command "boot". If you prefer to run "boot" manually, set bootdelay to -1. This will cause u-boot to go directly to a command prompt. You can set bootdelay to whatever you want.

```
setenv bootdelay -1
saveenv
```

7.3.1.7.6 Environment variable hwconfig and optical 10G

At this point, enter the u-boot command "printenv hwconfig" as shown below.

```
=> printenv hwconfig
hwconfig=fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1
```

If you see the value above, and you plan to use the 10G copper interfaces (or no 10G at all), then all is well. You may skip to the next section.

If you plan to use optical 10G interfaces, you must add information to hwconfig. This is important. The optical interface will operate erratically without it. It is easiest to do this using the u-boot "editenv" command. Whatever you type will be appended to hwconfig. The text you need to append is

```
;fsl_fm2_xau_i_phy:xfi;fsl_fm1_xau_i_phy:xfi
```

The leading semicolon is needed. Do another "printenv hwconfig" to ensure that the value is correct and then a saveenv as shown below.

```
=> printenv hwconfig
hwconfig=fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1;fsl_fm2_xau_i_phy:xfi;fsl_fm1_xau_i_ph
y:xfi
=> saveenv
```

7.3.1.7.7 Booting Linux

At this point, you need to reset once again into bank 4 (u-boot command "pixis altbank") and run the u-boot "boot" command, either manually or by letting it happen automatically after the count-down.

Linux will boot and give you a login prompt.

Login as user "root" with password "root".

An "ifconfig -a" should show only one FMan (fm) ethernet interface, fm1-gb1. This is the P4080DS motherboard 1G ethernet interface. You can set its IP address and use it as an ordinary Linux network interface if you wish.

If you `cat /root/SOURCE_THIS`, you will see the commands needed to run the "reflector" example application. This is the first application you should examine. Please see its user manual for more information.

```
[root@p4080 /root]# cat /root/SOURCE_THIS
cd /usr/etc
fmc -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst.xml -a
reflector
```

7.3.1.7.8 Using tftp for the kernel, device-tree, and file-system

It can be slow to reflash the kernel, device tree, and file system every time you run `ltib` to change a USDPAAs application (see section [Known limitations](#) on page 714). As an alternative, you can use the following u-boot commands (from bank 4) to boot Linux.

```
tftpboot 01000000 usdpaa/uImage
tftpboot 02000000 usdpaa/p4080ds-usdpaa.dtb
tftpboot 02100000 p4080ds/initramfs.cpio.gz.u-boot
boot
```

However, in this case the addresses in the "bootm" run by `bootcmd` need to be different. So, set `bootcmd` as follows.

```
setenv bootcmd setenv bootargs root=/dev/ram rw console=ttyS0,115200 usdpaa_mem=256M
bportals=s0-1 qportals=s0-1 \; bootm 01000000 02100000 02000000
saveenv
```

7.3.1.8 Using configurations other than SerDes 0xe

Most NXP testing, examples, and discussion of examples in documentation assumes the use of SerDes 0xe and 22 Gbps of ethernet connectivity to USDPAAs in the form of 2 x 1 Gbps + 2 x 10 Gbps.

It is possible to use USDPAAs in other configurations and some examples will be summarized below.

See the document "NXP DPAA SDK <version>: Selecting Ethernet Interfaces" for background information. It is distributed with the NXP DPAA SDK.

7.3.1.8.1 SGMII (4 x 1 Gbps) card and one XAUI (10 Gbps) card

Goal (using Linux names for network interfaces, see [Selecting ethernet interfaces for USDPAAs](#) on page 703):

- fm1-gb1 used by Linux kernel
- fm2-gb2 used by USDPAAs
- fm2-gb3 used by USDPAAs
- fm2-10g used by USDPAAs

Method:

- SGMII card goes in P4080DS slot 3.
- XAUI card goes in P4080DS slot 4.
- Continue to use SerDes 0xe and RCW file `R_PPSXX_0xe/rcw_2sgmii_1500mhz.bin` in the P4080DS NOR flash.
- Boot into bank 4 (assuming you are using bank 4 for USDPAAs).
- Add text `";serdes:fsl_srds_lpd_b3=0xf"` to u-boot environment variable `hwconfig`, `saveenv`, and reset the system again into bank 4. This disables fm1-10g. The leading ";" separates the information added to `hwconfig` from what was already there.

- Delete references to fm1-10g from the fmc configuration file you will use. For example, delete the following text from it:

```
<engine name="fm0">
  <port type="10G" number="0" policy="hash_ipv4_policy_4"/>
</engine>
```

Run the USDPAA application.

7.3.1.8.2 SGMII (4 x 1 Gbps) card and no XAUI (10 Gbps) card

Goal (using Linux names for network interfaces, see table [Selecting ethernet interfaces for USDPAA](#) on page 703):

- fm1-gb1 used by Linux kernel
- fm2-gb0 used by USDPAA
- fm2-gb1 used by USDPAA
- fm2-gb2 used by USDPAA
- fm2-gb3 used by USDPAA

Method:

- SGMII card goes in P4080DS slot 3.
- Use SerDes 0x10 and RCW file: `R_PPSXN_0x10/rcw_5g_1500mhz.bin` in the P4080DS NOR flash. This is a different RCW file than the one discussed above.
- Boot into bank 4 (assuming you are using bank 4 for USDPAA).
- Add text `;,serdes:fsl_srds_lpd_b2=0xf` to u-boot environment variable `hwconfig`, `saveenv`, and reset the system again into bank 4. This disables fm2-10g. The leading `;` separates the information added to `hwconfig` from what was already there. Note that fm1-10g is never available in SerDes 0x10.
- Delete references to any of the 10 Gbps interfaces and ensure references to all USDPAA 1 Gbps interfaces are present in the fmc configuration file. For example, it should like the following after you edit it:

```
<cfgdata>
  <config>
    <engine name="fm1">
      <port type="1G" number="0" policy="hash_ipv4_policy_5"/>
      <port type="1G" number="1" policy="hash_ipv4_policy_6"/>
      <port type="1G" number="2" policy="hash_ipv4_policy_7"/>
      <port type="1G" number="3" policy="hash_ipv4_policy_8"/>
    </engine>
  </config>
</cfgdata>
```

Run the USDPAA application.

7.3.1.9 Known limitations

- Interrupts for QMan and BMan portals used in user-space by USDPAA threads are not necessarily affine to the CPU to which the portal is assigned (ie. to the portal where stashing is performed and where the threads are advised to be affine). This is a current limitation of the UIO interface in the kernel which does not give the QMan/BMan drivers explicit control over interrupt affinity. However given that interrupts are generally used to implement sleeping/blocking semantics (eg. when idle), this is not expected to have a significant impact. As a workaround, if the need arises, the user can manually override the interrupt-affinity via the `procs` controls available at `/proc/irq`.
- Only 1 Gbps full-duplex operation is supported on 1 Gbps ethernet links. It is also true that 10 Gbps links may only be used at 10 Gbps, but in this case the reason is a P4080DS board-level hardware limitation.

- Present release does not permit working with all four SGMII 1 Gbps ethernet ports and a XAUI 10 Gbps ethernet port at the same time. This would appear possible using SerDes 0x10, but it is not due to FMan buffer size constraints and support for jumbo frames. This is, at heart, a P4080 SoC hardware limitation, but future releases will provide greater flexibility.

7.3.1.10 Document history

Table 99. Document history

Rev	Notes
1.0	Initial version.
1.1	FQID detail and more.
1.2	Description of proof-of-concept added.
1.3	Add TX FQIDs and discussion of application timing
1.4	Documentation of configuration files.
1.5	Instructions on running on the P4080DS. This version describes the first release of the proof-of-concept.
1.6	Corrected reserving 512 bytes in buffers.
1.7	Updated performance table for case when FMan automatically releases buffers.
1.8	Updated for 10G XAUI + 4x1G SGMII support and some optimizations.
1.9	Changed document title. Updated for phase 0 release.
1.10	Updated for USDPAAs phase 1 release.
1.11	Late updates for phase 1 release including advice on non-22Gbps runs.
1.12	Add discussion of use cases beyond SerDes 0xe and 22 Gbps of ethernet connectivity for USDPAAs.
1.13	Fixed minor typographical errors and made other small improvements.
1.14	Updated for SDK v1.0.
1.15	Updated for SDK v1.3.
1.16	Added T4 & B4 platforms for SDK v1.3.1.

7.3.2 USDPAAs Multiple Processor Support User Guide

7.3.2.1 USDPAAs Multiple Process Support

Describes the modifications done in the QorIQ SDK1.2 to provide USDPAAs multiple process support.

The changes introduced in the SDK V1.2 release and are described in this document as a set of changes between SDK V1.1 and the SDK V1.2 releases. Concurrently, a set of interim releases (IRn) are being developed to provide DPAA Offload support

in the SDK. The IR2 release integrates support for multiple USDPAA processes. As such, the delta between SDK V1.1 and SDK V1.2 described in this document also applies as the delta between IR1 and IR2 releases.

7.3.2.2 USDPAA User/Kernel Device Interface

For SDK 1.1 most USDPAA resources were hard-coded into the USDPAA applications, there being only a single USDPAA process. For SDK 1.2 each process opens the “/dev/fsl_usdpaa” device once and uses this process driver for all that process’s resource management.

SDK 1.1 QMan/BMan resources

Prior to the SDK 1.2 release, a simplifying assumption of a single process, together with assumptions about unused resources in the device tree led to hard-coded resources. This was the case for buffer pools, congestion groups, pool channels and frame queues.

SDK 1.1 DMA Memory

The only USDPAA resource that was kernel-managed was the mapping of DMA memory, via the

```
/dev/fsl_usdpaa_shmem
```

device. This also had the simplifying assumption that the entire memory region reserved by the kernel would always be mapped in its entirety by the unique USDPAA process, meaning that the functions in the USDPAA “dma_mem” API did not need any parameter to specify which DMA region was implied as there was only one region.

SDK 2.1 Process Driver

The “fsl_usdpaa_shmem” device has been renamed to simply “/dev/fsl_usdpaa” and is also referred to as the “process” device (within USDPAA, this is handled via the “process” driver), because the intention is for each process to open this device once and to use it for all that process’s resource management.

SDK 2.1 QMan and BMan Resources

As of the 1.2 release, the process device supports ioctl() commands for (de)allocating resources to (and from) the kernel. So multiple USDPAA processes as well as any datapath logic in the kernel will all be using the same allocator for resources.

SDK 2.1 DMA Memory

The DMA mapping of the device remains but is enhanced to allow multiple regions and sub-regions to be allocated and mapped out of the total memory reservation. These regions are mapped independently and can be allocated by USDPAA apps, via the “dma_mem” driver, on the fly. The allocated regions have the same size and alignment limitations that come from the use of TLB1 entries to map them for fault-less access during datapath operations, but the kernel management of these regions is maximally optimal. Ie. any combination of regions that can conceivably fit within the total memory reservation will always be obtainable, independent of the order in which the USDPAA processes request the allocation of those regions.

Other features of the allocation and mapping of DMA regions:

- They can be process-private (“unnamed”) or shareable (“named”). Named regions can be mapped by multiple processes.
- For shared regions, the process driver in the kernel provides a sleep-based locking scheme that the USDPAA dma_mem driver uses to synchronise buffer allocations within a sub-region across multiple processes.

SDK 2.1 Resource Tracking

Because USDPAA processes open the /dev/fsl_usdpaa device and perform all resource management through that file-descriptor, the kernel device driver can track what resources are allocated and deallocated by those process. When such a process exits (intentionally or otherwise) the file-descriptor is cleaned up and this allows the device driver to check which resources had not been explicitly released by the application. If there are unreleased frame queues, buffer pools, pool channels or congestion groups, the kernel driver will issue leak warnings to the kernel log (and/or the serial console). Examples:

```
USDPAA process leaking 10 FQIDs  
USDPAA process leaking 4 QPOOLS
```

Leaked resources are not automatically returned to the allocators, because the current drivers do not yet support automatic clean up and recovery of resources that are left in an undefined (and possibly volatile) state.

Even when applications explicitly deallocate resources back to the kernel-managed allocators (subsequently described here in the "Multi-process PPAC Applications" topic), there is no protection against applications that fail to first put those resources back into their expected "power-on" states. As such, an application that does not correctly clean up resources can, for the current version of the SDK (1.2), pollute the allocators with resources that will be allocated out to other users and lead to undefined results.

7.3.2.3 USDPAA Resource Management

Describes QMan and BMan resource availability and how to declare these resources at system initialization.

QMan and BMan Portals

In SDK 1.1, QMan and BMan portals were declared in the device-tree with properties that pre-determined whether they were for use in the kernel or USDPAA (the latter were marked by the "fsl.usdpaa-portal" property) as well as a pre-determined CPU-affinity (the "cpu-handle" property links to a CPU node).

For SDK 1.2 and later versions, portals are declared in the device-tree simply as hardware resources, with no specification of what the portals will be used for nor which CPU they will be used from. The kernel parses all portals into an internal list, from/(to) which they can be (de)allocated as required.

The QMan driver will, by default, try to allocate a distinct portal for each core and initialise it for kernel use. This behaviour can be influenced by the use of the "qportals" boot argument, to use fewer portals and share them between cores. The mechanism by which a portal can be shared involves cores that do not have their own portals being "slaves" to a core that does have its own portal. Portal processing (interrupts, polling, changing dequeue masks, [etc]) is still performed only on the core to which the portal has been assigned, but software running on slave cores can perform software-initiated commands (enqueues, management commands, [etc]) on the shared portal due.

Once the kernel has initialised portals for its own use, it will allocate and export all the remaining portals as UIO devices for USDPAA use. When a USDPAA application thread initialises a portal for use, the opening of the UIO device triggers kernel logic to configure the portal as affine to the CPU the USDPAA thread is executing on.

QMan Frame Queues (FQs) in SDK 1.1

The allocation of FQs was not coordinated between user-space and kernel-space. Kernel-space FQ allocations would, by default, acquire FQIDs from buffer pool zero, which was statically seeded (via device-tree entries) with a range of values from 0x100 (255) to 0x1ff (511) inclusive, though this behaviour could be overridden by the presence of a "fsl,fqid-range" node in the device-tree which would bypass the buffer pool allocator and instead implemented a software allocator using the given range.

User-space FQ allocations on the other hand always used a software allocator implementation whose range was hard-coded (in source code) to be from 0x200 (512) to 0x3ff (1023).

QMan FQs in SDK 1.2

In kernel-space, support for using buffer pool zero as a special case for FQ allocations has been entirely removed. Now, FQ allocations are only possible if the device-tree contains a "fsl,fqid-range" node. There are device-tree include files (arch/powerpc/boot/dts/fsl/qorIQ-dpaa-res*.dtsi) that declare default allocation ranges.

User-space FQ allocations are now always handled by using the resource allocation ioctl() commands in the USDPAA "process" driver. I.e. user-space FQ allocations are sourced from the allocator residing in kernel-space, and so multiple user-space processes and kernel code are using a common allocator.

QMan Congestion Group Records (CGRs) for SDK 1.1

There was no facility at all for providing allocation of CGRs, and indeed there was no knowledge on the part of kernel or user-space as to how many CGRs were physically present in the hardware – any CGR-dependent software would simply be making its own assumptions about what CGRIDs were safe to use relative to the hardware and any other CGR-dependent software.

QMan CGRs for SDK 1.2

The kernel now implements a CGRID allocator which is seed by a “fsl,cgrid-range” node in the device-tree. The user-space has the same API, and its allocations are routed in the kernel via the “process” driver in the same way as was mentioned for frame queues.

QMan Pool Channels in SDK 1.1

There was no facility for providing allocations of pool-channels, however the device-tree did represent how many pool-channels were present in hardware (this was primarily to allow network devices to be statically configured to use particular pool-channels via device-tree linkage). The kernel-space driver could then enforce its knowledge of what pool-channels were available, but did not coordinate the resource so independent entities of software would need to avoid conflicts via its own means. The user-space driver also used the device-tree to determine the physically-available pool-channels, and provided no coordination of the resources.

QMan Pool Channels in SDK 1.2

The use of device-tree linkage between network nodes and pool-channels in previous versions is gone. The kernel network driver has been adjusted to dynamically allocate its pool-channel instead. As with the other resource types, the USDPAA driver now has the same pool-channel allocation API as the kernel, with user-space allocation operations going via the “process” driver to be handled by the allocator in the kernel.

BMan Buffer Pools in SDK 1.1

The kernel used to determine the number of buffer pools available by looking at the SoC version. Some buffer pools would be represented by device-tree nodes in order to support device-tree linkage with network nodes and in doing so could optionally be seed with ranges of values specified by node properties. The kernel would, by default, implement a BPID allocator that would automatically include all physically available buffer pools that were not explicitly mentioned in device-tree nodes (ie. device-tree nodes acted as reservations against being allocated). An optional “fsl,bpool-range” node could be used to override this behaviour, implementing a software allocator in the same way as “fsl,fqid-range” does.

BMan Buffer Pools in SDK 1.2

If the device-tree contains a “fsl,bpid-range” node (previously named “fsl,bpool-range”). There are device-tree include files

```
arch/powerpc/boot/dts/fsl/qorIQ-dpaa-res*.dtsi
```

that declare default allocation ranges. User-space FQ allocations are now always handled by using the resource allocation ioctl() commands in the USDPAA “process” driver. ie. user-space FQ allocations are sourced from the allocator residing in kernel-space, and so multiple user-space processes and kernel code are using a common allocator.

Changes for FMan Resources

The only change to FMan resource management caused by the SDK 1.2 changes for USDPAA multi-process support is the removal of statically-assigned pool channels for ethernet interfaces. The “dpaa_eth” kernel driver now dynamically allocates a pool-channel during initialisation, and uses it for all the network interfaces it instantiates.

USDPAA DMA Memory for SDK 1.1

The kernel driver used an early-boot hook to reserve a memory region of a hard-coded size (configurable at the expense of a kernel recompile), and the USDPAA “dma_mem” driver would open the “/dev/fsl_usdpaa_shmem” device on behalf of the unique application process and mmap() all the of reserved memory. It was not possible to map less memory than that, it was not possible to create named/shared mappings, and so this was one of the reasons it was not possible to run multiple USDPAA processes.

USDPAA DMA Memory for SDK 1.2

The kernel driver now requires a boot-argument (“usdpaa_mem=<size>[,<num_tlb1>”) to trigger the reservation of USDPAA memory early during the kernel boot, otherwise no memory is reserved. The reservation of multiple TLB1 indices for use by USDPAA is also possible by passing a comma-separated argument. Using multiple TLB1 indices allows simultaneous mappings from USDPAA processes to DMA regions without any risk of fault handling overheads (note that if two processes map the same shared region, that requires 2 TLB1 indices). See section 2.2.3 for information on the device that exposes this memory to mapping from USDPAA applications.

7.3.2.4 BMan and QMan API

SDK 1.2 USDPAAs processes performs resource management through the /dev/fsl_usdpaa file-descriptor .

BMan Modifications - fsl_bman.h

The changes described in this section apply to the BMan API in kernel-space and user-space unless otherwise specified. For more information on specific APIs (eg. the “partial” parameter to allocation functions or API return values) please see the comments in the header file that declares the interface (or consult the API reference manual).

Removal of “recovery” API

bman_recovery_cleanup_bpid() and bman_recovery_exit() have been removed, as they were never more than non-functional stubs and were conflicting with ongoing development.

New BPID allocation API

```
int bman_alloc_bpid_range(u32 *result, u32 count, u32 align, int partial);
static inline int bman_alloc_bpid(u32 *result)
{
    int ret = bman_alloc_bpid_range(result, 1, 0, 0);
    return (ret > 0) ? 0 : ret;
}

void bman_release_bpid_range(u32 bpid, unsigned int count);
static inline void bman_release_bpid(u32 bpid)
{
    bman_release_bpid_range(bpid, 1);
}
```

QMan Modifications - fsl_qman.h

The changes described here apply to the QMan API in kernel-space and user-space unless otherwise specified.

Removal of “recovery” API for SDK 1.2

qman_recovery_cleanup_fq() and qman_recovery_exit() have been removed, as they were never more than non-functional stubs and were conflicting with ongoing development.

Removal of Buffer-Pool Based FQ Allocator API

qm_fq_new(), qm_fq_free(), and the QM_FQ_FREE_* flags have been removed, as they were rendered unnecessary and awkward, in particular as they used to support “wait” options that were useful when deallocating FQIDs to a buffer pool but have no sane interpretation with the software-implemented allocator.

Removal of 'struct qman_portal_config::has_stashing' for SDK 1.2

Stashing can now be assumed as enabled in all environments (kernel-space, user-space, with Linux running natively, under the topaz hypervisor, or under KVM), so support for stashing-disabled operation has been removed for the sake of optimisation.

Removal of 'struct qman_fq_cb::dc_ern' for SDK 1.2

The callbacks associated with a frame queue object no longer support a DC_ERN handler (as these never worked properly because the concept is fundamentally unworkable). The lowest level at which DC_ERN messages can meaningfully be handled is at the portal level.

New API for handling DC_ERNs in SDK 1.2

It is possible to register a handler for DC_ERN messages with the portal affine to the running CPU, or as a global fallback for any portals that don't have their own handler.

```
void qman_set_dc_ern(qman_cb_dc_ern handler, int affine);
```

Removal of “NULL FQ” API in SDK 1.2

qman_get_null_cb(), qman_set_null_cb(), and QMAN_INITFQ_FLAG_NULL flag have been removed, as this functionality was considered marginal, had no known use-case, and was conflicting with ongoing development.

New API to Obtain Portal Channel for SDK 1.2

As portals are dynamically allocated, initialised, and assigned to CPUs during boot up, it became necessary for a user to be able to determine the channel ID for the portal associated with a given CPU, eg. in order to schedule frame queues such that dequeues would be handled on that CPU core.

```
enum qm_channel qman_affine_channel(int cpu);
```

New Pool-Channel Allocation API for SDK 1.2

For QMan pool-channel allocation:

```
int qman_alloc_pool_range(u32 *result, u32 count, u32 align, int partial);
static inline int qman_alloc_pool(u32 *result)
{
    int ret = qman_alloc_pool_range(result, 1, 0, 0);
    return (ret > 0) ? 0 : ret;
}

void qman_release_pool_range(u32 id, unsigned int count);
static inline void qman_release_pool(u32 id)
{
    qman_release_pool_range(id, 1);
}
```

New QMan CGR allocation API for SDK 1.2

```
int qman_alloc_cgrid_range(u32 *result, u32 count, u32 align, int partial);
static inline int qman_alloc_cgrid(u32 *result)
{
    int ret = qman_alloc_cgrid_range(result, 1, 0, 0);
    return (ret > 0) ? 0 : ret;
}

void qman_release_cgrid_range(u32 id, unsigned int count);
static inline void qman_release_cgrid(u32 id)
{
    qman_release_cgrid_range(id, 1);
}
```

7.3.2.5 USDPAA Thread and Global API

The API changes described apply to user-space (USDPAA).

fsl_d.h -- Thread Initialization simplified

The SDK 1.1 Thread initialization was verbose:

```
int qman_thread_init(int cpu, int recovery_mode); /* remove for SDK 1.2 */
int bman_thread_init(int cpu, int recovery_mode); /* remove for SDK 1.2 */
int qman_thread_init(void);
int bman_thread_init(void);
```

The SDK 1.2 Thread initialization is compact:

```
int qman_thread_init(void);
int bman_thread_init(void);
```

Recovery support (which was non-functional) has been removed so 'recovery_mode' is no longer a parameter. As for the 'cpu' parameter, portals are now dynamically bound to CPUs, so these functions will allocate any unused portal and it will be automatically bound to the CPU on which the caller is executing.

fsi_d.h -- Global Initialization

The SDK 1.1 Global initialization was verbose:

```
int qman_global_init(int recovery_mode);           /* remove for SDK 1.2 */
int bman_global_init(int recovery_mode);          /* remove for SDK 1.2 */
int qman_global_init(void);
int bman_global_init(void);
```

The SDK 1.2 Global initialization is compact:

```
int qman_global_init(void);
int bman_global_init(void);
```

As before, the 'recovery_mode' parameter is removed because the non-functional recovery interfaces have been removed.

7.3.2.6 USDPAA DMA API

The API changes described apply to DMA user-space (USDPAA).

dma_mem.h

The key change to this interface is that there can now be more than one DMA region available to each USDPAA process, so there is support for creating multiple such mappings, and as such most of the functions require that the DMA region be supplied as a parameter where there was no such need before.

Setup, or creation of DMA maps

The dma_mem driver used to initialise in a parameterless manner, creating the unique DMA map via dma_mem_setup(void), which has been removed. Instead, maps are created by the application using the following interfaces:

```
struct dma_mem;
#define DMA_MAP_FLAG_SHARED    0x01
#define DMA_MAP_FLAG_ALLOC    0x08
#define DMA_MAP_FLAG_NEW      0x02
#define DMA_MAP_FLAG_LAZY     0x04
#define DMA_MAP_FLAG_READONLY 0x10
struct dma_mem *dma_mem_create(uint32_t flags, const char *map_name,
                               size_t len);
```

The significance of the flags is described in more detail within the dma_mem.h header. To summarise, the SHARED flag creates a mapping to a new or existing DMA region that can be mapped by multiple USDPAA processes ('map_name' is the identifier for the region), otherwise a new region and mapping created that remains private to the process. If NEW is not specified the DMA region must already exist, whereas if NEW is specified the region must not already exist unless LAZY is also specified. LAZY refers to "lazy initialisation," meaning that multiple processes can independently issue the same API call with the same name and specifying both the NEW and LAZY flags, with the result being that the named region will be allocated only once (by whichever process "wins the race") and mapped into all the requesting processes.

If ALLOC is specified, then all processes that map the same region can use the dma_mem_memalign() and dma_mem_free() interfaces to allocate blocks from the region in a coordinated way. Without ALLOC, the region is created "raw," meaning the user manipulates the entire region directly without any allocator functionality provided by the dma_mem driver.

Raw Memory Regions

If a DMA region and mapping is created with the RAW flag, it can then be accessed via;

```
void *dma_mem_raw(struct dma_mem *map, size_t *len);
```

Memory allocation

These functions are similar to those in SDK 1.1, with the exception that they require a parameter to indicate which DMA map to use. For SDK 1.1:

```
void *dma_mem_memalign(size_t boundary, size_t size);    /* SDK 1.1 version */  
void dma_mem_free(void *ptr, size_t size);              /* SDK 1.1 version */
```

Additional parameter for SDK 1.2:

```
void *dma_mem_memalign(struct dma_mem *map, size_t boundary, size_t size);  
void dma_mem_free(struct dma_mem *map, void *ptr);
```

Distinguishing DMA regions

```
struct dma_mem *dma_mem_findv(void *v);  
struct dma_mem *dma_mem_findp(dma_addr_t p);
```

Physical/virtual address conversion

As with memory allocation, these functions require a parameter to indicate which DMA map to use, but are otherwise similar to those in SDK 1.1. In the case where the DMA map is not known, use the functions mentioned in “Distinguishing DMA regions” first. Note, these functions are actually implemented as inlines with some nasty details involving casts that should be ignored, this is simply because these routines are performance critical in packet-processing processing;

```
static inline void *dma_mem_ptov(struct dma_mem *map, dma_addr_t p) { ... }  
static inline dma_addr_t dma_mem_vtop(struct dma_mem *map, void *v) { ... }
```

Legacy interfaces, “dma_mem_generic”

In order to facilitate porting of legacy applications to the new dma_mem API, the following mechanism is provided. A global variable within the dma_mem driver, dma_mem_generic, is NULL by default but can be set by the application once it has created a DMA mapping. From that point on, it could use the following __dma_mem_*() functions, which do not require a DMA map parameter.

```
extern struct dma_mem *dma_mem_generic;  
static inline void *__dma_mem_ptov(dma_addr_t p)  
{  
    return dma_mem_ptov(dma_mem_generic, p);  
}  
static inline dma_addr_t __dma_mem_vtop(void *v)  
{  
    return dma_mem_vtop(dma_mem_generic, v);  
}  
static inline void *__dma_mem_memalign(size_t boundary, size_t size)  
{  
    return dma_mem_memalign(dma_mem_generic, boundary, size);  
}  
static inline void __dma_mem_free(void *ptr)  
{  
    return dma_mem_free(dma_mem_generic, ptr);  
}
```

That is, in order to port a legacy application (which worked on the assumption of there being a unique, canonical DMA mapping for all DMA operations), it should suffice to;

1. During application initialisation, create a default DMA map using dma_mem_create(), and assign that to dma_mem_generic,
2. Change all the legacy dma_mem_*() calls in the application to __dma_mem_*().

7.3.2.7 USDPAAs netcfg.h

The API changes described apply to user-space (USDPAAs).

For SDK 1.1 and previous versions:

```
struct usdpaa_netcfg_info {
    uint8_t num_cgrids;           /* SDK 1.1 and previous */
    uint32_t *cgrids;           /* SDK 1.1 and previous */
    uint8_t num_pool_channels;   /* SDK 1.1 and previous */
    enum qm_channel *pool_channels; /* SDK 1.1 and previous */
    uint8_t num_ethports;       /* Number of ports */
    [...]
```

For SDK 1.2 and subsequent versions:

```
struct usdpaa_netcfg_info {
    uint8_t num_ethports;       /* Number of ports */
    [...]
```

'struct usdpaa_netcfg_info' no longer specifies CGR and pool-channel resources to applications. There are now allocators in the QMan API for both these resource types, and they no longer need to come from the network configuration. Note that these resources were previously coming from hard-coded work-arounds if present at all.

See the "USDPAAs Resource Management Modifications for SDK 1.2" topic for details on CGRs and QMan Pool Channels.

7.3.2.8 Kernel configuration

Kconfig settings for the fsl_qbman driver have changed.

SDK 1.2 Kconfig changes:

- CONFIG_FSL_DPA_HAVE_IRQ is removed, IRQ support is always enabled.
- CONFIG_FSL_BMAN_PORTAL is removed, support for BMan portals is always enabled.
- CONFIG_FSL_QMAN_PORTAL is removed, support for QMan portals is always enabled.
- CONFIG_FSL_QMAN_PORTAL_DISABLEAUTO_DCA is removed, QMan portals are always enabled for DCA consumption of DQRR (dequeue response ring) entries.
- CONFIG_FSL_QMAN_NULL_FQ_DEMUX is removed, see the BMan and QMan API Modifications topic, Removal of "NULL FQ" API heading.
- CONFIG_FSL_QMAN_DQRR_PREFETCHING is removed, the driver is now optimised to always assume stashing is always enabled, so support for pre-fetching is removed. This removes a run-time check from the critical path.

7.3.2.9 Device Tree (Excluding QMan/BMan Resource Ranges)

Device tree changes, excluding the QMan/BMan resource ranges.

QMan/BMan portals

Portals no longer have "fsl,usdpaa-portal" or "cpu-handle" properties.

```
qportal1: qman-portal@4000 {
    cell-index = <0x1>;
    compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";
    reg = <0x4000 0x4000 0x101000 0x1000>;
    interrupts = <106 0x2 0 0>;
    fsl,qman-channel-id = <0x1>;
    cpu-handle = <&cpu1>;
```

```
};
```

BPID 0 FQ-allocator

No buffer pool device tree node for BPID 0, because we no longer support that legacy mechanism for FQID allocation.

QMan Pool channels

Pool channel nodes have been removed and replaced by “pool channel range” nodes.

BMan buffer pools

Buffer pool nodes have not been removed, because they are still linked to by network-related nodes. However they are now ignored by the fsl_qbman driver and so no longer contain the “fsl,bpool-cfg” property type. Eventually, network configuration will obtain buffer pools dynamically, at which point there should be no more need for individual buffer pool nodes in the device-tree. (There is no current plan for when this deprecation will occur.)

Ethernet Interfaces

Ethernet nodes no longer have “fsl,qman-channel” properties linking them to pool channels.

```
ethernet@2 {  
    compatible = "fsl,p4080-dpa-ethernet", "fsl,dpa-ethernet";  
    fsl,fman-mac = <&enet2>;  
};
```

7.3.2.10 Device Tree (QMan/BMan Resource Ranges)

These device tree resource properties share a common format.

QMan and BMan Portals

Portals no longer have “fsl,usdpaa-portal” or “cpu-handle” properties. For SDK 1.1 and previous:

```
qportal1: qman-portal@4000 {  
    cell-index = <0x1>;  
    compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";  
    reg = <0x4000 0x4000 0x101000 0x1000>;  
    interrupts = <106 0x2 0 0>;  
    fsl,qman-channel-id = <0x1>;  
    fsl,usdpaa-portal; /* SDK 1.1 only */  
    cpu-handle = <&cpu1>;  
    fsl,qman-pool-channels = <&qpool4 &qpool5 &qpool6  
                            &qpool7 &qpool8 &qpool9  
                            &qpool10 &qpool11 &qpool12  
                            &qpool13 &qpool14 &qpool15>; /*SDK 1.1 only */  
};
```

For SDK 1.2 and later versions:

```
qportal1: qman-portal@4000 {  
    cell-index = <0x1>;  
    compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";  
    reg = <0x4000 0x4000 0x101000 0x1000>;  
    interrupts = <106 0x2 0 0>;  
    fsl,qman-channel-id = <0x1>;  
    cpu-handle = <&cpu1>;  
};
```

BPID 0 FQ-allocator

No buffer pool device tree node for BPID 0, because we no longer support that legacy mechanism for FQID allocation. See section 3.2.

QMan Pool Channels

Pool channel nodes have been removed (and replaced by “pool channel range” nodes, see the corresponding item below). See section 3.4.

BMan Buffer Pools

Buffer pool nodes have not been removed, because they are still linked to by network-related nodes. However they are now ignored by the fsl_qbman driver and so no longer contain the “fsl,bpool-cfg” property type. Eventually, network configuration will obtain buffer pools dynamically, at which point there should be no more need for individual buffer pool nodes in the device-tree. There is no precise plan for when this deprecation will occur though.

QMan and BMan Resource Ranges (Allocation)

The following resource properties share a common format, with a 2-tuple specifying a base+count pair. Eg. if the pool-channel range property specifies “<0x21 0xf>”, that corresponds to a range of pool channel IDs ranging from 33 (0x21) to 47 (0x21+0xf-1), inclusive.

FQID

“FQID range” nodes have been added to specify FQs that are available for dynamic allocation. These do not yet include all the FQs that are available in the system, because there are still some legacy requirements for pre-configured FQIDs that have not been updated to use dynamic allocation. These nodes were previously supported but were not enabled by the default device trees, whereas they are now used in all cases, via the including of arch/powerpc/boot/dts/fsl/qoriq-dpaar-es*.dtsti files (which did not exist in SDK 1.1) as follows:

```
qman-fqids@0 {
    compatible = "fsl,fqid-range";
    fsl,fqid-range = <256 256>;
};
```

CGRID Dynamic Allocation

“CGRID range” nodes have been added to specify CGRs that are available for dynamic allocation. Example:

```
qman-cgrids@0 {
    compatible = "fsl,cgrid-range";
    fsl,cgrid-range = <0 256>;
};
```

Pool Channel

“Pool channel range” nodes have been added to specify pool channels that are available for dynamic allocation. Example:

```
qman-pools@0 {
    compatible = "fsl,pool-channel-range";
    fsl,pool-channel-range = <0x21 0xf>;
};
```

BPID

“BPID range” nodes have been added to specify buffer pools that are available for dynamic allocation. Example:

```
bman-bpids@0 {
    compatible = "fsl,bpid-range";
    fsl,bpid-range = <32 32>;
};
```

7.3.2.11 USDPAA Boot Arguments

Without this boot-argument, no memory is reserved by the “fsl_usdpaa” driver and so no memory will be available for creating DMA mappings in USDPAA applications.

usdpaa_mem

The usdpaa_mem argument is not set by default by u-boot, device-trees or any other resource. The user must add it explicitly to boot commands in order to be able to run USDPAA applications. The size can be expressed using the standard suffixes for memory size notation (“256M”, “1G”, etc).

A second (and optional) argument allows multiple TLB1 indices to be reserved for mapping regions within the memory reservation. Without this extra argument, the default behaviour is to reserve only 1 TLB1 entry. I.e. “usdpaa_mem=64M,1” is equivalent to “usdpaa_mem=64M”. Note that the handling of page faults for software access to these resources will be satisfied by using the reserved TLB1 entries in a round-robin fashion, so if there are more mappings between user-space processes and DMA regions than there are TLB1 entries, and all of those mappings are being actively used at run-time, then performance will degenerate.

Since single TLB1 entries can only map power of 4 memory sizes a set of 1 or more TLB1 entries is required in to map a DMA region into each process. The USDPAA driver will use the minimum number of TLB1 entries possible to map each DMA region.

If a single process maps to two DMA regions, that would require two sets of TLB1 entries. But the same is true if two distinct processes map to one private DMA region each. So the number of TLB1 entries required is really the number of pairings between user-space processes and distinct DMA regions, or “the number of mappings”, to put it another way. The total number of TLB1 entries available depends on the SoC version, so one should check this when determining how many TLB1 entries to dedicate to USDPAA.

Note that a typical kernel would internally only use 3 or so TLB1 entries, and all the remaining entries are normally reserved by the “hugetlb” driver for a similar kind of large-page fault-handling algorithm as that implemented for USDPAA – indeed, the reason USDPAA does not just use “hugetlb” directly is that it has no mechanisms to support user-space obtaining physical addresses and performing DMA.

One must also determine whether HugeTLB support is required. This decision determines the number of TLB1 entries for USDPAA use and, in turn, how many processes and active DMA mappings used. The USDPAA could use most of the TLB1 entries if HugeTLB support is not used.

"qportals" and "bportal;s"

By default, the kernel will attempt to allocate portals for each online core. If however the “qportals” (for QMan portals) and/or “bportals” (for BMan portals) boot-arguments are specified, this behaviour will be overridden. These boot-arguments specify cores that should be assigned portals to them, with the implication being that cores that are not specified will need to “slave” off the cores that are assigned.

Table 100. boot-argument: "qportals=1,s3-4"

Core	Portal	Association
0	B	slave
1	A	affine unshared
2	C	slave
3	B	affine shared
4	C	affine shared
5	B	slave
7	B	slave

7.3.2.12 USDPA Virtualisation and Partitioning

In a partitioned system, it is necessary to assign each partition non-conflicting subsets of the hardware resources.

All the resources mentioned in section 3 can and should be divided up, with each instance of Linux receiving distinct portals and other dynamically-allocated resources via their respective device-trees.

Some use-cases may intentionally share resources between partitions, in which coordination of the corresponding resource IDs is up to the application. The resources provided to the partitions by the device-tree are inherently managed by the drivers themselves, and these must be mutually-exclusive because the drivers in the distinct partitions have no innate coordination.

7.3.2.13 Multi-process PPAC Applications

A description of how the networking applications are affected by, and have been adapted for, multi-process support.

The USDPA toolkit provides a template called “PPAC” (Packet Processing Application Core) for simple networking applications, together with some applications called “PPAM”s (Packet Processing Application Module) that are written on top of this template. These networking applications notably include “reflector” and “ipfwd”.

Running multiple distinct PPAM application processes (or “instances” as they are sometimes described) requires that each process have its own dedicated set of FMAN interfaces, buffer pools, and cores.

FMan Interfaces

By default, a PPAM application will automatically configure and use all available FMan interfaces, unless a specific set of interfaces is specified via the “-i” option. For example:

```
<app1> -i fm1-10g, fm2-10g
<app2> -i fm2-gb2, fm2-gb3
```

The names of FMAN interfaces that can be used with “-i” option are as follows.

Table 101. FMan “-i” Options

FMAN1	Fman2
fm1-gb0	fm2-gb0
fm1-gb1	fm2-gb1
fm1-gb2	fm2-gb2
fm1-gb32	fm2-gb3
fm1-gb4	fm2-gb4
fm1-10g	fm2-10g

As per SERDES protocol, a set of FMAN interfaces can be chosen to run with an application.

Buffer pool restrictions

The device tree specifies the ethernet nodes such that each has a set of buffer pools for FMan to use when receiving frames. One restriction of the PPAC multi-process support is that the buffer pools used by FMan interfaces in one process must not also be used by any FMan interfaces in any other process. As such the device tree may need to be adjusted to ensure that any reuse of a buffer pool by more than one FMan interface must only occur when those FMan interfaces will always belong

to the same process. For example, if one application wants to use fm1-10g and fm2-10g and another application is going to use fm2-gb2 and fm2-gb3, then corresponding ethernet nodes might look like:

```
ethernet@4 {
    compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
    fsl,bman-buffer-pools = <&bp9>;
    fsl,qman-frame-queues-rx = <0x5a 1 0x5b 1>;
    fsl,qman-frame-queues-tx = <0x7a 1 0x7b 1>;
};

ethernet@9 {
    compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
    fsl,bman-buffer-pools = <&bp9>;
    fsl,qman-frame-queues-rx = <0x66 1 0x67 1>;
    fsl,qman-frame-queues-tx = <0x86 1 0x87 1>;
};

ethernet@7 {
    compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
    fsl,bman-buffer-pools = <&bp8>;
    fsl,qman-frame-queues-rx = <0x60 1 0x61 1>;
    fsl,qman-frame-queues-tx = <0x80 1 0x81 1>;
};

ethernet@8 {
    compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
    fsl,bman-buffer-pools = <&bp8>;
    fsl,qman-frame-queues-rx = <0x62 1 0x63 1>;
    fsl,qman-frame-queues-tx = <0x82 1 0x83 1>;
};
```

Seeding buffer pools

Each PPAC-based application process will initialise the (usually 3) buffer pools used by each FMan interface that belongs to it. (If a buffer pool is used by more than one interface, it will only be initialised once.) By default the number of buffers to allocate for the triplet of pools used by an FMan interface is 0 for the first two pools and 1728 for the third. The default allocation triplet can be overridden via the “-b” option. For example, to continue the earlier example of a two-process scenario, and to have each process allocate 1600 buffers for the first pool used by any network interface and 0 for the second and third pools, they would use the following arguments :

```
<app1> -b 1600:0:0 -i fm1-10g, fm2-10g
<app2> -b 1600:0:0 -i fm2-gb2, fm2-gb3
```

Cores

By default, each PPAC-based application process will start a single thread running on core 1. Additional threads can be started and stopped on arbitrary cores using the interactive CLI, but the first/primary thread can not be removed without ending the process. So for multi-process scenarios, it is better for each application instance to specify a core or set of cores as part of the command line. For example, to split 8 cores in half between our two hypothetical application processes;

```
<app1> 0..3 -b 1600:0:0 -i fm1-10g, fm2-10g
<app2> 4..7 -b 1600:0:0 -i fm2-gb2, fm2-gb3
```

7.3.2.14 Limitations

The use of dynamic allocation of resources does not address the appropriate state of an allocation.

A resource that is deallocated by one user can subsequently be allocated by another and it will be in the same state it was left, for better or worse. As such, stability and integrity of (and between) datapath applications is entirely a matter of cooperation. Future releases will implement measures to quiesce and recover such resources to make the system more robust in the face of individual application failures.

For the reasons explained above, most types of resources can be leaked by USDPAAs applications that exit (or crash) before deallocating them. Although an application can explicitly deallocate a resource in a bad state, the risk of a resource being in a bad state when an application exits without having deallocated it is considered too great – so it is leaked rather than allowing it lead to undefined behaviour in future uses.

An exception to the last comment is for QMan and BMan portals, which due to them being UIO-based means they are implicitly “deallocated” when a process exits. In a future release, portals will likely no longer be UIO-based, and in any case, they will likely be “cleaned up” on process-exit.

7.4 USDPAAs Applications

7.4.1 USDPAAs ceetm Demo User Guide

7.4.1.1 Introduction

This document describes the usage of "ceetm_demo" application which is built on USDPAAs PPAC architecture. User can experience the functionality of QMan CEETM with this application and can also learn how to use user space CEETM driver API from source code of this application.

The concept of QMan CEETM is beyond scope of this document. The user should refer to QMan user manual for this part.

7.4.1.2 Overview of ceetm demo

ceetm_demo is an USDPAAs application built on PPAC architecture. It works in a way like reflector – the traffic sent to one ethernet interface will be reflected back from the same interface. Please refer to reflector user guide for details. Besides, ceetm_demo enables traffic management on Tx side and differentiates traffic flows according to TOS value in IPv4 header.

In order to provide flexibility at most, we use a xml-format configuration file to depict configuration of CEETM which can be parsed by ceetm_demo at initialization phase. User can experience different cases of traffic management just by changing configuration file before launching ceetm_demo.

If users want to learn how to utilize CEETM driver APIs for their own case, please see function 'net_if_ceetm_init' in ceetm_demo.c as reference.

7.4.1.3 Features of ceetm demo

ceetm_demo application has following features that provide flexibility to user to experience functionality of QMan CEETM:

- Shaping on LNI and Channel is configurable
- Number of CQs and weighted groups(A and B) is configurable
- SP CQ and Group can be configured for CR/ER eligible or both
- The weight of CQ is configurable
- Distinguish flows by TOS field in IPv4 protocol which is used to identify unique CQ on egress

7.4.1.4 CEETM use case

CEETM configuration may vary from case to case. Here we only show the way to implement one case which is described in QMan 3.x document. Other scenarios could be easily implemented by changing configuration file only.

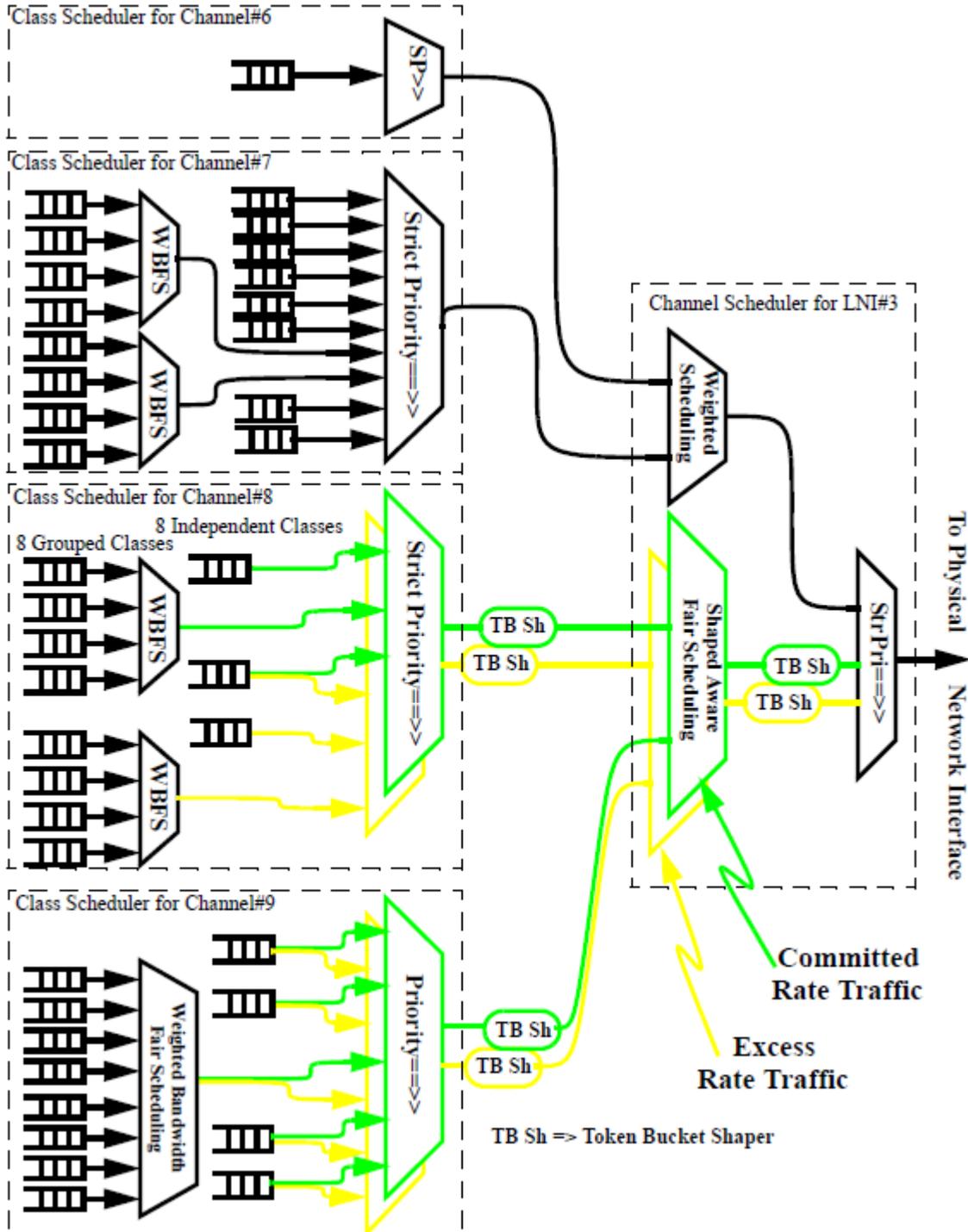


Figure 119. CEETM use case

This scenario depicts the following:

- Channels #6,#7,#8 and #9 have been configured to be scheduled by the channel scheduler for LNI#3 (i.e. all packets from these channels are directed via LNI#3 to the physical network interface coupled by configuration to LNI#3).
- Channels #6 and #7 have been configure to be “unshaped”. Packets from these channels will not be subjected to shaping at the channel level and will feed the top priority level within the LNI which is also not subjected to shaping. Their class schedulers will not distinguish between CR and ER opportunities.

- Channels #8 and #9 have been configured to be “shaped”. Their class schedulers will distinguish between CR and ER opportunities. The CR/ER packets to be sent from each channel shall be subjected to a pair of CR/ER token bucket shapers specific to that channel. The aggregate of CR/ER packet from these channels shall be subject to a pair of CR/ER token bucket shapers specific to LNI#3.
- Channels #6 has only one class in use. That class queue will behave as if it were a channel queue and a peer to channel #7. Unused classes do not have to be configured as such - simply not used.
- Channel #7 has all 16 classes in use
 - The group classes have been configured as 2 groups (A and B) of 4 classes.
 - The priority of the groups A and B have both been set to be immediately below independent class 5. In a case of similar configuration group A has higher priority than group B.
- Channels #8 has 3 independent classes and 2 groups of 4 grouped classes in use.
 - The priorities of the class groups A and B have been set to be immediately below independent class 0 and class 2 respectively.
 - Independent class 0 and class group A have been configured to request and fulfill only CR packet opportunities.
 - Independent class 1 has been configured to request and fulfill both CR and ER packet opportunities.
 - Independent class 2 and class group B have been configured to request and fulfill only ER packet opportunities.
- Channels #9 has 4 independent classes and 1 groups of 8 grouped classes in use.
 - The group classes have been configured as 1 group (A) of 8 classes.
 - All independent classes and the class group (A) have been configured to request and fulfill both CR and ER packet opportunities.

7.4.1.5 CEETM configuration file

CEETM configuration file is used to depict a model of traffic management in xml format which can then be parsed by ceetm_demo application. User could experience the effect of dual-shaper and dual-scheduler of CEETM by changing this configuration file.

```
<ceetm>
  <lmi control="shaped" cr="2g" er="2g">
    <channel control="unshaped" group="0">
      <cq idx='0' />
    </channel>
    <channel control="unshaped" group="2">
      <groupA idx="6" />
      <groupB idx="6" />
      <cq idx='0' />
      <cq idx='1' />
      <cq idx='2' />
      <cq idx='3' />
      <cq idx='4' />
      <cq idx='5' />
      <cq idx='6' />
      <cq idx='7' />
      <cq idx="8" weight="1" />
      <cq idx="9" weight="1" />
      <cq idx="10" weight="1" />
      <cq idx="11" weight="1" />
      <cq idx="12" weight="1" />
      <cq idx="13" weight="2" />
      <cq idx="14" weight="4" />
      <cq idx="15" weight="8" />
    </channel>
  </lmi>
</ceetm>
```

```
</channel>
<channel control="shaped" group="2" cr="500m" er="50m">
  <groupA idx="1" op="cr" />
  <groupB idx="3" op="er" />
  <cq idx='0' op="cr" />
  <cq idx='1' op="both" />
  <cq idx='2' op="er" />
  <cq idx="8" weight="1" />
  <cq idx="9" weight="1" />
  <cq idx="10" weight="1" />
  <cq idx="11" weight="1" />
  <cq idx="12" weight="1" />
  <cq idx="13" weight="1" />
  <cq idx="14" weight="1" />
  <cq idx="15" weight="1" />
</channel>
<channel control="shaped" group="1" cr="250m" er="100m">
  <groupA idx="2" op="both" />
  <cq idx='0' op="both" />
  <cq idx='1' op="both" />
  <cq idx='2' op="both" />
  <cq idx='3' op="both" />
  <cq idx="8" weight="1" />
  <cq idx="9" weight="2" />
  <cq idx="10" weight="4" />
  <cq idx="11" weight="8" />
  <cq idx="12" weight="16" />
  <cq idx="13" weight="32" />
  <cq idx="14" weight="64" />
  <cq idx="15" weight="128" />
</channel>
</lni>
</ceetm>
```

- <ceetm>: Root element
- <lni>
 - control: 'shaped' if shaping is placed on this interface; 'unshaped' otherwise
 - cr: bandwidth for Commit Rate of interface
 - er: bandwidth for Excess Rate of interface
- <channel>
 - control: 'shaped' if shaping is placed on this interface; 'unshaped' otherwise
 - grou: '0' – no group; '1' – only groupA enabled; '2' – both groupA/B are enabled
 - cr: bandwidth for Commit Rate of channel
 - er: bandwidth for Excess Rate of channel
- <cq>
 - idx: CQ id, ranging from 0 to 15
 - op: 'cr' – cr eligible; 'er' – er eligible; or 'both'. Only applicable for SP CQ
 - weight: the weight of CQ. Only applicable for grouped CQ

74.1.6 Running ceetm_demo

User should boot board with USDPAAs mode. After logging into the board as 'root', run following command

```

login: root
Password:
root@t2080qds:~# cd /usr/etc
root@t2080qds:/usr/etc# fmc -c usdpaa_config_t2_serdes_66_16.xml -p
usdpaa_policy_hash_ipv4.xml -a
root@t2080qds:/usr/etc# ceetm_demo -c usdpaa_config_t2_serdes_66_16.xml -p
usdpaa_policy_hash_ipv4.xml -f ceetm_cfg.xml
Found /fsl,dpaa/dpa-fman0-oh@2, Tx Channel = 80a, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@0, Tx Channel = 802, FMAN = 0, Port ID = 2
Found /fsl,dpaa/ethernet@1, Tx Channel = 803, FMAN = 0, Port ID = 3
Found /fsl,dpaa/ethernet@2, Tx Channel = 804, FMAN = 0, Port ID = 2
Found /fsl,dpaa/ethernet@3, Tx Channel = 805, FMAN = 0, Port ID = 3
Found /fsl,dpaa/ethernet@8, Tx Channel = 800, FMAN = 0, Port ID = 0#
Found /fsl,dpaa/ethernet@9, Tx Channel = 801, FMAN = 0, Port ID = 1
Configuring for 4 network interfaces
Allocated DMA region size 0x1000000
Released 0 bufs to BPID 7
Released 0 bufs to BPID 8
Released 8192 bufs to BPID 9
Thread uid:0 alive (on cpu 1)
ceetm> add 2..7
Thread uid:1 alive (on cpu 2)
Thread uid:2 alive (on cpu 3)
Thread uid:3 alive (on cpu 4)
Thread uid:4 alive (on cpu 5)
Thread uid:5 alive (on cpu 6)
Thread uid:6 alive (on cpu 7)
ceetm>

```

When application starts, only 1 thread runs. User can add more threads by 'add' command to improve throughput, or specify cpu range on which thread runs. For example

```

root@t2080qds:/usr/etc# ceetm_demo 2..7 -c usdpaa_config_t2.xml -p
usdpaa_policy_hash_ipv4.xml -f ceetm_cfg.xml

```

In this case, 6 threads run after application starts.

The PCD configuration file may vary from board to board, here is summary of file name for boards that support CEETM.

```

B4860QDS: usdpaa_config_b4_serdes_0x2a_0x98.xml
T4240QDS: usdpaa_config_t4_serdes_1_1_6_6.xml
T2080QDS: usdpaa_config_t2_serdes_66_16.xml

```

74.1.7 Generate traffic flows

ceetm_demo works in a way like 'reflector'. It enhances traffic differentiation on egress side. The egress flow will be forwarded to class queue according to 'tos' value in IPv4 header.

TOS is a 8-bit value, 4 msb in this case means CHANNEL id within a LNI while 4 lsb means CQ id (0 – 0xf) within a CHANNEL. So if you want to egress traffic flow through CQ 10 in channel 2, the tos vaule should be 0x2a.

User can see the effect of packet scheduling by sending traffic flows with different TOS values.

Currently QMan CEETM supports dual-rate shapers and dual-scheduler on two levels. Please refer to QMan user guide for elaboration about them.

7.4.2 DPAA Offloading Applications Users Guide

7.4.2.1 Introduction

This document supplies several methods for modifying the QorIQ USDPAA offloading applications using the DPA offloading driver API.

The document explores the following target use cases providing detailed and interactive examples. These include:

- USDPAA classifier demo application
- USDPAA IP fragmentation demo application
- USDPAA IP reassembly demo application
- USDPAA IPSec offloading application
- USDPAA IPSec & IP Forwarding offloading application using Network Function Layer

See [Appendix A](#) of the DPAA Offloading Applications User Guide for list of supported processors.

The following table provides a short summary of each example.

Table 102. Summary of USDPAA offloading applications

Application	Description
USDPAA classifier demo application	Provides an example on how to use basic frame manager features such as parse, classify, distribution and header manipulation.
USDPAA IP fragmentation demo application	Illustrates how to configure basic IP fragmentation provided by the Frame Manager. For frame manager version 3 devices the reassembly application also demonstrates the Virtual Storage Profile configuration and selection for reassembled flows.
USDPAA IP reassembly demo application	Explains how to configure basic IP reassembly provided by the Frame Manager. For Frame Manager version 3 devices the reassembly application also demonstrates the Virtual Storage Profile configuration and selection for reassembled flows.
USDPAA IPSec offloading application	Reveals how to use standard Linux tools to configure IPSec offloading using the DPA offloading driver.
USDPAA IPSec & IP forwarding offloading application using Network Function Layer	Demonstrates how IPSec and IP forwarding services can be offloaded using the DPAA offloading <i>Network Function Layer</i>

Before using any of the applications in this document, you need to build and install the DPAA offloading drivers as described in section "[Appendix B - Enabling DPA Offloading Drivers in the Linux Kernel](#)".

7.4.2.2 Altering the classifier_demo application

This application demonstrates how packet classification can be offloaded to the frame manager by using the DPA offloading driver. It shows several functionalities of the classifier, traffic manager(on CEETM capable platforms) and statistics components.

7.4.2.2.1 Overview

The application performs the following operations on the created resources (a set of three DPA Classifier Exact Match tables managed by key and a group of DPA Stats counters):

- Create an Exact Match Table
- Insert entry in an Exact Match Table
- Delete entry by Key in an Exact Match Table
- Flush an Exact Match Table
- Free an Exact Match Table
- Dynamically create header manipulations
- Set up header manipulation on a table entry
- Free the header manipulation associated with a table entry
- Create a DPA Stats class counter for the entire Exact Match Table for each table created
- Retrieve synchronously and asynchronously the DPA Stats counters
- Remove all created DPA Stats counters

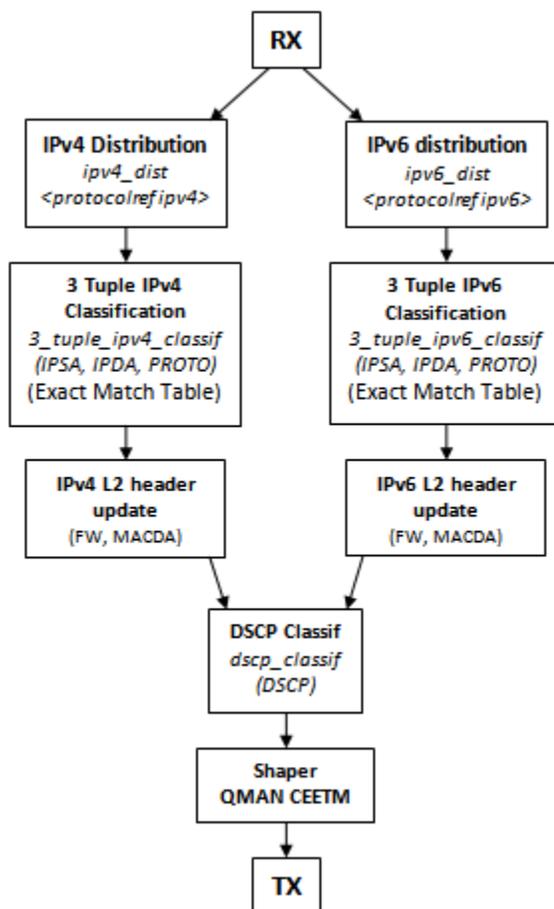


Figure 120. Classifier Demo Application Scheme

The application has defined a number CLI commands, each is meant to validate different functionalities. The user can add or remove classification entries by typing the command “add4” for IPv4, respectively “add6” for IPv6. For removing the inserted entries, the commands “rem4,” or “rem6” must be used in the USDPAA PPAM command line.

When a frame is received, it is first distributed based on protocol, then if the table lookup process results in a HIT condition, the frame is sent to the next classifier table where, based on DSCP value, the frame is forwarded to a logical frame queue corresponding to a traffic manager class queue in case of CEETM capable platforms. At the traffic manager level, based on priorities, the frame is forwarded or not to the TX port. On P platforms the frame is enqueued on the first fqid configured for the TX port.

A header manipulation chain is attached to each of the entries which are inserted in the IPv4/IPv6 Exact Match table. Each header manipulation chain consists of a TTL decrement, a PAT (Port Address Translation) and a forwarding header manipulation (L2 header update). The values in each header manipulation operation are updated at runtime based on the information passed by the user along with the add4/6 command.

During the life cycle of the application, the user has the possibility to perform insert or delete classification keys or to read values of the DPA Stats class counters by entering the following commands in the USDPAA PPAM command line:

Table 103. Application

Command Line	Description
add4 <IPSA><IPDA><PROTO><new-MACDA><new-SPORT><new-DPORT>	dynamically insert a new entry in 3_tuple_ipv4_classif Exact Match table.
add6 <IPSA><IPDA><PROTO><new-MACDA><new-SPORT><new-DPORT>	dynamically insert a new entry in 3_tuple_ipv6_classif Exact Match table.
rem4 <IPSA><IPDA><PROTO>	dynamically remove an entry from 3_tuple_ipv4_classif Exact Match table.
rem6 <IPSA><IPDA><PROTO>	dynamically remove an entry from 3_tuple_ipv6_classif Exact Match table.
get_classif_stats	retrieve the statistics for the created classifier table counters in asynchronous mode and print the returned values.
get_classif_stats_sync	retrieve the statistics for the created classifier table counters in synchronous mode and print the returned values.
get_traffic_stats	retrieve the statistics for the created traffic manager/ethernet counters in asynchronous mode and print the returned values.
get_traffic_stats_sync	retrieve the statistics for the created traffic manager/ethernet counters in synchronous mode and print the returned values.
reset_stats	reset the statistics for all created DPA Stats counters.

In order to have a successful operation, the user must verify that the returned values of the counters statistics matches the number of sent frames.

Classifier demo is a USDPAA application that requires the following resources:

- `classif_demo_config.xml` - config file
- `classif_demo_policy.xml` - policy file

This user space application initializes the classification schemes using the FMC library, and afterwards it manages the DPA Classifier table.

The application receives traffic on the configured Ethernet port. The traffic is split in IPv4 and IPv6 flows, and then lookups are performed in the DPA Classifier table. If the result is HIT, the frame is sent to the next configured classifier table, the `dscp_classif`.

On B4 platforms, at this point, based on DSCP, the frame is enqueued to the appropriate logical frame queue. The traffic manager based on its internal algorithms decides to send or not the packet back to be transmitted on the interface from which it was received.

On the other hand, in case of P4 platforms, the traffic is classified based on the DSCP table entries, shown in the tables below, and then enqueued on the fqid configured for the TX port used by the application.

If the result is MISS, the frame is silently dropped in both cases.

The application creates a two Exact Match tables managed by key, none of them contains prefilled entries, but the user can dynamically insert or remove keys with `add4/6` or `rem4/6` PPAM CLI commands. Each table supports a maximum of 64 keys. Each key is composed from the following parts: IPSA, IPDA and protocol. The add commands have an additional field MACDA that is used for the forwarding header manipulation.

Another Exact Match table is populated by the application with 12 entries that represent the main QOS traffic classes AFxy. Those entries are static, defined in the application and cannot be modified. For each traffic class a CEETM channel is defined, each channel has three CEETM class queues (corresponding to the level of drop precedence) and a class congestion group (that aggregates all dropped traffic) defined. Due to lack of CEETM support on P4 platforms all traffic is sent from the DSCP classification table to the first fqid configured on the TX port and the table structure is the same as represented below.

Table 104. Application QOS Table (Classifier Exact Match Table)

Channel	Class Queue	QOS	DSCP	TOS	Entry
CH0	CQ0	AF11	0x0A	0x28	0
CH0	CQ1	AF12	0x0C	0x30	1
CH0	CQ2	AF13	0x0E	0x38	2
CH1	CQ0	AF21	0x12	0x48	3
CH1	CQ1	AF22	0x14	0x50	4
CH1	CQ2	AF23	0x16	0x58	5
CH2	CQ0	AF31	0x1A	0x68	6
CH2	CQ1	AF32	0x1C	0x70	7
CH2	CQ2	AF33	0x1E	0x78	8
CH3	CQ0	AF41	0x22	0x88	9
CH3	CQ1	AF42	0x24	0x90	10
CH3	CQ2	AF43	0x26	0x98	11

NOTE

Classifier demo application does not use the cores or the IP stack for performing classification (look-up actions) or header manipulation operation (protocol operations). It uses the frame manager features for traffic classification and header manipulation.

7.4.2.2.2 Running classifier_demo

Classifier_demo application supports the following platforms:

- P2041
- P4080
- B4860
- B4420
- T4240
- T2080
- LS1043A

To run the application, you may need to use a specific device tree file that comes with the Linux DPA offloading driver. The following shows you how to produce this device tree file for your platform.

7.4.2.2.2.1 Application environment specifications

This application uses the setup pictured below with the following connections:

Table 105. Port Connections

SoC	Traffic Port
P4080	fm1-mac0
P2041	fm0-mac1
B4860	fm0-mac5
B4420	fm0-mac3
T4240	fm0-mac4
T2080	fm0-mac3
LS1043A	fm0-mac0

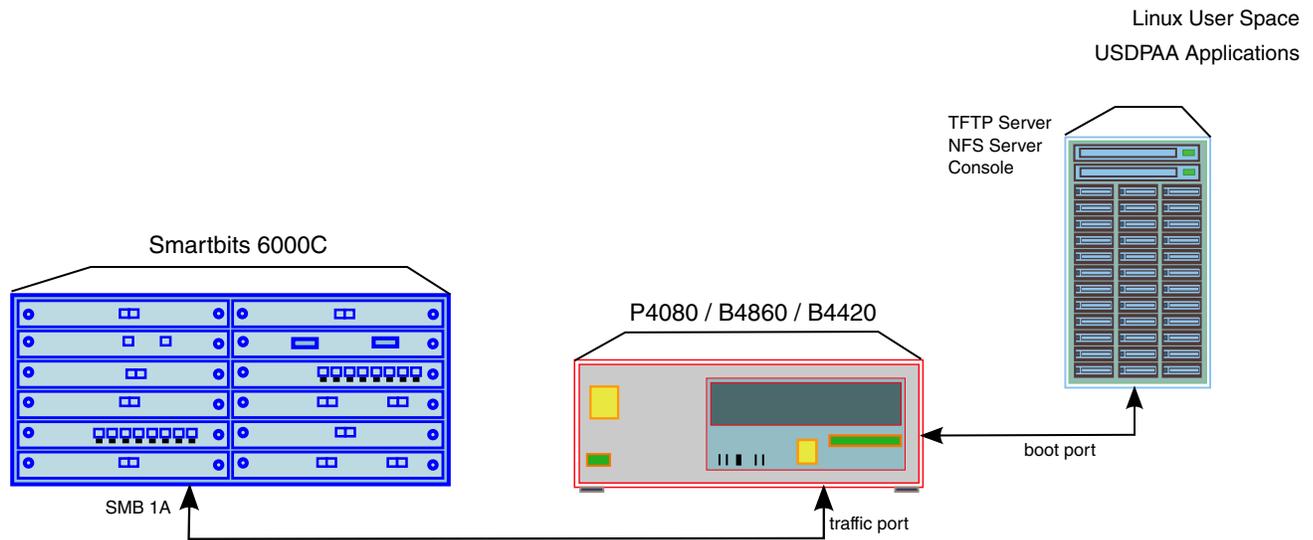


Figure 121. Use Case Set-up

If you want *classifier_demo* application to use a different traffic port than the ones listed above, you need to update the XML port configuration file for your platform and to use the proper command line arguments for your new traffic port. Please refer to the paragraph **Application start-up and configuration** for further details.

74.2.2.2.2 Application start-up and configuration

Use the steps described later in **Compiling the Device Tree for USDPAAs Applications** to generate a device tree binary file. Boot the board with the compiled kernel, `arch/powerpc/boot/uImage`, and the DTB file.

To enable the scatter-gather support in the DPAA Ethernet driver add the following to the boot arguments:

```
setenv othbootargs "fsl_fm_max_frm=9600"
```

The following commands assume that the application runs on a B4860QDS board. When running on a different supported platform, please use the proper configuration file and the parameters listed in the section above. Run these commands to set up the USDPAAs network configuration and PCD resources used by the application:

```
export DEF_CFG_PATH=/usr/etc/classifier_demo_config-b4860.xml
export DEF_PCD_PATH=/usr/etc/classifier_demo_policy.xml
export DEF_PDL_PATH=/etc/fmc/config/hxs_pdl_v3.xml
```

First ensure that the following device files have been created:

- `/dev/dpa_classifier`
- `/dev/dpa_stats`

The `classifier_demo` can be run with the following command:

```
/usr/bin/classifier_demo -f 0 -t 6
```

The following arguments specify the frame manager ports used by the application:

- `-f [FMan index]`
- `-t [Traffic Port index]`

The output displayed by the application would be something similar with the following:

```
Found /fsl,dpaa/dpa-fman0-oh@2, Tx Channel = 80a, FMAN = 0, Port ID = 1
Found /fsl,dpaa/dpa-fman0-oh@3, Tx Channel = 80b, FMAN = 0, Port ID = 2
Found /fsl,dpaa/dpa-fman0-oh@4, Tx Channel = 80c, FMAN = 0, Port ID = 3
Found /fsl,dpaa/ethernet@4, Tx Channel = 806, FMAN = 0, Port ID = 4
```

```
Found /fsl,dpaa/ethernet@5, Tx Channel = 807, FMAN = 0, Port ID = 5
Found /fsl,dpaa/ethernet@16, MAC-LESS node
Found /fsl,dpaa/ethernet@17, MAC-LESS node
Configuring for 1 network interface
Allocated DMA region size 0x1000000
dpa_classifier_demo: using the following config file: /usr/etc/classifier_demo_config-b4860.xml
dpa_classifier_demo: using the following PCD file: /usr/etc/classifier_demo_policy.xml
dpa_classifier_demo: using the following PDL file: /etc/fmc/config/hxs_pdl_v3.xml
cpu hotplug daemon not running: No such file or directory
dpa_classifier_demo is assuming FMan:0 and port:6
Successfully initialized CEETM resources for FMan 0 Port 6

Successfully CREATED DPA Classifier DSCP table (td=0)
Successfully CREATED DPA Classifier Exact Match table (td=1).
Successfully Modified Miss Action for DPA Classifier Exact Match table (td=1).
Successfully CREATED DPA Classifier Exact Match table (td=2).
Successfully Modified Miss Action for DPA Classifier Exact Match table (td=2).
DPA Stats library successfully initialized

Successfully Initialized DPA Stats instance: 0
Successfully created DPA Stats counter: 0
Successfully created DPA Stats counter: 1
Successfully created DPA Stats counter: 2
Successfully created DPA Stats counter: 3
Successfully created DPA Stats counter: 4
Successfully created DPA Stats counter: 5
Released 0 bufs to BPID 7
Released 0 bufs to BPID 8
Released 8192 bufs to BPID 9
Thread uid:0 alive (on cpu 1)
```

7.4.2.2.3 Running the application

At this point the user can start sending traffic with the traffic generator.

By running a set of add4 and add6 commands after the application is started, the user can populate the IPv4 and IPv6. The commands for populating the IPv4 table are:

```
> add4 192.168.1.1 128.224.10.10 tcp 44:00:00:00:00:01 881 921
> add4 192.168.25.1 128.224.20.10 udp 44:00:00:00:00:02 882 922
> add4 192.168.50.1 128.224.30.10 TCP 44:00:00:00:00:03 883 923
> add4 192.168.75.1 128.224.40.10 UDP 44:00:00:00:00:04 884 924
> add4 192.168.100.1 128.224.50.10 6 44:00:00:00:00:05 885 925
> add4 192.168.125.1 128.224.60.10 17 44:00:00:00:00:06 886 926
```

The commands for populating the IPv6 table are:

```
> add6 3ffe:1944:0100:000a:0000:00bc:2500:0d0b 1ffe:2044:ba00:320a:1100:00bc:25a1:010b TCP
66:00:00:00:00:01 6011 501
> add6 3ffe:1944:0200:000a:0000:00bc:2500:0d0b 1ffe:2044:ba00:320a:1100:00bc:25a1:020b UDP
66:00:00:00:00:02 6012 502
> add6 3ffe:1944:0300:000a:0000:00bc:2500:0d0b 1ffe:2044:ba00:320a:1100:00bc:25a1:030b 6
66:00:00:00:00:03 6013 503
> add6 3ffe:1944:0400:000a:0000:00bc:2500:0d0b 1ffe:2044:ba00:320a:1100:00bc:25a1:040b 17
66:00:00:00:00:04 6014 504
```

```
> add6 3ffe:1944:0500:000a:0000:00bc:2500:0d0b 1ffe:2044:ba00:320a:1100:00bc:25a1:050b tcp
66:00:00:00:00:05 6015 505
```

```
> add6 3ffe:1944:0600:000a:0000:00bc:2500:0d0b 1ffe:2044:ba00:320a:1100:00bc:25a1:060b udp
66:00:00:00:00:06 6016 506
```

After entering those commands the content of the DPA Classifier Exact Match tables will be:

Table 106. IPv4 Classification Table (Classifier Exact Match Table)

Flow	IP SRC	IP DST	Protocol
1	192.168.1.1	128.224.10.10	TCP
2	192.168.25.1	128.224.20.10	UDP
3	192.168.50.1	128.224.30.10	TCP
4	192.168.75.1	128.224.40.10	UDP
5	192.168.100.1	128.224.50.10	TCP
6	192.168.125.1	128.224.60.10	UDP

Table 107. IPv6 Classification Table (Classifier Exact Match Table)

Flow	IP SRC	IP DST	Protocol
1	3ffe:1944:0100:000a: 0000:00bc:2500:0d0b	1ffe:2044:ba00:320a: 1100:00bc:25a1:010b	TCP
2	3ffe:1944:0200:000a: 0000:00bc:2500:0d0b	1ffe:2044:ba00:320a: 1100:00bc:25a1:020b	UDP
3	3ffe:1944:0300:000a: 0000:00bc:2500:0d0b	1ffe:2044:ba00:320a: 1100:00bc:25a1:030b	TCP
4	3ffe:1944:0400:000a: 0000:00bc:2500:0d0b	1ffe:2044:ba00:320a: 1100:00bc:25a1:040b	UDP
5	3ffe:1944:0500:000a: 0000:00bc:2500:0d0b	1ffe:2044:ba00:320a: 1100:00bc:25a1:050b	TCP
6	3ffe:1944:0600:000a: 0000:00bc:2500:0d0b	1ffe:2044:ba00:320a: 1100:00bc:25a1:060b	UDP

By sending with a traffic generator flows identically to the ones described in the tables above, the user will notice that the traffic is received back with the TTL decremented and the MACDA, L3 source port and destination ports updated as specified in the *add4 / add6* command that inserted the entry. On P4 platforms, all traffic is received back. If additional flows are defined in the traffic generator configuration, the miss statistics will also be increased. Based on the traffic sent the user can read the statistics by running the *get_classif_stats/get_classif_stats_sync* or *get_traffic_stats/get_traffic_stats_sync* commands. In order to have a successful operation, the user must verify that the returned values of the counter statistics matches the number of sent frames.

In order to shut down the application you can type *quit* in the application console.

7.4.2.3 Adapting the fragmentation_demo application

The application demonstrates how IP fragmentation can be offloaded to the FMan by using the direct XML configuration and the DPA offloading driver.

7.4.2.3.1 Overview

The IP fragmentation function works strictly on offline ports and performs fragmentation of IPv4 and IPv6 packets into IP fragments according to RFC 791 and RFC 2460. The DPA Stats component usage is highlighted in the same application by creating single and class counters for the IP Fragmentation packet header manipulation.

The application initializes the IP fragmentation and the Parse-Classify-Police-Distribute (PCD) description through the use of the FMC Library.

Traffic is received on a backhaul Rx port (BP) by USDPAAs fragmentation_demo application. IP packets that pass the Parse-Classify and Distribute descriptor configured on the Rx port are received in the application. The application swaps the Ethernet MAC addresses and enqueue the traffic in the offline port. This port performs a classification based on VLAN and, depending on the value, it performs IP fragmentation if the frame size is larger than a pre-configured value. Afterwards it forwards the obtained fragments back to the Tx queue of the BP port.

The user has the possibility of performing a number of actions on the DPA Stats counters by typing the following commands in the USDPAAs PPAM command line:

Table 108. USDPAAs PPAM commands

Command Line	Description
get_stats	retrieve the statistics for the created counters in an asynchronous mode and print the returned values.
get_stats_sync	retrieve the statistics for the created counters in a synchronous mode and print the returned values.
reset_stats	reset the statistics for the created counters.

To have a successful operation, you should verify that the returned values of the counters statistics matches the number of sent frames.

NOTE

The fragmentation demo application is different from the fragmentation performed in the Linux IP stack because the demo does not use the cores or the IP stack for performing fragmentation operation. It uses the frame manager for fragmentation processing.

7.4.2.3.2 Running fragmentation_demo

fragmentation_demo application supports the following platforms:

- P2041
- P4080
- B4860
- B4420
- T4240
- T2080
- LS1043A

In order to run it you may need to use a specific device tree file that comes with the DPA offloading driver. Please read below about how to produce this device tree file for your platform.

7.4.2.3.2.1 Application environment specifications

The environment is the same as described for the classifier_demo application. Please see **Classifier_demo: Application environment specifications**.

If you need the *fragmentation_demo* application to run on a different traffic port than the defaults, you will need to adapt the XML port configuration file for your platform and use the proper command line arguments for your new traffic port. Please refer to the paragraph **Application start-up and configuration** for further details.

7.4.2.3.2.2 Application start-up and configuration

Use the steps described later in **Appendix A: Compiling the Device Tree for IP Offloading** to generate a device tree binary file. Boot the board with the compiled kernel, arch/powerpc/boot/uImage, and the DTB file.

To enable the scatter-gather support in the DPAA Ethernet driver add the following to the boot arguments:

```
setenv othbootargs "fsl_fm_max_frm=9600"
```

The following commands assume that the application runs on a B4860QDS board. When running on a different supported platform, please use the proper parameters listed in **Running Classifier_demo: Application environment specifications**. For setting up the USDPAA network configuration and PCD resources used by the application run the following commands:

```
export DEF_CFG_PATH=/usr/etc/fragmentation_demo_config-b4860.xml
export DEF_PCD_PATH=/usr/etc/fragmentation_demo_policy.xml
export DEF_PDL_PATH=/etc/fmc/config/hxs_pdl_v3.xml
```

First ensure that the following device files have been created:

- /dev/dpa_stats

The *fragmentation_demo* can be run with the following command:

```
/usr/bin/fragmentation_demo -f 0 -t 6 -o 2
```

The following arguments specify the FMan ports used by the application:

- -f [FMan index]
- -o [Offline Port index]
- -t [BP Rx port index]

The output displayed by the application would be something similar with the following:

```
Found /fsl,dpaa/dpa-fman0-oh@2, Tx Channel = 80a, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@0, Tx Channel = 802, FMAN = 0, Port ID = 0
Found /fsl,dpaa/ethernet@1, Tx Channel = 803, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@2, Tx Channel = 804, FMAN = 0, Port ID = 2
Found /fsl,dpaa/ethernet@5, Tx Channel = 807, FMAN = 0, Port ID = 5
Configuring for 2 network interfaces
Allocated DMA region size 0x1000000
Released 0 bufs to BPID 4
fragmentation_demo is assuming FMan:0 and eth:6 and offline port:2
Released 0 bufs to BPID 7
Released 0 bufs to BPID 8
Warn: drained 8192 bufs from BPID 9
Released 8192 bufs to BPID 9
```

```
Thread uid:0 alive (on cpu 1)
fragmentation_demo >
```

74.2.3.2.3 Running the application

The user should send IPv4 or IPv6 frames generated by an external traffic generator. The following example of IPv4 frames was used to validate the IPv4 Fragmentation:

Table 109. fragmentation_demo IPv4 Input Traffic

MAC SA	VLAN TCI	IP Source	Protocol	Size	IP Fragment Offset
00:00:00:00:00:29	0x2614	192.168.1.1	UDP	1280	0
00:00:00:00:00:2a	0x2615	192.168.1.2	UDP	1280	0
00:00:00:00:00:2b	0x2616	192.168.1.3	UDP	1280	0
00:00:00:00:00:2c	0x2617	192.168.1.4	UDP	1280	0

IPv4 frames that have the VLAN TCI value equal to 0x2614 or 0x2616 are fragmented into fragments of maximum 256 bytes in size, while frames that have the VLAN TCI value equal to 0x2615 or 0x2617 are fragmented into fragments of maximum 512 bytes in size.

IPv6 frames that have the VLAN TCI value equal to 0x6800 or 0x6802 are fragmented into fragments of maximum 256 bytes in size, while frames that have the VLAN TCI value equal to 0x6801 or 0x6803 are fragmented into fragments of maximum 512 bytes in size.

The following example of IPv6 frames was used to validate the IPv6 fragmentation:

Table 110. fragmentation_demo IPv6 input traffic

MAC SA	VLAN TCI	IP Source	Protocol	Size
00:00:00:00:00:2d	0x6800	3FFE:1944:0400:000A:0000:00BC:2500:0D01	UDP	1024
00:00:00:00:00:2e	0x6801	3FFE:1944:0400:000A:0000:00BC:2500:0D02	UDP	1024
00:00:00:00:00:2f	0x6802	3FFE:1944:0400:000A:0000:00BC:2500:0D03	UDP	1024
00:00:00:00:00:30	0x6803	3FFE:1944:0400:000A:0000:00BC:2500:0D04	UDP	1024

After sending the traffic, the user can see the statistics values for the IP fragmentation process. The counter “OH_FRAG1” stands for a single counter of type Fragmentation with MTU of 256 bytes, the counter “OH_FRAG2” stands for single counter of type Fragmentation with MTU of 512 bytes and both objects are also grouped in a class counter of type Fragmentation.

```
STATISTICS:          FRAG_TOTAL_FRAMES FRAG_FRAMES FRAG_GEN_FRAGS
OH_FRAG1           :                0          0          0
OH_FRAG2           :                0          0          0
CLS_MBR_OH_FRAG1  :                0          0          0
CLS_MBR_OH_FRAG1  :                0          0          0
```

In order to shut down the application you can type *quit* in the application console.

7.4.2.4 Manipulating the reassembly_demo application

This application demonstrates how IP reassembly can be offloaded to the FMan by using the direct XML configuration and the DPA offloading driver.

7.4.2.4.1 Overview

This application demonstrates how IP reassembly can be offloaded to the FMan by using the direct XML configuration and the DPA offloading driver. The IP Reassembly feature can be configured on an RX port or an Offline port and performs reassembly on detected IPv4 and IPv6 protocol fragments. For B4 platforms, the application also provides Virtual Storage Profile support. The DPA Stats component usage is highlighted in the same application by creating single counters for Ethernet and Reassembly, and both single and class counters for Classification Nodes.

The application initializes the Reassembly functionality and the Parse-Classify-Police-Distribute (PCD) description through the use of the FMC Library.

Traffic is received on a backhaul Rx port (BP) by USDPAA reassembly_demo application. IP fragments are detected and reassembled before arriving into the application. The application swaps Ethernet MAC addresses and enqueues the traffic back to the Tx queue of the same port.

The user has the possibility of performing a number of actions on the DPA Stats counters by typing the following commands in the USDPAA PPAM command line:

Table 111. DPA Stats class counter actions

Command Line	Description
get_stats	retrieve the statistics for the created counters in an asynchronous mode and print the returned values.
get_stats_sync	retrieve the statistics for the created counters in a synchronous mode and print the returned values.
reset_stats	reset the statistics for the created counters.

To have a successful operation, the user must verify that the returned values of the counters statistics matches the number of sent frames for each stage of the test.

7.4.2.4.1.1 Virtual storage profiles

The virtual storage profile is only available on frame manager v3. The configuration is transparent to the user. Three storage profiles are defined in the policy file `reassembly_demo_policy-v3.xml`:

```
<vsp name="Default_VSP" base="0"/>
<vsp name="IPv4_Reass_VSP" base="1"/>
<vsp name="IPv6_Reass_VSP" base="2"/>
```

The first storage profile, Default_VSP is also the default storage profile of the port. The second and the third storage profiles are used for IPv4 and IPv6 flows. Each of the storage profiles described above has a set buffer pool associated with it (up to four buffer pools can be defined per vsp)

The buffer pools are dynamically allocated and initialized by `reassembly_demo`.

When an IPv4 flow is reassembled / classified on inbound port, the IPv4_Reass_VSP is used and the buffers are selected from one of the buffer pools available for this VSP. When an IPv6 flow is reassembled/classified on inbound port, the

IPv6_Reass_VSP with buffers from one of the buffer pools available for this VSP is used. For any other flows the Default_VSP is selected.

The VSP selection mechanism described in this paragraph is totally transparent to the user and doesn't affect the use case behavior in any way.

In order to test the non-consistent storage profile functionality the user needs to take into consideration that if a fragment that enters the reassembly distribution is not the first fragment, the default VSP of the port is selected. Otherwise according to the classification, if the first fragment has the L4 destination port equal to a specific value, the IPv4 VSP is selected. If the reassembled frame has fragments in buffers from different buffer pools, then there is a NCSP event triggered and the frame is enqueued to the NCSP queue. For testing this particular case the user has to take care sending the first fragment of the IPv4 frame (the one with the Fragment Offset flag with value 0) after any other fragment (Ex: fragment3, fragment2, fragment1, fragment4). For the moment the NCSP could be tested only with IPv4 frames.

7.4.2.4.1.2 Differences from Linux IP stack

Reassembly_demo application does not use the cores or the IP stack for performing reassembly operation. It uses the frame manager for processing the reassembly.

7.4.2.4.2 Running reassembly_demo

reassembly_demo application supports the following platforms:

- P2041
- P4080
- B4860
- B4420
- T4240
- T2080
- LS1043A

In order to run it you may need to use a specific device tree file that comes with the DPA offloading driver. Please read below about how to produce this device tree file for your platform.

7.4.2.4.2.1 Application environment specifications

The environment is the same as described for the classifier_demo application. Please see **Classifier_demo: Application environment specifications**.

If you need the *reassembly_demo* application to run on a different traffic port than the defaults, you will need to

1. adapt the XML port configuration file for your platform
2. adapt the DTS file that Linux kernel boots up with and
3. use the proper command line arguments for your new traffic port.

Please refer to the paragraph **Application start-up and configuration** for further details.

7.4.2.4.2.2 Application start-up and configuration

Use the steps described later in **Appendix A: Compiling the Device Tree for IP Reassembly** to generate a device tree binary file. Boot the board with the compiled kernel, `arch/powerpc/boot/uImage`, and the DTB file.

To enable the scatter-gather support in the DPAA Ethernet driver add the following to the boot arguments:

```
setenv othbootargs "fsl_fm_max_frm=9600"
```

The following commands assume that the application runs on a B4860QDS board. When running on a different supported platform, please use the proper parameters listed in **Running Classifier_demo: Application environment**

specifications. For setting up the USDPAAs network configuration and PCD resources used by the application run the following commands:

```
export DEF_CFG_PATH=/usr/etc/reassembly_demo_config-b4860.xml
export DEF_PCD_PATH=/usr/etc/reassembly_demo_policy.xml
export DEF_PDL_PATH=/etc/fmc/config/hxs_pdl_v3.xml
```

First ensure that the following device files have been created:

- /dev/dpa_stats

The `reassembly_demo` can be run with the following command:

```
/usr/bin/reassembly_demo -f 0 -t 6
```

The following arguments specify the FMan ports used by the application:

- -f [FMan index]
- -t [Traffic Port index]

The output displayed by the application would be something similar with the following:

```
Loading configuration
Found /fsl,dpaa/dpa-fman0-oh@2, Tx Channel = 80a, FMAN = 0, Port ID = 1
Found /fsl,dpaa/dpa-fman0-oh@3, Tx Channel = 80b, FMAN = 0, Port ID = 2
Found /fsl,dpaa/dpa-fman0-oh@4, Tx Channel = 80c, FMAN = 0, Port ID = 3
Found /fsl,dpaa/ethernet@0, Tx Channel = 802, FMAN = 0, Port ID = 0
Found /fsl,dpaa/ethernet@1, Tx Channel = 803, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@2, Tx Channel = 804, FMAN = 0, Port ID = 2
Found /fsl,dpaa/ethernet@5, Tx Channel = 807, FMAN = 0, Port ID = 5
Configuring for 1 network interface
Allocated DMA region size 0x1000000
Released 4096 bufs to BPID 1
Released 4096 bufs to BPID 2
Released 8192 bufs to BPID 3
reassembly_demo is assuming FMan:0 and MAC:6
Released 0 bufs to BPID 7
Released 0 bufs to BPID 8
Released 8192 bufs to BPID 9
Thread uid:0 alive (on cpu 1)
reassembly_demo >
```

7.4.2.4.2.3 Running the application

The user should send IPv4 or IPv6 frames generated by an external traffic generator. The following example of IPv4 and IPv6 frames can be used to validate the IP reassembly:

Table 112. reassembly_demo IPv4 & IPv6 Input Traffic

VLAN TCI	IP Source	Protocol	Size (w/ CRC)	ID field	Flags	Fragment Offset
0x0001	192.168.0.1	UDP	298	5	MF	0
0x0001	192.168.0.1	IPv4	298	5	MF	256 (32 x 8)

Table continues on the next page...

Table 112. reassembly_demo IPv4 & IPv6 Input Traffic (continued)

VLAN TCI	IP Source	Protocol	Size (w/ CRC)	ID field	Flags	Fragment Offset
0x0001	192.168.0.1	IPv4	298	5	MF	512 (64 x 8)
0x0001	192.168.0.1	IPv4	298	5	0	768 (96 x 8)
0x0002	192.168.0.2	UDP	298	5	MF	0
0x0002	192.168.0.2	IPv4	298	5	MF	256 (32 x 8)
0x0002	192.168.0.2	IPv4	298	5	MF	512 (64 x 8)
0x0002	192.168.0.2	IPv4	298	5	0	768 (96 x 8)
0x0003	192.168.0.3	UDP	298	5	MF	0
0x0003	192.168.0.3	IPv4	298	5	MF	256 (32 x 8)
0x0003	192.168.0.3	IPv4	298	5	MF	512 (64 x 8)
0x0003	192.168.0.3	IPv4	298	5	0	768 (96 x 8)
0x0004	192.168.0.4	UDP	298	5	MF	0
0x0004	192.168.0.4	IPv4	298	5	MF	256 (32 x 8)
0x0004	192.168.0.4	IPv4	298	5	MF	512 (64 x 8)
0x0004	192.168.0.4	IPv4	298	5	0	768 (96 x 8)
0x0001	3FFE:1944:0100:000A: 0000:00BC:2500:0D0B	UDP	414	5	MF	0
0x0001	3FFE:1944:0100:000A: 0000:00BC:2500:0D0B	IPv6	414	5	MF	344
0x0001	3FFE:1944:0100:000A: 0000:00BC:2500:0D0B	IPv6	414	5	MF	688
0x0001	3FFE:1944:0100:000A: 0000:00BC:2500:0D0B	IPv6	414	5	0	1032
0x0002	3FFE:1944:0200:000A: 0000:00BC:2500:0D0B	UDP	414	5	MF	0
0x0002	3FFE:1944:0200:000A: 0000:00BC:2500:0D0B	IPv6	414	5	MF	344
0x0002	3FFE:1944:0200:000A: 0000:00BC:2500:0D0B	IPv6	414	5	MF	688

Table continues on the next page...

Table 112. reassembly_demo IPv4 & IPv6 Input Traffic (continued)

VLAN TCI	IP Source	Protocol	Size (w/ CRC)	ID field	Flags	Fragment Offset
0x0002	3FFE:1944:0200:000A: 0000:00BC:2500:0D0B	IPv6	414	5	0	1032
0x0003	3FFE:1944:0300:000A: 0000:00BC:2500:0D0B	UDP	414	5	MF	0
0x0003	3FFE:1944:0300:000A: 0000:00BC:2500:0D0B	IPv6	414	5	MF	344
0x0003	3FFE:1944:0300:000A: 0000:00BC:2500:0D0B	IPv6	414	5	MF	688
0x0003	3FFE:1944:0300:000A: 0000:00BC:2500:0D0B	IPv6	414	5	0	1032
0x0004	3FFE:1944:0400:000A: 0000:00BC:2500:0D0B	UDP	414	5	MF	0
0x0004	3FFE:1944:0400:000A: 0000:00BC:2500:0D0B	IPv6	414	5	MF	344
0x0004	3FFE:1944:0400:000A: 0000:00BC:2500:0D0B	IPv6	414	5	MF	688
0x0004	3FFE:1944:0400:000A: 0000:00BC:2500:0D0B	IPv6	414	5	0	1032

The IP Reassembly feature is considered to be working properly if for the provided example of IP fragments are reassembled in the following frames:

Table 113. Expected Reassembled Frames

IP Source	Protocol	Size	Flags	Fragment Offset
192.168.0.1	IPv4/UDP	1066	0	0
192.168.0.2	IPv4/UDP	1066	0	0
192.168.0.3	IPv4/UDP	1066	0	0
192.168.0.4	IPv4/UDP	1066	0	0
3FFE: 1944:0100:0 00A: 0000:00BC: 2500:0D0B	IPv6/UDP	1438	0	0

Table continues on the next page...

Table 113. Expected Reassembled Frames (continued)

IP Source	Protocol	Size	Flags	Fragment Offset
3FFE: 1944:0100:0 00A: 0000:00BC: 2500:0D0B	IPv6/UDP	1438	0	0
3FFE: 1944:0100:0 00A: 0000:00BC: 2500:0D0Bf	IPv6/UDP	1438	0	0
3FFE: 1944:0100:0 00A: 0000:00BC: 2500:0D0B	IPv6/UDP	1438	0	0

After sending the traffic, the user can validate the statistics values for the IP reassembly process on the Ethernet interface used to send and receive the traffic and the Classification Node:

```
reassembly_demo > get_stats_sync

REASS      : TIMEOUT RFD_POOL_BUSY INT_BUFF_BUSY EXT_BUFF_BUSY SG_FRAGS DMA_SEM NCSP
              0          0          0          0          0          0          0
REASS_IPV4 : FRAMES FRAGS_VALID FRAGS_TOTAL FRAGS_MALFORMED FRAGS_DISCARDED AUTOLEARN_BUSY
EXCEED_16FRAGS
              0          0          0          0          0          0          0
REASS_IPV6 : FRAMES FRAGS_VALID FRAGS_TOTAL FRAGS_MALFORMED FRAGS_DISCARDED AUTOLEARN_BUSY
EXCEED_16FRAGS
              0          0          0          0          0          0          0
ETH        : DROP_PKTS  BYTES  PKTS BC_PKTS MC_PKTS CRC_ALIGN_ERR UNDERSIZE_PKTS OVERSIZE_PKTS
              0          0      0      0      0          0          0          0
ETH        : FRAGMENTS JABBERS 64BYTE_PKTS 65_127BYTE_PKTS 128_255BYTE_PKTS 256_511BYTE_PKTS
512_1023BYTE_PKTS 1024_1518BYTE_PKTS
              0          0          0          0          0          0          0
0
ETH        : OUT_PKTS OUT_DROP_PKTS OUT_BYTES IN_ERRORS OUT_ERRORS IN_UNICAST_PKTS OUT_UNICAST_PKTS
              0          0          0          0          0          0          0
CNT_CLASSIF: IPv6_KEY0 IPv6_KEY1 IPv6_KEY2 IPv6_KEY3 MISS
              0          0          0          0          0
CLS_CLASSIF: IPv6_KEY0 IPv6_KEY1 IPv6_KEY2 IPv6_KEY3 MISS
              0          0          0          0          0
CNT_CLASSIF: IPv4_KEY0 IPv4_KEY1 IPv4_KEY2 IPv4_KEY3 MISS
              0          0          0          0          0
CLS_CLASSIF: IPv4_KEY0 IPv4_KEY1 IPv4_KEY2 IPv4_KEY3 MISS
              0          0          0          0          0
```

In order to shut down the application you can type *quit* in the application console.

7.4.2.5 Customizing the ipsec_offload application

USDPAA ipsec_offload is a multi-threaded application which applies a subset of IPsec transformations to the IPv4/IPv6 traffic.

7.4.2.5.1 IPsec_offload application overview

The IPv4/IPv6 traffic is processed between two Ethernet ports. The inbound port expects encrypted/tunneled traffic while the outbound port expects clear-text traffic. The classification, encryption/decryption, authentication, encapsulation/decapsulation is accomplished using the DPA offload driver which connects and configures DPAA traffic processing accelerators (Fman, SEC) to completely offload IPsec processing based on common Linux configuration tools (setkey, ip xfrm, arp, ip neigh). The following features are currently supported:

- ESP tunnel mode.
- TOS/ECN/DSCP propagation.

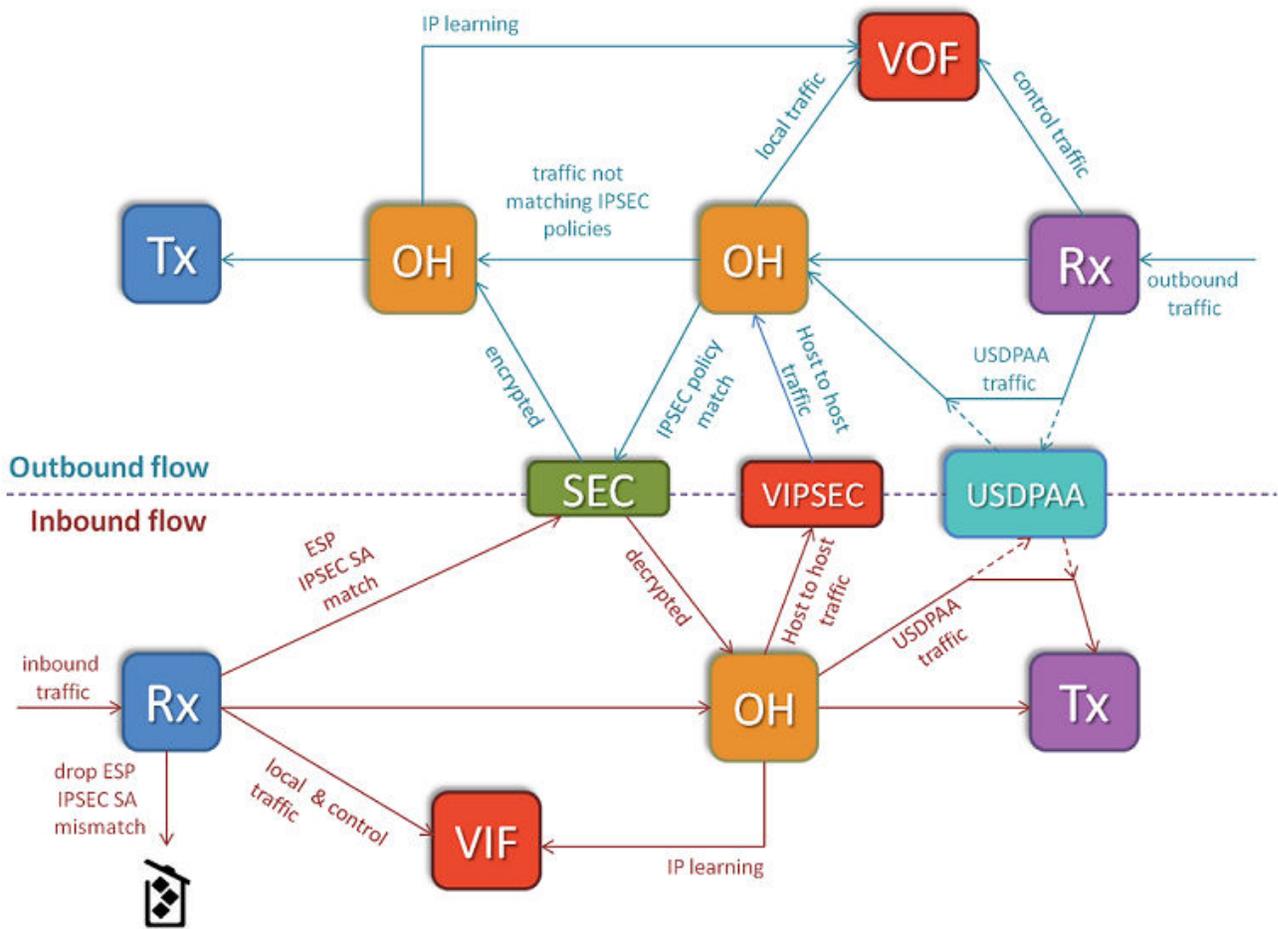


Figure 122. IPsec_offload flows

7.4.2.5.1.1 IPsec_offload outbound flows

The clear text traffic is received on the **outbound Rx** and classified based on UDP destination port. Frames matching UDP destination port 2152 are received by USDPA frame processing threads. The rest of the frames are sent directly to **pre-encryption OH** port. On this port the traffic is classified according to configured IPsec traffic selectors. Frames for which IPsec policy is applicable are sent to SEC for security processing. Optionally, this port performs IP fragmentation for frames matching IPsec policy if frame size is larger than a configured value (*mtu_pre_enc*). Traffic not matching IPsec policy bypass security processing and are enqueued directly to post-encryption OH port. The SEC output frames are enqueued to **post-**

encryption OH port. On this port the Ethernet source and destination addresses are updated with Tx port MAC address and next hop MAC address respectively.

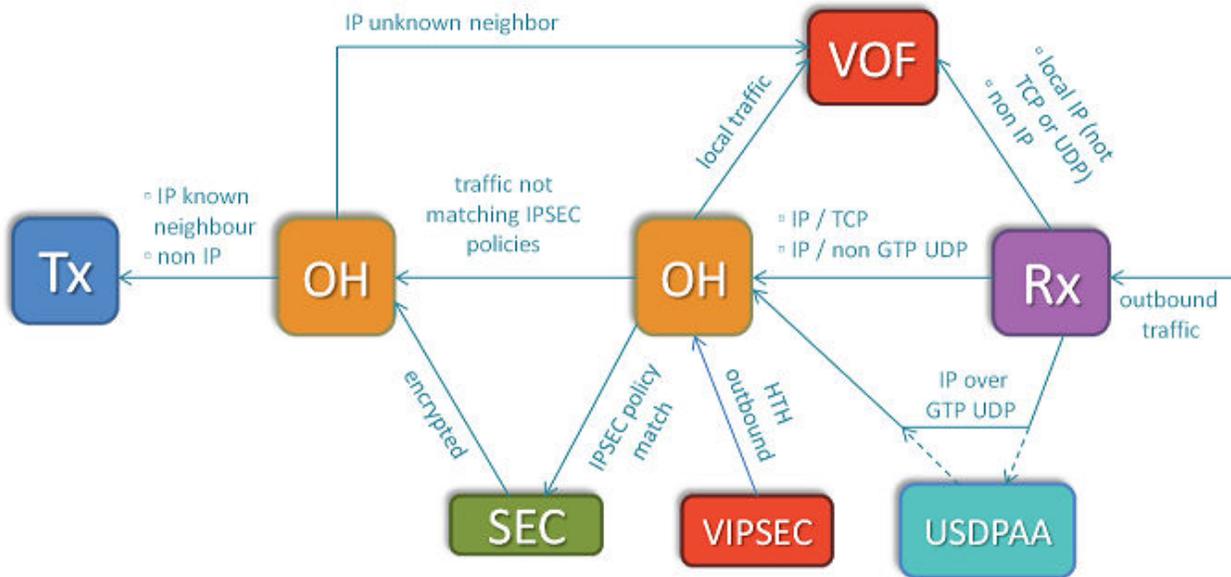


Figure 123. Outbound processing

Below are some flow diagrams that explain (as per the application's PCD) how frames are being processed at each node.

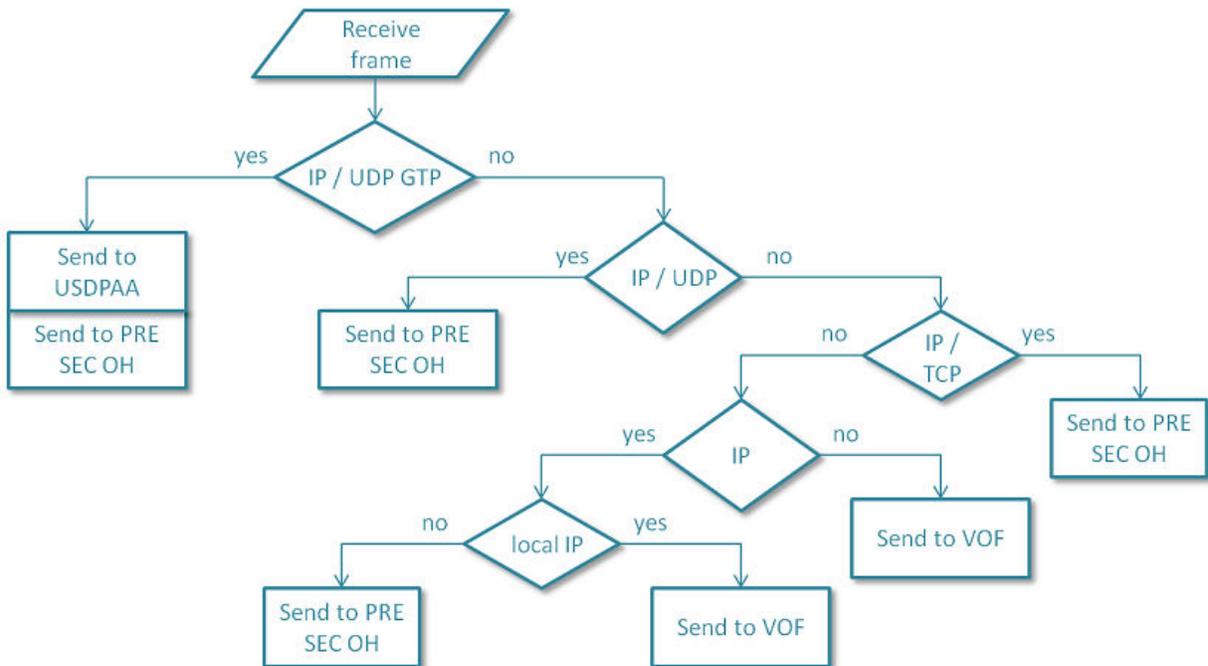


Figure 124. Outbound Rx

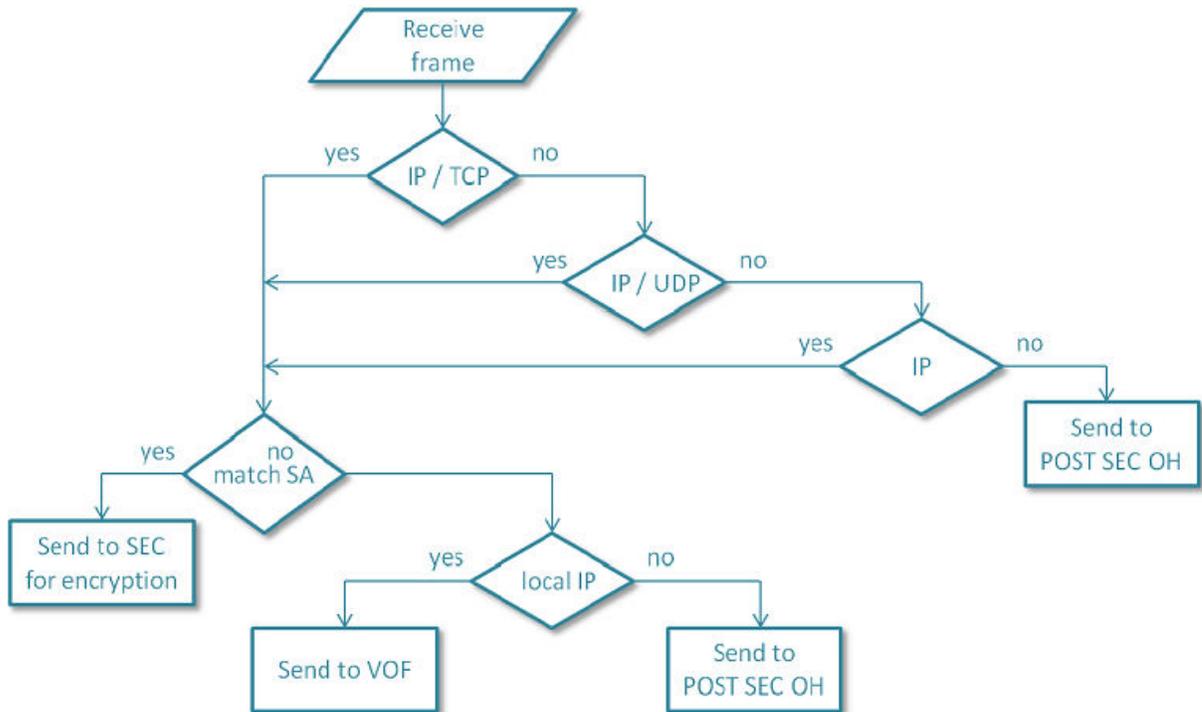


Figure 125. Outbound Pre SEC OH

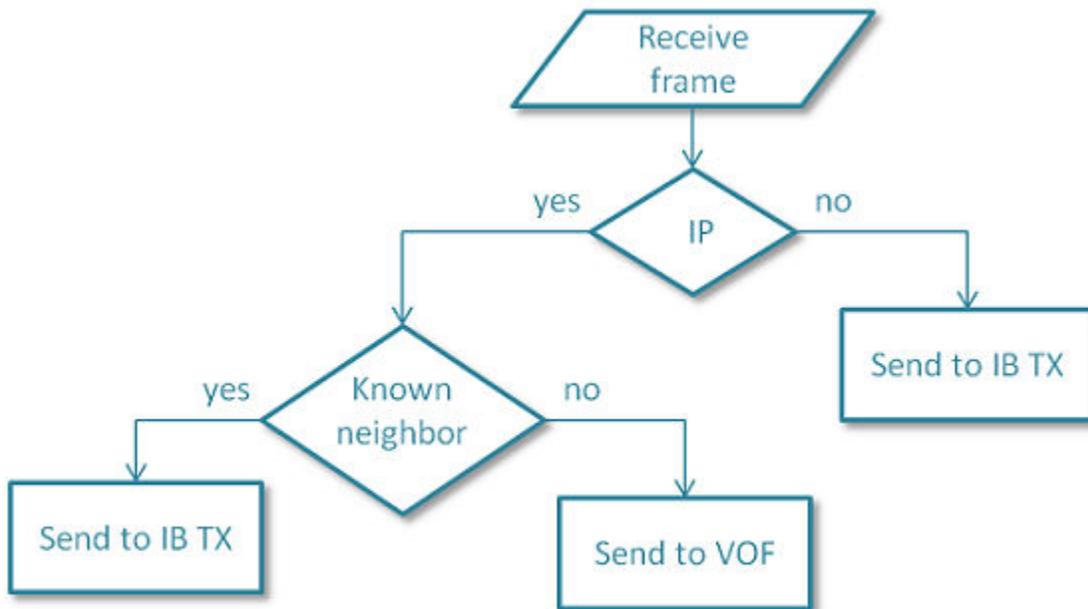


Figure 126. Outbound Post SEC OH

7.4.2.5.1.2 IPsec_offload inbound flows

IPsec traffic (IPv4/IPv6 ESP and UDP-encap ESP) received on the inbound port is classified according to offloaded security associations. Frames for which a security association is found are sent to the appropriate SEC engine input queues for

decryption/decapsulation; traffic for which a security association is not found is dropped. Non-IPsec traffic is directly sent to **inbound** OH port. Decrypted and non-IPsec traffic is reassembled if fragmented, optionally passes inbound policy verification and further a post IPsec classification which sends frames matching UDP destination port 2152 to USDPAA frame processing threads. Frames not matching UDP destination port 2152 have their Ethernet source and destination MAC addresses updated with Tx port MAC address and next hop MAC address respectively.

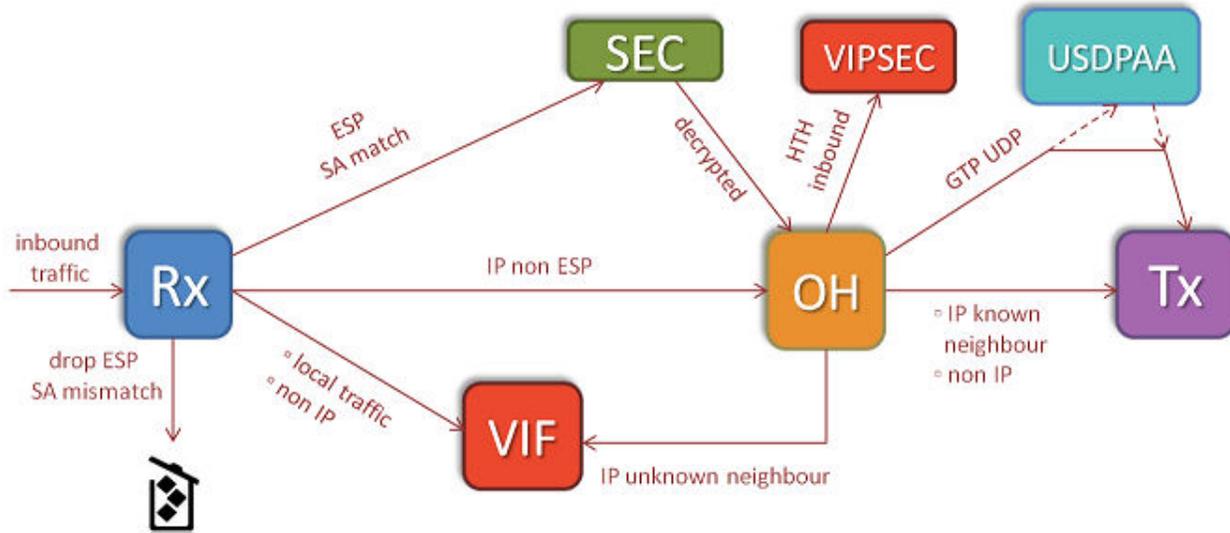


Figure 127. Inbound processing

Below are some flow diagrams that explain (as per the application's PCD) how frames are being processed at each node.

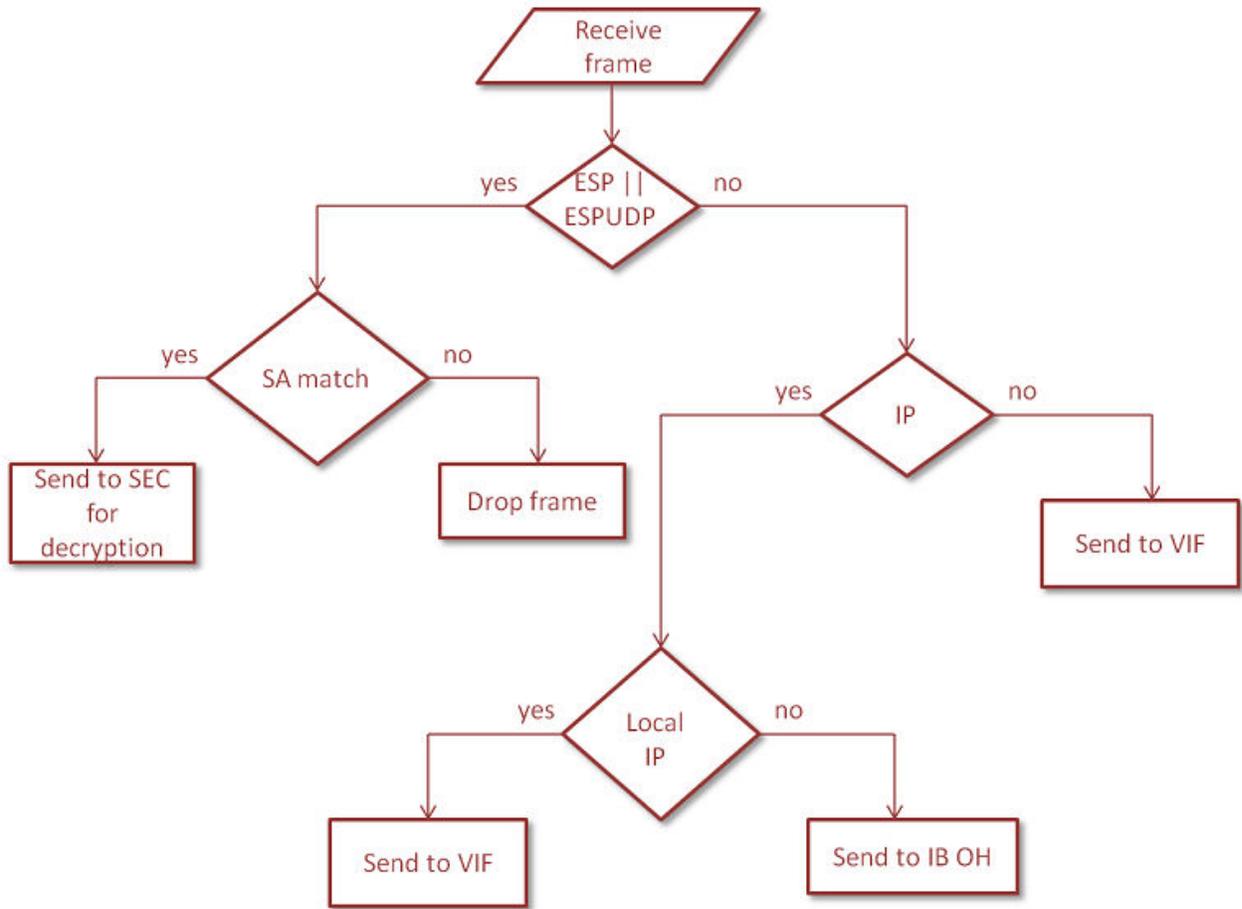


Figure 128. Inbound Rx

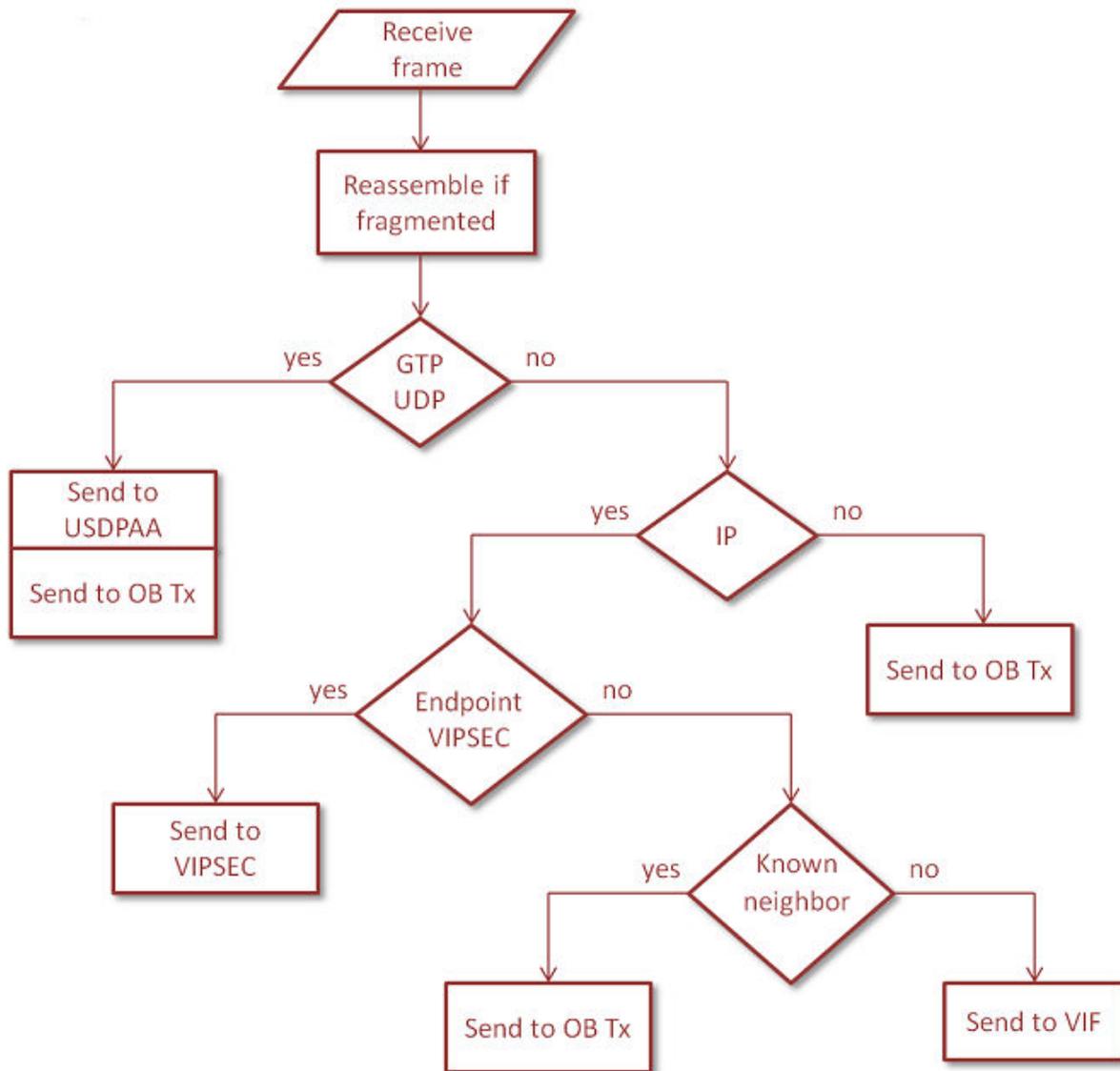


Figure 129. Inbound OH

7.4.2.5.1.3 Differences from existing ipsecfwd application

This USDPAA application allows complete offload of IPSec data paths, thus relieving the cores from the tasks of submitting/receiving frames to/from the SEC engine prior or after tunnel payload processing and from frame forwarding after IPSec processing. Core intervention is performed only for tasks related to application specific processing. Also, the application supports standard Linux tools for configuration.

7.4.2.5.2 Running the Application

The procedures for configuring the environment, starting up and configuring the Ipsec_offload application are discussed.

7.4.2.5.2.1 Application environment specifications

The application can run in a real network setup configuration and inter-operate with other IPsec gateways. There is also an option to put the inbound port in loopback mode and return IPsec traffic to the inbound port. The details on how to run the application refers to this mode.

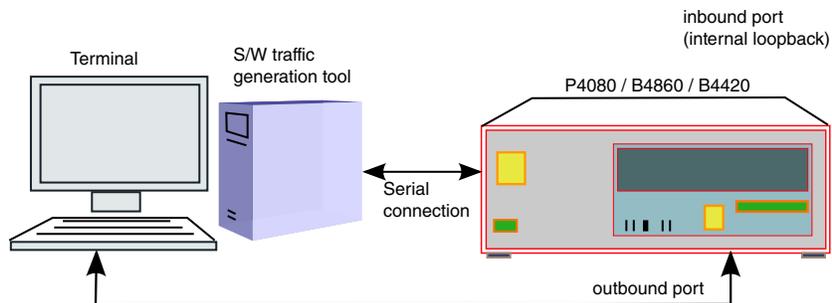


Figure 130. Use case set-up in loopback mode

Table 114. Port connections

SoC	Inbound Port	Outbound Port
P4080	fm1-mac1	fm1-mac0
P2041	fm0-mac2	fm0-mac1
B4860	fm0-mac4	fm0-mac5
B4420	fm0-mac2	fm0-mac3
T4240	fm0-mac1	fm0-mac0
T2080	fm0-mac2	fm0-mac3

If you need the *ipsec_offload* application to run on different traffic ports than the ones listed above, you will need to

1. adapt the XML port configuration file for your platform and
2. adapt the DTS file that Linux kernel boots up with.

Please refer to the paragraph **Application start-up and configuration** for further details.

7.4.2.5.2.2 Application Start-up and Configuration

In this section you will find out how to run the *ipsec_offload* application. All the examples below are given for a B4860QDS board and assume the standard SDK environment is being used (with the standard RCW, u-boot, *-usdpaa-shared-interfaces.dts).

Note: When running on a different platform, please modify the parameters accordingly as shown in the tables below.

Custom scenarios that alter the default configurations involve changes that are out of the scope of this document.

Setup the environment

Use the steps described in Appendix A - **Compiling the device tree for IPsec offload** to generate a device tree binary file. Boot the board with the compiled kernel (`arch/powerpc/boot/uImage`) and the DTB file.

To enable the scatter-gather support in the DPAA Ethernet driver add the following boot arguments in uBoot:

```
setenv othbootargs "fsl_fm_max_frm=9600"
```

Setup USDPAAs configurations and PCD resources

The configuration xml and the PCD xml vary according to platform. Choose the appropriate files for your platform as per the table below. For running the application in single mode or with multiple instances consult the corresponding sections of the user guide.

```
export DEF_CFG_PATH="/usr/etc/ipsec_offload_config_b4860.xml"  
export DEF_PCD_PATH="/usr/etc/ipsec_offload_pcd_b4.xml"  
export DEF_SWP_PATH="/usr/etc/ipsec_offload_swp.xml"  
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"
```

Table 115. XML config files

SOC	DEF_CFG_PATH	DEF_PCD_PATH
P4080	/usr/etc/ ipsec_offload_config_p4080.xml	/usr/etc/ipsec_offload_pcd_p4.xml
P2041	/usr/etc/ ipsec_offload_config_p2041.xml	/usr/etc/ipsec_offload_pcd_p2.xml
B4860	/usr/etc/ ipsec_offload_config_b4860.xml	/usr/etc/ipsec_offload_pcd_b4.xml
B4420	/usr/etc/ ipsec_offload_config_b4420.xml	/usr/etc/ipsec_offload_pcd_b4.xml
T4240	/usr/etc/ ipsec_offload_config_t4240.xml	/usr/etc/ipsec_offload_pcd_t4.xml
T2080	/usr/etc/ ipsec_offload_config_t2080.xml	/usr/etc/ipsec_offload_pcd_t2.xml

Start the application

Following is a brief description of ipsec_offload's parameters and an explanation on how to set them:

-c: the configuration file for the platform (platform specific - see table above)

-p: policy file

--vif: virtual inbound interface index

--vof: virtual outbound interface index

--vipsec: virtual interface for host to host tunnels

These interfaces are used by Linux to receive and send local traffic - IP traffic matching Linux IP addresses and non-IP traffic like ARP.

The VIF and VOF parameters are always macless nodes. To choose what interfaces to use, look for interfaces with one of the following ethernet addresses:

```
00:11:22:33:44:55 or 00:11:22:33:44:66 or 00:11:22:33:44:77
```

Any combination of these nodes can be used.

The VIPSEC can be either a macless interface or an oNIC interface (for B4 and T4 platforms only). The ipsec_offload oNIC interface always has 00:11:22:33:44:88 as an ethernet address.

Note: the virtual interfaces parameters have to be provided in the following order: vif, vof, vipsec.

--ib-loop: configures the inbound port in loop mode - all frames received on the Tx port will be fed back to the Rx port

To run the application on B4860QDS, execute the following command:

```
/usr/bin/ipsec_offload -c /usr/etc/ipsec_offload_config_b4860.xml -p /usr/etc/
ipsec_offload_policy.xml --vif macless0 --vof eth1 --vipsec eth2 --ib-loop
```

To shut down the application you can type the command *quit* in the application console. At shutdown the application flushes its offloading tables, but it does not flush the items that were configured in the Linux kernel. Therefore, in case you have performed additional route, neighbor or IPsec configurations using the Linux tools it is more than a good idea to revert/flush them as well using the appropriate commands once the application was shut down.

IMPORTANT NOTE: Multiple SA's can be configured but if the VIF interface is UP and has an IP on it, all the SA's for outbound direction should have that IP as tunnel source, and all the SA's for inbound should have that VIF IP as destination. It is possible to create multiple SA's in which the IP has to be identical but vary the SPI, encryption or authentication keys or algorithm and so on. This constraint comes from the fact that VIF IP has to be the tunnel IP and this information is used to distribute SA's on different IPsec instances.

EXAMPLE:

```
VIF eth3
ifconfig eth3 192.168.60.1 netmask 255.255.255.0 up

add 192.168.60.1 192.168.50.1 esp 0x901 -m tunnel -E 3des-cbc "abcdefghijklmnopqrstuvwxyabplX" -A hmac-
sha1 "abcdefghijklmnopqrstuwx0";
add 192.168.60.1 192.168.50.2 esp 0x902 -m tunnel -E 3des-cbc "abcdefghijklmnopqrstuvwxyabply" -A hmac-
sha1 "abcdefghijklmnopqrstuwxM";
add 192.168.60.1 192.168.50.3 esp 0x903 -m tunnel -E 3des-cbc "abcdefghijklmnopqrstuvwxyabplZ" -A hmac-
sha1 "abcdefghijklmnopqrstuwxP";

Outbound
spdadd 172.13.0.1/32[any] 172.14.0.1/32[any] udp -P out ipsec esp/tunnel/192.168.60.1-192.168.50.1/
require;
spdadd 172.13.0.2/32[any] 172.14.0.2/32[any] udp -P out ipsec esp/tunnel/192.168.60.1-192.168.50.2/
require;
spdadd 172.13.0.3/32[any] 172.14.0.3/32[any] udp -P out ipsec esp/tunnel/192.168.60.1-192.168.50.3/
require;

Inbound
spdadd 172.14.0.1/32[any] 172.13.0.1/32[any] udp -P in ipsec esp/tunnel/192.168.50.1-192.168.60.1/
require;
spdadd 172.14.0.2/32[any] 172.13.0.2/32[any] udp -P in ipsec esp/tunnel/192.168.50.2-192.168.60.1/
require;
spdadd 172.14.0.3/32[any] 172.13.0.3/32[any] udp -P in ipsec esp/tunnel/192.168.50.3-192.168.60.1/
require;
```

Following script can be used as an example:

```
#!/bin/sh
echo "flush;" | setkey -c
echo "spdflush;" | setkey -c

for i in `seq $1`
do
    spi=`expr 201 + $i`
```

```
echo "spdadd 172.16.0.$i/32[any] 172.17.0.$i/32[any] udp
-P out ipsec esp/tunnel/192.168.100.1-192.168.200.$i/require;" | setkey -c

echo "spdadd 172.17.0.$i/32[any] 172.16.0.$i/32[any] udp
-P in ipsec esp/tunnel/192.168.200.$i-192.168.100.1/require;" | setkey -c

echo "add 192.168.100.1 192.168.200.$i esp 0x$spi
-E 3des-cbc \"abcdefghipqrstuvwxyzabcde\"
-A hmac-sha1 \"abcdefghipqrstuvwxyzabcde\";" | setkey -c

echo "add 192.168.200.$i 192.168.100.1 esp 0x$spi
-E 3des-cbc \"abcdefghipqrstuvwxyzabcde\"
-A hmac-sha1 \"abcdefghipqrstuvwxyzabcde\";" | setkey -c

done
```

Application configuration for IPsec

IPSec can be configured for offloading using setkey tool. To add an SA and two corresponding SPs add the following lines in a setkey.conf file:

```
flush;
spdflush;
add 192.168.100.1 192.168.200.1 esp 0x201
-E 3des-cbc "abcdefghipqrstuvwxyzabcde"
-A hmac-sha1 "abcdefghipqrstuvwxyzabcde";
spdadd 172.16.0.1/32[any] 172.17.0.1/32[any] udp
-P out ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
spdadd 172.17.0.1/32[any] 172.16.0.1/32[any] udp
-P in ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
```

Now run the command:

```
setkey -f setkey.conf
```

These commands create an ESP tunnel with the following endpoints: IP src 192.168.100.1 – IP dst 192.168.200.1, SPI 0x201, 3DES-CBC encryption and HMAC-SHA1 authentication. The UDP traffic coming from 172.16.0.1 going to 172.17.0.1 will be tunneled using the defined tunnel.

To configure Linux interfaces, neighboring and routing for the loopback setup run the following commands:

```
#enable IP forwarding
echo "1" > /proc/sys/net/ipv4/ip_forward
#inbound interface
ifconfig macless0 192.168.100.1 netmask 255.255.255.0 up
echo "1" > /proc/sys/net/ipv4/conf/macless0/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/macless0/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/macless0/accept_local
#outbound interface
ifconfig eth1 172.16.0.254 netmask 255.255.0.0 up
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth1/accept_local
#route to tunnel destination
route add -net 192.168.200.0/24 gw 192.168.100.254 dev macless0
#route to tunneled traffic destination - outbound interface in loop mode
route add -net 172.17.0.0/16 gw 172.16.0.1 dev eth1
#static neigh entry for loopback mode
```

```
ip neigh add 192.168.100.254 lladdr <inbound_port_mac_address> dev macless0
```

74.2.5.2.3 Running Traffic

Assuming that the traffic is generated with a directly connected Linux box, you need to configure the Linux box to output frames for 172.16.0.1 on the connected interface (ethX):

```
ifconfig <ethX> 172.16.0.2 netmask 255.255.0.0 up
```

You can use the hping tool to generate UDP packets and tcpdump to capture traffic on interface ethX. Each sent frame should be returned by the application.

```
hping -2 -s 2152 -p 2152 -c 1 172.17.0.1
```

At any point you can use the following commands to show packet and byte statistics.

- ```
sa_stats <sa-id>
```

This command shows the statistics for a particular SA.

- ```
ipsec_stats <ipsec-instance-id>
```

This command shows the global IPsec statistics for the specified IPsec instance. This includes the miss counters for the inbound and outbound direction.

74.2.5.3 ipsec_offload application flow

The application follows PPAC/PPAM design. PPAM is responsible for initialization and for forwarding received frames (UDP destination port 2152) between outbound port and outbound/inbound offline ports. The following tasks are part of the initialization:

- Get command line configuration parameters
- Allocate memory and create buffer pools for additional required buffer pools (i.e – IP fragmentation, IP reassembly)
- Compile and apply PCD model for the application
- Init IPsec offloading component
- Start XFRM and neighbouring notification processing
- Mappings required by USDPAAs to forward frames between ports

Offloading work is performed by XFRM and neighbour notifications processing threads. Each thread listens on a specific Netlink socket and receives messages containing the event and associated information. This information is used to configure IPsec offloading datapath.

74.2.5.4 IPSEC tunnel setup using ipsec offload application and Strongswan with IKEv2

The application can work in conjunction with strongswan client using IKEv2 protocol. The figure below illustrates a possible setup that creates a tunnel between DUT and REF devices.

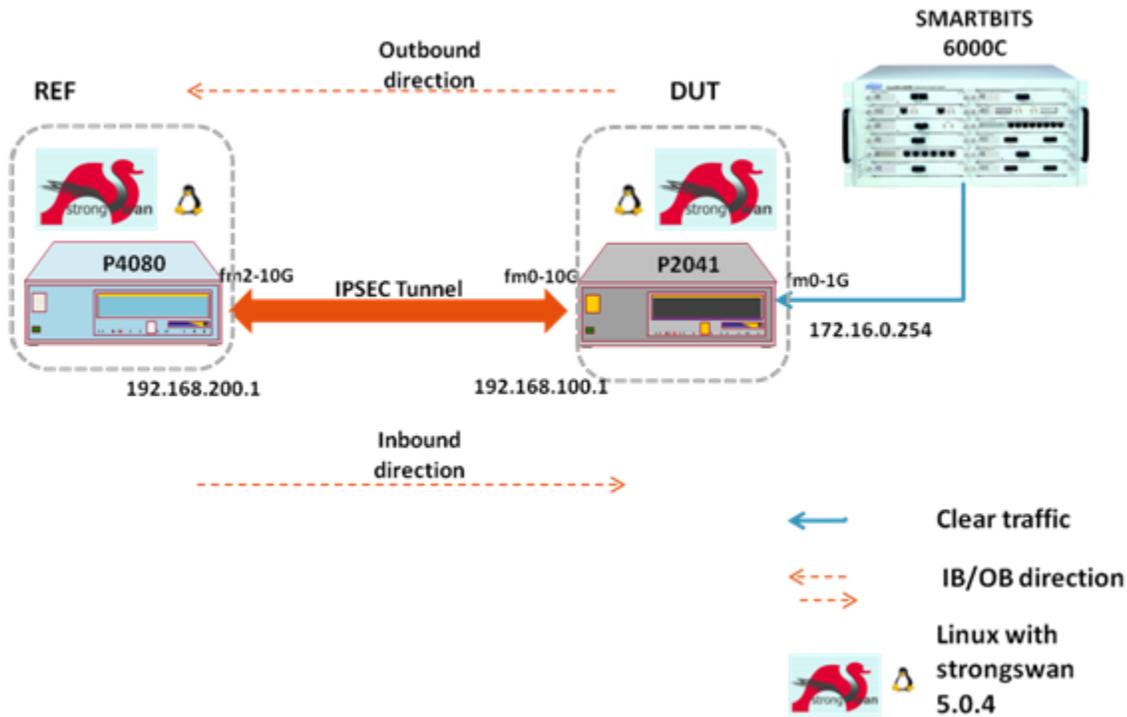


Figure 131. IPSEC tunnel setup with Strongswan and IKEv2

REF can be any client machine and DUT can be any board with **DPAA capabilities**. The **smartbit** can also be replaced by another client machine.

The tunnel is negotiated by both machines and the security associations created on DUT, are offloaded. On **outbound direction**, traffic is sent from **SMARTBITS**, encrypted on DUT and then decrypted on the REF.

On **inbound direction** traffic can be sent from REF using for instance *wget* application:

```
wget http://172.16.0.1
```

7.4.2.5.4.1 REF (Linux with Strongswan 5.0.4) configuration

- ipsec.conf (configuration file used by strongswan):

```
conn %default
    ikelifetime=60m
    keylife=20m
    rekeymargin=3m
    keyingtries=1
    authby=secret
    keyexchange=ikev2
    mobike=no

conn ref
    left=192.168.200.1
    leftsubnet=172.17.0.0/16
    right=192.168.100.1
    rightsubnet=172.16.0.0/16
    auto=add
    reauth=no
    esp=3des-sha1
```

- ipsec.secrets(pre shared secret used by strongswan):

```
# /etc/ipsec.secrets - strongSwan IPsec secrets file

192.168.100.1 192.168.200.1 : PSK 0sv+NkxY9LLZvwj4qCC2o/gGrWDF2d21jL
```

To start the IKEv2 daemon on REF, the following command will be executed:

```
ipsec start
```

Notes:

No tunnels are created yet.

The interfaces and route configuration for REF machine is the following (it can be different according to the needs):

```
fm2-10g  Link encap:Ethernet  HWaddr 00:04:9f:00:03:09
         inet addr:192.168.200.1  Bcast:192.168.200.255  Mask:255.255.255.0
         inet6 addr: fe80::204:9fff:fe00:309/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:1496 errors:0 dropped:304 overruns:0 frame:0
         TX packets:785 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:1000
         RX bytes:344650 (336.5 KiB)  TX bytes:112250 (109.6 KiB)
         Memory:fe5f0000-fe5f0fff

fm2-10g:0 Link encap:Ethernet  HWaddr 00:04:9f:00:03:09
         inet addr:192.168.100.254  Bcast:192.168.100.255  Mask:255.255.255.0
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         Memory:fe5f0000-fe5f0fff

Kernel IP routing table
Destination      Gateway         Genmask         Flags Metric Ref    Use Iface
172.16.0.0       192.168.100.1  255.255.0.0    UG    0      0      0 fm2-10g
192.168.1.0     0.0.0.0        255.255.255.0  U      0      0      0 eth0
192.168.100.0   0.0.0.0        255.255.255.0  U      0      0      0 fm2-10g
192.168.200.0  0.0.0.0        255.255.255.0  U      0      0      0 fm2-10g
```

74.2.5.4.2 DUT (Linux with Strongswan 5.0.4) configuration

- ipsec.conf (configuration file used by strongswan):

```
conn %default
    ikelifetime=120m
    keylife=60m
    rekeymargin=3m
    keyingtries=1
    authby=secret
    keyexchange=ikev2
    mobike=no

conn dut
    left=192.168.100.1
    leftsubnet=172.16.0.0/16
    reauth=no
    right=192.168.200.1
    rightsubnet=172.17.0.0/16
    auto=start
```

```
#lifetime=360s
#rekeyfuzz=0%
#margintime=330s
#rekey=yes
esp=3des-sha1
```

- ipsec.secrets(pre shared secret used by strongswan):

```
# /etc/ipsec.secrets - strongSwan IPsec secrets file

192.168.100.1 192.168.200.1 : PSK 0sv+NkxY9LLZvwj4qCC2o/gGrWDF2d21jL
```

To start the IKEv2 daemon on DUT, the following command will be executed:

```
ipsec start
```

Notes:

Tunnels will be created -due to *auto=start* option.

The daemon must be started after the ipsec offload application

The interfaces and route configuration for DUT machine is the following (it can be different according to the needs):

```
eth0      Link encap:Ethernet  HWaddr 00:04:9f:02:09:fe
          inet addr:192.168.100.1  Bcast:192.168.100.255  Mask:255.255.255.0
          inet6 addr: fe80::204:9fff:fe02:9fe/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:6 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:328 (328.0 B)

eth1      Link encap:Ethernet  HWaddr 00:04:9f:02:09:fa
          inet addr:172.16.0.254  Bcast:172.16.255.255  Mask:255.255.0.0
          inet6 addr: fe80::204:9fff:fe02:9fa/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3 errors:6 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:238 (238.0 B)

Kernel IP routing table
Destination      Gateway          Genmask         Flags Metric Ref    Use Iface
0.0.0.0          192.168.1.1     0.0.0.0         UG    0      0      0 eth2
172.16.0.0       0.0.0.0         255.255.0.0     U     0      0      0 eth1
172.17.0.0       192.168.100.254 255.255.0.0     UG    0      0      0 eth0
192.168.0.0      0.0.0.0         255.255.252.0   U     0      0      0 eth2
192.168.100.0    0.0.0.0         255.255.255.0   U     0      0      0 eth0
192.168.200.0    192.168.100.254 255.255.255.0   UG    0      0      0 eth0
```

74.2.5.4.3 Running ipsec offload application on DUT using 10G port setup

In the proposed setup, DUT and REF have a 10G connection. In this situation the ipsec offload application must be started using a **1G port for outbound** and a **10G port for inbound**.

Before starting the application, the xml configuration file that defines the ethernet ports, must be modified - in order to contain the 10G port.

The code snippet from below shows the modification:

```
diff --git a/apps/ipsec_offload/ipsec_offload_config_p2041.xml b/apps/ipsec_offload/
ipsec_offload_config_p2041.xml
index 61bfdac..1d34bb1 100644
--- a/apps/ipsec_offload/ipsec_offload_config_p2041.xml
+++ b/apps/ipsec_offload/ipsec_offload_config_p2041.xml
@@ -38,9 +38,9 @@

<cfgdata>
    <config>
-       <engine name="fm1">
-           <port type="1G" number="2" policy="ib_rx_policy"/>
+       <engine name="fm0">
+           <port type="10G" number="0" policy="ib_rx_policy"/>
+           <port type="OFFLINE" number="1" policy="ib_oh_post_policy"/>
+           <port type="OFFLINE" number="2" policy="ob_oh_pre_policy"/>
+           <port type="OFFLINE" number="3" policy="ob_oh_post_policy"/>
```

Note:

The port numbers may change due to different hardware configuration.

The ipsec offload application will be started using the following commands:

```
export DEF_CFG_PATH="/usr/etc/ipsec_offload_config_p2041.xml"
export DEF_PCD_PATH="/usr/etc/ipsec_offload_pcd_p2.xml"
export DEF_SWP_PATH="/usr/etc/ipsec_offload_swp.xml"
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"

/usr/bin/ipsec_offload \
-c /usr/etc/ipsec_offload_config_p2041.xml \
-p /usr/etc/ipsec_offload_policy.xml \
-s /usr/etc/ipsec_offload_swp.xml \
--fm 0 \
--ob_eth 1,1 --ib_eth 0,2 \
--ib-oh 1 --ob-oh-pre 2 --ob-oh-post 3 \
--max-sa 16 --vif eth0 --vof eth1 --vipsec eth2 --mtu-pre-enc 500
```

Notes:

max-sa parameter must reflect the number of SA pairs. Looking in the pcd file, there are defined 16 entries for inbound esp_cc classification table. In this case there will be 16 security associations. Interfaces vif, vof and vipsec may change due to different hardware configuration.

From another console, after ipsec offload startup, the following commands will be executed:

```
echo "1" > /proc/sys/net/ipv4/ip_forward
ifconfig eth0 192.168.100.1 netmask 255.255.255.0 up
echo "1" > /proc/sys/net/ipv4/conf/eth0/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth0/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth0/accept_local
ifconfig eth1 172.16.0.254 netmask 255.255.0.0 up
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth1/accept_local
```

```
route add -net 192.168.200.0/24 gw 192.168.100.254 dev eth0
route add -net 172.17.0.0/16 gw 192.168.100.254 dev eth0
```

Start strongswan:

ipsec start

Note:

A tunnel will be negotiated between DUT and REF.

7.4.2.5.4.4 Rekeying process with Strongswan and ipsec offload

This chapter shortly presents how to enable the rekey process using strongswan. (For more details please consult *strongswan wiki page*)

When a security association expires (after a period of time or after a number of processed packets/bytes) the operating system will notify strongswan. The expired security associations will be updated (new key material will be generated).

Ipsec offload application is able to catch the notifications and will update the offloaded SAs. The DPAA offload function used for rekeying, guarantees that no packets will be lost during this process.

To enable rekeying on DUT (see **DUT configuration** chapter), uncomment (remove "#") the following lines:

```
#lifetime=360s
#rekeyfuzz=0%
#margintime=330s
#rekey=yes
```

before starting strongswan and ipsec offload application.

The above lines will cause the created tunnels to expire at every 30 seconds (For more details please consult *strongswan wiki page* for rekeying process).

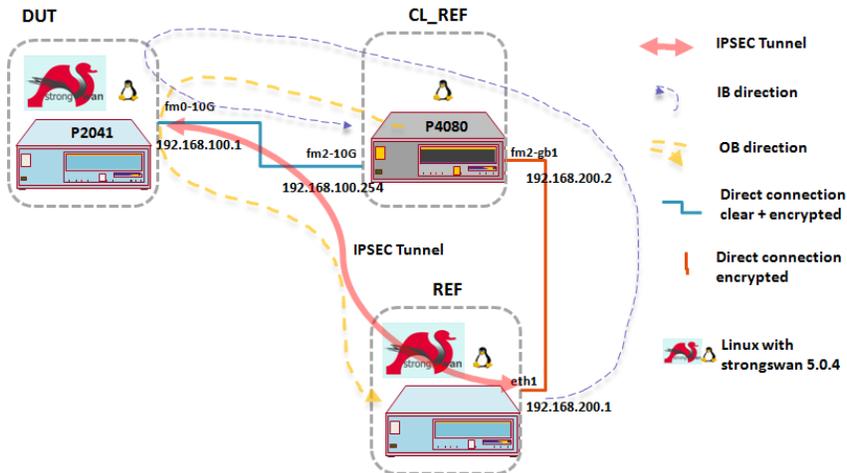
Notes:

On inbound direction, traffic must be injected before the 30 seconds interval expires. Otherwise the security association that has expired, will stay in rekey process.

7.4.2.5.4.5 Ipsec offload application in single port configuration

Configuration setup

The figure below represents the ipsec tunnel setup using one port - for inbound and outbound directions.



REF can be any machine and DUT can be any board with **offload capabilities**. CL_REF can be any machine with a 10G port available. The difference between this setup and the previous depicted setup is that the DUT machine has only one port available on the offload path, on which will be applied both policies - for inbound and outbound directions.

On outbound direction traffic is sent from CL_REF with a traffic generator application (e.g wget or hping), encrypted on DUT sent back on CL_REF, forwarded to REF where is decrypted.

On inbound direction traffic is sent from REF with a traffic generator application (e.g wget or hping), encrypted on REF, forwarded by CL_REF , decrypted on DUT and sent back to CL_REF

Note that the tunnel is created between DUT and REF. CL_REF will be used for forwarding.

Strongswan daemon is configured in the same way as in the previous setup - both for DUT and REF.

DUT configuration

The following commands will be run on DUT after application start up but before starting strongswan:

```
echo "1" > /proc/sys/net/ipv4/ip_forward
ifconfig eth0 192.168.100.1 netmask 255.255.255.0 up
echo "1" > /proc/sys/net/ipv4/conf/eth0/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth0/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth0/accept_local
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth1/accept_local
route add -net 192.168.200.0/24 gw 192.168.100.254 dev eth0
route add -net 172.16.0.0/16 gw 192.168.100.254 dev eth0
ip neigh add 192.168.100.254 lladdr [CL_REF mac addr] dev eth0
```

CL_REF mac addr represents the ethernet address of the CL_REF interface that is connected with DUT port

Route configuration on DUT:

```
root@p2041rdb:/root# route
Kernel IP routing table
Destination      Gateway          Genmask          Flags Metric Ref    Use Iface
default          192.168.1.1     0.0.0.0          UG    0     0      0 eth2
172.16.0.0       192.168.100.254 255.255.0.0     UG    0     0      0 eth0
192.168.0.0      *                255.255.252.0   U     0     0      0 eth2
192.168.100.0    *                255.255.255.0   U     0     0      0 eth0
192.168.200.0    192.168.100.254 255.255.255.0   UG    0     0      0 eth0
```

REF configuration

```
eth1      Link encap:Ethernet HWaddr A0:36:9F:19:FE:14
          inet addr:192.168.200.1 Bcast:192.168.200.255 Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:335 errors:0 dropped:0 overruns:0 frame:0
          TX packets:140421 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:39930 (38.9 KiB) TX bytes:36437402 (34.7 MiB)

eth1:0    Link encap:Ethernet HWaddr A0:36:9F:19:FE:14
          inet addr:172.17.0.2 Bcast:172.17.255.255 Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

192.168.100.0 192.168.100.254 255.255.255.0 UG 0 0 0 eth1
192.168.100.0 *                255.255.255.0 U 0 0 0 eth1
10.171.81.0   *                255.255.255.0 U 0 0 0 eth0
192.168.200.0 *                255.255.255.0 U 0 0 0 eth1
172.16.0.0    192.168.200.1 255.255.0.0     UG 0 0 0 eth1
default      10.171.81.254 0.0.0.0          UG 0 0 0 eth0
```

CL_REF configuration

```
fm2-10g    Link encap:Ethernet HWaddr 00:10:18:BA:E4:05
          inet addr:192.168.100.254 Bcast:192.168.100.255 Mask:255.255.255.0
          inet6 addr: fe80::210:18ff:feba:e404/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:9000 Metric:1
          RX packets:1100 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7691 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:148952 (145.4 KiB) TX bytes:3449552 (3.2 MiB)
          Interrupt:59 Memory:d6000000-d6012800

fm2-10g:0  Link encap:Ethernet HWaddr 00:10:18:BA:E4:05
          inet addr:172.16.0.1 Bcast:172.16.255.255 Mask:255.255.0.0
          UP BROADCAST RUNNING MULTICAST MTU:9000 Metric:1
          Interrupt:59 Memory:d6000000-d6012800

fm2-gb1    Link encap:Ethernet HWaddr 00:10:18:BA:E4:04
          inet addr:192.168.200.2 Bcast:192.168.200.255 Mask:255.255.255.0
          inet6 addr: fe80::210:18ff:feba:e404/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
          RX packets:942 errors:0 dropped:0 overruns:0 frame:0
          TX packets:940 errors:0 dropped:0 overruns:0 carrier:0
```

```
collisions:0 txqueuelen:1000
RX bytes:1981970 (1.8 MiB) TX bytes:215370 (210.3 KiB)
Interrupt:67 Memory:d8000000-d8012800
```

Kernel IP routing table

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
192.168.100.0	*	255.255.255.0	U	0	0	0	fm2-10g
10.171.81.0	*	255.255.255.0	U	0	0	0	eth0
192.168.200.0	*	255.255.255.0	U	0	0	0	fm2-gb1
172.16.0.0	*	255.255.0.0	U	0	0	0	fm2-10g
172.17.0.0	192.168.100.1	255.255.0.0	UG	0	0	0	fm2-10g
default	10.171.81.254	0.0.0.0	UG	0	0	0	eth0

Notes:

Forwarding must be enabled between fm2-10g and fm2-gb1.

Running the ipsec offload applicaion in single port configuration

For this setup, the single port xml files will be used.

The config xml for single port is like the snippet below:

```
<cfgdata>
  <config>
    <engine name="fm0">
-       <port type="1G" number="2" policy="ib_ob_rx_policy"/>
+       <port type="10G" number="0" policy="ib_ob_rx_policy"/>
      <port type="OFFLINE" number="1" policy="ib_oh_post_policy"/>
      <port type="OFFLINE" number="2" policy="ob_oh_pre_policy"/>
      <port type="OFFLINE" number="3" policy="ob_oh_post_policy"/>
    </engine>
  </config>
</cfgdata>
```

Note:

The port numbers may change due to different hardware configuration.

The ipsec offload application will be started using the following commands:

```
export DEF_CFG_PATH="/usr/etc/ipsec_offload_config_p2041_1p.xml"
export DEF_PCD_PATH="/usr/etc/ipsec_offload_pcd_p2_1p.xml"
export DEF_SWP_PATH="/usr/etc/ipsec_offload_swp.xml"
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"
/usr/bin/ipsec_offload \
-c /usr/etc/ipsec_offload_config_p2041_1p.xml \
-p /usr/etc/ipsec_offload_policy_1p.xml \
-s /usr/etc/ipsec_offload_swp.xml \
--vif eth0 --vof eth0 --vipsec eth3
```

Notes:

Interfaces vif, vof and vipsec may change due to different hardware configuration.

The number of maximum SA pairs can be determined by looking in the pcd file, there are defined 16 entries for inbound esp_cc classification table. In this case there will be 16 security associations.

From another console, after ipsec offload startup, the following commands will be executed:

```
echo "1" > /proc/sys/net/ipv4/ip_forward
ifconfig eth0 192.168.100.1 netmask 255.255.255.0 up
echo "1" > /proc/sys/net/ipv4/conf/eth0/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth0/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth0/accept_local
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth1/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth1/accept_local
route add -net 192.168.200.0/24 gw 192.168.100.254 dev eth0
route add -net 172.16.0.0/16 gw 192.168.100.254 dev eth0
ip neigh add 192.168.100.254 lladdr 00:10:18:BA:E4:05 dev eth0
```

Start strongswan:

ipsec start

Note:

A tunnel will be negotiated between DUT and REF.

Sending traffic on outbound direction

wget was used for sending traffic. Any other application that can generate tcp/udp packets, can be used.

```
wget --no-proxy http://172.17.0.1/
```

The tcpdump output on CL_REF will be similar to this:

```
8:12:14.823907 00:10:18:ba:e4:05 (oui Unknown) > 00:04:9f:02:09:fb (oui Unknown), ethertype IPv4 (0x0800), length 74: 172.16.0.1.52699 > 172.17.0.1.http
18:12:14.823988 00:04:9f:02:09:fb (oui Unknown) > 00:10:18:ba:e4:05 (oui Unknown), ethertype IPv4 (0x0800), length 126: 192.168.100.1 > 192.168.200.1: ESP(spi=0xc718bd51,seq=0x13), length 92
```

On REF the tcpdump output will be similar to:

```
18:23:18.829555 00:10:18:ba:e4:04 (oui Unknown) > a0:36:9f:19:fe:14 (oui Unknown), ethertype IPv4 (0x0800), length 126: 192.168.100.1 > 192.168.200.1: ESP(spi=0xc115218e,seq=0xe), length 92
18:23:18.829555 00:10:18:ba:e4:04 (oui Unknown) > a0:36:9f:19:fe:14 (oui Unknown), ethertype IPv4 (0x0800), length 74: 172.16.0.1.51413 > 172.17.0.1.http
```

Sending traffic on inbound direction

wget was used for sending traffic. Any other application that can generate tcp/udp packets, can be used.

```
wget --no-proxy http://172.16.0.1
```

The tcpdump output on REF will be similar to this:

```
Connecting to 172.16.0.1:80... 18:22:34.220792 a0:36:9f:19:fe:14 (oui Unknown) > 00:10:18:ba:e4:04 (oui Unknown), ethertype IPv4 (0x0800), length 126: 192.168.200.1 > 192.168.100.1: ESP(spi=0xc4163da8,seq=0x2), length 92
```

On CL_REF the tcpdump output will be similar to:

```
18:09:07.192481 00:10:18:ba:e4:05 (oui Unknown) > 00:04:9f:02:09:fb (oui Unknown), ethertype IPv4
(0x0800), length 126: 192.168.200.1 > 192.168.100.1: ESP(spi=0xc0ce4d0a,seq=0x12), length 92
18:09:07.192625 00:04:9f:02:09:fb (oui Unknown) > 00:10:18:ba:e4:05 (oui Unknown), ethertype IPv4
(0x0800), length 74: 172.17.0.2.39485 > 172.16.0.1.http
```

Single port with aggregation option

In single port configuration, ipsec offload application supports an operating mode called aggregation.

Let's suppose there are three SAs on inbound direction: SAi1, SAi2 and SAi3 and one SA on outbound: SAo1. Encrypted traffic received on inbound, would match one of the previous SAs- SAi1 to SAi3. Consider that the decrypted traffic have the following selectors:

```
for SAi1: 172.16.0.0/16 172.17.0.0/16 udp
for SAi2: 172.18.0.0/16 172.19.0.0/16 udp
for SAi3: 172.20.0.0/16 172.21.0.0/16 udp
```

Consider that on outbound direction the policies corresponding to SAo1 will have the above selectors.

If aggregation is enabled, decrypted traffic from SAi1 to SAi3 will be forwarded to the outbound offline port pre-encryption. The forwarded traffic will match the outbound policies , will be encrypted with the SAo1 and sent on the Tx port.

For **aggregation mode**, the ipsec offload application will be started using the following commands:

```
export DEF_CFG_PATH="/usr/etc/ipsec_offload_config_p2041_1p.xml"
export DEF_PCD_PATH="/usr/etc/ipsec_offload_pcd_p2_1p.xml"
export DEF_SWP_PATH="/usr/etc/ipsec_offload_swp.xml"
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"
/usr/bin/ipsec_offload \
-c /usr/etc/ipsec_offload_config_p2041_1p.xml \
-p /usr/etc/ipsec_offload_policy_1p.xml \
-s /usr/etc/ipsec_offload_swp.xml \
--vif eth0 --vof eth0 --vipsec eth3 --aggreg
```

Note:

Aggregation is available only in single port configuration. In this situation, the decrypted traffic must match the configured policies on outbound direction.(see aforementioned explanation)

Sample setkey configuration file:

```
#!/usr/sbin/setkey -f

# Flush the SAD and SPD
flush;
spdflush;

add 192.168.200.1 192.168.100.1 esp 0x201 -m tunnel
-E 3des-cbc 0xbaba42c2c0d033052267d10a2683325c56fa5d35c0b202d8
-A hmac-sha1 0xdae1554cc000111b208fad023a8feffcb4e421;

add 192.168.200.1 192.168.100.2 esp 0x202 -m tunnel
-E 3des-cbc 0xbaba42c2c0d033052267d10a2683325c56fa5d35c0b202d8
-A hmac-sha1 0xdae1554cc000111b208fad023a8feffcb4e421;

add 192.168.200.1 192.168.100.3 esp 0x203 -m tunnel
-E 3des-cbc 0xbaba42c2c0d033052267d10a2683325c56fa5d35c0b202d8
```

```
-A hmac-sha1 0xdadae1554cc000111b208fad023a8feffcb4e421;  
  
add 192.168.100.1 192.168.200.1 esp 0x204 -m tunnel  
-E 3des-cbc 0x3a3842c2c0d033052267d10a2683325c56fa5d35c0b202d8  
-A hmac-sha1 0xf1a9e1554cc000111b208fad023a8feffcb4e421;  
  
spdadd 172.16.0.0/16[any] 172.17.0.0/16[any] udp  
-P in ipsec esp/tunnel/192.168.200.1-192.168.100.1/require;  
  
spdadd 172.16.0.0/16[any] 172.17.0.0/16[any] udp  
-P out ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;  
  
spdadd 172.18.0.0/16[any] 172.19.0.0/16[any] udp  
-P in ipsec esp/tunnel/192.168.200.1-192.168.100.2/require;  
  
spdadd 172.18.0.0/16[any] 172.19.0.0/16[any] udp  
-P out ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;  
  
spdadd 172.20.0.0/16[any] 172.21.0.0/16[any] udp  
-P in ipsec esp/tunnel/192.168.200.1-192.168.100.3/require;  
  
spdadd 172.20.0.0/16[any] 172.21.0.0/16[any] any  
-P out ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
```

74.2.5.5 Ipsec offload application in multiple instances configuration

Steps to initialize first instance - for FMAN 1

The following commands assume that the application runs on a P4080 board. Altering the presented port configurations involves various changes, including device tree updates, and that is not covered by this document.

For setting up the USDPAAs network configuration and PCD resources used by the application run the following commands:

```
export DEF_CFG_PATH="/usr/etc/ipsec_offload_config_p4080_1p_fman1_instance.xml "  
export DEF_PCD_PATH="/usr/etc/ipsec_offload_pcd_p4_1p.xml "  
export DEF_SWP_PATH="/usr/etc/ipsec_offload_swp.xml "  
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml "
```

Start the application with the following command:

```
/usr/bin/ipsec_offload \  
-c /usr/etc/ipsec_offload_config_p4080_1p_fman1_instance.xml \  
-p /usr/etc/ipsec_offload_policy_1p_fman1_instance.xml \  
-s /usr/etc/ipsec_offload_swp.xml \  
--vif eth4 --vof eth4 --vipsec macless0
```

Configure interfaces:

```
echo "1" > /proc/sys/net/ipv4/ip_forward  
ifconfig eth4 192.168.100.1 netmask 255.255.255.0 up
```

```

echo "1" > /proc/sys/net/ipv4/conf/eth4/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth4/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth4/accept_local
route add -net 192.168.200.0/24 gw 192.168.100.254 dev eth4
route add -net 172.17.0.0/16 gw 192.168.100.254 dev eth4
ip neigh add 192.168.100.254 lladdr 00:e0:0c:00:93:05 dev eth4
arp -s 172.16.0.1 00:11:22:22:11:00 dev eth4

```

Create tunnels and policies for the first instance:

```

add 192.168.100.1 192.168.200.1 esp 0x201 -m tunnel -E 3des-cbc "kabcdefghipqrstuvwxyzabcde" -A hmac-sha1 "abcdefghipqrstuvwxyzabcde";
add 192.168.200.1 192.168.100.1 esp 0x201 -m tunnel -E 3des-cbc "kabcdefghipqrstuvwxyzabcde" -A hmac-sha1 "abcdefghipqrstuvwxyzabcde";
spdadd 172.16.0.1/32[any] 172.17.0.1/32[any] udp -P out ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
spdadd 172.17.0.1/32[any] 172.16.0.1/32[any] udp -P in ipsec esp/tunnel/192.168.200.1-192.168.100.1/require;
spdadd 172.16.0.1/32[any] 172.17.0.1/32[any] icmp -P out ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
spdadd 172.17.0.1/32[any] 172.16.0.1/32[any] icmp -P in ipsec esp/tunnel/192.168.200.1-192.168.100.1/require;

```

Validate ESP operation

Send traffic that matches the outbound policies and see that ESP packets are created. Verify also that statistics on the outbound SA increment with the number of ESP created packets.

Send ESP packets to the board and see that they are decapsulated correctly and check also the statistics.

Step to initialize the second instance - for FMAN 0

The following commands assume that the application runs on a P4080 board. Altering the presented port configurations involves various changes, including device tree updates, and that is not covered by this document.

For setting up the USDPAA network configuration and PCD resources used by the application run the following commands:

```

export DEF_CFG_PATH="/usr/etc/ipsec_offload_config_p4080_1p_fman0_instance.xml"
export DEF_PCD_PATH="/usr/etc/ipsec_offload_pcd_p4_1p.xml"
export DEF_SWP_PATH="/usr/etc/ipsec_offload_swp.xml"
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"

```

Start the application with the following command:

```

/usr/bin/ipsec_offload \
-c /usr/etc/ipsec_offload_config_p4080_1p_fman0_instance.xml \
-p /usr/etc/ipsec_offload_policy_1p_fman0_instance.xml \
-s /usr/etc/ipsec_offload_swp.xml \

```

```
--vif eth3 --vof eth3 --vipsec eth5
```

Configure interfaces:

```
echo "1" > /proc/sys/net/ipv4/ip_forward
ifconfig eth3 192.168.60.1 netmask 255.255.255.0 up
echo "1" > /proc/sys/net/ipv4/conf/eth3/disable_policy
echo "1" > /proc/sys/net/ipv4/conf/eth3/disable_xfrm
echo "1" > /proc/sys/net/ipv4/conf/eth3/accept_local
route add -net 192.168.50.0/24 gw 192.168.60.254 dev eth3
route add -net 172.13.0.0/16 gw 192.168.60.254 dev eth3
ip neigh add 192.168.60.254 lladdr 00:e0:0c:00:93:05 dev eth3
arp -s 172.14.0.1 00:11:22:22:11:00 dev eth3
```

Create tunnels and policies for the second instance:

```
add 192.168.60.1 192.168.50.1 esp 0x901 -m tunnel -E 3des-cbc "abcdefghipqrstuvwxyzabplX" -A hmac-
sha1 "abcdefghipqrstuvwxyzab0";
add 192.168.50.1 192.168.60.1 esp 0x901 -m tunnel -E 3des-cbc "abcdefghipqrstuvwxyzabplm" -A hmac-
sha1 "abcdefghipqrstuvwxyzab";
spdadd 172.13.0.1/32[any] 172.14.0.1/32[any] udp -P out ipsec esp/tunnel/192.168.60.1-192.168.50.1/
require;
spdadd 172.14.0.1/32[any] 172.13.0.1/32[any] udp -P in ipsec esp/tunnel/192.168.50.1-192.168.60.1/
require;
spdadd 172.13.0.1/32[any] 172.14.0.1/32[any] icmp -P out ipsec esp/tunnel/
192.168.60.1-192.168.50.1/require;
spdadd 172.14.0.1/32[any] 172.13.0.1/32[any] icmp -P in ipsec esp/tunnel/192.168.50.1-192.168.60.1/
require;
```

Validate ESP operation

Send traffic that matches the outbound policies and see that ESP packets are created. Verify also that statistics on the outbound SA increment with the number of ESP created packets. Send ESP packets to the board and see that they are decapsulated correctly and check also the statistics.

IMPORTANT NOTE: Multiple SA's can be configured but if the VIF interface is UP and has an IP on it, all the SA's for outbound direction should have that IP as tunnel source, and all the SA's for inbound should have that VIF IP as destination. It is possible to create multiple SA's in which the IP has to be identical but vary the SPI, encryption or authentication keys or algorithm and so on. This constraint comes from the fact that VIF IP has to be the tunnel IP and this information is used to distribute SA's on different IPsec instances.

7.4.2.5.6 Host to host tunnels

Overview

The ipsec_offload usecase can be used to protect data flows between a pair of hosts. An IPsec tunnel can be setup to provide end to end encryption / decryption between the two hosts. The interface that the ipsec_offload uses for host to host tunnels is the one provided via the 'vipsec' command line argument.

Configuration and start-up

This section details the configuration sequence needed to enable host to host tunnels on a P4080DS board

- Run the usecase:

```
export DEF_CFG_PATH="/usr/etc/ipsec_offload_config_p4080.xml"  
export DEF_PCD_PATH="/usr/etc/ipsec_offload_pcd_p4.xml"  
export DEF_SWP_PATH="/usr/etc/ipsec_offload_swp.xml"  
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"  
/usr/bin/ipsec_offload \  
--vif macless0 --vof eth3 --vipsec eth4
```

- Configure the ethernet interfaces:

```
echo "1" > /proc/sys/net/ipv4/ip_forward  
  
ifconfig macless0 192.168.100.1 netmask 255.255.255.0 up  
echo "1" > /proc/sys/net/ipv4/conf/macless0/disable_policy  
echo "1" > /proc/sys/net/ipv4/conf/macless0/disable_xfrm  
echo "1" > /proc/sys/net/ipv4/conf/macless0/accept_local  
  
route add -net 192.168.200.0/24 gw 192.168.100.254 dev macless0  
ip neigh add 192.168.100.254 lladdr 00:04:9f:02:87:26 dev macless0  
  
ifconfig eth4 172.20.0.1 up  
echo "1" > /proc/sys/net/ipv4/conf/eth4/disable_policy  
echo "1" > /proc/sys/net/ipv4/conf/eth4/disable_xfrm  
echo "1" > /proc/sys/net/ipv4/conf/eth4/accept_local  
  
echo "1" > /proc/sys/net/ipv4/conf/macless0/arp_accept  
echo "1" > /proc/sys/net/ipv4/conf/macless0/proxy_arp
```

- Create IPsec tunnels using the following setkey script:

```
flush;  
spdf flush;  
add 192.168.100.1 192.168.200.1 esp 0x201 -m tunnel -E 3des-cbc "abcdefghipqrstuvwxyzabcde" -A  
hmac-sha1 "abcdefghipqrstuvwxyzabcde";  
add 192.168.200.1 192.168.100.1 esp 0x201 -m tunnel -E 3des-cbc "abcdefghipqrstuvwxyzabcde" -A  
hmac-sha1 "abcdefghipqrstuvwxyzabcde";  
spdadd 172.20.0.1/32[any] 172.21.0.1/32[any] icmp -P out ipsec esp/tunnel/  
192.168.100.1-192.168.200.1/require;  
spdadd 172.21.0.1/32[any] 172.20.0.1/32[any] icmp -P in ipsec esp/tunnel/  
192.168.200.1-192.168.100.1/require;  
spdadd 172.20.0.1/32[any] 172.21.0.1/32[any] udp -P out ipsec esp/tunnel/  
192.168.100.1-192.168.200.1/require;  
spdadd 172.21.0.1/32[any] 172.20.0.1/32[any] udp -P in ipsec esp/tunnel/  
192.168.200.1-192.168.100.1/require;
```

UDP and ICMP traffic going out of eth4 will be encrypted and sent via the 192.168.100.1-192.168.200.1 tunnel. UDP and ICMP traffic coming out of the 192.168.200.1-192.168.100.1 tunnel will be decrypted and the frames addressed to the vipsec interface will be forwarded accordingly.

Traffic

Connect the other end-point of the tunnel by configuring the following on a test station which is linked to the inbound port of the board.

- Configure the interfaces:

```
ifconfig eth1 172.21.0.1/16 up
ifconfig eth1:0 192.168.200.1/24 up
route add -net 172.20.0.0/16 dev eth1
route add -net 192.168.100.0/24 dev eth1
```

- Configure IPsec tunnels by running the following setkey script:

```
flush;
spdf flush;
add 192.168.100.1 192.168.200.1 esp 0x201 -m tunnel -E 3des-cbc "abcdefghijklmnopqrstuvwxyabcde" -A
hmac-sha1 "abcdefghijklmnopqrstuvwxya";
add 192.168.200.1 192.168.100.1 esp 0x201 -m tunnel -E 3des-cbc "abcdefghijklmnopqrstuvwxyabcde" -A
hmac-sha1 "abcdefghijklmnopqrstuvwxya";
spdadd 172.21.0.1/32[any] 172.20.0.1/32[any] icmp -P out ipsec esp/tunnel/
192.168.200.1-192.168.100.1/require;
spdadd 172.20.0.1/32[any] 172.21.0.1/32[any] icmp -P in ipsec esp/tunnel/
192.168.100.1-192.168.200.1/require;
spdadd 172.21.0.1/32[any] 172.20.0.1/32[any] udp -P out ipsec esp/tunnel/
192.168.200.1-192.168.100.1/require;
spdadd 172.20.0.1/32[any] 172.21.0.1/32[any] udp -P in ipsec esp/tunnel/
192.168.100.1-192.168.200.1/require;
```

Outbound traffic

On the P4080 board, configure a route to the test station – all traffic destined to that network will go through the vipsec interface:

```
route add -net 172.21.0.0/16 dev eth4
```

Send an ICMP request from the board to the test station:

```
ping 172.21.0.1 -c 1
```

Check the Tx statistics on the inbound port (eth_stats 1 1) and the outbound SA stats (sa_stats 0). They will each be incremented by one.

On the test station run a tcpdump on the eth1 interface:

```
15:33:16.196378 IP 192.168.100.1 > 192.168.200.1: ESP(spi=0x00000201,seq=0x6), length 116
15:33:16.196378 IP 172.20.0.1 > 172.21.0.1: ICMP echo request, id 1864, seq 0, length 64
15:33:16.196448 IP 192.168.200.1 > 192.168.100.1: ESP(spi=0x00000201,seq=0x14), length 116
```

The ICMP request is seen as both encrypted and then decrypted. Also the encrypted ICMP reply can be seen.

Inbound traffic

Send a ICMP request from the test station to the board:

```
ping 172.20.0.1 -c 1
```

Check the inbound SA stats (sa_stats 1). They will be incremented by one.

On the P4080 run tcpdump on the vipsec interface:

```
17:11:05.282030 IP 172.21.0.1 > 172.20.0.1: ICMP echo request, id 11607, seq 1, length 64
17:11:05.282050 IP 172.20.0.1 > 172.21.0.1: ICMP echo reply, id 11607, seq 1, length 64
```

Only clear traffic will be seen on this interface.

Host to host tunnels via oNIC

oNIC is a new Ethernet driver which is an improved solution (with CSUM offload and zero-copy) for macless. Currently it is available only on the T4 and B4 platforms.

An oNIC device has been integrated in the ipsec_offload usecase – the vipsec device can now be oNIC instead of macless. This transition is seemingly transparent - the only thing the user needs to do is to specify an oNIC node instead of a macless node as the vipsec command line parameter.

The oNIC interface can be identified from the DTS (it has the "fsl,dpa-ethernet-generic" attribute). For example for a B4 board:

```
ethernet@19 {
    compatible = "fsl,b4860-dpa-ethernet-generic", "fsl,dpa-ethernet-generic";
    fsl,qman-frame-queues-rx = <7000 2>;
    fsl,qman-frame-queues-tx = <7008 1>;
    fsl,oh-ports = <&oh2 &oh3>;
    local-mac-address = [00 11 22 33 44 88];
};
```

When running ipsec_offload pass the corresponding ethernet interface as the vipsec:

```
eth6  Link encap:Ethernet  HWaddr 00:11:22:33:44:88
      BROADCAST MULTICAST  MTU:1500  Metric:1
      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
      TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

7.4.2.6 Using the dpa_offload - the NF API offloading demo application

This application demonstrates how IPSec and IP forwarding services can be offloaded to the DPAA using the *Network Function Layer* DPAA offloading drivers extension.

7.4.2.6.1 Overview

This application shows how IPSec services, as defined by the *RFC 4301*, and IP forwarding services can be offloaded from the Linux user space by using the USDPAAs, the *Network Function Layer* and the DPAA offloading drivers. It equally

demonstrates how the *Network Function Layer* offloading services can be integrated with a standard operating system's IP forwarding and IPSec configuration tools.

This application overrides the Linux operating system's data path configuration and it offloads into DPAA:

- IPSec tunnels that the user configures by using the *setkey* tool
- IP routes that are configured by the use of the *ip route* and *ip neigh* tools
- inbound ingress rules configured by the use of the application command line interface (CLI)

NOTE

This application is a pilot and has the following limitations:

Table 116. DPAA Offloading Application Limitations

Limitation Description
This application does not yet support IPv6 IPSec tunnels & policies offloading.
This application does not yet support IPSec <i>red side fragmentation</i> offloading.
This application currently supports offloading only IPSec tunnels with one policy per direction.
This application does not yet support the IPSec DSCP/ECN copy feature.
This application does not yet support IPSec <i>inbound policy verification</i> .
This application currently supports only IPSec policies with specific L4 ports. It does not yet support the <i>ANY</i> specifier or L4 port ranges in the IPSec policy.
The application does not yet support IPSec <i>host-to-host tunnels</i> .
The application does not yet support IPSec and IP forwarding offloading using multiple instances in parallel.

7.4.2.6.2 Running dpa_offload

The dpa_offload demo application supports the following platforms:

- B4860

In order to run it you will need to use a specific device tree file that comes with the DPA offloading driver. Please read below about how to produce this device tree file for your platform.

7.4.2.6.2.1 Application environment specifications

The application can run in a real network setup configuration and inter-operate with other IPSec gateways. There is also an option to put the inbound port in loopback mode and return IPSec traffic to the outbound port. The details on how to run the application refers to this mode.

The simplest hardware setup is the one described in the **ipsec_offload Application environment specifications** section.

Table 117. Port connections

SoC	Inbound Port	Outbound Port
B4860	fm1-mac5	fm1-mac6

7.4.2.6.2.2 Application start-up and configuration

Use the steps described later in **Appendix A: Compiling the device tree for Network Function Layer offloading** to generate a device tree binary file. Boot the board with the compiled kernel, `arch/powerpc/boot/uImage`, and the DTB file.

To enable the scatter-gather support in the DPAA Ethernet driver add the following to the boot arguments:

```
setenv othbootargs "fsl_fm_max_frm=9600"
```

The following commands assume that the application runs on a B4860QDS board. When running on a different supported platform, please use the proper parameters listed in **Running dpa_offload: Application environment specifications**. For setting up the USDPAAs network configuration and PCD resources used by the application run the following commands:

```
export DEF_CFG_PATH="/usr/etc/dpa_offload_config_b4860.xml"
export DEF_POL_PATH="/usr/etc/dpa_offload_policy.xml"
export DEF_PCD_PATH="/usr/etc/dpa_offload_pcd_b4.xml"
export DEF_SWP_PATH="/usr/etc/dpa_offload_swp.xml"
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"
```

First ensure that the following device files have been created:

- `/dev/dpa_classifier`
- `/dev/dpa_stats`
- `/dev/dpa_ipsec`

Following is a brief description of `dpa_offload` application's parameters and an explanation on how to set them:

- `--vif`, virtual inbound interface name;
- `--vof`, virtual outbound interface name;
- `--vipsec`, virtual interface for host-to-host IPsec tunneling; this needs to be provided, but it is not currently used due to an application limitation;
- `--ib-loop`, set the inbound interface in loopback mode;
- `--disable-ib-ecn`, disable inbound DSCP/ECN field copy; you should always use this due to the current limitation of the application;
- `--disable-ob-ecn`, disable outbound DSCP/ECN field copy; you should always use this due to the current limitation of the application.

The *VIF* and *VOF* arguments are the names of the shared interfaces that the application is using as inbound and outbound interfaces. These are the Ethernet interface names mentioned in the **Application environment specifications** section above for the platform that you are running on.

The *VIPSEC* interface is a macless interface. Please select the interface with the following Ethernet address:

```
00:11:22:33:44:55
```

The `dpa_offload` application can be run with the following command:

```
/usr/bin/dpa_offload --vif fm1-mac5 --vof fm1-mac6 --vipsec macless0 --disable-ib-ecn --disable-ob-ecn --ib-loop
```

The output displayed by the application would be something similar with the following:

```
Found /fsl,dpaa/dpa-fman0-oh@2, Tx Channel = 80a, FMAN = 0, Port ID = 1
Found /fsl,dpaa/dpa-fman0-oh@3, Tx Channel = 80b, FMAN = 0, Port ID = 2
Found /fsl,dpaa/dpa-fman0-oh@4, Tx Channel = 80c, FMAN = 0, Port ID = 3
Found /fsl,dpaa/ethernet@8, Tx Channel = 800, FMAN = 0, Port ID = 9
Found /fsl,dpaa/ethernet@4, Tx Channel = 806, FMAN = 0, Port ID = 5
Found /fsl,dpaa/ethernet@5, Tx Channel = 807, FMAN = 0, Port ID = 6
Found /fsl,dpaa/ethernet@16, MAC-LESS node
```

Linux User Space
USDPAAs Applications

```
Found /fsl,dpaa/ethernet@17, MAC-LESS node
Found /fsl,dpaa/ethernet@18, MAC-LESS node
Found /fsl,dpaa/ethernet@19, Tx Channel = 80a, FMAN = 0, Port ID = 0
..... USDPAAs Configuration .....

Network interfaces: 5

+ Fman 0, MAC 1 (OFFLINE);
  tx_channel_id: 0x80a
  fqid_rx_hash:
    (PCD: start 0x2e00, count 1)
    (PCD: start 0x10e4, count 1)
  fqid_rx_def: 0x6f
  fqid_rx_err: 0x6e

+ Fman 0, MAC 2 (OFFLINE);
  tx_channel_id: 0x80b
  fqid_rx_hash:
    (PCD: start 0x10e6, count 1)
  fqid_rx_def: 0x69
  fqid_rx_err: 0x68

+ Fman 0, MAC 3 (OFFLINE);
  tx_channel_id: 0x80c
  fqid_rx_hash:
    (PCD: start 0x10e8, count 1)
  fqid_rx_def: 0x71
  fqid_rx_err: 0x70

+ Fman 0, MAC 5 (1G);
  mac_addr: 00:e0:0c:00:4c:04
  tx_channel_id: 0x806
  fqid_rx_hash:
    (PCD: start 0x10e0, count 1)
  fqid_rx_def: 0x59
  fqid_rx_err: 0x58
  fqid_tx_err: 0x78
  fqid_tx_confirm: 0x79
  buffer pool: (bpid=16, count=2048 size=1728, addr=0x0)

+ Fman 0, MAC 6 (1G);
  mac_addr: 00:e0:0c:00:4c:05
  tx_channel_id: 0x807
  fqid_rx_hash:
    (PCD: start 0x3e00, count 1)
    (PCD: start 0x10e2, count 1)
  fqid_rx_def: 0x5b
  fqid_rx_err: 0x5a
  fqid_tx_err: 0x7a
  fqid_tx_confirm: 0x7b
  buffer pool: (bpid=16, count=2048 size=1728, addr=0x0)

Interface fman 0, index 1: enable rx
Interface fman 0, index 2: enable rx
Interface fman 0, index 3: enable rx
Mapping Rx FQ 0x4000740:0 --> Tx FQID 319
Interface fman 0, index 5: enable rx
Interface name fml-mac5: enabled RX
Mapping Rx FQ 0x4000780:0 --> Tx FQID 321
Mapping Rx FQ 0x40007c0:0 --> Tx FQID 321
Interface fman 0, index 6: enable rx
```

```

Interface name fm1-mac6: enabled RX
Released 0 bufs to BPID 34
Released 16384 bufs to BPID 16
cpu0/0: > WARNING (FM-PCD) [CPU00, drivers/net/ethernet/freescale/fman/Peripherals/FM/Pcd/fm_kg.c:1074 BuildSchemeRegs]:
cpu0/0: baseFqid is 0.cpu0/0:
cpu0/0: > WARNING (FM-PCD) [CPU00, drivers/net/ethernet/freescale/fman/Peripherals/FM/Pcd/fm_kg.c:1074 BuildSchemeRegs]:
cpu0/0: baseFqid is 0.cpu0/0:
IB policy verification not activated
Identifying IPv4 routing tables
    0) "ob_post_ipv4_route_cc" ... cc_node=0x10c04200
    1) "ib_post_ipv4_route_cc" ... cc_node=0x10c03f70
2 tables.
Identifying IPv4 rule tables
    0) "ib_ipv4_rule_cc" ... cc_node=0x108372f8
1 tables.
Identifying IPv6 routing tables
    0) "ob_post_ipv6_route_cc" ... cc_node=0x10c041e8
    1) "ib_post_ipv6_route_cc" ... cc_node=0x10c03f58
2 tables.
Identifying IPv6 rule tables
    0) "ib_ipv6_rule_cc" ... cc_node=0x10bfdb00
1 tables.
Initializing IPv4 route tables...
    - td=rt_table_no=13 ... ccnode=0x10c04200
    - td=rt_table_no=14 ... ccnode=0x10c03f70
Initializing IPv4 rule tables...
    - td=15, ccnode=0x108372f8, ifname=fm1-mac5
Initializing IPv6 route tables...
    - td=rt_table_no=16 ... ccnode=0x10c041e8
    - td=rt_table_no=17 ... ccnode=0x10c03f58
Initializing IPv6 rule tables...
    - td=18, ccnode=0x10bfdb00, ifname=fm1-mac5
Hit Ctrl+C, send SIGINT or write quit to terminate.
>

```

To shut down the application you can type the command *quit* in the application console. At shutdown the application flushes its offloading tables, but it does not flush the items that were configured in the Linux kernel. Therefore, in case you have performed additional route, neighbor or IPsec configurations using the Linux tools it is more than a good idea to revert/flush them as well using the appropriate commands once the application was shut down.

Typing *help* in the application console will display the available application CLI commands.

7.4.2.6.2.3 Running the application

First step to take is to configure the IP addresses on the inbound and outbound interfaces of the host. For example:

```

ip addr add 192.168.100.1/24 dev fm1-mac5

ip addr add 172.16.0.254/16 dev fm1-mac6

```

Before continuing, you need to bring down the VIF interface so that it doesn't interfere with the IPsec policies setup:

```
ip link set dev fm1-mac5 down
```

IPSec can be configured for offloading using the `setkey` tool. To create an IPSec tunnel (SA) and two security policies (one for inbound and one for outbound) add the following lines to a `setkey.conf` text file:

```
flush;
spdf flush;

add 192.168.100.1 192.168.200.1 esp 0x201
-E 3des-cbc "abcdefghijklmnopqrstuvwxyzabcde"
-A hmac-sha1 "abcdefghijklmnopqrstuvwxyz";

spdadd 172.16.0.1/32 [1230] 172.17.0.1/32 [2600] udp
-P out ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
spdadd 172.17.0.1/32 [2600] 172.16.0.1/32 [1230] udp
-P in ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
```

To execute the script and set up the tunnels, you can run the command:

```
setkey -f setkey.conf
```

These commands will create an ESP tunnel with the following endpoints: IP src 192.168.100.1 - IP dst 192.168.200.1, SPI 0x201, 3DES-CBC encryption and HMAC-SHA1 authentication. The UDP traffic coming from 172.16.0.1, with UDP source port 2130 and going to 172.17.0.1, with UDP destination port 2600 will be directed through the defined tunnel.

Once the IPSec tunnels are set up, we can bring the VIF interface back up:

```
ip link set dev fm1-mac5 up
```

You are ready now to add your IP gateways and IP routes using commands like:

```
ip neigh add 172.16.0.1 lladdr 00:10:18:BA:E4:04 dev fm1-mac6
ip neigh add 192.168.100.254 lladdr 68:05:ca:12:2f:0f dev fm1-mac5
```

When running in loopback mode you should configure the next hop on the inbound port with its own Ethernet address (marked in bold in the example above).

IP routes can be configured using the `ip route` tool as in:

```
ip route add 192.168.200.0/24 via 192.168.100.254 table 13 dev fm1-mac5
ip route add 172.17.0.0/16 via 172.16.0.1 table 14 dev fm1-mac6
```

The route table numbers can be found in the application startup information by looking after the *Initializing IP route tables* section. The first table in the section is always the outbound route table, while the second is the inbound route table. Please note that there are different initialization sections for IPv4 and IPv6 routing tables.

IP ingress rules can currently be configured for the inbound direction only. For this purpose you have the `ib_rule_add4`, `ib_rule_add6`, `ib_rule_del4` and `ib_rule_del6` commands available. These rules apply to the clear traffic received on the unprotected (inbound) interface. The usual syntax of an `add rule` command is:

```
ib_rule_add4 <src_ip>/<prefix> <dst_ip>/<prefix> <priority> <dest_route_table_no> [ <hex_tos> ]
```

where

- `src_ip/prefix` is the source net IP address;
- `dst_ip/prefix` is the destination net IP address;
- `priority` is the priority level that you want this rule to have relative to the other rules in the ingress IP rule table; please note that there cannot be two rules with the same priority;
- `dest_route_table_no` is the number of the **outbound** route table that this flow will be directed to; identification of the IP route tables was described earlier in this paragraph;

- *hex_tos* is an optional argument specifying the *Type Of Service* or *Traffic Class* in hexadecimal representation.

IP rules apply to inbound clear traffic allowing it to be propagated to the specified outbound IP routing tables. All inbound clear traffic flows that does not match any of the configured ingress IP rules is silently discarded.

The application also supports displaying IPsec statistics. You can get SA related statistics using the *sa_stats* CLI command and also global IPsec statistics using the *ipsec_stats* CLI command.

7.4.2.7 References

1. [P4080/B4860/B4420] Integrated Multicore Communication Processor Family Reference Manual (pdf)
2. USDPAAs PPAC User Guide (Knowledge Center)
3. QMan/BMan API Reference (Knowledge Center)

7.4.2.8 Appendix A – Preparing DPA Offloading DTB Files

Detailed reference for compiling the device tree for the following DPAA offloading scenarios:

1. USDPAAs
2. IP Offloading
3. IP Reassembly
4. Shared interfaces
5. NF Offloading

For these SoC products:

1. P2041
2. P4080
3. B4860
4. B4420
5. T4240
6. T2080
7. LS1043A (only USDPAAs, IP Offloading and IP Reassembly scenarios)

7.4.2.8.1 Compiling the device tree for USDPAAs applications

Compiling the device tree for P4080

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/p4080ds-usdpaa.dts
arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o p4080ds-usdpaa.dtb
arch/powerpc/boot/dts/p4080ds-usdpaa.dts
```

The DTB file *p4080ds-usdpaa.dtb* will be built in the current directory.

Compiling the device tree for P2041

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/p2041rdb-usdpaa.dts
arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o p2041rdb-usdpaa.dtb
arch/powerpc/boot/dts/p2041rdb-usdpaa.dts
```

The DTB file `p2041rdb-usdpaa.dtb` will be built in the current directory.

Compiling the device tree for B4860

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/b4860qds-usdpaa.dts
arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o b4860qds-usdpaa.dtb
arch/powerpc/boot/dts/b4860qds-usdpaa.dts
```

The DTB file `b4860qds-usdpaa.dtb` will be built in the current directory.

Compiling the device tree for B4420

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/b4420qds-usdpaa.dts
arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o b4420qds-usdpaa.dtb
arch/powerpc/boot/dts/b4420qds-usdpaa.dts
```

The DTB file `b4420qds-usdpaa.dtb` will be built in the current directory.

Compiling the device tree for T4240

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/t4240qds-usdpaa.dts
arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o t4240qds-usdpaa.dtb
arch/powerpc/boot/dts/t4240qds-usdpaa.dts
```

The DTB file `t4240qds-usdpaa.dtb` will be built in the current directory.

Compiling the device tree for T2080

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/t2080qds-usdpaa.dts
arch/powerpc/boot/dts
```

```
scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o t2080qds-usdpaa.dtb
arch/powerpc/boot/dts/t2080qds-usdpaa.dts
```

The DTB file `t2080qds-usdpaa.dtb` will be built in the current directory.

Compiling the device tree for LS1043A

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/fsl-ls1043a-rdb-usdpaa.dts arch/arm64/boot/dts/freescale
make freescale/fsl-ls1043a-rdb-usdpaa.dtb
```

The DTB file `fsl-ls1043a-rdb-usdpaa.dtb` will be generated in the `arch/arm64/boot/dts/freescale` subdirectory. LS1043A boot up mechanism may require also that you build an *ITB* file which aggregates your Linux kernel image, *DTB* file and your root file system (in case you are using RAMBOOT). This file can be generated using the *mkimage* tool.

7.4.2.8.2 Compiling the device tree for IP offloading

Compiling the device tree for P4080

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/p4080ds-usdpaa.dts
arch/powerpc/boot/dts/

cp drivers/staging/fsl_dpa_offload/dts/p4080si-pre.dtsi
arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/p4080si-chosen-offld.dtsi
arch/powerpc/boot/dts/fsl/p4080si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o p4080ds-usdpaa-offload.dtb
arch/powerpc/boot/dts/p4080ds-usdpaa.dts
```

The DTB file `p4080ds-usdpaa-offload.dtb` will be built in the current directory.

Compiling the device tree for P2041

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/p2041rdb-usdpaa.dts
arch/powerpc/boot/dts/

cp drivers/staging/fsl_dpa_offload/dts/p2041si-pre.dtsi
arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/p2041si-chosen-offld.dtsi
arch/powerpc/boot/dts/fsl/p2041si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o p2041rdb-usdpaa-offload.dtb
arch/powerpc/boot/dts/p2041rdb-usdpaa.dts
```

The DTB file `p2041rdb-usdpaa-offload.dtb` will be built in the current directory.

Compiling the device tree for B4860

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/b4860qds-usdpaa.dts
  arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/b4860si-pre.dtsi
  arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/b4860si-chosen-offld.dtsi
  arch/powerpc/boot/dts/fsl/b4860si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o
  b4860qds-usdpaa-offload.dtb
  arch/powerpc/boot/dts/b4860qds-usdpaa.dts
```

The DTB file `b4860qds-usdpaa-offload.dtb` will be built in the current directory.

Compiling the device tree for B4420

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/b4420qds-usdpaa.dts
  arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/b4420si-pre.dtsi
  arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/b4420si-chosen-offld.dtsi
  arch/powerpc/boot/dts/fsl/b4420si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o
  b4420qds-usdpaa-offload.dtb
  arch/powerpc/boot/dts/b4420qds-usdpaa.dts
```

The DTB file `b4420qds-usdpaa-offload.dtb` will be built in the current directory.

Compiling the device tree for T4240

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/t4240qds-usdpaa.dts
  arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/t4240si-pre.dtsi
  arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/t4240si-chosen-offld.dtsi
  arch/powerpc/boot/dts/fsl/t4240si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o
  t4240qds-usdpaa-offload.dtb arch/powerpc/boot/dts/t4240qds-usdpaa.dts
```

The DTB file `t4240qds-usdpaa-offload.dtb` will be built in the current directory.

Compiling the device tree for T2080

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/t2080qds-usdpaa.dts
  arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/t208xsi-pre.dtsi
  arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/t2080si-chosen-offld.dtsi
  arch/powerpc/boot/dts/fsl/t2080si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o
  t2080qds-usdpaa-offload.dtb arch/powerpc/boot/dts/t2080qds-usdpaa.dts
```

The DTB file `t2080qds-usdpaa-offload.dtb` will be built in the current directory.

Compiling the device tree for LS1043A

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/fsl-ls1043a-rdb-usdpaa.dts arch/arm64/boot/dts/freescale

cp drivers/staging/fsl_dpa_offload/dts/fsl-ls1043a.dtsi arch/arm64/boot/dts/freescale

cp drivers/staging/fsl_dpa_offload/dts/ls1043a-chosen-offload.dtsi arch/arm64/boot/dts/freescale/
  ls1043a-chosen.dtsi

make freescale/fsl-ls1043a-rdb-usdpaa.dtb
```

The DTB file `fsl-ls1043a-rdb-usdpaa.dtb` will be generated in the `arch/arm64/boot/dts/freescale` subdirectory. LS1043A boot up mechanism may require also that you build an *ITB* file which aggregates your Linux kernel image, *DTB* file and your root file system (in case you are using RAMBOOT). This file can be generated using the *mkimage* tool.

7.4.2.8.3 Compiling the device tree for IP reassembly

Compiling the device tree for P4080

On P4080 there is no specific device tree building process for IP reassembly. You can use the one described in **Compiling the Device Tree for USDPAA Applications**.

Compiling the device tree for P2041

On P2041 there is no specific device tree building process for IP reassembly. You can use the one described in **Compiling the Device Tree for USDPAA Applications**.

Compiling the device tree for B4860

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/b4860qds-usdpaa.dts
  arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/b4860si-pre.dtsi
```

```
arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/b4860si-chosen-reass.dtsi
arch/powerpc/boot/dts/fsl/b4860si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o b4860qds-usdpaa-reass.dtb
arch/powerpc/boot/dts/b4860qds-usdpaa.dts
```

The DTB file `b4860qds-usdpaa-reass.dtb` will be built in the current directory.

Compiling the device tree for B4420

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/b4420qds-usdpaa.dts
arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/b4420si-pre.dtsi
arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/b4420si-chosen-reass.dtsi
arch/powerpc/boot/dts/fsl/b4420si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o b4420qds-usdpaa-reass.dtb
arch/powerpc/boot/dts/b4420qds-usdpaa.dts
```

The DTB file `b4420qds-usdpaa-reass.dtb` will be built in the current directory.

Compiling the device tree for T4240

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/t4240qds-usdpaa.dts
arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/t4240si-pre.dtsi
arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/t4240si-chosen-reass.dtsi
arch/powerpc/boot/dts/fsl/t4240si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o t4240qds-usdpaa-reass.dtb
arch/powerpc/boot/dts/t4240qds-usdpaa.dts
```

The DTB file `t4240qds-usdpaa-reass.dtb` will be built in the current directory.

Compiling the device tree for T2080

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/t2080qds-usdpaa.dts
arch/powerpc/boot/dts

cp drivers/staging/fsl_dpa_offload/dts/t208xsi-pre.dtsi
arch/powerpc/boot/dts/fsl
```

```
cp drivers/staging/fsl_dpa_offload/dts/t2080si-chosen-reass.dtsi
arch/powerpc/boot/dts/fsl/t2080si-chosen.dtsi

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o t2080qds-usdpaa-reass.dtb
arch/powerpc/boot/dts/t2080qds-usdpaa.dts
```

The DTB file `t2080qds-usdpaa-reass.dtb` will be built in the current directory.

Compiling the device tree for LS1043A

Go to your Linux kernel source code directory and run the following commands:

```
cp drivers/staging/fsl_dpa_offload/dts/fsl-ls1043a-rdb-usdpaa.dts arch/arm64/boot/dts/freescale

cp drivers/staging/fsl_dpa_offload/dts/fsl-ls1043a.dtsi arch/arm64/boot/dts/freescale

cp drivers/staging/fsl_dpa_offload/dts/ls1043a-chosen-reass.dtsi arch/arm64/boot/dts/freescale/
ls1043a-chosen.dtsi

make freescale/fsl-ls1043a-rdb-usdpaa.dtb
```

The DTB file `fsl-ls1043a-rdb-usdpaa.dtb` will be generated in the `arch/arm64/boot/dts/freescale` subdirectory. LS1043A boot up mechanism may require also that you build an *ITB* file which aggregates your Linux kernel image, *DTB* file and your root file system (in case you are using RAMBOOT). This file can be generated using the *mkimage* tool.

7.4.2.8.4 Compiling the device tree for ipsec offload

Compiling the device tree for P4080

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/p4080si-pre.dtsi
arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/p4080si-chosen-offld.dtsi
arch/powerpc/boot/dts/fsl/p4080si-chosen.dtsi

cp drivers/staging/fsl_dpa_offload/dts/p4080ds-usdpaa-shared-interfaces.dts
arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o p4080ds-usdpaa-shared-mac.dtb
arch/powerpc/boot/dts/p4080ds-usdpaa-shared-interfaces.dts
```

The DTB file `p4080ds-usdpaa-shared-mac.dtb` will be built in the current directory.

Compiling the device tree for P2041

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/p2041si-pre.dtsi  
arch/powerpc/boot/dts/fsl
```

```
cp drivers/staging/fsl_dpa_offload/dts/p2041si-chosen-offfld.dtsi  
arch/powerpc/boot/dts/fsl/p2041si-chosen.dtsi
```

```
cp drivers/staging/fsl_dpa_offload/dts/p2041rdb-usdpaa-shared-interfaces.dts  
arch/powerpc/boot/dts
```

```
scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o p2041rdb-usdpaa-shared-mac.dtb  
arch/powerpc/boot/dts/p2041rdb-usdpaa-shared-interfaces.dts
```

The DTB file `p2041rdb-usdpaa-shared-mac.dtb` will be built in the current directory.

Compiling the device tree for B4860

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/b4860si-pre.dtsi  
arch/powerpc/boot/dts/fsl
```

```
cp drivers/staging/fsl_dpa_offload/dts/b4860si-chosen-offfld.dtsi  
arch/powerpc/boot/dts/fsl/b4860si-chosen.dtsi
```

```
cp drivers/staging/fsl_dpa_offload/dts/b4860qds-usdpaa-shared-interfaces.dts  
arch/powerpc/boot/dts
```

```
scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o b4860qds-usdpaa-shared-mac.dtb  
arch/powerpc/boot/dts/b4860qds-usdpaa-shared-interfaces.dts
```

The DTB file `b4860qds-usdpaa-shared-mac.dtb` will be built in the current directory.

Compiling the device tree for B4420

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/b4420si-pre.dtsi  
arch/powerpc/boot/dts/fsl
```

```
cp drivers/staging/fsl_dpa_offload/dts/b4420si-chosen-offfld.dtsi  
arch/powerpc/boot/dts/fsl/b4420si-chosen.dtsi
```

```
cp drivers/staging/fsl_dpa_offload/dts/b4420qds-usdpaa-shared-interfaces.dts  
arch/powerpc/boot/dts
```

```
scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o b4420qds-usdpaa-shared-mac.dtb  
arch/powerpc/boot/dts/b4420qds-usdpaa-shared-interfaces.dts
```

The DTB file `b4420qds-usdpaa-shared-mac.dtb` will be built in the current directory.

Compiling the device tree for T4240

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/t4240si-pre.dtsi
arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/t4240si-chosen-offld.dtsi
arch/powerpc/boot/dts/fsl/t4240si-chosen.dtsi

cp drivers/staging/fsl_dpa_offload/dts/t4240qds-usdpaa-shared-interfaces.dts
arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o t4240qds-usdpaa-shared-mac.dtb
arch/powerpc/boot/dts/t4240qds-usdpaa-shared-interfaces.dts
```

The DTB file `t4240qds-usdpaa-shared-mac.dtb` will be built in the current directory.

Compiling the device tree for T2080

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/t208xsi-pre.dtsi
arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/t2080si-chosen-offld.dtsi
arch/powerpc/boot/dts/fsl/t2080si-chosen.dtsi

cp drivers/staging/fsl_dpa_offload/dts/t2080qds-usdpaa-shared-interfaces.dts
arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o t2080qds-usdpaa-shared-mac.dtb
arch/powerpc/boot/dts/t2080qds-usdpaa-shared-interfaces.dts
```

The DTB file `t2080qds-usdpaa-shared-mac.dtb` will be built in the current directory.

7.4.2.8.5 Compiling the device tree for Network Function Layer offloading

Compiling the device tree for B4860

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/b4860si-pre.dtsi
arch/powerpc/boot/dts/fsl

cp drivers/staging/fsl_dpa_offload/dts/b4860si-chosen-offld.dtsi
arch/powerpc/boot/dts/fsl/b4860si-chosen.dtsi

cp drivers/staging/fsl_dpa_offload/dts/b4860qds-usdpaa-nf-offload.dts
arch/powerpc/boot/dts

scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o b4860qds-usdpaa-nf-offload.dtb
arch/powerpc/boot/dts/b4860qds-usdpaa-nf-offload.dts
```

The DTB file `b4860qds-usdpaa-nf-offload.dtb` will be generated in the current directory.

7.4.2.9 Appendix B - Enabling DPA Offloading Drivers in the Linux Kernel

DPA offloading works only with the DPA offloading Linux kernel drivers. In order to build the DPA offloading Linux drivers, you need to follow the steps presented in this paragraph.

Go to your Linux kernel source code directory and type the command

```
make menuconfig
```

In the build menu that is presented, enable the following option

```
Device Drivers
  Staging Drivers
    <*> Freescale Datapath Offloading Driver
```

In this example the driver is built into the Linux kernel. If you would rather use a loadable module, then you can select the "M" option.

```
Device Drivers
  Staging Drivers
    <M> Freescale Datapath Offloading Driver
```

You also need to remember to `insmod` the driver at runtime before running the DPA offloading applications.

Some of the DPA offloading applications require that the FMan driver resource allocation algorithm be disabled. In order to do that, in the build menu make sure the option

```
Device Drivers
  Network device support
    Ethernet driver support
      Freescale devices
        Frame Manager support
          Freescale Frame Manager (datapath) support
            [ ] Enable FMan dynamic resource allocation algorithm
```

is disabled.

The last step is to actually launch the kernel build using the `make` command.

7.4.2.10 Revision history

This table summarizes revisions to this document.

Table 118. Revision history

Revision	Date	Description
0	08/2013	Initial public release.

7.4.3 DPAA Offloading Drivers Reference Manual

7.4.3.1 Introduction

This document covers some components implementation details, but the main focus is the programming interfaces (APIs) exposed by these components.

7.4.3.2 DPA Classifier

The DPA Classifier is a module which allows software to accelerate lookups and decisions based on frame content using DPA.

The software must provide the initial FMan coarse classification nodes to be managed by the Classifier tables and must set up a set of **classification rules**. Rules can be added, edited or removed at any time. The DPA Classifier table will manage the provided FMan resource (coarse classification node) and may extend it if necessary during runtime.

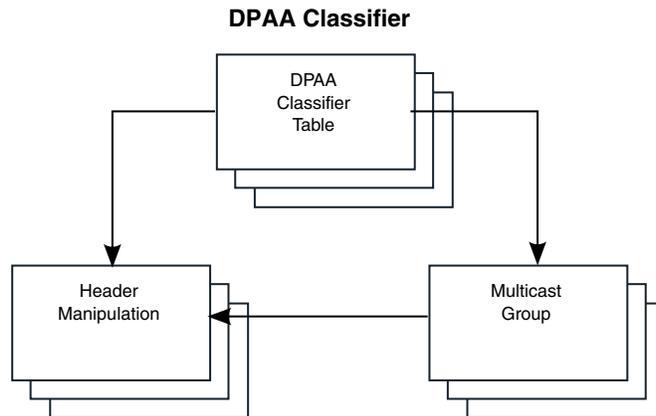


Figure 132. DPA Classifier control objects

7.4.3.2.1 Table

The DPA Classifier's main control object is the DPA Classifier Table. A general functional diagram of the DPA Classifier table is shown below.

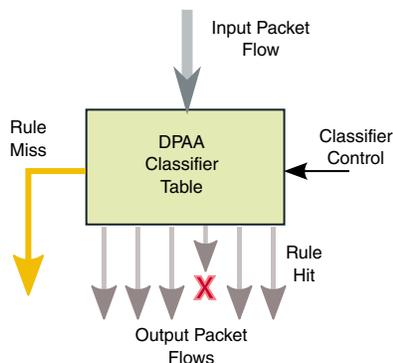


Figure 133. DPA Classifier control objects

Before using the DPA Classifier, the software must create a PCD skeleton. The easiest way to create the PCD skeleton is by using the Fman Configuration tool (FMC).

The user software is responsible for keeping the synchronization between the static FMan resources that it creates at init time and the way it uses DPA Classifier. For instance, it is responsible for properly configuring and linking the FMan KeyGen schemes and the FMan Coarse classification nodes so that the proper keys are extracted from the frames. The DPA Classifier will only assume that the key generation process is correct and fit for that type of table.

A classification rule (also called a classifier entry) is an action that will be executed when the entry is hit. The entries in the table are indexed by lookup key, which can have different formats depending on the table.

The result of a table lookup for an input packet is either:

- a hit condition, which means that the key generated from the input packet data (usually header fields) has matched with the key description associated with a rule in the table or

- a miss condition, which means that the key generated from the input packet data did not match the key description of any entry (rule) in the table.

Depending on the type of the lookup, the miss condition might exist or not. For certain types of lookup (e.g. indexed lookup), the miss condition cannot be defined. The DPA classifier supports the following types of lookups:

Table 119. Port Connections

Lookup Type	Description
Exact match lookup	The hardware will search the table for a key which is an exact sequence of bytes. The key can be used as is, or can be masked. If the key matches exactly over the key of an entry, a hit occurs. If there is no match with any of the keys in the table, a miss situation occurs.
Indexed lookup	The hardware will use a sequence of bits from the key to create an index. The index will be used as a direct index in the table and the action from the entry with that index will be executed. For this type of table there cannot be any miss as the index table is required to be filled with all entries at initialization time. Also, keys from an indexed table cannot be (ever) removed. They can only be modified.
HASH lookup	The hardware will calculate a HASH value out of the key generated from the input packet data, and it will use this masked HASH value as a first index for a table lookup. This will result in finding a HASH set. In the second phase, an exact match lookup is performed among the rules in that particular HASH set to find the exact table entry which matches the lookup key. If a match is found, a lookup hit occurs and the associated action is executed. Otherwise, a lookup miss condition occurs.

Exact match lookup is offering more performance on tables with few entries, while HASH tables offer more performance on densely populated tables. However, the memory consumption for exact match tables is lower and more efficient than for HASH tables, hence it is recommended to use this type of table whenever possible.

The action associated with a table entry falls into 3 possible categories:

1. **Drop frame** action;
2. **Enqueue frame** action - send the frame to a specified frame queue; this action has several options such as enable policing, enable statistics, header manipulation operations;
3. Send frame to **anew lookup** - the frame is directed towards a new DPA Classifier table;
4. **Multicast** - replicate the frame and send it to multiple destinations by using a multicast group.

The DPA Classifier table has two possible ways to manage entries. If entry management **by key** is selected, the DPA Classifier keeps internally a **shadow table** for the entries. This allows it to offer features which are additional to those offered by the low level driver alone, such as lookups and transparent entry index management.

If entry management **by reference** is selected, the DPA Classifier will identify the entries only by their Id (reference), and not by their key. The shadow table is not created in this case. The application will keep the mapping between the keys and their table entry Ids and this will help DPA Classifier to provide more performance for runtime operations.

The DPA Classifier can work with certain *prefilled tables*, which are

- prefilled HASH tables,

- prefilled exact match tables.

Prefilled tables are tables which are populated by the user application before DPA Classifier takes control of the FMan coarse classification node. This is not applicable for indexed tables, where the user always provides all the entries at initialization time. The user application must communicate to the DPA Classifier how many static entries are present in the table at initialization time. If the number of static entries is *n*, the DPA Classifier module will consider that the static entries are the first *n* entries in the table and with the highest priority (when priorities are enabled). Working with prefilled tables comes with certain limitations listed below.

- DPA Classifier can only work in key management mode with prefilled tables. Prefilled tables managed by reference are not supported.
- Prefilled classifier HASH tables do not support adding entries with header manipulations.

7.4.3.2.1.1 Table API

There are four selections for the DPA Classifier table API:

- Create and Remove Tables
- Insert, Remove or Update Tables
- Update Default Table Policy
- Table Look-up

7.4.3.2.1.1.1 Create and Remove Tables

Table 120. API to Create and Remove Tables

Function Name	Description	Input Parameters	Output Parameters
<code>dpa_classif_table_create</code>	Creates a DPA Classifier table using a coarse classification node. This call does not allocate MURAM.	<ul style="list-style-type: none"> • handle of the initial Cc node; • the type of table to create; • the table size; • other table params. 	Descriptor of the newly created DPA Classifier table.
<code>dpa_classif_table_free</code>	Releases all resources associated with a DPA Classifier table and destroys it.	<ul style="list-style-type: none"> • descriptor of the DPA Classifier table to destroy. 	None.

7.4.3.2.1.1.1.1 Function: `dpa_classif_table_create`

Syntax

```
int dpa_classif_table_create(const struct dpa_cls_tbl_params *params,
int *td);

struct dpa_cls_tbl_params
{
void *cc_node;
void *distribution;
void *classification;
enum dpa_cls_tbl_type type;
enum dpa_cls_tbl_entry_mgmt entry_mgmt;
union
```

```
{
    struct dpa_cls_tbl_hash_params  hash_params;
    struct dpa_cls_tbl_indexed_params  indexed_params;
    struct dpa_cls_tbl_exact_match_params exact_match_params;
};
unsigned int    prefilled_entries;
};

enum dpa_cls_tbl_type
{
    DPA_CLS_TBL_HASH = 0,    /**< HASH table */
    DPA_CLS_TBL_INDEXED,    /**< Indexed table */
    DPA_CLS_TBL_EXACT_MATCH /**< Exact match table */
};

struct dpa_cls_tbl_hash_params
{
    unsigned int num_sets;
    unsigned int max_ways;
    unsigned int hash_offs;
    uint8_t key_size;
};

struct dpa_cls_tbl_indexed_params
{
    unsigned int entries_cnt;
};

struct dpa_cls_tbl_exact_match_params
{
    unsigned int entries_cnt;
    uint8_t key_size;
    bool use_priorities;
};

enum dpa_cls_tbl_entry_mgmt
{
    DPA_CLS_TBL_MANAGE_BY_KEY = 0,
    DPA_CLS_TBL_MANAGE_BY_REF
};
```

Parameters

- **params** - data structure containing the parameters of the table to create;
 - **cc_node** - handle of the Cc node to start with; please refer to the function description for more information;
 - **distribution** - handle of a FMan distribution to send frames to instead of enqueueing them. If this handle is provided (not NULL) the enqueue action will only select the frame queue, but it will not actually enqueue the frame to the selected frame queue. Instead it will send the frame to the indicated distribution for further processing.
 - **classification** - handle of a FMan classification to send frames after distribution. For FMan v3 previously determined frame queue will be retained as frame destination. This parameter may be used to perform header manipulations operations such as fragmentation after the enqueue decision has been made.
 - **type** - the type of table to create; configured using `dpa_cls_tbl_type` enum;
 - **entry_mgmt** - table entry management mechanism for runtime; configured using `dpa_cls_tbls_entry_mgmt` enum;

NOTE

Not all types of tables support all types of entry management. Indexed tables can only be managed by key, because table entry insert doesn't make any sense on this type of table. Only entry modification is available for this type of tables. In this case there is no way for the classifier to provide the user an entry reference for the entry management by reference mechanism to work.

• **hash_params**

- num_sets – the number of sets in the HASH table; this controls the table size;
- max_ways – the maximum number of ways (depth) of the HASH table; this controls the capability of the HASH to resolve conflicts;
- hash_offs – lets the user control how the HASH value is used to determine the HASH set; this is the offset in bits from the MSB of the calculated HASH value where DPA Classifier should start applying the mask to get the index for HASH set lookup;
- key_size - the size of the lookup keys in bytes.

• **indexed_params**

- entries_cnt - the number of entries to be stored in the indexed table;

• **exact_match_params**

- entries_cnt - maximum number of entries expected to be stored in the table;
- key_size – the size of the keys in bytes;
- use_priorities - Use priorities for each entry in table if set to true.

- **prefilled_entries** - Number of entries in the table which are pre-filled (already used). The assumption is always that these entries are the first entries in the table and with the highest priority. There are several limitations for prefilled tables relative to the normal (blank) tables: a) prefilled HASH tables cannot be managed by ref; b) header manipulations created using the DPA Classifier header manipulation API cannot be used on prefilled tables.

- **td** - a location where the function will return the descriptor of the newly created table, in case of success; this descriptor will be further used by the application to refer to this DPA Classifier table in other API function calls.

Description

Configures and initializes a DPA Classifier table using a coarse classification node generated with the FMan Configuration tool. This function never allocates new MURAM space.

Once the DPA Classifier takes control of a FMan Cc node, all management must be performed through its API. If applications use different APIs to modify the Cc node's properties in the same time while the DPA Classifier has ownership of the node, unpredictable behavior and data inconsistency can occur. Each type of DPA Classifier table requires a coarse classification node of a special type as its initial Cc node. The types of the initial Cc nodes relative to the types of DPA classifier tables are specified below.

Table 121.

DPA Classifier table type	Required initial FMan Cc node type
Exact Match Table	Match Table Cc Node
Indexed Table	Indexed Cc Node
HASH Table	HASH Table Cc Node

Depending on the value of the **entry_mgmt** parameter, the behavior of the runtime functions changes. In a table created with `DPA_CLS_TBL_MANAGE_BY_KEY`, runtime modify / delete / stats operations can be accomplished using any of the runtime

function versions. For tables which use `DPA_CLS_TBL_MANAGE_BY_REF`, only the runtime function versions with the suffix “by_ref” are supported for modification / deletion / stats.

The `DPA_CLS_TBL_MANAGE_BY_REF` is an entry management mode which is optimized in terms of speed and memory consumption of the DPA Classifier. Management by reference (Id), however, needs the application to take care of the mapping between lookup keys and entry Ids, because the Classifier only uses the entry Ids.

If `DPA_CLS_TBL_MANAGE_BY_KEY` is selected, the DPA Classifier will keep an internal shadow table for the mapping between the lookup keys and the actual table entries. This way, the user can specify entries also by their key and the DPA Classifier will do the (software) mapping to determine the entry Id.

For the HASH and exact match DPA classifier tables, the miss action of the table is stored separately and is not included in the number of entries requested by the user. Indexed tables never have miss action.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if one of the function arguments or parameters of the table was incorrect;
- `-ENOSYS`, if a feature selected by the table parameters is not supported;
- `-ENOMEM`, if there is no more memory to create a new DPA classifier table; this may mean that either the internal management data structures of the table could not be allocated, or the shadow table (if one was requested) could not be allocated;
- `-EBUSY`, if a FMan low level driver function call has failed.

7.4.3.2.1.1.1.2 Function: `dpa_classif_table_free`

Syntax

```
int dpa_classif_table_free(int td);
```

Parameters

- **td** - descriptor of the DPA Classifier table to destroy.

Description

Releases all resources associated with a DPA Classifier table and destroys it.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if the provided table descriptor is not valid;
- `-EBUSY`, if a FMan low level driver function call has failed.

7.4.3.2.1.1.2 Insert, Remove or Update Table Entries

Table 122. API to Insert, Remove or Update Table Entries

Function Name	Description	Input Parameters	Output Parameters
<code>dpa_classif_table_insert_entry</code>	Adds an entry (classification rule) in the specified DPA Classifier table.	<ul style="list-style-type: none"> • descriptor of the DPA Classifier table where the entry needs to be added; • the key description (its format depends on the type of table); • the action description in case of hit. • entry priority (if necessary). 	The new entry's reference (or entry Id) in case of success.
<code>dpa_classif_table_delete_entry_by_key</code>	Removes an entry in the specified DPA Classifier table. The entry is identified by the lookup key.	<ul style="list-style-type: none"> • descriptor of the DPA Classifier table containing the entry to be removed; • the key description of the entry to be removed. 	None.
<code>dpa_classif_table_delete_entry_by_ref</code>	Removes an entry in the specified DPA Classifier table. The entry is identified by its reference (Id).	<ul style="list-style-type: none"> • descriptor of the DPA Classifier table containing the entry to be removed; • the reference of the entry to be removed. 	None.
<code>dpa_classif_table_modify_entry_by_key</code>	Modifies an entry in the specified DPA Classifier table. The entry is identified by the lookup key.	<ul style="list-style-type: none"> • descriptor of the DPA Classifier table containing the entry to be modified; • the key description of the entry to be modified; • the new key description to replace the existing key description; • the new action description in case of hit. 	None.

Table continues on the next page...

Table 122. API to Insert, Remove or Update Table Entries (continued)

Function Name	Description	Input Parameters	Output Parameters
<code>dpa_classif_table_modify_entry_by_ref</code>	<p>Modifies an entry in the specified DPA Classifier table.</p> <p>The entry is identified by reference (Id).</p>	<ul style="list-style-type: none"> • descriptor of the DPA Classifier table containing the entry to be modified; • the reference of the entry to be modified; • the new key description to replace the existing key description; • the new action description in case of hit. 	None.
<code>dpa_classif_table_get_entry_stats_by_key</code>	<p>Returns statistics for a specified entry in a specified table. Optionally reset the statistics once they are read.</p> <p>The entry is identified by the lookup key.</p> <p>NOTE As statistics will be further handled by the DPA Stats component, this function is now obsolete. Please take into account that it will be removed in the next releases of DPA Classifier.</p>	<ul style="list-style-type: none"> • descriptor of the DPA Classifier table containing the entry to return statistics for; • the key description of the classification rule to return statistics for; • indication whether the statistics should be reset once they are read. 	The requested entry statistics.
<code>dpa_classif_table_get_entry_stats_by_ref</code>	<p>Returns statistics for a specified entry in a specified table. Optionally reset statistics once they are read.</p> <p>The entry is identified by its reference (Id).</p> <p>NOTE As statistics will be further handled by the DPA Stats component, this function is now obsolete. Please take into account that it will be removed in the next releases of DPA Classifier.</p>	<ul style="list-style-type: none"> • descriptor of the DPA Classifier table containing the entry to return statistics for; • reference of the classification rule to return statistics for; • indication whether the statistics should be reset once they are read. 	The requested entry statistics.

Table continues on the next page...

Table 122. API to Insert, Remove or Update Table Entries (continued)

Function Name	Description	Input Parameters	Output Parameters
dpa_classif_table_get_miss_stats <div style="text-align: center;"> <p>NOTE</p> <p>As statistics will be further handled by the DPA Stats component, this function is now obsolete. Please take into account that it will be removed in the next releases of DPA Classifier.</p> </div>	Returns miss statistics for a specified table. The table is identified by its descriptor (td).	descriptor of the DPA Classifier table to return statistics for.	The requested table miss statistics.

7.4.3.2.1.1.2.1 Function: dpa_classif_table_insert_entry

Syntax

```

int dpa_classif_table_insert_entry(
    int      td,
    const struct dpa_offload_lookup_key *key,
    const struct dpa_cls_tbl_action *action,
    int      priority,
    int      *entry_id);

struct dpa_cls_tbl_action
{
    enum dpa_cls_tbl_action_type  type;
    bool      enable_statistics;
    union
    {
        struct dpa_cls_tbl_enq_action_desc  enq_params;
        struct dpa_cls_tbl_next_table_desc  next_table_params;
        struct dpa_cls_tbl_mcast_group_desc  mcast_params;
    };
};

enum dpa_cls_tbl_action_type
{
    DPA_CLS_TBL_ACTION_NONE = 0,
    DPA_CLS_TBL_ACTION_DROP,
    DPA_CLS_TBL_ACTION_ENQ,
    DPA_CLS_TBL_ACTION_NEXT_TABLE,
    DPA_CLS_TBL_ACTION_MCAST
};

struct dpa_cls_tbl_enq_action_desc
{
    bool      override_fqid;
    uint32_t  new_fqid;
}

```

```
struct dpa_cls_tbl_policer_params *policer_params;
int hmd;
uint8_t new_rel_vsp_id;
};

struct dpa_cls_tbl_next_table_desc
{
int next_td;
int hmd;
};

struct dpa_cls_tbl_mcast_group_desc
{
int grpd;
int hmd;
};

struct dpa_cls_tbl_policer_params
{
bool modify_policer_params;
bool shared_profile;
unsigned int new_rel_profile_id;
};
```

Parameters

- **td** - descriptor of the DPA Classifier table to insert in;
- **key** - pointer to the key description (its format depends on the type of table);
- **action** - the action description in case of hit;
 - **type** – the type of action to take: drop, enqueue, send to next table, send to multicast group; the drop action descriptor type doesn't need any further parameters; selected from the `dpa_cls_tbl_action_type` enum; action type NONE is not accepted;
 - **enable_statistics** – true to enable statistics for this action; this attribute is ignored if the action is of type send to next table;
 - **enq_params**
 - **override_fqid** – true if the attribute `new_fqid` will override the KeyGen selected frame queue;
 - **new_fqid** – if `override_fqid` is true, this holds the explicit frame queue Id where to place the frame;
 - **policer_params** – policing parameters; if NULL, no policing is performed during the enqueue operation
 - **modify_policer_params** – true if the default policer parameters will be overridden;
 - **shared_profil** - true if this policer profile is shared between ports; relevant only if `modify_policer_params` is also true;
 - **new_rel_profile_id** - this parameter should indicate the policer profile offset within the port's policer profiles or from the SHARED window; relevant only if `modify_policer_params` is true;
 - **hmd** – header manipulation object descriptor, or `DPA_OFFLD_DESC_NONE` if no header manipulation is required for the enqueue action; the header manipulation object defined by the provided descriptor must be a header manipulation chain head;
 - **new_rel_vsp_id** – new virtual storage profile Id. This parameter is mandatory when `override_fqid` is set to true and the port has virtual storage profiles defined. Otherwise it is not used..
 - **mcast_params** – this parameter is used when the action is multicast.

- **grp** – multicast group descriptor. This value identifies an existing multicast group.
- **hmd** – header manipulation object descriptor, or `DPA_OFFLD_DESC_NONE` if no header manipulation is required. The header manipulation object defined by the provided descriptor must be a header manipulation chain head. The header manipulation operation will be performed before sending the frame to the multicast group.
- **next_table_params**
 - **next_td** – descriptor of the next DPA Classifier table where the frame will be sent for further classification.
 - **hmd** - descriptor of the header manipulation chain to use before sending the frames to the next table, or `DPA_OFFLD_DESC_NONE` if no header manipulation is required. The header manipulation object defined by the provided descriptor must be a header manipulation chain head.
- **priority** - the priority of the entry. This parameter is meaningful only if `td` is an exact match table with priority per entries. The priority value of the entry influences the position of the entry in the table relative to the other entries. Entries with lower priority values go to the top of the table. Priority values can be negative. If two entries with the same priority are inserted in the table, they will be positioned one after the other in the table, with the older one first.
- **entry_id** - reference to the newly created entry returned by this function on success.

Description

Adds an entry (classification rule) in the specified DPA Classifier table. If the MURAM for the table was pre-allocated, this operation doesn't consume MURAM.

NOTE

The hardware currently doesn't support longest prefix match on the exact match or HASH tables. If there are more entries in the table that match the lookup (e.g. because of their mask) the first one will always be returned by the hardware lookup.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- `-EEXIST`, if the specified table entry already exists;
- `-EINVAL`, if there is an error in the provided entry parameters;
- `-EBUSY`, if a FMan low level driver function call has failed;
- `-ENOMEM`, if there is no more memory for adding or managing a new table entry;
- `-ENOSPC`, if there is no more room to add a new table entry (table is full).

7.4.3.2.1.1.2.2 Function: `dpa_classif_table_delete_entry_by_key`

Syntax

```
int dpa_classif_table_delete_entry_by_key(
    int    td,
    const struct dpa_offload_lookup_key *key);
```

Parameters

- **td** - descriptor of the DPA Classifier table containing the entry to be removed;
- **key** - the key description of the entry to be removed.

Description

Removes an entry in the specified DPA Classifier table. The entry is identified by the lookup key. If the MURAM for the table was pre-allocated, this function doesn't free up any MURAM space.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- -EINVAL, if there are errors in the parameters provided to the function;;
- -ENOSYS, if the function was illegally called for a table DPA_CLS_TBL_MANAGE_BY_REF;
- -ENODEV, if the entry with the specified key was not found in the table;
- -EBUSY, if a FMan low level driver function call has failed.

7.4.3.2.1.1.2.3 Function: dpa_classif_table_delete_entry_by_ref

Syntax

```
int dpa_classif_table_delete_entry_by_ref(int td, int entry_id);
```

Parameters

- **td** - descriptor of the DPA Classifier table containing the entry to be removed;
- **entry_id** - the reference (Id) of the entry to be removed.

Description

Removes an entry in the specified DPA Classifier table. The entry is identified by its reference (Id). If the MURAM for the table was pre-allocated, this function doesn't free up any MURAM space.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- -EINVAL, if there are errors in the parameters provided to the function;
- -EBUSY, if a FMan low level driver function call has failed.

7.4.3.2.1.1.2.4 Function: dpa_classif_table_modify_entry_by_key

Syntax

```
int dpa_classif_table_modify_entry_by_key(  
    int td,  
    const struct dpa_offload_lookup_key *key,  
    const struct dpa_cls_tbl_entry_mod_params *mod_params);  
  
enum dpa_cls_tbl_modify_type  
{  
    DPA_CLS_TBL_MODIFY_KEY = 0,  
    DPA_CLS_TBL_MODIFY_ACTION,  
    DPA_CLS_TBL_MODIFY_KEY_AND_ACTION  
};
```

```

struct dpa_cls_tbl_entry_mod_params
{
    enum dpa_cls_tbl_modify_type    type;
    struct dpa_offload_lookup_key    *key;
    struct dpa_cls_tbl_action        *action;
};

```

Parameters

- **td** - descriptor of the DPA Classifier table containing the entry to be modified;
- **key** - the key description of the entry to be modified;
- **mod_params** - the new parameters to replace the existing key descriptor and/or action descriptor;
 - **type** - specifies the desired modification; can be either key descriptor modification only, action descriptor modification only, or both; configured using the `dpa_cls_tbl_modify_type` enum;
 - **key** - new key descriptor to replace the descriptor that is indexing this entry; this is ignored if the type of modification is `DPA_CLS_TBL_MODIFY_ACTION`;
 - **action** - the new action description in case of hit; this is ignored if the type of modification is `DPA_CLS_TBL_MODIFY_KEY`.

Description

Modifies an entry in the specified DPA Classifier table. The entry is identified by the lookup key. This function never allocates new MURAM space.

The entry modification types `DPA_CLS_TBL_MODIFY_KEY` and `DPA_CLS_TBL_MODIFY_KEY_AND_ACTION` are only available for exact match tables.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOSYS`, if the required type of modification is not supported or if the function was illegally called for a table `DPA_CLS_TBL_MANAGE_BY_REF`;
- `-EBUSY`, if a FMan low level driver function call has failed.

7.4.3.2.1.1.2.5 Function: `dpa_classif_table_modify_entry_by_ref`

Syntax

```

int dpa_classif_table_modify_entry_by_ref(
    int                td,
    int                entry_id,
    const struct dpa_cls_tbl_entry_mod_params *mod_params);

```

Parameters

- **td** - descriptor of the DPA Classifier table containing the entry to be modified;
- **entry_id** - the reference (Id) of the entry to be modified;
- **mod_params** - the new parameters to replace the existing key descriptor and/or action descriptor;

Description

Modifies an entry in the specified DPA Classifier table. The entry is identified by its reference (Id). This function never allocates new MURAM space. The entry modification types `DPA_CLS_TBL_MODIFY_KEY` and `DPA_CLS_TBL_MODIFY_KEY_AND_ACTION` are only available for exact match tables.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOSYS`, if the required type of modification is not supported;
- `-EBUSY`, if a FMan low level driver function call has failed.

7.4.3.2.1.1.2.6 Function: `dpa_classif_table_flush`

Syntax

```
int dpa_classif_table_flush(int td);
```

Parameters

- `td` - descriptor of the DPA Classifier table to flush;

Description

Removes all the entries in a DPA Classifier Table. After this operation is completed the entries cannot be recovered.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;;
- `-EBUSY`, if a FMan low level driver call has failed.

7.4.3.2.1.1.2.7 Function: `dpa_classif_table_get_entry_stats_by_key`

Syntax

NOTE

As statistics will be further handled by the DPA Stats component, this function is now obsolete. Please take into account that it will be removed in the next releases of DPA Classifier.

```
int dpa_classif_table_get_entry_stats_by_key(  
    int      td,  
    const struct dpa_offload_lookup_key *key,  
    struct dpa_cls_tbl_entry_stats *stats);  
  
struct dpa_cls_tbl_entry_stats  
{  
    uint32_t  pkts;  
    uint32_t  bytes;  
};
```

Parameters

- **td** - of the DPA Classifier table containing the entry to return statistics for;
- **key** - key description of the classification rule to return statistics for;
- **stats** - pointer to a structure allocated especially to receive the requested statistics;
 - **pkts** - the total number of packets that have hit the entry;
 - **bytes** - the total number of bytes in all the packets that have hit the entry.

Description

Returns the statistics for a specified entry in a specified table. The entry is identified by the lookup key.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- **-EINVAL**, if there are errors in the parameters provided to the function;
- **-ENOSYS**, if the function was illegally called for a table `DPA_CLS_TBL_MANAGE_BY_REF`;
- **-ENODEV**, if the entry with the specified key was not found in the table.

7.4.3.2.1.1.2.8 Function: `dpa_classif_table_get_entry_stats_by_ref`

Syntax

NOTE

As statistics will be further handled by the DPA Stats component, this function is now obsolete. Please take into account that it will be removed in the next releases of DPA Classifier.

```
int dpa_classif_table_get_entry_stats_by_ref(
    int          td,
    int          entry_id,
    struct dpa_cls_tbl_entry_stats *stats);
```

Parameters

- **td** - descriptor of the DPA Classifier table containing the entry to return statistics for;
- **entry_id** - reference (Id) of the classification rule to return statistics for;
- **stats** - pointer to a structure allocated especially to receive the requested statistics.

Description

Returns the statistics for a specified entry in a specified table. The entry is identified by its reference (Id).

Return Value

The function returns zero on success or a negative error code otherwise. The returned error code is the following:

- **-EINVAL**, if there are errors in the parameters provided to the function.

7.4.3.2.1.1.2.9 Function: `dpa_classif_table_get_miss_stats`

Syntax

NOTE

As statistics will be further handled by the DPA Stats component, this function is now obsolete. Please take into account that it will be removed in the next releases of DPA Classifier.

```
int dpa_classif_table_get_miss_stats(  
    int          td,  
    struct dpa_cls_tbl_entry_stats *stats);
```

Parameters

- **td** - descriptor of the DPA Classifier table containing the entry to return statistics for;
- **stats** - pointer to a structure allocated especially to receive the requested statistics.

Description

Returns the statistics for a specified table identified by its descriptor (td).

Return Value

The function returns zero on success or a negative error code otherwise. The returned error code is the following:

- **-EINVAL**, if there are errors in the parameters provided to the function.
- **-EPERM**, if there are errors while retrieving the statistics.

7.4.3.2.1.1.3 Update Default Table Policy

Table 123. API to Update Default Table Policy

Function Name	Description	Input Parameters	Output Parameters
<code>dpa_classif_table_modify_miss_action</code>	Modifies the action taken in case of lookup miss condition.	<ul style="list-style-type: none">• descriptor of the DPA Classifier table to modify the miss action for;• pointer to the new action description in case of lookup miss.	None.

7.4.3.2.1.1.3.1 Function: `dpa_classif_table_modify_miss_action`

Syntax

```
int dpa_classif_table_modify_miss_action(  
    int          td,  
    const struct dpa_cls_tbl_action *miss_action);
```

Parameters

- **td** - descriptor of the DPA Classifier table to modify the miss action for;
- **miss_action** - pointer to the new action description in case of lookup miss;

Description

Modifies the action taken in case of lookup miss condition.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOSYS`, if the user requested an unsupported modification of the miss action;
- `-EBUSY`, if a FMan low level driver function call has failed.

7.4.3.2.1.1.4 Table Lookup

Table 124. API for Table Lookup

Function Name	Description	Input Parameters	Output Parameters
<code>dpa_classif_table_lookup_by_key</code>	Performs a software lookup in the specified table. If successful (i.e. the entry exists in that table) the action descriptor for that entry is returned. The entry is identified by the lookup key.	<ul style="list-style-type: none"> • descriptor of the DPA Classifier table to search in; • pointer to the key descriptor of the entry to search for. 	The action descriptor properties in case the entry is found in the table.
<code>dpa_classif_table_lookup_by_ref</code>	Performs a software lookup in the specified table. If successful (i.e. the entry exists in that table) the action descriptor for that entry is returned. The entry is identified by its reference (Id).	<ul style="list-style-type: none"> • descriptor of the DPA Classifier table to search in; • the reference of the entry to search for; 	The action descriptor properties in case the entry is found in the table.

7.4.3.2.1.1.4.1 Function: `dpa_classif_table_lookup_by_key`

Syntax

```
int dpa_classif_table_lookup_by_key(
    int          td,
    const struct dpa_offload_lookup_key *key,
    struct dpa_cls_tbl_action          *action);
```

Parameters

- **td** - descriptor of the DPA Classifier table to search in;
- **key** - pointer to the key descriptor of the entry to search for;
- **action** - pointer to a specially allocated structure to receive the action descriptor properties in case the entry is found in the table; this structure is to be considered valid only when the function returns zero.

Description

Performs a lookup in the specified table for an entry specified by a key. If successful (i.e. the entry exists in that table) the action descriptor for that entry is returned. This function works only if entry management by key was selected for the table.

Please be aware that this is not a hardware accelerated lookup. This lookup is performed by the DPA Classifier in its internal shadow tables. It is recommended to use this function with consideration.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOSYS`, if the function was illegally called for a table `DPA_CLS_TBL_MANAGE_BY_REF`;
- `-ENODEV`, if the entry with the specified key was not found in the table.

7.4.3.2.1.1.4.2 Function: `dpa_classif_table_lookup_by_ref`

Syntax

```
int dpa_classif_table_lookup_by_ref(  
int          td,  
    int          entry_id,  
    struct dpa_cls_tbl_action *action);
```

Parameters

- **td** - descriptor of the DPA Classifier table to search in;
- **key** - pointer to the key descriptor of the entry to search for;
- **action** - pointer to a specially allocated structure to receive the action descriptor properties in case the entry is found in the table; this structure is to be considered valid only when the function returns zero.

Description

Returns the action descriptor for an entry specified by a reference (Id) existing in a specified table. This function is using the internal shadow tables of the DPA Classifier, hence it can be called only for tables created with the `DPA_CLS_TBL_MANAGE_BY_KEY` flag.

Return Value

The function returns zero on success or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOSYS`, if the function was illegally called for a table `DPA_CLS_TBL_MANAGE_BY_REF`;

7.4.3.2.2 Header Manipulation

Header manipulation can be performed on any frame in the following situations:

- when it hits the enqueue action,
- when it is sent to a multicast group or
- when it is sent to a new classification (new lookup).

The available header manipulations are designed based on specific network processes and are listed below.

- NAT (Network Address and Port Translation)
- Forwarding
- Remove header manipulation
- Insert header manipulation
- Update L3/L4 header manipulation
- VLAN specific header manipulation
- MPLS specific header manipulation

Each of these header manipulations can consist of one or more header manipulation **operations**, but this is transparent to the DPA Classifier user. Some of the header manipulation operations may be optional and can be disabled. Header manipulations can be chained in sequence to create more complex actions however, care must be taken at the order of operations. Certain operations may be impacted by previous insert or remove operations which modify the offsets of the headers in the frame.

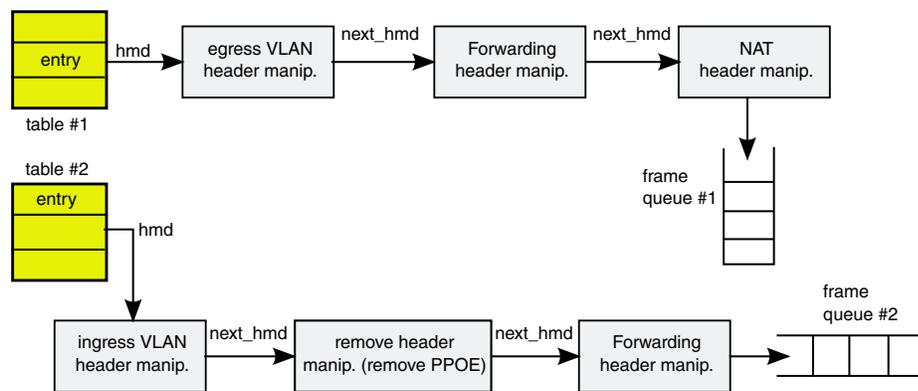


Figure 134. Header manipulation operations chain example

A header manipulation operations chain is always created starting from the tail and working your way towards the head of the chain. The last header manipulation in the chain is identified by passing `next_hmd=DPA_OFFLD_DESC_NONE` (no other header manipulations are beyond it). The chain head is identified by calling the appropriate header manipulation operation create function using the argument `chain_head=true`. When DPA Classifier identifies the chain head, it initializes the low level driver resources for the entire header manipulation chain. The header manipulation operations chain is now ready to be attached to a DPA Classifier table entry.

To attach an existing header manipulation chain to a DPA Classifier table entry, one needs to set the `hmd` attribute of the enqueue parameters (`dpa_cls_tbl_enq_action_desc`) related to the new table entry (`dpa_cls_tbl_action`) to point to the descriptor of the header manipulation chain head. Setting the `hmd` to point to any other header manipulation operation different from a chain head will result in an error. Once the table insert function is called, the table entry and the header manipulation chain enter effect. It is **not possible** to use header manipulations in tables created in different PCDs. It is possible to share the same header manipulation operations **chain** on multiple table entries.

When there is no more need for the header manipulation chain, the header manipulation operations that make it up can be removed in any order. The DPA Classifier will refuse to free header manipulation operations belonging to chains which are in use (i.e. still attached to table entries). One must delete all table entries or detach the header manipulation chain from all entries in order to be able to remove operations from the chain. The low level driver resources are removed only when the chain head operation is removed. It is **not possible** to reuse parts of a header manipulation chain. Header manipulation operation parameters can be modified at runtime using a set of “modify” functions. However, the user application cannot change the **type** of the header manipulation operation. The proper modify function must be called for the type of the existing header manipulation operation. If the header manipulation operation type is not matched to the **modify** function, an error is returned by the **modify** function.

7.4.3.2.2.1 Header Manipulation API

There are two selections for the DPA Classifier table API:

7.4.3.2.2.1.1 Create and Remove Operations

Table 125. API to Create and Remove Header Manipulation Operations

Function Name	Description	Input Parameters	Output Parameters
<code>dpa_classif_set_nat_hm</code>	Creates / imports a NAT type header manipulation object.	<ul style="list-style-type: none"> • NAT header manipulation parameters; • descriptor of the next header manipulation object in chain; • indication whether this header manipulation object is the head of the header manipulation chain; • optional low level resources in case they need to be imported instead of created. 	The descriptor of the current header manipulation object once it is created.
<code>2 dpa_classif_set_fwd_hm</code>	Creates / imports a forwarding type header manipulation object.	<ul style="list-style-type: none"> • forwarding header manipulation parameters; • descriptor of the next header manipulation object in chain. • indication whether this header manipulation object is the head of the header manipulation chain; • optional low level resources in case they need to be imported instead of created. 	The descriptor of the current header manipulation object once it is created.

Table continues on the next page...

Table 125. API to Create and Remove Header Manipulation Operations (continued)

Function Name	Description	Input Parameters	Output Parameters
<code>dpa_classif_set_remove_hm</code>	Creates / imports a remove type header manipulation object.	<ul style="list-style-type: none"> • ingress (remove) header manipulation parameters; • descriptor of the next header manipulation object in chain. • indication whether this header manipulation object is the head of the header manipulation chain; • optional low level resources in case they need to be imported instead of created. 	The descriptor of the current header manipulation object once it is created.
<code>dpa_classif_set_insert_hm</code>	Creates / imports an insert type header manipulation object.	<ul style="list-style-type: none"> • egress insert header manipulation parameters; • descriptor of the next header manipulation object in chain; • indication whether this header manipulation object is the head of the header manipulation chain; • optional low level resources in case they need to be imported instead of created. 	The descriptor of the current header manipulation object once it is created.
<code>dpa_classif_set_update_hm</code>	Creates / imports an update type header manipulation object.	<ul style="list-style-type: none"> • egress update header manipulation parameters; • descriptor of the next header manipulation object in chain. • indication whether this header manipulation object is the head of the header manipulation chain; • optional low level resources in case they need to be imported instead of created. 	The descriptor of the current header manipulation object once it is created.

Table continues on the next page...

Table 125. API to Create and Remove Header Manipulation Operations (continued)

Function Name	Description	Input Parameters	Output Parameters
dpa_classif_set_vlan_hm	Creates / imports a VLAN specific header manipulation (either ingress or egress) object.	<ul style="list-style-type: none"> VLAN specific header manipulation parameters; descriptor of the next header manipulation object in chain. indication whether this header manipulation object is the head of the header manipulation chain; optional low level resources in case they need to be imported instead of created. 	The descriptor of the current header manipulation object once it is created.
dpa_classif_set_mpls_hm	Creates / imports a MPLS specific header manipulation object.	<ul style="list-style-type: none"> MPLS specific header manipulation parameters; descriptor of the next header manipulation object in chain. indication whether this header manipulation object is the head of the header manipulation chain; optional low level resources in case they need to be imported instead of created. 	The descriptor of the current header manipulation object once it is created.
dpa_classif_free_hm	Releases a header manipulation object.	<ul style="list-style-type: none"> descriptor of the header manipulation object to destroy. 	None.

7.4.3.2.2.1.1.1 Function: dpa_classif_set_nat_hm

Syntax

```
int dpa_classif_set_nat_hm(const struct dpa_cls_hm_nat_params *nat_params,
    int next_hmd, int *hmd, bool chain_head,
    const struct dpa_cls_hm_nat_resources *res);

/* Definition of a NAT related header manipulation */
struct dpa_cls_hm_nat_params {
    int flags;
    enum dpa_cls_hm_nat_proto proto;
    enum dpa_cls_hm_nat_type type;
};
```

```
union {
    struct dpa_cls_hm_traditional_nat_params nat;
    struct dpa_cls_hm_nat_pt_params nat_pt;
};
uint16_t sport;
uint16_t dport;
/*
 * More attributes may be added to this structure when ICMP NAT
 * support will be available.
 */
void *fm_pcd;
bool reparse;
};

/* NAT header manipulation low level driver resources */
struct dpa_cls_hm_nat_resources {
    void *l3_update_node;
    void *l4_update_node;
};

/* Supported protocols for NAT header manipulations */
enum dpa_cls_nat_proto {
    DPA_CLS_NAT_PROTO_UDP,
    DPA_CLS_NAT_PROTO_TCP,
    DPA_CLS_NAT_PROTO_ICMP,
    DPA_CLS_NAT_PROTO_LAST_ENTRY
};

/* NAT operation type */
enum dpa_cls_hm_nat_type {
    DPA_CLS_HM_NAT_TYPE_TRADITIONAL,
    DPA_CLS_HM_NAT_TYPE_NAT_PT,
    DPA_CLS_HM_NAT_TYPE_LAST_ENTRY
};

/*
 * Flag values indicating the possible fields to be updated with the
 * NAT header manipulation
 */
enum dpa_cls_hm_nat_flags {
    DPA_CLS_HM_NAT_UPDATE_SIP = 0x01,
    DPA_CLS_HM_NAT_UPDATE_DIP = 0x02,
    DPA_CLS_HM_NAT_UPDATE_SPORT = 0x04,
    DPA_CLS_HM_NAT_UPDATE_DPORT = 0x08,
};

/* Traditional NAT parameters */
struct dpa_cls_hm_traditional_nat_params {
    struct dpa_offload_ip_address sip;
    struct dpa_offload_ip_address dip;
};

/* NAT-PT parameters */
struct dpa_cls_hm_nat_pt_params {
    enum dpa_cls_hm_nat_pt_type type;
    union {
        struct ipv4_header ipv4;
        struct ipv6_header ipv6;
    } header;
};
```

```
/* Type of protocol translation for NAT */  
enum dpa_cls_hm_nat_pt_type {  
    DPA_CLS_HM_NAT_PT_IPv6_TO_IPv4,  
    DPA_CLS_HM_NAT_PT_IPv4_TO_IPv6  
};
```

Parameters

- **nat_params** - NAT header manipulation parameters:
 - **flags** – NAT operation flags specify which fields in the packet should be updated; this is a combination of the values in the `dpa_cls_hm_nat_flags` enum; combine the values using the `or` logical operand;
 - **proto** – protocol to perform NAT for; selected from `dpa_cls_nat_proto` enum ;
 - **type** – selects the flavor of NAT to configure; selected from the `dpa_cls_hm_nat_type` enum;
 - **nat** – traditional NAT header manipulation parameters; used only when traditional NAT is selected using the `type` attribute;
 - **sip** - new source IP address; used only when selected using the `flags` attribute;
 - **dip** - new destination IP address; used only when selected using the `flags` attribute;
 - **nat_pt** - NAT-PT header manipulation parameters; used only when NAT-PT is selected using the `type` attribute;
 - **type** - specifies the protocol replacement for NAT-PT: either IPv4-to-IPv6 or IPv6-to-IPv4;
 - **header** - data for protocol header replacement:
 - **ipv4** - IPv4 header data to replace IPv6 with;
 - **ipv6** - IPv6 header data to replace IPv4 with;
 - **sport** - new L4 protocol source port number; used when selected using the `flags` attribute;
 - **dport** - new L4 protocol destination port number; used only when selected using the `flags` attribute;

NOTE

More attributes may be added to this structure when ICMP NAT support will be available.

- **fm_pcd** - handle to the low level driver PCD to use when creating the header manipulation object. This is necessary only when the header manipulation object is created. If the header manipulation low level driver resources are imported (i.e. `res` argument is provided), `fm_pcd` is irrelevant;
- **reparse** - *TRUE* to force re-parsing of the packet after this NAT;
- **next_hmd** - descriptor of the next header manipulation object in chain; used only when creating a new header manipulation object; `DPA_OFFLD_DESC_NONE` can be provided as next HM if this header manipulation object is not chained to other header manipulation objects;
- **hmd** - the location where the function will return the descriptor of the current header manipulation object once it is created;
- **chain_head** - true if this header manipulation operation is the head of this header manipulations chain. This notifies the DPA classifier that the low level driver resources can be initialized for this header manipulations chain;
- **res** - optional low level driver resources in case the low level header manipulation infrastructure is allocated externally and should be only imported. If this pointer is `NULL`, the low level resources will be automatically allocated and managed by the DPA Classifier;

NOTE

When using external resource allocation it is the responsibility of the user to ensure consistency between the header manipulation parameters provided here and those used when the resources were allocated. There is no way for the DPA Classifier to verify the header manipulation parameters.

- **I3_update_node** – a handle to a header manipulation node which may combine a local IPv4/IPv6 update header manipulation with an IP protocol replace. This is a FMan driver header manipulation node handle and it is mandatory for the import to succeed;
- **I4_update_node** - handle to the local TCP/UDP update header manipulation node. This is a FMan driver header manipulation node handle and it is optional (can be NULL in case no L4 header updates are necessary for this NAT flow);

Description

Creates or imports a NAT type header manipulation object. If the function is successful it returns at the hmd location the descriptor of the created header manipulation object.

If the res parameter is provided, the function will import the low level driver resources specified therein rather than create them. In this case the fm_pcd handle in the parameters structure is not used and can be provided as NULL. When working in this mode the function doesn't allocate MURAM.

If the res parameter is not provided (i.e. is NULL) the function will create the low level resources which may result in MURAM allocation. Low level driver resources for the entire header manipulation chain are allocated only for the chain head (i.e. when chain_head parameter is provided as true). When working in this mode the fm_pcd handle in the parameters structure is necessary for the function to succeed.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- -EINVAL, if there are errors in the parameters provided to the function;
- -ENOMEM, if dynamic memory allocations have failed;
- -EBUSY, if a FMan low level driver function call has failed.

7.4.3.2.2.1.1.2 Function: dpa_classif_set_fwd_hm

Syntax

```
int dpa_classif_set_fwd_hm(
    const struct dpa_cls_hm_fwd_params *fwd_params,
    int next_hmd,
    int *hmd,
    bool chain_head
    const struct dpa_cls_hm_fwd_resources *res);

struct dpa_cls_hm_fwd_params {
    enum dpa_cls_hm_out_if_type out_if_type;
    union {
        struct dpa_cls_hm_fwd_l2_param eth;
        struct dpa_cls_hm_fwd_pppoe_param pppoe;
        struct dpa_cls_hm_fwd_ppp_param ppp;
    };
    struct dpa_cls_hm_ip_frag_params ip_frag_params;
    void *fm_pcd;
    bool reparse;
};

enum dpa_cls_hm_out_if_type {
    DPA_CLS_HM_IF_TYPE_ETHERNET,
    DPA_CLS_HM_IF_TYPE_PPPOE,
    DPA_CLS_HM_IF_TYPE_PPP,
```

```
DPA_CLS_HM_IF_TYPE_LAST_ENTRY
};

struct dpa_cls_hm_ip_frag_params {
    uint16_t    mtu;
    uint8_t     scratch_bpid;
    enum dpa_cls_hm_frag_df_action df_action;
};

enum dpa_cls_hm_frag_df_action {
    DPA_CLS_HM_DF_ACTION_FRAG_ANYWAY,
    DPA_CLS_HM_DF_ACTION_DONT_FRAG,
    DPA_CLS_HM_DF_ACTION_DROP
};

struct dpa_cls_hm_fwd_l2_param {
    uint8_t     macda[ETH_ALEN];
    uint8_t     macsa[ETH_ALEN];
};

struct dpa_cls_hm_fwd_pppoe_param {
    struct dpa_cls_hm_fwd_l2_param l2;
    struct pppoe_header  pppoe_header;
};

struct dpa_cls_hm_fwd_ppp_param {
    uint16_t     ppp_pid;
};

/* IP forwarding header manipulation low level driver resources */
struct dpa_cls_hm_fwd_resources {
    void *fwd_node;
    void *pppoe_node;
    void *ip_frag_node;
};
```

Parameters

- **fwd_params** - IP forwarding header manipulation parameters:
 - **out_if_type** – output interface type; based on this selection the DPA Classifier decides which header manipulations are needed to perform forwarding; selected from `dpa_cls_hm_out_if_type` enum;
 - **eth** – necessary parameters for an Ethernet output interface:
 - **macda** – new Ethernet destination MAC address to update the L2 header;
 - **macsa** – new Ethernet source MAC address to update the L2 header;
 - **pppoe** – necessary parameters for a PPPoE output interface:
 - **l2** – L2 header update parameters;
 - **pppoe_header** – PPPoE header to be inserted in the packets. The PPPoE payload length field is updated automatically (you can set it to zero).
 - **ppp** – necessary parameters for a PPP output interface:
 - **ppp_pid** – PPP PID value to use in the PPP header to be inserted.
 - **ip_frag_params** – parameters related to optional IP fragmentation:

- **mtu** – interface Maximum Transfer Unit in bytes. Use zero if no IP fragmentation should be performed (disable IP fragmentation);
- **scratch_bpuid** – scratch buffer pool ID. This is necessary for the IP fragmentation on FMan v2 devices only. On FMan v3 or newer devices this parameter is ignored. It is also ignored if IP fragmentation is disabled;
- **df_action** - specifies how to deal with packets with DF flag on;

NOTE

Note: IP fragmentation is not supported in this context. Please use the “update” type header manipulation instead if IP fragmentation is needed.

- **fm_pcd** – handle to the low level driver PCD to use when creating the header manipulation object. This is necessary only when the header manipulation object is created. If the header manipulation low level driver resources are imported (i.e. res argument is provided), fm_pcd is irrelevant;
- **reparsed** - *TRUE* to force re-parsing of the packet headers after this forwarding header manipulation;
- **next_hmd** - descriptor of the next header manipulation object in chain; used only when creating a new header manipulation object; DPA_OFFLD_DESC_NONE can be provided as next HM if this header manipulation object is not chained to other header manipulation objects;
- **hmd** - the location where the function will return the descriptor of the current header manipulation object once it is created;
- **chain_head** - true if this header manipulation operation is the head of this header manipulations chain. This notifies the DPA classifier that the low level driver resources can be initialized for this header manipulations chain;
- **res** - optional low level driver resources in case the low level header manipulation infrastructure is allocated externally and should be only imported. If this pointer is NULL, the low level resources will be automatically allocated and managed by the DPA Classifier;

NOTE

When using external resource allocation it is the responsibility of the user to ensure consistency between the header manipulation parameters provided here and those used when the resources were allocated. There is no way for the DPA Classifier to verify the header manipulation parameters.

- **fwd_node** - handle to the forwarding header manipulation node. In case of an Ethernet or PPPoE output interface this is a local header replace header manipulation node (for Ethernet MAC addresses). In case of a PPP output interface this is a header manipulation node which combines a protocol specific header removal (for Ethernet and VLAN tags) with a local header insert manipulation. This is a FMan driver header manipulation node handle and it is mandatory for the import to succeed.

NOTE

PPPoE protocol specific header manipulation is not supported yet.

- **pppoe_node** - Handle to the PPPoE specific node. This is an internal protocol specific insert PPPoE header manipulation node. This is a FMan driver header manipulation node handle and it is optional (can be NULL in case the output interface type is not PPPoE).
- **ip_frag_node** - handle to the IP fragmentation node. This is a FMan driver header manipulation node handle and it is optional (can be NULL in case no IP fragmentation is enabled for this IP forwarding flow).

Description

Creates or imports a forwarding type header manipulation object. This type of header manipulation can be configured to do either of the following:

- L2 header update
- L2 header update + IP fragmentation

L2 header update operation is performed according to the output port type:

- for Ethernet ports = MAC address update
- for PPPoE ports = PPPoE header insert
- for PPP ports = PPP header insert

DPA Classifier takes into account an Ethernet/IP frame to start with and, depending on the selection of output interface type, it decides what header manipulations are necessary.

IP fragmentation cannot be enabled all alone in the forwarding header manipulation context. If only IP fragmentation is needed on a packet flow (without any other header manipulation ops), the “update” type header manipulation is recommended instead of the “forwarding” type header manipulation. For additional details, please refer to `dpa_classif_set_update_hm`.

If the `res` parameter is provided, the function will import the low level driver resources specified therein rather than create them. In this case the `fm_pcd` handle in the parameters structure is not used and can be provided as NULL. When working in this mode the function doesn't allocate MURAM.

If the `res` parameter is not provided (i.e. is NULL) the function will create the low level resources which may result in MURAM allocation. Low level driver resources for the entire header manipulation chain are allocated only for the chain head (i.e. when `chain_head` parameter is provided as true). When working in this mode the `fm_pcd` handle in the parameters structure is necessary for the function to succeed.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- -EINVAL, if there are errors in the parameters provided to the function;
- -ENOMEM, if dynamic memory allocations have failed;
- -EBUSY, if a FMan low level driver function call has failed.

7.4.3.2.2.1.1.3 Function: `dpa_classif_set_remove_hm`

Syntax

```
int dpa_classif_set_remove_hm(
    const struct dpa_cls_hm_remove_params *remove_params,
    int      next_hmd,
    int      *hmd,
    bool     chain_head,
    const struct dpa_cls_hm_remove_resources *res);

struct dpa_cls_hm_remove_params {
    enum dpa_cls_hm_remove_type  type;
    struct dpa_cls_hm_custom_rm_params custom;
    void      *fm_pcd;
    bool      reparse;
};

enum dpa_cls_hm_remove_type {
    DPA_CLS_HM_REMOVE_ETHERNET, /* removes ETH and all QTags */
    DPA_CLS_HM_REMOVE_PPPoE,   /* removes ETH, all QTags and PPPoE */
    DPA_CLS_HM_REMOVE_PPP,
    DPA_CLS_HM_REMOVE_CUSTOM,
    DPA_CLS_HM_REMOVE_LAST_ENTRY
};

struct dpa_cls_hm_custom_rm_params {
    uint8_t  offset;
    uint8_t  size;
};
```

```
};

/* Ingress remove header manipulation low level driver resources */
struct dpa_cls_hm_remove_resources {
    void *remove_node;
};
```

Parameters

- **remove_params** - ingress (remove) header manipulation parameters:
 - **type** - selects the type of the remove header manipulation operation to perform. Protocol specific header removals don't need any further parameters; selected from `dpa_cls_hm_remove_type` enum.

NOTE

Protocol specific PPPoE header removal is not yet supported by the FMan microcode.

- **custom** - parameters for the custom remove header manipulation. If type is anything else than "custom remove", these parameters are ignored;
 - **offset** - offset in bytes, relative to the start of the packet, to start removing data from;
 - **size** - the size in bytes of the section to remove;
 - **fm_pcd** - handle to the low level driver PCD to use when creating the header manipulation object. This is necessary only when the header manipulation object is created. If the header manipulation low level driver resources are imported (i.e. `res` argument is provided), `fm_pcd` is irrelevant;
 - **reparse** - *TRUE* to force re-parsing of packet headers after this header remove;
- **next_hmd** - descriptor of the next header manipulation object in chain; used only when creating a new header manipulation object; `DPA_OFFFLD_DESC_NONE` can be provided as next HM if this header manipulation object is not chained to other header manipulation objects;
- **hmd** - the location where the function will return the descriptor of the current header manipulation object once it is created;
- **chain_head** - true if this header manipulation operation is the head of this header manipulations chain. This notifies the DPA classifier that the low level driver resources can be initialized for this header manipulations chain;
- **res** - optional low level driver resources in case the low level header manipulation infrastructure is allocated externally and should be only imported. If this pointer is `NULL`, the low level resources will be automatically allocated and managed by the DPA Classifier;

NOTE

When using external resource allocation it is the responsibility of the user to ensure consistency between the header manipulation parameters provided here and those used when the resources were allocated. There is no way for the DPA Classifier to verify the header manipulation parameters.

- **remove_node** - handle to either a header removal node or a protocol specific header removal node (for Ethernet and all VLAN tags). This is a FMan driver header manipulation node handle and it is mandatory for the import to succeed.

Description

Creates or imports a remove type header manipulation object.

The `REMOVE_CUSTOM` header manipulation action is a raw data remove operation using an offset (relative to the start of the packet) and a size (in bytes of the area to remove) and it doesn't involve any checksum updates to the frame. The operation is not aware of any protocols that may exist in the frame. The offset and removal size restrictions that apply here are those of the FMan microcode.

If the `res` parameter is provided, the function will import the low level driver resources specified therein rather than create them. In this case the `fm_pcd` handle in the parameters structure is not used and can be provided as `NULL`. When working in this mode the function doesn't allocate MURAM.

If the `res` parameter is not provided (i.e. is `NULL`) the function will create the low level resources which may result in MURAM allocation. Low level driver resources for the entire header manipulation chain are allocated only for the chain head (i.e. when `chain_head` parameter is provided as true). When working in this mode the `fm_pcd` handle in the parameters structure is necessary for the function to succeed.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOMEM`, if dynamic memory allocations have failed;
- `-EBUSY`, if a FMan low level driver function call has failed.

7.4.3.2.2.1.1.4 Function: `dpa_classif_set_insert_hm`

Syntax

```
int dpa_classif_set_insert_hm(
    const struct dpa_cls_hm_insert_params *insert_params,
    int      next_hmd,
    int      *hmd,
    bool     chain_head,
    const struct dpa_cls_hm_insert_resources *res);

struct dpa_cls_hm_insert_params {
    enum dpa_cls_hm_insert_type  type;
    union {
        struct dpa_cls_hm_eth_ins_params eth;
        struct dpa_cls_hm_pppoe_ins_params pppoe;
        uint16_t      ppp_pid;
        struct dpa_cls_hm_custom_ins_params custom;
    };
    void      *fm_pcd;
    bool     reparse;
};

enum dpa_cls_hm_insert_type {
    DPA_CLS_HM_INSERT_ETHERNET, /* Insert Ethernet + QTags */
    DPA_CLS_HM_INSERT_PPPOE, /* Insert PPPoE, ETH and QTags */
    DPA_CLS_HM_INSERT_PPP,
    DPA_CLS_HM_INSERT_CUSTOM, /* General insert */
    DPA_CLS_HM_INSERT_LAST_ENTRY
};

/*
 * Maximum number of VLAN tags supported by the insert header manipulation
 */
#define DPA_CLS_HM_MAX_VLANS      6

/* Ethernet header insert params */
struct dpa_cls_hm_eth_ins_params {
    struct ethhdr  eth_header;
    unsigned int  num_tags;
```

```

struct vlan_header  qtag[DPA_CLS_HM_MAX_VLANs];
};

struct dpa_cls_hm_pppoe_ins_params {
    struct dpa_cls_hm_eth_ins_params eth;
    struct pppoe_header  pppoe_header;
};

struct dpa_cls_hm_custom_ins_params {
    uint8_t  offset;
    uint8_t  size;
    const uint8_t  *data;
};

/* Egress insert header manipulation low level driver resources */
struct dpa_cls_hm_insert_resources {
    void *insert_node;
};

```

Parameters

- **insert_params** - egress insert header manipulation parameters:
 - **type** - specifies the type of insert header manipulation; selected from the `dpa_cls_hm_egress_ins_type` enum;
 - **eth** - Ethernet header insert parameters if **type** is “insert Ethernet”:
 - **eth_header** - Ethernet header to insert;
 - **num_tags** - number of VLAN tags to insert; if zero, no VLAN tags will be inserted in the packet;
 - **qtag** – relevant only if `num_tags` is not zero; contains an array with the data of the VLAN tags to insert;
 - **pppoe** - PPPoE header insert parameters if **type** is “insert PPPoE”;
 - **eth** - parameters of the Ethernet header to insert together with PPPoE header;
 - **pppoe_header** – PPPoE header to insert;
 - **ppp_pid** - PPP PID value to use in the PPP header if **type** is “insert PPP”;
 - **custom** - custom insert header manipulation operation parameters. These are relevant only if a custom insert header manipulation operation is selected.
 - **offset** – the offset in bytes relative to the start of the packet where the new header will be inserted;
 - **size** – the size in bytes of the new header to be inserted;
 - **data** – pointer to the buffer containing the data of the new header to be inserted.
 - **fm_pcd** - handle to the low level driver PCD to use when creating the header manipulation object. This is necessary only when the header manipulation object is created. If the header manipulation low level driver resources are imported (i.e. `res` argument is provided), **fm_pcd** is irrelevant;
 - **reparse** - *TRUE* to force re-parsing of the packet headers after this header insert;
- **next_hmd** - descriptor of the next header manipulation object in chain; used only when creating a new header manipulation object; `DPA_OFFFLD_DESC_NONE` can be provided as next HM if this header manipulation object is not chained to other header manipulation objects;
- **hmd** - the location where the function will return the descriptor of the current header manipulation object once it is created;
- **chain_head** - true if this header manipulation operation is the head of this header manipulations chain. This notifies the DPA classifier that the low level driver resources can be initialized for this header manipulations chain;

- **res** - optional low level driver resources in case the low level header manipulation infrastructure is allocated externally and should be only imported. If this pointer is NULL, the low level resources will be automatically allocated and managed by the DPA Classifier;

NOTE

When using external resource allocation the user shall ensure consistency between the header manipulation parameters provided here and those used when the resources were allocated. The DPA Classifier cannot verify the header manipulation parameters.

- **insert_node** - handle to either an internal header insert or an internal protocol specific header insert node. This is a FMan driver header manipulation node handle and it is mandatory for the import to succeed.

Description

Creates or imports an insert type header manipulation object.

The `INSERT_CUSTOM` header manipulation action is a raw data insert operation using an offset (relative to the start of the packet), a size (in bytes of the area to insert) and a data pointer and it doesn't involve any checksum updates to the frame. The operation is not aware of any protocols that may exist in the frame. The offset and insert size restrictions that apply here are those of the FMan microcode.

If the **res** parameter is provided, the function will import the low level driver resources specified therein rather than create them. In this case the `fm_pcd` handle in the parameters structure is not used and can be provided as NULL. When working in this mode the function doesn't allocate MURAM.

If the **res** parameter is not provided (i.e. is NULL) the function will create the low level resources which may result in MURAM allocation. Low level driver resources for the entire header manipulation chain are allocated only for the chain head (i.e. when `chain_head` parameter is provided as true). When working in this mode the `fm_pcd` handle in the parameters structure is necessary for the function to succeed.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOMEM`, if dynamic memory allocations have failed;
- `-EBUSY`, if an FMan low level driver function call has failed.

7.4.3.2.2.1.1.5 Function: `dpa_classif_set_update_hm`

Syntax

```
int dpa_classif_set_update_hm(
    const struct dpa_cls_hm_update_params *update_params,
    int next_hmd,
    int *hmd,
    bool chain_head,
    const struct dpa_cls_hm_update_resources *res);

struct dpa_cls_hm_update_params {
    int op_flags;
    union {
        struct ipv4_header new_ipv4_hdr;
        struct ipv6_header new_ipv6_hdr;
    } replace;
    union {
```

```

    struct dpa_cls_hm_l3_update_params l3;
    struct dpa_cls_hm_l4_update_params l4;
} update;

struct dpa_cls_hm_ip_frag_params ip_frag_params;
void      *fm_pcd;
bool      reparse;
};

/* Op flags */
enum dpa_cls_hm_update_op_flags {
    DPA_CLS_HM_UPDATE_NONE      = 0,

    DPA_CLS_HM_UPDATE_IPv4_UPDATE  = 0x01,
    DPA_CLS_HM_UPDATE_IPv6_UPDATE  = 0x02,
    DPA_CLS_HM_UPDATE_UDP_TCP_UPDATE = 0x04,

    DPA_CLS_HM_REPLACE_IPv4_BY_IPv6 = 0x08,
    DPA_CLS_HM_REPLACE_IPv6_BY_IPv4 = 0x10
};

/* Field flags */
enum dpa_cls_hm_l3_field_flags {
    DPA_CLS_HM_IP_UPDATE_IPSA      = 0x01,
    DPA_CLS_HM_IP_UPDATE_IPDA      = 0x02,
    DPA_CLS_HM_IP_UPDATE_TOS_TC     = 0x04,
    DPA_CLS_HM_IP_UPDATE_ID         = 0x08,
    DPA_CLS_HM_IP_UPDATE_TTL_HOPL_DECREMENT = 0x10
};

enum dpa_cls_hm_l4_field_flags {
    DPA_CLS_HM_L4_UPDATE_SPORT      = 0x01,
    DPA_CLS_HM_L4_UPDATE_DPORT      = 0x02,
    DPA_CLS_HM_L4_UPDATE_CALCULATE_CKSUM = 0x04
};

/* L3 protocols field update parameters */
struct dpa_cls_hm_l3_update_params {
    struct dpa_offload_ip_address ipsa;
    struct dpa_offload_ip_address ipda;
    uint8_t      tos_tc;
    uint16_t     initial_id;

    /* select a combination from enum dpa_cls_hm_l3_field_flags */
    int          field_flags;
};

/* L4 protocols field update parameters */
struct dpa_cls_hm_l4_update_params {
    uint16_t sport; /* new L4 source port value */
    uint16_t dport; /* new L4 destination port value */

    /* select a combination from enum dpa_cls_hm_l4_field_flags */
    int          field_flags;
};

/* Update header manipulation low level driver resources */
struct dpa_cls_hm_update_resources {
    void *update_node;
};

```

```
void *ip_frag_node;  
};
```

Parameters

- **update_params** - egress update header manipulation parameters:
 - **op_flags** - flags defining the header manipulation operations to perform. They are a combination of the flags defined in the `dpa_cls_hm_update_op_flags` enum; only one flag from each group (e.g, update group, replace group, etc) can be selected; combine the values using the or logical operand;enum;
 - **replace** - parameters for header replace operations:
 - **new_ipv4_hdr** - IPv4 header data. This header can be used either for IPv4 field updates or for IPv6 to IPv4 header replace.
 - **new_ipv6_hdr** - IPv6 header data. This header can be used either for IPv6 field updates or for IPv4 to IPv6 header replace
 - **update** - parameters for protocol specific header update operations:
 - **I3** - L3 protocol field values. This data is used for L3 protocol header updates:
 - **ipsa** - new source IP address;
 - **ipda** - new destination IP address;
 - **tos_tc** - new TOS (for IPv4) or Traffic Class (for IPv6);
 - **initial_id** - initial IPv4 ID; this is used only if `op_flags` selected IPv4 update;
 - **field_flags** - a combination of flags designating the header fields to replace; the available options are defined in the `dpa_cls_hm_l3_field_flags` enum; combine the values using the or logical operand;
 - **I4** - L4 protocol field values. This data is used for L4 protocol header updates:
 - **sport** - new L4 source port value;
 - **dport** - new L4 destination port value;
 - **field_flags** - a combination of flags designating the header fields to replace; the available options are defined in the `dpa_cls_hm_l4_field_flags` enum; combine the values using the or logical operand;
 - **ip_frag_params** - IP fragmentation parameters. This is an optional operation and can be disabled if `op_flags` is not `DPA_CLS_HM_UPDATE_NONE`.

NOTE

There is currently a limitation when using IP fragmentation. IP fragmentation cannot be combined with other header manipulations, hence it can be used only as single update by setting the `op_flags` to `DPA_CLS_HM_UPDATE_NONE`.

- **fm_pcd** - handle to the low level driver PCD to use when creating the header manipulation object. This is necessary only when the header manipulation object is created. If the header manipulation low level driver resources are imported (i.e. `res` argument is provided), `fm_pcd` is irrelevant;
- **reparse** - `TRUE` to force re-parsing of packet headers after this header update;
- **next_hmd** - descriptor of the next header manipulation object in chain; used only when creating a new header manipulation object; `DPA_OFFLD_DESC_NONE` can be provided as next HM if this header manipulation object is not chained to other header manipulation objects;
- **hmd** - the location where the function will return the descriptor of the current header manipulation object once it is created;
- **chain_head** - true if this header manipulation operation is the head of this header manipulations chain. This notifies the DPA classifier that the low level driver resources can be initialized for this header manipulations chain;

- **res** - optional low level driver resources in case the low level header manipulation infrastructure is allocated externally and should be only imported. If this pointer is NULL, the low level resources will be automatically allocated and managed by the DPA Classifier;

NOTE

When using external resource allocation it is the responsibility of the user to ensure consistency between the header manipulation parameters provided here and those used when the resources were allocated. There is no way for the DPA Classifier to verify the header manipulation parameters.

- **update_node** - handle to a header manipulation node with different header manipulations enabled, depending on the options selected in the parameters: local IPv4/IPv6 update header manipulation, a local TCP/UDP update header manipulation or an internal IP header replace. This is a FMan driver header manipulation node handle and it is optional (can be NULL in case no L3 or L4 field updates or header replace features are enabled for this flow);
- **ip_frag_node** - handle to the IP fragmentation node. This is a FMan driver header manipulation node handle and it is optional (can be NULL in case no IP fragmentation is enabled for this flow).

Description

Creates or imports an update type header manipulation object.

The updates performed by this type of header manipulation are protocol specific updates. The FMan is aware of the protocols being manipulated, hence the following checksums are also updated:

Table 126. Checksum updates

op_flags	Updated checksums
IPv4_UPDATE	IPv4 header checksum
UDP_TCP_UPDATE	TCP/UDP checksum only if it is not zero
IPv6_UPDATE	no checksums updated
REPLACE_IPv4_BY_IPv6	no checksums updated
REPLACE_IPv6_BY_IPv4	IPv4 header checksum

TCP/UDP checksum calculation can also be forced using the op_flags set to **UDP_TCP_UPDATE** and the L4 field_flags enabling **CALCULATE_CKSUM** option.

If the res parameter is provided, the function will import the low level driver resources specified therein rather than create them. In this case the fm_pcd handle in the parameters structure is not used and can be provided as NULL. When working in this mode the function doesn't allocate MURAM

If the res parameter is not provided (i.e. is NULL) the function will create the low level resources which may result in MURAM allocation. Low level driver resources for the entire header manipulation chain are allocated only for the chain head (i.e. when chain_head parameter is provided as true). When working in this mode the fm_pcd handle in the parameters structure is necessary for the function to succeed.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- -EINVAL, if there are errors in the parameters provided to the function;
- -ENOMEM, if dynamic memory allocations have failed;
- -EBUSY, if an FMan low level driver function call has failed.

7.4.3.2.2.1.1.6 Function: dpa_classif_set_vlan_hm

Syntax

```
int dpa_classif_set_vlan_hm(
const struct dpa_cls_hm_vlan_params  vlan_params,
int      next_hmd,
int      *hmd,
bool     chain_head,
const struct dpa_cls_hm_vlan_resources *res);

enum dpa_cls_hm_vlan_type {
    DPA_CLS_HM_VLAN_INGRESS,
    DPA_CLS_HM_VLAN_EGRESS,
    DPA_CLS_HM_VLAN_LAST_ENTRY
};

struct dpa_cls_hm_vlan_params {
    enum dpa_cls_hm_vlan_type  type;
    union {
        struct dpa_cls_hm_ingress_vlan_params ingress;
        struct dpa_cls_hm_egress_vlan_params egress;
    };
    void      *fm_pcd;
    bool     reparse;
};

/*
 * Maximum number of VLAN tags supported by the insert header manipulation
 */
#define DPA_CLS_HM_MAX_VLANS      6
/* Standard size of the DSCP-to-VPri mapping table */
#define DPA_CLS_HM_DSCP_TO_VLAN_TABLE_SIZE  32

struct dpa_cls_hm_egress_vlan_params {
    enum dpa_cls_hm_vlan_update_type update_op;
    unsigned int  num_tags;
    struct vlan_header  qtag[DPA_CLS_HM_MAX_VLANS];
    union {
        uint8_t vpri;
        uint8_t dscp_to_vpri[DPA_CLS_HM_DSCP_TO_VPRI_TABLE_SIZE];
    } update;
};

enum dpa_cls_hm_vlan_update_type {
    DPA_CLS_HM_VLAN_UPDATE_NONE,
    DPA_CLS_HM_VLAN_UPDATE_VPri, /* manual VPri update */
    DPA_CLS_HM_VLAN_UPDATE_VPri_BY_DSCP,
    DPA_CLS_HM_VLAN_UPDATE_LAST_ENTRY
};

enum dpa_cls_hm_vlan_count {
    DPA_CLS_HM_VLAN_CNT_NONE,
    DPA_CLS_HM_VLAN_CNT_1QTAG,      /* outer QTag */
    DPA_CLS_HM_VLAN_CNT_2QTAGS,    /* outer most 2 QTags */
    DPA_CLS_HM_VLAN_CNT_3QTAGS,    /* outer most 3 QTags */
    DPA_CLS_HM_VLAN_CNT_4QTAGS,    /* outer most 4 QTags */
    DPA_CLS_HM_VLAN_CNT_5QTAGS,    /* outer most 5 QTags */
};
```

```

DPA_CLS_HM_VLAN_CNT_6QTAGS,      /* outer most 6 QTags */
DPA_CLS_HM_VLAN_CNT_ALL_QTAGS,
DPA_CLS_HM_VLAN_CNT_LAST_ENTRY
};

struct dpa_cls_hm_ingress_vlan_params {
    enum dpa_cls_hm_vlan_count  num_tags;
};

/* VLAN specific header manipulation low level resources */
struct dpa_cls_hm_vlan_resources {
    void *vlan_node;
};

```

Parameters

- **vlan_params** - VLAN specific header manipulation parameters:

- **type** - selects the type of the VLAN specific header manipulation: either “ingress” or “egress”; selected from `dpa_cls_hm_vlan_type` enum;
- **ingress** - parameters for ingress VLAN header manipulations:
 - **num_tags** – number of VLAN tags to remove;

NOTE

Only the `num_tags=none` and `num_tags=all` are currently supported.

- **egress** - parameters for egress VLAN header manipulations:
 - **update_op** – the type of desired VLAN tag update operation (if any); update operations are performed only on the outer most VLAN tag; selected from the `dpa_cls_hm_vlan_update_type` enum;
 - **num_tags** – number of VLAN tags to insert; if zero, no VLAN tags will be inserted in the packet and only update operations will be performed on existent VLAN tags; the combination `num_tags=0` and `update_op=none` (nothing to do) doesn't make sense and it is rejected;
 - **qtag** – relevant only if `num_tags` is not zero; contains an array with the data of the VLAN tags to insert;
 - **update_params** – VLAN tag update parameters if an `update_op` was selected;
 - **vpri** – new VPri field value if `update_op` selects manual VPri update;
 - **dscp_to_vpri** – DSCP-to-VPri mapping table to use for VPri field update if `update_op` selects VPri update by mapping to DSCP;
 - **fm_pcd** - handle to the low level driver PCD to use when creating the header manipulation object. This is necessary only when the header manipulation object is created. If the header manipulation low level driver resources are imported (i.e. `res` argument is provided), `fm_pcd` is irrelevant;
 - **reparse** - *TRUE* to force re-parsing of packet headers after this VLAN header update;
- **next_hmd** - descriptor of the next header manipulation object in chain; used only when creating a new header manipulation object; `DPA_OFFLD_DESC_NONE` can be provided as next HM if this header manipulation object is not chained to other header manipulation objects;
- **hmd** - the location where the function will return the descriptor of the current header manipulation object once it is created;
- **chain_head** - true if this header manipulation operation is the head of this header manipulations chain. This notifies the DPA classifier that the low level driver resources can be initialized for this header manipulations chain;
- **res** - optional low level driver resources in case the low level header manipulation infrastructure is allocated externally and should be only imported. If this pointer is `NULL`, the low level resources will be automatically allocated and managed by the DPA Classifier;

NOTE

When using external resource allocation it is the responsibility of the user to ensure consistency between the header manipulation parameters provided here and those used when the resources were allocated. There is no way for the DPA Classifier to verify the header manipulation parameters.

- **vlan_node** - handle to a header manipulation node with different operations depending on the selected type of VLAN specific header manipulation. In case of VLAN ingress header manipulation this is a VLAN protocol specific removal node. In case of VLAN egress header manipulation this is a header manipulation node which may combine an internal header insert (in case there are VLANs to insert) with a protocol specific VLAN update operation. This is a FMan driver header manipulation node handle and it is mandatory for the import to succeed.

Description

Creates or imports a VLAN specific header manipulation (either ingress or egress) object. VLAN header manipulations apply to both IPv4 as well as IPv6 frames. VLAN tags are added to the Ethernet frame header hence it will work with any type of L3 protocol (even different from IP).

Removal of VLAN tags is performed based on VLAN tag availability in the incoming packet. If the user configured to remove a specific number of VLAN tags and the incoming packets don't have that many VLAN tags, the FMan will remove as many tags as it finds up to the configured number of VLAN tags to remove. If the incoming packet is untagged, no action will be taken and the packet will remain unchanged. This is considered normal operation and no error or other notification will be issued in this case.

If the `res` parameter is provided, the function will import the low level driver resources specified therein rather than create them. In this case the `fm_pcd` handle in the parameters structure is not used and can be provided as NULL. When working in this mode the function doesn't allocate MURAM.

If the `res` parameter is not provided (i.e. is NULL) the function will create the low level resources which may result in MURAM allocation. Low level driver resources for the entire header manipulation chain are allocated only for the chain head (i.e. when **chain_head** parameter is provided as true). When working in this mode the `fm_pcd` handle in the parameters structure is necessary for the function to succeed.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- -EINVAL, if there are errors in the parameters provided to the function;
- -ENOMEM, if dynamic memory allocations have failed;
- -EBUSY, if an FMan low level driver function call has failed.

7.4.3.2.2.1.1.7 Function: `dpa_classif_set_mpls_hm`

Syntax

```
int dpa_classif_set_mpls_hm(const struct dpa_cls_hm_mpls_params
*mpls_params, int next_hmd, int *hmd, bool chain_head,
const struct dpa_cls_hm_mpls_resources *res);

struct dpa_cls_hm_mpls_params {
    enum dpa_cls_hm_mpls_type type;
    struct mpls_header mpls_hdr[DPA_CLS_HM_MAX_MPLS_LABELS];
    unsigned int num_labels;
    void *fm_pcd;
    bool reparse;
};
```

```

/*
 * Maximum number of MPLS labels supported by the insert header
 * manipulation
 */
#define DPA_CLS_HM_MAX_MPLS_LABELS    6

enum dpa_cls_hm_mpls_type {
    DPA_CLS_HM_MPLS_INSERT_LABELS,
    DPA_CLS_HM_MPLS_REMOVE_ALL_LABELS,
    DPA_CLS_HM_MPLS_LAST_ENTRY
};

/* MPLS specific header manipulation low level driver resources */
struct dpa_cls_hm_mpls_resources {
    void *ins_rm_node;
};

```

Parameters

- **mpls_params** - MPLS specific header manipulation parameters;
 - **type** - specifies the type of header manipulation; selected from `dpa_cls_hm_mpls_type` enum; the “remove all MPLS labels” operation type doesn’t need any other parameters;
 - **mpls_hdr** - the MPLS labels to insert if the operation type is “insert MPLS labels”;
 - **num_labels** - number of MPLS labels to insert; this is relevant only if the operation type is “insert MPLS labels”;
 - **fm_pcd** - handle to the low level driver PCD to use when creating the header manipulation object. This is necessary only when the header manipulation object is created. If the header manipulation low level driver resources are imported (i.e. `res` argument is provided), `fm_pcd` is irrelevant;
 - **reparse** - *TRUE* to force reparsing of packet headers after this MPLS header update;
- **next_hmd** - descriptor of the next header manipulation object in chain; used only when creating a new header manipulation object; `DPA_OFFLD_DESC_NONE` can be provided as next HM if this header manipulation object is not chained to other header manipulation objects;
- **hmd** - the location where the function will return the descriptor of the current header manipulation object once it is created;
- **chain_head** - true if this header manipulation operation is the head of this header manipulations chain. This notifies the DPA classifier that the low level driver resources can be initialized for this header manipulations chain;
- **res** - optional low level driver resources in case the low level header manipulation infrastructure is allocated externally and should be only imported. If this pointer is `NULL`, the low level resources will be automatically allocated and managed by the DPA Classifier;
 - **ins_rm_node** - handle to the protocol specific header insert (MPLS) or to the protocol specific header removal (MPLS) node. This is a FMan driver header manipulation node handle and it is mandatory for the import to succeed.

Description

Creates or imports a MPLS specific header manipulation object.

If the `res` parameter is provided, the function will import the low level driver resources specified therein rather than create them. In this case the **fm_pcd** handle in the parameters structure is not used and can be provided as `NULL`. When working in this mode the function doesn’t allocate MURAM.

If the `res` parameter is not provided (i.e. is `NULL`) the function will create the low level resources which may result in MURAM allocation. Low level driver resources for the entire header manipulation chain are allocated only for the chain head (i.e. when `chain_head` parameter is provided as true). When working in this mode the **fm_pcd** handle in the parameters structure is necessary for the function to succeed.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- -EINVAL, if there are errors in the parameters provided to the function;
- -ENOMEM, if dynamic memory allocations have failed;
- -EBUSY, if an FMan low level driver function call has failed.

7.4.3.2.2.1.1.8 Function: dpa_classif_free_hm

Syntax

```
int dpa_classif_free_hm(int hmd);
```

Parameters

- **hmd** - descriptor of the header manipulation object to destroy.

Description

Releases a header manipulation object and frees up all related resources allocated for it. The header manipulation operations can be removed in any order. The low level driver resources are removed only when the chain head operation is removed.

The function fails if the header manipulation object is in use (i.e. if it is part of a header manipulation chain which is still attached to an existing DPA Classifier entry action descriptor).

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- -EINVAL, if there are errors in the parameters provided to the function;
- -EBUSY, if the header manipulation operation is still in use, or if an FMan low level driver function call has failed.

7.4.3.2.2.1.2 *Modify Operations at Runtime*

Table 127. API to Modify Header Manipulation Operations at Runtime

Function Name	Description	Input Parameters	Output Parameters
dpa_classif_modify_nat_hm	Modify the parameters of an existing NAT header manipulation.	<ul style="list-style-type: none"> • new NAT header manipulation parameters; • flags indicating which parameters of the NAT header manipulation must be modified. 	None.

Table continues on the next page...

Table 127. API to Modify Header Manipulation Operations at Runtime (continued)

Function Name	Description	Input Parameters	Output Parameters
<code>dpa_classif_modify_fwd_hm</code>	Modify the parameters of an existing forwarding type header manipulation.	<ul style="list-style-type: none"> new forwarding header manipulation parameters; flags indicating which parameters of the forwarding header manipulation must be modified. 	None.
<code>dpa_classif_modify_remove_hm</code>	Modify the parameters of an existing remove type header manipulation.	<ul style="list-style-type: none"> new remove header manipulation parameters; flags indicating which parameters of the remove header manipulation must be modified. 	None.
<code>dpa_classif_modify_insert_hm</code>	Modify the parameters of an existing insert header manipulation.	<ul style="list-style-type: none"> new insert header manipulation parameters; flags indicating which parameters of the insert header manipulation must be modified. 	None.
<code>dpa_classif_modify_update_hm</code>	Modify the parameters of an existing update header manipulation.	<ul style="list-style-type: none"> new update header manipulation parameters; flags indicating which parameters of the update header manipulation must be modified. 	None.
<code>dpa_classif_modify_vlan_hm</code>	Modify the parameters of an existing VLAN specific header manipulation.	<ul style="list-style-type: none"> new VLAN specific header manipulation parameters; flags indicating which parameters of the VLAN specific header manipulation must be modified. 	None.
<code>dpa_classif_modify_mpls_hm</code>	Modify the parameters of an existing MPLS specific header manipulation.	<ul style="list-style-type: none"> new MPLS specific header manipulation parameters; flags indicating which parameters of the MPLS specific header manipulation must be modified. 	None.

7.4.3.2.2.1.2.1 Function: `dpa_classif_modify_nat_hm`

Syntax

```
int dpa_classif_modify_nat_hm(int    hmd,  
const struct dpa_cls_hm_nat_params *new_nat_params,  
int    modify_flags);  
  
enum dpa_cls_hm_nat_modify_flags {  
    DPA_CLS_HM_NAT_MOD_FLAGS    = 0x01,  
    DPA_CLS_HM_NAT_MOD_SIP      = 0x02,  
    DPA_CLS_HM_NAT_MOD_DIP      = 0x04,  
    DPA_CLS_HM_NAT_MOD_SPORT    = 0x08,  
    DPA_CLS_HM_NAT_MOD_DPORT    = 0x10,  
    DPA_CLS_HM_NAT_MOD_IP_HDR   = 0x20  
};
```

Parameters

- **hmd** - descriptor of the NAT header manipulation object to modify parameters for; if the header manipulation designated by this descriptor is not a NAT header manipulation, the function will fail;
- **new_nat_params** - data structure containing the header manipulation attributes to modify;
- **modify_flags** - combination of flags indicating which header manipulation attributes to modify (and hence indicating which of the attributes in the `new_nat_params` data structure are valid). Select the flag values from the `dpa_cls_hm_nat_modify_flags` enum and combine them using the "or" logical operand. There are some flag combinations which are illegal:
 - `DPA_CLS_HM_NAT_MOD_IP_HDR` cannot be used in combination with either of:
 1. `DPA_CLS_HM_NAT_MOD_SIP` OR
 2. `DPA_CLS_HM_NAT_MOD_DIP`.

Description

Modify the parameters of an existing NAT header manipulation.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-EBUSY`, if an FMan low level driver function call has failed.

7.4.3.2.2.1.2.2 Function: `dpa_classif_modify_fwd_hm`

Syntax

```
int dpa_classif_modify_fwd_hm(int    hmd,  
const struct dpa_cls_hm_fwd_params *new_fwd_params,  
int    modify_flags);  
  
enum dpa_cls_hm_fwd_modify_flags {  
    DPA_CLS_HM_FWD_MOD_ETH_MACSA    = 0x01,  
    DPA_CLS_HM_FWD_MOD_ETH_MACDA    = 0x02,  
    DPA_CLS_HM_FWD_MOD_PPPOE_HEADER = 0x04,  
    DPA_CLS_HM_FWD_MOD_PPP_PID      = 0x08,  
};
```

```
DPA_CLS_HM_FWD_MOD_IP_FRAG_MTU = 0x10,
DPA_CLS_HM_FWD_MOD_IP_FRAG_SCRATCH_BPID = 0x20,
DPA_CLS_HM_FWD_MOD_IP_FRAG_DF_ACTION = 0x40
};
```

Parameters

- **hmd** - descriptor of the forwarding header manipulation object to modify parameters for; if the header manipulation designated by this descriptor is not a forwarding header manipulation, the function will fail;
- **new_fwd_params** - data structure containing the header manipulation attributes to modify;
- **modify_flags** - combination of flags indicating which header manipulation attributes to modify (and hence indicating which of the attributes in the `new_fwd_params` data structure are valid). Select the flag values from the `dpa_cls_hm_fwd_modify_flags` enum and combine them using the "or" logical operand. There are some flag combinations which are illegal:
 - `DPA_CLS_HM_FWD_MOD_PPP_PID` cannot be used in combination with either of:
 - `DPA_CLS_HM_FWD_MOD_ETH_MACSA`,
 - `DPA_CLS_HM_FWD_MOD_ETH_MACDA` OR
 - `DPA_CLS_HM_FWD_MOD_PPPOE_HEADER`.

Description

Modify the parameters of an existing forwarding type header manipulation.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOMEM`, if dynamic memory allocations have failed;
- `-EBUSY`, if an FMan low level driver function call has failed.

7.4.3.2.2.1.2.3 Function: `dpa_classif_modify_remove_hm`

Syntax

```
int dpa_classif_modify_remove_hm(int          hmd,
const struct dpa_cls_hm_remove_params *new_remove_params,
int          modify_flags);

enum dpa_cls_hm_remove_modify_flags {
    DPA_CLS_HM_RM_MOD_TYPE           = 0x01,
    DPA_CLS_HM_RM_MOD_CUSTOM_OFFSET = 0x02,
    DPA_CLS_HM_RM_MOD_CUSTOM_SIZE   = 0x04
};
```

Parameters

- **hmd** - descriptor of the remove header manipulation object to modify parameters for; if the header manipulation designated by this descriptor is not a remove header manipulation, the function will fail;

- **new_remove_params** - data structure containing the header manipulation attributes to modify;
- **modify_flags** - combination of flags indicating which header manipulation attributes to modify (and hence indicating which of the attributes in the new_remove_params data structure are valid). Select the flag values from the dpa_cls_hm_remove_modify_flags enum and combine them using the "or" logical operand.

Description

Modify the parameters of an existing remove type header manipulation.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- -EINVAL, if there are errors in the parameters provided to the function;
- -EBUSY, if an FMan low level driver function call has failed.

7.4.3.2.2.1.2.4 Function: dpa_classif_modify_insert_hm

Syntax

```
int dpa_classif_modify_insert_hm(int          hmd,
const struct dpa_cls_hm_insert_params  *new_insert_params,
int          modify_flags);

enum dpa_cls_hm_insert_modify_flags {
    /* Ethernet and PPPoE insert group */
    DPA_CLS_HM_INS_MOD_ETH_HEADER      = 0x01,
    DPA_CLS_HM_INS_MOD_QTAGS_ARRAY     = 0x02,
    DPA_CLS_HM_INS_MOD_PPPoE_HEADER    = 0x04,

    /* PPP insert group */
    DPA_CLS_HM_INS_MOD_PPP_PID         = 0x08,

    /* Custom insert group */
    DPA_CLS_HM_INS_MOD_CUSTOM_OFFSET   = 0x10,
    DPA_CLS_HM_INS_MOD_CUSTOM_DATA     = 0x20
};
```

Parameters

- **hmd** - descriptor of the insert header manipulation object to modify parameters for; if the header manipulation designated by this descriptor is not an insert header manipulation, the function will fail;
- **new_insert_params** - data structure containing the header manipulation attributes to modify;
- **modify_flags** - combination of flags indicating which header manipulation attributes to modify (and hence indicating which of the attributes in the new_insert_params data structure are valid). Select the flag values from the dpa_cls_hm_insert_modify_flags enum and combine them using the "or" logical operand. The user can only combine flags from a single group (either only Ethernet and PPPoE insert group flags or PPP insert group flags, and so on).

Description

Modify the parameters of an existing insert header manipulation.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOMEM`, if dynamic memory allocations have failed;
- `-EBUSY`, if an FMan low level driver function call has failed.

7.4.3.2.2.1.2.5 Function: `dpa_classif_modify_update_hm`

Syntax

```
int dpa_classif_modify_update_hm(int          hmd,
const struct dpa_cls_hm_update_params *new_update_params,
int          modify_flags);

enum dpa_cls_hm_update_modify_flags {
    DPA_CLS_HM_UPDATE_MOD_IPHDR          = 0x0001,

    /* L3 protocol flags group */
    DPA_CLS_HM_UPDATE_MOD_SIP            = 0x0002,
    DPA_CLS_HM_UPDATE_MOD_DIP            = 0x0004,
    DPA_CLS_HM_UPDATE_MOD_TOS_TC         = 0x0008,
    DPA_CLS_HM_UPDATE_MOD_IP_ID          = 0x0010,
    DPA_CLS_HM_UPDATE_MOD_L3_FLAGS       = 0x0020,

    /* L4 protocol flags group */
    DPA_CLS_HM_UPDATE_MOD_SPORT          = 0x0040,
    DPA_CLS_HM_UPDATE_MOD_DPORT          = 0x0080,
    DPA_CLS_HM_UPDATE_MOD_L4_FLAGS       = 0x0100,

    DPA_CLS_HM_UPDATE_MOD_IP_FRAG_MTU    = 0x0200,
    DPA_CLS_HM_UPDATE_MOD_IP_FRAG_SCRATCH_BPID = 0x0400,
    DPA_CLS_HM_UPDATE_MOD_IP_FRAG_DF_ACTION = 0x0800
};
```

Parameters

- **hmd** - descriptor of the update header manipulation object to modify parameters for; if the header manipulation designated by this descriptor is not an update header manipulation, the function will fail;
- **new_update_params** - data structure containing the header manipulation attributes to modify;
- **modify_flags** - combination of flags indicating which header manipulation attributes to modify (and hence indicating which of the attributes in the `new_update_params` data structure are valid). Select the flag values from the `dpa_cls_hm_update_modify_flags` enum and combine them using the "or" logical operand. Avoid combinations between flags from the L3 protocol group with flags from the L4 protocol group.

Description

Modify the parameters of an existing update header manipulation.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;

- -EBUSY, if an FMan low level driver function call has failed.

7.4.3.2.2.1.2.6 Function: dpa_classif_modify_vlan_hm

Syntax

```
int dpa_classif_modify_vlan_hm(int          hmd,
const struct dpa_cls_hm_vlan_params *new_vlan_params,
int          modify_flags);

enum dpa_cls_hm_vlan_modify_flags {
    DPA_CLS_HM_VLAN_MOD_INGRESS_NUM_QTAGS    = 0x01,

    DPA_CLS_HM_VLAN_MOD_EGRESS_QTAGS        = 0x02,
    DPA_CLS_HM_VLAN_MOD_EGRESS_UPDATE_OP    = 0x04,
    DPA_CLS_HM_VLAN_MOD_EGRESS_VPRI         = 0x08,
    DPA_CLS_HM_VLAN_MOD_EGRESS_DSCP_TO_VPRI_ARRAY = 0x10
};
```

Parameters

- **hmd** - descriptor of the VLAN specific header manipulation object to modify parameters for; if the header manipulation designated by this descriptor is not a VLAN specific header manipulation, the function will fail;
- **new_vlan_params** - data structure containing the header manipulation attributes to modify;
- **modify_flags** - combination of flags indicating which header manipulation attributes to modify (and hence indicating which of the attributes in the new_vlan_params data structure are valid). Select the flag values from the dpa_cls_hm_vlan_modify_flags enum and combine them using the "or" logical operand. The flag DPA_CLS_HM_VLAN_MOD_INGRESS_NUM_QTAGS cannot be combined with any other flags.

Description

Modify the parameters of an existing VLAN specific header manipulation.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- -EINVAL, if there are errors in the parameters provided to the function;
- -ENOMEM, if dynamic memory allocations have failed;
- -EBUSY, if an FMan low level driver function call has failed.

7.4.3.2.2.1.2.7 Function: dpa_classif_modify_mpls_hm

Syntax

```
int dpa_classif_modify_mpls_hm(int    hmd,
const struct dpa_cls_hm_mpls_params *new_mpls_params,
int          modify_flags);

enum dpa_cls_hm_mpls_modify_flags {
    DPA_CLS_HM_MPLS_MOD_NUM_LABELS = 0x01,
    DPA_CLS_HM_MPLS_MOD_HDR_ARRAY  = 0x02,
};
```

```
};
```

Parameters

- **hmd** - descriptor of the MPLS specific header manipulation object to modify parameters for; if the header manipulation designated by this descriptor is not a MPLS specific header manipulation, the function will fail;
- **new_mpls_params** - data structure containing the header manipulation attributes to modify;
- **modify_flags** - combination of flags indicating which header manipulation attributes to modify (and hence indicating which of the attributes in the `new_mpls_params` data structure are valid). Select the flag values from the `dpa_cls_hm_mpls_modify_flags` enum and combine them using the or logical operand.

Description

Modify the parameters of an existing MPLS specific header manipulation.

Return Value

Zero if successful or a negative error code otherwise. The returned error codes are the following:

- `-EINVAL`, if there are errors in the parameters provided to the function;
- `-ENOMEM`, if dynamic memory allocations have failed;
- `-EBUSY`, if an FMan low level driver function call has failed.

7.4.3.2.3 Multicast

The multicast module allows creation and management of specific entries into the classifier tables, entries identified as **multicast groups**. A group has **at least one member** identified as an enqueue action. The user can define a maximum number of members inside a group. Members can also have header manipulation chains attached. Header manipulations on any member, except the last member, **must not change the frame's header protocol stack and size**. No such limitation exists on the last member from a group. A virtual storage profile Id can also be set on each member of a group.

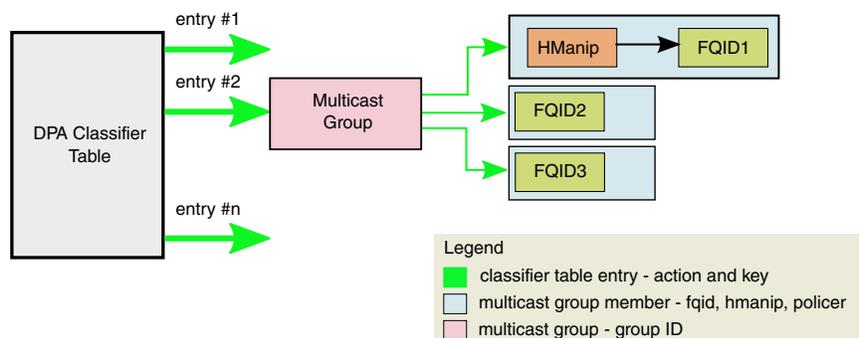


Figure 135. Multicast example

After creating a multicast group, new members can be added by filing the `membr_params`. The newly created member is identified by the `md` descriptor. Up to 64 members can be added in a group. Another step after creating a group is to create an entry of type multicast in a classification table. When calling `dpa_classif_table_insert_entry` the group descriptor (`grp_d`) will be provided as input parameter for the new entry. If an entry of type multicast is no longer needed it can be removed using the `entry_id` or the call `dpa_classif_table_delete_entry_by_key` (the entry is identified by key). Members of the multicast group can be individually removed by calling `dpa_classif_mcast_remove_member`. The group can be removed by calling `dpa_classif_mcast_free_group`.

7.4.3.2.3.1 Multicast API

There are two selections for the DPA Multicast API:

7.4.3.2.3.1.1 Create and Remove Multicast Groups

Table 128. API to Create and Remove Multicast Groups

Function Name	Description	Input Parameters	Output Parameters
dpa_classif_mcast_create_group	Creates a multicast group with one initial member. If external group handle is given the function imports an existing group from preallocated resources	<ul style="list-style-type: none"> max number of members in multicast group; handle of the PCD needed to create the group enqueue parameters of the initial member. prefilled members – number of members in the imported group. 	Descriptor of the newly created multicast group.
dpa_classif_mcast_free_group	Removes an existing group.	<ul style="list-style-type: none"> group descriptor identifying the multicast group to remove. 	None.

7.4.3.2.3.1.1.1 Function: dpa_classif_mcast_create_group

Syntax

```
int dpa_classif_mcast_group_create(struct dpa_cls_mcast_group_params
    *mcast_group_params, int *grpId);

/* Multicast group parameters */
struct dpa_cls_mcast_group_params {
    uint8_t    max_members;
    void      *fm_pcd;
    struct dpa_cls_tbl_enq_action_desc first_member_params;
    unsigned int  prefilled_members;
    void      *group;
    void      *distribution;
};
```

Parameters

- mcast_group_params** - group parameters
 - max_members** - maximum number of members in group.
 - fm_pcd** - handle of the PCD needed when creating a group.
 - member_params** – enqueue parameters for the member (fqid, policer, header manipulation descriptor, storage profile id).
 - override_fqid** – nonzero if the action descriptor corresponding to the member is of type new classification result and zero if the action descriptor is of type keep classification result;

- **new_fqid** – if `override_fqid` is nonzero, this holds the explicit frame queue Id where to place the frame;
- **policer_params** – policing parameters; if NULL, no policing is performed during the enqueue operation.
 - **modify_policer_params** – nonzero if the default policer parameters will be overridden;
 - **shared_profile** - nonzero if this policer profile is shared between ports; relevant only if `modify_policer_params` is nonzero;
 - **new_rel_profile_id** - this parameter should indicate the policer profile offset within the port's policer profiles or from the SHARED window; relevant only if `modify_policer_params` is nonzero;
- **hmd** – header manipulation object descriptor, or -1 if no header manipulation is required for the enqueue action; the header manipulation object defined by the provided descriptor must be a header manipulation chain head;
- **spid** – storage profile id. This parameter will specify the storage profile, in order to allocate new external buffers for the frame descriptor received on the member.
- **prefilled_members** – Number of members that already exist in the imported group.
- **group** – External group handle given as input parameter for an import operation.
- **distribution** – External distribution handle. When provided, replicated frames are not enqueued to members' frame queues. They are sent to this distribution.
- **grp** - Address of an integer variable in which the function will return the group descriptor. This value will be used in other function calls, to identify the group on which operations will be performed.

Description

This function creates or imports a multicast group. The new group is identified by the group descriptor - **grp**. It can be assigned to a new entry in the classification table using the function `dpa_classif_table_insert_entry`, which will have the type `DPA_CLS_TBL_ACTION_MCAST` for action parameter. If external group handle is provided, preallocated resources will be used – and the existing group will be used to add new members to it. Prefilled members will specify the number of existing members in the imported group.

Return Value

The function will return 0 on success and a negative value otherwise. The returned error codes are the following:

- -EINVAL, if `mcast_group_params` parameter is NULL;
- -EINVAL, if `grp` parameter is NULL;
- -EINVAL, if `max_member` is 0 or has a value greater than maximum number of supported members (64);
- -EINVAL, if invalid header manip descriptor (`hmd`) for the member defined in the new group. (A new group has at least one member);
- -EINVAL, if header manipulation descriptor of the member (`hmd`) is not a chain head;
- -EINVAL, if the group could not be created when low level function was invoked;
- -ENOMEM, if no more memory for the new multicast group;
- -ENOMEM, if no more memory for descriptor table when allocating descriptor for the new group;
- -ENOMEM, if no more memory for the member index array used to assign member ids;
- -ENOSYS, if policing params were provided for the first group member.

7.4.3.2.3.1.1.2 Function: `dpa_classif_mcast_free_group`

Syntax

```
int dpa_classif_mcast_free_group(int grpd);
```

Parameters

- **grpd** - Multicast group descriptor. This value identifies a group that was created before this function call.

Description

This function removes an existing group identified by the **grpd** parameter.

Return Value

The function will return 0 on success and a negative value otherwise. The returned error codes are the following:

- -EINVAL, if Invalid group descriptor grpd provided to the function;
- -EINVAL, if the group could not be removed when low level function was invoked.

7.4.3.2.3.1.2 Add and Remove Multicast Group Members

Table 129. API to Add and Remove Multicast Group Members

Function Name	Description	Input Parameters	Output Parameters
dpa_classif_mcast_add_member	Adds a new member to an existing group	<ul style="list-style-type: none"> • group descriptor • enqueue parameters of the new member (fqid, policer, header manip desc, spid) 	Descriptor of the newly created member.
dpa_classif_mcast_remove_member	Removes an existing member from an existing group. The member is identified by its member descriptor.	<ul style="list-style-type: none"> • group descriptor • member descriptor identifying the member to be removed from the group. 	None.

7.4.3.2.3.1.2.1 Function: dpa_classif_mcast_add_member

Syntax

```
int dpa_classif_mcast_add_member(int grpd,
    struct dpa_cls_tbl_enq_action_desc member_params, int *md);
```

Parameters

- **grpd** - Multicast group descriptor. This value identifies a group that was created before this function call.
- **member_params** - enqueue parameters for the member (fqid, policer, header manipulation descriptor, storage profile.
 - **override_fqid** – nonzero if the action descriptor corresponding to the member is of type new classification result and zero if the action descriptor is of type keep classification result;
 - **new_fqid** – if override_fqid is nonzero, this holds the explicit frame queue Id where to place the frame;

- **policer_params** – policing parameters; if NULL, no policing is performed during the enqueue operation.
- **modify_policer_params** – nonzero if the default policer parameters will be overridden;
- **shared_profile** - nonzero if this policer profile is shared between ports; relevant only if modify_policer_params is nonzero;
- **new_rel_profile_id** - this parameter should indicate the policer profile offset within the port's policer profiles or from the SHARED window; relevant only if modify_policer_params is nonzero;
- **hmd** – header manipulation object descriptor, or -1 if no header manipulation is required for the enqueue action; the header manipulation object defined by the provided descriptor must be a header manipulation chain head;
- **spid** – storage profile id. This parameter will specify the storage profile, in order to allocate new external buffers for the frame descriptor received on the member.
- **md** - Address of an integer variable in which the function will return the member descriptor. This is a reference to an internal index updated by the FMD driver when a member is added or removed from the group.

Description

This function adds a new member to an existing group identified by the **grp**d parameter. The new created member will be identified by the member descriptor md.

Return Value

The function will return 0 on success and a negative value otherwise. The returned error codes are the following:

- -EINVAL, if an invalid group descriptor (grp)d was provided to the function;
- -EINVAL, if member_params parameter is NULL;
- -EINVAL, if md is NULL;
- -EINVAL, if invalid header manip descriptor (hmd) for the new member;
- -EINVAL, if header manipulation descriptor (hmd) of the member is not a chain head;
- -EINVAL, if the multicast member could not be added when low level function was invoked;
- -ENOSPC, if the current number of members reached maximum value max_members (defined when the group was created);
- -ENOSYS, if policing params were provided.

7.4.3.2.3.1.2.2 Function: dpa_classif_mcast_remove_member

Syntax

```
int dpa_classif_mcast_remove_member(int grp_d, int md);
```

Parameters

- **grp**d - Multicast group descriptor. This value identifies a group that was created before this function call.
- **md** - member descriptor which identifies the member inside a group

Description

This function removes an existing member from a group identified by the **grp**d parameter. The member is identified by member descriptor.

Return Value

The function will return 0 on success and a negative value otherwise. The returned error codes are the following:

- -EINVAL, if an invalid group descriptor (`grp_d`) was provided to the function;
- -EINVAL, if an invalid member descriptor `md` (value is negative or greater than `max_members`) was provided to the function;
- -EINVAL, if the multicast member could not be removed when low level function was invoked.

7.4.3.3 DPA IPsec

The DPA IPsec module exports a set of functions which can be used to:

- initialize the DPA IPsec module internal data structures;
- create and configure full inbound IPsec hardware accelerated paths;
- create and configure full outbound IPsec hardware accelerated paths;
- replace expired SAs (after rekeying) without packet loss.

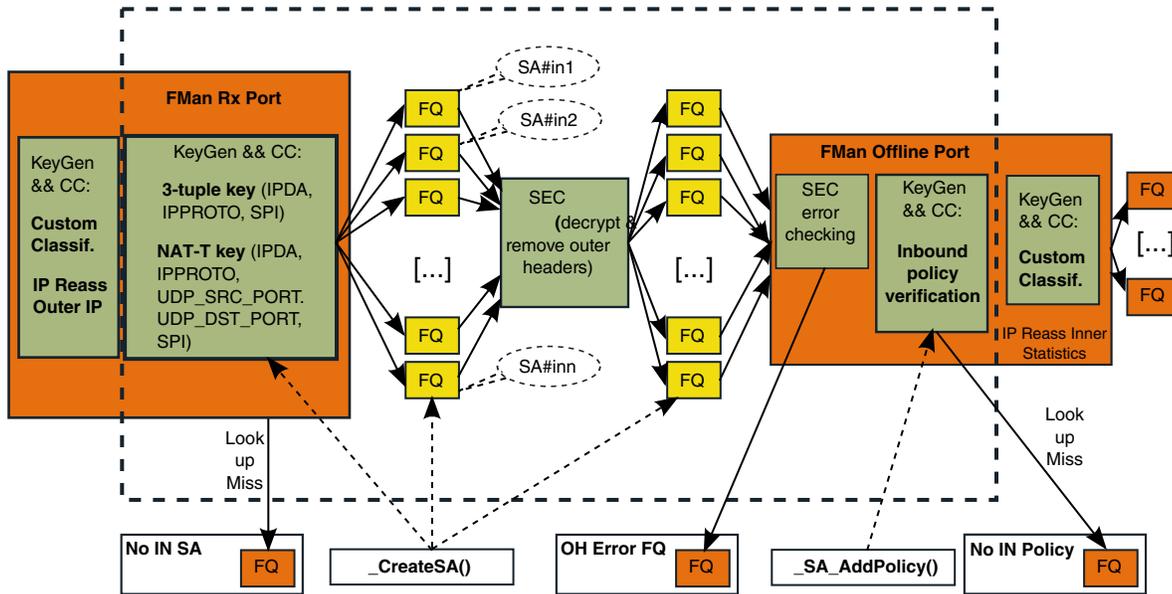


Figure 136. DPA IPsec inbound path architecture overview

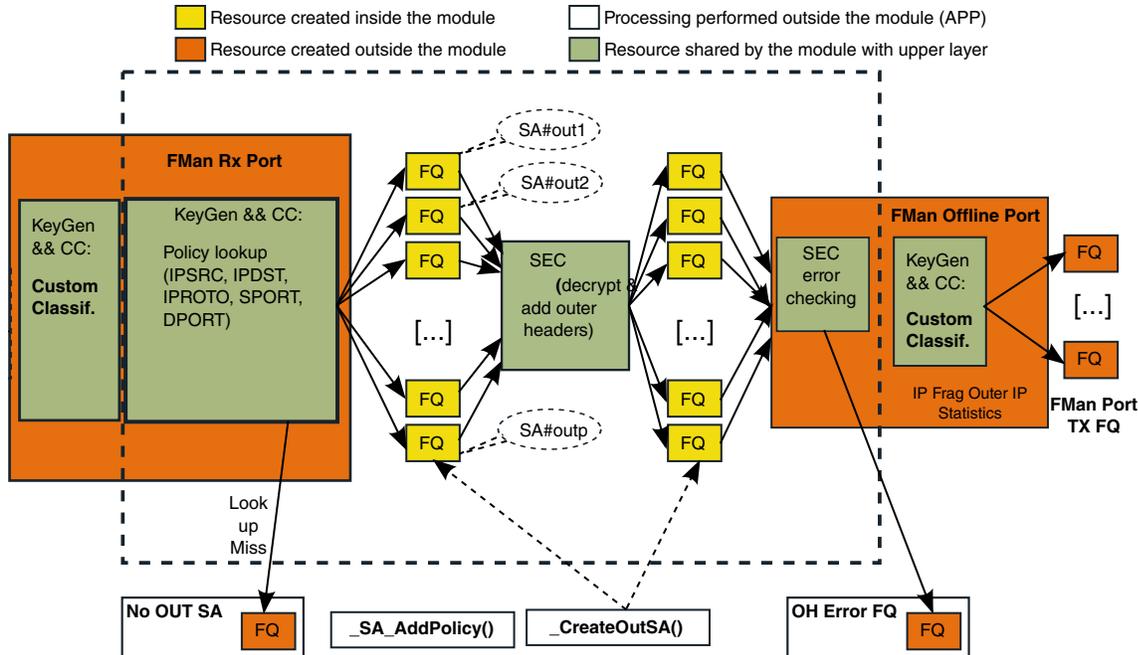


Figure 137. DPA IPsec outbound path architecture overview

After the accelerated paths for the inbound (figure above) and the outbound (figure above) are configured, the calling application can send and receive IPsec protected packets just by enqueueing and dequeuing frames. Although, from the calling applications perspective, all paths (inbound or outbound) are configured using similar function calls, the implementation of these hardware accelerated paths differs substantially. The DPA IPsec API is responsible for hiding these implementation details.

7.4.3.1 Initialization

Inbound path creation

Initialization is a mandatory step for enabling IPsec offloading. The initialization functions are designed to create and initialize the internal data structures of the module, which will later be used as a base for creating inbound or outbound accelerated IPsec processing paths. These configuration tasks are performed only once and are meant to enable the required hardware blocks, apply an initial configuration and initialize features that do not require dynamic configuration.

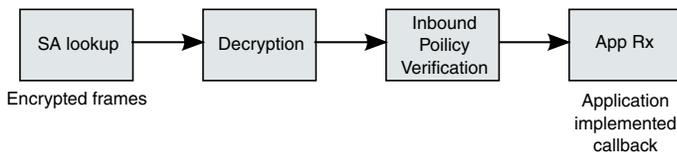


Figure 138. Inbound processing decomposition

Creating an inbound accelerated IPsec path means configuring the hardware blocks to classify the encrypted packets, decapsulate and decrypt them, perform required policy checks on the decrypted packet (as stated in the RFCs) and finally deliver the resulting packet to the calling application. The hardware blocks that process the packets before and after decryption and decapsulation, can also be configured to perform IP reassembly. Before decryption IP reassembly acts on the encrypted packets, while after decryption it acts on the inner, decrypted packets.

Because IP reassembly can (presently) only be enabled on a per port basis in FMan, it is beyond the scope of the DPA IPsec API to provide a means to configure and enable this feature. It is the responsibility of the upper layer applications to enable and configure IP reassembly when it initializes the PCD skeleton.

Outbound path creation

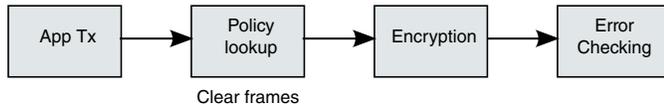


Figure 139. Outbound processing decomposition

An accelerated outbound IPSec path is supposed to use the hardware blocks to speed up the processing of an IP packet from the point where it is delivered for IPSec processing by the calling application and till a correct IPSec encapsulated packet has been constructed. Different hardware blocks have to be configured to classify the supplied packet and identify the correct outbound SA, encrypt and encapsulate the packet and finally check for errors and deliver the packet to an offline port. The calling application must implement on this offline port the classification needed to forward the packet. This processing is outside the scope of the DPA IPSec module.

7.4.3.3.2 DPA IPSec API

There are several selections for the DPA IPSec API:

7.4.3.3.2.1 IPSec Engine Startup and Shutdown

Table 130. API for IPSec Engine Startup and Shutdown

Function Name	Description
<code>dpa_ipsec_init</code>	The model's layout is fixed so most of the required configuration parameters are held internally. The function builds the necessary infrastructure for managing future runtime operations.
<code>dpa_ipsec_free</code>	Releases all the resources allocated previously by this module and also the managed (run-time added) ones.

7.4.3.3.2.1.1 Function: `dpa_ipsec_init`

Syntax

```

int dpa_ipsec_init(const struct dpa_ipsec_params *params,
                  int *dpa_ipsec_id);

#define DPA_IPSEC_KEY_FIELD_SIP 0x01
#define DPA_IPSEC_KEY_FIELD_DIP 0x02
#define DPA_IPSEC_KEY_FIELD_PROTO 0x04
#define DPA_IPSEC_KEY_FIELD_DSCP 0x08
#define DPA_IPSEC_KEY_FIELD_SPORT 0x10
#define DPA_IPSEC_KEY_FIELD_ICMP_TYPE 0x10
#define DPA_IPSEC_KEY_FIELD_DPORT 0x20
#define DPA_IPSEC_KEY_FIELD_ICMP_CODE 0x20

enum dpa_ipsec_proto {
  DPA_IPSEC_PROTO_TCP_IPV4 = 0,
  DPA_IPSEC_PROTO_TCP_IPV6,
  DPA_IPSEC_PROTO_UDP_IPV4,
  DPA_IPSEC_PROTO_UDP_IPV6,
  DPA_IPSEC_PROTO_ICMP_IPV4,
  DPA_IPSEC_PROTO_ICMP_IPV6,
};

```

```
DPA_IPSEC_PROTO_SCTP_IPV4,  
DPA_IPSEC_PROTO_SCTP_IPv6,  
DPA_IPSEC_PROTO_ANY_IPV4,  
DPA_IPSEC_PROTO_ANY_IPV6,  
DPA_IPSEC_MAX_SUPPORTED_PROTOS  
};  
  
enum dpa_ipsec_sa_type {  
    DPA_IPSEC_SA_IPV4 = 0,  
DPA_IPSEC_SA_IPV4_NATT,  
    DPA_IPSEC_SA_IPV6,  
    DPA_IPSEC_MAX_SA_TYPE  
};  
  
enum dpa_ipsec_data_off {  
    DPA_IPSEC_DATA_OFF_NONE = 0,  
    DPA_IPSEC_DATA_OFF_1_BURST,  
    DPA_IPSEC_DATA_OFF_2_BURST,  
    DPA_IPSEC_DATA_OFF_3_BURST  
};  
  
struct dpa_ipsec_pol_table {  
    int dpa_cls_td;  
    uint8_t key_fields;  
};  
  
struct dpa_ipsec_pre_sec_in_params {  
    int dpa_cls_td[DPA_IPSEC_MAX_SA_TYPE];  
    int gw_change_en;  
};  
  
struct dpa_ipsec_pre_sec_out_params {  
    struct dpa_ipsec_pol_table table[DPA_IPSEC_MAX_SUPPORTED_PROTOS];  
};  
  
struct dpa_ipsec_post_sec_in_params {  
    enum dpa_ipsec_data_off data_off;  
    uint16_t tx_ch;  
    int dpa_cls_td;  
    bool do_pol_check;  
    uint8_t key_fields;  
    bool use_ipv6_pol;  
    uint16_t base_flow_id;  
};  
  
struct dpa_ipsec_post_sec_out_params {  
    enum dpa_ipsec_data_off data_off;  
    uint16_t tx_ch;  
};  
  
struct dpa_ipsec_fqid_range {  
    uint32_t start_fqid;  
    uint32_t end_fqid;  
};  
  
struct dpa_ipsec_params {  
    struct dpa_ipsec_pre_sec_in_params pre_sec_in_params;  
    struct dpa_ipsec_post_sec_in_params post_sec_in_params;  
    struct dpa_ipsec_pre_sec_out_params pre_sec_out_params;
```

```
struct dpa_ipsec_post_sec_out_params post_sec_out_params;
void      *fm_pcd;
uint16_t   qm_sec_ch;
unsigned int max_sa_pairs;
    struct dpa_ipsec_fqid_range *fqid_range;
    unsigned int max_sa_manip_ops;
};
```

Parameters

- **params** - structure containing all the parameters required for initializing a DPA IPsec instance:
 - **pre_sec_in_params** – structure containing the parameters required to configure and initialize the inbound SA lookup block:
 - **dpa_cls_td[..]** – array of descriptors for DPA Classifier tables, indexed by SA type (i.e. IPv4, IPv4 with NAT-T support, IPv6 – `dpa_ipsec_sa_type` enum), which will be used for identifying the SA that should process an incoming frame; the mechanisms used by the DPA IPsec instance to choose the table in which a policy will be inserted.
 - **gw_change_en** – enable the remote gateway address change detection mechanism (currently not supported)
 - **post_sec_in_params** - structure containing the parameters required to configure and initialize the inbound policy verification block:
 - **data_off** – the data offset that will be configured in the frame descriptor of all decrypted frames; configured using the `dpa_ipsec_data_off` enum; please refer to function description for details;
 - **tx_ch** – the qman channel id associated to the post decryption offline port
 - **dpa_cls_td** - the DPA IPsec module expects to receive from the upper layer a table descriptor for a DPA Classifier Indexed Table object which it will use to determine on which SA a frame was decrypted; it is mandatory that the table descriptor provided here is that of an indexed table
 - **do_pol_check** – enable the inbound policy verification
 - **key_fields** – the policy key structure (which fields of the policy parameters structure should be included in the policy key); relevant only if `do_pol_check` is TRUE; configured using the `DPA_IPSEC_KEY_FIELD_*` macros;
 - **use_ipv6_pol** - activate support for IPv6 policies. Allows better MURAM management. Relevant only if `do_pol_check = TRUE`.
 - **base_flow_id** – the base value from which inbound SA flow ID values can be allocated incrementally;
 - **pre_sec_out_params** - structure containing the parameters required to configure and initialize the outbound policy lookup block:
 - **table[..]** – array of structures defining the parameters for the DPA Classifier tables that will be used for offloading policy lookup; this array is indexed by policy protocol type and IP header version (`dpa_ipsec_proto` enum); the mechanisms used by the DPA IPsec instance to choose the table in which a policy will be inserted:
 - **dpa_cls_td** – table descriptors for DPA Classifier Exact Match Table objects, which will be used to determine which SA should be used for encrypting the frame;
 - **key_fields** – defines the policy key structure (which fields of the policy parameters structure should be included in the key) used in the table identified by the `dpa_cls_td` parameter; configured using the `DPA_IPSEC_KEY_FIELD_*` macros;
 - **post_sec_out_params** - structure containing the parameters required to configure and initialize the outbound error checking block:
 - **data_off** – the data offset that will be configured in the frame descriptor of all encrypted frames; configured using the `dpa_ipsec_data_off` enum; please refer to function description for details;
 - **tx_ch** – the qman channel id associated to the post encryption offline port;

- **fm_pcd** - handle to the low level driver PCD
- **qm_sec_ch** – the qman channel id associated to the SEC;
- **max_sa_pairs** – the maximum number of SAs pairs supported by the DPA IPsec module;
- **fqid_range** – pointer to a structure defining the boundaries of a FQID range from which the DPA IPsec instance should allocate fqids for the frame queues it creates internally; if this parameter is NULL, then the DPA IPsec instance will try to acquire fqids using the default allocation mechanism;
- **max_sa_manip_ops** – maximum number of manipulation objects (used for variable IP header length support or DSCP / ECN propagation support) to be preallocated at initialization; this value should be incremented with the number of header manipulations per every outbound policy; if 0, the required manipulation objects will be allocated at runtime;
- **dpa_ipsec_id** - upon successful initialization, this parameter will contain the identifier of this DPA IPsec instance; this identifier must be passed to all the other runtime functions when they are called (currently not supported).

Description

This function is used to initialize a DPA IPsec instance. When this function exits all the required internal structures of the DPA IPsec instance will be initialize and the other API functions can be used to configure offloading for outbound and inbound IPsec flows.

Some of the parameters of this function are handles to DPA Classifier Table objects. These tables must be initialized before this function is called.

The DPA IPsec instance expects to receive from the upper layer one or more table descriptors for DPA Classifier Table objects which it will use to perform SA lookup. Usually a hash table is used for this purpose, because it supports a large number of entries, but an exact match table can also be used.

If not all the tables for SA lookup are necessary (e.g. no IPv6 or NAT-T support is required), then the upper layer can pass invalid descriptors for the tables that will not be used, thus saving resources.

If more than one classifier table is used for SA lookup, the `max_sa_pairs` field will represent the sum of all the SAs that can be classified on the inbound path using all of the tables identified by valid descriptors (see the fields of the `pre_sec_in_params` structure)

If DPA Classifier Hash Tables are used for SA lookup (this is recommended), the value of the `max_sa_pairs` field should be smaller than the sum of the maximum number of entries supported by each hash table (`dpa_cls_td_*` in the `pre_sec_in_params`), because a hash table does not offer a uniform distribution of entries in its buckets. Also, the size of the indexed table in the `post_sec_in_params` structure must be greater or equal to the value of `max_sa_pairs` field.

The order in which the fields of the policy parameters structure should be place in the key is given by the values of the key fields identifiers (the `DPA_IPSEC_KEY_FIELD_*` macros) used for configuring the key structure in the `key_fields` members of the `pre_sec_out_params` and `post_sec_in_params` structures. The field the lowest identifier value is the first in the key and the field whith the highest identifier value is the last in the key.

The `data_off` is important to be configured correctly, especially when further processing of the encrypted / decrypted frames is expected to take place. Some of the modules, that will process the frames after they have been encrypted / decrypted by the SEC, require that extra space be available in the data buffers, in front of the actual data (e.g. IP Frag., IP Reass.).

The SEC can be configured not to place the output data at the beginning of the output buffer, but it restricts the values of the offset to be used. The `data_off` parameter is represented as a multiple of the SEC burst size, the maximum burst size used by the SEC when performing memory accesses. The SEC burst size can be either 32 or 64 bytes and it is controlled by a bit in the SEC control register. The default value for the SEC burst size is 64 bytes. Based on this, the following guidelines must be considered when configuring the `data_off` parameter:

- if classification is to be performed on the frames, after the SEC finishes encrypting / decrypting them a minimum offset of 64 bytes must be configured;
- if IP fragmentation / reassembly is to be performed on the frames, after the SEC finishes encrypting / decrypting them, an offset of 192 bytes should be configured;

- in any other situation an offset of 0 should be configured or a value consistent with the requirements of other applications / modules that are known to process the frames after SEC.

The value of the `base_flow_id` is important to be set correctly when the post decryption offline port is also used to perform other classifications based on flow ID. In these cases, it is recommended that a range of values for flow ID be reserved for all other classifications not related to IPSec, starting from flow ID value 0 and the `base_flow_id` to be equal to the number of flow ID values in that range. In all other cases the value of this field should be 0.

If the `gw_change_en` field value is `FALSE`, the remote gateway address change detection mechanism will be disabled. If the value is `TRUE`, this mechanism will be enabled and the calling application will be notified if an event of this type occurs through the special callback provided at SA creation time. If not callback is provided and such an event occurs, the frame that generated this event will be processed according to the miss action of the inbound SA lookup table.

The `max_sa_manip_ops` field can be used to preallocate a number of manipulation objects that will be used later (when an SA is offloaded) for the variable IP header length, variable IP version and DSCP / ECN propagation features and thus eliminate the need to allocate MURAM memory at runtime. The preallocated objects are placed in an internally managed pool and the implementation will extract objects from this pool whenever a SA that requires one is offloaded and put an object back into the pool whenever a SA that uses one is removed.

When calculating the value of this field the following aspects should be taken into consideration:

- for each outbound policy with header manipulation enabled one such manipulation object is required;
- for each inbound SA with or without the DSCP / ECN propagation or IP header length or variable IP version feature enabled one such manipulation object is required;
- for each inbound SA with the variable IP header length feature enabled one such manipulation object is required;
- for an inbound SA that has enabled both the DSCP / ECN propagation and the variable IP header length features only one manipulation object is required.

If the value of this field is 0 and one of those features is enabled, the IPSec implementation will allocate the required manipulation objects at runtime.

Return Value

The function will return 0 on success and a negative value otherwise. The returned error codes are the following:

- `-EPERM`, if a DPA IPSec instance is already initialized (multiple DPA IPSec instances are not supported yet) or the circular queue for inbound flow ID management could not be initialized;
- `-EINVAL`, if invalid function arguments or DPA IPSec initialization parameters were provided;
- `-ENOMEM`, if memory could not be allocated for one of the internal structures;
- `-ENOMEM`, if the internal ID management queues could not be initialized;
- `-EFAULT`, if a bad argument was passed to a internal function;
- `-EDOM`, if the size of the internal ID management queues was exceeded while trying to populate them;
- `-EBUSY`, if the creation of the CCNodes required for inbound policy verification has failed (as a result of an error in FMD function call);
- all the error codes returned by the `dpa_classifier_create_table` function, which is used for creating tables for inbound policy verification;
- `-ENOSPC`, if the deferred work workqueue used in the rekeying process could not be initialized.

7.4.3.3.2.1.2 Function: *dpa_ipsec_free*

Syntax

```
int dpa_ipsec_free(int dpa_ipsec_id);
```

Parameters

- **dpa_ipsec_id** - identifier of the DPA IPsec instance that should be freed - (currently not supported).

Description

Releases all the resources allocated previously for this instance of the DPA IPsec API.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- all the error codes returned by the `dpa_ipsec_flush_all_sa` function.

7.4.3.3.2.2 Create, Remove and Manage SAs

Table 131. API for Creating, Removing and Managing SAs

Function Name	Description
<code>dpa_ipsec_create_sa</code>	Prepares the accelerated path for IPsec flows that will use this SA to encrypt or decrypt packets. Specific packets will be directed through this flow using policies (outbound) or specific SA selectors (inbound)
<code>dpa_ipsec_remove_sa</code>	Releases resources used by an SA.
<code>dpa_ipsec_flush_all_sa</code>	Removes all offloaded SAs from the DPA IPsec module
<code>dpa_ipsec_disable_sa</code>	Disables an SA before removal. No other packets will be processed using this SA. Resources associated to this SA will not be freed.
<code>dpa_ipsec_sa_modify</code>	Modifies all or just a part of the parameters of a previously offloaded SA.
<code>dpa_ipsec_sa_get_stats</code>	Returns statistics for a specified SA.
<code>dpa_ipsec_get_sa_context</code>	Retrieves information about a SA based on the packet that was processed by that SA.
<code>dpa_ipsec_sa_get_out_path</code>	Retrieves information which can be used to apply IPsec processing to a packet without performing the outbound policy lookup.

7.4.3.3.2.1 Function: *dpa_ipsec_create_sa*

Syntax

```
int dpa_ipsec_create_sa(int dpa_ipsec_id,
struct dpa_ipsec_sa_params *sa_params,
int *sa_id);

#define DPA_IPSEC_HDR_COPY_TOS 0x01
#define DPA_IPSEC_HDR_COPY_DF 0x02
#define DPA_IPSEC_HDR_DEC_TTL 0x04
#define DPA_IPSEC_HDR_COPY_DSCP 0x08
#define DPA_IPSEC_HDR_COPY_ECN 0x10

enum dpa_ipsec_direction {
    DPA_IPSEC_INBOUND = 0, /* Inbound */
    DPA_IPSEC_OUTBOUND /* Outbound */
};

struct dpa_ipsec_init_vector {
    uint8_t *init_vector;
    uint8_t length;
};

enum dpa_ipsec_arw {
    DPA_IPSEC_ARSNONE = 0, /* No Anti Replay Protection */
    DPA_IPSEC_ARS32 = 1, /* 32 bit Anti Replay Window size */
    DPA_IPSEC_ARS64 = 3 /* 64 bit Anti Replay Window size */
};

enum dpa_ipsec_cipher_alg {
    DPA_IPSEC_CIPHER_ALG_3DES_CBC_HMAC_96_MD5_128 ,
    DPA_IPSEC_CIPHER_ALG_3DES_CBC_HMAC_96_SHA_160 ,
    DPA_IPSEC_CIPHER_ALG_3DES_CBC_HMAC_MD5_128 ,
    DPA_IPSEC_CIPHER_ALG_3DES_CBC_HMAC_SHA_160 ,
    DPA_IPSEC_CIPHER_ALG_3DES_CBC_HMAC_SHA_256_128 ,
    DPA_IPSEC_CIPHER_ALG_3DES_CBC_HMAC_SHA_384_192 ,
    DPA_IPSEC_CIPHER_ALG_3DES_CBC_HMAC_SHA_512_256 ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_HMAC_96_MD5_128 ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_HMAC_96_SHA_160 ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_AES_XCBC_MAC_96 ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_HMAC_MD5_128 ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_HMAC_SHA_160 ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_HMAC_SHA_256_128 ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_HMAC_SHA_384_192 ,
    DPA_IPSEC_CIPHER_ALG_NULL_ENC_HMAC_SHA_512_256 ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_HMAC_96_MD5_128 ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_HMAC_96_SHA_160 ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_AES_XCBC_MAC_96 ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_HMAC_MD5_128 ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_HMAC_SHA_160 ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_HMAC_SHA_256_128 ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_HMAC_SHA_384_192 ,
    DPA_IPSEC_CIPHER_ALG_AES_CBC_HMAC_SHA_512_256 ,
    DPA_IPSEC_CIPHER_ALG_AES_CTR_HMAC_96_MD5_128 ,
    DPA_IPSEC_CIPHER_ALG_AES_CTR_HMAC_96_SHA_160 ,
    DPA_IPSEC_CIPHER_ALG_AES_CTR_AES_XCBC_MAC_96 ,
    DPA_IPSEC_CIPHER_ALG_AES_CTR_HMAC_MD5_128 ,
```

```

    DPA_IPSEC_CIPHER_ALG_AES_CTR_HMAC_SHA_160    ,
    DPA_IPSEC_CIPHER_ALG_AES_CTR_HMAC_SHA_256_128 ,
    DPA_IPSEC_CIPHER_ALG_AES_CTR_HMAC_SHA_384_192 ,
    DPA_IPSEC_CIPHER_ALG_AES_CTR_HMAC_SHA_512_256
};

struct dpa_ipsec_sa_crypto_params {
enum dpa_ipsec_cipher_alg alg_suite;
    uint8_t      *cipher_key;
    uint8_t      cipher_key_len;
    uint8_t      *auth_key;
    uint8_t      auth_key_len;
};

typedef int (*dpa_ipsec_gw_change_cb)
(int dpa_ipsec_id,
    int sa_id,
    struct dpa_ipsec_ip_address new_src_ip);

enum dpa_ipsec_sa_mode {
    DPA_IPSEC_SA_MODE_TUNNEL = 0,
    DPA_IPSEC_SA_MODE_TRANSPORT
};

enum dpa_ipsec_sa_proto {
    DPA_IPSEC_SA_PROTO_ESP = 0,
    DPA_IPSEC_SA_PROTO_AH
};

struct dpa_ipsec_sa_out_params {
    struct dpa_ipsec_init_vector *init_vector;
    unsigned int      ip_ver;
    uint16_t          ip_hdr_size;
    uint8_t           *outer_ip_header;
    uint8_t           *outer_udp_header;
    uint16_t          post_sec_flow_id;
    uint8_t           dscp_start;
    uint8_t           dscp_end;
};

struct dpa_ipsec_sa_in_params {
    enum dpa_ipsec_arw      arw;
    bool      use_var_iphdr_len;
    struct dpa_offload_ip_address src_addr;
    struct dpa_offload_ip_address dest_addr;
    bool      use_udp_encap;
    uint16_t  src_port;
    uint16_t  dest_port;
    struct dpa_cls_tbl_action policy_miss_action;
    struct dpa_cls_tbl_action post_ipsec_action;
    dpa_ipsec_gw_change_cb *gw_change_cb;
}

struct dpa_ipsec_sa_params {
    uint32_t      spi;
    bool      use_ext_seq_num;
    uint64_t      start_seq_num;
    unsigned int  l2_hdr_size;
    enum dpa_ipsec_sa_mode sa_mode;
    enum dpa_ipsec_sa_proto sa_proto;
};

```

```
uint8_t      hdr_upd_flags;
uint8_t      sa_wqid;
uint8_t      sa_bpid;
uint16_t     sa_bufsize;
bool         enable_stats;
bool         enable_extended_stats;
struct dpa_ipsec_sa_crypto_params  crypto_params;
enum dpa_ipsec_direction          sa_dir;
union {
    struct dpa_ipsec_sa_in_params  sa_in_params;
    struct dpa_ipsec_sa_out_params sa_out_params;
};
};
```

Parameters

- **dpa_ipsec_id** - identifier of the DPA IPsec instance - **(currently not supported)**
- **sa_params** - structure holding the parameters for this SA:
 - **spi** – the IPsec security parameter index for this SA
 - **use_ext_seq_num** – specify whether extended sequence numbers should be used
 - **start_seq_num** – the current sequence number for this SA; this value will be automatically be incremented for each packet processed using this SA
 - **l2_hdr_size** – the size of the Ethernet header, including any VLAN information
 - **sa_mode** – transport or tunnel mode; configured using `dpa_ipsec_sa_mode` enum - **(currently not supported)**
 - **sa_proto** – IPsec protocol used for protecting data (ESP or AH); configured using `dpa_ipsec_sa_proto` enum - **(currently not supported)**
 - **hdr_upd_flags** – a set of flags describing what header fields should be propagated from the inner header to the outer header and vice versa; these flags can be used to control the copying of the entire TOS field or the DSCP bits or the ECN bits from inner to outer header and vice versa, the copying of the DF bit (only) from the inner header to the outer header and the automatic update (decrement) of the TTL in the outer / inner header; configured using the `DPA_IPSEC_HDR_*` macros;
 - **sa_wqid** - work queue ID for all the queues of this SA
 - **sa_bpid** – the buffer pool from which the SEC should acquire buffers for encrypted / decrypted frames
 - **sa_bufsize** – SEC output frames buffer pool buffer size
 - **enable_stats** – enable support for gathering traffic statistics on this SA
 - **enable_extended_stats** - enable input traffic (extended) statistics on this SA. If `enable_stats` is set to `false`, this setting is ignored.
- **crypto_params** – protection used by this SA
 - **alg_suite** – identifier for the encryption and authentication algorithms used by this SA; configured using the `dpa_ipsec_cipher_alg` enum;
 - **cipher_key** – address of the IPsec cipher key
 - **cipher_key_len** - length in bytes of the cipher key
 - **auth_key** – address of the IPsec authentication key
 - **auth_key_len** – length in bytes of the authentication key
- **sa_dir** – SA type (inbound / outbound); configured using the `dpa_ipsec_direction` enum;

- **sa_in_params** – parameters specific for inbound SAs:
 - **arw** - type of anti replay protection requested; configured using the `dpa_ipsec_arw` enum;
 - **use_var_iphdr_len** – enable support for variable IP header length
 - **src_addr** – source IP address in the outer header
 - **dest_addr** – destination IP address in the outer header
 - **use_udp_encap** – specify whether this is a NAT-T SA;
 - **src_port** – source udp port value in the UDP encapsulation header; relevant only if `useUdpEncap` is true
 - **dest_port** – destination udp port value in the UDP encapsulation header; relevant only if `useUdpEncap` is true
 - **policy_miss_action** – action descriptor specifying the action that will be performed for frames that fail inbound policy verification; relevant only if inbound policy verification is enabled;
 - **post_ipsec_action** – action descriptor specifying the default action for frames that have passed inbound policy verification; if inbound policy verification is disabled this is the default action for decrypted frames on this SA
 - **gw_change_cb** – pointer to a callback used by the DPA IPSec module to signal to the upper layer that a remote gateway change event has occurred on this SA - (currently not supported)
- **sa_out_params** – parameters specific for outbound SAs:
 - **init_vector** – address of the initialization vector to be used; if this parameter is NULL the internal random number generator of the SEC will be used to generate an IV
 - **ip_ver** – type of addresses in the outer header (IPv4 or IPv6); configured using the version number (4 or 6);
 - **ip_hdr_size** – size of the outer IP header that will be used to encapsulate the encrypted frames; this must include all IP options
 - **outer_ip_header** – pointer to the buffer containing the IP encapsulation header
 - **outer_udp_header** - pointer to the buffer containing the UDP encapsulation header, used for enabling NAT-T; if this pointer is NULL UDP encapsulation will not be used
 - **post_sec_flow_id** – this value will be attached to this SAs frame queue that comes from the SEC; the upper layer can configure classification based on this value on the post encryption offline port.
 - **dscp_start** – this value represents DSCP range start value. Will be ignored if the DSCP selector is not enabled for this SA
 - **dscp_end** – this value represents DSCP range end value. Will be ignored if the DSCP selector is not enabled for this SA
- **sa_id** - address of an integer variable in which the function will return the SA identifier if it succeeds in offloading the SA; this identifier will be used in further calls to DPA IPSec functions to refer to this particular SA.

Description

This function performs all the necessary steps required to set up a pair of frame queues that can be used to encrypt or decrypt frames according to the parameters of the SA.

For outbound SAs, the calling application must also call the `dpa_ipsec_sa_add_policy` function in order to have a fully functional IPSec hardware accelerated path.

For inbound SAs the function is also responsible for adding an entry in the pre decryption table in order to identify frames that should be decrypted using this SA and direct them to the correct frame queue. A call to this function is sufficient to create a functional inbound hardware accelerated IPSec path, if the use of inbound policy verification is not desired.

In case of error the implementation of this function will attempt to automatically rollback any changes it has performed and free all resources. If this attempt is successful the `sa_id` parameter will contain an invalid ID (e.g. the value -1). If this attempt fails then the `sa_id` parameter will contain the actual ID which was reserved for this SA and it is the responsibility of the

calling application to call `dpa_ipsec_remove_sa` with this ID, in order to release any resources that were allocated by `dpa_ipsec_create_sa`, but could not be released during the automatic rollback process.

In case the SA per DSCP feature needs to be enabled for an outbound SA, `dscp_start` and `dscp_end` should be specified. The values will be used to configure each selector that will have the feature enabled.

When configuring the automatic update of inner header fields or outer header fields, functionalities enabled by setting different bits in the `hdr_upd_flags` field, the following restrictions must be taken into consideration:

- The copying of the entire TOS / DiffServ fields, the automatic decrement of the TTL field and the copying of the DF bit, enabled by setting the `DPA_IPSEC_HDR_COPY_TOS`, the `DPA_IPSEC_HDR_DEC_TTL` and the `DPA_IPSEC_HDR_COPY_DF` flags, in the `hdr_upd_flags` field, can be performed only when using the same type of IP header for the inner and outer headers (i.e. IPv4 and IPv4 or IPv6 and IPv6)
- Propagation of the DSCP and / or ECN fields cannot be configured together with the copying of the TOS / DiffServ fields, but this option is available regardless of the IP version of inner and outer headers.

Return Value

The function will return 0 on success and a negative value otherwise. The returned error codes are the following:

- `-EPERM`, if no DPA IPsec instance is initialized;
- `-EINVAL`, if:
 - invalid function arguments or SA initialization parameters were provided;
 - the calling application requested activation of NAT-T support for an IPv6 SA;
 - the maximum number of inbound SAs that can be offloaded was reached;
 - a bad argument was passed to an internal function;
 - authentication and encryption keys could not be mapped to a job ring device;
 - the size of the calculated key for SA lookup is greater than the maximum key size of the classifier table in which it should be placed;
- `-EDOM`, if:
 - the maximum number of SAs that can be offloaded was reached;
 - an unused flow ID value could not be retrieved from the internal pool;
- `-ERANGE`, a new FQID could not be allocated using the default fqid allocation mechanism;
- `-ENODEV`, if no job ring device could be retrieved for generating the authentication split key;
- `-EFAULT`, if a bad argument was passed to an internal function;
- `-EAGAIN`, if the FMD manipulation object required for implementing variable header length support, could not be initialized;
- `-ENOMEM`, if:
 - there are no more available classifier tables for implementing inbound policy verification;
 - an error occurred during preparation for split key generation;
 - a new FQID could not be allocated from the fqid pool;
- error codes returned by `caam_jr_enqueue` function;
- error codes returned by the `qman_create_fq` and `qman_init_fq` functions;
- error codes returned by the `dpa_classif_table_insert_entry` (used for inserting inbound SA lookup keys, linking indexed table and inbound policy verification table and configuring the default inbound SA action), `dpa_classif_table_get_params` and `dpa_classif_table_modify_miss_action` functions.

7.4.3.3.2.2 Function: *dpa_ipsec_remove_sa*

Syntax

```
int dpa_ipsec_sa_remove(int sa_id);
```

Parameters

- **sa_id** - id of the SA that is to be deleted, as returned by the `dpa_ipsec_sa_create` function.

Description

This function destroys all the objects associated with the SA including all the policies. In case of error, the calling application can reuse the ID to try again to destroy this SA and all associated resources, by calling the function multiple times. It is the responsibility of the calling application to decide when it should give up on trying to release this SA's resources (if all previous calls to this function have failed).

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `-EPERM`, if no DPA IPsec instance is initialized;
- `-EINVAL`, if invalid function arguments were provided;
- `-EAGAIN`, if:
 - a lock on the SA could not be acquired;
 - a lock on the SA's child could not be acquired;
 - one of the policies attached to the SA could not be removed;
- `-EINPROGRESS`, if the SA is a child in the rekeying process;
- `-ENOTRECOVERABLE`, if the SA lookup key cannot be removed from the classifier table;
- `-EFAULT`, if a bad argument was passed to an internal function;
- `-EDOM`, if an error occurred while trying to release the SA's ID;
- `-ETIME`, if the SA is in the rekeying process and a timeout occurred while waiting for the SA's FQ TO SEC to be drained;
- `-EIO`, if the SA is in the rekeying process and the child's SA FQs could not be scheduled;
- `-EUCLEAN`, if the SA is in the rekeying process and its resources could not be freed (FQs);
- `-EDQUOT`, if the SA is in the rekeying process and its resources could not be recycled;
- `-EBUSY`, if a timeout occurred while waiting for the SA's FQs to be drained;
- error codes returned by `dpa_classif_table_delete_entry_by_ref`, if an error occurred during the inbound SA lookup table update process or an error occurred while trying to unlink the policy verification table from the indexed table;
- error codes returned by the `dpa_ipsec_sa_flush_policies`, if an error occurred while trying to remove the policies associated with this SA;
- error codes returned by the `qman_retire_fq` and `qman_oos_fq` functions.

7.4.3.3.2.2.3 Function: *dpa_ipsec_flush_all_sa*

Syntax

```
int dpa_ipsec_flush_all_sa(int dpa_ipsec_id);
```

Parameters

- **dpa_ipsec_id** - identifier of the DPA IPsec instance.

Description

This function is used to remove all the SAs offloaded in the specified DPA IPsec instance. The internal resources are recycled. In case of error, the calling application can call this function multiple times until it is successful or it decides to give up on trying to release the resources associated to the SAs that have not been already destroyed by previous calls.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `-EPERM`, if no DPA IPsec instance is initialized.
- `-EAGAIN`, if one or more SAs could not be removed.

7.4.3.3.2.2.4 Function: *dpa_ipsec_disable_sa*

Syntax

```
int dpa_ipsec_sa_disable(int sa_id);
```

Parameters

- **sa_id** - id of the SA that is to be disabled, as returned by the `dpa_ipsec_sa_create` function

Description

This function prepares a SA for removal. After this function returns the SA can no longer be used, because the hardware components were reconfigured in order to deactivate packet processing using this SA.

The frames that have already been classified and placed in the DPA IPsec internal queues, awaiting processing, will not be dropped and will continue to be processed in the order they arrived until the queues become empty.

The resources associated with this SA, including the internal queues must be, afterwards, destroyed by the calling application using the `dpa_ipsec_remove_sa` function.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `-EINVAL`, if invalid function arguments were provided
- `-EAGAIN`, if a lock on the SA could not be acquired
- `-EINPROGRESS`, if the SA is currently in the rekeying process
- `-ENOTRECOVERABLE`, if the SA lookup key cannot be removed from the classifier table.
- `-EBADSLT`, if not all the policies associated to an outbound SA could be removed.

7.4.3.3.2.2.5 Function: *dpa_ipsec_sa_modify*

Syntax

```

int dpa_ipsec_sa_modify(int sa_id,
    struct dpa_ipsec_sa_modify_prm *modify_prm);

enum dpa_ipsec_sa_modify_type {
    DPA_IPSEC_SA_MODIFY_ARS = 0,
    DPA_IPSEC_SA_MODIFY_SEQ_NUM,
    DPA_IPSEC_SA_MODIFY_EXT_SEQ_NUM,
    DPA_IPSEC_SA_MODIFY_CRYPTO
};

struct dpa_ipsec_sa_modify_prm {
    enum dpa_ipsec_sa_modify_type type;
    union {
        enum dpa_ipsec_arw arw;
        uint64_t seq_num;
        struct dpa_ipsec_sa_crypto_params crypto_params;
    };
};

enum dpa_ipsec_sa_operation_code {
    DPA_IPSEC_SA_MODIFY_ARS_DONE = 0,
    DPA_IPSEC_SA_MODIFY_SEQ_NUM_DONE,
    DPA_IPSEC_SA_MODIFY_EXT_SEQ_NUM_DONE,
    DPA_IPSEC_SA_MODIFY_CRYPTO_DONE,
    DPA_IPSEC_SA_GET_SEQ_NUM_DONE
};

```

Parameters

- **sa_id** - identifier of the SA for which to modify parameters.
- **modify_prm** - used to define what changes must be done to this SA.
 - **type** – type of parameters; configured using the `dpa_ipsec_sa_modify_prm` enum; currently we support only the `DPA_IPSEC_SA_MODIFY_ARS`.
 - **arw** – anti replay type to change to; relevant only if type is `DPA_IPSEC_SA_MODIFY_ARS`
 - **seq_num** – 32 bit or extended sequence number depending on how the SA was created by `dpa_ipsec_create_sa`. Only the least significant word is used for 32 bit SEQ
 - **crypto_params** – change the crypto parameters for this SA to the ones specified by this structure; relevant only if type is `DPA_IPSEC_SA_MODIFY_CRYPTO`

Description

This function can be used to modify the parameters of an already offloaded SA while running.

This function is asynchronous basically when returning with error code 0 means that the operation was triggered but not actually finished. When it is finished the post SEC offline port will enqueue to its error frame queue a frame descriptor having user error status (`0x38402102`) and in the payload the SA id of the security association that has been modified along with the modify operation code. This status is set by the SEC engine when the modify operation is finished and being a user error no other SEC errors have this status.

The upper layer should periodically check the frames from post SEC offline port error frame queue, searching for frames that have SEC user error status (0x38402102). Basically when a frame with this status is received for sure the modify operation was finished. Another check can be done by calling again the `dpa_ipsec_sa_modify` function with the SA id set in the frame descriptor's payload and with the same `modify_prm` parameter. If all well the function should return an `-EALREADY` error code signifying that the SA has been changed to the specified parameters.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `-EINVAL`, if invalid function arguments were provided:
 - invalid function arguments were provided
 - invalid type for modify parameters
 - SA id does not refer to an inbound SA
- `-ENXIO`, could not dma map an address used by SEC engine.
- `-ETXTBSY`, Failed to enqueue frames to the 'TO SEC FQ' of the SA represented by SA id.
- `-EBUSY`, could not acquire the lock for SA structure represented by the SA id.
- `-EALREADY`, the proposed change is already in place for this SA.
- `-EOPNOTSUPP`, Operation not supported, currently we support changes only of `DPA_IPSEC_SA_MODIFY_ARS` type.

7.4.3.3.2.2.6 Function: `dpa_ipsec_sa_get_stats`

Syntax

```
int dpa_ipsec_sa_get_stats(int sa_id, struct dpa_ipsec_sa_stats *sa_stats);

struct dpa_ipsec_sa_stats {
    uint32_t packets_count;
    uint32_t bytes_count;
    uint32_t input_packets;
};
```

Parameters

- **sa_id** - identifier of the SA for which to remove all policies.
- **sa_stats** - pointer to a structure allocated especially to receive the requested statistics:
 - **packets_count** – number of packets processed through this SA
 - **bytes_count** – total number of bytes in all packets that were processed by this SA
 - **input_packets** - the total number of packets (including bad frames) that entered this SA. This statistic information is available only if you have set the `enable_extended_stats` attribute of the SA to `true`. Otherwise this is reported as zero. For an inbound SA this counter translates into the number of packets that were received by this SA, including frames that were later discarded due to decryption errors. For an outbound SA this translates into the total number of packets that were sent on this SA, including packets that failed due to encryption errors.

Description

In case of success the `sa_stats` will be filled with valid statistics information. In order to be able to retrieve statistical information using this function, the `enable_stats` flag must be set when the SA is created. Otherwise statistical information will not be available for the SA and this function will always return an error.

Return Value

The function returns 0 on success or an error code otherwise. The returned error codes are the following:

- `-EINVAL`, if invalid function arguments were provided.
- `-EBUSY`, if a lock on the SA could not be acquired.
- `-EPERM`, if the SA was created without statistics support enabled.

7.4.3.3.2.2.7 Function: `dpa_ipsec_get_sa_context`

Syntax

TBD

Parameters

TBD

Description

This function will be used to retrieve information about the SA that was used to process a packet.

This information can be useful when an error has occurred during IPSec processing and the control application receives the affected packet in a offline port's error frame queue. In this case the control application needs to identify the SA which was used to process that packet.

The function can also be called when inbound policy verification is performed outside the DPA IPSec module (e.g. in software). The module that will perform the inbound policy verification will use the function to retrieve information about what SA was used for decrypting the packet.

Return Value

TBD

7.4.3.3.2.2.8 Function: `dpa_ipsec_sa_get_out_path`

Syntax

```
int dpa_ipsec_sa_get_out_path(int sa_id, uint32_t *fqid);
```

Parameters

- `sa_id` - identifier of the SA for which to retrieve the frame queue.
- `fqid` - identifier of the frame queue to SEC for the specified SA in order to bypass outbound policy verification and apply IPSec processing.

Description

This function will be used to retrieve information which will allow the upper layer to bypass outbound policy lookup and directly apply IPsec processing on a packet using the SA provided as a parameter to the function. In this case it is the responsibility of the calling application to ensure that the correct SA is used to process the packets.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `-EINVAL`, if invalid function arguments were provided.
- `-EBUSY`, if, a lock on the SA could not be acquired.
- `-ENODEV`, if the SA is not used.
- `-EPERM`, if the SA is not an outbound SA.

7.4.3.3.2.2.9 Function: `dpa_ipsec_get_stats`

Syntax

```
int dpa_ipsec_get_stats(int dpa_ipsec_id, struct dpa_ipsec_stats *stats);

struct dpa_ipsec_stats {
    uint32_t inbound_miss_pkts;
    uint32_t inbound_miss_bytes;
    uint32_t outbound_miss_pkts;
    uint32_t outbound_miss_bytes;
};
```

Parameters

- **`dpa_ipsec_id`** - `dpa_ipsec` instance Id returned by a previous call to `dpa_ipsec_init`
- **`stats`** - pointer to a structure allocated especially to receive the global IPsec statistics
 - **`inbound_miss_pkts`** – number of packets that missed the inbound SA lookup
 - **`inbound_miss_bytes`** – total number of bytes in all packets that missed the inbound SA lookup
 - **`outbound_miss_pkts`** - number of packets that missed the outbound policy lookup
 - **`outbound_miss_bytes`** - total number of bytes in all packets that missed the outbound policy lookup

Description

This function returns the IPsec global statistics for the specified IPsec instance. In case of success the `stats` will be populated with valid statistics information.

Return Value

The function returns 0 on success or an error code otherwise. The returned error codes are the following:

- `-EINVAL`, if invalid function arguments were provided.

7.4.3.3.2.3 Add and Remove Policies

Table 132. API to Add and Remove Policies

Function Name	Description
<code>dpa_ipsec_sa_add_policy</code>	Adds a new policy (for inbound or outbound) policy verification. It adds keys and directs packets through a certain flow (SA).
<code>dpa_ipsec_sa_remove_policy</code>	Removes a previously added policy.
<code>dpa_ipsec_sa_flush_policies</code>	Removes all policies associated to a SA.
<code>dpa_ipsec_sa_get_policies</code>	Retrieves all the policies linked to a specified SA.

7.4.3.3.2.3.1 Function: `dpa_ipsec_sa_add_policy`

Syntax

```

int dpa_ipsec_sa_add_policy(int sa_id,
                           struct dpa_ipsec_policy_params *policy_params);

struct dpa_ipsec_l4_params {
    uint16_t    src_port;
    uint16_t    src_port_mask;
    uint16_t    dest_port;
    uint16_t    dest_port_mask;
};

struct dpa_ipsec_icmp_params {
    uint8_t     icmp_type;
    uint8_t     icmp_type_mask;
    uint8_t     icmp_code;
    uint8_t     icmp_code_mask;
};

enum dpa_ipsec_df_action {
    DPA_IPSEC_DF_ACTION_DISCARD = 0,
    DPA_IPSEC_DF_ACTION_OVERRIDE,
    DPA_IPSEC_DF_ACTION_CONTINUE
};

enum dpa_ipsec_pol_dir_params_type {
    DPA_IPSEC_POL_DIR_PARAMS_NONE = 0,
    DPA_IPSEC_POL_DIR_PARAMS_MANIP,
    DPA_IPSEC_POL_DIR_PARAMS_ACT,
};

struct dpa_ipsec_pol_dir_params {
    enum dpa_ipsec_pol_dir_params_type type;
    union {
        struct    int manip_desc;
        struct dpa_cls_tbl_action in_action;
    };
};

```

```
};

struct dpa_ipsec_policy_params {
    struct dpa_ipsec_ip_address  src_addr;
    uint8_t                    src_prefix_len;
    struct dpa_ipsec_ip_address  dest_addr;
    uint8_t                    dest_prefix_len;
    uint8_t                    protocol;
    bool                       masked_proto;
    bool                       use_dscp;
    union {
        struct dpa_ipsec_l4_params l4;
    };
    struct dpa_ipsec_icmp_params icmp;
    struct dpa_ipsec_pol_dir_params dir_params;
    int                       priority;
};
```

Parameters

- **sa_id** - id of the SA that this policy is linked
- **policy_params** - structure containing the selectors for the offloaded policy:
 - **src_addr** – source IP address
 - **src_prefix_len** - the length of the mask to be applied on the source IP address
 - **dest_addr** – destination IP address
 - **dest_prefix_len** - the length of the mask to be applied on the destination IP address
 - **protocol** - the expected value in the IP header protocol field
 - **masked_proto** – enable / disable masking of the protocol field in the policy key;
 - **use_dscp** – enable / disable DSCP value in policy selector;
 - **l4** – structure containing parameters for layer 4 type of policies:
 - **src_port** – source port value; relevant only for selectors for L4 protocols
 - **src_port_mask** – mask to be applied on the source port value;
 - **dest_port** – destination port value; relevant only for selectors for L4 protocols
 - **dest_port_mask** – mask to be applied on the destination port value;
 - **icmp** – structure containing parameters for icmp policies:
 - **icmp_type** – the type field in the ICMP header; relevant only if protocol field is of type ICMP
 - **icmp_type_mask** – mask to be applied on the ICMP type field value;
 - **icmp_code** – the code field in the ICMP header; relevant only if protocol field is of type ICMP
 - **icmp_code_mask** – mask to be applied on the ICMP code field value;
 - **dir_params** – direction specific parameter for the security policy:
 - **type** – type of parameters; configured using the `dpa_ipsec_pol_dir_params_type` enum; if equal to `DPA_IPSEC_POL_DIR_PARAMS_NONE`, the other fields' values are ignored;
 - **manip_desc** – fragmentation descriptor or header manipulation chain descriptor for policies attached to outbound SAs; relevant only if type is equal to `DPA_IPSEC_POL_DIR_PARAMS_MANIP`;

- **in_action** – the action that should be performed if the decrypted frames match the policy key; if this field is NULL, the frames that pass inbound policy verification will be handled according to the default SA action configured at SA creation time; if this field is not NULL, it will override, for this policy only, the SA default action; this field is relevant only for policies attached to inbound SAs and if the type field is equal to `DPA_IPSEC_POL_DIR_PARAMS_ACT`;
- **priority** – the priority of this policy.

Description

This function can be used to configure the policies for a SA.

For outbound paths, the use of this function is mandatory, because the policy selectors passed as parameters to it will be used to classify frames and identify the correct SA that should be used for encryption.

For inbound paths the use of this function is mandatory if inbound policy verification was activated for this DPA IPsec instance and is prohibited if it was not activated. In case the inbound policy verification was activated, the policy selectors received as parameters will be used to check if the decrypted frames arrived on the correct SA.

In case the SA per DSCP feature needs to be enabled for an outbound SA selector, `use_dscp` must be set as true. In this case a number of selectors will be created according to DSCP range specified for the SA.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `EINVAL`, if:
 - invalid function arguments were provided
 - the parameters of the classifier table could not be retrieved
 - no classifier table was configured for this type of policy
- `EFAULT`, if a bad argument was passed to a internal function.
- `EBUSY`, if a lock on the SA could not be acquired.
- `EPERM`, if:
 - the SA is a parent in the rekeying process and is of type outbound
 - the SA is a child in the rekeying process and is of type inbound
- `ENOMEM`, if memory could not be allocated for the new policy.
- `EAGAIN`, if the FMD header manipulation object for fragmentation could not be created.
- error codes returned by the `dpa_classif_table_insert_entry` (used for inserting policy keys).

7.4.3.3.2.3.2 Function: `dpa_ipsec_sa_remove_policy`

Syntax

```
int dpa_ipsec_sa_remove_policy(int sa_id,
                               struct dpa_ipsec_policy_params *policy_params);
```

Parameters

- **sa_id** - id of the SA that this policy is linked.

- **policy_params** - structure with policy selectors, used for identify the policy that must be deleted; for detailed information regarding the policy selectors please see the description of the parameters for `dpa_ipsec_sa_add_policy` function

Description

This function is used to remove a previously offloaded policy. The policy to be removed is identified by a policy parameters structure that is identical to the one that was passed in the `dpa_ipsec_sa_add_policy` function when the policy was offloaded.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `EINVAL`, if invalid function arguments were provided.
- `EBUSY`, if:
 - a lock on the SA could not be acquired
 - the FMD manipulation object used for fragmentation could not be removed
- `EFAULT`, if a bad argument was passed to a internal function
- `EPERM`, if:
 - the SA is a parent in the rekeying process and is of type outbound
 - the SA is a child in the rekeying process and is of type inbound
- `EDOM`, if a corresponding policy was not found in the SA's list of associated policies.
- error codes returned by the `dpa_classif_table_delete_entry_by_ref` (used for removing policy keys).

7.4.3.3.2.3.3 Function: `dpa_ipsec_sa_flush_policies`

Syntax

```
int dpa_ipsec_sa_flush_policies(int sa_id);
```

Parameters

- **sa_id** - identifier of the SA for which to remove all policies..

Description

This function is used to remove all the policies for the given SA. The internal resources are recycled.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `EINVAL`, if invalid function arguments were provided.
- `EBUSY`, if, a lock on the SA could not be acquired.
- `EFAULT`, if a bad argument was passed to a internal function.
- `EAGAIN`, if one or more policies could not be removed.
- error codes returned by the `dpa_classif_table_delete_entry_by_ref` (used for removing policy keys).

7.4.3.3.2.3.4 Function: *dpa_ipsec_sa_get_policies*

Syntax

```
int    dpa_ipsec_sa_get_policies(int sa_id,
                                struct dpa_ipsec_policy_params *policy_params,
                                int    *num_pol);
```

Parameters

- **sa_id** - identifier of the SA for which to retrieve all policies.
- **policy_params** - placeholder for the array of policy structures associated to the specified SA; the array of policy parameters will be allocated internally by this function.
- **num_pol** - number of policy parameters structures in the returned policy array.

Description

This function is used to retrieve all the policies associated to a given SA. The array that will hold the retrieved policies must be allocated by the calling application.

In order to determine the correct size for this array, the calling application must first call the function with the `policy_params` parameter set to NULL. In this case the function will return in the `num_pol` parameter the number of policies that are linked to the specified SA. Then the calling application can allocated a policy array in which there is enough space for all the policies linked to the SA. After calling the function again, this time passing the address of the newly allocated array as the `policy_params` parameter, it will receive the information about all the policies linked to the specified SA.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `EINVAL`, if invalid function arguments were provided.
- `EBUSY`, if, a lock on the SA could not be acquired.
- `EFAULT`, if a bad argument was passed to a internal function.
- `EAGAIN`, if there are more policies in the SA list than the maximum number of policies that can be copied in the provided buffer.

7.4.3.3.2.4 Rekeying

Table 133. API for IPsec Rekeying

Function Name	Description
<code>dpa_ipsec_sa_rekeying</code>	Performs all the required actions for changing and expired SA with a newly negotiated SA, while ensuring no packets are lost during this process.

7.4.3.3.2.4.1 Function: *dpa_ipsec_sa_get_rekeyings*

Syntax

```
int      dpa_ipsec_sa_rekeying(int sa_id,
                               struct dpa_ipsec_sa_params *sa_params,
                               dpa_ipsec_rekey_event_cb *rekey_event_cb,
                               bool auto_rmv_old_sa, int *new_sa_id);

typedef int (*dpa_ipsec_rekey_event_cb)
(int dpa_ipsec_id,
    int sa_id,
    int error);
```

Parameters

- **sa_id** - id of the SA that has expired and needs to be replaced.
- **sa_params** - structure containing the parameters of the new SA; for a detailed description of these parameters see the description of `dpa_ipsec_create_sa`;
- **rekey_event_c** - callback used to report the status of the rekeying process, when it is completed; this callback is only supported in Kernel space
 - **dpa_ipsec_id** – id of the IPsec instance in which the SA rekeying process is performed;
 - **sa_id** – id of the SA for which the rekeying process was started;
 - **error** – status of the rekeying process;
- **auto_rmv_old_sa** - valid only for inbound rekeying. Auto remove old SA when encrypted traffic starts flowing on the new SA.
- **new_sa_id a** - address of an integer variable where the id of the newly created SA will be stored.

Description

This function is used to replace an expired SA with a new valid SA.

The rekeying procedure was designed in such a way that there is no service interruption, no packets are lost and the links between the policies and the old SA are automatically transferred to the new SA. Furthermore the rekeying process ensures that the packet order is preserved.

The `auto_rmv_old_sa` option has meaning only when rekeying an inbound SA as follows:

- If TRUE the DPA IPSEC will create the new SA and remove the old SA when it has received a frame encrypted with the new SA. This is done disregarding that the SA might be in its lifetime period.
- If FALSE the DPA IPSEC will create the new SA and let the old SA in place. Practically the two SAs will run in parallel. It is the responsibility of the user to call `dpa_ipsec_remove_sa` when the old SA expires (hardlimit reached). This option may be used when the user wants to receive frames encrypted with old SA which were reordered by the network and arrive after the new SA is already in use.

This function is the only one that uses an asynchronous error reporting mechanism. If an error occurs during the process of replacing the old SA with the new SA, which is an asynchronous process, the function will use the `dpa_ipsec_rekey_event_cb` to report the error to the calling application. If no error occurred during the rekeying process, the same callback will be used to signal successful completion of this task (i.e. error = 0).

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- -EINVAL, if:
 - no DPA IPsec instance is initialized
 - invalid function arguments or SA initialization parameters were provided
 - the calling application requested activation of NAT-T support for an IPv6 SA
 - authentication and encryption keys could not be mapped to a job ring device
 - the size of the calculated key for SA lookup is greater than the maximum key size of the classifier table in which it should be placed
- -EBUSY, if a lock on the SA could not be acquired.
- -EEXIST, if a bad argument was passed to a internal function.
- -EDOM, if a bad argument was passed to a internal function.
- -ERANGE, if a bad argument was passed to a internal function.
- -ENODEV, if a bad argument was passed to a internal function.
- -EFAULT, if a bad argument was passed to a internal function.
- -ENOMEM, if a bad argument was passed to a internal function.
 - an error occurred during preparation for split key generation
 - a new FQID could not be allocated from the fqid pool
- -EAGAIN, if there are more policies in the SA list than the maximum number of policies that can be copied in the provided buffer.
- error codes returned by `caam_jr_enqueue` function.
- error codes returned by the `qman_create_fq` and `qman_init_fq` functions.
- error codes returned by the `dpa_classif_table_insert_entry` (used for inserting inbound SA lookup keys) and `dpa_classif_table_get_params` functions
- -EUSERS, if the function `dpa_classif_table_modify_entry_by_ref`, used for modifying the actions associated to the outbound security policies, failed and only some of the security policies were transferred to the new SA
- error codes returned by the `qman_schedule_fq` function

The following error codes can be returned in the error parameter of the `dpa_ipsec_rekey_event_cb` callback:

- for outbound SAs:
 - -EUCLEAN, if during the rekeying process the removal of the TO SEC FQ of old SA failed. The upper layer has to call the `dpa_ipsec_remove_sa` at a later time (not from callback) to try again to free the old SA resources. The new SA is active.
 - -EDQUOT, if recycling of old SA's memory resources failed. The upper layer has to call the `dpa_ipsec_remove_sa` function at a later time (not from callback) to try again to recycle these resources.
- for inbound SAs:
 - -ENOTRECOVERABLE, if the removal of the old SA's table entry in in the SA lookup classifier table has failed. If this error is received the upper layer must take into consideration that the system is in a vulnerable state, because it will still accept packets that match the old SA's parameters, and it should consider rebooting the system.
 - -EUCLEAN, if during the rekeying process the removal of the TO SEC FQ of old SA failed. The upper layer has to call the `dpa_ipsec_remove_sa` at a later time (not from callback) to try again to free the old SA resources. The new SA is active.
 - -EDQUOT, if recycling of old SA's memory resources failed. The upper layer has to call the `dpa_ipsec_remove_sa` function at a later time (not from callback) to try again to recycle these resources.

7.4.3.4 DPA Statistics

The DPA Stats module exports a set of functions that can be used to:

- initialize the DPA Stats module;
- create a DPA Stats single counter;
- create a DPA Stats class counter;
- retrieve the values for a series of counters.

7.4.3.4.1 Initialization

Initialization is a mandatory step in using the DPA Stats component. The initialization function has the purpose of creating and initializing internal data structures that will be further used at runtime. The allocated internal structures will be later

Counter creation

Counter creation is an important step, as the counter is the basic element used to retrieve specific information. There are two types of counters that can be created: a single counter and a class counter.

A **single counter** is meant to retrieve information from one source, which can be either a hardware resource or a software resource.

A **class counter** has the purpose of retrieving information from multiple sources of the same type.

Counters are of different types; as such counter creation is responsible of configuring the interface with different hardware blocks or software components in order to be able to obtain the corresponding values. During counter creation no hardware block or software component is initialized, that is the responsibility of the user software. The counter relies on an already initialized hardware or software component and will only configure the mechanism it needs to gain the desired information. It is also the user responsibility to insure synchronization between the initialized hardware and software components and the created counters. As such a counter needs to be removed if the underlying hardware or software component was destroyed.

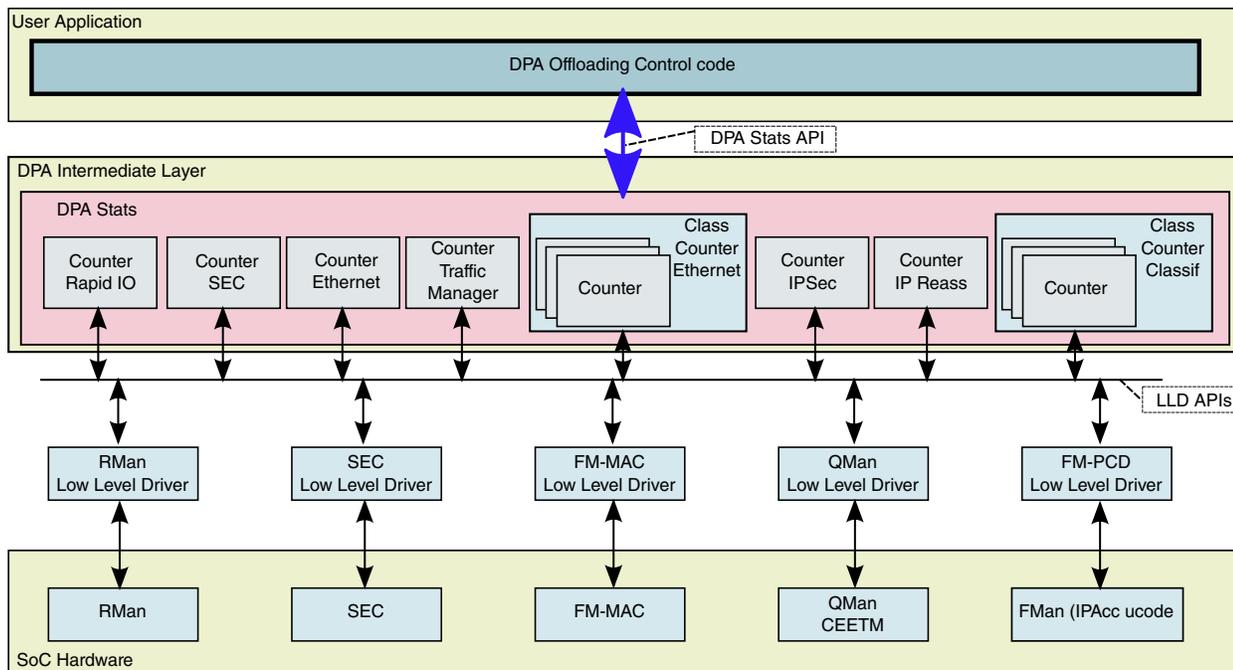


Figure 140. DPA Stats architecture

Counter retrieve

The user has the possibility of retrieving the values for one or multiple counters. For each counter, the retrieve operation means a direct interaction with the hardware or software component. As such, the response time of the retrieve operation depends on the number of counters the request is made of and the underlying hardware or software component response time.

The result of a retrieve operation is written by the DPA Stats component in a memory area provided by the user application and the user application will be informed on the number of bytes written either when the calling function returns or via a callback function.

7.4.3.4.2 DPA Statistics API

There are several selections for the DPA Statistics API:

7.4.3.4.2.1 Initialize and Free Statistics

Table 134. API to Initialize and Free Statistics

Function Name	Description	Input Parameters	Output Parameters
<code>dpa_stats_init</code>	Configures and initializes a DPA Stats instance.	<ul style="list-style-type: none"> • maximum number of counters that will be managed by the DPA Stats instance. • pointer to the memory area where the values of the counters will be written by the DPA Stats instance. • length of the memory area expressed in number of bytes. 	Identifier of the newly created DPA Stats instance.
<code>dpa_stats_free</code>	Releases all resources associated with a DPA Stats instance and destroys it.	<ul style="list-style-type: none"> • identifier of the DPA Stats instance to destroy. 	None.

7.4.3.4.2.1.1 Function: `dpa_stats_init`

Syntax

```
int dpa_stats_init(const struct dpa_stats_params *params, int
                  *dpa_stats_id);

struct dpa_stats_params
{
  unsigned int max_counters;
  void *storage_area;
  unsigned int storage_area_len;
};
```

Parameters

- **params** - data structure containing the parameters needed to initialize the DPA Stats;
 - **max_counters**: maximum number of counters supported by the DPA Stats module.
 - **storage_area**: pointer to the memory area where the values of the counters will be written by the DPA Stats instance after a retrieve operation was successful.
 - **storage_area_len**: length of the memory area expressed in number of bytes.
- **dpa_stats_id** - a location where the function will return the identifier of the DPA Stats instance, in case of success; this id will be further used to create counters that are attached to the DPA Stats instance.

Description

This function is used to initialize a DPA Stats instance. The purpose of the function is to allocate and initialize all the internal structures that will be further needed by the DPA Stats instance.

The storage area can be a pointer to a memory allocated using C programming standard library routines or a pointer to a memory area allocated using NXP's "shmem" application programming interface. Memory allocated using "shmem" is a contiguous physical memory that is accessible both in user-space application and in kernel-space driver implementation through corresponding user-space and kernel space virtual addresses. This type of memory allows the DPA Stats driver to write the statistics values in the storage area using a "zero copy" approach and is an optimized solution to transfer information from kernel-space to user-space.

Memory area allocated in user-space using C programming routines is not mapped in kernel-space so the transfer between kernel-space DPA Stats driver and user-space application is performed using memory copy from kernel-to-user space and vice-versa.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- **EINVAL**, if one of the function arguments or parameters was incorrect;
- **ENOMEM**, if there is no more memory to create an internal data structure needed by the DPA Stats module.
- **EPERM**, if the component is already initialized.
- **ENOSPC**, if the operation of creating a single threaded workqueue failed.
- **EDOM**, if the number of provided counters is above the maximum allowed.

7.4.3.4.2.1.2 Function: *dpa_stats_free*

Syntax

```
int dpa_stats_free(int dpa_stats_id);
```

Parameters

- **id** - identifier of the DPA Stats instance to destroy.

Description

Releases all resources associated with a DPA instance and destroys the instance.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `EINVAL`, if the provided instance identifier is not valid;
- `EDOM`, if the provided counter identifier could not be released.

7.4.3.4.2.2 Create, Remove or Modify Counters

Table 135. API to Create Remove or Modify Counters

Function Name	Description	Input Parameters	Output Parameters
<code>dpa_stats_create_counter</code>	Configures and initializes a DPA Stats counter. The counter will be used to address a single hardware or software resource.	<ul style="list-style-type: none"> • identifier of the DPA Stats instance the counter will belong to. • the type of counter that needs to be created • parameters needed to create the counter, depending of the counter type • pointer to a location where the function will return an unique identifier for the counter in case of success. 	Identifier for the counter. The returned identifier will be further used to address the counter.
<code>dpa_stats_create_class_counter</code>	Configures and initializes a DPA Stats counter. The counter will be used to address multiple hardware or software resources of the same time.	<ul style="list-style-type: none"> • identifier of the DPA Stats instance the class counter will belong to. • the type of class counter that needs to be created. • parameters needed to create the class counter, depending of the class counter type. • pointer to a location where the function will return an unique identifier for the class counter in case of success. 	Identifier for the class counter in case of success. The returned identifier will be further used to address the class counter.

Table continues on the next page...

Table 135. API to Create Remove or Modify Counters (continued)

Function Name	Description	Input Parameters	Output Parameters
<code>dpa_stats_modify_class_counter</code>	<p>Modifies a member of a certain class counter.</p> <p>The class member can be invalidated or updated, by providing a new source.</p>	<ul style="list-style-type: none"> • identifier of the DPA Stats class counter for which the member will be updated. • the type of the member that needs to be modified. • parameters needed to modify the class member. • the index of the member in the class counter. 	None.
<code>dpa_stats_remove_counter</code>	<p>Removes a single counter or a class counter. The resources occupied by the counter are marked as empty.</p>	<ul style="list-style-type: none"> • identifier of the DPA Stats counter or class counter that will be removed. 	None.

7.4.3.4.2.2.1 Function: `dpa_stats_create_counter`

Syntax

```
int dpa_stats_create_counter(int dpa_stats_id,
    const struct dpa_stats_cnt_params *params,int *dpa_stats_cnt_id);

struct dpa_stats_cnt_params
{
    enum dpa_stats_cnt_type type;
    union
    {
        struct dpa_stats_cnt_eth    eth_params;
        struct dpa_stats_cnt_reass  reass_params;
        struct dpa_stats_cnt_frag   frag_params;
        struct dpa_stats_cnt_plcr   plcr_params;
        struct dpa_stats_cnt_classif_tbl  classif_tbl_params;
        struct dpa_stats_cnt_classif_node  classif_node_params;
        struct dpa_stats_cnt_ipsec   ipsec_params;
        struct dpa_stats_cnt_traffic_mng  traffic_mng_params;
    };
};

enum dpa_stats_cnt_type
{
    DPA_STATS_CNT_ETH = 0,
    DPA_STATS_CNT_REASS,
    DPA_STATS_CNT_FRAG,
    DPA_STATS_CNT_POLICER,
    DPA_STATS_CNT_CLASSIF_TBL,
    DPA_STATS_CNT_CLASSIF_NODE,
    DPA_STATS_CNT_CLASSIF,
    DPA_STATS_CNT_IPSEC,
    DPA_STATS_CNT_TRAFFIC_MNG,
};
```

```

DPA_STATS_CNT_RAPIDIO
};

enum dpa_stats_cnt_sel
{
  DPA_STATS_CNT_NUM_OF_BYTES = 0,
  DPA_STATS_CNT_NUM_OF_PACKETS,
  DPA_STATS_CNT_NUM_ALL
};

enum dpa_stats_cnt_eth_sel
{
  DPA_STATS_CNT_ETH_DROP_PKTS      = 0x00000001,
  DPA_STATS_CNT_ETH_BYTES          = 0x00000002,
  DPA_STATS_CNT_ETH_PKTS           = 0x00000004,
  DPA_STATS_CNT_ETH_BC_PKTS        = 0x00000008,
  DPA_STATS_CNT_ETH_MC_PKTS        = 0x00000010,
  DPA_STATS_CNT_ETH_CRC_ALIGN_ERR   = 0x00000020,
  DPA_STATS_CNT_ETH_UNDERSIZE_PKTS  = 0x00000040,
  DPA_STATS_CNT_ETH_OVERSIZE_PKTS  = 0x00000080,
  DPA_STATS_CNT_ETH_FRAGMENTS      = 0x00000100,
  DPA_STATS_CNT_ETH_JABBERS        = 0x00000200,
  DPA_STATS_CNT_ETH_64BYTE_PKTS    = 0x00000400,
  DPA_STATS_CNT_ETH_65_127BYTE_PKTS = 0x00000800,
  DPA_STATS_CNT_ETH_128_255BYTE_PKTS = 0x00001000,
  DPA_STATS_CNT_ETH_256_511BYTE_PKTS = 0x00002000,
  DPA_STATS_CNT_ETH_512_1023BYTE_PKTS = 0x00004000,
  DPA_STATS_CNT_ETH_1024_1518BYTE_PKTS = 0x00008000,
  DPA_STATS_CNT_ETH_OUT_PKTS       = 0x00010000,
  DPA_STATS_CNT_ETH_OUT_DROP_PKTS  = 0x00020000,
  DPA_STATS_CNT_ETH_OUT_BYTES      = 0x00040000,
  DPA_STATS_CNT_ETH_IN_ERRORS      = 0x00080000,
  DPA_STATS_CNT_ETH_OUT_ERRORS     = 0x00100000,
  DPA_STATS_CNT_ETH_IN_UNICAST_PKTS = 0x00200000, DPA_STATS_CNT_ETH_OUT_UNICAST_PKTS =
0x00400000,
  DPA_STATS_CNT_ETH_ALL            = 0x00800000
};

enum dpa_stats_cnt_eth_id
{
  DPA_STATS_ETH_1G_PORT0 = 0,
  DPA_STATS_ETH_1G_PORT1,
  DPA_STATS_ETH_1G_PORT2,
  DPA_STATS_ETH_1G_PORT3,
  DPA_STATS_ETH_1G_PORT4,
  DPA_STATS_ETH_10G_PORT0
};

struct dpa_stats_cnt_eth_src
{
  uint8_t      engine_id;
  enum dpa_stats_cnt_eth_id eth_id;
};

struct dpa_stats_cnt_eth
{
  struct dpa_stats_cnt_eth_src src;
  unsigned int cnt_sel;
};

```

```
enum dpa_stats_cnt_reass_gen_sel
{
    DPA_STATS_CNT_REASS_TIMEOUT          = 0x00000001,
    DPA_STATS_CNT_REASS_RFD_POOL_BUSY    = 0x00000002,
    DPA_STATS_CNT_REASS_INT_BUFF_BUSY    = 0x00000004,
    DPA_STATS_CNT_REASS_EXT_BUFF_BUSY    = 0x00000008,
    DPA_STATS_CNT_REASS_SG_FRAGS        = 0x00000010,
    DPA_STATS_CNT_REASS_DMA_SEM         = 0x00000020,
    DPA_STATS_CNT_REASS_NON_CONSISTENT_SP = 0x00000040,
    DPA_STATS_CNT_REASS_GEN_ALL         = 0x00000080
};

enum dpa_stats_cnt_reass_ipv4_sel
{
    DPA_STATS_CNT_REASS_IPv4_FRAMES     = 0x00000100,
    DPA_STATS_CNT_REASS_IPv4_FRAGS_VALID = 0x00000200,
    DPA_STATS_CNT_REASS_IPv4_FRAGS_TOTAL = 0x00000400,
    DPA_STATS_CNT_REASS_IPv4_FRAGS_MALFORMED = 0x00000800,
    DPA_STATS_CNT_REASS_IPv4_FRAGS_DISCARDED = 0x00001000,
    DPA_STATS_CNT_REASS_IPv4_AUTOLEARN_BUSY = 0x00002000,
    DPA_STATS_CNT_REASS_IPv4_EXCEED_16FRAGS = 0x00004000,
    DPA_STATS_CNT_REASS_IPv4_ALL       = 0x00008000
};

enum dpa_stats_cnt_reass_ipv6_sel
{
    DPA_STATS_CNT_REASS_IPv6_FRAMES     = 0x00010000,
    DPA_STATS_CNT_REASS_IPv6_FRAGS_VALID = 0x00020000,
    DPA_STATS_CNT_REASS_IPv6_FRAGS_TOTAL = 0x00040000,
    DPA_STATS_CNT_REASS_IPv6_FRAGS_MALFORMED = 0x00080000,
    DPA_STATS_CNT_REASS_IPv6_FRAGS_DISCARDED = 0x00100000,
    DPA_STATS_CNT_REASS_IPv6_AUTOLEARN_BUSY = 0x00200000,
    DPA_STATS_CNT_REASS_IPv6_EXCEED_16FRAGS = 0x00400000,
    DPA_STATS_CNT_REASS_IPv6_ALL       = 0x00800000
};

struct dpa_stats_cnt_reass
{
    void *reass;
    unsigned int cnt_sel;
};

enum dpa_stats_cnt_frag_sel
{
    DPA_STATS_CNT_FRAG_TOTAL_FRAMES     = 0x00000001,
    DPA_STATS_CNT_FRAG_FRAMES          = 0x00000002,
    DPA_STATS_CNT_FRAG_GEN_FRAGS       = 0x00000004,
    DPA_STATS_CNT_FRAG_ALL             = 0x00000008
};

struct dpa_stats_cnt_frag
{
    void *frag;
    unsigned int cnt_sel;
};

enum dpa_stats_cn_plcr_sel
{
    DPA_STATS_CNT_PLCR_GREEN_PKTS      = 0x00000001,
    DPA_STATS_CNT_PLCR_YELLOW_PKTS     = 0x00000002,
```

```
DPA_STATS_CNT_PLCR_RED_PKTS      = 0x00000004,  
DPA_STATS_CNT_PLCR_RECOLOR_YELLOW_PKTS = 0x00000008,  
DPA_STATS_CNT_PLCR_RECOLOR_RED_PKTS = 0x00000010,  
DPA_STATS_CNT_PLCR_ALL          = 0x00000020  
};  
  
struct dpa_stats_cnt_plcr  
{  
    void *plcr;  
    unsigned int cnt_sel;  
};  
  
enum dpa_stats_cnt_classif_sel {  
    DPA_STATS_CNT_CLASSIF_BYTES      = 0x00000010,  
    DPA_STATS_CNT_CLASSIF_PACKETS   = 0x00000020,  
    DPA_STATS_CNT_CLASSIF_RMON_RANGE1 = 0x00000040,  
    DPA_STATS_CNT_CLASSIF_RMON_RANGE2 = 0x00000080,  
    DPA_STATS_CNT_CLASSIF_RMON_RANGE3 = 0x00000100,  
    DPA_STATS_CNT_CLASSIF_RMON_RANGE4 = 0x00000200,  
    DPA_STATS_CNT_CLASSIF_RMON_RANGE5 = 0x00000400,  
    DPA_STATS_CNT_CLASSIF_RMON_RANGE6 = 0x00000800,  
    DPA_STATS_CNT_CLASSIF_RMON_RANGE7 = 0x00001000,  
    DPA_STATS_CNT_CLASSIF_RMON_RANGE8 = 0x00002000,  
    DPA_STATS_CNT_CLASSIF_RMON_RANGE9 = 0x00004000,  
    DPA_STATS_CNT_CLASSIF_RMON_RANGE10 = 0x00008000,  
};  
  
struct dpa_stats_cnt_classif_tbl  
{  
    int      td;  
    const struct dpa_offload_lookup_key *key;  
    unsigned int      cnt_sel;  
};  
  
enum dpa_stats_classif_node_type  
{  
    DPA_STATS_CLASSIF_NODE_HASH = 0,  
    DPA_STATS_CLASSIF_NODE_INDEXED,  
    DPA_STATS_CLASSIF_NODE_EXACT_MATCH  
};  
  
struct dpa_stats_cnt_classif_node  
{  
    void      *cc_node;  
    enum dpa_stats_classif_node_type ccnode_type;  
    struct dpa_offload_lookup_key *key;  
    unsigned int      cnt_sel;  
};  
  
struct dpa_stats_cnt_ipsec  
{  
    int      sa_id;  
    enum dpa_stats_cnt_sel      cnt_sel;  
};  
  
enum dpa_stats_cnt_traffic_mng_src  
{  
    DPA_STATS_CNT_TRAFFIC_CLASS = 0,  
    DPA_STATS_CNT_TRAFFIC_CG  
};
```

```
struct dpa_stats_cnt_traffic_mng
{
    enum dpa_stats_cnt_traffic_mng_src  src;
    void          *traffic_mng;
    enum dpa_stats_cnt_sel      cnt_sel;
};
```

Parameters

- **dpa_stats_id** - identifier of the DPA Stats instance the counter belongs to.
- **params** - structure holding the parameters needed to create a counter:
 - **type**: the type of the counter that needs to be created: Ethernet counter, Reassembly counter, Fragmentation Counter, Policer Counter, Classifier Table counter, Classification Node Counter, IPSec counter, Traffic Manager counter or RapidIO counter (selected from `dpa_stats_cnt_type` enum).
 - **eth_params**: structure holding the parameters needed to create an Ethernet counter
 - **src**: structure holding the parameters needed to identify the source of the Ethernet counter
 - **engine_id**: identifier of the engine the Ethernet interface belongs to
 - **eth_id**: identifier of the Ethernet interface, identifier that is relative to the engine the interface belongs to
 - **cnt_sel**: selection from Ethernet available counters: enum `dpa_stats_cnt_eth_sel`
 - **reass_params**: structure holding the parameters needed to create an IP Reassembly counter
 - **reass**: handle of IP Reassembly object
 - **cnt_sel**: selection from Reassembly available counters, enum `dpa_stats_cnt_reass_gen_sel`, `dpa_stats_cnt_reass_ipv4_sel` or `dpa_stats_cnt_reass_ipv6_sel`
 - **frag_params**: structure holding the parameters needed to create an IP Fragmentation counter
 - **frag**: handle of IP Fragmentation object
 - **cnt_sel**: selection from Fragmentation available counters, enum `dpa_stats_cnt_frag_sel`
 - **plcr_params**: structure holding the parameters needed to create a Policer counter
 - **plcr**: handle of Policer object
 - **cnt_sel**: selection from Policer available counters, enum `dpa_stats_cn_plcr_sel`
 - **classif_tbl_params**: structure holding the parameters needed to create a Classifier Table counter
 - **td**: descriptor of the table used to perform the Classification
 - **key**: pointer to the key descriptor. If *key* is NULL, DPA Stats will provide the table *miss* condition statistics in this counter.
 - **byte**: pointer to an array of bytes representing the key
 - **mask**: the mask to store for this key
 - **size**: the size of the key
 - **cnt_sel**: selection from Classifier Table available counters, enum `dpa_stats_cnt_classif_sel` or enum `dpa_stats_cnt_frag_sel`
 - **classif_node_params**: structure holding the parameters needed to create a Classification Node counter
 - **cc_node**: handle of the Cc node used to perform the Classification
 - **ccnode_type**: the type of FMAN Classification Node

- **key**: pointer to the key descriptor. If *key* is NULL, DPA Stats will provide the CC node's *miss* condition statistics in this counter.
 - **byte**: pointer to an array of bytes representing the key
 - **mask**: the mask to store for this key
 - **size**: the size of the key
- **cnt_sel**: selection from Classification Node available counters, enum `dpa_stats_cnt_classif_sel`
- **ipsec_params**: structure holding the parameters needed to create an IPSec counter
 - **sa_id**: identifier of the Security Association
 - **cnt_sel**: single selection from IPSec available counters, enum `dpa_stats_cnt_sel`
- **traffic_mng_params**: structure holding the parameters needed to create a Traffic Manager counter
 - **src**: the type of the source used for the Traffic Manager counter
 - **traffic_mng**: depending on the Traffic Manager source, the 'traffic_mng' has a different meaning: it represents a pointer to a structure of type 'qm_ceetm_cq' in case the traffic source is a "Class Queue" or a pointer to a structure of type 'qm_ceetm_ccg' in case the traffic source is a "Class Congestion Group"
 - **cnt_sel**: single selection from Traffic Manager counter, `dpa_stats_cnt_sel` enum
- **rapidio_params**: structure holding the parameters needed to create a RapidIO counter
- **dpa_stats_cnt_id** - address of an integer variable in which the function will return the counter identifier if it succeeds in creating it; this identifier will be used in further calls to DPA Stats functions to refer to this particular counter.

Description

This function performs all the necessary steps required to create and initialize a single counter. A single counter means a counter that is used to retrieve information from one source, source that can be either a hardware resource or a software resource.

During counter creation no memory is allocated, but instead the function uses internal structures that were allocated during DPA Stats instance initialization. On success, the function returns a unique counter identifier that should be further used on calls that refer to that counter.

The first parameter of the structure used to create a counter is the type of the counter. Depending on the type of the counter, the user needs to provide the parameters necessary for that type of counter.

For the Ethernet counter, the user can select any combinations of counters from the enumeration `dpa_stats_cnt_eth_sel`.

For the Reassembly counter, the user can have combination of selections from the three groups of counters: counters that are common both to IPv4 and IPv6 protocols, counters specific to IPv4 protocol and counters specific to IPv6 protocol:

`dpa_stats_cnt_reass_gen_sel`, `dpa_stats_cnt_reass_ipv4_sel` and `dpa_stats_cnt_reass_ipv6_sel`.

For the Fragmentation and Policer counter, the user can have combination of selections from the corresponding enumerations: `dpa_stats_cnt_frag_sel` and `dpa_stats_cnt_plcr_sel`.

In terms of Classification, there are two types of Classification counters: a Classification Table counter and a Classification Node counter. For a Classification Table counter, the user can retrieve the statistics values for a specific table entry, statistics values that can be any selection or combination of selection from `dpa_stats_cnt_frag_sel` or `dpa_stats_cnt_classif_sel`. The user should be aware that when the same fragmentation object is applied on more than one Classifier Table entries, the statistics values represent the values for the entire number of entries on which the same fragmentation object is applied.

For IPSec and Traffic Manager counters, the user has the possibility of selection from the enum `dpa_stats_cnt_sel`, which return the number of bytes, number of frames or both of them but the returned value has a different meaning, depending on the type of counter and source of the counter.

The DPA Stats component verifies every valid source provided during counter creation by trying to retrieve the corresponding statistics.

The value for a single statistics value is of 4 bytes, but in case multiple statistics values were selected by combining them the value returned will be 4 bytes multiplied with the number of statistics selected.

Return Value

This function returns 0 for success or an error code otherwise. The returned error codes are the following:

- EPERM, if no DPA Stats instance is initialized.
- EINVAL, if one of the function arguments or parameters was incorrect.
- EDOM, if the number of previously created counters reached the maximum preconfigured number of counters.

7.4.3.4.2.2 Function: *dpa_stats_create_class_counter*

Syntax

```
int dpa_stats_create_class_counter(int dpa_stats_id,
    const struct dpa_stats_cls_cnt_params *params, int *dpa_stats_cnt_id);

struct dpa_stats_cls_cnt_params
{
    int    class_members;
    enum dpa_stats_cnt_type type;
    union
    {
        struct dpa_stats_cls_cnt_eth    eth_params;
        struct dpa_stats_cls_cnt_reass  reass_params;
        struct dpa_stats_cls_cnt_frag   frag_params;
        struct dpa_stats_cls_cnt_plcr   plcr_params;
        struct dpa_stats_cls_cnt_classif_tbl  cls_tbl_params;
        struct dpa_stats_cls_cnt_classif_node  cls_node_params;
        struct dpa_stats_cls_cnt_classif  cls_params;
        struct dpa_stats_cls_cnt_ipsec   ipsec_params;
        struct dpa_stats_cls_cnt_traffic_mng  traffic_mng_params;
        struct dpa_stats_cls_cnt_rapidio  rapidio_params;
    };
};

struct dpa_stats_cls_cnt_eth
{
    struct dpa_stats_cnt_eth_src *src;
    unsigned int cnt_sel;
};

struct dpa_stats_cls_cnt_reass
{
    void **reass;
    unsigned int cnt_sel;
};

struct dpa_stats_cls_cnt_frag
{
    void **frag;
    unsigned int cnt_sel;
};

struct dpa_stats_cls_cnt_plcr
{
```

```
void **plcr;
unsigned int cnt_sel;
};

enum dpa_stats_classif_key_type
{
    DPA_STATS_CLASSIF_SINGLE_KEY = 0,
    DPA_STATS_CLASSIF_PAIR_KEY
};

struct dpa_offload_lookup_key_pair
{
    struct dpa_offload_lookup_key *first_key;
    struct dpa_offload_lookup_key *second_key;
};

struct dpa_stats_cls_cnt_classif_tbl
{
    int td;
    enum dpa_stats_classif_key_type key_type;

    union {
        struct dpa_offload_lookup_key **keys;
        struct dpa_offload_lookup_key_pair **pairs;
    };
    unsigned int cnt_sel;
};

enum dpa_stats_classif_node_type
{
    DPA_STATS_CLASSIF_NODE_HASH = 0,
    DPA_STATS_CLASSIF_NODE_INDEXED,
    DPA_STATS_CLASSIF_NODE_EXACT_MATCH
};

struct dpa_stats_cls_cnt_classif_node
{
    void *cc_node;
    enum dpa_stats_classif_node_type cnode_type;
    struct dpa_offload_lookup_key **keys;
    unsigned int cnt_sel;
};

struct dpa_stats_cls_cnt_ipsec
{
    int *sa_id;
    enum dpa_stats_cnt_sel cnt_sel;
};

struct dpa_stats_cls_cnt_traffic_mng
{
    enum dpa_stats_cnt_traffic_mng_src src;
    void **traffic_mng;
    enum dpa_stats_cnt_sel cnt_sel;
};

struct dpa_stats_cls_cnt_rapidio;
```

Parameters

- **dpa_stats_id** - identifier of the DPA Stats instance the class counter belongs to.
- **params** - structure holding the parameters needed to create a counter:
 - **class_members**: the number of members or sources of the same type the class counter provides information for
 - **type**: the type of the class counter that needs to be created: Ethernet counter, Reassembly counter, Fragmentation Counter, Policer Counter, Classification Table, Classification Node counter, IPSec counter, Traffic Manager counter or RapidIO counter (selected from `dpa_stats_cnt_type` enum).
 - **eth_params**: structure holding the parameters needed to create an Ethernet class counter
 - **src**: an array of structures holding the parameters needed to identify multiple sources. Each structure identifies one Ethernet source:
 - **engine_id**: identifier of the engine the Ethernet interface belongs to
 - **eth_id**: identifier of the Ethernet interface, identifier that is relative to the engine the interface belongs to
 - **cnt_sel**: selection of Ethernet available counters, `dpa_stats_cnt_eth_sel` enum
 - **reass_params**: structure holding the parameters needed to create an IP Reassembly counter
 - **reass**: an array of IP Reassembly objects
 - **cnt_sel**: selection of counters from Reassembly available counters, `dpa_stats_cnt_reass_gen_sel`, `dpa_stats_cnt_reass_ipv4_sel` and `dpa_stats_cnt_reass_ipv6_sel` enums
 - **frag_params**: structure holding the parameters needed to create an IP Fragmentation counter
 - **frag**: an array of IP Fragmentation objects
 - **cnt_sel**: selection of counters from Fragmentation available counters, `dpa_stats_cnt_frag_sel` enum
 - **plcr_params**: structure holding the parameters needed to create a Policer counter
 - **plcr**: an array of Policer objects
 - **cnt_sel**: selection of counters from Policer available counters, `dpa_stats_cn_plcr_sel` enum
 - **cls_tbl_params**: structure holding the parameters needed to create a Classification Table counter
 - **td**: descriptor of the table used to perform the Classification
 - **key_type**: the type of the key used to identify an entry in the Classification
 - **keys**: Pointer to an array of keys, where each element of the array can be either a key that identifies a specific entry or NULL in order to obtain the statistics for the *miss* entry.
 - **pairs**: Pointer to an array of "pair-keys" where each element of the array can either be a "pair-key" that identifies a specific entry, or NULL. It is allowed to specify a *miss* table relationship in each of the key pairs by providing NULL for that specific key (i.e. first or second key). If one wants to get statistics for the *miss-miss* pair, you can provide NULL for both keys inside the key pair, or you can simply provide NULL for the pair itself in this array.
 - **cnt_sel**: selection from Classification Table available counters, `dpa_stats_cnt_classif_sel` and `dpa_stats_cnt_frag_sel` enums
 - **cls_node_params**: structure holding the parameters needed to create a Classification Node counter
 - **cc_node**: handle of the Cc node used to perform the Classification
 - **ccnode_type**: the type of FMAN Classification Node
 - **keys**: Pointer to an array of keys, where each element of the array can be either a key that identifies a specific entry or NULL in order to obtain the statistics for the *miss* entry.
 - **cnt_sel**: selection from Classification Node available counters, `dpa_stats_cnt_classif_sel` enum
 - **ipsec_params**: structure holding the parameters needed to create an IPSec counter

- **sa_id**: identifier of the Security Association
- **cnt_sel**: single selection from IPsec available counters, `dpa_stats_cnt_sel` enum
- **traffic_mng_params**: structure holding the parameters needed to create a Traffic Manager counter
- **src**: the type of the source used for the Traffic Manager counter
- **traffic_mng**: depending on the Traffic Manager source, the 'traffic_mng' has a different meaning: it represents an array of pointers to structures of type 'qm_ceetm_cq' in case the traffic source is a "Class Queue" or an array of pointers to structures of type 'qm_ceetm_ccg' in case the traffic source is a "Class Congestion Group"
- **cnt_sel**: single selection from Traffic Manager counter, `dpa_stats_cnt_sel` enum
- **rapidio_params**: structure holding the parameters needed to create a RapidIO counter
- **dpa_stats_cnt_id** - address of an integer variable in which the function will return the counter identifier if it succeeds in creating it; this identifier will be used in further calls to DPA Stats functions to refer to this particular counter.

Description

This function performs all the necessary steps required to create and initialize a class counter. A class counter means a counter that is used to retrieve information from multiple sources of the same type, source which can be either a hardware resource or a software resource.

During class counter creation no memory is allocated but instead the function uses internal structures that were allocated during DPA Stats instance initialization. On success, the function returns a unique counter identifier that should be further used on calls that refer to that counter.

The first parameter of the structure used to create a counter is the type of the counter. Depending on the type of the counter, the user needs to provide the parameters necessary for that type of counter.

For the Ethernet counter, the user can select any combinations of counters from the enumeration `dpa_stats_cnt_eth_sel`.

For the Fragmentation and Policer counter, the user can have combination of selections from the corresponding enumerations: `dpa_stats_cnt_frag_sel` and `dpa_stats_cnt_plcr_sel`.

In terms of Classification, there are two types of Classification counters: a Classification Table counter and a Classification Node counter. For a Classification Table counter, the user can retrieve the statistics values for multiple table entries, statistics values that can be any selection or combination of selection from `dpa_stats_cnt_frag_sel` or `dpa_stats_cnt_classif_sel`. The user should be aware that when the same fragmentation object is applied on more than one Classifier Table entries, the statistics values represent the values for the entire number of entries on which the same fragmentation object is applied. In order to identify multiple table entries, the user application needs to provide an array of keys, keys which can be either single keys or pair of keys. When a pair of keys is used, the first key will identify one entry from the table, while the second key identifies an entry connected with the first entry on a "next" action.

For IPsec and Traffic Manager counters, the user has the possibility of selection from the enum `dpa_stats_cnt_sel`, which return the number of bytes, number of frames or both of them but the returned value has a different meaning, depending on the type of counter and source of the counter.

For two types of class counter, meaning Classification Table and IPsec, the user can provide an invalid source, in which case the returned value of the associated statistics will be 0. In this way, the user application can provide during creation only a subset of valid sources and update the rest of the sources during run-time. For a Classification Table counter an invalid source means a NULL pointer for a specific key, in case the key type is single, or a NULL pointer for the first key, in case the key type is pair of keys. For an IPsec counter, an invalid source means an invalid security association identifier, whose value is equal to -1.

The DPA Stats component verifies every valid source provided during counter creation by trying to retrieve the corresponding statistics.

The value for a single statistics value is of 4 bytes, but in case multiple statistics values were selected by combining them, the value returned will be 4 bytes multiplied with the number of statistics selected.

Return Value

This function returns 0 for success or an error code otherwise. The returned error codes are the following:

- `EPERM`, if no DPA Stats instance is initialized.
- `EINVAL`, if one of the function arguments or parameters was incorrect.
- `EDOM`, if the number of previously created counters reached the maximum preconfigured number of counters.

7.4.3.4.2.2.3 Function: `dpa_stats_modify_class_counter`

Syntax

```
int dpa_stats_modify_class_counter(int dpa_stats_cnt_id,  
    const struct dpa_stats_cls_member_params *params, int member_index)  
  
struct dpa_stats_cls_member_params  
{  
    enum dpa_stats_cls_member_type type;  
    union  
    {  
        struct dpa_offload_lookup_key *key;  
        struct dpa_offload_lookup_key_pair *pair;  
        int sa_id;  
    };  
};
```

Parameters

- **`dpa_stats_cnt_id`** - identifier of the class counter the member belongs to.
- **`params`** - structure holding the parameters needed to update a member of a class:
 - **`type`**: the type of the class member that needs to be updated: single key, pair key or security association identifier (selected from `dpa_stats_cls_member_type` enum)
 - **`key`**: a key descriptor to identify a specific table entry, or NULL to identify the table *miss* counter
 - **`pair`**: a pair of key descriptors to identify a specific table entry. It is allowed to specify a miss table relationship in each of the key pairs by providing NULL for that specific key (i.e. first or second key). If one wants to get statistics for the *miss-miss* pair, you can provide NULL for both keys inside the key pair, or you can simply provide NULL for the pair.
 - **`sa_id`**: a security association identifier
- **`member_index`** - the position of the member in the counter's class.

Description

This function is used to modify a member of a class counter of type Classification Table or IPSec, identified through the position of the member in the class. The user application can invalidate a specific member, in which case the returned statistics value is 0, can set a valid source (key, pair or sa id) in case the member of the class was created with an invalid source or can even update a valid source.

A source is considered invalid in the following cases:

- single key: if the provided key byte and key mask are NULL
- pair key: if the provided first key byte and mask are NULL
- sa id: if the provided security association identifier is -1

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `EINVAL`, if the provided class counter identifier is not valid;

7.4.3.4.2.2.4 Function: `dpa_stats_remove_counter`

Syntax

```
int dpa_stats_remove_counter(int dpa_stats_cnt_id);
```

Parameters

- `dpa_stats_cnt_id` - of the counter (single or class counter) it needs to be removed.

Description

This function is used to remove a previously created DPA Stats single or class counter. The memory occupied by the internal structures of this counter is not released, but instead it is marked as empty and can be used the next time a counter is created. After a counter is removed, it can no longer be used to retrieve values for it.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- `EINVAL`, if the provided instance identifier is not valid;
- `EDOM`, if the provided counter identifier could not be released.

7.4.3.4.2.3 Retrieve or Reset Counters

Table 136. API to Retrieve or Reset Counters

Function Name	Description	Input Parameters	Output Parameters
<code>dpa_stats_get_counters</code>	Retrieves the values for multiple single or class counters.	<ul style="list-style-type: none"> • an array of single or class counters identifiers for which to retrieve values. • the size of the array of requested counters. • flag to specify if requested counters should be reset after their values are retrieved. • offset in the memory area provided at DPA Stats initialization where to write the counter values. • pointer to a location where the function will return the number of bytes that should be written by the DPA Stats in the storage area in case the operation was successful • pointer to a callback function to be called by the DPA Stats module when counter values were written in the storage area or NULL in case the request is a synchronous operation 	The number of bytes written by the DPA Stats in the storage area in case the operation was successful.
<code>dpa_stats_reset_counters</code>	Reset the values for multiple single or class counters.	<ul style="list-style-type: none"> • an array of single or class counters identifiers for which to reset the statistics. • the size of the array of counters to be reset. 	None.

7.4.3.4.2.3.1 Function: `dpa_stats_get_counters`

Syntax

```
int dpa_stats_get_counters(struct dpa_stats_cnt_request_params params,
    int *cnts_len, dpa_stats_request_cb request_done);

typedef void (*dpa_stats_request_cb)(int dpa_stats_id, unsigned int storage_area_offset, unsigned
    int cnts_written, int bytes_written);

struct dpa_stats_cnt_request_params
```

```

{
    int      *cnts_ids;
    unsigned int cnts_ids_len;
    bool     reset_cnts;
    unsigned int storage_area_offset;
};

```

Parameters

- **params** structure holding the parameters needed to retrieve the values for a group of counters
 - **cnts_ids**: pointer to an array of counter identifiers for which to retrieve values
 - **cnts_ids_len**: the size of array of counters to retrieve values
 - **reset_cnts**: flag to specify if counters provided in the cnts_ids array should be reset after the retrieve operation
 - **storage_area_offset**: offset in the storage area where to receive the values of the counters
- **cnts_len**: address of an integer variable in which the function will return the number of bytes occupied by the requested counters values
- **request_done** - pointer to a callback function that will be called by the DPA Stats module when the retrieve operation finished writing the counter values in the storage area or NULL in case the retrieve operation is synchronous
 - **dpa_stats_id**: DPA Stats instance identifier
 - **storage_area_offset**: offset in the storage area where the retrieve operation wrote the counter values
 - **cnts_written**: number of counters that were written in the storage area
 - **bytes_written**: number of bytes that were written in the storage area or the error code in case an error occurred

Description

This function is used to retrieve values for a group of counters. The statistics will be retrieved in the order given by the counter identifiers position in the array cnts_ids.

Each counter statistics value represents cumulative counters and is stored on 4 bytes. The user application has the possibility of resetting the counters values after the retrieve operation, by enabling the reset_cnts flag.

For a counter of type Classification Table or IPSec, in case the user application explicitly provided an invalid source, the corresponding statistics value will be 0 and the retrieve operation is considered successful.

A counters retrieve request can be either synchronous or asynchronous. The difference between them is made through the request_done callback. In case the corresponding parameter is NULL, the request will be synchronous and the counters values will be available in the storage area when the function returns. In case an error occurred during the retrieve operation, the function will return the corresponding error. If the user application provided the request_done callback, the retrieve operation is asynchronous and the callback will be called by the DPA Stats module as soon as the counters values are written in the storage area. In case of an error, the number of bytes_written will have a negative value and it will store the error code.

The user application needs to be aware of the length of the counters values before making the call to the retrieve operation. The only check the DPA Stats module performs is to verify if the offset provided by the call and the size of the counters goes beyond the length of the storage area. If that is not the case, the counters values are written starting with the storage_area_offset and will override any information that might be available in the memory area it needs to write.

Return Value

The function returns 0 on success and an error code otherwise. The returned error codes are the following:

- **EINVAL**, if invalid function arguments were provided;

- `ENOENT`, if an error occurred during statistics retrieve for an Ethernet counter;
- `ESRCH`, if an error occurred during statistics retrieve for a Reassembly counter;
- `EINTR`, if an error occurred during statistics retrieve for a Fragmentation counter;
- `EIO`, if an error occurred during statistics retrieve for a Classifier Table counter;
- `ENXIO`, if an error occurred during statistics retrieve for a Classification Node counter;
- `E2BIG`, if an error occurred during statistics retrieve for an IPSec counter.

7.4.3.4.2.3.2 Function: `dpa_stats_reset_counters`

Syntax

```
int dpa_stats_reset_counters(int *cnts_ids, unsigned int cnts_ids_len);
```

Parameters

- `cnts_ids` - pointer to an array of counter identifiers for which to reset the statistics.
- `cnts_ids_len` - the size of array of counters identifiers to reset.

Description

This function is used to reset the statistics for multiple single or class counters.

Return Value

The function returns 0 on success and an error code otherwise. The returned error code is the following:

- `EINVAL`, if invalid function arguments were provided;

7.4.3.4.2.4 Create or Remove Sampling Group

Table 137. API to Create or Remove Sampling Group

Function Name	Description	Input Parameters	Output Parameters
<code>dpa_stats_create_sampling_group</code>	Create a sampling group.	TBD	None.
<code>dpa_stats_remove_sampling_group</code>	Remove a sampling group.	TBD	None.

7.4.3.4.2.4.1 Function: `dpa_stats_create_sampling_group`

Syntax

TBD

Parameters

TBD

Description

This function will allow creating a sampling group for a number of counters. The purpose of a sampling group is to perform counters sampling at a specific frequency in order to assure counter rollover. By creating sampling groups, the application will always receive cumulative counters that take in account also the number of rollovers.

Return Value

TBD

7.4.3.4.2.4.2 Function: dpa_stats_create_sampling_group

Syntax

TBD

Parameters

TBD

Description

This function is used to remove a previously created sampling group.

Return Value

TBD

7.4.3.5 Network Function Layer

The *Network Function Layer* is a DPAA offloading driver extension which provides a service level API in Linux user space to enable NXP customers to easily configure hardware acceleration for specific standard network services on different NXP devices directly from their user space applications. The Network Function API is **platform agnostic** and is presently available for two families of NXP devices:

- the **Layerscape** family - LS2085A
- the **QorIQ** family equipped with DPAA 1.x network functions accelerator - P devices, B devices and T devices

On QorIQ devices the Network Function Layer (a.k.a. **NF Layer**) is based on the current DPAA 1.x offloading drivers.

The Network Function Layer includes the following types of network services. The network services supported on DPAA 1.x devices are marked in the table below.

Table 138. Network Function Layer Services

Service Name	Supported on DPAA 1.x?
Firewall	
IP Forwarding (IPv4 and IPv6)	yes
IPSec	yes
Interface and Network Namespace Management	

Network Function API runtime services work identically across the entire list of supported NXP devices.

As far as the initialization is concerned, each NXP devices family has its specific mechanism for initializing its network hardware accelerator. Due to the major differences in design and concepts, NF API could not align the initialization across device families in a clear and easy-to-use way. As a consequence, each type of device has its own initialization sequence before starting to use NF API.

In QorIQ DPAA 1.x based devices, the offloading application / integration layer is in charge of the platform initialization. The following is a most commonly used initialization sequence.

1. Interpret PCD configuration file and create PCD model (using *fmplib*, for instance)
2. Initialize network interfaces (typically using *USDPAAs*)
3. Initialize the acceleration infrastructure (queues, buffer pools, classification tables, etc.) (typically using *USDPAAs*)
4. Update the PCD model with runtime information
5. Apply the PCD model and configure FMan (using *fmplib* and *fmlib*, for instance)
6. Provide initialization data to the NF Layer
7. Use the NF Layer for runtime network function services offloading configuration

The [Network Function API reference manual](#) is provided as a Doxygen generated documentation with the QorIQ Linux SDK.

7.4.3.6 References

Table 139. References

Index	Title
1	USDPAAs PPAC User Guide
2	QMan/BMan API Guide
2	Frame Manager Configuration Tool Reference Manual
2	Frame Manager Configuration Tool: Examples

7.4.4 USDPAAs PPAC User Manual

7.4.4.1 USDPAAs PPAC Users Manual

The Packet Processing Application Core (PPAC), as a component of the USDPAAs software framework that directly access queue manager (QMan) and buffer manager (BMan), helps develop and maintain common packet processing code between multiple USDPAAs sample applications developed by NXP.

This document provides the following information.

- PPAC and PPAM detail.
- Layout of a PPAC/PPAM application.
- User controls to alter PPAC/PPAM application behavior.

Common packet processing code reduces maintenance costs, keeping common packet processing centralized and supporting updates done once rather than per application. Application-specific packet processing code is part of the application Packet Processing Application Module (PPAM). Together, the application specific PPAM and the common PPAC make up the complete application.

Users of USDPAAs are not required to use the PPAC/PPAM model. Indeed, NXP only intends to extend PPAC capabilities to meet needs for sample USDPAAs applications rather than for other generic usage.

This document assumes familiarity with the following concepts and documentation.

- QMan/BMan API Reference Manual
- USDPAAs User Guide
- Linux user-space programming (POSIX, pthreads, etc.)

7.4.4.2 Overview of PPAC

The “reflector” application, developed as a small, stand-alone application for the USDPAAs driver components, converted to a PPAC/PPAM application.

Origins of PPAC.

Early in USDPAAs development, the “reflector” application was developed as a small, stand-alone application written on top of the USDPAAs driver components. This application was eventually converted to a PPAC/PPAM application. Currently, there exists in the SDK another application, called `hello_reflector`, which is a new version of that stand-alone application.

the reflector application is very simple in terms of its packet processing logic (it flips Ethernet and IPv4 headers in regular IPv4 packets and forwards them back out the interface they arrive on, whilst discarding everything else), it facilitated and continues to facilitate the development, testing, and benchmarking of many generic aspects of packet-processing applications, not least of which are:

- parsing and enacting device configuration.
- driver initialisation.

Goals of the PPAC Users Manual.

With the extension of configurability and addition of features to the basic reflector application, and the requirement for other USDPAAs packet-processing applications that share many of the same “general” requirements mentioned above, it was determined that the generic packet-processing application logic in reflector should be separated from the logic implementing the reflector-specific packet-processing “decision”.

Definition of PPAC and PPAM

The division of the packet-processing applications into a common, general infrastructure and application-specific components gives rise to the following two acronyms that now make up any PPAC-based application.

- **PPAC:** Packet-Processing Application Core. This is the generic application framework, implementing all of the “generic aspects of packet-processing applications” listed above. In the simplest case (the reflector application), this is essentially everything except the logic that flips the packet headers and selects the input interface as the forwarding destination. In particular, this includes the `main()` function .
- **PPAM:** Packet-Processing Application Module This specialises PPAC into a “real application” by implementing the missing piece – namely the “packet-processing” specifics of the desired application. Each PPAM is compiled and linked with the PPAC component, but produces a stand-alone (and PPAM-specific) application binary.

Object Orientation

For some of the discussion that will follow, it may be helpful to use an object-oriented metaphor where PPAC is a base-class for packet-processing applications such that the packet-processing function is a pure virtual method (which is to say, non-existent and so must be implemented by derived classes). In this metaphor, PPAC-based applications are instantiations of derived classes, called PPAMs. Now all of this is implemented in C and the object-oriented metaphor should not be taken too literally, but it may help as an illustrative tool.

Limitation of Scope

PPAC supports a limited set of features; the application framework does not provide a solution for all conceivable USDPAAs-based applications. PPAC provides as much consistency as possible for the packet-processing applications that are bundled with the USDPAAs component of the SDK. That is, PPAC delivers a consistent user experience in both the look-and-feel and the elimination of code duplication. The scope and complexity of PPAC is intentionally limited to the current development, testing, and benchmarking requirements of USDPAAs and the applications bundled with it. In particular, it is more important that PPAC (and PPAM) code be sufficiently comprehensible that application authors can understand how it works, than to

provide a complete infrastructural solution for all potential USDPAA-based applications. With this in mind, please see the “Chronology of a DPAA application” section, which attempts to provide pointers to key sections of PPAC code specifically to help with the development of non-PPAC applications.

Performance Challenges

Any abstraction underlying USDPAA applications needs to avoid imposing additional levels of function calls or other indirection. Eg. for 64 byte packets, when using between 1 and 4 CPUs on a p4080 DS, the peak processing rate of reflector application averages out to approximately 170 CPU cycles per-packet. An ad-hoc experiment to add just one extra level of indirection to that processing path introduced an additional overhead of ~20 cycles, causing a 12% performance degradation.

The same concern for function calls also applies to the data structure relationships between the PPAC and PPAM components. Use of pointers to relate discontinuous data-structures would necessarily lead to additional indirection and latency in the fast-path processing logic.

As it happens, splitting the fast-path logic of a data-path application into an abstraction layer (PPAC) and an application-specific layer (PPAM) whilst unifying the corresponding data-structures and without introducing any additional level of function calls is not impossible, but nor is it trivial.

7.4.4.3 Overview of PPAC Method and Implementation

the PPAC/PPAM interface is implemented by a strategic use of inlining.

Method

The reasons mentioned above should help explain why the PPAC/PPAM interface is implemented by a strategic use of inlining. It has been kept as simple as reasonably possible, but is clearly less straightforward than it would be if the afore-mentioned performance considerations were not a factor. The resulting implementation is organized in such a way that the compiler is able to inline the fast-path code of PPAC and PPAM together, as though they were written as a single (or “flat”) application. As such, the PPAM version of reflector has no performance degradation relative to earlier non-PPAM versions, despite the PPAC/PPAM split. Note that all non-performance-critical code is compiled and linked conventionally within the PPAC library, without inlining. The application PPAM determines for itself how much code to implement within the inlining scope.

From both the data-structure and function-call points of view, the inlining relationship between PPAC and PPAM needs to resolve a circular dependency:

- PPAM within PPAC: the fast-path PPAC packet-processing logic needs to “compile in” the PPAM packet-processing logic – i.e. to have the compiler expand the PPAM-specific packet-processing logic “inline”. Similarly the PPAC data-structure representing the interface and FQ state being handled needs to encompass any corresponding PPAM-specific state. I.e. the combined data-structures need to be declared statically rather than linked via indirection.
- PPAC within PPAM: the PPAM packet-processing logic will invariably need to issue one or more enqueue/transmit operations (or a release/drop operation) as a result of its packet-processing decision, and these operations must necessarily be coordinated by PPAC (otherwise generic mechanisms like buffer-management, flow-control, [etc] cannot be coordinated). So any PPAC-provided functions to be used by the PPAM packet-processing logic will need to be expanded “inline”.

Thus, for at least a subset of the PPAC and PPAM code, there will be two layers of inlining; PPAC within PPAM within PPAC. The following diagram shows the behaviour of a PPAC-based application where maximum inlining of the packet-processing code is used (this is the case with the reflector application).

By way of example, the reflector application presents a very minimal PPAM component, essentially the “header-flipping” function, and so it presents only the one C source file that is used to inline the PPAC machinery and compile within that “inlining” scope. The IPFwd application on the other hand implements significant components of a network stack and routing logic, so much of its logic (including some that is used in the packet-processing fast-path) is conventional C code without inlining. The fact remains that the PPAC abstraction itself does not contribute any additional functional indirection, in order to retain the property of providing no performance overhead relative to an equivalent application written without the use of such an abstraction.

The following diagram illustrates a PPAC-based application where only a subset of the packet-processing code is inlined. This example proposes that each FQ carries packets for one or more network flows (due to classification and hashing in FMan), and that the PPAM associates each such FQ with state to implement a “recently-processed flow cache” (or “RPFC”

as we'll call it for the purposes of this illustration), which in its simplest form would be a single-entry. It's worth noting that if network traffic is classified/hashed into multiple Rx FQs, then the probability increases that consecutive packets on a given FQ are from the same flow. If the packet matches a flow in the RPFC then packet-processing will complete entirely within the inlined/optimised code. Furthermore, if FQ context-stashing is enabled, then this RPFC state would be pre-positioned in the CPU's L1 or L2 cache and so packet-processing would complete without any memory latencies. The “slow path” is the case where the packet does not match in the RPFC, the processing of the packet will then involve code that is not inlined into the packet-handler, and will likewise use state that is not necessarily present in processor cache.

Implementing a New PPAM

The upcoming sections provide a guide to the noteworthy PPAC files, data-structures, and interfaces, and also enumerate those required of a minimal PPAM application. Additionally, the “reflector” application should be used as a reference example to understanding this interface and for starting out new PPAM applications, and indeed could be copied and modified to begin doing such work due to its simplicity.

As was mentioned at the start of this document, such descriptions are provided to enable creation of quick sample applications that require the same common packet processing as other USDPAA sample applications. The following sections also help users understand the implementation of PPAC/PPAM sample applications so that they can extract any useful capability there into their own non-PPAC application.

About the Upcoming Document Sections

In the upcoming sections of this document, the PPAC/PPAM specifics will be introduced in the following order;

- PPAC files: this section will document the files that make up PPAC and describe their characteristics.
- PPAM files: this section documents the minimal set of files that a PPAM (i.e. PPAC-based application) needs to provide.
- Packet-processing data structures: this section documents the data types used in PPAC-based applications, which necessarily includes the sub-types to be implemented by PPAMs.
- PPAM-provided functions: this section describes how PPAM can implement its data-types and the corresponding handler functions that are invoked by the PPAC infrastructure.
- PPAC-provided functions: this section describes those PPAC functions that can be called from within PPAM implementations. Most interestingly, this describes the interfaces available to PPAM packet-handling logic for the transmitting or dropping of frames.

7.4.4.4 PPAC Files

The PPAC abstraction comprises the following five (5) files.

File: `apps/include/ppac_interface.h`

This header declares the network interface structure, “struct ppac_interface”, which combines the PPAC and PPAM structures into one, thus it requires that the PPAM declarations already be defined. This type encompasses all Rx and Tx FQs.

File: `apps/include/ppac.h`

This header declares a variety of definitions required by both PPAC and PPAM logic, which includes:

- pre-compiler symbols controlling parameters and options for PPAC compilation.
- pre-declarations of PPAM functions to be called by PPAC code.
- pre-declarations of PPAC functions to be called by PPAM code.
- pre-declarations of PPAC functions called between the code that is inlined into PPAM and the code that is compiled in to the PPAC library.
- PPAC inline code that needs to be available at a wider scope than that of `apps/include/ppac.c`.
- pre-declarations of global variables required by inline functions.
- pre-declarations of global variables representing weakly-linked PPAC constants (ie. that a PPAM can override).

File: `3.3apps/include/ppac.c`

This “header” contains the fast-path PPAC code that needs to be expanded/inlined with the corresponding fast-path PPAM code. It also contains any other code that needs to exist at the same scope as the fast-path code. The main characteristic of this file is that it must be included exactly one time, by the PPAM code, and it must be included after PPAM has declared its own fast-path hooks (the included PPAC fast-path logic calls these PPAM hooks). This file includes:

- PPAC-defined QMan callbacks for handling dequeued frames from each class of Rx FQ. (These invoke PPAM-specific hooks which are thereby inlined into a single function by the compiler.)
- the interface manipulation code (setup, teardown, enable, disable) that binds these fast-path callbacks to their corresponding FQ objects and provides fast-path as well as setup and teardown hooks to PPAM.

File: apps/ppac/main.c

This file contains the PPAC code that can be compiled independently of PPAM and made available as a linkable library (libusdpaa_ppac.a). This includes:

- global constants and variables that are not required directly from inline fast-path logic.
- FQ, buffer pool, and CGR manipulation code (setup, teardown, etc).
- thread management (including IPC).
- buffer pool management.
- global setup and teardown (invokes the per-interface setup/teardown logic contained within apps/include/ppac.c)

File: apps/ppac/ppac.lids

This file is a linker script to be used when compiling PPAM applications. The CLI implementation in PPAC allows both PPAC and PPAM code to declare and implement commands to be added to the interface, and the underlying definitions compile these into a dedicated linker section. By linking the resulting application with this linker-script, the location and length of this linker section is known to PPAC CLI code. (The alternative to using this approach is to have PPAC and PPAM dynamically build a list of CLI command handlers, which requires a coordinated initialisation phase, whereas this linker-based mechanism is static.)

Use of this linker script is shown in the PPAM section covering <app-dir>/Makefile.am.

7.4.4.5 PPAM Files

The PPAM application may be composed of as little as one C file, and of course may be significantly more complex. The main requirement is that exactly one C file include the PPAC inline machinery contained in apps/include/ppac.c.

Other C files may include the other headers in order to share PPAC definitions as widely as required.

The three (3) files listed here correspond to those implemented by the reflector PPAM (which uses two files in order to separate some definitions into a header for the sake of clarity). A far more elaborate PPAM example can be found in the case of the IPFwd PPAM.

File: <app-dir>/ppam_interface.h

This file is defined by reflector and IPFwd PPAMs for the sake of clarity - there is no technical or other reason for such a header to exist separately from any other PPAM files. In reflector, this header contains the PPAM-specific data-structures that are pre-requisites for the definition of PPAC data-types contained in ppac_interface.h. i.e. the PPAM code includes ppam_interface.h prior to including ppac.c (which is what requires and includes ppac_interface.h). PPAM code can just as easily implement the same definitions elsewhere, so long as they are defined prior to the one-off inclusion of ppac.c.

File: <app-dir>/*.c

This section refers to the primary PPAM C file that includes the PPAC inline machinery. It must ensure that the required data-structures are defined prior to the inclusion of ppac_interface.h, and it must ensure that the required PPAM hooks are declared before inclusion of ppac.c. In the case of reflector, the data-structures are defined in ppam_interface.h as explained above, and apps/reflector/reflector.c defines and implements the PPAM hooks as inline functions prior to including ppac.c. (They only need to be pre-declared before inclusion of ppac.c if you prefer to implement them afterwards. If you don't wish to use inline functions to save function jumps, then they can be implemented in other C files and resolved by the linker.)

The PPAM must also define the symbols that are required by PPAC in order to link, though this does not necessarily need to be done within the primary PPAM C file that is including the PPAC inline machinery. In the reflector case however, these symbols are in fact instantiated in the same C file, for simplicity's sake. These symbols relate to the CLI, and as reflector adds no commands to the base commands implemented within PPAC, so the required PPAM symbols have trivial implementations.

File: <app-dir>/Makefile.am

This section presumes that the PPAM application is being built by the USDPAA build system. This need not be the case, but an external build-system would still need to address the same requirements in its own way. The following definitions are used to build and link the reflector application;

```
bin_PROGRAMS = reflector

AM_CFLAGS := -I$(TOP_LEVEL)/apps/include

reflector_SOURCES := reflector.c
reflector_LDADD := usdpaa_ppac usdpaa_syscfg usdpaa_qbman usdpaa_fman \
                  usdpaa_dma_mem usdpaa_of
reflector_LDFLAGS := $(LIBXML2_LDFLAGS) $(LIBEDIT_LDFLAGS) \
                    -T $(TOP_LEVEL)/apps/ppac/ppac.lds
```

The key points here are:

- the PPAC logic contained in apps/include/ needs to be accessible via includes of <ppac.c>, <ppac.h>, and <ppac_interface.h>.
- the application needs to link with libusdpaa_ppac.a, and by dependence needs to link with the other named libraries that provide USDPAA configuration parsing and driver functionality.
- the linker flags need to provide any dependencies for linking with Gnome Libxml2. In the SDK containing USDPAA, this is set to point to the installation directory for system libraries and to resolve with the “xml2”, “z” (zip), and “m” (math) libraries.
- the linker flags need to provide any dependencies for linking with Editline. In the SDK containing USDPAA, this is set to point to the installation directory for system libraries and to resolve with the “edit”, and “curses” (ncurses) libraries.
- the ppac.lds linker script should be used to add the array-delimitation definitions required by the PPAC CLI implementation.

When implementing a new PPAM application, the above model can be followed – simply create a new sub-directory within the build-tree, add that directory's name to the SUBDIRS definition in parent directory's Makefile.am file, and copy-paste-and-modify the above “reflector” definitions to your requirements:

- if your directory is in fact to host sub-directories with their own libraries and/or applications, then add a SUBDIRS definition to your Makefile.am file (for further information about this, look at the IPFwd reference application).
- change all occurrences of “reflector” to the name of your desired application (or if your directory is only going to host sub-directories with their own targets, then remove everything except the SUBDIRS definition described in the preceding point).
- change the “*_SOURCES” attribute to describe the C files that need to be compiled. The above example for reflector only lists one C file because that's all it contains, but less trivial applications will have more (the list should be space-separated, with back-slashes for multi-line entries).
- Extend “*_LDADD” and “*_LDFLAGS” attributes if required.

7.4.4.6 Packet-processing data structures

A description of the data-structures that are used within PPAC (and PPAM) for handling network interfaces and the FQs they contain.

Within each such PPAC type, the data-structure contains a PPAM-specific equivalent, allowing PPAM to “sub-class” PPAC types for its application-specific purposes (to use the object-oriented metaphor mentioned in section 2.2.2). In section 6, these PPAM types will be described, as will the handler functions PPAM can provide for them (i.e. the “pure virtual methods” that can be implemented in “derived classes”, to continue the object-orientation imagery).

File: struct ppac_interface

(Defined in `apps/include/ppac_interface.h`.) This structure represents a network interface and associated state. In keeping with the object-oriented metaphor, this structure includes a PPAM-defined structure within it, “`struct ppam_interface`”, which the PPAM can use to define and maintain interface-wide state of its own.

This structure contains the following state:

- a pointer to information about the configuration of the FMan ethernet port corresponding to this “interface”
- a PPAM-defined “`struct ppam_interface`” object for use by PPAM processing.
- an array of QMan FQ objects representing the Tx FQs for the interface.
- objects representing the unique Rx FQs associated with the interface . These structures are described subsequently.
- a linked-list of PCD ranges, each represented by the “`struct ppac_pcd_range`” type. Each PCD range is an array of objects representing the “hashed” Rx FQs for a policy of the interface. In high-performance datapath applications, FMan distributes most traffic to this range of FQs based on parse and classification processing.

File: `struct ppac_rx_{error|default|hash}`

These structures represent individual FQs that FMan enqueues frames to during Rx processing and that are dequeued to software portals for application processing. The “error” FQ receives frames due to any errors in FMan Rx processing, otherwise frames are either enqueued to one of many “hash” FQs (due to FMan parse and classify processing) or to the “default” FQ. For each structure type, the PPAC structure includes a PPAM-defined structure within it, “`struct ppam_rx_*`”, which the PPAM can use to define and maintain FQ-wide state of its own.

These structures contain the following state:

- the “`struct qman_fq`” object used with the QMan driver for manipulating the hardware FQ object as well as handling dequeued frames. The nature of the QMan driver API is such that this object is also the “FQ context” that gets stashed to processor cache if the FQ is configured for any stashing at all. This object is intentionally the first element in the per-FQ PPAC structure, because the application can control how many cache lines of “FQ context” should be stashed during dequeue operations, so any adjacent state following this object can also benefit from dequeue stashing.
- a PPAM-defined “`struct ppam_rx_*`” object for use by PPAM processing. As mentioned in the previous item, QMan dequeue processing can be configured to stash this state to processor cache during the dequeue operation in hardware, potentially avoiding or minimising the possibility for cache-misses in fast-path processing.

File: `struct ppac_tx_{error|confirm}`

These structures are similar to those described for “`struct ppac_rx_*`”, so the details will not be repeated here. The key difference is that these FQs are enqueued to by FMan during Tx processing and so represent the consequence of an attempt by software to forward already-processed traffic, rather the consequence of Rx processing of frames that software has not yet seen.

The “error” FQ naturally will receive any frames where FMan or QMan encountered an error during transmit (implying that the packet was not transmitted successfully).

The “confirm” FQ is intended to return frames back to software that have been successfully transmitted, but this is not a mode of operation that PPAC currently enables (nor are there currently any options to turn this on).

For current PPAC application purposes, all transmitted frames are composed of buffers sourced from BMan pools, and so the Tx FQs are configured to make FMan autonomously release all buffers from the transmitted frames back to the pools they belong to. Use of a “confirm” FQ is unnecessary here and would just degrade system performance. If an application is going to transmit traffic from buffers that do not originate from BMan buffer pools, then it may be necessary to reconfigure the Tx FQs to use the confirmation feature.

7.4.4.7 PPAM-provided Functions

PPAC/PPAM-based applications are driven by PPAC itself which implements the `main()` function as well as the run-to-completion functions that execute in the worker threads.

Here are listed the functions that can be implemented by a PPAM application. Application-specific behaviour of PPAM applications is determined by the way in which PPAMs implement the interfaces that PPAC expects to exist. A subset of these interfaces are performance-critical, so as previously described they can be implemented so as to benefit from compiler inlining.

When considering the object-oriented metaphor, these PPAM-provided functions correspond to pure virtual methods in the PPAC base-class that the PPAM derived class should implement.

Many of these functions naturally map to the PPAM-specific structures that are embedded in each of the corresponding PPAC data structures defined in the previous section, so will be documented in the same sequence. The exceptions are the “global initialisation” and “polling” handlers, which are not scoped to any particular interface or FQ. (In object-oriented language, these latter handlers would be considered “class methods”.)

Global Initialisation

The handlers described in this section are called when initialising and cleaning the application process itself and the individual worker threads created within that process. The PPAC library declares weakly-linked versions of these interfaces, so a PPAM is not required to implement its own versions unless it needs such hooks (though if it does the PPAM-provided versions will be called instead of the PPAC-provided fallbacks).

Process Initialization and Cleanup

The “init” function is called when the application starts up (prior to creation of any network interface structures or any worker threads) and the “finish” function is called when the application is exiting (after all worker threads and network interfaces are destroyed). Any state managed by these hooks should be implemented using (non-thread-local) global variables.

```
int ppam_init(void) ;
void ppam_finish(void) ;
```

If `ppam_init()` returns non-zero, that is considered failure and application initialisation will be abandoned.

Worker Thread Initialisation and Cleanup

These functions are called when a worker thread starts up or is being destroyed. Any state managed by these hooks should be implemented using thread-local global variables, i.e. using the “`__thread`” gcc attribute.

```
int ppam_thread_init(void) ;
void ppam_thread_finish(void) ;
```

As usual, if the initialisation function returns non-zero, that is considered failure and so creation of the worker thread will be abandoned. Note also that thread-local QMan/BMan portals are already initialised when `ppam_thread_init()` is called and are not destroyed until after `ppam_thread_finish()` has returned, so it is safe to use portal-dependent QMan/BMan APIs within these functions.

Polling Handlers

The run-to-completion loop of each worker thread (implemented by PPAC) can optionally call a PPAM-provided polling function to allow PPAM applications to perform general-purpose processing. E.g. to implement background processing tasks, generate frames for transmit that are not in reaction to a received frame (IPC), etc. As with the global initialisation handlers, PPAC defines a weakly-linked version of this polling function, so PPAMs are not required to implement it unless they need it. There is also a thread-local global variable that PPAM code can be set non-zero whenever it wishes the run-to-completion loop to call this polling handler.

```
extern __thread int ppam_thread_poll_enabled;
int ppam_thread_poll(void) ;
```

If `ppam_thread_poll_enabled` is zero (the default), the `ppam_thread_poll()` function is not called from within the run-to-completion loop (minimising overhead in the case that no polling hook is required). Note that the weakly-linked version of

ppam_thread_poll() implemented by PPAC will intentionally kill the application, because it is illegal to set ppam_thread_poll_enabled non-zero unless the PPAM application implements its own ppam_thread_poll() function.

File: struct ppam_interface

This is the PPAM-specific object representing a network interface. It and its PPAC-wrapper form the “parent” for the objects that representing the individual FQs A PPAM must implement the following hooks for this object;

Initialisation

This function is called as the network interface is initialised but prior to all Rx or Tx FQ objects being initialised, so this hook is called before all ppam_rx_*_init() or ppam_tx_*_init() hooks are called for object that belong to the this interface. The parameters provide a pointer to the PPAM-specific interface state (which is uninitialised on entry and should be initialised to the PPAM's requirements), a pointer to the configuration information for this network interface, and also indicates to PPAM the numbe of Tx FQs that will be initialised for this interface;

```
int ppam_interface_init(struct ppam_interface *p,  
                       const struct fm_eth_port_cfg *cfg,  
                       unsigned int num_tx_fqs);
```

If this function returns non-zero, it will be interpreted as an error code and initialisation of the interface will be abandoned.

Cleanup

This function is called as the network interface is being cleaned up and after all Rx and Tx FQ objects have been destroyed, so this hook is called after all ppam_rx_*_finish() and ppam_tx_*_finish() hooks are called for objects that belong to this interface.

```
void ppam_interface_finish(struct ppam_interface *p);
```

Tx FQ Enumeration

The ppam_interface_init() function described earlier notifies the PPAM of the number of Tx FQs that will be initialised for this interface. So the purpose of this function is to notify the PPAM as each individual Tx FQID is (dynamically) allocated and initialised for this purpose. I.e. the PPAM can use the initial ppam_interface_init() hook to know how many Tx FQs the interface will have (e.g. in order to allocate a sufficiently large array, or to verify that an existing static array is large enough) and then use the subsequent ppam_interface_tx_fqid() hooks to obtain each such FQID. This process precedes all the Rx FQ hooks described below.

```
void ppam_interface_tx_fqid(struct ppam_interface *p,  
                           unsigned idx,  
                           uint32_t fqid);
```

7.4.4.8 PPAM Rx and Tx

These are the PPAM-specific objects representing the Rx-related FQs of a network interface. A PPAM must implement the following hooks for these objects;

PPAM Rx Initialization

These functions are called just prior to the corresponding Rx-related FQ being initialised by PPAC. The parameters provide a pointer to the PPAM-specific FQ state (which is uninitialised on entry and should be initialised to the PPAM's requirements), a pointer to the PPAM-specific state for the interface this FQ belongs to (which will have already been initialised by the PPAM's own ppam_interface_init() function), and a pointer to the QMan dequeue-stashing configuration that will be used to initialise the FQ (the PPAM can modify this structure before returning in order to modify the stashing configuration to be applied to the FQ). Note that the “hash” function also provides an index parameter, as the “rx_hash” FQs form arrays called “PCD ranges”. If the PPAM logic needs to know the number (and order) of these ranges, and the number of “hash” FQs within each of them, it can determine that from the ::list field of the “cfg” parameter passed to the ppam_interface_init() hook.

```
int ppam_rx_error_init(struct ppam_rx_error *p,  
                      struct ppam_interface *_if,
```

```

        struct qm_fqd_stashing *stash_opts);
int ppam_rx_default_init(struct ppam_rx_default *p,
                        struct ppam_interface *_if,
                        struct qm_fqd_stashing *stash_opts);
int ppam_rx_hash_init(struct ppam_rx_hash *p,
                     struct ppam_interface *_if,
                     unsigned idx,
                     struct qm_fqd_stashing *stash_opts);

```

If these functions return non-zero, it will be interpreted as an error code and initialisation of the interface will be abandoned.

PPAM Rx Cleanup

These functions are called as the network interface is being cleaned-up, each such hook is called just prior to the corresponding FQ object being destroyed. The interface-wide `ppam_interface_finish()` hook is last to be called once all FQs belonging to the interface have been cleaned up. The parameters match those to the corresponding `_init()` functions, with the exception that no stashing configuration is provided (because the FQ is being destroyed, not initialised).

```

void ppam_rx_error_finish(struct ppam_rx_error *p,
                         struct ppam_interface *_if);
void ppam_rx_default_finish(struct ppam_rx_default *p,
                            struct ppam_interface *_if);
void ppam_rx_hash_finish(struct ppam_rx_hash *p,
                         struct ppam_interface *_if,
                         unsigned idx);

```

PPAM Rx Packet Processing

These functions essentially represent the PPAM's "fast-path," particular in the "hash" case. When processing dequeued frames, the QMan driver invokes the callback associated with the FQ object, which is implemented by PPAC as part of the "inlined" component of the code. These PPAC-implemented callbacks handle any "infrastructure" responsibilities for processing the dequeued frame and the FQ it was dequeued from (e.g. providing support for order preservation or restoration, flow-control, etc), but they defer to these PPAM-specific packet-processing hooks to examine the packets and determine what to do with them. The parameters provide a pointer to the PPAM-specific FQ state, a pointer to the PPAM-specific state for the interface this FQ belongs to (except in the "hash" case), and a pointer to the QMan DQRR entry containing the frame descriptor and other status about the dequeue operation for the frame (and the FQ it was dequeued from, e.g. whether the dequeue left empty).

If these hooks are implemented as inlines, then the compiler is able expand the PPAC and PPAM packet-processing logic into a single function layer, and many also make other optimisations possible (such as not having to prepare and pass parameters to the PPAM hooks if they're unused, etc).

```

void ppam_rx_error_cb(struct ppam_rx_error *p,
                     struct ppam_interface *_if,
                     const struct qm_dqrr_entry *dqrr);
void ppam_rx_default_cb(struct ppam_rx_default *p,
                        struct ppam_interface *_if,
                        const struct qm_dqrr_entry *dqrr);
void ppam_rx_hash_cb(struct ppam_rx_hash *p,
                     const struct qm_dqrr_entry *dqrr);

```

Note that the "hash" callback hook does not provide the PPAM-specific state for the interface, nor does it provide the index of the hash callback. These two omissions are primarily justified by the observation that this is the most performance-defining code-path and so should not include any potentially-unnecessary overheads, whether or not the packet-processing is fully inlined. Also, the "error" and "default" FQs are global to the interface and so packets that arrive on them represent events that are in some way "exceptional", whereas packet-processing on the "hash" FQs should be the normal case and less likely to depend on interface-global state. If this is not the case and the "hash" callback does require the interface state and/or the index, then that state should be added to the "struct ppam_rx_hash" type and initialised during the `ppam_rx_hash_init()` hook,

ensuring that it does not need to be “computed” by PPAC for passing as a parameter, and may also benefit from stashing of the “struct ppam_rx_hash” structure during dequeue operation.

It is the responsibility of these callbacks to determine what to do with the frame and in doing so should commit to the corresponding action before returning. The APIs available for this are described in a later section called “PPAC-provided functions”; but in essence they provide “drop” and “forward” mechanisms.

PPAM Tx

As with the description of these data-structures in the previous section (“Packet-processing data structures”), the PPAM-specific functions for them are very similar to those described for “struct ppac_rx_” and so will not be repeated here. As mentioned in the data-structure discussion, the key difference is that these FQs are enqueued to by FMan during Tx processing and so represent the consequence of an attempt by software to forward already-processed traffic, rather the consequence of Rx processing of frames that software has not yet seen. Please see that data-structure discussion for more specifics.

PPAM Tx Initialization

```
int ppam_tx_error_init(struct ppam_tx_error *p,  
                      struct ppam_interface *_if,  
                      struct qm_fqd_stashing *stash_opts);  
int ppam_tx_confirm_init(struct ppam_tx_confirm *p,  
                        struct ppam_interface *_if,  
                        struct qm_fqd_stashing *stash_opts);
```

PPAM Tx Cleanup

```
void ppam_tx_error_finish(struct ppam_tx_error *p,  
                         struct ppam_interface *_if);  
void ppam_tx_confirm_finish(struct ppam_tx_confirm *p,  
                           struct ppam_interface *_if);
```

PPAM Packet Processing

```
void ppam_tx_error_cb(struct ppam_tx_error *p,  
                     struct ppam_interface *_if,  
                     const struct qm_dqrr_entry *dqrr);  
void ppam_tx_default_cb(struct ppam_tx_confirm *p,  
                       struct ppam_interface *_if,  
                       const struct qm_dqrr_entry *dqrr);
```

7.4.4.9 PPAC Provided Functions

Describes the functions that PPAC makes available for use from PPAM application code.

PPAC Packet Processing

The PPAM packet-processing hooks must adhere to a simple set of rules when processing dequeued frames in order to ensure sane functioning of the system. For every packet examined, exactly one of two mutually-exclusive actions must be taken prior to returning from the packet-processing hook; “drop” or “forward”. (The “forward” APIs are called “send”, for the sole reason that the latter has fewer letters.) The “drop” case is very simple, while the “forward” case presents some options..

Note that these APIs assume that no intermediary processing of the frame is required, and in particular do not prescribe any way of dealing with accelerator hardware (cryptographic, pattern-matching, or otherwise). See section 7.2 for more information.

PPAC Packet Processing - Drop a Frame

If a PPAM packet-processing hook has examined a frame and determined that it should not be transmitted, it must drop it using the following API. The parameter is the frame-descriptor to be dropped, which would usually be “&dqrr->fd”, where “dqrr” is the parameter passed to the packet-processing hook.

```
void ppac_drop_frame(const struct qm_fd *fd);
```

Note that this function is actually implemented as part of the PPAC inline machinery, and so just as the PPAM hook can be inlined into the PPAC packet-processing code, this “drop” action will likewise be inlined into the processing chain, and permit the compiler to look for any possible optimisations.

PPAC Packet Processing - Forward a Frame

If a PPAM packet-processing hook has examined a frame and determined that it should be forwarded, it must use the following API exactly once. The parameters specify the frame-descriptor to be forwarded (see the `ppac_drop_frame()` description above for more info about this parameter), as well as the FQID of the Tx FQ to which it should be forwarded. Please refer back to the “PPAM-provided functions”, and specifically the “Tx FQ enumeration” subsection to recall how a PPAM is notified of the Tx FQIDs for each of the network interfaces. This information provided during initialisation should be organised by the PPAM implementation in order for it to be able to determine target FQIDs when processing packets.

```
void ppac_send_frame(u32 fqid, const struct qm_fd *fd);
```

As with dropping a frame, this function is also implemented by PPAC as an inline, ensuring that it can be compiled (and potentially optimised) into the packet-processing logic.

PPAC Packet Processing - Forwarding Multiple Frames

If a frame needs to be forwarded multiple times (e.g. for multicast), `ppac_send_frame()` must still be called exactly once. Supplementary transmit actions are achieved by using the following API;

```
void ppac_send_secondary_frame(u32 fqid, const struct qm_fd *fd);
```

The parameters are the same as for `ppac_send_frame()`.

PPAC Packet Processing - Dealing with Accelerators

As mentioned in the previous section, the current packet-processing functionality in PPAC provides no support for accelerators. It is likely that future versions of PPAC will provide something in this regard, but for now interactions with accelerators is presumed to be outside the scope of the PPAC infrastructure, and so PPAMs will communicate with accelerators via their driver interfaces. The upshot of this is that any flow-control or other infrastructural support within PPAC will be “blind” to frames and frame queues going between the application and the accelerator hardware.

If a PPAM packet-processing function chooses not to make a drop-or-forward decision immediately, then it must do so at a later time in order to avoid leaking the frame (and all buffer resources associated with it). This delayed transmit action will presumably occur from within the dequeue callback for the FQ that brings the accelerator’s response back from the accelerator.

PPAC Packet Processing - Accelerator Limitations

The use of accelerators, or indeed any other deferral by the PPAM packet-handler to issue a drop-or-forward action “at a later time” infers some limitations on PPAC functions.

- ORP (Order Restoration) support must be disabled in PPAC, the infrastructure for ORP relies on the drop or forward PPAC interfaces being called from within the PPAM’s packet-handling function.
- Order-preservation support must also be disabled in PPAC, for much the same reasons as for ORP.

PPAC Packet Processing - Application Generated Frames

It is possible for applications to generate new frames for transmission rather than strictly forwarding received frames. E.g. the polling hooks available to PPAM (see section 6.2) might be used by the PPAM application to generate test traffic, send work-items to accelerators, respond to IPC requests from other threads or applications, or manage the migration of traffic between USDPAA and non-USDPAA interfaces (e.g. wireless). The limitations described in section 7.2.1 with respect to

accelerators apply here if the transmissions are to go out the USDPAA-managed network interfaces. Moreover, if the transmissions are targeted to those network interfaces, then the frame descriptors must reference buffers that are sources from BMan buffer pools (as the network hardware will autonomously release them back to BMan post-transmission).

PPAC Packet Processing - Application Terminated Frames

Frames that are received via the PPAC-provided packet-handling interface may be fully-consumed by the PPAM application, the assumption is not that they are strictly for forwarding. However, the current PPAC implementation does present some limitations in this regard;

- the frames that are not forwarded must instead be “dropped” from the PPAC perspective (by calling the `ppac_drop_frame()` function), meaning that the BMan buffers within them will be deallocated back to their respective buffer pools.
- if the application intends to use the packet contents after the packet has been “dropped”, its contents must be copied to other memory before doing so. (It is possible that future PPAC enhancements will provide a means to avoid this limitation.)
- if the application intends to delay dropping the frame until it has finished with it, meaning that it will not be dropped before returning from the packet-handling routine, then the same limitations that are mentioned in section 7.2.1 apply here (for the same reasons).

7.4.4.10 PPAC Setting

PPAC-based applications are compiled using a certain set of options that are, for the most part, defined in the header located at `apps/include/ppac.h`. The following sections describe the most useful options for modification, if alternative application behaviour is desired.

PPAC Order Preservation

Order preservation is a functionality of the QMan software portal interface that allows processing of Rx FQs across multiple portals to retain order when transmitted corresponding Tx FQs. The technique only applies to frames dequeued from a given Rx FQ that are all transmitted out the same Tx FQ (which is the case for “reflector”, and also the case for “IPFwd” when frames are from the same flow). The mechanism requires two QMan features, “HOLDACTIVE” and “enqueue DCA”. The former ensures that a FQ that has been dequeued to a software portal from should remain bound to that portal until all the corresponding DQRR entries have been consumed. The latter ensures that a DQRR entry is consumed by QMan itself once it has dispatched the corresponding enqueue (Tx) command. Together, for any given Rx/Tx FQ pair, the processing via pool-channels and multiple CPUs does not allow frame processing to get out of order. (For more information on this feature, consult the QMan/BMan API Guide.)

Use of “HOLDACTIVE” is mutually exclusive with another QMan option “AVOIDBLOCK”, which is selected by default in PPAC. So to enable order-preservation, one must change the settings from :

```
#undef PPAC_HOLDACTIVE
#undef PPAC_ORDER_PRESERVATION
#define PPAC_AVOIDBLOCK
to;
#define PPAC_HOLDACTIVE
#define PPAC_ORDER_PRESERVATION
#undef PPAC_AVOIDBLOCK
```

PPAC Order Restoration

Order restoration is a feature of the QMan hardware that allows frame enqueue operations to be re-assembled in an ORP (“Order Restoration Point”) prior to enqueueing onto the destination FQ. (ORPs are in fact FQ objects in hardware that are configured for use as restoration contexts). The subject of ORP (and its various behavioural parameters) is beyond the scope of this document. Please consult the QMan Reference Manual for more detail.

To enable order restoration, one must change the setting from;

```
#undef PPAC_ORDER_RESTORATION
to;
#define PPAC_ORDER_RESTORATION
```

PPAC Monitoring Rx/Tx Fill-Levels via CGR

If CGR-based monitoring is enabled, then two Congestion Group Records will be configured, with all Rx FQs for all interfaces being subscribed to one, and all Tx FQs being subscribed to the other. The CGRs are not configured to perform any flow-control (i.e. no tail-drop nor WRED options are enabled), so this simply allows the user to monitor the overall fill-level of frame queues in the system, in particular to determine whether build-up is occurring before or after the software-processing phase. The thresholds used for the CGRs are determined from the two constants also defined in `ppac.h`, `PPAC_CGR_RX_PERFQ_THRESH` and `PPAC_CGR_TX_PERFQ_THRESH`. These constants, combined with the number of FQs, define the thresholds for CGR congestion-entry. QMan in turn defines the CGR exit-threshold to be 7/8 the entry-threshold (to provide hysteresis), and so these combined entry/exit events will be logged to `stdout` independently for the Rx and Tx CGRs.

An extra command, “`cgr`”, becomes available in the CLI when this feature is compiled in, which will query and display all the fields of both CGRs. Note however that this option introduces extra latency, and more critically, extra contention within the system. This is because QMan must lock the CGR for each enqueue and dequeue event that relates to it, meaning that the forwarding of a single packet through the system requires 2 lock/unlock pairs for each CGR (thus 4 lock/unlock pairs). A small but noticeable performance degradation should be expected when running in this mode. (Real-world use of CGRs would not subscribe all Rx/Tx FQs from all interfaces to a single CGR, so this scalability issue should not be a concern for production software usage of CGR-based congestion management.)

To enable this feature, change:

```
#undef PPAC_CGR
to;
#define PPAC_CGR
```

PPAC Other Settings

Many other settings used by PPAC (and thus PPAC-based apps) are defined in `ppac.h` but they are less intended for ad-hoc manipulation than those mentioned above. However a curious user may wish to examine some of these, and perhaps search out their usage within the source-code, in order to see how they are used and explore some of the driver interfaces and application design in this way.

7.4.4.11 PPAC Buffers

When starting up, PPAC applications seed the 3 buffer pools that the FMan is configured to use for Rx processing.

To improvement restartability, the pools are first drained of any stale contents, after which they are seeded using allocations from the “`/dev/fsl_usdpaa_shmem`” DMA device, which is described in the USDPAAs User Guide. The buffer pool IDs that must be used, and the size of the buffers they must each hold, is required to match the settings in FMan – these are currently hardcoded in the `include/internal/conf.h` header, and must be kept in-sync with the FMan settings.

The three buffer pools initialized and used by PPAC (and the default USDPAAs configuration of FMan) have the following attributes;

Table 140. Buffer Pool attributes

Buffer Pool ID (BPID)	Buffer Size (used by Fman)	Number of Buffers
7	320	0
8	704	0
9	1728	0x2000

When processing frames, buffers are allocated by FMan on Rx and then released by FMan after Tx.

The current default buffer pool settings from include/internal/conf.h follow;

```
#define DMA_MEM_PATH          "/dev/fsl-usdpaa-shmem"
#define DMA_MEM_BP1_BPID     7
#define DMA_MEM_BP1_SIZE     320
#define DMA_MEM_BP1_NUM      0 /* 0*320==0 (0MB) */
#define DMA_MEM_BP2_BPID     8
#define DMA_MEM_BP2_SIZE     704
#define DMA_MEM_BP2_NUM      0 /* 0*704==0 (0MB) */
#define DMA_MEM_BP3_BPID     9
#define DMA_MEM_BP3_SIZE     1728
#define DMA_MEM_BP3_NUM      0x2000 /* 0x2000*1728==13.5MB */
#define DMA_MEM_BPOOL \
    (DMA_MEM_BP1_SIZE * DMA_MEM_BP1_NUM + \
    DMA_MEM_BP2_SIZE * DMA_MEM_BP2_NUM + \
    DMA_MEM_BP3_SIZE * DMA_MEM_BP3_NUM) /* 13.5MB */
```

And the application implements the seeding of these buffers according to the following structure in apps/ppac/main.c;

```
/* Seed buffer pools according to the configuration symbols */
const struct ppac_bpool_static {
    int bpid;
    unsigned int num;
    unsigned int sz;
} ppac_bpool_static[] = {
    { DMA_MEM_BP1_BPID, DMA_MEM_BP1_NUM, DMA_MEM_BP1_SIZE },
    { DMA_MEM_BP2_BPID, DMA_MEM_BP2_NUM, DMA_MEM_BP2_SIZE },
    { DMA_MEM_BP3_BPID, DMA_MEM_BP3_NUM, DMA_MEM_BP3_SIZE },
    { -1, 0, 0 }
};
```

7.4.4.12 Chronology of a DPAA application

Provides pointers to key locations in the PPAC code where tasks are performed that any USDPAAs-based application would need to implement, in particular for applications that will not be based on PPAC or anything similar.

For more information on each of the named APIs, such as parameters, return codes, semantics, [etc], please consult the relevant documentation (e.g. the “USDPAAs User Guide”, the “Queue Manager, Buffer Manager API Reference Manual” etc.).

DPAA Application - Global Initialization

This section covers the steps performed prior to initialising USDPAAs threads, i.e. prior to binding pthreads to QMan and BMan portals.

DPAA Application - Global Initialization for Device Tree Parsing

The USDPAAs “of” driver needs to parse the device-tree before any dependent driver or application layers are activated. This is performed via of_init(), as seen early in apps/ppac/main.c:main(). This does not do any configuration of devices, it simply parses the information in the device-tree in order to make it available for the various device drivers that will subsequently initialise. (Note that “of” refers to the “Open Firmware” standard upon which u-boot and linux device-tree handling is based, and the APIs for manipulating such device-trees are typically prefixed with “of_” in those environments too.)

DPAA Application - Global Initialization for Network Device Configuration Parsing

The application-level “usdpaa_netcfg” interface parses the device-tree and the two XML files used with FMC to program the FMan devices. This interface is in turn built on top of the lower-level interface associated with the USDPAAs “FMan” driver (and initialisation of “usdpaa_netcfg” will implicitly initialise the “FMan” driver layer). See include/usdpaa/{usdpaa_netcfg.h,fman.h}, which is used by PPAC in apps/ppac/main.c:main() where it calls usdpaa_netcfg_acquire().

USDPAAs apps should choose to bypass the “usdpaa_netcfg” layer and optionally the “FMan” driver layer too if it wishes to implement its own parsing of the configuration, or even obtain the configuration information from another source.

DPAA Application - Global Initialization for QMan and BMan

The USDPAA QMan and BMan drivers currently initialise some global resources using compiled-in constants. This setup is driven by the `qman_global_init()` and `bman_global_init()` APIs. This is done in `apps/ppac/main.c:main()`.

DPAA Application - DMA Memory Initialization

The USDPAA “`dma_mem`” driver initialises and manages access to a memory-mapped region of physically-contiguous memory with trivial virtual/physical address conversion and no page-faulting. This is via the `dma_mem_setup()` API, as called in `apps/ppac/main.c:main()`.

DPAA Application - Thread Initialization

This section covers the setup of USDPAA threads.

DPAA Application - Portal Interrupts

A running thread can request the allocation and configuration of a thread-affine QMan or BMan portal using the `qman_thread_init()/bman_thread_init()` APIs. This is done in `apps/ppac/main.c:worker_fn()`, after the thread has been spawned from `apps/ppac/main.c:worker_new()`. Note that this specifies to the QMan/BMan drivers the CPU that the portals need to be affine to, but it is up the application when (or even if) to bind the thread to the nominated CPU. In the case of PPAC, this is performed immediately prior to initialising the portals, via the `pthread_setaffinity_np()` API.

DPAA Application - Thread-local Enqueue Objects

Rather than using distinct QMan FQ objects for each FQID that it wishes to enqueue to, PPAC instead declares a thread-local, global FQ object for enqueueing to any FQID. This is done in `apps/ppac/main.c:worker_fn()`, search for “`local_fq`”. This means that PPAC and PPAM logic only needs to know and specify the FQIDs of the various Tx FQs for each interface rather than tracking objects for each such FQ. Moreover, it also avoids the sharing of Tx FQ objects across multiple threads (and thus across CPUs) unnecessarily, which is to avoid the potential for cache-coherency overheads. This has its disadvantages too – when PPAC performs an enqueue, it must directly substitute the FQID within the thread-local FQ object. Also, in the event of any ERNs (enqueue rejection notifications), the PPAC-implemented callback cannot rely on the FQ object to identify the FQID (or interface) which relates to the failed enqueue, so would have to interpret that information from the FQID contained in the notification. The PPAC currently does no such demuxing and simply drops any frames that are rejected.

DPAA Application - Portal dequeue mapping of pool-channels

The network interface configuration mentioned earlier, obtained via `usdpaa_netcfg_acquire()`, can be used to identify the QMan pool-channels that the network interfaces' Rx FQs are scheduled to. Using this information, the application can determine the pool-channels it wishes the initialised portal to dequeue from. The PPAC code computes this mask in `apps/ppac/main.c:main()` (search for “`sdqcr`”), and then each thread applies it to its QMan portal in `apps/ppac/main.c:worker_fn()` via the `qman_static_dequeue_add()` API.

DPAA Application - Thread run-to-completion loop

This section covers the run-to-completion loop of a USDPAA application thread once it has been initialised. For USDPAA, the QMan and BMan portals are usually set in run-to-completion mode, meaning that none of the portal interrupt sources are enabled and all portal duties are configured to be processed by polling instead of interrupt-handling. The expectation is therefore that the application thread will repeatedly “poll” the portals within the core-loop of its execution.

DPAA Application - Fast Path Processing

For QMan, the run-to-completion processing is split into two halves. The “fast” part deals with processing of the portal's dequeue ring (DQRR), because this is the most speed-critical mechanism and also the most hardware-optimised. In particular, with portal ring-stashing enabled (as it is by default in USDPAA), processing of this ring rarely involves any cache-stalls, whether there is dequeue work available to be processed or not, so it can be executed quickly and in a tight-loop without any large fixed overheads. (Processing that involves reading a cache-inhibited register, on the other hand, will always have a minimum overhead due to the latency of issuing a read to hardware and waiting for the response.)

So the core loop of PPAC-based application threads call `qman_poll_dqrr()` frequently, such that other kinds of processing are far less frequent. Or put another way, most iterations of the thread's run-to-completion loop do nothing except call `qman_poll_dqrr()`. See `apps/ppac/main.c:main()`, the calls to `qman_poll_dqrr()`.

DPAA Application - Slow Path Processing

For QMan, the other side of run-to-completion processing encompasses “everything else,” i.e. all portal processing except DQRR. This is because polling for other processing duties, even if there is no work to be done, will require a minimal overhead in order to interrogate hardware. In the case of BMan, there is no “fast” path as such, as there is no analogy for DQRR. The speed-critical processing of releasing buffers to (and acquiring buffers from) buffer pools does not require maintenance from the run-to-completion loop (they are command based and self-maintaining).

So PPAC-based application threads calls `qman_poll_slow()` and `bman_poll_slow()` within the run-to-completion loop, but use a throttling scheme to ensure that their overheads are not incurred on every iteration. See `apps/ppac/main.c:main()`.

DPAA Application - Packet Processing

The task of examining a packet (or frame) and determining the appropriate action is triggered by the “Fast path processing” mentioned above. When `qman_poll_dqrr()` is called and dequeued frames are discovered and handled, callbacks will be invoked for the FQ objects to which the dequeued frames belonged. So packet-processing occurs within callbacks registered with FQ objects. In the case of PPAC, the main example for the performance-critical case is in `apps/include/ppac.c:cb_dqrr_rx_hash()`. However, this handler defers all the significant work to the PPAM hook, as indeed that is what PPAMs are - they implement the packet-handling specifics. As can be seen from `cb_dqrr_rx_hash()`, it calls `ppam_rx_hash_cb()` to process the packet. See `apps/reflector/reflector.c:ppam_rx_hash_cb()`. This in turn defers all work to `apps/reflector/reflector.c:reflect_cb()` (because in some configurations, the same processing is performed on the “rx_default” FQ too, so the implementation is factored out).

This packet processing uses `dma_mem_ptov(qm_fd_addr())` to extract the frame address from the frame-descriptor (whether the frame descriptor also specifies an offset or not) and converts it from a device-physical to a usable pointer/virtual address. The packet is then examined, and the outcome is either that the packet is dropped via `ppac_drop_frame()` or it is forwarded via `ppac_send_frame()`.

Note also that in the case of the frame being forwarded, the FQID to which it is transmitted is derived from the PPAM's own FQ state that was initialised during the network interface initialisation phase. In particular, the Tx FQIDs for each interface are captured as the interface initialises, so that when the Rx FQs for the same interface are initialised, the PPAM-specific state immediately makes a fixed assignment for each Rx FQ to one of the Tx FQs. (As a “reflector” this is possible because all packets need to go back out the interface they come in on. This is quite different to “IPFwd” and other forwarding applications of course.)

7.4.4.13 A non-PPAC comparison, “hello_reflector”

A stand-alone version of reflector was created in order to illustrate how to move from a PPAM-based prototype to a stand-alone code-base for development of stand-alone user applications. To emphasise its “hello world” nature, this application is called “hello_reflector”.

“hello_reflector” is located in `apps/hello_reflector/`. Many of the features of the PPAC/PPAM version are missing from the stand-alone version in order to keep it simple (and besides which, those features are provided from the PPAC infrastructure rather than from the reflector PPAM itself, so re-implementing those features in a new, stand-alone form would not illustrate anything useful). The “hello_reflector” code is also intended to be self-documenting, so there is no user guide for it nor will it be discussed in great deal here. However, comparing the reflector PPAM code with `hello_reflector` should make it clear that the packet-processing is fundamentally the same. Furthermore, the single `hello_reflector` C file implements all the initialisation described in section 10 outside of PPAC, so should be instructive for those looking to write stand-alone (non-PPAC) USDPAA applications.

7.4.5 USDPAA Reflector and PPAC User Guide SDK

7.4.5.1 Introduction

The User Space Datapath Acceleration Architecture (USDPAA) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document describes the “reflector” application and the PPAC abstraction on which it is built, both contained in the USDPAA package. This application serves as a reference example for working with the USDPAA interface, as well as providing

a benchmark for USDPAAs system performance. More elaborate PPAC-based applications (such as "ipfwd") have their own user guide documents, but do not repeat the PPAC discussion found here.

Furthermore, a stripped-down "hello_reflector" application exists within USDPAAs that is implemented without using the PPAC abstraction. This application is missing many of the features of PPAC, but is intended to provide a usable comparison between PPAC-based and stand-alone applications, eg. for customers prototyping with PPAC and then looking to develop stand-alone production code. As "hello_reflector" is very simplistic, the source code is self-documenting and it will not be discussed in any detail here. A short guide to executing the "hello_reflector" application will be provided at the end of this document.

A variant of hello_reflector called hello_reflector "short circuit" is also provided which basically tests the sanity of the hardware path for the packet flow by this way without any processing by the core on the received packets.

7.4.5.1.1 Intended audience

This document is intended for software developers and system architects who work with the USDPAAs framework.

The material is technical in nature. The reader is assumed to be familiar with:

- General Linux software development, operation, and configuration for Power architecture devices in particular.
- Familiarity with the concept of device drivers and their role in operating systems.
- Linux network administration and use.
- The NXP Linux SDK for DPAA-based SoCs.
- At least broadly, the DPAA hardware blocks.
- The USDPAAs User Guide document (this document covers only the reflector application).
- Linux UIO (User space I/O) driver infrastructure (USDPAAs drivers for QMan/BMan portals use the standard UIO mechanisms for memory-mapping and interrupt-handling).

The P4080 was the first NXP SoC to incorporate the DPAA. As such, it is used in many examples. However, USDPAAs is intended to apply in the same manner to all DPAA-based SoCs.

7.4.5.1.2 Change history

Table 141. Change History

Version	Updates
1.0	Creation of reflector UG for phase1 release, based on the general UG prepared for the phase0 release. - tidy up of text - describe phase1-specific additions; PPAC/PPAM, IRQ mode, etc.
1.1	Add section on testing reflector using a Linux PC.
1.2	Fix minor typographical errors.
1.3	Updates for v1.0 release. - split PPAC-common material from reflector specifics - corrected and updated buffer pool descriptions
<i>Table continues on the next page...</i>	

Table 141. Change History (continued)

Version	Updates
1.4	Updates for v1.1 release - document "hello_reflector" - document cmd-line args and environment-variables - update sample console output - update buffer definitions

7.4.5.2 Overview of reflector

The reflector application is a simple packet-processing USDPAAs application. The main purpose of USDPAAs user space drivers (as opposed to more conventional kernel-mediated device access) is raw I/O performance, and the reflector application focuses on this. Reflector is basically all I/O - it receives and sends frames, but does very little processing on them. It demonstrates the I/O capabilities of the DPAA hardware and corresponding user-space drivers.

7.4.5.3 Overview of PPAC

The source code to reflector has been reorganized into two parts; the "PPAC" (Packet-Processing Application Core) and a "PPAM" (Packet-Processing Application Module). The idea behind this is that many packet-processing applications (particularly any variations on the theme of network forwarding) would likely only differ from reflector in the way they make their "forwarding decision". The essential difference is the logic that looks at the packet and determines what to do with it. The PPAM portion implements this application specific logic. On the other hand, the PPAC component represents the common infrastructure to support such PPAMs; initializing devices, handling flow-control, implementing a CLI (Command-Line Interface), managing threads and buffers, [etc]. For USDPAAs demonstration applications, the use of a common PPAC component allows for easier code development and maintenance and a uniform application look and feel.

Users are not obliged to use PPAC for their USDPAAs application development. Moreover users could choose to implement PPAC-based applications (eg. for quick prototyping) and then later port their work to a stand-alone product. A stand-alone version of reflector, called "hello_reflector", has been provided to illustrate this principle. Please note that the emphasis in "hello_reflector" is on simplicity, so it does not (re)implement stand-alone equivalents of all the features and flexibility of the PPAC environment.

Additionally, if the user only wants to first check the sanity of the hardware path for the packet flow and later wants to send the packets to the core for the processing, he can first run hello_reflector in "short circuit" mode and if the traffic is received back on the hardware path, he can now run hello_reflector and so involve the core in the packet processing.

In "short circuit" hello reflector, packets are dequeued from the interface and received on Rx FQs. These packets are then sent out from the QMAN channel on which Tx FQs are scheduled.

7.4.5.4 PPAC details

However at an implementation level, there is a limit to the degree of abstraction one can obtain. E.g. for 64 byte packets, using between 1 and 4 CPUs on a p4080 DS, reflector processing averages out to approximately 170 CPU cycles per-packet. Adding just one extra level of indirection to that processing path can add enough overhead to make a visible performance difference to the application (one such ad-hoc experiment showed an additional overhead of ~20 cycles, causing a 12% performance degradation). For this reason, the PPAC/PPAM interface is implemented by a strategic use of inlining. The resulting PPAC implementation and interface is organized in such a way that the compiler is able to inline the fast-path code of PPAC and PPAM together, as though they were written as a single (or "flat") application. Indeed, the PPAM version of reflector, despite the PPAC/PPAM split, has no performance degradation relative to earlier non-PPAM versions.

7.4.5.4.1 IRQ mode for sleeping when idle

A new feature of PPAC (and thus all PPAM applications like reflector and "ipfwd") is support for interrupts and sleeping. As USDPAA's QMan and BMan drivers use the standard Linux UIO subsystem, the behavior of interrupt-handling is in keeping with UIO semantics. Note that the `qman_irqsource_*`() and `bman_irqsource_*`() APIs must be used prior to entering a blocking `read()`, `select()`, `poll()`, [etc] to configure the relevant QMan/BMan portals to report activity via interrupt. Failure to do so may cause the application to block indefinitely waiting for interrupts that won't occur because the portals are configured for polling. Similarly, when leaving IRQ/blocking mode to run in polling mode (and in particular to post-process the events that caused interrupts), one must again use the "irqsource" APIs, this time to configure the portals to be processed by polling APIs rather than interrupts.

For an illustration of PPAC's use of IRQ mode, see `apps/ppac/main.c`, specifically the core loop of the `worker_fn()` function. This loop migrates from polling mode to a blocking "IRQ mode" built around `select()` whenever reflector has looped a certain number of times without any forward progress. If input traffic is significantly below processing capacity, then it should be possible to observe reflector going in to (and out of) blocking `select()`s, even if the traffic is never completely stopped. In such cases, any latency associated with sleeping and scheduling is absorbed by system buffering long enough for reflector to wake up and resume polling mode (and catch up on the buffered backlog). This mechanism allows reflector to share CPUs with other tasks more effectively (and consume less power) provided the throughputs are low or busy enough to make this acceptable.

7.4.5.4.2 Buffers

When starting up, PPAC applications seed the 3 buffer pools that the Fman is configured to use for Rx processing. To improve restartability, the pools are first drained of any stale contents, after which they are seeded using allocations from the `"/dev/fsl_usdpaa_shmem"` DMA device, which is described in the USDPAA User Guide.

The three buffer pools initialized and used by PPAC (and the default USDPAA configuration of Fman) have the following attributes.

Table 142. Buffer Pool attributes

Pool ID	Buffer Size (used by Fman)	Number of Buffers
7	320	0
8	704	0
9	1728	0x2000

When processing IPv4 frames, buffers are allocated by FMan on Rx and then released by FMan after Tx.

The source code specifies the buffer pool requirements in `include/internal/conf.h`:

```

#define DMA_MEM_BP1_BPID      7
#define DMA_MEM_BP1_SIZE     320
#define DMA_MEM_BP1_NUM      0 /* 0*320==0 (0MB) */
#define DMA_MEM_BP2_BPID      8
#define DMA_MEM_BP2_SIZE     704
#define DMA_MEM_BP2_NUM      0 /* 0*704==0 (0MB) */
#define DMA_MEM_BP3_BPID      9
#define DMA_MEM_BP3_SIZE     1728
#define DMA_MEM_BP3_NUM      0x2000 /* 0x2000*1728==13.5MB) */
#define DMA_MEM_BPOOL \
    (DMA_MEM_BP1_SIZE * DMA_MEM_BP1_NUM + \
     DMA_MEM_BP2_SIZE * DMA_MEM_BP2_NUM + \
     DMA_MEM_BP3_SIZE * DMA_MEM_BP3_NUM) /* (13.5MB) */

```

And the application implements the seeding of these buffers according to the following structure in apps/ppac/main.c:

```
/* Seed buffer pools according to the configuration symbols */
const struct ppac_bpool_static {
    int bpid;
    unsigned int num;
    unsigned int sz;
} ppac_bpool_static[] = {
    { DMA_MEM_BP1_BPID, DMA_MEM_BP1_NUM, DMA_MEM_BP1_SIZE },
    { DMA_MEM_BP2_BPID, DMA_MEM_BP2_NUM, DMA_MEM_BP2_SIZE },
    { DMA_MEM_BP3_BPID, DMA_MEM_BP3_NUM, DMA_MEM_BP3_SIZE },
    { -1, 0, 0 }
};
```

7.4.5.4.3 Compile-time configuration

PPAC-based application are compiled using a certain set of options that are currently defined in the header located at apps/include/ppac.h. The following describes the most useful options for modification if alternative application behaviour is desired;

7.4.5.4.3.1 Order preservation

Order preservation is a functionality of the QMan software portal interface that allows processing of Rx FQs across multiple portals to retain order when transmitted corresponding Tx FQs. The technique only applies to frames dequeued from a given Rx FQ that are all transmitted out the same Tx FQ (which is the case for "reflector", and also the case for "ipfwd" when frames are from the same flow). The mechanism requires two QMan features, "HOLDACTIVE" and "enqueue DCA". The former ensures that a FQ that has been dequeued to a software portal from should remain bound to that portal until all the corresponding DQRR entries have been consumed. The latter ensures that a DQRR entry is consumed by QMan itself once it has dispatched the corresponding enqueue (Tx) command. Together, for any given Rx/Tx FQ pair, the processing via pool-channels and multiple CPUs does not allow frame processing to get out of order. (For more information on this feature, consult the QMan/BMan API Guide.)

Use of "HOLDACTIVE" is mutually exclusive with another QMan option "AVOIDBLOCK", which is selected by default in PPAC. So to enable order-preservation, one must change the settings from;

```
#undef PPAC_2FWD_HOLDACTIVE
#undef PPAC_2FWD_ORDER_PRESERVATION
#define PPAC_2FWD_AVOIDBLOCK
```

to;

```
#define PPAC_2FWD_HOLDACTIVE
#define PPAC_2FWD_ORDER_PRESERVATION
#undef PPAC_2FWD_AVOIDBLOCK
```

7.4.5.4.3.2 Monitoring Rx/Tx fill-levels via CGR

If CGR-based monitoring is enabled, then two Congestion Group Records will be configured, with all Rx FQs for all interfaces being subscribed to one, and all Tx FQs being subscribed to the other. The CGRs are not configured to perform any flow-control (ie. no tail-drop nor WRED options are enabled), so this simply allows the user to monitor the overall fill-level of frame queues in the system, in particularly to determine whether build-up is occurring before or after the software-processing phase. The thresholds used for the CGRs are determined from the two constants also defined in ppac.h, PPAC_CGR_RX_PERFQ_THRESH and PPAC_CGR_TX_PERFQ_THRESH. These constants, combined with the number of FQs, define the thresholds for CGR congestion-entry. Qman in turn defines the CGR exit-threshold to be 7/8 the entry-threshold (to provide hysteresis), and so these combined entry/exit events will be logged to stdout independently for the Rx and Tx CGRs.

An extra command, "cli", becomes available in the CLI when this feature is compiled in, which will query and display all the fields of both CGRs. Note however that this option introduces extra latency, and more critically, extra contention within the system. This is because Qman must lock the CGR for each enqueue and dequeue event that relates to it, meaning that the forwarding of a single packet through the system requires 2 lock/unlock pairs for each CGR (thus 4 lock/unlock pairs). A small but noticeable performance degradation should be expected when running in this mode. (Real-world use of CGRs would not subscribe all Rx/Tx FQs from all interfaces to a single CGR, so this scalability issue should not be a concern for production software usage of CGR-based congestion management.)

To enable this feature, change;

```
#undef PPAC_CGR
```

to;

```
#define PPAC_CGR
```

7.4.5.4.3.3 Other settings

Many other settings used by PPAC (and thus PPAC-based apps) are defined in ppac.h but they are less intended for ad-hoc manipulation than those mentioned above. However a curious user may wish to examine some of these, and perhaps search out their usage within the source-code, in order to see how they are used and explore some of the driver interfaces and application design in this way.

7.4.5.5 Running reflector

The following comments about executing reflector apply to all PPAM applications - PPAMs may of course define new command-line (and environment-variable) behaviour, but the following behaviour is all generic to PPAC (reflector does not implement any extensions of its own).

After logging in to the p4080 DS environment as "root", the SOURCE_THIS file in the "/root" directory can be used to simplify running "fmc" (to configure FMan for the required network device configuration) and starting up reflector, as can be seen from the following:

```
login: root
Password:
[root@p4080 root]# cat /root/SOURCE_THIS
cd /usr/etc
fmc -c usdpaa_config_p4_serdes_0xe.xml -p usdpaa_policy_hash_ipv4.xml -a
reflector
[root@p4080 root]# source /root/SOURCE_THIS
Found /fsl,dpaa/dpa-fman0-oh@1, Tx Channel = 46, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@4, Tx Channel = 40, FMAN = 0, Port ID = 0
Found /fsl,dpaa/ethernet@7, Tx Channel = 63, FMAN = 1, Port ID = 2
Found /fsl,dpaa/ethernet@8, Tx Channel = 64, FMAN = 1, Port ID = 3
Found /fsl,dpaa/ethernet@9, Tx Channel = 60, FMAN = 1, Port ID = 0
Qman: FQID allocator includes range 512:128
Bman: BPID allocator includes range 56:8
Configuring for 4 network interfaces and 4 pool channels
FSL dma_mem device mapped (phys=0xe0000000,virt=0x70000000,sz=0x1000000)
Thread uid:0 alive (on cpu 1)
Release 0 bufs to BPID 7
Release 0 bufs to BPID 8
Release 8192 bufs to BPID 9
reflector starting
Thread uid:0 alive (on cpu 1)
reflector>
```

Reflector starts up with a single thread running on CPU 1 by default, with all the network interfaces enabled. The CLI (Command-Line Interface) allows you to add and remove additional threads to enable the use of multiple CPUs, with the only restriction being that the primary thread on CPU 1 can not be removed (except by shutting down the application).

Reflector can alternatively be started as a single thread on a different CPU by executing it with the desired CPU as its sole argument:

```
[root@p4080 root]# reflector 5
```

Alternatively, multiple threads can be started by specifying a range of CPUs:

```
[root@p4080 root]# reflector 3..7
```

By default, reflector is compiled to load the XML configuration and PCD files passed to the "fmc" tool in the afore-mentioned SOURCE_THIS script. Indeed, it is important that reflector always load the same configuration as is passed to "fmc". If "fmc" is run with different XML inputs, eg. when running on another board than p4080ds and/or when using a different SERDES configuration, then reflector must be instructed to load non-default XML files to match. This can be achieved either by setting the DEF_PCD_PATH and DEF_CFG_PATH environment variables, or by using short or long command-line arguments: Eg. the following 3 examples achieve the same thing:

```
[root@p4080 root]# reflector -c my_cfg.xml -p my_pcd.xml

[root@p4080 root]# reflector --fm-config my_cfg.xml --fm-pcd my_pcd.xml

[root@p4080 root]# export DEF_CFG_PATH=my_cfg.xml
[root@p4080 root]# export DEF_PCD_PATH=my_pcd.xml
[root@p4080 root]# reflector
```

Note also that PPAC implements a CLI for all PPAM applications, which means that it expects a console to be present on 'stdin'. If the reflector process is backgrounded, or run as a daemon, then the process will pause waiting for it to receive control of console input. If the user wishes to launch reflector or any other PPAM without the use of the console, they should instruct it ignore input and not run the CLI by passing the "-n" or "--non-interactive" command-line arguments:

```
[root@p4080 root]# reflector --non-interactive &
```

7.4.5.6 PPAC (and reflector) CLI commands

The following commands are illustrated in the context of reflector (carrying on from the session started in the previous section), but the commands are common to all PPAC-based applications.

To add a thread on a single CPU (e.g. CPU 2):

```
reflector> add 2
```

To add threads on a range of CPUs:

```
reflector> add 3..6
```

To list the threads (this also queries each thread, verifying that they aren't blocked):

```
reflector> list
Thread uid:0 alive (on cpu 1)
Thread uid:1 alive (on cpu 2)
Thread uid:2 alive (on cpu 3)
Thread uid:3 alive (on cpu 4)
Thread uid:4 alive (on cpu 5)
Thread uid:5 alive (on cpu 6)
```

To remove a thread by its UID:

```
reflector> rm uid:2
Thread uid:2 killed (cpu 3)
```

To remove a thread running on a given CPU:

```
reflector> rm 4
Thread uid:3 killed (cpu 4)
```

If more than one thread is running on a given CPU (only possible if the device-tree has been updated to reserve multiple portals for USDPAAs on the same CPU), then this removes the first thread found running on the named CPU. A range of CPUs can likewise be specified:

```
reflector> rm 5..6
Thread uid:4 killed (cpu 5)
Thread uid:5 killed (cpu 6)
```

To perform a controlled shutdown of reflector (this includes disabling the network ports):

```
reflector> quit
```

7.4.5.7 Running hello_reflector

The stand-alone "hello_reflector" application, which is not PPAC-based, can be executed in exactly the same environment as the PPAC-based "reflector". Instead of executing "reflector", one executes "hello_reflector" instead:

```
[root@p4080 root]# hello_reflector
```

Hello_reflector always starts up on CPU 0, and by default it only starts a single thread. Multiple threads can be started by providing the "-n" argument, in which case that many CPUs will be used starting with CPU 0 and counting upwards.

```
[root@p4080 root]# hello_reflector -n 5
```

Hello_reflector does not implement a CLI, so the only way to gracefully shut it down is to enter a Ctrl-C on the controlling console, or equivalently issue a SIGINT signal to the hello_reflector process.

Alternative configuration and PCD files can be provided to hello_reflector by using the "-c" and "-p" arguments respectively. These should always match the XML files that are passed to "fmc":

```
[root@p4080 root]# hello_reflector -p my_pcd.xml -c my_cfg.xml
```

7.4.5.8 Running hello_reflector (short circuit)

Issue following command to run hello_reflector in "short circuit" mode:

```
[root@p4080 root]# hello_reflector -c /usr/etc/usdpaa_config_p4_serdes_0xe.xml -
p /usr/etc/usdpaa_policy_hash_ipv4.xml -n 1 -sc
```

To test sc mode, stop the process (ctrl + Z) and feed the traffic to FMAN port (using spirent), you will still see the rx traffic due to the fact that the destination channel for RX queues is tx_channel.

Follow these steps precisely :

1. Hang up process by ctrl+z, this will avoid using cpu and will not impact traffic flow.
2. To quit process, you have to recover process by 'fg %1' then press 'ctrl+c' to quit.

7.4.5.9 Testing reflector

Functional testing of the "reflector" application is possible by connecting any subset of the P4080 USDPAAs network interfaces to a conventional computer, assumed to be a Linux PC using an ethernet switch as shown in the figure below.

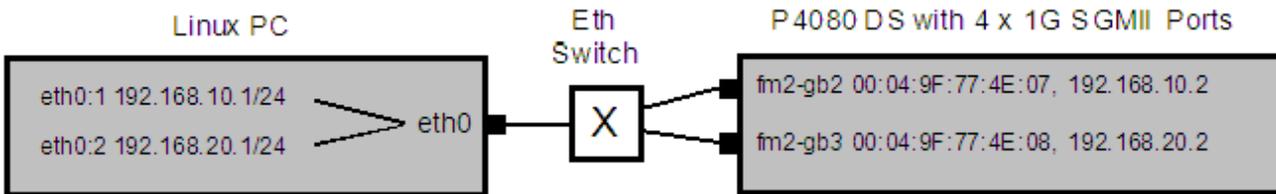


Figure 141. Testing Reflector

The figure assumes that the default USDPAAs SerDes 0xe reset configuration word (RCW) is used. This configuration provides 2 x 10 Gbps and 2 x 1 Gbps ethernet interfaces to the USDPAAs application. The test case in the figure assumes that only the 2 x 1 Gbps interfaces will be used. These are the two interfaces on the SGMII riser card that are closest to the P4080DS motherboard. This test will work even if 10 Gbps XAUI riser cards are not fitted in the P4080DS.

See the main USDPAAs User Guide for more information on network interfaces.

To perform the test, boot the P4080 and run the example application as described in section [Running reflector](#) on page 911.

The IP and ethernet MAC addresses used below are examples. You can change them as long as you are consistent. The most important thing is to be sure of the MAC addresses on the P4080DS board. Again, see the main USDPAAs User Guide.

On the Linux PC, create eth0:1 through eth0:3 via, for example:

```
sudo ifconfig eth0:1 192.168.10.1 netmask 255.255.255.0
sudo ifconfig eth0:2 192.168.20.1 netmask 255.255.255.0
```

The reflector application does not respond to ARP requests, so static ARP entries for all of the P4080 USDPAAs interfaces must be created on the Linux PC:

```
sudo arp -s 192.168.10.2 00:04:9F:77:4E:07
sudo arp -s 192.168.20.2 00:04:9F:77:4E:08
```

Then, from the Linux PC ping one of the P4080 interfaces, e.g. "ping 192.168.10.2". This causes the following sequence:

1. Linux PC sends ICMP echo request to the switch.
2. At least for the first ping, the switch will flood the request to all P4080 ports. It will be dropped by all but the correct one.
3. The P4080 receives the frame, swaps the IP and MAC addresses, and sends the frame back out the MAC it came in on.
4. The Linux PC receives the frame, an ICMP echo request.
5. The Linux PC responds to the request.
6. The response is received by the P4080 which swaps the IP and MAC addresses and sends the frame back out on the MAC it came in on.
7. The Linux receives the response.
8. The original ping is now complete. The Linux PC just "pinged itself via the P4080".

It is possible to also test with different packet types using this same technique. For example, have a telnet server running on the Linux PC and then do "telnet 192.168.10.2" from the Linux PC. The Linux PC will telnet to itself via the P4080. One can also use ssh in the same way.

To test performance, it is best to use dedicated ethernet traffic generation equipment such as a Spirent Test Center. Such equipment can inject packets into the P4080 at known and very high rates and measure the rate of the packets that egress from the P4080.

7.4.6 NXP USDPAA IPFWD User Manual Rev. 1.2

7.4.6.1 About this Book

The User Space Datapath Acceleration Architecture (USDPAA) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document provides the following:

- A summary of the USDPAA IPFwd application
- Execution steps for USDPAA IPFwd application from the NXP yocto package on the P4080DS/P3041DS/P5020DS/T4240QDS/B4860QDS/B4420QDS.

Conventions

This document uses the following conventions:

Courier is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.

7.4.6.2 Introduction

7.4.6.2.1 Purpose

This document describes the USDPAA IPv4 forwarding application. This application documents the USDPAA IPv4 demonstration applications forwarding flow and its performance measured on a P4080DS.

7.4.6.3 Overview

The USDPAA IPv4 forwarding (IPFwd) application is a multi-threaded application that routes IPv4 packets from one ethernet interface to another on all QorIQ platforms. The routing is done based on the source IP address and destination IP address in the frame. Any combination of the cores can run a USDPAA IPv4 Forwarding application thread in USDPAA SDK v1.1.

IPFwd packet-processing:

- Receives ethernet frames on all USDPAA ethernet interfaces.
- Discards (and deallocates buffers for) all non-IPv4 frames.
- For IPv4 frames processing takes place as defined in section [Overview of packet flow](#): on page 916

7.4.6.3.1 USDPAA IPv4 forwarding application flow

The IPFwd application has two main phases. There is an initial configuration phase and a subsequent packet processing phase.

The configuration phase executes when the application starts. Application threads are created and global initialization of resources is done which is the part of PPAC (see USDPAA PPAC User Guide for more details). The configuration phase also includes PPAM (i.e. IPFwd) related initialization. Once the configuration phase is completed the IPFwd application moves to the packet processing phase. This application provides a command-line interface to enable users to add and remove routing table and ARP cache entries at any given time. For each user input, the appropriate information is communicated to the IPFwd application via standard Posix IPC. Messages are placed onto the message queue till they are received by the IPFwd application. Note that the IPFwd application does not dynamically resolve ARP – missing ARP entries will result in the application dropping the packet.

In the packet processing phase, the loop migrates from polling mode to a blocking “IRQ mode” whenever IPFwd has looped a certain number of times without any forward progress. For more information on IRQ mode please refer to “USDPAA PPAC User Guide.” In polling mode – the application constantly looks for data to process on its dedicated QMan portal . Network traffic is classified and distributed by the FMan to frame queues based on source and destination IP address in the packet. There is an associated handler that processes the packets arriving on each frame queue.

7.4.6.3.1 Overview of packet flow:

1. Route table entries are populated using the IPFwd configuration commands at any given time.
2. Packet is received by the FMan, which uses 2-tuple (Src IP & Dest IP) to hash the packet to a Rx frame queue.
3. The classified packet is presented to the USDPAA IPFwd application thread running on one of the cores - the distribution is based on the portal and FQ setup done by the application during its portal and frame queue initialization. The packet is subjected to IPv4 Forwarding route lookup.
4. Based on the packet destination, the application transmits the packet by enqueueing it on a frame queue destined for the appropriate Tx interface.

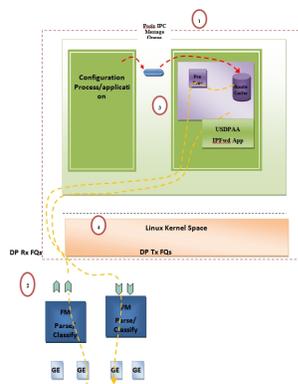


Figure 142. Packet flow for USDPAA IPv4 Forwarding

7.4.6.4 Overview of PPAC

The source code to IPFwd has been reorganized into two parts; the “PPAC” (Packet-Processing Application Core) and a “PPAM” (Packet-Processing Application Module). The PPAM portion implements the IPFWD application specific logic of processing the packet to forward it. On the other hand, the PPAC component represents the common infrastructure to support PPAM; initializing devices, handling flow-control, implementing a CLI (Command-Line Interface), managing threads and buffers. PPAC details can be found in the document “USDPAA PPAC User Guide”.

7.4.6.5 IPFwd related PPAC Details

7.4.6.5.1 Compile-time configuration

PPAC-based application are compiled using a certain set of options that are currently defined in the header located at apps/include/ppac.h. The following describes the most useful options for modification if alternative application behaviour is desired.

7.4.6.5.1.1 Order Preservation in IPFWD

This section describes how user can enable Order Preservation in IPFWD application. By default Order Preservation is disabled in IPFWD application and in order to enable it the user will have to re-compile the binary by making following changes to the source code.

In file, usdpaa/apps/include/ppac.h you can find these two lines.

```
/* Application options */
```

```
#undef PPAC_2FWD_HOLDACTIVE /* Process each FQ on one portal at a time */
#undef PPAC_2FWD_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */
Change the above to
#define PPAC_HOLDACTIVE /* Process each FQ on one portal at a time */
#define PPAC_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */
```

And then compile usdpaa once again. Now run the IPFWD application with Order Preservation.

7.4.6.5.1.2 Order Restoration in IPFWD

Order restoration is the functionality of QMan software portal interface which restores the relative temporal order of a flow of frames (sequence of frames) to that observed before transmitting to the destination Frame Queue and Order Definition Point (ODP) takes note of the correct order of packets before start processing by using the sequence number. Use of "HOLDACTIVE" is mutually exclusive with another QMan option "AVOIDBLOCK", which is selected by default in PPAC. To enable order-restoration, the user will have to re-compile the binary by making following changes to the source code.

```
/* Application options */
#undef PPAC_2FWD_HOLDACTIVE /* Process each FQ on one portal at a time */
#undef PPAC_2FWD_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */
#undef PPAC_2FWD_ORDER_RESTORATION /* Use ORP */
#define PPAC_2FWD_AVOIDBLOCK /* No full-DQRR blocking of FQs */
```

Change the above to

```
#undef PPAC_HOLDACTIVE /* Process each FQ on one portal at a time */
#undef PPAC_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */
#define PPAC_ORDER_RESTORATION /* Use ORP */
#define PPAC_AVOIDBLOCK /* No full-DQRR blocking of FQs */
```

And then compile usdpaa once again. Now run the IPFWD application with Order Restoration. Implementation note: Order restoration has been implemented such that each PCD Frame queue has a corresponding ORP (order restoration point) frame queue associated with it. Each ORP is configured with default window settings as seen below

```
#define PPAC_ORP_WINDOW_SIZE 7 /* 0->32, 1->64, 2->128, ... 7->4096 */
#define PPAC_ORP_AUTO_ADVANCE 1 /* boolean */
#define PPAC_ORP_ACCEPT_LATE 3 /* 0->no, 3->yes (for 1 & 2->see RM) */
```

Here the ORP window size is set to be 4K, auto advance window size as 4K and accept late arrival window size as 8K. This ensures that no traffic is getting dropped but are always accepted below and at Zero loss throughput. Beyond zero loss throughput, as usual packets would be dropped and thus you can see mis-ordering.

ORP FQ descriptor attributes settings:

- Prefer in cache
- No "HOLDACTIVE"
- No "AVOIDBLOCK"
- ORP enabled

Assumption: To see the effect of Order Restoration in IPFwd application the user must use separate streamblocks as a source of traffic. If not done so, mis-ordering would be seen.

Key observation: It has been observed in IPFwd application that use of "HOLDACTIVE" with traffic generated using separate streamblocks, all the packets are IN sequence. Therefore, it is recommended that if user wants to see the real effect of Order

restoration in IPFwd application he should use “AVOIDBLOCK” with “RESTORATION” and not “HOLDACTIVE” with “RESTORATION”

7.4.6.5.1.3 Monitoring Rx/Tx fill-levels and flow-control via CGR

If CGR-based monitoring is enabled, then two Congestion Group Records will be configured, with all Rx FQs for all interfaces being subscribed to one, and all Tx FQs being subscribed to the other. This simply allows the user to monitor the overall fill-level of frame queues in the system, in particular to determine whether build-up is occurring before or after the software-processing phase. Refer to section “Monitoring Rx/Tx fill-levels via CGR ” of USDPAA PPAC User Guide for more details. To enable this feature, in ppac.h change;

```
#undef PPAC_CGR
```

```
to;
```

```
#define PPAC_CGR
```

The CGRs can also be configured to perform flow-control using Congestion state tail drop by setting CSTD_EN bits . Each congestion group record can be configured to track either byte counts or frame counts in all frame queues in the Congestion Group. When the threshold set for each CGR is exceeded, the CS bit is set in the CGR, and the congestion group is said to have entered congestion. At this point the incoming frames are marked for discard and QMAN will generate enqueue rejections to the producer. When the group’s I_BCNT returns below the threshold (minus approximately 1/8 of the threshold to provide hysteresis), the CS bit is cleared, and the congestion group’s state exits congestion. To enable tail drop, in ppac.h change;

```
#undef PPAC_CGR          /* Track rx and tx fill-levels via CGR */
```

```
#undef PPAC_CSTD         /* CGR tail-drop */
```

```
#undef PPAC_CSCN        /* Log CGR state-change notifications */
```

```
to
```

```
#define PPAC_CGR          /* Track rx and tx fill-levels via CGR */
```

```
#define PPAC_CSTD         /* CGR tail-drop */
```

```
#undef PPAC_CSCN        /* Log CGR state-change notifications */
```

And then compile usdpaa once again. Now run the IPFWD application with CGR tail drop enabled. To test this feature PPAC CLI provides a command “cgr” which will query and display all the fields of both CGRs. On pumping the traffic to IPFWD application at full line rate, the instantaneous group byte count value I_BCNT(Instantaneous frame/byte count) must be maintained lesser than the CGR threshold set for each congestion group. Here is one such cgr command output:

```
> cgr
```

```
Rx CGR ID: 10, selected fields;
```

```
cscn_en: 0
```

```
cscn_targ: 0x00800000
```

```
cstd_en: 1
```

```
cs: 0
```

```
cs_thresh: 0x00_0000_1000
```

```
mode: 1
```

```
i_bcnt: 0x00_0000_0e1e
```

```
a_bcnt: 0x00_0000_0e1e
```

```
Tx CGR ID: 11, selected fields;
```

```
cscn_en: 0
```

```
cscn_targ: 0x00800000
cstd_en: 1
cs: 0
cs_thresh: 0x00_0000_0200
mode: 1
i_bcnc: 0x00_0000_0002
a_bcnc: 0x00_0000_0004
```

On the other hand if this feature of Congestion Group tail drop is disabled in IPFWD application I_BCNT is never maintained below CGR threshold value with traffic at full line-rate. This can be checked by compiling the IPFWD application with;

```
#define PPAC_CGR          /* Track rx and tx fill-levels via CGR */
#undef PPAC_CSTD         /* CGR tail-drop */
#undef PPAC_CSCN        /* Log CGR state-change notifications */
```

Now on pumping traffic at full line-rate, atleast one of the CGRs must go into congestion state and its I_BCNT should be above CGR threshold value. Here is the sample output:

```
> cgr
Rx CGR ID: 10, selected fields;
cscn_en: 1
cscn_targ: 0x00800000
cstd_en: 0
cs: 0
cs_thresh: 0x00_0000_1000
mode: 1
i_bcnc: 0x00_0000_0006
a_bcnc: 0x00_0000_0004
Tx CGR ID: 11, selected fields;
cscn_en: 1
cscn_targ: 0x00800000
cstd_en: 0
cs: 1
cs_thresh: 0x00_0000_0200
mode: 1
i_bcnc: 0x00_0000_5fb7
a_bcnc: 0x00_0000_5fb8
```

Thus, the above test showcases the flow control achieved by enabling congestion group tail drop in IPFWD application.

7.4.6.6 PPAM related compile time configuration

7.4.6.6.1 One million route support

By default IPfwd application can work only with 1K routes. The user may want to run it for higher number of routes, so there is an option available in the header located at apps/ipfwd/include/app_common.h. To enable one million route support, the user will have to re-compile the binary by making following changes to the source code.

```
#undef ONE_MILLION_ROUTE_SUPPORT
```

Change the above line to

```
#define ONE_MILLION_ROUTE_SUPPORT
```

And then compile usdpaa once again. Now run the IPFWD application for one million routes. There is a sample shell script available at /usr/bin/ipfwd_20G_1Mroutes.sh in the rootfs. It creates one million routes using the 2 x 10G interfaces. It can assign ip addresses to the interfaces, add an ARP entry and assumes the netmask to be 255.255.255.0. The following table summarizes the setting done by this script.

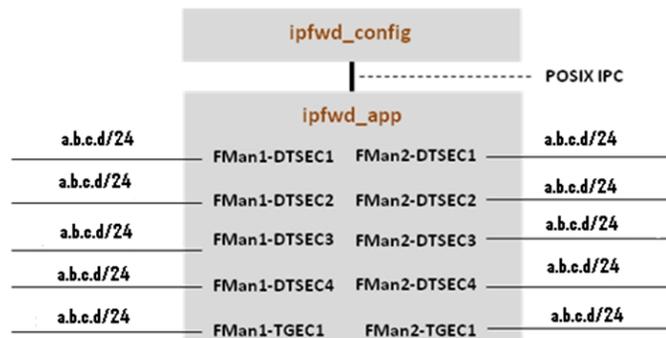
Port ID	Source IP Address	Destination IP Address (for each src IP addr)	Default Gateway IP Address
FM1:TGEC1	192.168.24.2 to 192.168.24.255	192.168.29.2 .. 255	192.168.29.2
	192.168.25.2 to 192.168.25.255	192.168.29.2 .. 255	192.168.29.2
	192.168.26.2 to 192.168.26.255	192.168.29.2 .. 255	192.168.29.2
	192.168.27.2 to 192.168.27.255	192.168.29.2 .. 255	192.168.29.2
	192.168.28.2 to 192.168.28.255	192.168.29.2 .. 255	192.168.29.2
	192.168.1.2 to 192.168.1.255	192.168.29.2 .. 255	192.168.29.2
	192.168.18.2 to 192.168.18.255	192.168.29.2 .. 255	192.168.29.2
	192.168.30.2 to 192.168.30.255	192.168.29.2 .. 255	192.168.29.2
	192.168.31.2 to 192.168.31.91	192.168.29.2 .. 91	192.168.29.2
FM2:TGEC2	192.168.29.2 to 192.168.29.255	192.168.24.2 .. 255	192.168.24.2
	192.168.2.2 to 192.168.2.255	192.168.24.2 .. 255	192.168.24.2
	192.168.3.2 to 192.168.3.255	192.168.24.2 .. 255	192.168.24.2
	192.168.4.2 to 192.168.4.255	192.168.24.2 .. 255	192.168.24.2
	192.168.5.2 to 192.168.5.255	192.168.24.2 .. 255	192.168.24.2
	192.168.6.2 to 192.168.6.255	192.168.24.2 .. 255	192.168.24.2
	192.168.17.2 to 192.168.17.255	192.168.24.2 .. 255	192.168.24.2
	192.168.19.2 to 192.168.19.255	192.168.24.2 .. 255	192.168.24.2

Table continues on the next page...

Table continued from the previous page...

	192.168.20.2 to 192.168.20.91	192.168.24.2 .. 91	192.168.24.2
--	----------------------------------	--------------------	--------------

7.4.6.7 IPFWD Application Suite



The figure above shows the structure of the IPFWD USDPAA application suite. Its purpose is to forward IPv4 packets between the interfaces shown. No IP address is assigned to any of the interfaces by default. Using `ipfwd_config` command as mentioned in section [Assign IP address to interfaces](#) on page 938 IP address can be assigned to all these interfaces. Each interface has a fixed netmask shown in the figure. The notation "/24" refers to a netmask of 255.255.255.0. The MAC addresses of these interfaces are determined by u-boot environment variables `ethaddr`, `eth1addr`, `eth2addr`, etc.

The `ipfwd` application will respond to ARP requests on these interfaces. However, the application will not generate ARP requests to determine the destination MAC address of forwarded packets. An `ipfwd_config` command should be used to set these MAC addresses.

It is important to understand that the application can use only a subset of these interfaces at any one time. This is due to pin multiplexing rules on the P4080 SoC. The set of available interfaces is determined by a number of factors:

1. The interface set made available by the selected SerDes Protocol and u-boot variable "hwconfig". See the SDK document "NXP DPAA SDK X.Y: Selecting Ethernet Interfaces". This document is distributed with the DPAA SDK.
2. The Linux device tree determines which subset of the available interfaces is for use by USDPAA applications. See the USDPAA User Guide for more information.
3. Finally, the `fmc` configuration file passed by command line argument to `ipfwd_app` determines the subset of these interfaces that the application will attempt to initialize.

In the default SerDes 0xe example, the interfaces used by `ipfwd_app` are:

- FMan1-TGEC1
- FMan2-DTSEC3
- FMan2-DTSEC4
- FMan2-TGEC1

Running the `ipfwd` application suite involves four steps:

1. Run the `fmc` application to configure the FMan hardware instances.
2. Run `ipfwd_app`
3. Run `ipfwd_config` repeatedly to add routes to `ipfwd`'s route cache.
4. Run "`ipfwd_config -O`" to start application processing.

Specific examples showing these steps are provided in other sections of this document.

7.4.6.8 Possible configuration scenario for IPFWD

IPfwd application can run in different configuration scenario. The table below shows the configuration files and corresponding sample shell script existing in the repository.

xml file	Sample shell script	Number of routes (as per sample shell script)
usdpaa_config_p4_serdes_0xe.xml (2x10G+2x1G)	ipfwd_20G.sh	1012 routes (2x10G)
	ipfwd_22G.sh	1022 routes (2x10G+2x1G)
usdpaa_config_p3_p5_serdes_0x36.xml (5x1G+10G)	ipfwd_15G.sh	1000 routes (5x1G+10G)



Figure 1: usdpaa_config_p4_serdes_0xe.xml

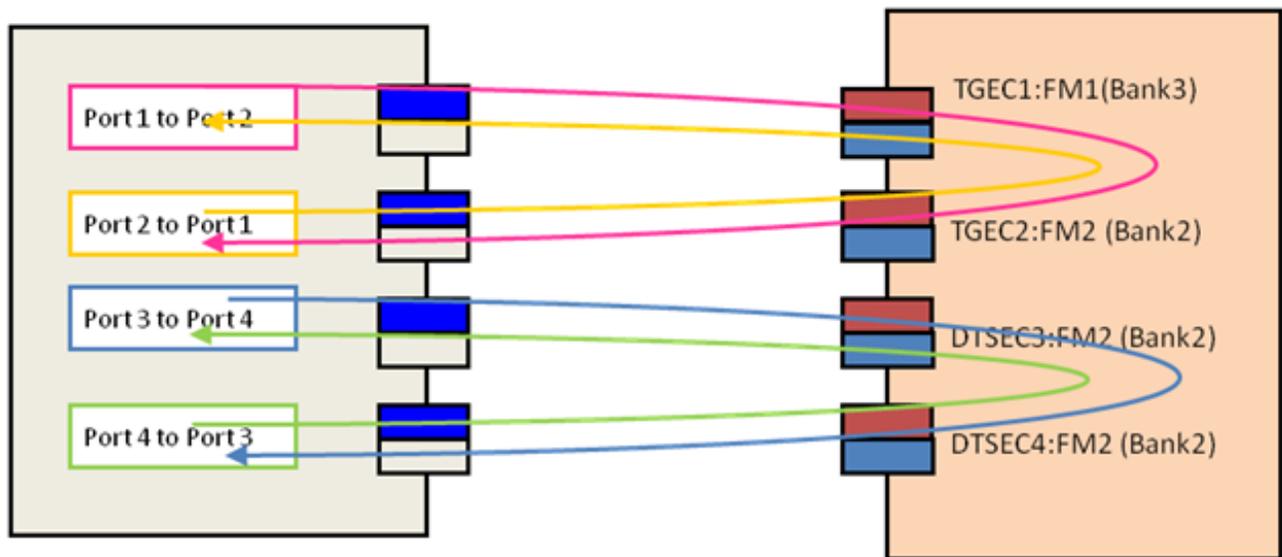
The figure above shows the ethernet interfaces available as per configuration file “ usdpaa_config_p4_serdes_0xe.xml”. It contains the 1 RGMII port (FMAN1, DTSEC 2), 2 SGMII ports (FMan2, DTSECs 3- 4) and 2 XAUI ports (FMAN1, TGEC1 and FMAN2-TGEC2). It is the user’s wish to use any combination of ports available with the configuration file. The above table shows the sample shell scripts that user can use for this configuration. If user uses 2x10G, following is the flow configuration.

Test Center <-> P4080



Port 1 to Port 2 (506 flows):
 Src: 192.168.60.2 - 192.168.60.23
 Dst: 192.168.160.2 - 192.168.160.24
 Port 2 to Port 1 (506 flows):
 Src: 192.168.160.2 - 192.168.160.23
 Dst: 192.168.60.2 - 192.168.60.24

Figure 2 : Flow Configuration for 2x10G on P4080DS

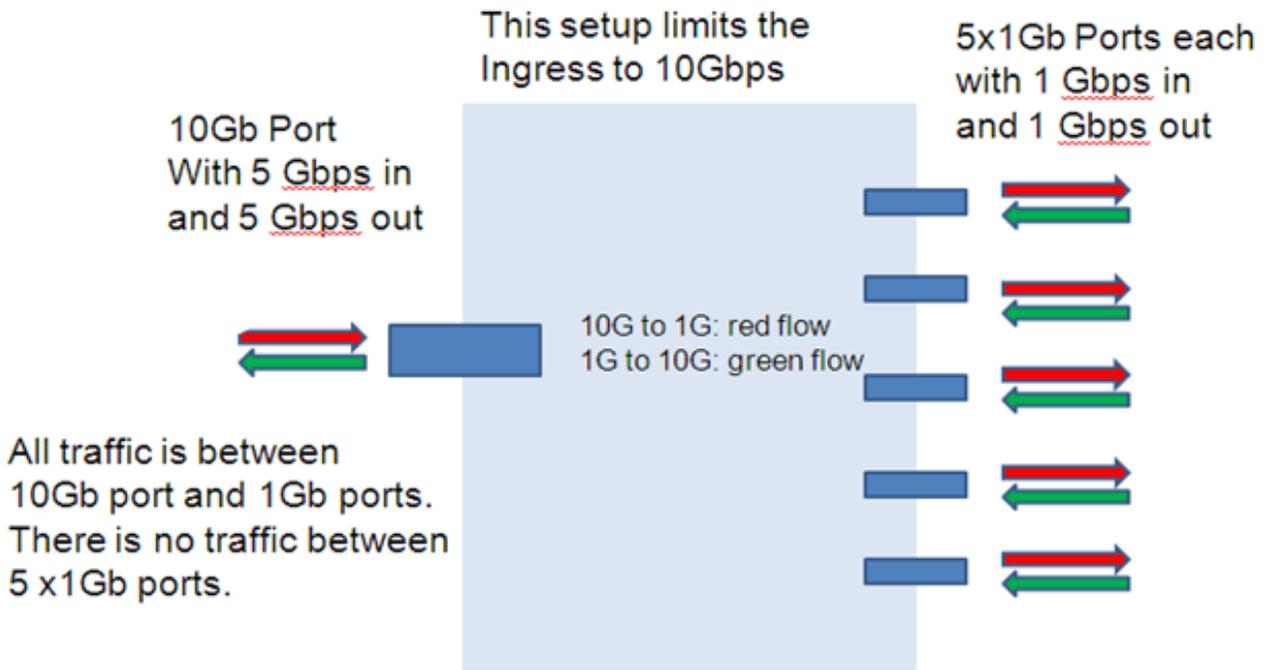


```

Port 1 to Port 2 (462 flows):
Source: 192.168.60.2 - 192.168.60.22
Destination: 192.168.160.2 - 192.168.160.23
Port 2 to Port 1 (462 flows) :
Source: 192.168.160.2 - 192.168.160.22
Destination: 192.168.60.2 - 192.168.60.23
Port 3 to Port 4 (46 flows):
Source: 192.168.130.2 - 192.168.130.22
Destination: 192.168.140.2 - 192.168.140.23
Port 2 to Port 1 (46 flows) :
Source: 192.168.140.2 - 192.168.140.22
Destination: 192.168.130.2 - 192.168.130.23
  
```

Figure 3 : configuration for 2x10G and 2x1G on P4080

For running IPFwd on P3041DS/P5020DS, the user can use " usdpaa_config_p3_p5_serdes_0x36.xml". Following is the flow configuration.



Left: 10G port (port6: fm1-10g)
Right: top-to-bottom: 1G port (port1-port6)

Figure 4 : Flow Configuration for 10G on P5020DS and P3041DS

```
Port 1 to Port 6 (100 flows):
  Src: 192.168.10.2 - 192.168.10.11
  Dst: 192.168.60.2 - 192.168.60.11

Port 2 to Port 6 (100 flows):
  Src: 192.168.20.2 - 192.168.20.11
  Dst: 192.168.60.2 - 192.168.60.11

Port 3 to Port 6 (100 flows):
  Src: 192.168.30.2 - 192.168.30.11
  Dst: 192.168.60.2 - 192.168.60.11

Port 4 to Port 6 (100 flows):
  Src: 192.168.40.2 - 192.168.40.11
  Dst: 192.168.60.2 - 192.168.60.11

Port 5 to Port 6 (100 flows):
  Src: 192.168.50.2 - 192.168.50.11
  Dst: 192.168.60.2 - 192.168.60.11

Port 6 to Port 1 (100 flows):
  Src: 192.168.60.2 - 192.168.60.11
  Dst: 192.168.10.2 - 192.168.10.11

Port 6 to Port 2 (100 flows):
  Src: 192.168.60.2 - 192.168.60.11
  Dst: 192.168.20.2 - 192.168.20.11

Port 6 to Port 3 (100 flows):
  Src: 192.168.60.2 - 192.168.60.11
  Dst: 192.168.30.2 - 192.168.30.11

Port 6 to Port 4 (100 flows):
  Src: 192.168.60.2 - 192.168.60.11
  Dst: 192.168.40.2 - 192.168.40.11

Port 6 to Port 5 (100 flows):
  Src: 192.168.60.2 - 192.168.60.11
  Dst: 192.168.50.2 - 192.168.50.11
```

How to run if user has no XAUI but only SGMII card?

Assume user does not have a XAUI card and wants to run IPFwd using only the SGMII ports as shown below.



This configuration contains the 1 RGMII port (FMAN1, DTSEC 2), 4 SGMII ports (FMan2, DTSECs 1- 4). The user can modify the configuration file and sample shell scripts as per requirement. The configuration file would contain the following

```
<cfgdata>
<config>
<engine name="fm1">
<port type="1G" number="0" policy="hash_ipsec_src_dst_spi_policy6"/>
<port type="1G" number="1" policy="hash_ipsec_src_dst_spi_policy7"/>
<port type="1G" number="2" policy="hash_ipsec_src_dst_spi_policy8"/>
<port type="1G" number="3" policy="hash_ipsec_src_dst_spi_policy9"/>
</engine>
</config>
</cfgdata>
```

User can create a new shell script which would make routes between the 4x1G. Here is the content of the script.

```
net_pair_routes()
{
for net in $1 $2
do
ipfwd_config -P $pid -B -s 192.168.$net.2 -c $3 \
-d 192.168.$(expr $1 + $2 - $net).2 -n $4 -g \
192.168.$(expr $1 + $2 - $net).2
done
}
ipfwd_config -P $pid -F -a 192.168.110.1 -i 6
```

```

ipfwd_config -P $pid -F -a 192.168.120.1 -i 7
ipfwd_config -P $pid -F -a 192.168.130.1 -i 8
ipfwd_config -P $pid -F -a 192.168.140.1 -i 9
ipfwd_config -P $pid -G -s 192.168.110.2 -m 02:00:c0:a8:6e:02 -r true
ipfwd_config -P $pid -G -s 192.168.120.2 -m 02:00:c0:a8:78:02 -r true
ipfwd_config -P $pid -G -s 192.168.130.2 -m 02:00:c0:a8:82:02 -r true
ipfwd_config -P $pid -G -s 192.168.140.2 -m 02:00:c0:a8:8c:02 -r true
# 1024

net_pair_routes 110 120 16 16 # 2 * 16 * 16 = 512
net_pair_routes 130 140 16 16 # 2 * 16 * 16 = 512

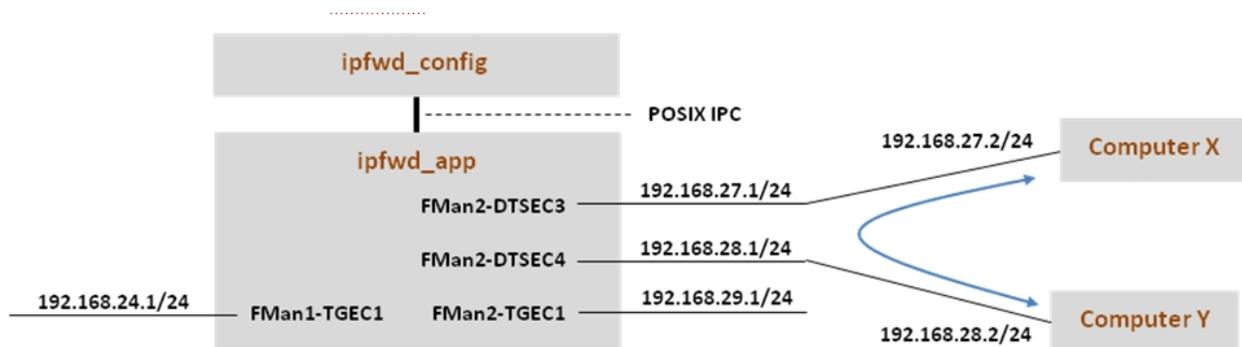
echo IPFwd Route Creation completed

Now traffic can be run as per the routes created.

```

7.4.6.9 Using Two Computers to Test the IPFWD Application Suite

This section describes how to configure the IPFWD application suite to forward packets between two ordinary computers as shown in the following figure. It is assumed that the two 1 Gbps Ethernet interfaces provided by SerDes 0xe are used.



Assign IP addresses to all the interfaces. The IP addresses used here for Computer X and Computer Y are examples. Their MAC addresses must be known as they will be needed in the commands below.

Keep in mind that ipfwd_app has no default IP address assigned to the interfaces. This means that you must first assign IP addresses to the interfaces and then use IP addresses for Computer X and Computer Y that are on the subnets shown in the figure. Please be careful with the network parameters. Any mistake will prevent packets from flowing from end to end.

Follow these steps on Computer X:

1. Set the ip address for eth0 interface

```
ifconfig eth0 192.168.27.2 up
```

Set the default gateway for subnet 192.168.27.1

```
route add default gw 192.168.27.1 eth0
```

2. Add arp for gw

```
arp -s 192.168.27.1 "MAC address for 192.168.27.1 on P4080"
```

Follow these steps on Computer Y:

1. Set the ip address for eth0 interface

```
ifconfig eth0 192.168.28.2 up
```

Set the default gateway for subnet 192.168.28.1

```
route add default gw 192.168.28.1 eth0
```

2. Add arp for gw

```
arp -s 192.168.28.1 "MAC address for 192.168.28.1 on P4080"
```

The commands to perform on P4080 are:

```
# Boot the P4080 and login as root.
```

```
Assign IP address to fm1-gb1
```

```
ifconfig fm1-gb1 <IPADD> up
```

```
cd /usr/etc
```

```
fmc -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_32_fq.xml -a
```

```
# Assume use of cores 1 - 7 ipfwd_app 1..7
```

```
# Now ssh to P4080 linux on other terminal
```

```
ssh root@<IPADD>
```

give IP address as assigned to fm1-gb1 in the beginning

```
# Now assign ip address to the interfaces
```

```
# First run command to check what all enabled interfaces are available
```

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
ipfwd_config -P 2536 -E -a true
```

```
Interface number: 11
```

```
PortID=1:5 is FMan interface node
```

```
with MAC Address
```

```
02:00:c0:a8:65:fe
```

```
Interface number: 9
```

```
PortID=1:3 is FMan interface node
```

```
with MAC Address
```

```
02:00:c0:a8:5b:fe
```

```
Interface number: 8
```

```
PortID=1:2 is FMan interface node
```

```
with MAC Address
```

```
02:00:c0:a8:51:fe
```

```
Interface number: 5
```

```
PortID=0:5 is FMan interface node
```

```
with MAC Address
```

```
02:00:c0:a8:33:fe
```

```
Are all the Enabled Interfaces
```

```
# Assign IP address to the interfaces. Use the same interface number displayed as an output on giving the above command
```

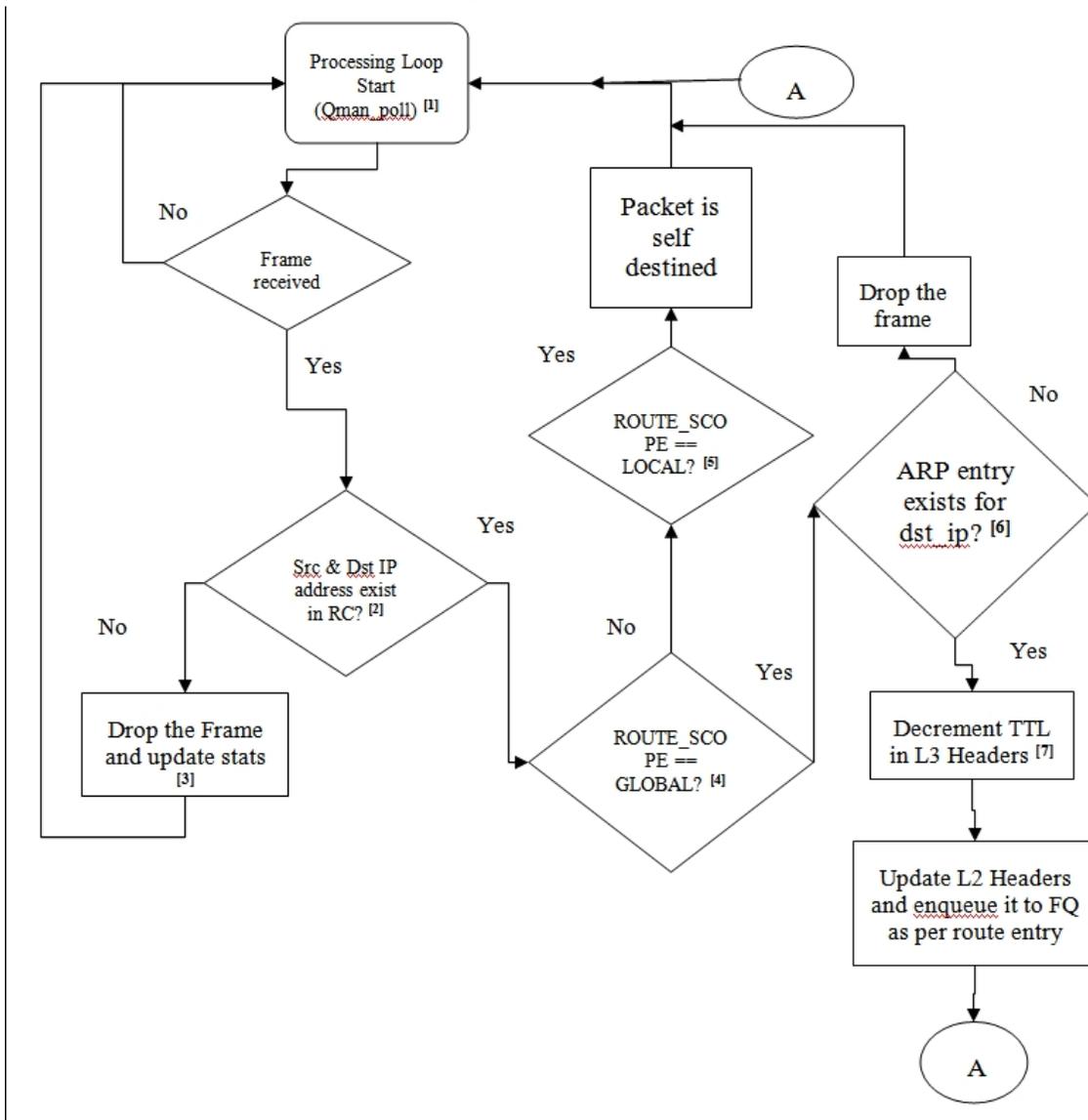
```
ipfwd_config -P 2536 -F -a 192.168.24.1 -i 5
ipfwd_config -P 2536 -F -a 192.168.28.1 -i 9
ipfwd_config -P 2536 -F -a 192.168.27.1 -i 8
ipfwd_config -P 2536 -F -a 192.168.29.1 -i 11

# Now enter routes and MAC addresses. Format of a MAC address is
# aa:bb:cc:dd:ee:ff where the letters are hexadecimal digits.

ipfwd_config -P 2536 -B -s 192.168.27.2 -d 192.168.28.2 -g 192.168.28.2
ipfwd_config -P 2536 -G -s 192.168.28.2 -m COMPUTER_Y_MAC_ADDRESS -r true
ipfwd_config -P 2536 -B -s 192.168.28.2 -d 192.168.27.2 -g 192.168.27.2
ipfwd_config -P 2536 -G -s 192.168.27.2 -m COMPUTER_X_MAC_ADDRESS -r true
# Computer X and Computer Y need to be told to route via the P4080.
# On Computer X (assuming it runs Linux), enter this command as root:
route add -net 192.168.28.0 netmask 255.255.255.0 gw 192.168.27.1
# On Computer Y (assuming it runs Linux), enter this command as root:
route add -net 192.168.27.0 netmask 255.255.255.0 gw 192.168.28.1
# Now, traffic can pass between Computer X and Computer Y. For example, on Computer X
# enter:
ping 192.168.28.2
```

7.4.6.10 Flowchart for packet processing

This topic provides a flowchart for packet processing.



7.4.6.10.1 Description of Flow chart

1. All of the threads enter the processing loop where they call Qman_poll till a frame is received.
2. If a frame is received, the application checks whether the source and destination IP address exist in the Route cache. For this it calls rc_entry_fast_lookup(), which does the fast route look up by using the hash value provided by the FMan. The hash value is indexed into the route table. The IPFWD application does not change its route cache in response to seeing the first packet in a flow. Instead the route cache is set only by commands, as mentioned in section [IPv4 forward application Configuration command](#) on page 936.
3. If the route entry for this frame is not present in the Route cache, the frame is dropped and statistics are updated. It then continues with Qman_poll.
4. If the route entry for this frame exists in the Route cache, it checks for the scope of the frame. If the scope is GLOBAL, the frame is sent for forwarding.
5. If the scope is not GLOBAL but LOCAL, the frame is self-destined. It then continues with Qman_poll.

6. If the frame is to be forwarded, check if ARP entry exists in ARP table for destination IP address. IPFWD application will not dynamically resolve the ARP. So if sending packets to IPFWD using a regular computer, the user will have to create static ARP entries.
7. If ARP entry exists, TTL is decremented in L3 header.
8. Finally, the L2 header is updated, which includes changing the dst MAC address in L2 header. The frame is then enqueued to the TX FQ.

7.4.6.10.2 Running IPv4 forwarding on P4080DS board

The instructions below describe how to run IPFWD. Traffic should only be directed to the P4080DS once the application is running and configuration via the ipfwd_config application is completed.

- On linux prompt, assign IP address to fm1-gb1

```
$ ifconfig fm1-gb1 <IPADD> up
```

- Configure FMan PCD using fmc with the XML files in /usr/etc:

```
$ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
$ fmc -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_32_fq.xml -a
```

To setup the FMan to distribute traffic to 1024 ingress frame queues per port:

```
$ fmc -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_1024_fq.xml -a
```

- Run IPFWD application

*The main IPFwd application binary is called **ipfwd_app**. The application can run on multiple cores as specified by the first parameter to the application. To run it to handle traffic distributed over 32 ingress frame queues per port:*

```
$ ipfwd_app <m..n>
```

By default ipfwd_app uses us_config_serdes_0xe.xml and us_policy_hash_ipv4_src_dst_32_fq.xml files.

To run for scenario 1024 ingress frame queues per port

```
$ ipfwd_app <m..n> -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_1024_fq.xml
```

IPFWD application command syntax:

```
[root@p4080 etc]# ipfwd_app --usage
```

```
Usage: ipfwd_app [-n?V] [-c FILE] [-d SIZE] [-i FILE] [-p FILE] [--fm-config=FILE]
```

```
 [--non-interactive] [--fm-pcd=FILE] [--cpu-range] [--help]
```

```
 [--usage] [--version] [cpu-range]
```

IPFWD application run command:

```
[root@p4080 root]# cd /usr/etc
```

```
<_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_32_fq.xml -a
```

If the user wants to use all interfaces :

```
[root@p4080 etc]# ipfwd_app 1..7
```

If the user wants to use ONLY selective interfaces :

```
[root@p4080 etc]# ipfwd_app 1..7 -i fm1-10g,fm2-gb2
```

```
[1] 5363
```

Linux User Space
USDPAA Applications

```
[root@p4080 etc]# Found /fsl,dpaa/ethernet@9
Found /fsl,dpaa/ethernet@8
Found /fsl,dpaa/ethernet@7
Found /fsl,dpaa/ethernet@4
Qman: FQID allocator includes range 512:128
Bman: BPID allocator includes range 56:8
Configuring for 4 network interfaces and 4 pool channels
FSL dma_mem device mapped (phys=0xf8000000,virt=0x70000000,sz=0x4000000)
Thread uid:0 alive (on cpu 1)
Release 16384 bufs to BPID 7
Release 4096 bufs to BPID 8
Release 4096 bufs to BPID 9

ipfwd_app starting
Message queue to send: /mq_snd_2536
Message queue to receive: /mq_rcv_2536
Thread uid:1 alive (on cpu 2)
Thread uid:2 alive (on cpu 3)
Thread uid:3 alive (on cpu 4)
Thread uid:4 alive (on cpu 5)
Thread uid:5 alive (on cpu 6)
Thread uid:6 alive (on cpu 7)
```

If, in the run application command, cpu-range is given i.e. "ipfwd_app <m..n>" IPFWD application starts threads on cpu-range m..n. The main thread (by default on CPU 1) then does global initialization needed by the application, including starting other application threads.

If, on the other hand, run application command is given without any cpu-range i.e. "ipfwd_app" IPFWD application starts up with a single thread running on CPU 1 by default, which does global initialization needed by the application and enables all the network interfaces.

The CLI (Command-Line Interface) allows you to add and remove additional threads to enable the use of multiple CPUs, with the only restriction being that the primary thread on CPU 1 cannot be removed (except by shutting down the application).

To add a thread on a single CPU (e.g. CPU 2):

```
> add 2
```

To add threads on a range of CPUs:

```
> add 3..6
```

To list the threads (this also queries each thread, verifying that they aren't blocked):

```
> list
```

```
Thread alive on cpu 1
Thread alive on cpu 2
Thread alive on cpu 3
Thread alive on cpu 4
```

Thread alive on cpu 5

Thread alive on cpu 6

To enable all interfaces

> macs on

To disable all interfaces

> macs off

To perform a controlled shutdown of ipfwd (this includes disabling the network ports):

> quit

- Once the application starts, it can receive the configuration commands. Run application configuration script.

For creating route entries, the binary *ipfwd_config* is run. This binary processes the configuration request from the user (using the command line) and populates the configuration via Linux standard posix IPC messages to the IPFwd application.

The shell script mentioned below contains sample commands to add route entries. Detailed description of all *ipfwd_config* commands is provided in section [IPv4 forward application Configuration command](#) on page 936.

SSH to p4080 linux on another terminal:

```
$ ssh root@<IPADD> (give the IP address as assigned to fm1-gb1 in the beginning)
```

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

```
ipfwd_20G.sh "pid"
```

```
$ ipfwd_20G.sh 2536
```

There are two example shell scripts available and those are "ipfwd_20G.sh" and "ipfwd_22G.sh". ipfwd_20G.sh creates routes for only the 2 x 10G interfaces and ipfwd_22G.sh creates routes for 2 x 10G and 2 x 1G interfaces. They can assign IP addresses to the interfaces, add an ARP entry and assumes the netmask to be 255.255.255.0. The following table summarizes the settings done by these scripts. Check section [Possible configuration scenario for IPFWD](#) on page 922 for more details.

For the IPFwd application to forward traffic successfully, traffic destined for the P4080DS ports must have the appropriate source and destination addresses.

Console messages are printed for each entry added to the routing table. Once all configuration is completed, the application moves to the packet processing phase - the message "Application Started successfully" is printed on the console. At this point, traffic can be sent to the IPFWD application, which would do its processing on the cpu-range specified by the user on the application command-line.

7.4.6.10.3 Running IPv4 forwarding on P3041/P5020 board

The instructions below describe how to run IPFWD on P3041/P5020. Traffic should only be directed to P3041/P5020 once the application is running and configuration via the *ipfwd_config* application is completed. · On linux prompt, assign IP address to eth0

```
$ ifconfig eth0 <IPADD> up
```

Running IPFWD

- Configure FMan PCD using fmc with the XML files in /usr/etc:

```
$ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
$ fmc -c usdpaa_config_p3_p5_serdes_0x36.xml -p usdpaa_policy_hash_ipv4.xml -a
```

- Run IPFWD application

If the user wants to use all interfaces :

```
$ ipfwd_app <m..n> -c usdpaa_config_p3_p5_serdes_0x36.xml -p usdpaa_policy_hash_ipv4.xml
```

If the user wants to use ONLY selective interfaces :

```
$ ipfwd_app m..n -c usdpaa_config_p3_p5_serdes_0x36.xml -p usdpaa_policy_hash_ipv4.xml -i fm1-10g,fm1-gb4
```

For P3041, m..n can be 0..3. For P5020, m..n can be 0..1.

SSH to p4080 linux on another terminal:

```
$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)
```

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

```
ipfwd_15G.sh "pid"
```

```
$ ipfwd_15G.sh 2536
```

There is an example shell script available named as ipfwd_15G.sh creates routes for only the 5 x 1G and 1x10G interfaces. It can assign ip addresses to the interfaces, add an ARP entry and assumes the netmask to be 255.255.255.0. Check section [Possible configuration scenario for IPFWD](#) on page 922 for more details. Now traffic can be run as per the routes created.

7.4.6.10.4 Running IPv4 forwarding on T4240 board

The instructions below describe how to run IPFWD on T4240. Traffic should only be directed to T4240 once the application is running and configuration via the ipfwd_config application is completed. · On linux prompt, assign IP address to eth0

```
$ ifconfig eth0 <IPADD> up
```

Running IPFWD

- Configure FMan PCD using fmc with the XML files in /usr/etc:

```
$ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
$ fmc -c usdpaa_config_t4_serdes_1_1_6_6.xml -p usdpaa_policy_hash_ipv4.xml -a
```

- Run IPFWD application

```
$ ipfwd_app m..n -c usdpaa_config_t4_serdes_1_1_6_6.xml -p usdpaa_policy_hash_ipv4.xml -d 0x10000000
```

For T4240, m..n can be 0..23.

SSH to t4240 linux on another terminal:

```
$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)
```

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

```
ipfwd_20G.sh "pid"
```

```
$ ipfwd_20G.sh 2536
```

There is an example shell script available named as ipfwd_20G.sh creates routes for only the 2x10G interfaces. It can assign ip addresses to the interfaces, add an ARP entry and assumes the netmask to be 255.255.255.0. Now traffic can be run as per the routes created.

7.4.6.10.5 Running IPv4 forwarding on B4860 board

The instructions below describe how to run IPFWD on B4860. Traffic should only be directed to B4860 once the application is running and configuration via the ipfwd_config application is completed. On linux prompt, assign IP address to eth0

```
$ ifconfig eth0 <IPADD> up
```

Running IPFWD

- Configure FMan PCD using fmc with the XML files in /usr/etc:

```
$ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
$ fmc -c usdpaa_config_b4_serdes_0x2a_0x98.xml -p usdpaa_policy_hash_ipv4.xml -a
```

- Run IPFWD application

```
$ ipfwd_app m..n -c usdpaa_config_b4_serdes_0x2a_0x98.xml -p usdpaa_policy_hash_ipv4.xml
```

For B4860, m..n can be 0..7.

SSH to b4860 linux on another terminal:

```
$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)
```

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

```
ipfwd_20G.sh "pid"
```

```
$ ipfwd_20G.sh 2536
```

There is an example shell script available named as ipfwd_20G.sh creates routes for only the 2x10G interfaces. It can assign ip addresses to the interfaces, add an ARP entry and assumes the netmask to be 255.255.255.0. Now traffic can be run as per the routes created.

7.4.6.10.6 USDPAA IP Fwd performance gap between 6 core and 8 core

USDPAA IPfwd performance for 8 core is less than 6 core on e6500 series. USDPAA IP Fwd application need to run using "-s" option to bridge the gap between two configuration.

7.4.6.10.7 PPAC (and IPFwd) CLI commands

The following commands are illustrated in the context of IPFwd, but the commands are common to all PPAC-based applications.

To add a thread on a single CPU (e.g. CPU 2):

```
> add 2
```

To add threads on a range of CPUs:

```
> add 3..6
```

To list the threads (this also queries each thread, verifying that they aren't blocked):

```
> list
```

```
Thread uid:0 alive (on cpu 1)
```

```
Thread uid:1 alive (on cpu 2)
```

```
Thread uid:2 alive (on cpu 3)
```

```
Thread uid:3 alive (on cpu 4)
```

Linux User Space
USDPAAs Applications

Thread uid:4 alive (on cpu 5)

Thread uid:5 alive (on cpu 6)

Thread uid:6 alive (on cpu 7)

To remove a thread by its UID:

```
> rm uid:2
```

Thread uid:2 killed (cpu 3)

To remove a thread running on a given CPU:

```
> rm 5 Thread uid:4 killed (cpu 5)
```

To enable all interfaces:

```
> macs on
```

To disable all interfaces:

```
> macs off
```

To perform a controlled shutdown of ipfwd (this includes disabling the network ports):

```
> quit
```

To query cgr

```
> cgr
```

Rx CGR ID: 10, selected fields;

cscn_en: 0

cscn_targ: 0x00800000

cstd_en: 1

cs: 0

cs_thresh: 0x00_0000_1000

mode: 1

i_bcnc: 0x00_0000_0e1e

a_bcnc: 0x00_0000_0e1e

Tx CGR ID: 11, selected fields;

cscn_en: 0

cscn_targ: 0x00800000

cstd_en: 1

cs: 0

cs_thresh: 0x00_0000_0200

mode: 1

i_bcnc: 0x00_0000_0002

a_bcnc: 0x00_0000_0004

7.4.6.11 IPv4 forward application Configuration command

7.4.6.11.1 Syntax

The syntax is as follows:

```
$ [root@p4080 bin]# ipfwd_config --help
Usage: ipfwd_config [OPTION...]

-B, --routeadd=TYPE adding a route
-C, --routedel=TYPE deleting a route
-E, --showintf=TYPE show interfaces
-F, --intfconf=TYPE change intf config
-G, --arpadd=TYPE adding a arp entry
-H, --arpdel=TYPE deleting a arp entry
-O, --Start/ Go=TYPE Start the processing of packets
-?, --help Give this help list
--usage Give a short usage message
-V, --version Print program version
```

7.4.6.11.1 Command to show all enabled interfaces and their interface numbers

The command to show all the enabled interfaces while running IPv4 forward is as follows:

```
Ipfwd_config -P pid -E -a true
```

Table 143. Field Description (show all enabled interfaces)

Parameter	Description	Mandatory	Format/ Value
-a	Show all the interfaces	Yes	true

Command to show all enabled interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipfwd_config -P 2536 -E -a true
```

Interface number: 11

PortID=1:5 is FMan interface node

with MAC Address

02:00:c0:a8:65:fe

Interface number: 9

PortID=1:3 is FMan interface node

with MAC Address

02:00:c0:a8:5b:fe

Interface number: 8

PortID=1:2 is FMan interface node

with MAC Address

02:00:c0:a8:51:fe

Interface number: 5

PortID=0:5 is FMan interface node

with MAC Address

02:00:c0:a8:33:fe

Are all the Enabled Interfaces [root@p4080 bin]#

74.6.11.1.2 Help for show all enabled interfaces command

The command to obtain help for show all enabled interfaces command is as follows:

```
ipfwd_config -E -help
```

Help for show all enabled interfaces

```
[root@p4080 etc]# ipfwd_config -E --help
```

Usage: -E [OPTION...]

-a, --a=ALL All interfaces

-, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

74.6.11.1.3 Assign IP address to interfaces

The command to assign IP address while running IPv4 forward is as follows:

```
ipfwd_config -P pid -F -a 192.168.60.1 -i <Interface number>
```

NOTE

Note: The interface number to be used here must be one of the numbers that got displayed as the output of "show all enabled interfaces command" in section [Command to show all enabled interfaces and their interface numbers](#) on page 937.

Table 144. Field description (assign IP address to interfaces)

Parameter	Description	Mandatory	Format/ Value
-a	IP Address	Yes	a.b.c.d
-i	Interface number	Yes	0-11 (Choose this number from "show all enabled interfaces" command output)

Assign IP address to interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
ipfwd_config -P 2536 -F -a 192.168.60.1 -i 5
```

IPADDR assigned = 0xc0a83c01 to interface num 5

Intf Configuration Changed successfully

7.4.6.11.1.4 Help for assign IP address to interfaces

The command to obtain help for assign IP address to interfaces command is as follows:

```
ipfwd_config -F --help
```

Help for assign IP address to interfaces

```
[root@p4080 etc]# ipfwd_config -F --help
```

Usage: -F [OPTION...]

-a, --a=IPADDR IP Address

-i, --i=IFNAME If Name

-, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

7.4.6.11.1.5 Adding a Route Entry

The command to add a route while running IPv4 forward is as follows:

```
ipfwd_config -P pid -B -s a.b.c.d -d b.c.d.e -g c.d.e.f
```

Table 145. Field Description (Adding a Route Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Source IP Address	Yes	a.b.c.d
-d	Destination IP Address	Yes	a.b.c.d
-g	Gateway IP Address	Yes	a.b.c.d

Adding a Route Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipfwd_config -P 2536 -B -s 192.168.29.2 -d 192.168.24.2 -g 192.168.24.2
```

Route Entry Added successfully

```
[root@p4080 bin]#
```

7.4.6.11.1.6 Help for Route Entry Addition

The command to obtain help for route entry addition is as follows:

```
ipfwd_config -B --help
```

Help for Adding a Route Entry

```
[root@p4080 bin]# ipfwd_config -B --help
```

Usage: -B [OPTION]

- d, --d=DESTIP Destination IP
- f, --f=FLOWID Flow ID - (0 - 1024) {Default: 0}
- g, --g=GWIP Gateway IP
- s, --s=SRCIP Source IP
- t, --t=TOS Type of Service - (0 - 256) {Default: 0}
- , --help Give this help list
- usage Give a short usage message
- V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

7.4.6.11.17 Deleting a Route Entry

The command to delete a route while running IPv4 forward is as follows:

```
ipfwd_config -P pid -C -s a.b.c.d -d b.c.d.e
```

Table 146. Field Description (Deleting a Route Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Source IP Address	Yes	a.b.c.d
-d	Destination IP Address	Yes	a.b.c.d

Deleting a Route Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipfwd_config -p 2536 -C -s 192.168.29.2 -d 192.168.24.2 Route Entry Deleted successfully
```

7.4.6.11.18 Help for Deleting a Route Entry

The command to obtain help for route entry deletion is as follows:

```
ipfwd_config -C --help
```

Help for Deleting a Route Entry

```
[root@p4080 bin]# ipfwd_config -C --help
```

Usage: -C [OPTION...]

- d, --d=DESTIP Destination IP
- s, --s=SRCIP Source IP
- t, --t=TOS Type of Service - (0 - 256) {Default: 0}
- , --help Give this help list
- usage Give a short usage message
- V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

7.4.6.11.1.9 Starting the Application Processing

The command to start the application processing phase is as follows:

```
ipfwd_config -O
Starting the Application Processing
[root@p4080 bin]# ipfwd_config -O
Application Started successfully
```

7.4.6.11.1.10 Adding an ARP Entry

The command to add an ARP entry while running IPv4 forward is as follows:

```
ipfwd_config -P pid -G -s a.b.c.d -m aa:bb:cc:dd:ee [-r true]
```

Table 147. Field Description (Adding an ARP Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Gateway IP Address	Yes	a.b.c.d
-m	Mac Address	Yes	aa:bb:cc:dd:ee
-r	Replace existing entry	No	true/ false {Default: false}

Adding an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipfwd_config -P 2536 -G -s 192.168.24.2 -m 02:00:c0:a8:33:fd -r true
ARP Entry Added successfully
```

7.4.6.11.1.11 Help for ARP Entry Addition

The command to obtain help for ARP entry addition is as follows:

```
ipfwd_config -G --help
Help for Adding an ARP Entry
[root@p4080 etc]# ipfwd_config -G --help
Usage: -G [OPTION...]
-m, --m=MACADDR MAC Address
-r, --r=Replace Replace Exiting Entry - true/ false {Default:
false}
-s, --s=IPADDR IP Address
-?, --help Give this help list
--usage Give a short usage message
-V, --version Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

7.4.6.11.1.12 Deleting an ARP Entry

The command to delete an ARP while running the IPv4 forward is as follows:

```
ipfwd_config -P pid -H -s a.b.c.d
```

Table 148. Field Description (Deleting an ARP Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Gateway IP Address	Yes	a.b.c.d

Deleting an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipfwd_config -P 2536 -H -s 192.168.24.2
```

```
Arp Entry Deleted successfully
```

```
[root@p4080 bin]#
```

7.4.6.11.1.13 Help for Deleting an ARP Entry

The command to obtain help for ARP entry deletion is as follows:

```
ipfwd_config -H --help
```

Help for Deleting an ARP Entry

```
[root@p4080 etc]# ipfwd_config -H --help
```

```
Usage: -H [OPTION...]
```

```
-s, --s=IPADDR IP Address
```

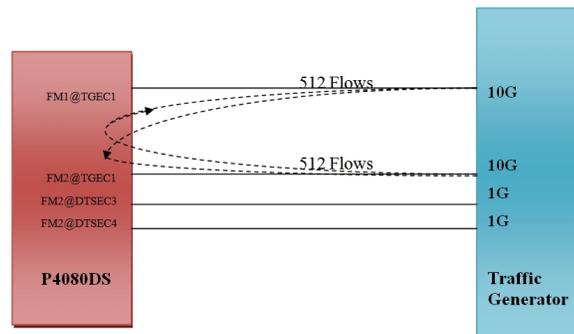
```
-, --help Give this help list
```

```
--usage Give a short usage message
```

```
-V, --version Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

7.4.6.12 Traffic Generation



7.4.6.13 References

1. USDPAAs PPAC User Guide
2. QMan/BMan API Guide

7.4.7 NXP USDPAAs IPSecfwd User Manual

7.4.7.1 Introduction

The User Space Data Path Acceleration Architecture (USDPAAs) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document provides the following:

- A summary of the USDPAAs IPSecfwd application.
- Execution steps for the IPSecfwd application.

7.4.7.1.1 Purpose

This document describes the USDPAAs IPsec forwarding application. This application documents the USDPAAs IPSecfwd demonstration applications forwarding flow.

7.4.7.1.2 Change History

Table 149. Change History

Version	Updates
1.1	Creation of IPsecfwd UG for phase v1.0 release. - PPAC/PPAM overview - Basic application flow - Commands to use - Testing IPsecfwd application
1.2	Addition of t4/b4 sections

7.4.7.2 USDPAA IPsecfw application

The USDPAA IPsec forwarding (IPsecfw) application is a multi-threaded application that routes IPv4 packets from one Ethernet interface to another after performing encryption/decryption if required on P4080/P3041/P5020/T4240/B4860 systems. The configuration done for the system and the type of the packet is used to determine the type of operation to be performed on the packet. The routing is done based on the source IP address and destination IP address in the frame.

Any combination of the 8 cores on the P4080 can run a USDPAA IPsec Forwarding application thread in USDPAA SDK v1.0.

7.4.7.2.1 Application Overview

The IPsecfw application can route IPv4 traffic directly over interfaces as well as over the IPsec tunnel between two network nodes connected over different subnets with the assistance of the IPsec protocol security provided by SEC4.0 block. The application works as an extension of the IPv4 forwarding application with encryption/decryption in addition to route lookup and packet forwarding. The IPsec processing tunnel table contains information on frame queues toward SEC4.0 for the IPsec protocol processing of the packet before it is sent out on the egress interface. IPsec can use an extended sequence number (ESN) optionally that is authenticated but not transmitted. If an ESN is found in the PDB, it is given to the authentication CHA in the last. The ESN is incremented whenever the Seq Num rolls over.

The IPsecfw application has two main phases. There is an initial configuration phase and a subsequent packet processing phase.

The configuration phase executes when the application starts. Application threads are created and global initialization of resources is done which is the part of PPAC (see USDPAA PPAC User Guide for more details). The configuration phase also includes PPAM (i.e. IPsecfw) related initialization. Once the configuration phase is completed the IPsecfw application moves to the packet processing phase.

The application provides a command-line interface to enable users to add and remove routing table, ARP cache entries and SA entries at any given time. For each user input, the appropriate information is communicated to the IPsecfw application via standard Posix IPC. Messages are placed onto the message queue till they are received by the IPsecfw application. Note that the IPsecfw application does not dynamically resolve ARP - missing ARP entries will result in the application dropping the packet.

In the packet processing phase, the loop migrates from polling mode to a blocking iIRQ mode whenever IPsecfw has looped a certain number of times without any forward progress. For more information on IRQ mode please refer to iUSDPAA PPAC User Guidei. In polling mode - the application constantly looks for data to process on its dedicated QMan portal. Network traffic is classified and distributed by the FMan to frame queues based on source and destination IP address in the packet. There is an associated handler that processes the packets arriving on each frame queue.

7.4.7.2.2 Packet Flow

The IPsecfw application is capable of routing packets with the security processing support from the P4080's SEC4.0 engine. After all the initialization and configurations, each core enters the polling loop and polls for packets from FMan(PPAC), or SEC4.0 (PPAM). The loop dequeues packets from the frame queue associated to pool channels shared by the application cores.

Once a frame is dequeued, it is sent for processing based on callback handler in contextB of the frame queue response ring entry. When the frame is ingressed from FMAN, then the callback handler is called which performs the route lookup and tunnel lookup; and then sends the packet to SEC4.0 block/FMan. For the frame returning from SEC4.0, the encap callback handler is called for a frame coming after encryption or decap callback handler is called for a frame coming after decryption.

The steps for the packet flow for ingress from FMan are as follows:

1. Once the packet is dequeued from FMan, DQRR callback handler associated with Fman is called.
2. After a basic sanity check of the packet (packet header is correct, packet checksum is correct etc.), the packet is checked for the ESP protocol.
 - a. If the packet is ESP, a direct tunnel lookup happens based on FMan's PCD hash value and the frame is sent to SEC4.0 engine

- b. If the packet is non-ESP, a route lookup based on FMan's PCD happens. If there is an SA entry, it is sent to SEC4.0 block after updating annotations in frame with next level route destination.
 - c. If the packet is non-ESP and the route is normal, the packet is sent to FMan interface.
 - d. If no route is found, the packet is discarded.
3. When the packet is dequeued from SEC4.0 Engine, the DQRR callback handler associated with SEC4.0 frame queues is called. Encap callback handler or Decap callback handler will be called from based on the callback handler in contextB of DQRR ring entry.
 4. The packet is now taken directly to the FMan interface as the destination field was already filled while sending the packet to the SEC4.0 Engine.

A typical packet flow in a simulated P4080 environment is shown below.

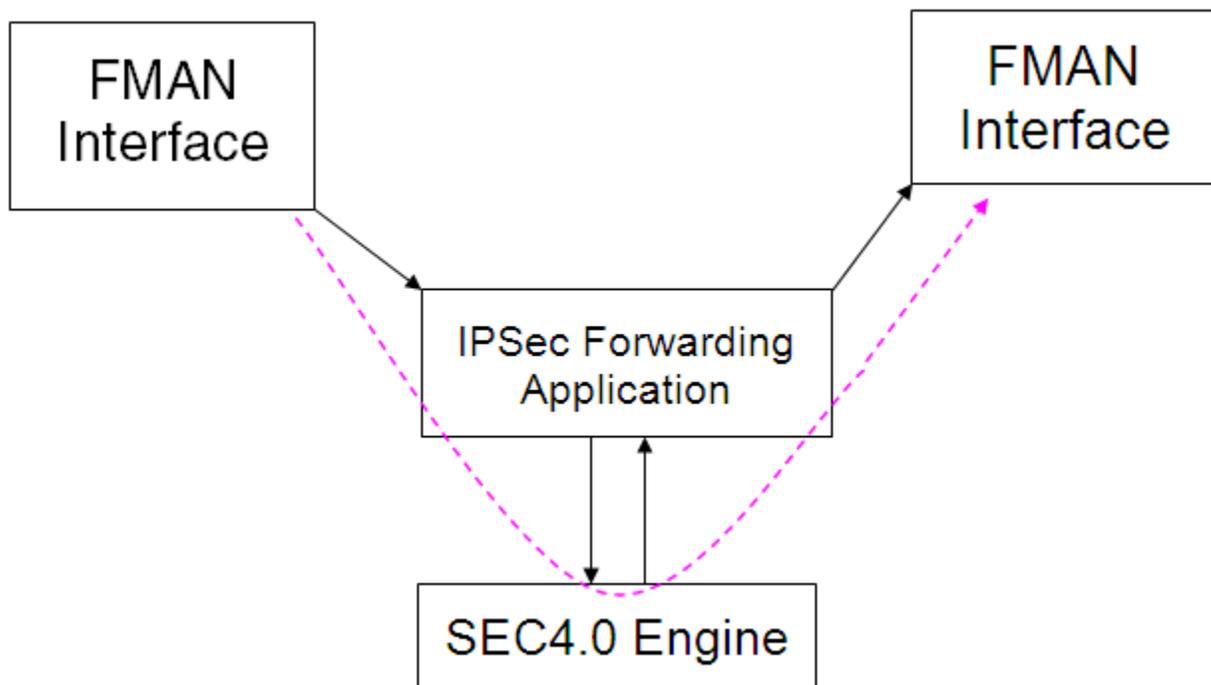


Figure 143. IPsecfw Application basic packet flow

7.4.7.2.3 Overview of IPsecfw packet processing

IPsecfw Steady state processing or Forwarding consists of two parts

- Outbound processing: when un-encrypted packets are received by the system and tunneled using IPsecfw application.
- Inbound processing: when encrypted packets are received by the systems and are decrypted by the system.

7.4.7.2.3.1 Outbound processing:

1. *Pre-processing:*
 - a. Packet is received by the FM which uses 2-tuple (Src IP, Dest IP) to hash the packets to different Core Rx Queues.
 - b. This packet is picked up by the core and first subjected to IPv4 Forwarding lookup.
 - c. If the action is to do IPsec processing, then it looks up for a corresponding entry in the SADB, which has the SA information and Queue-ids for the SA used by SEC4.0.
2. *Crypto-processing:*
 - a. SEC4.0 dequeues the job from egress Queue toward SEC4.0.

- b. The SEC4.0 encrypts, authenticates and adds the ESP header and outer IP header to the packet. It then enqueues the processed packet to the Ingress Queue from SEC4.0.

3. Post-processing:

- a. The core now dequeues the processed packet from the SEC4.0 on Out FQ and sends it to the IPv4 Forwarding module (since there is a new Outer-IP header).
- b. IPv4 Forwarding module does a lookup and transmits the packet from the egress interface.
- c. If the SEC4.0 result indicates an error in the Crypto processing of the packet, it is discarded and statistics (packet, octets, and errors - per SA / Interface) are updated.

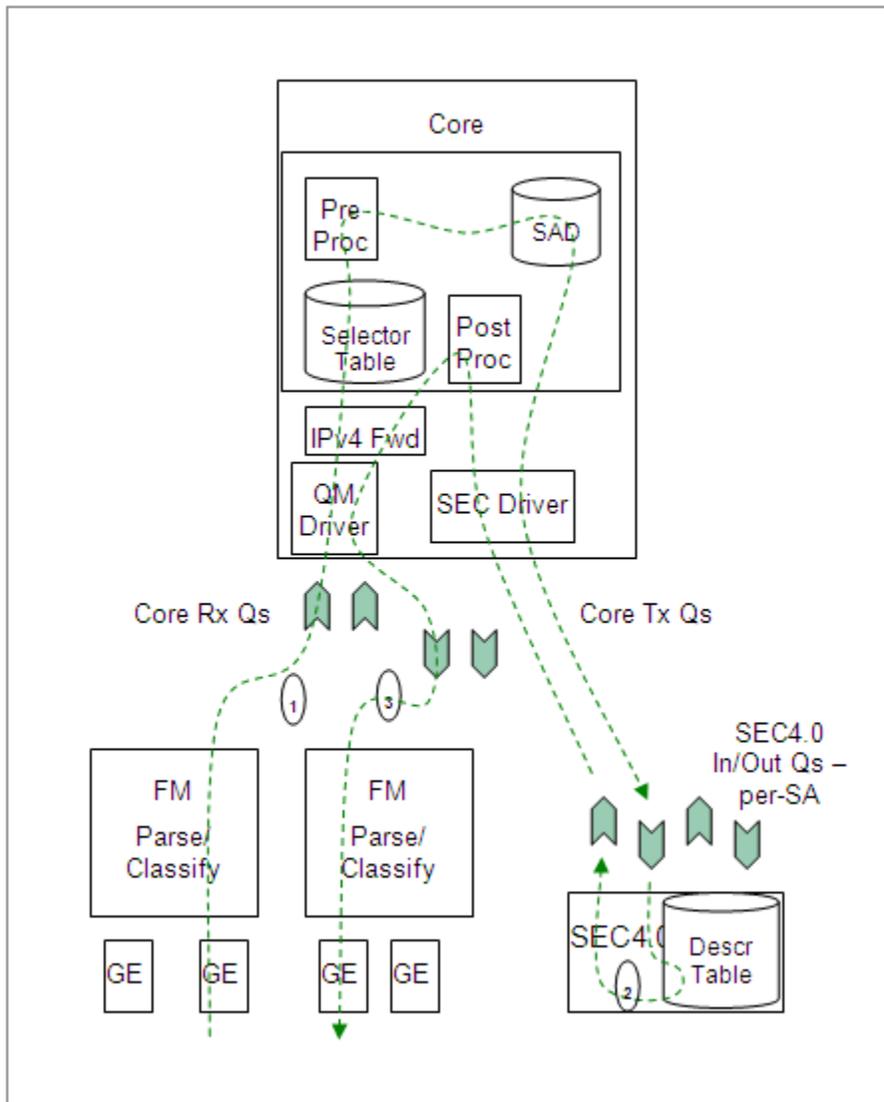


Figure 144. IPsecfw Outbound processing

74.72.3.2 Inbound Processing:

1. Pre-processing:

- a. The IPsec tunneled packet is received by the FM, which uses a 3-tuple (Dest-IP, Src-IP, SPI) to hash the packet to different Core Rx Queues.

- b. The packet is picked up and undergoes IPv4 Forwarding lookup. If packet is self-destined, and protocol is ESP, it is sent to IPsec Forwarding module for processing.
 - c. The SA info in SADB yields a SEC4.0-In Queue for the SA. The packet is sent to that SEC4.0 on its ingress FQ.
2. Crypto-processing:
- a. The SEC4.0 decrypts, authenticates and enqueues the processed packet to the SEC4.0-Out Queue found in its input FQs contextB.
3. Post-processing:
- a. The core de-queues the processed packet from the SEC4.0-Out Queue, and subjects the packet to a Selector-table lookup.
 - b. The packet is handed over to IPv4 Forwarding module, which does a route cache lookup and transmits the packet from the egress interface.
 - c. If the SEC4.0 result indicates an error in the Crypto processing of the packet, it is discarded and statistics are updated.

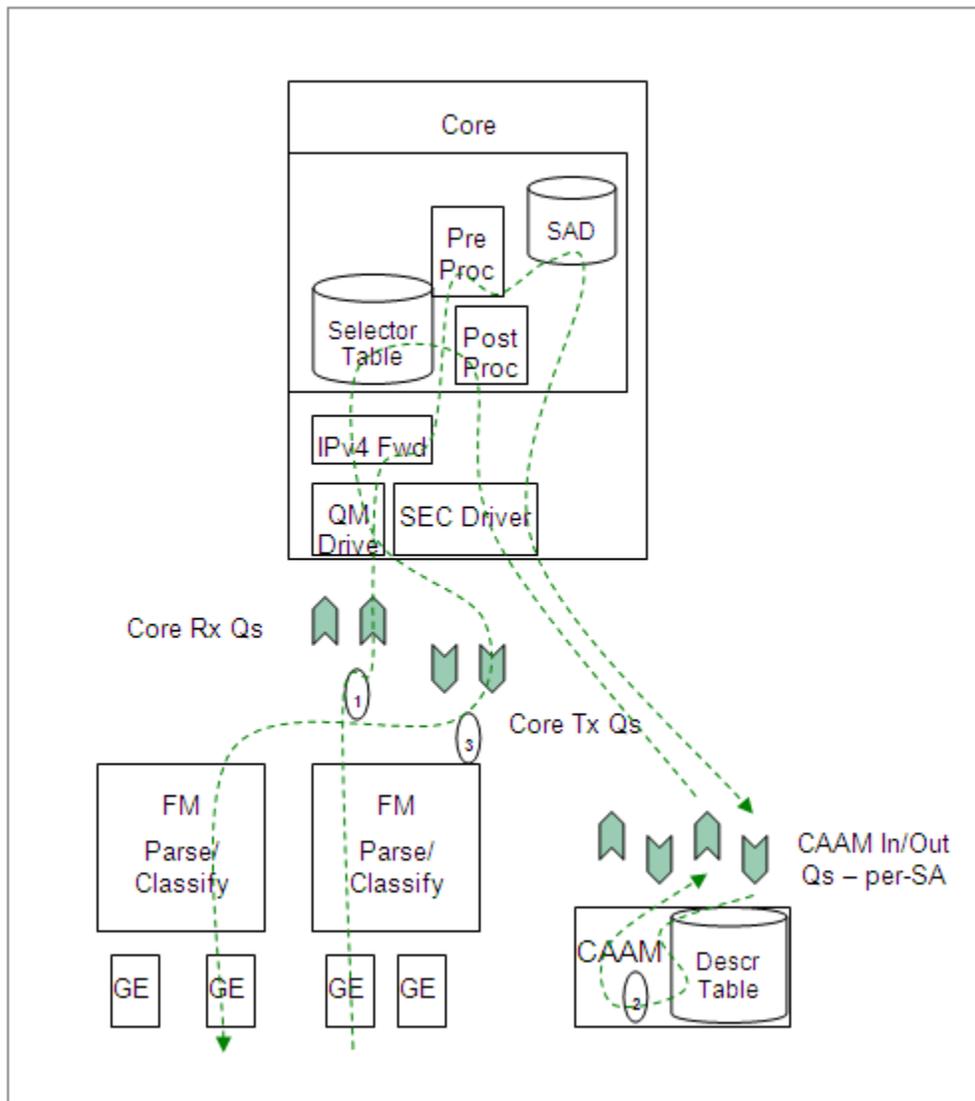


Figure 145. IPsecfw Inbound processing

7.4.7.2.4 Flow chart for IpSecfwd packet processing

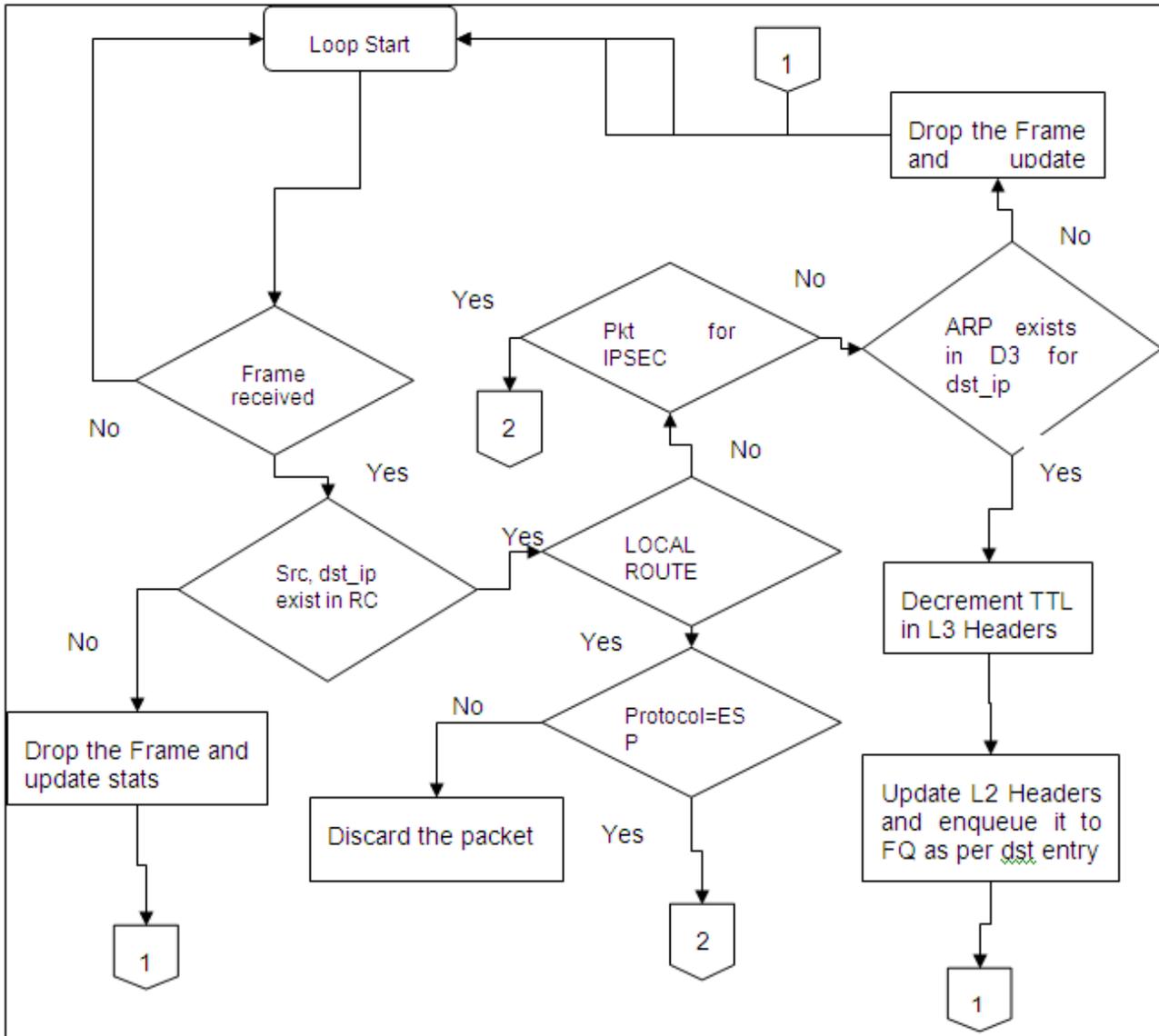


Figure 146. Flow chart for IpSecfwd packet processing

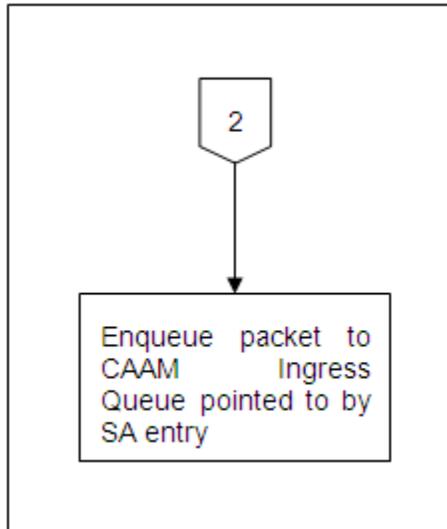


Figure 147. IPsecfw packet processing

7.4.7.3 Overview of PPAC

The source code to IPsecfw has been reorganized into two parts; the "PPAC" (Packet-Processing Application Core) and a "PPAM" (Packet-Processing Application Module). The PPAM portion implements the IPsecfw application specific logic of processing the packet and forwarding it. On the other hand, the PPAC component represents the common infrastructure to support PPAM; initializing devices, handling flow-control, implementing a CLI (Command-Line Interface), managing threads and buffers. PPAC details can be found in the document "USDPAAs PPAC User Guide".

7.4.7.4 IPsecfw related PPAM Details

7.4.7.4.1 In-Place Encryption/Decryption

IPsecfw application uses in-place Encryption/Decryption when packet is sent to SEC4.0 block. In-place Encryption/Decryption is supported in IPsecfw application by using the same output buffer as the input buffer.

In case of Encryption the size of the output packet is more than the size of input packet due to addition of tunnel header, padding of extra bytes etc. The FMam is configured to acquire the buffer from BMan which is large enough to accommodate the output packet after encryption.

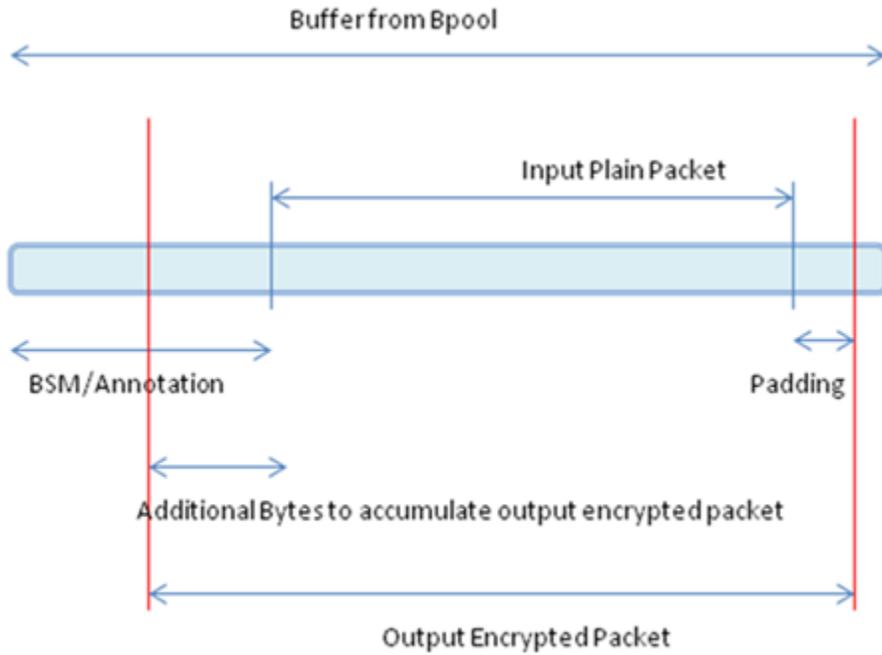


Figure 148. IPsec In-place Encryption

In case of Decryption the output packet size is smaller than the Input packet size. So the output plain packet in case of Decryption can easily be accommodated in the buffer acquired by FMan for storing the Input packet.

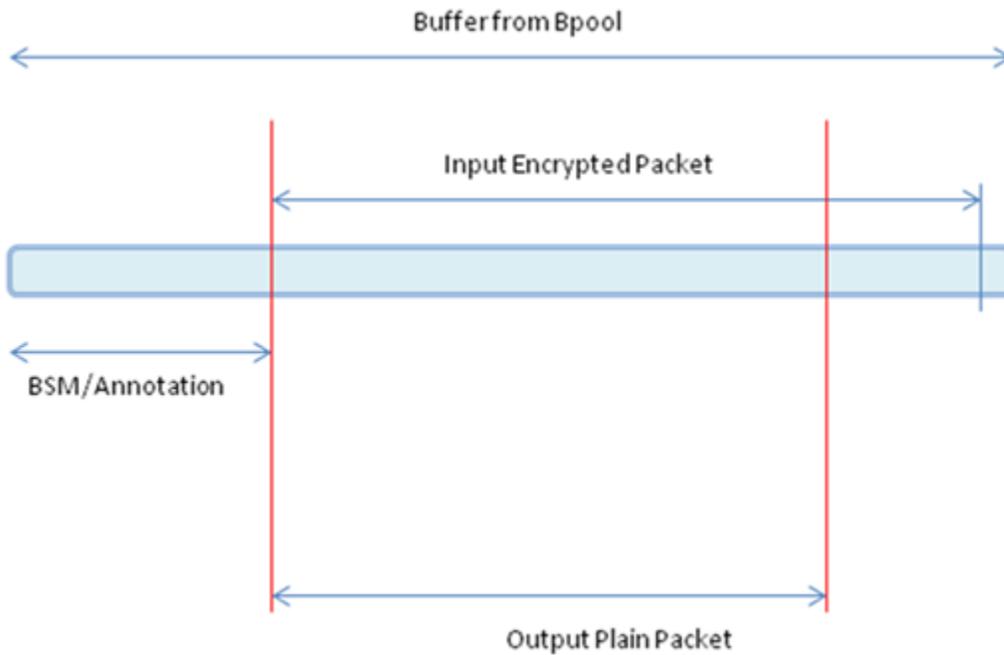


Figure 6: IPsec In-place Decryption

Figure 149. IPsec In-place Decryption

7.4.7.5 Secfwd application suite

The figure below shows the structure of the IPsecfwd USDPAA application suite. Its purpose is to encrypt/decrypt and forward IPv4 packets between the interfaces shown. No IP address is assigned to any of the interfaces by default. Using ipsecfwd_config command as mentioned in section 5.3.1.3 IP address can be assigned to all these interfaces. Each interface has a fixed netmask shown in the figure. The notation "/24" refers to a netmask of 255.255.255.0. The MAC addresses of these interfaces are determined by u-boot environment variables ethaddr, eth1addr, eth2addr, etc.

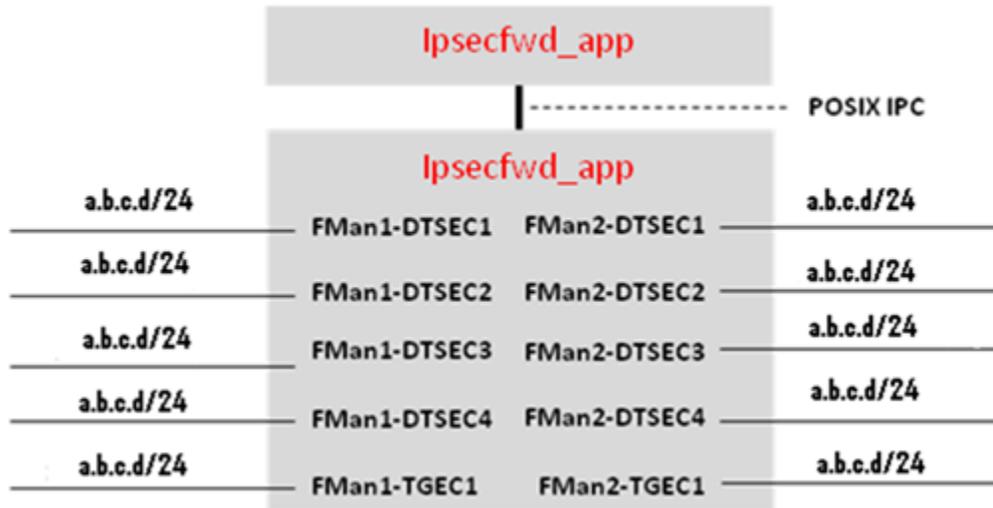


Figure 150. IPsecfwd application suite

The ipsecfwd application will respond to ARP requests on these interfaces. However, the application will not generate ARP requests to determine the destination MAC address of forwarded packets. An ipsecfwd_config command should be used to set these MAC addresses.

It is important to understand that the application can use only a subset of these interfaces at any one time. This is due to pin multiplexing rules on the P4080 SoC. The set of available interfaces is determined by a number of factors:

1. The interface set made available by the selected SerDes Protocol and u-boot variable "hwconfig". See the SDK document "NXP DPAA SDK X.Y: Selecting Ethernet Interfaces". This document is distributed with the DPAA SDK.
2. The Linux device tree determines which subset of the available interfaces is for use by USDPAA applications. See the USDPAA User Guide for more information.
3. Finally, the fmc configuration file passed by command line argument to ipsecfwd_app determines the subset of these interfaces that the application will attempt to initialize.

In the default SerDes 0xe example, the interfaces used by ipsecfwd_app are:

- FMan1-TGEC1
- FMan2-DTSEC3
- FMan2-DTSEC4
- FMan2-TGEC1

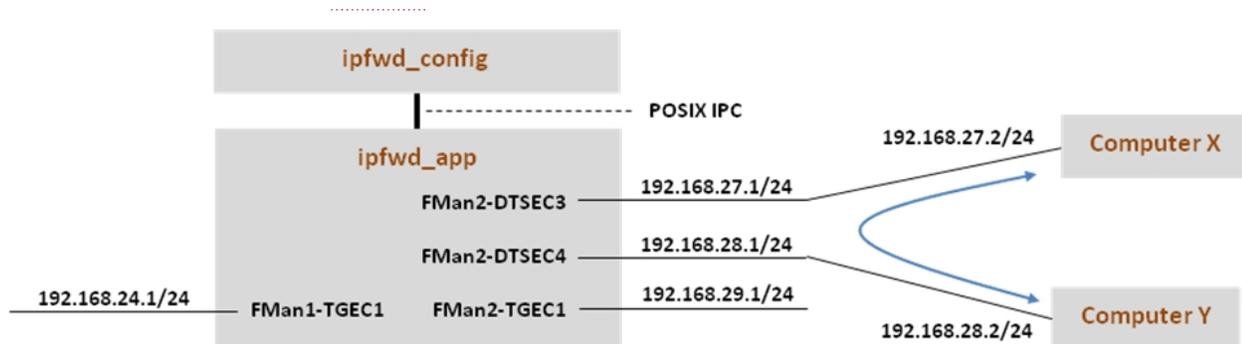
Running the ipsecfwd application suite involves four steps:

1. Run the fmc application to configure the FMan hardware instances.
2. Run ipsecfwd_app
3. Run ipsecfwd_config repeatedly to add SA Entries/routes.

Specific examples showing these steps are provided in other sections of this document.

74.75.1 Using Two Computers to Test the IPFWD Application Suite

This section describes how to configure the IPFWD application suite to forward packets between two ordinary computers as shown in the following figure. It is assumed that the two 1 Gbps Ethernet interfaces provided by SerDes 0xe are used.



Assign IP addresses to all the interfaces. The IP addresses used here for Computer X and Computer Y are examples. Their MAC addresses must be known as they will be needed in the commands below.

Keep in mind that `ipfwd_app` has no default IP address assigned to the interfaces. This means that you must first assign IP addresses to the interfaces and then use IP addresses for Computer X and Computer Y that are on the subnets shown in the figure. Please be careful with the network parameters. Any mistake will prevent packets from flowing from end to end.

Follow these steps on Computer X:

1. Set the ip address for eth0 interface

```
ifconfig eth0 192.168.27.2 up
```

Set the default gateway for subnet 192.168.27.1

```
route add default gw 192.168.27.1 eth0
```

2. Add arp for gw

```
arp -s 192.168.27.1 "MAC address for 192.168.27.1 on P4080"
```

Follow these steps on Computer Y:

1. Set the ip address for eth0 interface

```
ifconfig eth0 192.168.28.2 up
```

Set the default gateway for subnet 192.168.28.1

```
route add default gw 192.168.28.1 eth0
```

2. Add arp for gw

```
arp -s 192.168.28.1 "MAC address for 192.168.28.1 on P4080"
```

The commands to perform on P4080 are:

```
# Boot the P4080 and login as root.
```

```
Assign IP address to fm1-gb1
```

```
ifconfig fm1-gb1 <IPADD> up
```

```
cd /usr/etc
```

```
fmc -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_32_fq.xml -a
```

```
# Assume use of cores 1 - 7 ipfwd_app 1..7
```

```
# Now ssh to P4080 linux on other terminal
```

```
ssh root@<IPADD>
give IP address as assigned to fm1-gb1 in the beginning

# Now assign ip address to the interfaces
# First run command to check what all enabled interfaces are available
Note: check pid from application print "Message queue to send: /mq_snd_2536"
ipfwd_config -P 2536 -E -a true

Interface number: 11
PortID=1:5 is FMan interface node
with MAC Address
02:00:c0:a8:65:fe
Interface number: 9
PortID=1:3 is FMan interface node
with MAC Address
02:00:c0:a8:5b:fe
Interface number: 8
PortID=1:2 is FMan interface node
with MAC Address
02:00:c0:a8:51:fe
Interface number: 5
PortID=0:5 is FMan interface node
with MAC Address
02:00:c0:a8:33:fe
Are all the Enabled Interfaces

# Assign IP address to the interfaces. Use the same interface number displayed as an output on giving the above command

ipfwd_config -P 2536 -F -a 192.168.24.1 -i 5
ipfwd_config -P 2536 -F -a 192.168.28.1 -i 9
ipfwd_config -P 2536 -F -a 192.168.27.1 -i 8
ipfwd_config -P 2536 -F -a 192.168.29.1 -i 11

# Now enter routes and MAC addresses. Format of a MAC address is
# aa:bb:cc:dd:ee:ff where the letters are hexadecimal digits.

ipfwd_config -P 2536 -B -s 192.168.27.2 -d 192.168.28.2 -g 192.168.28.2
ipfwd_config -P 2536 -G -s 192.168.28.2 -m COMPUTER_Y_MAC_ADDRESS -r true
ipfwd_config -P 2536 -B -s 192.168.28.2 -d 192.168.27.2 -g 192.168.27.2
ipfwd_config -P 2536 -G -s 192.168.27.2 -m COMPUTER_X_MAC_ADDRESS -r true
# Computer X and Computer Y need to be told to route via the P4080.
# On Computer X (assuming it runs Linux), enter this command as root:
```

```
route add -net 192.168.28.0 netmask 255.255.255.0 gw 192.168.27.1
```

On Computer Y (assuming it runs Linux), enter this command as root:

```
route add -net 192.168.27.0 netmask 255.255.255.0 gw 192.168.28.1
```

Now, traffic can pass between Computer X and Computer Y. For example, on Computer X

enter:

```
ping 192.168.28.2
```

7.4.7.5.2 Running IPsecfd on P4080DS board

The instructions below describe how to run IPsecfd. Traffic should only be directed to the P4080DS once the application is running and configuration via the ipsecfd_config application is completed.

•On linux prompt, assign IP address to fm1-gb1

```
$ ifconfig fm1-gb1 <IPADD> up
```

•Configure FMan PCD using fmc with the XML files in /usr/etc:

```
$ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
$ fmc -c usdpaa_config_serdes_0xe.xml -p usdpaa_policy_hash_ipv4.xml -a
```

•Run IPsecfd application

The main IPsecfd application binary is called **ipsecfd_app**. The application can run on multiple cores as specified by the first parameter to the application. To run it to handle traffic distributed over 32 ingress frame queues per port:

```
$ ipsecfd_app <m..n>
```

By default ipsecfd_app uses usdpaa_config_serdes_0xe.xml and usdpaa_policy_hash_ipv4.xml files.

IPSECFWD application command syntax:

```
[root@p4080 etc]# ipsecfd_app --usage
Usage: ipsecfd_app [-n?V] [-c FILE] [-p FILE] [--fm-config=FILE]
      [--non-interactive] [--fm-pcd=FILE] [--cpu-range] [--help]
      [--usage] [--version] [cpu-range]
```

IPSECFWD application run command:

```
[root@p4080 root]# cd /usr/etc
<_config_serdes_0xe.xml -p usdpaa_policy_hash_ipv4.xml -a
[root@p4080 etc]# ipsecfd_app 1..7
[1] 5363
ipsecfd_app starting
Message queue to send: /mq_snd_2536
Message queue to receive: /mq_rcv_2536
```

If in the run application command, cpu-range is given i.e. ipsecfd_app <m..n> IPsecfd application starts threads on cpu-range m..n. The main thread (by default on CPU1), then does global initialization needed by the application, including starting other application threads.

If on the other hand run application command is given without any cpu-range i.e. `ipsecfwd_app` IPsecfwd application starts up with a single thread running on CPU 1 by default, which does global initialization needed by the application and enables all the network interfaces.

•Once the application starts it can receive the configuration commands. Run application configuration script

For creating SA entries, the binary `ipsecfwd_config` is run. This binary processes the configuration request from the user (using the command line) and populates the configuration via Linux standard posix IPC messages to the IPsecfwd application.

The shell script mentioned below contains sample commands to add SA entries.

SSH to p4080 linux on another terminal:

```
$ ssh root@<IPADD> (give the IP address as assigned to fm1-gb1 in the beginning)
```

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

```
ipsecfwd_20G.sh "pid"
```

```
$ ipsecfwd_enc_20G.sh 2536 or
$ ipsecfwd_dec_20G.sh 2536
```

One of the example shell scripts available is `ipsecfwd_enc_20G.sh` which creates SA Entries for encryption for only the 2 x 10G interfaces. They can assign IP addresses to the interfaces, add SA and ARP entries and assumes the netmask to be 255.255.255.0. The following table summarizes the settings done by this script.

Table 150. ipsecfwd_enc_20G.sh

Port ID	Source IP Address	Destination IP Address (for each src IP addr)	Source tunnel address	Destination tunnel address	Default Gateway IP Address
4	192.168.60.2 to 192.168.60.24	192.168.160.2 .. 24	192.168.60.2	192.168.60.1	192.168.160.2
9	192.168.160.2 to 192.168.160.24	192.168.60.2 .. 24	192.168.160.2	192.168.160.1	192.168.60.2

For the IPsecfwd application to send out traffic successfully, traffic destined for the P4080DS ports must have the appropriate source and destination addresses.

Console messages are printed for each entry added to the SA/routing table. Once all configuration is completed, the application moves to the packet processing phase - the message "Application Started successfully" is printed on the console. At this point, traffic can be sent to the IPsecfwd application which would do its processing on the cpu-range specified by the user on the application command-line.

7.4.7.5.3 Running IPsec forwarding on P3041/P5020 board

The instructions below describe how to run `IPsecfwd` on P3041/P5020. Traffic should only be directed to P3041/P5020 once the application is running and configuration via the `ipsecfwd_config` application is completed.

•On linux prompt, assign IP address to eth0

```
$ ifconfig eth0 <IPADD> up
```

- Configure FMan PCD *using fmc* with the XML files in /usr/etc:

```
$ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
$ fmc -c usdpaa_config_p3_p5_serdes_0x36.xml -p usdpaa_policy_hash_ipv4.xml -a
```

- Run IPsecfwd application

```
$ ipsecfwd_app <m..n> -c usdpaa_config_p3_p5_serdes_0x36.xml -p usdpaa_policy_hash_ipv4.xml
```

For P3041, m..n can be 0..3. For P5020, m..n can be 0..1.

SSH to p3041/p5020 linux on another terminal:

```
$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)
```

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

```
ipsecfwd_mix_15G.sh "pid"
```

```
$ ipsecfwd_mix_15G.sh 2536
```

This is an example shell script available which creates SA Entries for the 5 x 1G and 1x10G interfaces for back to back configuration.

7.4.7.5.4 Running IPsec forwarding on T4240 board

The instructions below describe how to run *IPsecfwd* on T4240. Traffic should only be directed to T4240 once the application is running and configuration via the *ipsecfwd_config* application is completed.

- On linux prompt, assign IP address to eth0

```
$ ifconfig eth0 <IPADD> up
```

- Configure FMan PCD *using fmc* with the XML files in /usr/etc:

```
$ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
$ fmc -c usdpaa_config_t4_serdes_1_1_6_6.xml -p usdpaa_policy_hash_ipv4.xml -a
```

- Run IPsecfwd application

```
$ ipsecfwd_app <m..n> -c usdpaa_config_t4_serdes_1_1_6_6.xml -p usdpaa_policy_hash_ipv4.xml -  
d 0x10000000 -b 0:0:1024
```

For T4240, m..n can be 0..23.

SSH to t4240 linux on another terminal:

```
$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)
```

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

```
ipsecfwd_mix_20G.sh "pid"
```

```
$ ipsecfwd_mix_20G.sh 2536
```

This is an example shell script available which creates SA Entries for the 2x10G interfaces for back to back configuration.

7.4.7.5.5 Running IPsec forwarding on B4860 board

The instructions below describe how to run *IPSecfwd* on B4860. Traffic should only be directed to B4860 once the application is running and configuration via the *ipsecfwd_config* application is completed.

- On linux prompt, assign IP address to eth0

```
$ ifconfig eth0 <IPADD> up
```

- Configure FMan PCD *using* *fmc* with the XML files in */usr/etc*:

```
$ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
$ fmc -c usdpaa_config_b4_serdes_0x2a_0x98.xml -p usdpaa_policy_hash_ipv4.xml -a
```

- Run IPsecfwd application

```
$ ipsecfwd_app <m..n> -c usdpaa_config_b4_serdes_0x2a_0x98.xml -p usdpaa_policy_hash_ipv4.xml
```

For B4860, m..n can be 0..7.

SSH to b4860 linux on another terminal:

```
$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)
```

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

```
ipsecfwd_mix_20G.sh "pid"
```

```
$ ipsecfwd_mix_20G.sh 2536
```

This is an example shell script available which creates SA Entries for the 2x10G interfaces for back to back configuration.

7.4.7.5.6 PPAC (and IPsecfwd) CLI commands

The following commands are illustrated in the context of IPsecfwd, but the commands are common to all PPAC-based applications.

The CLI (Command-Line Interface) allows you to add and remove additional threads to enable the use of multiple CPUs, with the only restriction being that the primary thread on CPU 1 cannot be removed (except by shutting down the application).

To add a thread on a single CPU (e.g. CPU 2):

```
> add 2
```

To add threads on a range of CPUs:

```
> add 3..6
```

To list the threads (this also queries each thread, verifying that they aren't blocked):

```
> list
```

```
Thread alive on cpu 1
```

```
Thread alive on cpu 2
```

```
Thread alive on cpu 3
```

```
Thread alive on cpu 4
```

```
Thread alive on cpu 5
```

```
Thread alive on cpu 6
```

To enable all interfaces

```
> macs on
```

To disable all interfaces

```
> macs off
```

To perform a controlled shutdown of ipsecfwd (this includes disabling the network ports):

```
> quit
```

7.4.7.5.7 IPsecfwd application Configuration command

7.4.7.5.7.1 Syntax

The syntax is as follows:

```
$ [root@p4080 bin]# ipsecfwd_config --help
Usage: ipsecfwd_config [OPTION...]
  -A --SAadd=TYPE      adding an SA entry
  -B, --routeadd=TYPE  adding a route
  -C, --routedel=TYPE  deleting a route
  -D --SAdel=TYPE      deleting an SA entry
  -E, --showintf=TYPE  show interfaces
  -F, --intfconf=TYPE  change intf config
  -G, --arpadd=TYPE    adding a arp entry
  -H, --arpdel=TYPE    deleting a arp entry
  -?, --help           Give this help list
      --usage          Give a short usage message
  -V, --version        Print program version
```

7.4.7.5.7.1.1 Command to show all enabled interfaces and their interface numbers

The command to show all the enabled interfaces while running IPv4 forward is as follows:

```
lpfwd_config -P pid -E -a true
```

Table 151. Field Description (show all enabled interfaces)

Parameter	Description	Mandatory	Format/ Value
-a	Show all the interfaces	Yes	true

Command to show all enabled interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# lpfwd_config -P 2536 -E -a true
```

```
Interface number: 11
PortID=1:5 is FMan interface node
with MAC Address
02:00:c0:a8:65:fe
Interface number: 9
PortID=1:3 is FMan interface node
with MAC Address
02:00:c0:a8:5b:fe
Interface number: 8
PortID=1:2 is FMan interface node
with MAC Address
02:00:c0:a8:51:fe
Interface number: 5
PortID=0:5 is FMan interface node
with MAC Address
02:00:c0:a8:33:fe
Are all the Enabled Interfaces [root@p4080 bin]#
```

7.4.75.7.1.2 Help for show all enabled interfaces command

The command to obtain help for show all enabled interfaces command is as follows:

```
ipsecfwd_config -E -help
Example 2. Help for show all enabled interfaces
[root@p4080 etc]# ipsecfwd_config -E --help
Usage: -E [OPTION...]
  -a, --a=ALL           All interfaces
  -?, --help            Give this help list
  --usage               Give a short usage message
  -V, --version         Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

7.4.75.7.1.3 Assign IP address to interfaces

The command to assign IP address while running IPsecfd is as follows:

```
ipsecfwd_config -P pid -F -a 192.168.60.1 -i <Interface number>
```

Note: The interface number to be used here must be one of the numbers that got displayed as the output of "show all enabled interfaces command" in section 2.8.1.1.

Table 152. Field description (assign IP address to interfaces)

Parameter	Description	Mandatory	Format/ Value
-a	IP Address	Yes	a.b.c.d

Table continues on the next page...

Table 152. Field description (assign IP address to interfaces) (continued)

-i	Interface number	Yes	0-11 (Choose this number from "show all enabled interfaces" command output)
----	------------------	-----	--

Example 3. Assign IP address to interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
ipsecfwd_config -P 2536 -F -a 192.168.60.1 -i 5
IPADDR assigned = 0xc0a83c01 to interface num 5
Intf Configuration Changed successfully
```

7.4.75.7.1.4 Help for assign IP address to interfaces

The command to obtain help for assign IP address to interfaces command is as follows:

```
ipsecfwd_config -F --help
```

Example 4. Help for assign IP address to interfaces

```
[root@p4080 etc]# ipsecfwd_config -F --help
Usage: -F [OPTION...]
  -a, --a=IPADDR           IP Address
  -i, --i=IFNAME           If Name
  -?, --help               Give this help list
      --usage               Give a short usage message
  -V, --version             Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

7.4.75.7.1.5 Adding an SA Entry

The command to add a SA while running IPsecfdw application is as follows:

```
ipsecfwd_config -P pid -A -a "AH SA configuration!" -e "encryption key" -s a.b.c.d -d b.c.d.e -g
c.d.e.f -G a.d.d.a -i 0 -r dir
```

Example 5. Adding an SA Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipsecfwd_config -P 2536 -A -a "AH SA configuration!" -e "This is 128 bits" -s
192.168.10.2 -d 192.168.60.2 -g 192.168.61.254 -G 192.168.60.99 -i 0 -r in
```

SA Entry Added successfully

```
[root@p4080 bin]#
```

For the purpose of using ESN (Extended Sequence Number) feature, user is provided two optional parameters. One is -x, which is intended to tell if user wants to use ESN option or not. Other is -v, which the user can configure with some starting sequence number for the packets. For example :

```
[root@p4080 bin]# ipsecfwd_config -P 2536 -A -a "AH SA configuration!" -e "This is 128 bits" -s 192.168.10.2 -d 192.168.60.2 -g 192.168.61.254 -G 192.168.60.99 -i 0 -r in -x 1 -v 4294967294
```

Table 153. Field Description (Adding an SA Entry)

Parameter	Description	Mandatory	Format/Value
-s	Source IP Address	Yes	a.b.c.d
-d	Destination IP Address	Yes	a.b.c.d
-g	Left tunnel IP Address	Yes	a.b.c.d
-G	Right Tunnel IP Address	Yes	a.b.c.d
-D	Default Gateway	No	a.b.c.d
-a	Authentication Key for HMAC-SHA1	No (Default key is taken if not supplied)	"string" of length 20 bytes (default = 0x2122_2324_2526_2728_292a_2b2c_2d2e_2f30_3132_3334;
-e	Encryption Key for AES-CBC	No (Default key is taken if not supplied)	"string" of length 16 bytes. (default = 0x0102_0304_0506_0708_090a_0b0c_0d0e_0f10
-i	SPI	Yes	Unsigned int
-r	Direction (encryption/ decryption)	Yes	in (decryption) or out (encryption)
-x	ESN option	No	Unsigned int
-v	Sequence number	No	Unsigned int

7.4.75.7.1.6 Help for SA Entry Addition

The command to help add a SA while running IPsecfwd application is as follows:

```
ipsecfwd_config -A --help
```

Example 6. Help for Adding an SA Entry

```
[root@p4080 bin]# ipsecfd_config -A --help
Usage: -A [OPTION...]
  -a, --a=AKEY           Authentication Key
  -d, --d=DESTIP         Destination IP
  -D, --dg=DEFGW        Default Gateway
  -e, --e=EKEY           Encryption Key
  -g, --ss=SRCGW         Source Gateway IP
  -G, --sd=SRCGW         Destination Gateway IP
  -i, --spi=SPI          SPI - 32 bit unsigned int
  -p, --p=PROTO          IPsec Proto type - ESP(0) {Default: 0}
  -r, --dir=DIR          DIR- in/ out
  -s, --s=SRCIP          Source IP
  -t, --t=ETYPE          Encryption Type - AES-CBC(0), 3DES-CBC(1)
                        {Default: 0}
  -v, --seq_num=SEQNUM   Sequence Number
  -x, --is_esn=ESN       Extended Sequence Number support
  -y, --y=ATYPE          Authentication Type - HMAC-SHA1(0) {Default: 0}
  -?, --help             Give this help list
      --usage            Give a short usage message
  -V, --version           Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

```
[root@p4080 bin]#
```

7.4.7.5.1.7 Deleting an SA Entry

The command to delete an SA while running IPsecfd is as follows:

```
ipsecfd_config -P pid -D -s 192.168.10.2 -d 192.168.60.2 -g 192.168.61.254 -G 192.168.60.99 -i 0
```

Example 7. Deleting an SA Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
root@p4080 bin]# ipsecfd_config -P 2536 -D -s 192.168.10.2 -d 192.168.60.2 -g 192.168.61.254 -G 192.168.60.99 -i 0
```

SA Entry Deleted successfully

```
[root@p4080 bin]#
```

Table 154. Field Description (Deleting an SA Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Source IP Address	Yes	a.b.c.d
-d	Destination IP Address	Yes	a.b.c.d
-g	Left tunnel IP Address	Yes	a.b.c.d

Table continues on the next page...

Table 154. Field Description (Deleting an SA Entry) (continued)

-G	Right Tunnel IP Address	Yes	a.b.c.d
-i	SPI	Yes	Unsigned int

7.4.7.5.7.1.8 1.4.1.8 Help for Deleting an SA Entry

The command to obtain help for SA entry deletion is as follows:

```
ipsecfwd_config -D --help
```

Example 8. Help for Deleting an SA Entry

```
[root@p4080 bin]# ipsecfwd_config -D --help
Usage: -D [OPTION...]
  -d, --d=DESTIP           Destination IP
  -g, --ss=SRCGW           Source gateway IP
  -G, --sd=DESTGW         Destination gateway IP
  -i, --spi=SPI           SPI
  -s, --s=SRCIP           Source IP
  -?, --help               Give this help list
  --usage                  Give a short usage message
  -V, --version            Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

```
[root@p4080 bin]#
```

7.4.7.5.7.1.9 Adding a Route Entry

The command to add a route while running IPsecfwd is as follows:

```
ipsecfwd_config -P pid -B -s a.b.c.d -d b.c.d.e -g c.d.e.f
```

Table 155. Field Description (Adding a Route Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Source IP Address	Yes	a.b.c.d
-d	Destination IP Address	Yes	a.b.c.d
-g	Gateway IP Address	Yes	a.b.c.d

Example 9. Adding a Route Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipsecfwd_config -P 2536 -B -s 192.168.29.2 -d 192.168.24.2 -g 192.168.24.2
```

Route Entry Added successfully

```
[root@p4080 bin]#
```

7.4.7.5.7.1.10 Help for Route Entry Addition

The command to obtain help for route entry addition is as follows:

```
ipsecfwd_config -B --help
```

Example 10. Help for Adding a Route Entry

```
[root@p4080 bin]# ipsecfwd_config -B --help
Usage: -B [OPTION]
  -d, --d=DESTIP           Destination IP
  -f, --f=FLOWID           Flow ID - (0 - 1024) {Default: 0}
  -g, --g=GWIP             Gateway IP
  -s, --s=SRCIP            Source IP
  -t, --t=TOS              Type of Service - (0 - 256) {Default: 0}
  -?, --help               Give this help list
      --usage              Give a short usage message
  -V, --version            Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

7.4.7.5.7.1.11 Deleting a Route Entry

The command to delete a route while running IPSecfwd is as follows:

```
ipsecfwd_config -P pid -C -s a.b.c.d -d b.c.d.e
```

Table 156. Field Description (Deleting a Route Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Source IP Address	Yes	a.b.c.d
-d	Destination IP Address	Yes	a.b.c.d

Example 11. Deleting a Route Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipsecfwd_config -P 2536 -C -s 192.168.29.2 -d 192.168.24.2 Route Entry
Deleted
successfully
```

7.4.7.5.7.1.12 Help for Deleting a Route Entry

The command to obtain help for route entry deletion is as follows:

```
ipsecfwd_config -C --help
```

Example 12. Help for Deleting a Route Entry

```
[root@p4080 bin]# ipsecfwd_config -C --help
Usage: -C [OPTION...]
  -d, --d=DESTIP           Destination IP
  -s, --s=SRCIP            Source IP
  -t, --t=TOS              Type of Service - (0 - 256) {Default: 0}
  -?, --help               Give this help list
      --usage              Give a short usage message
  -V, --version            Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

7.4.75.7.1.13 Adding an ARP Entry

The command to add an ARP entry while running IPsecfwd is as follows:

```
ipsecfwd_config -P pid -G -s a.b.c.d -m aa:bb:cc:dd:ee [-r true]
```

Table 157. Field Description (Adding an ARP Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Gateway IP Address	Yes	a.b.c.d
-m	Mac Address	Yes	aa:bb:cc:dd:ee
-r	Replace existing entry	No	true/ false {Default: false}

Example 14. Adding an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipsecfwd_config -P 2536 -G -s 192.168.24.2 -m 02:00:c0:a8:33:fd -r true
```

ARP Entry Added successfully

7.4.75.7.1.14 Help for ARP Entry Addition

The command to obtain help for ARP entry addition is as follows:

```
ipsecfwd_config -G --help
```

Example 15. Help for Adding an ARP Entry

```
[root@p4080 etc]# ipsecfwd_config -G --help
Usage: -G [OPTION...]
  -m, --m=MACADDR         MAC Address
  -r, --r=Replace         Replace Exiting Entry - true/ false {Default: false}
  -s, --s=IPADDR          IP Address
  -?, --help              Give this help list
      --usage              Give a short usage message
  -V, --version            Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

7.4.75.7.15 Deleting an ARP Entry

The command to delete an ARP while running the IPsecfd is as follows:

```
ipsecfwd_config -P pid -H -s a.b.c.d
```

Table 158. Field Description (Deleting an ARP Entry)

Parameter	Description	Mandatory	Format/ Value
-s	Gateway IP Address	Yes	a.b.c.d

Example 16. Deleting an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipsecfd_config -P 2536 -H -s 192.168.24.2  
Arp Entry Deleted successfully  
[root@p4080 bin]#
```

7.4.75.7.16 Help for Deleting an ARP Entry

The command to obtain help for ARP entry deletion is as follows:

```
ipsecfwd_config -H --help
```

Example 17. Help for Deleting an ARP Entry

```
[root@p4080 etc]# ipsecfd_config -H --help  
Usage: -H [OPTION...]  
-s, --s=IPADDR          IP Address  
-?, --help              Give this help list  
--usage                 Give a short usage message  
-V, --version           Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

7.4.75.7.17 Adding a high bandwidth tunnel

A high bandwidth tunnel is same as normal tunnel except that it has capability to carry high volume of traffic. The purpose of this feature is to provide high throughput when only single IPsec tunnel is created. For non-high bandwidth tunnel only a single core can process a tunnel's packet at any point of time. High bandwidth tunnel option allows multiple cores to process in parallel the packets of a single tunnel.

To create a high bandwidth tunnel "-b 1" should be appended to command for creating a new security association as shown below:

```
ipsecfwd_config -P pid -b 1
```

Table 159. Field Description (Create a high bandwidth tunnel)

Parameter	Description	Mandatory	Format/ Value
-b	High bandwidth tunnel enable	No	1/0 [enable/ disable (default: false)]

When a tunnel is in high bandwidth mode, it should show a higher throughput

7.4.7.6 References

1. USDPAAs PPAC User Guide
2. QMan/BMan API Guide

7.4.7.7 Revision History

Document revision history.

Table 160. Revision history

Version	Author	Description
1.0	Nipun Gupta	Initial Draft

7.4.8 NXP Simple Crypto User Manual

7.4.8.1 Introduction

About this Document

The User Space Data Path Acceleration Architecture (USDPAAs) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document provides the following:

- A summary of the USDPAAs "simple crypto" application.
- Execution steps for the "simple crypto" application.

Conventions

This document uses the following conventions:

`Courier` is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.

7.4.8.2 USDPAAs Simple Crypto Application

7.4.8.2.1 Overview

USDPAAs Crypto Application is a multi-threaded Linux User space process. This application exercises the interface to the SEC4.0 engine for raw mode encryption, decryption, and authentication of data. Using the P4080 DPAA, the application generates traffic for the SEC4.0 engine and also processes the output from the SEC4.0 engine. The application also generates performance data for its SEC4.0 interactions.

The applications' threads are created using the standard pthreads library. In Linux User Space, any of the 8 cores can be configured to run a USDPAAs Crypto application thread. A dedicated QMan software portal is assigned to a USDPAAs application thread and each thread is affined to a core. Each core has its own dedicated Frame Queues to interact with the SEC4.0 block. Traffic is directed to the SEC4.0 via frame descriptor enqueues onto QMan frame queues, which are configured to deliver the frames to SEC4.0 engine for processing.

The SEC4.0 engine processes packets on the basis of commands passed in the form of a shared descriptor. A pointer to the shared descriptor is passed to the SEC4.0 in the ingress (towards the security engine) frame queue descriptors' *contextA* field. The egress FQID is used by SEC4.0 to return output to the application - this is passed in the *contextB* field of the ingress frame queue descriptor. Different SEC4.0 shared descriptors are created for different operation like encryption and decryption.

7.4.8.2.2 Parameters to the application

The crypto application supports various runtime parameters. These configurable parameters are passed to the crypto application from the command line while running the application.

1. Mode: Mode can be PERF or CIPHER
 - a. **PERF** Mode: In PERF or Performance mode the application calculates the throughput of SEC4.0 processing of data (including enqueue and dequeue operations to and from SEC4.0 block).
 - b. **CIPHER** mode: CIPHER mode allows a test run to demonstrate the SEC4.0 throughput and also compares ciphertext generated by SEC4.0 with ciphertext of a standard test vector.
2. Algorithm: This argument specifies the Algorithm to perform on the data. It is passed to SEC4.0 block using a Shared Descriptor. The algorithm choices are documented in Section [Simple Crypto command syntax](#) on page 970.
3. Number of Iterations: This parameter specifies the number of iterations of data to be looped through the SEC4.0 engine in a test run.
4. Number of buffers: It specifies the total number of buffers to send to SEC4.0 block from each core in one encryption/ decryption or authentication iteration. These buffers are distributed among each core.
5. Size of the buffer: This argument is used only with PERF mode. It specifies the size of the each input buffers sent to SEC4.0.
6. Test set number: This argument is used only with CIPHER mode. It specifies the predefined test set to be used as the data set to send to SEC4.0. There are various test sets (with varying size and data) hardcoded in the application. These are documented in the Section [Simple Crypto command syntax](#) on page 970.
7. Number of cores: This is an optional parameter. By default the application uses all the active cores for the processing of the data sets. By specifying the number of cores, the user can limit the application threads to a specific number of cores.

7.4.8.2.3 Packet Flow

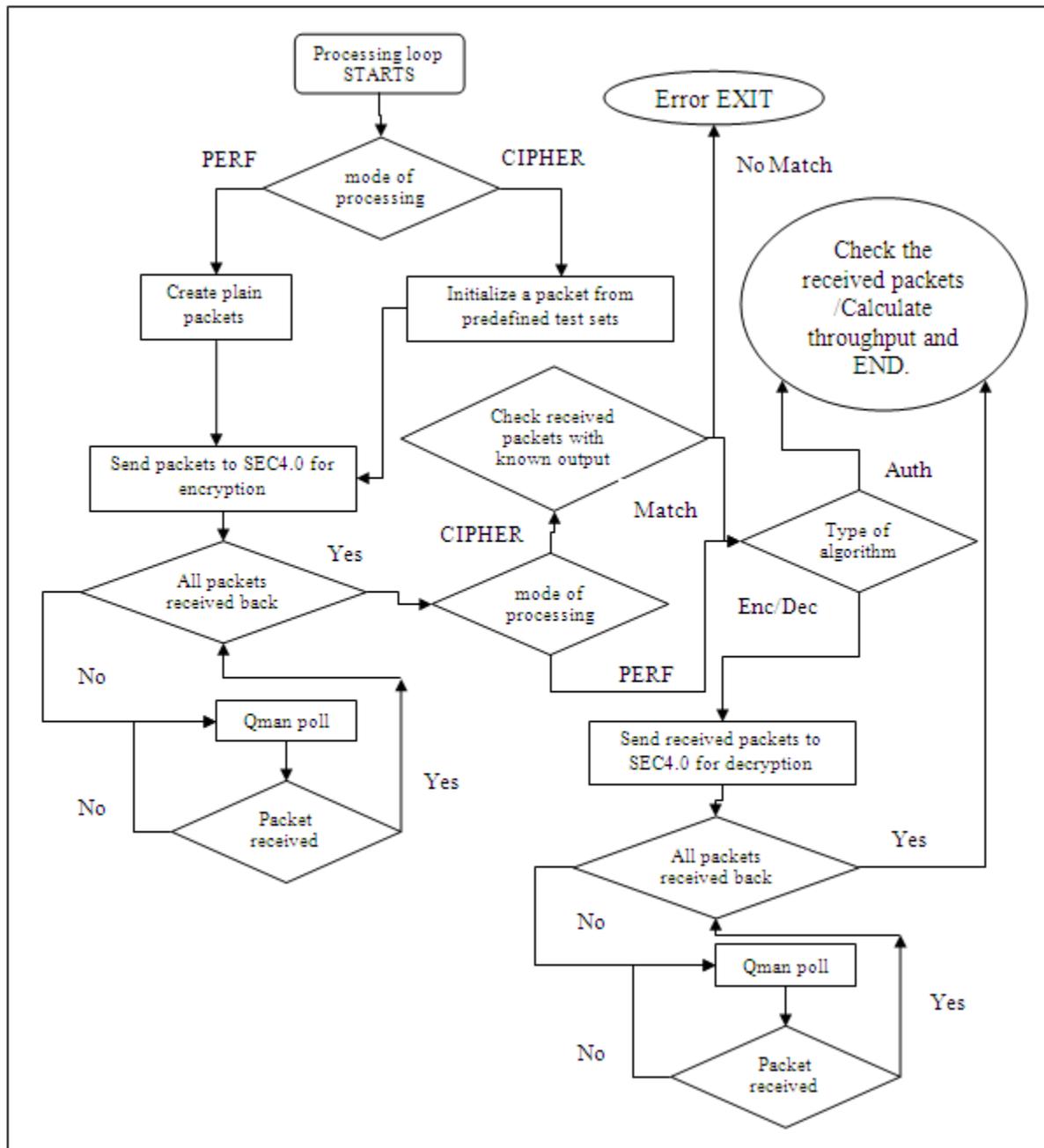


Figure 151. Packet Flow

After initializing the dedicated ingress and egress frame queues for a SEC4.0 operation, the application creates compound frame descriptors (FD's) and fills plain text or pre-defined "cipher" text in input buffers for SEC processing on the basis of the specified mode of operation. For PERF mode, this is plain text; and in the case of CIPHER mode it is pre-defined "cipher" text. The number of FD's equals to the total number of buffers. The FD's, which contain input and output buffer pointers, are first distributed among the cores and then enqueued to the ingress FQ's. The application thread then polls for output packets from the SEC4.0 egress frame queue until all the packets are received back after being encrypted. As the application uses dedicated Frame Queues between cores and SEC4.0, each core receives the packets which it has enqueued.

In case of CIPHER mode the application checks the received encrypted packets with the already known result.

Lastly, the application checks for the algorithm type. If the algorithm is authentication, then it calculates the throughput in millions of bits per second (Mbps) and exits. If the algorithm type is encryption/decryption, the application sends the same packet (which it received from SEC4.0 after encryption) back to SEC4.0 block for decryption. It does this by changing the pointers of output buffers to point to input buffers and vice-versa in FD's and enqueues these to the SEC4.0's ingress FQ's. It then polls the packet from the SEC4.0 egress frame queue until all the encrypted packets are received back after decryption. The application checks if the packet received after decryption is same as the original plain/cipher text (as a packet after encryption followed by decryption should be the same as the original packet). It then calculates the throughput in Mbps and exits.

7.4.8.2.4 Throughput calculation

The application measures the CPU cycles just before enqueueing the first packet on the FQ and just after receiving the last packet after processing from SEC4.0 for each iteration. The difference between these is the 'delta_cycles' which is accumulated over all the iterations.

Throughput of the application is reported in millions of bits per second (Mbps).

Throughput calculation involves the following parameters.

- 'l' is the number of iterations the application runs for in a test run
- 'n' is the total number of buffers
- 's' is the size of buffer
- 'cpu_freq' is the CPU frequency in MHz

The cycles per frame equals:

$$\text{cycles_per_frame} = (\text{delta_cycles}) / (l * n);$$

Throughput in Mbps equals:

$$\begin{aligned} \text{Throughput} &= (\text{cpu_freq} * \text{bits_per_byte} * s) / (\text{cycles_per_frame}); \\ &= (\text{cpu_freq} * 8 * s) / (\text{cycles_per_frame}); \end{aligned}$$

7.4.8.2.5 Running Simple Crypto Application on board

1. On the Linux prompt on USDPAAs, run the application by typing the following command:

```
simple_crypto -m <mode> -s <size> -n <num_buffer> -o <algo> -l <num_iterations> -t <test_set> [-c <num_cores>]
```

Refer to section [Simple Crypto command syntax](#) on page 970 for command syntax.

2. Upon successful completion, the application shows the following message on the USDPAAs boot core's console. In case of a failure, a failure message is displayed

```
INFO: SEC4.0 test PASSED
```

Also upon successful completion, the application reports SEC4.0 raw algorithm's throughput on boot core's console.

7.4.8.2.6 Simple Crypto command syntax

The command syntax is as follows:

```
[root@p4080 root]# simple_crypto --help
```

Usage: simple_crypto [OPTION...]

-c, --ncpus=CPUS

OPTIONAL PARAMETER

Number of cpus to work for the Application (1-8)(OPTIONAL)

-l, --itrnum=ITERATIONS

Number of iteration to repeat

-m, --mode=TEST MODE

test mode: specify one of the following

1 for perf

2 for cipher

Only the following two combinations are valid. All options are mandatory:

-m 1 -s <buf_size> -n <buf_num_per_core>

-o <algo> -l <itr_num>

or

-m 2 -t <test_set> -n <buf_num_per_core>

-o <algo> -l <itr_num>

-n, --bufnum=TOTAL BUFFERS

Number of buffers per core (1-6400)

-o, --algo=ALGORITHM

Cryptographic operation to be performed by SEC4.0

Specify one of the following:

1 for AES_CBC

2 for TDES_CBC

3 for SNOW_F8

4 for SNOW_F9

5 for KASUMI_F8

6 for KASUMI_F9

7 for CRC

8 for HMAC_SHA1

9 for SNOW_F8_F9(only with PERF mode)

-s, --bufsize=BUFFER SIZE

OPTION IS VALID ONLY IN PERF MODE

Buffer size (64, 128 ...upto 6400)

-t, --testset=TEST SET

OPTION IS VALID ONLY IN CIPHER MODE

provide following test set number:

AES_CBC: 1-4

TDES_CBC: 1-2

SNOW_F8: 1-5

SNOW_F9: 1-5

KASUMI_F8: 1-5

KASUMI_F9: 1-5

The USDPAA Simple Proto application demonstrates the usage of security coprocessor's capabilities in handling traffic in security protocols context

This document provides the following:

- A summary of the USDPAA "simple proto" application.
- Execution steps for running "simple proto" application.

Conventions

This document uses the following conventions:

`Courier` is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.

7.4.9.2 USDPAA Simple Proto Application

7.4.9.3 Overview

USDPAA Simple Proto application demonstrates usage of SEC engine protocols' integrity and confidentiality algorithms. Using the DPAA framework, the application generates traffic and enqueues it to SEC engine, process the output and generate performance data for its SEC interactions.

This is a multi-threaded Linux User Space application. Threads are created using pthreads library, each application thread has an assigned QMan software portal and is affined to a core. Each core has its own dedicated Frame Queues to interact with SEC. Traffic is injected to SEC via frame descriptors enqueued onto QMan frame queues.

The SEC engine processes packets on the basis of commands passed in the form of a shared descriptor. A pointer to the shared descriptor is passed to the SEC in the ingress frame queue descriptors' *contextA* field. The egress FQID is used by SEC to return output to the application - this is passed in the *contextB* field of the ingress frame queue descriptor. Different SEC shared descriptors are created for different protocols' operation.

7.4.9.4 Parameters to the application

The `simple_proto` application supports various runtime parameters. These configurable parameters are passed to the application from the command line when running the application.

1. Mode: Mode can be PERF or CIPHER
 - a. **PERF** Mode: In PERF or Performance mode the application calculates the throughput of SEC processing of data (including enqueue and dequeue operations to and from SEC block). Also, output frames from decapsulation are compared against input frames to encapsulation.
 - b. **CIPHER** mode: CIPHER mode allows a test run to demonstrate the SEC throughput and also compares ciphertext generated by SEC with ciphertext of a standard test vector.
2. Protocol: This argument specifies the protocol to be tested. It is passed to SEC block using a Shared Descriptor. The protocol choices are documented in Section [Simple Proto command syntax](#) on page 976. Each protocol has its own set of mandatory and/or optional parameters, which are explained in [MACSec protocol options](#) on page 978, [WiMAX protocol options](#) on page 978, [PDCP protocol options](#) on page 978 and [MBMS Protocol Options](#) on page 981
3. Number of Iterations: This parameter specifies the number of iterations of data to be looped through the SEC engine in a test run.
4. Number of buffers: This specifies the total number of buffers to send to SEC block from each core in one encryption/decryption or authentication iteration. These buffers are distributed among each core.
5. Size of the buffer: This argument is used only with PERF mode. It specifies the size of the each input buffers sent to SEC.

6. Test set number: This argument is used only with CIPHER mode. It specifies the predefined test set to be used as the data set to send to SEC. There are various test sets (with varying size and data) hardcoded in the application. These are documented in the Section [Simple Proto command syntax](#) on page 976.
7. Number of cores: This is an optional parameter. By default the application uses all the active cores for the processing of the data sets. By specifying the number of cores, the user can limit the application threads to a specific number of cores.
8. SEC Era: This is an optional parameter. This specifies the SEC era hardware block revision for which SEC descriptors will be generated. **By default, the application runs with default era set to value 2.** By specifying the SEC era, the user can set the right value for the targeted platform to test. For example, SEC era on the following platforms is:
 - 2 for P4080 TO2
 - 3 for P3041, P5020
 - 4 for P4080 TO3
 - 5 for P5040, B4860
 - 6 for T4240, T2080 and T1040
 - 7 for LS1021

7.4.9.5 Packet Flow

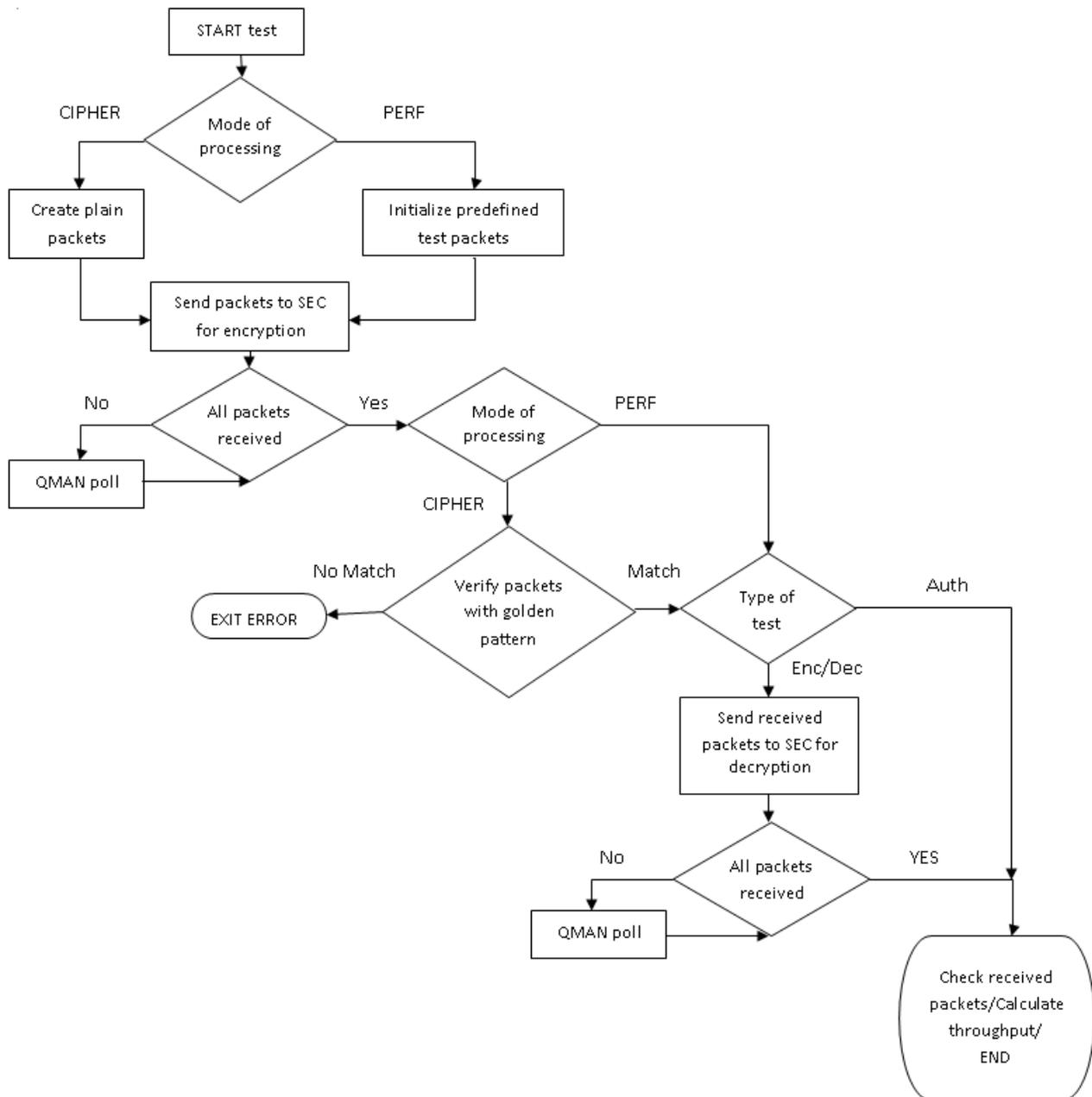


Figure 153. Packet Flow

After initializing the dedicated ingress and egress frame queues for a SEC operation, the application creates compound frame descriptors (FD's) and prepare input buffers for SEC processing based on specified mode of operation; for PERF mode, buffers are filled in with incrementing pattern plain text (for WiMAX encapsulation, SEC expects GMH aware input frames: GMH Header Type bit set to zero, the CRC indicatr bit set to one if CRC is included in the PDU, and the GMH LEN field updated accordingly to the input plain data length), and in the case of CIPHER mode, with golden pattern data. The number of FD's equals to the total number of buffers and are first distributed among the cores and then enqueued to the ingress FQ's. The application thread then polls for output packets from the SEC egress frame queue until all the packets are received back after being encapsulated. As the application uses dedicated Frame Queues between cores and SEC, each core receives the

packets which it has enqueued. In CIPHER test mode, the application checks the received encapsulated packets against a golden pattern.

The next step is the application to verify the type of test. If the test set is unidirectional, then it calculates the throughput in millions of bits per second (Mbps) and exits. If the test verifies both encapsulation/decapsulation, the application sends the packet which it received from SEC after encapsulation back to SEC block for decapsulation. It does this by interchanging the pointers to output and input buffers in FD's and enqueues these to the SEC's ingress FQ's. It then polls the packet from the SEC egress frame queue until all the encapsulated packets are received back after decapsulation. The application checks if the packet received after decapsulation is the same as the original plain/cipher text (as a packet after encapsulation followed by decapsulation should be the same as the original packet). It then calculates the throughput in Mbps and exits.

7.4.9.6 Throughput calculation

The application measures the CPU cycles just before enqueueing the first packet on the FQ and just after receiving the last packet after processing from SEC for each iteration. The difference between these is the 'delta_cycles' which is accumulated over all the iterations.

Throughput of the application is reported in millions of bits per second (Mbps).

Throughput calculation involves the following parameters.

- 'l' is the number of iterations the application runs for in a test run
- 'n' is the total number of buffers
- 's' is the size of buffer
- 'cpu_freq' is the CPU frequency in MHz

The cycles per frame equals:

$$\text{cycles_per_frame} = (\text{delta_cycles}) / (l * n);$$

Throughput in Mbps equals:

$$\text{Throughput} = (\text{cpu_freq} * \text{bits_per_byte} * s) / (\text{cycles_per_frame});$$
$$= (\text{cpu_freq} * 8 * s) / (\text{cycles_per_frame});$$

7.4.9.7 Running Simple Proto Application on board

1. On the Linux prompt on USDPAAs, run the application by typing the following command:

```
simple_proto -m <mode> -s <size> -n <num_buffer> -p <protocol> -l <num_iterations> -t <test_set> [-c <num_cores> -e <sec_era>]
```

Refer to section [Simple Proto command syntax](#) on page 976 for command syntax.

2. Upon successful completion, the application shows the following message on the USDPAAs boot core's console:

```
INFO: SEC4.0 test PASSED
```

Also upon successful completion, the application reports SEC4.0 raw algorithm's throughput on boot core's console.

In case of failure, a failure message is displayed on console.

7.4.9.8 Simple Proto command syntax

The command syntax is as follows:

```
root@p4080ds:~# simple_proto --help
Usage: simple_proto [OPTION...]

-c, --ncpus=CPUS           OPTIONAL PARAMETER
```

```

                                Number of cpus to work for the application(1-8)

-e, --sec_era=ERA                OPTIONAL PARAMETER

                                SEC Era version on the targeted platform(2-5)

-l, --itrnum=ITERATIONS          Number of iterations to repeat

-m, --mode=TEST MODE            Test mode:
                                1 for perf
                                2 for cipher

                                Following two combinations are valid only and all
                                options are mandatory:
                                -m 1 -s <buf_size> -n <buf_num_per_core> -p
                                <proto> -l <itr_num>
                                -m 2 -t <test_set> -n <buf_num_per_core> -p
                                <proto> -l <itr_num>

-n, --bufnum=TOTAL BUFFERS      Total number of buffers (1-6400). Both of Buffer
                                size and buffer number cannot be greater than 3200
                                at the same time.

-p, --proto=PROTOCOL            Cryptographic operation to perform by SEC:
                                1 for MACsec
                                2 for WiMAX
                                3 for PDCP
                                4 for SRTP
                                5 for WiFi
                                6 for RSA
                                7 for TLS
                                8 for IPsec
                                9 for MBMS

-s, --bufsize=BUFSIZE           OPTION IS VALID ONLY IN PERF MODE

                                Buffer size (64, 128 ... up to 6400). Note: Both
                                of Buffer size and buffer number cannot be greater
                                than 3200 at the same time.
                                The WiMAX frame size, including the FCS if
                                present, must be shorter than 2048 bytes.

-t, --testset=TEST SET          OPTION IS VALID ONLY IN CIPHER MODE

-?, --help                      Give this help list
--usage                          Give a short usage message

```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

NOTE

Depending on the hardware platform, the ncpus(-c) varies as follows: for p3041 between 1-4, for p5020 between 1-2, for B4860 between 1-8 and for T4240 between 1-24.

NOTE

The valid test set numbers are the following, per each protocol:

1. MACSec - 1 .. 5
 2. WiMAX - 1 .. 4
 3. PDCP - 1
 4. SRTP - 1
 5. WiFi - 1 .. 2
 6. RSA - 1 .. 2
 7. TLS - 1
 8. IPsec - 1
 9. MBMS:
 - a. MBMS PDU Type 0 - 1 .. 2
 - b. MBMS PDU Type 1 - 1 .. 3
 - c. MBMS PDU Type 3 - 1 .. 3
-

7.4.9.9 MACSec protocol options

For MACSec processing, the simple_proto application understands the following parameters (apart from the mandatory & optional ones specified in [Simple Proto command syntax](#) on page 976):

- -o, --algo : this is an optional parameter that, when set, allows the user to choose the cipher type. The cipher type can be GCM or GMAC; by default, the MACSec protocol will use GCM processing.

7.4.9.10 WiMAX protocol options

WiMAX processing is available only if SEC Era if equal or greater than 4.

For WiMAX processing, the simple_proto application understands the following parameters (apart from the mandatory & optional ones specified in [Simple Proto command syntax](#) on page 976):

- -a, --ofdma : this is an optional parameter that, when set, it enables OFDMA processing for WiMAX. By default, the WiMAX protocol offload does OFDM processing;
- -f, --fcs : this is an optional parameter that, when set, instructs the WiMAX protocol offload to compute the FCS over the input frame, making it longer by 4 bytes;
- -w, --ar_len=ARWIN : another optional parameter that enables the anti-replay mechanism in WiMAX protocol processing. This parameter also sets the anti-replay window length, which cannot exceed 64 frames.

7.4.9.11 PDCP protocol options

Packet Data Convergence Protocol (abbrev. PDCP) is one of the layers of the Radio Traffic Stack in UMTS and performs IP header compression and decompression, transfer of user data and maintenance of sequence numbers for Radio Bearers which are configured for lossless serving radio network subsystem (SRNS) relocation.

In simple_proto application, the following protocol sub-sets are tested & supported:

1. PDCP Control Plane;
2. PDCP User Plane;
3. PDCP Short MAC.

The PDCP ciphering & integrity algorithm combinations supported by simple_proto application are the following:

1. PDCP Control Plane:
 - a. NULL encryption & NULL integrity (EEA0/EIA0)
 - b. NULL encryption & SNOW f9 integrity (EEA0/EIA1)
 - c. NULL encryption & AES CMAC integrity (EEA0/EIA2)
 - d. NULL encryption & ZUC integrity (EEA0/EIA3)*
 - e. SNOW f8 encryption & NULL integrity (EEA1/EIA0)
 - f. SNOW f8 encryption & SNOW f9 integrity (EEA1/EIA1)
 - g. SNOW f8 encryption & AES CMAC integrity (EEA1/EIA2)
 - h. SNOW f8 encryption & ZUC integrity (EEA1/EIA3)*
 - i. AES CTR encryption & NULL integrity (EEA2/EIA0)
 - j. AES CTR encryption & SNOW f9 integrity (EEA2/EIA1)
 - k. AES CTR encryption & AES CMAC integrity (EEA2/EIA2)
 - l. AES CTR encryption & ZUC integrity (EEA2/EIA3)*
 - m. ZUC encryption & NULL integrity (EEA3/EIA0)
 - n. ZUC encryption & SNOW f9 integrity (EEA3/EIA1)
 - o. ZUC encryption & AES CMAC integrity (EEA3/EIA2)
 - p. ZUC encryption & ZUC integrity (EEA3/EIA3)*
2. PDCP User Plane:
 - a. NULL encryption (EEA0)
 - b. SNOW f8 encryption (EEA1)
 - c. AES CTR encryption (EEA2)
 - d. ZUC encryption (EEA3)*
3. PDCP Short MAC:
 - a. NULL integrity (EIA0)
 - b. SNOW f9 integrity (EIA1)
 - c. AES CMAC integrity (EIA2)
 - d. ZUC integrity (EIA3)*

NOTE

Starred combinations above are available only for platforms with SEC ERA greater than 4 (for instance P5040/B4860R1&R2/T4240/etc.). Attempting to run these combinations on platforms without the proper SEC ERA version will result in a SEC error.

NOTE

For the following combinations used for decapsulating PDCP PDUs, the SEC will return an error code similar to 0x3000XX0a if the last 4 bytes of the decapsulated frame (the ICV) are not set to the value of {0x00, 0x00, 0x00, 0x00}

1. EEA1/EIA0
2. EEA2/EIA0
3. EEA3/EIA0

The following parameters can be provided to the simple_proto application (besides the optional & mandatory parameters specified in [Simple Proto command syntax](#) on page 976):

Parameter	Explanation	Valid for Control Plane?	Valid for User Plane?	Valid for Short MAC?
-d, --direction	Selects the downlink direction for the inserted PDU; by default, the direction of the PDU is assumed to be uplink.	Yes, optional	Yes, optional	No
-i, --integrity	Selects the integrity algorithm to be used for processing the PDU	Yes, mandatory	No	Yes, mandatory
-r, --cipher	Selects the ciphering algorithm to be used for processing the PDU	Yes, mandatory	Yes, mandatory	No
-v, --hfn_ov	Enables the HFN value used for processing the PDU to be specified by the user.	Yes, optional	Yes, optional	No
-x, --snlen	Select the User Plane PDUs sequence number length. Three values are permitted: 0 = 12 bit Sequence Number PDU 1 = 7 bit Sequence Number PDU 2 = 15 bit Sequence Number PDU	No	Yes, optional	No
-y, --type	Selects the way the input PDU is to be treated: 0 = Control Plane 1 = User Plane 2 = Short MAC	Yes, mandatory	Yes, mandatory	Yes, mandatory

7.4.9.12 RSA operations options

For RSA processing, the simple_proto application understands the following parameter (apart from the mandatory & optional ones specified in [Simple Proto command syntax](#) on page 976):

- b, --form : this is an optional parameter that, when set, allows the user to choose one of the three RSA Decrypt Private Key formats:
 - 1 = Form 1 (default)
 - 2 = Form 2
 - 3 = Form 3

7.4.9.13 TLS protocol options

Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), are cryptographic protocols designed to provide communication security over the Internet.

simple_proto application tests and supports the TLS10 security protocol.

The following parameters can be provided to the simple_proto application (besides the optional & mandatory parameters specified in [Simple Proto command syntax](#) on page 976):

Parameter	Explanation	Optional / Mandatory
-j, --cipher	Selects the ciphering algorithm to be used for processing the PDU: 0 = AES-CBC	mandatory
-k, --integrity	Selects the integrity algorithm to be used for processing the PDU: 0 = HMAC-SHA1	mandatory
-g, --version	Select the SSL protocol version to run: 0 = SSL30 (not supported) 1 = TLS10 2 = TLS11 (not supported) 3 = TLS12 (not supported) 4 = DTLS10 (not supported)	mandatory

7.4.9.14 IPsec protocol options

Internet Protocol Security (IPsec) is a protocol suite for securing Internet Protocol (IP) communications by authenticating and encrypting each IP packet of a communication session.

simple_proto application tests and supports 3des & hmac-md5-96 IPsec ESP Tunnel mode.

The following parameters can be provided to the simple_proto application (besides the optional & mandatory parameters specified in [Simple Proto command syntax](#) on page 976):

Parameter	Explanation	Optional / Mandatory
-h, --cipher	Selects the ciphering algorithm to be used for processing the PDU: 0 = 3DES	mandatory
-q, --integrity	Selects the integrity algorithm to be used for processing the PDU: 0 = HMAC_MD5_96	mandatory

7.4.9.15 MBMS Protocol Options

MBMS SYNC protocol is defined in 3GPP TS 25.446 - MBMS synchronisation protocol (SYNC)

The MBMS Synchronisation protocol (SYNC) is located in the User plane of the Radio Network layer over the lu interface: the lu UP protocol layer.

The SYNC protocol for UTRAN is used to convey userdata associated to MBMS Radio Access Bearers.

The simple_proto application supports checking for the CRC validity of the following MBMS PDU Types:

1. MBMS PDU Type 0
2. MBMS PDU Type 1
3. MBMS PDU Type 3

The following table summarizes the behavior of the MBMS SYNC processing:

MBMS SYNC PDU	Default action	Header CRC fail action	Payload CRC fail action(s)
Type 0	Copy PDU	Drop PDU	N/A
Type 1	Copy PDU	Drop PDU	1. Update Payload CRC in PDU's header 2. Copy Header only
Type 3	Copy PDU	Drop PDU	1. Update Payload CRC in PDU's header 2. Copy Header only

The following table lists SEC return codes used for signaling the different actions the SEC takes in order to process the MBMS SYNC PDUs:

Processing result	SEC status/command
PDU Header & Payload CRC OK	0x0000_0000
Wrong PDU Header CRC	0x3000_XXAA
Wrong PDU Payload CRC	0x3000_XXAB

NOTE

The "XX" in the above rows is an internal offset used by SEC and can be safely masked out when checking the SEC status.

The following parameters can be provided to the simple_proto application (besides the optional & mandatory parameters specified in [Simple Proto command syntax](#) on page 976

Parameter	Explanation	Valid values
-z, --type	Selects the MBMS PDU Type to be processed	<ul style="list-style-type: none"> • 0 - MBMS PDU Type 0 • 1 - MBMS PDU Type 1 • 3 - MBMS PDU Type 2

7.4.10 SEC Descriptor construction library (DCL)

7.4.10.1 SEC Descriptor Construction Library (DCL)

A description of the SEC descriptor construction library (DCL) as a library within USDPAA.

The following is a description of the SEC descriptor construction library (DCL) as a library within USDPAA:

- [DCL Description](#) on page 983, provides a brief overview of the DCL
- [DCL Packaging](#) on page 983, describes how the DCL is packaged
- [DCL Files](#) on page 983, lists the supported files
- [DCL Functional Description](#) on page 983, describes the three layers of the DCL:
 - Lower layers-[Command Generator](#) on page 983, and [Descriptor Disassembler](#) on page 983
 - Upper layer [Upper-Tier DCL Functions - Descriptor Constructors](#) on page 997

7.4.10.2 DCL Description

The Descriptor Construction Library (DCL) provides a collection of simple functions capable of building SEC4.x descriptors for a wide range of purposes, either as a standalone library or integrated within a driver subsystem. Applications may use all, part, or none of the DCL's functions, or they may use DCL as a simple reference for their own construction functions.

The DCL package will evolve over time. Additional protocol support and descriptor utilities will be added in future software (SW) releases, as new applications for SEC 4.x are developed.

7.4.10.3 DCL Packaging

DCL is packaged with and used only by USDPAA sample applications.

Types in DCL are defined using POSIX conventions for the sake of external portability.

7.4.10.4 DCL Files

The following files are supported

- `cmdgen.c`-Descriptor command generator.
- `disasm.c`-Descriptor disassembler.
- `jobdesc.c`-Job descriptor constructors.
- `protoshared.c`-Shared/protocol descriptor constructors.
- `dcl.h`-Definitions for all published DCL functions.

7.4.10.5 DCL Functional Description

DCL consists of two layers, decomposed into three subsystems:

- A lower tier, comprising the following:
 - Command generator (described in [Command Generator](#) on page 983), colloquially referred to as "cmdgen".
 - Descriptor disassembler (described in [Descriptor Disassembler](#) on page 983)
- An upper tier, composed of descriptor constructors (described in [Upper-Tier DCL Descriptor Constructors](#) on page 984). This is dependent on the command generator in the lower tier.

7.4.10.6 Command Generator

The Command Generator is the lowest level of DCL functionality. Each function within it is capable of generating a single command/instruction in a SEC4.x descriptor and increments a "next in" pointer to the next descriptor word following the generated command.

Applications may use the command generator independently of any other DCL functionality.

In general, creation of any application needing to generate a descriptor on an individual command basis like this starts by building the first commands (or PDB data) past the header, until the body of the descriptor is complete. At this point, the full size of the descriptor is known; the application can then fill in the header using this size.

7.4.10.7 Descriptor Disassembler

The Descriptor Disassembler is meant to be a simple debug tool that can display the content of a constructed descriptor for the user to see in a simple "disassembled" representation. It is intended for developers to use as a visualization aid during development, or as a debug tool, in which descriptor content can be displayed on-the-fly in a human-decipherable form. It does not perform consistency checking, or otherwise identify problem areas in poorly formed descriptors.

The disassembler is a simple C function, and can be packaged in the library with the balance of DCL functions so that it may be linked into a higher-level application.

An example of a disassembled IPsec CBC decapsulation shared descriptor:

```
shrdesc: stidx=8 len=20 share-always
(pdb): [00] 0x00340001 0x00000000 0x00000000 0x00000000
(pdb): [04] 0x00000000 0x00000000 0x00000000
key: len=20 class2->keyreg inline
[00] 0x000e0f00 0x0d0f0a00 0x0d0f0a00 0x0d0f0a00
[04] 0x0d0f0a00
key: len=16 class1->keyreg inline
[00] 0x00e0f0a0 0x00d0f0a0 0x00e0f0a0 0x00d0f0a0
operation: type=decap-pcl ipsec aes-cbc hmac-sha1-96
```

An example of a disassembled IPsec CBC encapsulation shared descriptor:

```
shrdesc: stidx=23 len=35 share-always
(pdb): [00] 0x0000000d 0x00000000 0x00000000 0x00000000
(pdb): [04] 0x00000000 0x00000000 0x00000000 0x00000000
(pdb): [08] 0x00000034 0x34001045 0x00402512 0xd2860640
(pdb): [12] 0x7746430a 0xc046430a 0x160022d0 0x5d891888
(pdb): [16] 0x9cee1912 0xbc211080 0x00000898 0x0a080101
(pdb): [20] 0x22759aa6 0xdb143f08
key: len=20 class2->keyreg inline
[00] 0x000e0f00 0x0d0f0a00 0x0d0f0a00 0x0d0f0a00
[04] 0x0d0f0a00
key: len=16 class1->keyreg inline
[00] 0x00e0f0a0 0x00d0f0a0 0x00e0f0a0 0x00d0f0a0
operation: type=encap-pcl ipsec aes-cbc hmac-sha1-96
```

7.4.10.8 Upper-Tier DCL Descriptor Constructors

A higher level of functionality is provided through complex descriptor constructors. These constructors are single-purpose functions capable of generating complete descriptors from user specifications. These constructors fit into two categories, one for job descriptors and another for shared descriptors generally targeted to protocol processing.

These constructors are by no means the "definitive" reference to all possible descriptor permutations, nor are they meant to work with any specific application. Instead, they are meant to be general-purpose examples of descriptor construction. It is expected that, over time, this library will grow to accommodate a wide range of examples.

All constructor functions are dependent on the underlying command generator.

7.4.10.9 API reference

7.4.10.9.1 API reference command generator

7.4.10.9.1.1 cmd_insert_shared_hdr()

cmd_insert_shared_hdr(): Insert a shared descriptor header into a descriptor

```
u_int32_t *cmd_insert_shared_hdr(u_int32_t *descwd,
                                u_int8_t startidx,
                                u_int8_t desclen,
                                enum ctxsave ctxsave,
                                enum shrst share);
```

Inputs:

- `descwd`-pointer to target descriptor word to hold this command. Note that this should always be the first word of a descriptor.
- `startidx`-index to continuation of descriptor data, normally the first descriptor word past a PDB. This tells DECO what to skip over.
- `descrlen`-length of descriptor in words, including header.
- `ctxsave`-Saved or erases context when a descriptor is self-shared
 - `CTX_SAVE` = context saved between iterations
 - `CTX_ERASE` = context is erased
- `share`-Share state of this descriptor:
 - `SHR_NEVER` = Never share. Fetching is repeated for each processing pass.
 - `SHR_WAIT` = Share once processing starts.
 - `SHR_SERIAL` = Share once completed.
 - `SHR_ALWAYS` = Always share (except keys)

Returns:

Pointer to next incremental descriptor word past the header just constructed. If an error occurred, returns 0.

NOTE

Headers should normally be constructed as the final operation in the descriptor construction, because the start index and overall descriptor length will likely not be known until construction is complete. For this reason, there is little use to the "incremental pointer" convention. The exception is probably in the construction of simple descriptors where the size is easily known early in the construction process.

7.4.10.9.1.2 `cmd_insert_hdr()`

`cmd_insert_hdr()`: Insert a standard descriptor header into a descriptor

```
u_int32_t *cmd_insert_hdr(u_int32_t *descwd,
                          u_int8_t startidx,
                          u_int8_t descrlen,
                          enum shrst share,
                          enum shrnext sharenext,
                          enum execorder reverse,
                          enum mktrust mktrusted);
```

Inputs:

- `descwd`-pointer to target descriptor word to hold this command. Note that this should always be the first word of a descriptor.
- `startidx`-index to continuation of descriptor data, or if `sharenext = SHRNEXT_SHARED`, then specifies the size of the associated shared descriptor referenced in the following instruction.
- `descrlen`-length of descriptor in words, including header.
- `share`-Share state for this descriptor:
 - `SHR_NEVER`-Never share. Fetching is repeated for each processing pass.
 - `SHR_WAIT`-Share once processing starts.
 - `SHR_SERIAL`-Share once completed.

- `SHR_ALWAYS`-Always share (except keys)
- `SHR_DEFER`-Use the referenced sharedesc to determine sharing intent
- `sharenext`-Control state of shared descriptor processing
 - `SHRNXT_SHARED`-This is a job descriptor consisting of a header and a pointer to a shared descriptor only.
 - `SHRNXT_LENGTH`-This is a detailed job descriptor, thus `descLen` refers to the full length of this descriptor.
- `reverse`-Reverse execution order between this job descriptor, and an associated shared descriptor:
 - `ORDER_REVERSE`-execute this descriptor before the shared descriptor referenced.
 - `ORDER_FORWARD`-execute the shared descriptor, then this descriptor.
- `mktrusted-DESC_SIGN`-sign this descriptor prior to execution
 - `DESC_STD` -leave descriptor non-trusted

7.4.10.9.13 `cmd_insert_key()`

`cmd_insert_key()`: Insert a key command into a descriptor

```
u_int32_t *cmd_insert_key(u_int32_t      *descwd,  
                          u_int8_t      *key,  
                          u_int32_t      keylen,  
                          enum ref_type  sgref,  
                          enum key_dest  dest,  
                          enum key_cover cover,  
                          enum item_inline imm,  
                          enum item_purpose purpose);
```

Inputs:

- `descwd`-pointer to target descriptor word to hold this command
- `key`-pointer to key data as an array of bytes.
- `keylen`-pointer to key size, expressed in bits.
- `sgref`-pointer is actual data, or a scatter-gather list representing the key:
 - `PTR_DIRECT`-points to data
 - `PTR_SGLIST`-points to SEC4.x-specific scatter gather table. Cannot use if `imm = ITEM_INLINE`.
- `dest`-target destination in SEC4.x to receive the key. This may be:
 - `KEYDST_KEYREG`-Key register in the CHA selected by an `OPERATION` command.
 - `KEYDST_PK_E`-The 'e' register in the public key block
 - `KEYDST_MD_SPLIT`-Message digest IPAD/OPAD direct load.
- `cover`-Key was encrypted, and must be decrypted during the load. If trusted descriptor, use `TDEK`, else use `JDEK` to decrypt.
 - `KEY_CLEAR`-key is cleartext, no decryption needed
 - `KEY_COVERED`-key is ciphertext, decrypt.
- `imm`-Key can either be referenced, or loaded into the descriptor immediately following the command for improved performance.
 - `ITEM_REFERENCE`-a pointer follows the command.

- `ITEM_INLINE`-key data follows the command, padded out to a descriptor word boundary.
- `purpose`-Sends the key to the class 1 or 2 CHA as selected by an `OPERATION` command. If `dest` is `KEYDST_PK_E`, this must be `ITEM_CLASS1`.

Returns:

If successful, returns a pointer to the target word incremented past the newly-inserted command (including item pointer or inlined data). Effectively, this becomes a pointer to the next word to receive a new command in this descriptor. If error, returns 0

74.10.9.14 `cmd_insert_seq_key()`

`cmd_insert_key()`: Insert a key command into a descriptor using a sequence

```

u_int32_t *cmd_insert_key(u_int32_t      *descwd,
                          u_int32_t      keylen,
                          enum ref_type   sgref,
                          enum key_dest   dest,
                          enum key_cover  cover,
                          enum item_inline imm,
                          enum item_purpose purpose);

```

Inputs:

- `descwd`-pointer to target descriptor word to hold this command
- `keylen`-pointer to key size, expressed in bits.
- `sgref`-pointer is actual data, or a scatter-gather list representing the key:
 - `PTR_DIRECT`-points to data
 - `PTR_SGLIST`-points to SEC4.x-specific scatter gather table. Cannot use if `imm = ITEM_INLINE`.
- `dest`-target destination in SEC4.x to receive the key. This may be:
 - `KEYDST_KEYREG`-Key register in the CHA selected by an `OPERATION` command.
 - `KEYDST_PK_E`-The 'e' register in the public key block
 - `KEYDST_MD_SPLIT`-Message digest IPAD/OPAD direct load.
- `cover`-Key was encrypted, and must be decrypted during the load. If trusted descriptor, use `TDEK`, else use `JDEK` to decrypt.
 - `KEY_CLEAR`-key is cleartext, no decryption needed
 - `KEY_COVERED`-key is ciphertext, decrypt.
- `imm`-Key can either be referenced, or loaded into the descriptor immediately following the command for improved performance.
 - `ITEM_REFERENCE`-a pointer follows the command.
 - `ITEM_INLINE`-key data follows the command, padded out to a descriptor word boundary.
- `purpose`-Sends the key to the class 1 or 2 CHA as selected by an `OPERATION` command. If `dest` is `KEYDST_PK_E`, this must be `ITEM_CLASS1`.

Returns:

If successful, returns a pointer to the target word incremented past the newly-inserted command (including item pointer or inlined data). Effectively, this becomes a pointer to the next word to receive a new command in this descriptor. If error, returns 0

7.4.10.9.15 cmd_insert_proto_op_ipsec()

cmd_insert_proto_op_ipsec()-Insert an IPSec protocol operation command into a descriptor.

```
u_int32_t *cmd_insert_proto_op_ipsec(u_int32_t      *descwd,  
                                     u_int8_t      cipheralg,  
                                     u_int8_t      authalg,  
                                     enum protdir   dir);
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction. For an OPERATION instruction, this is normally the final word of a single descriptor.
- cipheralg-blockcipher selection for this protocol descriptor. This should be one of CIPHER_TYPE_IPSEC_.
- authalg-authentication selection for this protocol descriptor. This should be one of AUTH_TYPE_IPSEC_.
- dir-Select DIR_ENCAP for encapsulation, or DIR_DECAP for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

7.4.10.9.16 cmd_insert_proto_op_wimax()

cmd_insert_proto_op_wimax()-Insert an 802.16 WiMAX protocol OPERATION instruction into a descriptor. These can only operate as AES-CCM.

```
u_int32_t *cmd_insert_proto_op_wimax(u_int32_t      *descwd,  
                                     u_int8_t      mode,  
                                     enum protdir   dir);
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction. For an OPERATION instruction within the scope of a protocol descriptor, this is normally the final word of a single descriptor.
- mode-a nonzero value selects OFDMA, else assume OFDM operation.
- dir-Select DIR_ENCAP for encapsulation, or DIR_DECAP for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

7.4.10.9.17 cmd_insert_proto_op_wifi()

cmd_insert_proto_op_wifi()-Insert an 802.11 WiFi protocol OPERATION command into a descriptor.

```
u_int32_t *cmd_insert_proto_op_wifi(u_int32_t      *descwd,  
                                     enum protdir   dir);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction. For an `OPERATION` instruction within the scope of a protocol descriptor, this is normally the final word of a single descriptor.
- `dir`-Select `DIR_ENCAP` for encapsulation, or `DIR_DECAP` for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

7.4.10.9.1.8 `cmd_insert_proto_op_macsec()`

`cmd_insert_proto_op_macsec()` -Insert an MacSec protocol `OPERATION` instruction into a descriptor.

```
u_int32_t *cmd_insert_proto_op_macsec(u_int32_t *descwd,
                                     enum protdir dir);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction. For an `OPERATION` instruction within the scope of a protocol descriptor, this is normally the final word of a single descriptor.
- `dir`-Select `DIR_ENCAP` for encapsulation, or `DIR_DECAP` for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

7.4.10.9.1.9 `cmd_insert_proto_op_unidir()`

`cmd_insert_proto_op_unidir()` -Insert a unidirectional protocol `OPERATION` instruction into a descriptor.

```
u_int32_t *cmd_insert_proto_op_unidir(u_int32_t *descwd, u_int32_t protid,
                                     u_int32_t protinfo);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction. For an `OPERATION` instruction within the scope of a protocol descriptor, this is normally the final word of a single descriptor.
- `protid` -Select any `PROTID` field needed for a unidirectional protocol descriptor from `OP_PCLID_`.
- `dir`-Select `DIR_ENCAP` for encapsulation, or `DIR_DECAP` for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

7.4.10.9.1.10 `cmd_insert_alg_op()`

`cmd_insert_alg_op()` -Insert a simple algorithm `OPERATION` instruction into a descriptor.

```
u_int32_t *cmd_insert_alg_op(u_int32_t *descwd, u_int32_t optype,
                             u_int32_t algtype, u_int32_t algmode,
                             enum mdstatesel mdstate, enum icvsel icv,
                             enum algdir dir);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `optype`-specify as class 1 or 2 using `OP_TYPE_CLASSx_ALG`.

- `algtype`-cipher selection, specify one of `ALG_TYPE_`.
- `algmode`-cipher mode selection, specify one of `ALG_MODE_`. Some combinations are ORable depending on application.
- `mdstate`-if a message digest is being processed, selects the processing state. May be one of `MDSTATE_UPDATE`, `MDSTATE_INIT`, `MDSTATE_FINAL`, or `MDSTATE_COMPLETE`.
- `icv`-if processing a message digest, or a cipher with an including authentication function, then `ICV_CHECK_ON` selects an inline signature comparison on the computed result.
- `protid`-Select any `PROTID` field needed for a unidirectional protocol descriptor from `OP_PCLID_`.
- `dir`-Select `DIR_ENCAP` for encapsulation, or `DIR_DECAP` for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

7.4.10.9.11 `cmd_insert_pkha_op()`

`cmd_insert_pkha_op()`-Insert a PKHA-algorithm OPERATION instruction into a descriptor.

```
u_int32_t *cmd_insert_pkha_op(u_int32_t *descwd, u_int32_t pkmode);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `pkmode`-mode selection bits, an OR of `OP_ALG_PKMODE_` from one of the 3 possible PKHA sets (clear memory, modular arithmetic, copy memory).

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

7.4.10.9.12 `cmd_insert_seq_in_ptr()`

`cmd_insert_seq_in_ptr()`: Insert an SEQ IN PTR command into a descriptor

```
int *cmd_insert_seq_in_ptr(u_int32_t *descwd,  
                          u_int32_t *ptr,  
                          u_int32_t len,  
                          enum ref_type sgregf);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this command. For an OPERATION command, this is normally the final word of a single descriptor.
- `ptr`-bus address pointing to the input data buffer
- `len`-input length
- `sgregf`-pointer is actual data, or a scatter-gather list representing the key:
 - `PTR_DIRECT`-points to data
 - `PTR_SGLIST`-points to SEC4.x-specific scatter gather table.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

74.10.9.113 cmd_insert_seq_out_ptr()

cmd_insert_seq_out_ptr(): Insert an SEQ OUT PTR command into a descriptor

```
int *cmd_insert_seq_out_ptr(u_int32_t *descwd,
                           u_int32_t *ptr,
                           u_int32_t len,
                           enum ref_type sgreg);
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this command. For an OPERATION command, this is normally the final word of a single descriptor.
- ptr-bus address pointing to the output data buffer
- len-output length
- sgreg-pointer is actual data, or a scatter-gather list representing the key:
 - PTR_DIRECT-points to data
 - PTR_SGLIST-points to SEC4.x-specific scatter gather table.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

74.10.9.114 cmd_insert_load()

cmd_insert_load(): Insert a LOAD instruction into a descriptor:

```
u_int32_t *cmd_insert_load(u_int32_t *descwd, void *data,
                           u_int32_t class_access, u_int32_t sgflag,
                           u_int32_t dest, u_int8_t offset,
                           u_int8_t len, enum item_inline imm)
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction.
- data-pointer to data to be loaded.
- class_access
 - LDST_CLASS_IND_CCB-access class-independent objects in CCB
 - LDST_CLASS_1_CCB -access class 1 objects in CCB
 - LDST_CLASS_2_CCB -access class 2 objects in CCB
 - LDST_CLASS_DECO -access DECO objects
- sgflag-specify LDST_SGF if data reference points to a scatter/gather list representing the data.
- dest - internal destination for the LOAD. Should be one of LDST_SRCDEST_.
- offset - starting point for writing in the destination.
- len - length of data in bytes.
- imm - if specified as ITEM_INLINE, data is inlined into the descriptor immediately following the LOAD instruction.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

7.4.10.9.1.15 cmd_insert_seq_load()

cmd_insert_seq_load(): Insert a SEQ LOAD instruction into a descriptor:

```
int *cmd_insert_seq_load(u_int32_t *descwd,  
    unsigned int class_access,  
    int variable_len_flag,  
    unsigned char dest,  
    unsigned char offset,  
    unsigned char len);
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction.
- class_access
 - LDST_CLASS_IND_CCB-access class-independent objects in CCB
 - LDST_CLASS_1_CCB -access class 1 objects in CCB
 - LDST_CLASS_2_CCB -access class 2 objects in CCB
 - LDST_CLASS_DECO -access DECO objects
- variable_len_flag-use the variable input sequence length
 - dest-destination
 - offset-the start point for writing in the destination
 - len-length of data in bytes

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

7.4.10.9.1.16 cmd_insert_fifo_load()

cmd_insert_fifo_load(): Insert a FIFO LOAD instruction into a descriptor

```
u_int32_t *cmd_insert_fifo_load(u_int32_t *descwd, void *data, u_int32_t len,  
    u_int32_t class_access, u_int32_t sgflag,  
    u_int32_t imm, u_int32_t ext, u_int32_t type)
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction.
- data - pointer to data to be loaded.
- len - length of load data in bytes.
- class_access
 - LDST_CLASS_IND_CCB-access class-independent objects in CCB
 - LDST_CLASS_1_CCB-access class 1 objects in CCB
 - LDST_CLASS_2_CCB-access class 2 objects in CCB
 - LDST_CLASS_DECO-access DECO objects

- `sgflag`-data points to a scatter/gather list representing the data to be loaded.
- `imm` - specify `FIFOLDST_IMM` if `fdata` is to be included immediately following this instruction.
- `ext` - if length needs to be >16 bits, specify `FIFOLDST_EXT` to include the extended length in a word following the instruction.
- `type`-FIFO input data type specified as `FIFOLD_TYPE_*`

Returns:

Pointer to next incremental descriptor word past the instruction just constructed. If an error occurred, returns 0.

7.4.10.9.17 `cmd_insert_seq_fifo_load()`

`cmd_insert_seq_fifo_load()`: Insert a SEQ FIFO LOAD instruction into a descriptor

```
u_int32_t *cmd_insert_seq_fifo_load(u_int32_t *descwd, u_int32_t class_access,
                                   u_int32_t variable_len_flag,
                                   u_int32_t data_type, u_int32_t len)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `class_access`
 - `LDST_CLASS_IND_CCB`-access class-independent objects in CCB
 - `LDST_CLASS_1_CCB`-access class 1 objects in CCB
 - `LDST_CLASS_2_CCB`-access class 2 objects in CCB
 - `LDST_CLASS_DECO`-access DECO objects
- `variable_len_flag`-use the variable input sequence length
- `data_type`-FIFO input data type (`FIFOLD_TYPE_*` in `desc.h`)
- `len`-input data length

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

7.4.10.9.18 `cmd_insert_store()`

`cmd_insert_store()`: Insert a STORE instruction into a descriptor

```
u_int32_t *cmd_insert_store(u_int32_t *descwd, void *data,
                            u_int32_t class_access, u_int32_t sg_flag,
                            u_int32_t src, u_int8_t offset,
                            u_int8_t len, enum item_inline imm)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `data` - pointer to the data store location.
- `class_access`
 - `LDST_CLASS_IND_CCB`-access class-independent objects in CCB
 - `LDST_CLASS_1_CCB`-access class 1 objects in CCB
 - `LDST_CLASS_2_CCB`-access class 2 objects in CCB

- LDST_CLASS_DECO-access DECO objects
- sgflag-if LDST_SGF, the data pointer references a scatter/gather list describing the buffer to receive the stored data.
- src - data source specification, one of LDST_SRCDEST_
- offset-offset into source to begin store operation.
- len-store length in bytes.
- imm - if LDST_IMM, then the data to be stored follows the instruction in the descriptor.

Returns:

1. Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

7.4.10.9.1.19 cmd_insert_seq_store()

cmd_insert_seq_store(): Insert a SEQ STORE instruction into a descriptor

```
u_int32_t *cmd_insert_seq_store(u_int32_t *descwd, u_int32_t class_access,  
                               u_int32_t variable_len_flag, u_int32_t src,  
                               u_int8_t offset, u_int8_t len);
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction.
- class_access
 - LDST_CLASS_IND_CCB-access class-independent objects in CCB
 - LDST_CLASS_1_CCB-access class 1 objects in CCB
 - LDST_CLASS_2_CCB-access class 2 objects in CCB
 - LDST_CLASS_DECO-access DECO objects
- variable_len_flag-if LDST_VLF, uses the variable sequence output length.
- src - data source specification, one of LDST_SRCDEST_
- offset-offset into source to begin store operation.
- len-store length in bytes.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

7.4.10.9.1.20 cmd_insert_fifo_store()

cmd_insert_fifo_store(): Insert a FIFO STORE instruction into a descriptor

```
u_int32_t *cmd_insert_fifo_store(u_int32_t *descwd, void *data, u_int32_t len,  
                                u_int32_t class_access, u_int32_t sgflag,  
                                u_int32_t imm, u_int32_t ext, u_int32_t type)
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction.
- data - pointer to data to be stored from FIFO.
- len - length of data to store.
- class_access
 - LDST_CLASS_IND_CCB-access class-independent objects in CCB

- LDST_CLASS_1_CCB-access class 1 objects in CCB
- LDST_CLASS_2_CCB-access class 2 objects in CCB
- LDST_CLASS_DECO-access DECO objects
- sgflag-if FIFOLDST_SGF, data points to a scatter/gather list describing the buffer to be used for the store.
- imm - if FIFOLDST_IMM, store data is to be inlined into the descriptor itself, immediately following the generated instruction.
- ext-if FIFOLDST_EXT, length exceeds 16 bits, and therefore cannot be included in the instruction itself. Write the extended length out to a word following the instruction.
- type-FIFO input type, an OR combination of FIFOST_TYPE_type and last/flush bits for class1 and 2.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

7.4.10.9.121 cmd_insert_seq_fifo_store()

cmd_insert_seq_fifo_store(): Insert a SEQ FIFO STORE instruction into a descriptor

```
u_int32_t *cmd_insert_seq_fifo_store(u_int32_t *descwd, u_int32_t class_access,
                                     u_int32_t variable_len_flag,
                                     u_int32_t out_type, u_int32_t len)
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction.
- len - length of data to store.
- class_access
 - LDST_CLASS_IND_CCB-access class-independent objects in CCB
 - LDST_CLASS_1_CCB-access class 1 objects in CCB
 - LDST_CLASS_2_CCB-access class 2 objects in CCB
 - LDST_CLASS_DECO-access DECO objects
- sgflag-if FIFOLDST_SGF, data points to a scatter/gather list describing the buffer to be used for the store.
- imm - if FIFOLDST_IMM, store data is to be inlined into the descriptor itself, immediately following the generated instruction.
- ext-if FIFOLDST_EXT, length exceeds 16 bits, and therefore cannot be included in the instruction itself. Write the extended length out to a word following the instruction.
- type-FIFO input type, an OR combination of FIFOST_TYPE_type and last/flush bits for class1 and 2.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

7.4.10.9.122 cmd_insert_jump()

cmd_insert_jump(): Insert a JUMP instruction into a descriptor

```
u_int32_t *cmd_insert_jump(u_int32_t *descwd, u_int32_t jtype,
                           u_int32_t class, u_int32_t test, u_int32_t cond,
                           int8_t offset, u_int32_t *jmpdesc)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `jtype` - type of jump to perform, one of `JUMP_TYPE_`
- `class`
 - `CLASS_NONE` -jump is not a checkpoint
 - `CLASS_1` -jump is a checkpointing for a class 1 operation
 - `CLASS_2` -jump is a checkpoint for a class 2 operation
 - `CLASS_BOTH` -jump is a checkpoint for both classes
- `test` -selects how to assess the conditional test, one of `JUMP_TEST_`
- `cond` - OR combination of conditions to test, based on the test type selected in `test`. May be a combination of `JUMP_COND_`. Note that the JSL bit is factored into the definitions for `JUMP_COND_`, and therefore there are two possible combinational sets.
- `offset` -relative offset of descriptor words to jump to if `JUMP_TYPE_LOCAL` is selected. May be a positive or negative offset.
- `jmpdesc` -address of descriptor to jump to is `JUMP_TYPE_NONLOCAL` is selected.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

74.10.9.123 `cmd_insert_math()`

`cmd_insert_math()`: Insert a MATH instruction into a descriptor

```
u_int32_t *cmd_insert_math(u_int32_t *descwd, u_int32_t func,
                          u_int32_t src0, u_int32_t src1,
                          u_int32_t dest, u_int32_t len,
                          u_int32_t flagupd, u_int32_t stall,
                          u_int32_t immediate, u_int32_t *data)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `func` - math function to perform, one of `MATH_FUN_`.
- `src0` -first source operand, one of `MATH_SRC0_`.
- `src1` - second source operand, one of `MATH_SRC1_`. Note differences between what can be selected between `SRC0` and `SRC1`.
- `dest` - destination operand for the result, one of `MATH_DEST_`.
- `flagupd` - specify `MATH_NFU` if the flags should not be updated.
- `stall` -specify `MATH_STL` to cause the instruction to consume an extra clock cycle.
- `immediate` -specify `MATH_IFB` to use 4 bytes of immediate data when the length needs to remain as 8.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

7.4.10.9.1.24 cmd_insert_move()

cmd_insert_move(): Insert a MOVE instruction into a descriptor

```

u_int32_t *cmd_insert_move(u_int32_t *descwd, u_int32_t waitcomp,
                          u_int32_t src, u_int32_t dst, u_int8_t offset,
                          u_int8_t length)

```

Inputs:

- descwd - pointer to target descriptor word intended to hold this instruction.
- waitcomp - specify MOVE_WAITCOMP if intending to stall execution until the move completes.
- src - data source, one of MOVE_SRC_.
- dest - destination, one of MOVE_DEST_.
- offset - offset into source for move.
- length - length of data to move.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

7.4.10.9.2 Upper-Tier DCL Functions - Descriptor Constructors

A higher level of functionality is provided through complex descriptor constructors. These constructors are single-purpose functions capable of generating complete descriptors from user specifications. These constructors fit into two categories, one for job descriptors and another for shared descriptors generally targeted to protocol processing.

These constructors are by no means the “definitive” reference to all possible descriptor permutations, nor are they meant to work with any specific application. Instead, they are meant to be general-purpose examples of descriptor construction. It is expected that, over time, this library will grow to accommodate a wide range of examples.

All constructor functions are dependent on the underlying command generator.

7.4.10.9.2.1 Job descriptor constructors

7.4.10.9.2.1.1 cnstr_seq_jobdesc()

cnstr_seq_jobdesc(): Construct simple sequence job descriptor

```

int cnstr_seq_jobdesc(u_int32_t *jobdesc, unsigned short *jobdescsz,
                    u_int32_t *shrdesc, unsigned short shrdescsize,
                    unsigned char *inbuf, unsigned long insize,
                    unsigned char *outbuf, unsigned long outsize);

```

Inputs:

- jobdesc - pointer to buffer in which to build descriptor in
- jobdescsz - max size of descriptor build buffer
- shrdesc - pointer to associated shared descriptor holding session context
- shrdescsz - size of associated shared descriptor
- inbuf - pointer to input frame

- `in_size`-size of input frame
- `outbuf`-pointer to output frame
- `out_size`-size of output frame

Constructs a simple job descriptor, emulating QI-level frame processing behavior useful at the job queue level. Besides a target descriptor output, this constructor depends on three references.

1. A pointer to a shared descriptor to do the work. This is normally assumed to be some sort of a protocol-level shared descriptor.
2. A pointer to a packet/frame for input data
3. A pointer to a packet/frame for output data

The constructed descriptor is a simple reverse-order-execution descriptor, and has no provisions for other content specifications.

7.4.10.9.2.1.2 `cnstr_jobdesc_blkcipher_cbc()`

Construct a job descriptor capable of performing a CBC blockcipher operation:

```
int cnstr_jobdesc_blkcipher_cbc(u_int32_t *descbuf, u_int16_t *bufsz,
                               u_int8_t *data_in, u_int8_t *data_out,
                               u_int32_t datasz,
                               u_int8_t *key, u_int32_t keylen,
                               u_int8_t *iv, u_int32_t ivlen,
                               enum algdir dir, u_int32_t cipher,
                               u_int8_t clear);
```

Inputs:

- `descbuf` - Pointer to DMA-able buffer for descriptor construction.
- `bufsz` - Size of constructed descriptor (as output)
- `data_in` - Pointer to input message
- `data_out` - Pointer to output message
- `datasz` - Size of input/output messages
- `key` - Pointer to cipher key
- `keylen` - Size of cipher key
- `iv` - Pointer to cipher IV
- `ivlen` - Size of cipher IV
- `dir` - Direction of cipher operation, select `DIR_ENCRYPT` or `DIR_DECRYPT`
- `cipher` - Blockcipher algorithmselection chosen from `OP_ALG_ALGSEL_`.
- `clear` - Clear descriptor buffer before construction

Returns: -1 on construction error, 0 if construction succeeded.

7.4.10.9.2.1.3 `cnstr_jobdesc_hmac()`

Construct a job descriptor capable of performing an HMAC operation:

```
int32_t cnstr_jobdesc_hmac(u_int32_t *descbuf, u_int16_t *bufsize,
                          u_int8_t *msg, u_int32_t msgsz, u_int8_t *digest,
                          u_int8_t *key, u_int32_t cipher, u_int8_t *icv,
                          u_int8_t clear);
```

Inputs:

- `descbuf` - descriptor buffer
- `bufsize` - limit/returned descriptor buffer size
- `msg` - pointer to message being processed
- `msgsz` - size of message in bytes
- `digest` - output buffer for digest (size derived from cipher)
- `key` - key data (size derived from cipher)
- `cipher` - OP_ALG_ALGSEL_MD5/SHA1-512
- `icv` - HMAC comparison for ICV, NULL if no check desired
- `clear` - clear buffer before writing

Returns: -1 on construction error, 0 if construction succeeded.

7.4.10.9.2.1.4 `cnstr_jobdesc_mdsplitkey()`

Generate an MDHA "split key" from HMAC key content. A split key is a precomputed IPAD/OPAD pair; MDHA can save cycles during sequential packet processing on a flow by using the precomputed pair directly, and thus saving the pad generation step for each packet.

Generally, the split key is generated at flow setup time as a control-plane activity, thus, this step is performed as a job descriptor.

```
int cnstr_jobdesc_mdsplitkey(u_int32_t *descbuf, u_int16_t *bufsize,
                             u_int8_t *key, u_int32_t cipher,
                             u_int8_t *padbuf);
```

Inputs:

- `descbuf` - pointer to buffer to hold constructed descriptor
- `bufsize` - pointer to size of descriptor once constructed
- `key` - pointer to HMAC key to generate pad pair from. Key size is determined by cipher selection. Note that SHA224/384 pairs are not truncated to the digest size:

Table 161.

	Key size	Split key size	Buffer size
OP_ALG_ALGSEL_MD5	16	32	32
OP_ALG_ALGSEL_SHA1	20	40	48
OP_ALG_ALGSEL_SHA224	28	64	64
OP_ALG_ALGSEL_SHA256	32	64	64
OP_ALG_ALGSEL_SHA384	48	128	128
OP_ALG_ALGSEL_SHA512	64	128	128

- `cipher` - HMAC algorithm selection, one of OP_ALG_ALGSEL_
- `padbuf` - buffer to store generated ipad/opad. Should be 2x the untruncated HMAC keysize for the chosen cipher rounded up to the nearest 16-byte boundary (where 16 bytes = an AES blocksize). See table under "key" above.

Returns: -1 on construction error, 0 if construction succeeded.

7.4.10.9.2.1.5 *cnstr_jobdesc_aes_gcm()*

Construct a job descriptor capable of performing an AES-GCM operation:

```
int cnstr_jobdesc_aes_gcm(u_int32_t *descbuf, u_int16_t *bufsize,  
                        u_int8_t *key, u_int32_t keylen, u_int8_t *ctx,  
                        enum mdstatesel mdstate, enum icvsel icv, enum algdir dir,  
                        u_int8_t *in, u_int8_t *out, u_int16_t size, u_int8_t *mac);
```

Inputs:

- *descbuf* - pointer to buffer that will hold constructed descriptor
- *bufsiz* - pointer to size of descriptor once constructed
- *key* - pointer to AES key
- *keylen* - AES key length
- *ctx* - points to GCM context block. This is a concatenation of: MAC (128 bits), Yi (128 bits), Y0 (128 bits), IV (64 bits), and text bitsize (64 bits). See the AESA section of the blockguide for more information.
- *mdstate* - select `MDSTATE_UPDATE`, `MDSTATE_INIT`, or `MDSTATE_FINAL` if a partial MAC operation is desired, else select `MDSTATE_COMPLETE`.
- *icv* - select `ICV_CHECK_ON` if a MAC compare is requested.
- *dir* - select `DIR_ENCRYPT` or `DIR_DECRYPT` as needed for cipher operation
- *in* - Pointer to input text buffer
- *out* - Pointer to output data text
- *size* - Size of data to be processed
- *mac* - Pointer to output MAC. This can point to the head of context if an updated MAC is required for subsequent operations.

Returns: -1 on construction error, 0 if construction succeeded.

7.4.10.9.2.1.6 *cnstr_jobdesc_kasumi_f8()*

Construct a job descriptor capable of performing a Kasumi f8 (confidentiality) operation:

```
int cnstr_jobdesc_kasumi_f8(u_int32_t *descbuf, u_int16_t *bufsz,  
                          u_int8_t *key, u_int32_t keylen,  
                          enum algdir dir, u_int32_t *ctx,  
                          u_int8_t *in, u_int8_t *out, u_int16_t size);
```

Inputs:

- *descbuf* - pointer to buffer that will hold constructed descriptor
- *bufsiz* - pointer to size of descriptor once constructed
- *key* - pointer to KFHA cipher key
- *keylen* - cipher key length
- *dir* - select `DIR_ENCRYPT` or `DIR_DECRYPT` as needed
- *ctx* - points to preformatted f8 context block, containing the 32-bit count (word 0), bearer (word 1 bits 7:16), and cb (word 1 bits 17:31). Refer to the KFHA section of the block guide for more detail.
- *in* - Pointer to input data text

- `out` - Pointer to output data text
- `size` - Size of the data to be processed

Returns: -1 on construction error, 0 if construction succeeded.

7.4.10.9.2.1.7 *cnstr_jobdesc_kasumi_f9()*

Construct a job descriptor capable of performing a Kasumi f9 (authentication) operation:

```
int cnstr_jobdesc_kasumi_f9(u_int32_t *descbuf, u_int16_t *bufsz,
                           u_int8_t *key, u_int32_t keylen,
                           enum algdir dir, u_int32_t *ctx,
                           u_int8_t *in, u_int16_t size, u_int_t *mac);
```

Inputs:

- `descbuf` - pointer to buffer that will hold constructed descriptor
- `bufsiz` - pointer to size of descriptor once constructed
- `key` - pointer to cipher key
- `keylen` - size of cipher key
- `dir` - select `DIR_ENCRYPT` or `DIR_DECRYPT` as required
- `ctx` - points to preformatted f8 context block, containing 32-bit count (word 0), bearer (word 1 bits 0:5), direction (word 1 bit 6), ca (word 1 bits 7:16), cb (word 1 bits 17:31), fresh (word 2), and the ICV input (word 3). Refer to the KFHA section of the block guide for more detail
- `out` - pointer to input data
- `out_siz` - size of input data
- `mac` - pointer to output MAC

Returns: -1 on construction error, 0 if construction succeeded.

7.4.10.9.2.1.8 *cnstr_jobdesc_pkha_rsaexp()*

Construct a job descriptor capable of performing an RSA exponentiation operation:

```
int cnstr_jobdesc_pkha_rsaexp(u_int32_t *descbuf, u_int16_t *bufsz,
                              struct pk_in_params *pkin,
                              u_int8_t *out, u_int32_t out_siz,
                              u_int8_t clear);
```

Inputs:

- `descbuf` - pointer to buffer to hold descriptor
- `bufsiz` - pointer to size of written descriptor
- `pkin` - Values of A, B, E, and N
- `out` - Encrypted output
- `out_siz` - size of buffer for encrypted output
- `clear` - nonzero if descriptor buffer space is to be cleared before construction

Returns: -1 on construction error, 0 if construction succeeded.

7.4.10.9.2.1 *cnstr_jobdesc_dsaverify()*

Construct a job descriptor capable of performing DSA signature verification:

```
int cnstr_jobdesc_dsaverify(u_int32_t *descbuf, u_int16_t *bufsz,
                           struct dsa_pdb *dsadata, u_int8_t *msg,
                           u_int32_t msg_sz, u_int8_t clear);
```

Inputs:

- *descbuf* - pointer to descriptor buffer for construction
- *bufsz* - pointer to size of descriptor constructed (output)
- *dsadata* - pointer to DSA parameters
- *msg* - pointer to input message for verification
- *msg_sz* - size of message to verify
- *clear* - clear buffer before writing descriptor

Returns: -1 on construction error, 0 if construction succeeded.

7.4.10.9.2.2 Protocol/shared descriptor constructors

These constructors build a full protocol-level shared descriptor used for semi-autonomous processing of secured traffic through SEC4.x. Such descriptors function as single-pass processors (integrating cipher and authentication functions into a single logical step) with the added factor of performing protocol-level packet manipulation in the same step in the packet-handling process, by maintaining protocol-level connection state information within the descriptor itself.

7.4.10.9.2.2.1 *cnstr_pcl_shdsc_ipsec_cbc_decap()*

Note: this function is deprecated in 2.0, and will be removed in a future release. Use *cnstr_shdsc_ipsec_decap()* instead.

cnstr_pcl_shdsc_ipsec_cbc_decap(): Shared protocol-level descriptor for IPSec CBC decapsulation. This function can create a descriptor capable of either tunnel or transport mode processing.

```
int32_t cnstr_pcl_shdsc_ipsec_cbc_decap(u_int32_t *descbuf,
                                       u_int16_t *bufsize,
                                       struct pdbcont *pdb,
                                       struct cipherparams *cipherdata,
                                       struct authparams *authdata,
                                       u_int8_t clear);
```

Inputs:

- *descbuf*-Points to a buffer to construct the descriptor in. All SEC4.x descriptors are built of an array of up to sixty-three 32-bit words. If the caller wishes to construct a descriptor directly in the executable buffer, then that buffer must be hardware DMA-able, and physically contiguous.
- *bufsize*-Points to an unsigned 16-bit word with the maximum length of the buffer to hold the descriptor. This will be written back to with the actual size of the descriptor once constructed. (Note: bounds checking not yet implemented).
- *pdb*-Points to a block of data (struct *pdbcont*) used to describe the content of the Protocol Data Block to be maintained inside the descriptor. PDB content is protocol and mode specific:
 - *pdb.opthdrlen* = Size of inbound header to skip over.
 - *pdb.transmode* = *PDB_TUNNEL*/*PDB_TRANSPORT* for tunnel or transport handling for the next header.
 - *pdb.pclvers* = *PDB_IPV4*/*PDB_IPV6* as appropriate for this connection.
 - *pdb.seq.esn* = *PDB_NO_ESN* unless extended sequence numbers are to be supported, then *PDB_INCLUDE_ESN*.

- `pdb.seq/antirplysz = PDB_ANTIRPLY_NONE` if no antireplay window is to be maintained in the PDB. Otherwise may be `PDB_ANTIRPLY_32` for a 32-entry window, or `PDB_ANTIRPLY_64` for a 64-entry window.
- `cipherdata`-Points to a block of data used to describe the cipher information for encryption/decryption of packet content:
 - `algtype-one` of `CIPHER_TYPE_IPSEC_XXX`
 - `key`-pointer to the cipher key data
 - `keydata`-size of the key data in bits
- `authdata`-Points to a block of data used to describe the authentication information for validating the authenticity of the packet source.
 - `algtype-one` of `AUTH_TYPE_IPSEC_XXX`
 - `key`-pointer to the HMAC key data
 - `keydata`-size of the key data in bits
- `clear`-If nonzero, buffer is cleared before writing

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

7.4.10.9.2.2.2 `cnstr_pcl_shdsc_ipsec_cbc_encap()`

Note: this function is deprecated in 2.0, and will be removed in a future release. Use `cnstr_shdsc_ipsec_encap()` instead.

`cnstr_pcl_shdsc_ipsec_cbc_encap()`: Shared protocol-level descriptor for IPSec CBC encapsulation. This function can construct a descriptor for either transport or tunnel mode operation.

```
int32_t cnstr_pcl_shdsc_ipsec_cbc_encap(u_int32_t *descbuf,
                                       u_int16_t      *bufsize,
                                       struct pdbcont  *pdb,
                                       struct cipherparams *cipherdata,
                                       struct authparams *authdata,
                                       u_int8_t clear);
```

Inputs:

- `descbuf`-Points to a buffer to construct the descriptor in. All SEC4.x descriptors are built of an array of up to sixty-three 32-bit words. If the caller wishes to construct a descriptor directly in the executable buffer, then that buffer must be hardware DMA-able, and physically contiguous.
- `bufsize`-Points to an unsigned 16-bit word with the maximum length of the buffer to hold the descriptor. This will be written back to with the actual size of the descriptor once constructed. (Note: bounds checking not yet implemented).
- `pdb`-Points to a block of data (struct `pdbcont`) used to describe the content of the Protocol Data Block to be maintained inside the descriptor. PDB content is protocol and mode specific:
 - `pdbinfo.opthdrln` = Size of outbound IP header to be prepended to output.
 - `pdbinfo.opthdr` = Pointer to the IP header to be prepended to the output, of size `opthdrln`.
 - `pdbinfo.transmode` = `PDB_TUNNEL/PDB_TRANSPORT` for tunnel/transport handling for the next header.
 - `pdbinfo.pclvers` = `PDB_IPV4/PDB_IPV6` as appropriate for this connection.
 - `pdbinfo.seq.esn` = `PDB_NO_ESN` unless extended sequence numbers are to be supported, then `PDB_INCLUDE_ESN`.

- `pdbinfo.ivsrc = PDB_IV_FROM_PDB` if the IV is to be maintained in the PDB, else `PDB_IV_FROM_RNG` if the IV is to be generated internally by SEC4.x's random number generator.
- `cipherdata`-Points to a block of data used to describe the cipher information for encryption/decryption of packet content:
 - `algtype-one` of `CIPHER_TYPE_IPSEC_XXX`
 - `key`-pointer to the cipher key data
 - `keydata-size` of the key data in bits
- `authdata`-Points to a block of data used to describe the authentication information for validating the authenticity of the packet source.
 - `algtype-one` of `AUTH_TYPE_IPSEC_XXX`
 - `key`-pointer to the HMAC key data
 - `keydata-size` of the key data in bits
- `clear`-If nonzero, buffer is cleared before writing

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

7.4.10.9.2.2.3 `cnstr_shdsc_ipsec_encap()`

Construct a shared protocol-level descriptor capable of performing IPSec ESP packet encapsulation:

```
int32_t cnstr_shdsc_ipsec_encap(u_int32_t *descbuf, u_int16_t *bufsize,
                               struct ipsec_encap_pdb *pdb, u_int8_t *opthdr,
                               struct cipherparams *cipherdata,
                               struct authparams *authdata);
```

Inputs:

- `descbuf` - Pointer to buffer used for descriptor construction
- `bufsize` - Pointer to size to be written back upon completion
- `pdb` - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the encapsulation PDB, and its cipher-dependent unioned substructure.
- `opthdr` - Pointer to the optional header meant to be prepended to an encapsulated frame. Size of the optional header is defined in `pdb.opt_hdr_len`.
- `cipherdata` - Pointer to blockcipher transform definitions
- `authdata` - Pointer to authentication transform definitions. Note that an MDHA split key is to be used with this descriptor (potentially constructed using `cnstr_jobdesc_mdsplitkey()`), and so the size of the uncovered split key is to be specified here, not the size of the encrypted split key buffer. See the description of `cnstr_jobdesc_mdsplitkey()` for a detailed discussion of split key lengths versus buffer sizes

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

7.4.10.9.2.2.4 `cnstr_shdsc_ipsec_decap()`

Construct a shared protocol-level descriptor capable of performing IPSec ESP packet decapsulation:

```
int32_t cnstr_shdsc_ipsec_decap(u_int32_t *descbuf, u_int16_t *bufsize,
                               struct ipsec_encap_pdb *pdb,
```

```
struct cipherparams *cipherdata,  

struct authparams *authdata);
```

Inputs:

- `descbuf`
 - Pointer to buffer used for descriptor construction
- `bufsize`
 - Pointer to size to be written back upon completion
- `pdb`
 - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the decapsulation PDB, and it's cipher-dependent unioned substructure.
- `cipherdata`
 - Pointer to blockcipher transform definitions
- `authdata`
 - Pointer to authentication transform definitions. Note that an MDHA split key is to be used with this descriptor (potentially constructed using `cnstr_jobdesc_mdsplitkey()`), and so the size of the uncovered split key is to be specified here, not the size of the encrypted split key buffer. See the description of `cnstr_jobdesc_mdsplitkey()` for a detailed discussion of split key lengths versus buffer sizes

Returns:

1. -1 if the descriptor creation failed for any reason, zero if creation succeeded.

7.4.10.9.2.2.5 `cnstr_shdsc_wifi_encap()`

Construct a shared protocol-level descriptor capable of performing IEEE 802.11i WiFi packet encapsulation:

```
int32_t cnstr_shdsc_wifi_encap(u_int32_t *descbuf, u_int16_t *bufsize,  

                             struct wifi_encap_pdb *pdb,  

                             struct cipherparams *cipherdata);
```

Inputs:

- `descbuf`
 - Pointer to buffer used for descriptor construction
- `bufsize`
 - Pointer to size to be written back upon completion

- `pdb`
 - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the encapsulation PDB.
- `cipherdata`
 - Pointer to blockcipher transform definitions

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

7.4.10.9.2.2.6 *cnstr_shdsc_wifi_decap()*

Construct a shared protocol-level descriptor capable of performing IEEE 802.11i WiFi packet decapsulation:

```
int32_t cnstr_shdsc_wifi_decap(u_int32_t *descbuf, u_int16_t *bufsize,  
                             struct wifi_decap_pdb *pdb,  
                             struct cipherparams *cipherdata);
```

Inputs:

- `descbuf`
 - Pointer to buffer used for descriptor construction
- `bufsize`
 - Pointer to size to be written back upon completion
- `pdb`
 - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the decapsulation PDB.
- `cipherdata`
 - Pointer to blockcipher transform definitions

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

7.4.10.9.2.2.7 *cnstr_shdsc_wimax_encap()*

Construct a shared protocol-level descriptor capable of performing IEEE 802.16 WiMAX message encapsulation:

```
int32_t cnstr_shdsc_wimax_encap(u_int32_t *descbuf, u_int16_t *bufsize,  
                               struct wimax_encap_pdb *pdb,  
                               struct cipherparams *cipherdata,  
                               u_int8_t mode);
```

Inputs:

- `descbuf`

- Pointer to buffer used for descriptor construction

- `bufsize`

- Pointer to size value to be written back upon completion

- `pdb`

- Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the encapsulation PDB.

- `cipherdata`

- Pointer to cipher parameters. Only

`key`

and

`keylen`

are used for this descriptor.

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

7.4.10.9.2.2.8 *cnstr_shdsc_wimax_decap()*

Construct a shared protocol-level descriptor capable of performing IEEE 802.16 WiMAX message decapsulation:

```
int32_t cnstr_shdsc_wimax_decap(u_int32_t *descbuf, u_int16_t *bufsize,
                               struct wimax_decap_pdb *pdb,
                               struct cipherparams *cipherdata,
                               u_int8_t mode);
```

Inputs:

- `descbuf`

- Pointer to buffer used for descriptor construction

- `bufsize`

- Pointer to size value to be written back upon completion

- `pdb`

- Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the decapsulation PDB.

- `cipherdata`

- Pointer to cipher parameters. Only

`key`

and

```
keylen
```

are used for this descriptor.

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

7.4.10.9.2.2.9 *cnstr_shdsc_macsec_encap()*

Construct a shared protocol-level descriptor capable of performing IEEE 802.1AE MACsec message encapsulation:

```
int32_t cnstr_shdsc_macsec_encap(u_int32_t *descbuf, u_int16_t *bufsize,  
                                struct macsec_encap_pdb *pdb,  
                                struct cipherparams *cipherdata);
```

- ```
descbuf
```

  - Pointer to buffer used for descriptor construction
- ```
bufsize
```

 - Pointer to size value to be written back upon completion
- ```
pdb
```

  - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the encapsulation PDB.
- ```
cipherdata
```

 - Pointer to cipher parameters. Only

```
key
```

and

```
keylen
```

are used for this descriptor.

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

7.4.10.9.2.2.10 *cnstr_shdsc_macsec_decap()*

Construct a shared protocol-level descriptor capable of performing IEEE 802.1AE MACsec message decapsulation:

```
int32_t cnstr_shdsc_macsec_decap(u_int32_t *descbuf, u_int16_t *bufsize,  
                                 struct macsec_decap_pdb *pdb,  
                                 struct cipherparams *cipherdata);
```

Inputs:

- ```
descbuf
```

- Pointer to buffer used for descriptor construction

- `bufsize`

- Pointer to size value to be written back upon completion

- `pdb`

- Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the encapsulation PDB.

- `cipherdata`

- Pointer to cipher parameters. Only

- `key`

and

- `keylen`

are used for this descriptor.

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

#### 7.4.10.9.2.2.11 *cnstr\_shdsc\_snow\_f8()*

Construct a shared descriptor capable of performing SNOW UEA2 confidentiality message processing:

```
int32_t cnstr_shdsc_snow_f8(u_int32_t *descbuf, u_int16_t *bufsize,
 u_int8_t *key, u_int32_t keylen,
 enum algdir dir, u_int32_t count,
 u_int8_t bearer, u_int8_t direction);
```

Inputs:

- `descbuf`

- pointer to descriptor-under-construction buffer

- `bufsize`

- points to size to be updated at completion

- `key`

- cipher key

- `keylen`

- size of key in bits

- `dir`

- cipher direction (DIR\_ENCRYPT/DIR\_DECRYPT)

- `count`  
- UEA2 count value (32 bits)
- `bearer`  
- UEA2 bearer ID (5 bits)
- `direction`  
- UEA2 direction (1 bit)

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

#### 7.4.10.9.2.2.12 *cnstr\_shdsc\_snow\_f9()*

Construct a shared descriptor capable of performing SNOW UIA2 message authentication:

```
int32_t cnstr_shdsc_snow_f9(u_int32_t *descbuf, u_int16_t *bufsize,
 u_int8_t *key, u_int32_t keylen,
 enum algdir dir, u_int32_t count,
 u_int32_t fresh, u_int8_t direction);
```

Inputs:

- `descbuf`  
- Pointer to buffer for descriptor construction
- `bufsize`  
- Pointer to descriptor size to be updated upon completion
- `key`  
- Cipher key
- `keylen`  
- Size of cipher key
- `dir`  
- Cipher direction (  
`DIR_ENCRYPT/DIR_DECRYPT`  
)
- `count`  
- UEA2 count value (32 bits)
- `fresh`  
- UEA2 fresh value ID (32 bits)

- `direction`  
 - UEA2 direction (1 bit)
- `clear`  
 - Nonzero if descriptor buffer clear requested

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

#### 7.4.10.9.2.2.13 *cnstr\_shdsc\_kasumi\_f8()*

Construct a shared descriptor capable of performing Kasumi f8 confidentiality message processing:

```
int32_t cnstr_shdsc_snow_f8(u_int32_t *descbuf, u_int16_t *bufsize,
 u_int8_t *key, u_int32_t keylen,
 enum algdir dir, u_int32_t count,
 u_int8_t bearer, u_int8_t direction);
```

Inputs:

- `descbuf`  
 - pointer to descriptor-under-construction buffer
- `bufsize`  
 - points to size to be updated at completion
- `key`  
 - cipher key
- `keylen`  
 - size of key in bits
- `dir`  
 - cipher direction (  
`DIR_ENCRYPT/DIR_DECRYPT`  
 )
- `count`  
 -f8count value (32 bits)
- `bearer`  
 - f8 bearer ID (5 bits)
- `direction`  
 - f8 direction (1 bit)

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

#### 7.4.10.9.2.2.14 *cnstr\_shdsc\_kasumi\_f9()*

Construct a shared descriptor capable of performing Kasumi f9 message authentication:

```
int32_t cnstr_shdsc_snow_f9(u_int32_t *descbuf, u_int16_t *bufsize,
 u_int8_t *key, u_int32_t keylen,
 enum algdir dir, u_int32_t count,
 u_int32_t fresh, u_int8_t direction);
```

Inputs:

- descbuf - Pointer to buffer for descriptor construction
- bufsize - Pointer to descriptor size to be updated upon completion
- key - cipher key
- keylen - size of cipher key
- dir - cipher direction (DIR\_ENCRYPT/DIR\_DECRYPT)
- count - f9 count value (32 bits)
- fresh - f9 fresh value ID (32 bits)
- direction - f9 direction (1 bit)

Returns:

1. -1 if the descriptor creation failed for any reason, zero if creation succeeded.

#### 7.4.10.9.2.2.15 *cnstr\_shdsc\_cbc\_blkcipher()*

Construct a shared descriptor capable of performing CBC blockcipher confidentiality processing:

```
int32_t cnstr_shdsc_cbc_blkcipher(u_int32_t *descbuf, u_int16_t *bufsize,
 u_int8_t *key, u_int32_t keylen,
 u_int8_t *iv, u_int32_t ivlen,
 enum algdir dir, u_int32_t cipher,
 u_int8_t clear);
```

Inputs:

- descbuf  
- Pointer to buffer for descriptor construction
- bufsize  
- Pointer to descriptor size to be updated upon completion
- key  
- Pointer to cipher key
- keylen  
- Size of cipher key

- `iv`  
 - Pointer to IV data
- `ivsize`  
 - Size of IV
- `dir`  
 - Cipher direction (  
`DIR_ENCRYPT/DIR_DECRYPT`)
- `cipher`  
 - Cipher selection (  
`OP_ALG_ALGSEL_AES/DES/3DES`)
- `clear`  
 - Nonzero if descriptor buffer clear is requested

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

#### 7.4.10.9.2.2.16 *cnstr\_shdsc\_hmac()*

Construct a shared descriptor capable of performing hashed message authentication processing:

```
int32_t cnstr_shdsc_hmac(u_int32_t *descbuf, u_int16_t *bufsize,
 u_int8_t *key, u_int32_t cipher, u_int8_t *icv,
 u_int8_t clear);
```

Inputs:

- `descbuf`  
 - Pointer to buffer for descriptor construction
- `bufsize`  
 - Pointer to size of descriptor to be updated upon
- `key`  
 - Pointer to key data. Note that key length will be automatically selected based on the HMAC cipher chosen.
- `cipher`  
 - HMAC cipher selection, one of

`OP_ALG_ALGSEL_MD5/SHA1/SHA224/SHA256/SHA384/SHA512`

- `icv`  
- HMAC comparison for ICV, NULL if no check desired

- `clear`  
- Nonzero if descriptor buffer clear is requested

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

#### 7.4.10.9.2.2.17 `cnstr_pcl_shdsc_3gpp_rlc_decap()`

Note: this is a P4080 tapeout #1 operation, to be replaced in a future release.

Construct a shared descriptor capable of performing 3GPP RLC message decapsulation operations:

```
int32_t cnstr_pcl_shdsc_3gpp_rlc_decap(u_int32_t *descbuf, u_int16_t *bufsize,
 u_int8_t *key, u_int32_t keysz,
 u_int32_t count, u_int32_t bearer,
 u_int32_t direction,
 u_int16_t payload_sz, u_int8_t clear);
```

Inputs:

- `descbuf`  
- Pointer to buffer for descriptor construction
- `bufsize`  
- Pointer to size of descriptor to be updated upon completion
- `key`  
- Pointer to f8 cipher key
- `keysz`  
- Size of cipher key
- `count`  
- f8 count value
- `bearer`  
- f8 bearer value
- `direction`  
- f8 direction value
- `payload_sz`  
- Size of payload to be processed (descriptor generated does not use VLF).
- `clear`

- clear descriptor buffer before construction

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

#### 7.4.10.9.2.2.18 *cnstr\_pcl\_shdsc\_3gpp\_rlc\_encap()*

Note: this is a P4080 tapeout #1 operation, to be replaced in a future release.

Construct a shared descriptor capable of performing 3GPP RLC message encapsulation operations:

```
int32_t cnstr_pcl_shdsc_3gpp_rlc_encap(u_int32_t *descbuf, u_int16_t *bufsize,
 u_int8_t *key, u_int32_t keysz,
 u_int32_t count, u_int32_t bearer,
 u_int32_t direction,
 u_int16_t payload_sz);
```

Inputs:

- descbuf  
- Pointer to buffer for descriptor construction
- bufsize  
- Pointer to size of descriptor to be updated upon completion
- key  
- Pointer to f8 cipher key
- keysz  
- Size of cipher key
- count  
- f8 count value
- bearer  
- f8 bearer value
- direction  
- f8 direction value
- payload\_sz  
- Size of payload to be processed (descriptor generated does not use VLF).

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

### 7.4.10.9.3 Disassembler

#### 7.4.10.9.3.1 caam\_desc\_disasm()

```
caam_desc_disasm()
```

-Top-level descriptor disassembler

```
void caam_desc_disasm(u_int32_t *desc, u_int32_t opts);
```

Inputs:

- desc

-points to the descriptor to disassemble. First command must be a header, or shared header, and the overall size to disassemble is determined by the header. Does not handle a QI preheader as its first command, and cannot yet follow links in a list of descriptors.

- opts

- selects options to add to the disassembled output:

```
DISASM_SHOW_OFFSETS
```

- shows the index/offset of each instruction in the descriptor preceding the textual disassembly. This is useful for visualizing flow control changes in a descriptor, since any offset to a specific instruction in the disassembly will be displayed both as a relative number of instructions, and as the offset of the specific instruction.

```
DISASM_SHOW_RAW
```

- shows the hexadecimal value of each instruction before the displayed value of the instruction itself.

## 7.4.11 Runtime Assembler Library Reference

Use the Runtime Assembler Library to write SEC descriptors.

### 7.4.11.1 Runtime Assembler Library Reference

Use the Runtime Assembler Library to write SEC descriptors. This reference describes the structure, concept, functionality, and high level API.

The link below leads to a supplemental directory. Download the file you need from this set.

For more information see [Runtime Assembler Library Reference](#)

## 7.4.12 USDPAA PME Loopback User Guide

### 7.4.12.1 Introduction

The User Space Datapath Acceleration Architecture (USDPAA) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document describes the "pme\_loopback" application. This application serves as an example for working with the PME USDPAA interface, as well as providing a benchmark for PME USDPAA system performance.

### 7.4.12.1.1 Purpose

This document describes the USDPAA pme\_loopback application.

The material is technical in nature. The reader is assumed to be familiar with:

- General Linux software development, operation, and configuration for Power architecture devices in particular.
- Familiarity with the concept of device drivers and their role in operating systems.
- Linux network administration and use.
- The NXP Linux SDK for DPAA-based SoCs.
- At least broadly, the DPAA hardware blocks.
- The USDPAA User Guide document

### 7.4.12.1.2 Change history

Table 162. Change History

| Version | Updates                              |
|---------|--------------------------------------|
| 1.0     | Initial creation of pme_loopback UG. |

### 7.4.12.2 Overview of pme\_loopback

The pme\_loopback is an interactive command line driven USDPAA application which generates and consumes frames to/from the PME hardware accelerator via the PME USDPAA APIs. The pme\_loopback application focuses on demonstrating I/O performance through the PME from applications running on multiple cores. The application creates up to one core affine thread per core which sends a pre-generated frame to PME for scanning and processes the resulting scan result. The application does very little processing of the scan responses. it only does enough to determine if a match was found and updates state information.

When instructed via the CLI, each application thread does the following:

- initializes a pme\_ctx object
- prepares an FD and associated payload data for PME scanning
- commences sending pme scan requests and processing the scan results
- terminates sending scan request and process all expected scan result
- displays performance results
- disables and frees pme\_ctx object

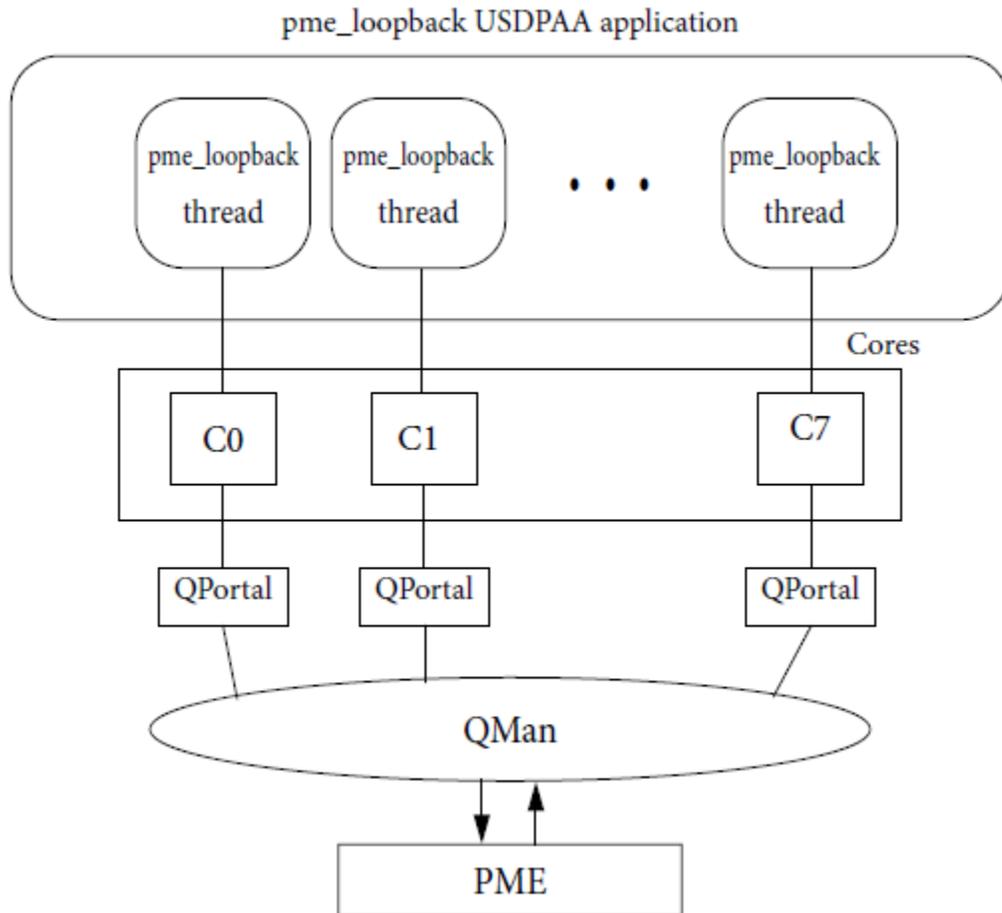
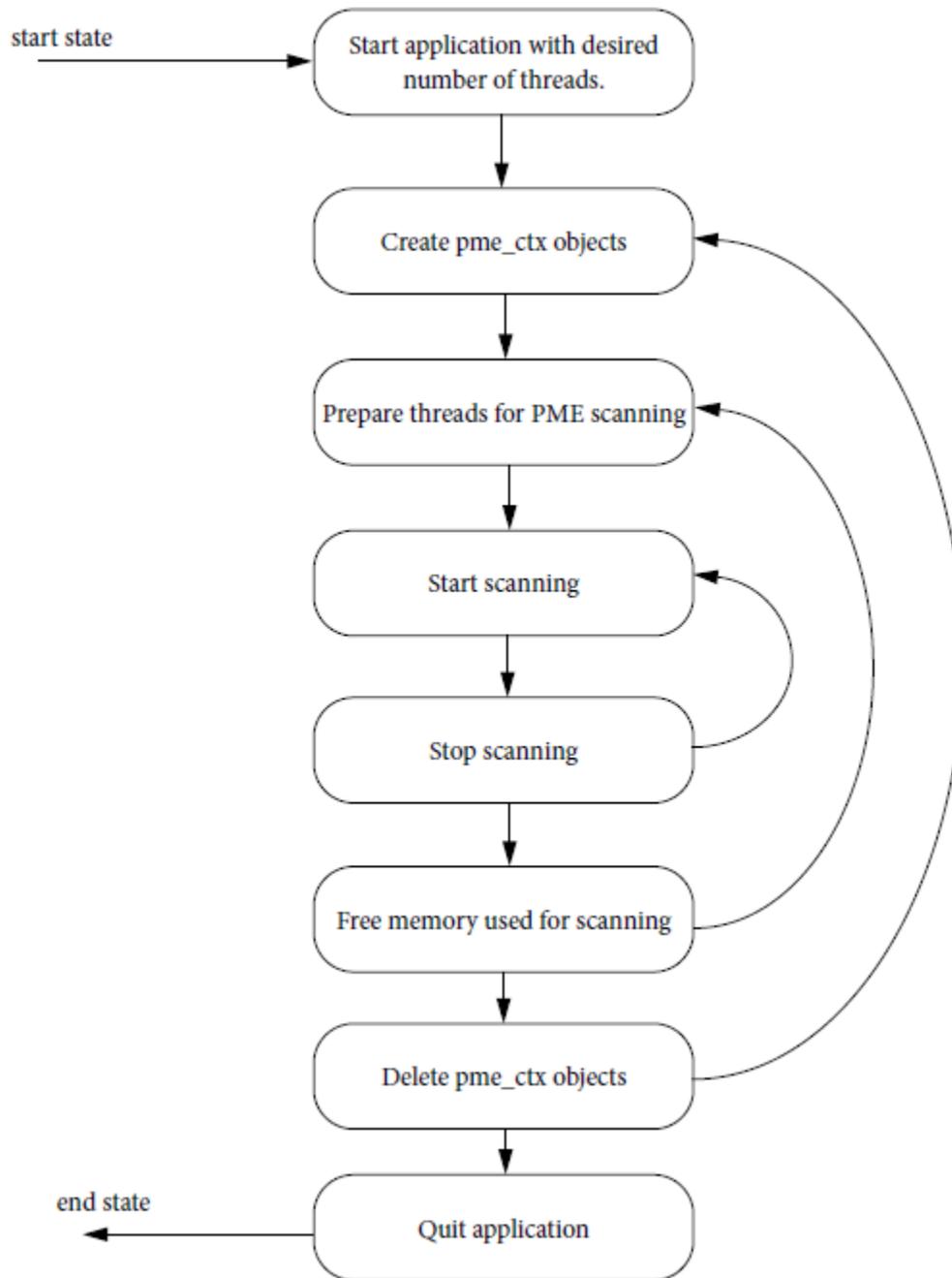


Figure 154. pme\_loopback block diagram

### 7.4.12.2.1 pme\_loopback application flow

The pme\_loopback application transitions into various states as instructed by the CLI commands sent.



**Figure 155. pme\_loopback state transition diagram**

Not all commands are valid depending on the current state. Unexpected behaviour may occur if an invalid command is sent. At this time the application does not validate if the request should not be performed. Below is the suggested method of running the pme\_loopback application. Detailed description of pme\_loopback cli commands can be found in [pme\\_loopback application syntax](#) on page 1020.

1. start the pme\_loopback application with the desired number of threads
2. Create the pme\_ctx objects associated with each thread via create\_ctx\_direct\_mode or create\_ctx\_flow\_mode commands.
3. prepare the threads FD data for scanning via the prep\_scan or prep\_scan\_2 commands

4. start the scanning loop via start\_scan command
5. after letting the application scan data through PME for a desired amount of time (e.g. 30 seconds), stop the scanning process via the stop\_scan command.
6. collect the performance results displayed by the application
7. free the memory used by the threads for scanning via the free\_mem command
8. return to step 3 if there is a desire to scan data with different characteristics (e.g. SUI size, SUI pattern, etc) as defined by the prep\_scan and prep\_scan\_2 commands. Otherwise proceed to the next step.
9. Delete the pme\_ctx objects associated with each thread via the delete\_ctx command.
10. To re-initialize the pme\_ctx objects associated with each thread return to step 2 or continue to next step
11. remove the threads and quit the application. The quit command will do both of these steps.

### 7.4.12.3 pme\_loopback application syntax

The pme\_loopback USDPAAs application is a command line driven application. Each command instructs the application to perform a task which is usually tightly co-related to a specific PME USDPAAs API.

Summary of commands:

```
pme_loopback_test [core_ids]
create_ctx_direct_mode [thread_ids]
create_ctx_flow_mode session_id ren [thread_ids]
prep_scan sui_size_in_bytes pattern_width low_threshold high_threshold use_compound_frame
[thread_ids]
prep_scan_2 sui_size_in_bytes pattern_data low_threshold high_threshold
use_compound_frame [thread_ids]
start_scan [thread_ids]
stop_scan [thread_ids]
display_stats [thread_ids]
clear_stats [thread_ids]
free_mem [thread_ids]
delete_ctx [thread_ids]
rm [core_ids]
add [core_ids]
list
help
quit
```

#### 7.4.12.3.1 pme\_loopback\_test

USDPAAs pme\_loopback test application

##### Synopsis

```
pme_loopback_test [core_ids]
```

##### Description

Start the pme\_loopback USDPAAs application on the specified cores. This is an interactive command line driven application. A prompt ">" is displayed awaiting instructions. If no option is specified a single USDPAAs thread is running on core 0.

##### OPTIONS

```
core_ids
```

Identifies which cores shall have a USDPAA thread created on. It identifies either a single core or a list of cores. The format is as follows:

```
single_core | first_core..last_core
single_core = minimum core id up to maximum core id
first_core < last_core
```

### EXAMPLES

To start `pme_loopback` on cores 0,1,2,3,4,5,6 and 7

```
pme_loopback_test 0..7
```

To start `pme_loopback` on core 2

```
pme_loopback_test 2
```

## 7.4.12.3.2 create\_ctx\_direct\_mode

Initialize a `pme_ctx` object on the specified threads.

### Synopsis

```
create_ctx_direct_mode [thread_ids]
```

### Description

Each specified thread will invoke the `pme_ctx_init()` API. This API initializes the `pme_ctx` object which is associated to the thread and will specify the `DIRECT` `pme_ctx` flag. If no option is specified the command is sent to all the threads.

### OPTIONS

`thread_ids`

Identifies which core affine thread shall receive this command. The `thread_id` has the same value as the `core_ids` indicated in [pme\\_loopback\\_test](#) on page 1020. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
where: first_thread < last_thread
```

### EXAMPLES

To initialize `pme_ctx` object in direct mode on all `pme_loopback` USDPAA threads:

```
>create_ctx_direct_mode
```

To initialize a `pme_ctx` object in direct mode on the USDPAA threads which are on cores 0 to 7:

```
>create_ctx_direct_mode 0..7
```

To initialize a `pme_ctx` object in direct mode on the USDPAA thread which is on core 5:

```
>create_ctx_direct_mode 5
```

## 7.4.12.3.3 create\_ctx\_flow\_mode

Initialize a `pme_ctx` object in flow mode on the specified threads.

### Synopsis

```
create_ctx_flow_mode session_id ren [thread_ids]
```

### Description

Each specified thread will invoke the `pme_ctx_init()` API. This API initializes the `pme_ctx` object which is associated to the thread. The `pme_ctx` will be initialized in flow mode and will have the specified `session_id` and whether its residue is on or off is determined by the `ren` parameter. If no `thread_ids` option is specified the command is sent to all the threads.

`session_id`

Index where the per-session context for this Flow will be accessed by SRE in the Session Context Table. The minimum `session_id` is 0. The maximum `session_id` is equal to `/dev/fsl-pme-dev/sre_session_ctx_num - 1`.

ex: # `cat /dev/fsl-pme-dev/sre_session_ctx_num 80`

Therefore the maximum `session_id` is 79.

`ren`

Indicates is residue is enabled or not. zero (0) indicates residue is disabled. One (1) indicated residue is enabled.

## OPTIONS

`thread_ids`

Identifies which core affine thread shall receive this command. The `thread_id` has the same value as the `core_ids` indicated in [pme\\_loopback\\_test](#) on page 1020. If not specified, all threads receive this command. The format is as follows:

`single_thread | first_thread..last_thread`

where: `first_thread < last_thread`

## EXAMPLES

To initialize `pme_ctx` object in flow mode on all `pme_loopback` USDPAAs threads with `session_id` zero and residue disabled:

```
>create_ctx_flow_mode 0 0
```

To initialize a `pme_ctx` object in flow mode with `session_id` zero and residue enabled on the USDPAAs threads which are on cores 0 to 7:

```
>create_ctx_flow_mode 0 1 0..7
```

To initialize a `pme_ctx` object in flow mode with `session_id` 1 and residue enabled on the USDPAAs thread which is on core 5:

```
>create_ctx_direct_mode 1 1 5
```

## 7.4.12.3.4 prep\_scan

Prepare the specified threads for `pme` scanning.

### Synopsis

```
prep_scan sui_size_in_byte pattern_width low_threshold high_threshold use_compound_frame [thread_ids]
```

### Description

Prepares the specified threads for `pme` scanning. This command will allocate memory for the SUI and will construct an appropriate FD: a compound frame is used if specified. The resulting FD will be used to repeatedly send `pme` scans. The thread will send scan requests until the `high_threshold` is reached. It will then switch to processing scan results until the `low_threshold` is reached at which point it will return to sending scans. The thread remains in this loop until the "stop\_scan" command is received.

The content of the SUI is determined as follows:

There is an internal 50 character alphabet

```
1 2 3 4 5 6 7 8 9 0 a b c d e f g h i j k l m n o p q r s t u v w x y z ! @ # $ % ^ & * () [] { } ?
" ;
```

The SUI which is of `sui_size_in_byte` byte long is filled with the above alphabet in an alternating pattern. Any remaining bytes are filled with the period "." symbol. The pattern is as follows:

- first character in the alphabet is repeated every 1 x pattern\_width bytes
- second character in the alphabet is repeated every 2 x pattern\_width bytes
- etc...

For example if a sui\_size of 65 and a pattern\_width of 5 is chosen the resulting pattern will emerge:

"1" is repeated every 5 bytes.  
 "2" is repeated every 10 bytes.  
 "3" is repeated every 15 bytes.  
 ...etc

```

1XXXX
12XXX
1X3XX
12X4X
1XXX5
123XX
1XXXX
12X4X
1X3XX
12XX5
1XXXX
1234X
1XXXX

```

Then the pme database can be populated with the pmm tool with appropriate signatures. For instance if the signature /4/ is set in the database and the above SUI is sent then there will be a match frequency of every 20 bytes.

If no thread\_ids option is specified the command is sent to all the threads.

`sui_size_in_bytes`

This number of bytes will be allocated for the SUI which will be used to construct a frame descriptor. Initially the entire SUI is filled with the period character ".".

`pattern_width`

The SUI is filled with an alternating pattern which is of this width. If zero, no alternating pattern will be used, the default all period "." pattern is used. The maximum value is 50.

`low_threshold`

The specified threads process scan results until the number of outstanding scans reduces to this amount.

`high_threshold`

The specified threads generate scan requests until the number of outstanding requests reaches this count: at which point the thread starts to process scan results.

`use_compound_frame`

A frame descriptor can be either be contiguous or a compound frame. If use\_compound\_frame is 0 then a contiguous frame format is used, otherwise a compound frame format is used. The compound frame output frame is of zero size bytes.

**OPTIONS**

`thread_ids`

Identifies which core affined thread shall receive this command. The thread\_id has the same value as the core\_ids indicated in [pme\\_loopback\\_test](#) on page 1020. If not specified, all threads receive this command. The format is as follows:

`single_thread | first_thread..last_thread`

where: `first_thread < last_thread`

## EXAMPLES

The following example does the following:

- prepare the SUI in all threads with SUI size of 65 bytes
- no pattern width will be used (default all period)
- a `low_threshold` in flight SUI count of 15 and a `high_threshold` in flight SUI count of 30
- `compound_frame` is not used (i.e. contiguous is used)

```
>prep_scan 65 0 15 30 0
```

### 7.4.12.3.5 prep\_scan\_2

Prepare the specified threads for pme scanning.

#### Synopsis

```
prep_scan_2 sui_size_in_bytes pattern_data low_threshold high_threshold use_compound_frame
[thread_ids]
```

#### Description

This is a second version of the `prep_scan` command. The command is similar to `prep_scan` except that this version has the `pattern_data` parameter instead of `pattern_width`. The `pattern_data` is the literal string of characters that will be copied into the SUI. If the `pattern_data` is larger than the SUI size then the `pattern_data` will be truncated. If the `pattern_data` is shorter than the SUI, the SUI is filled with the period "." character. This command will allocate memory for the SUI and will construct an appropriate frame descriptor (FD): a compound frame is used if specified. The resulting FD will be used to repeatedly send pme scans. The thread will send scan requests until the `high_threshold` is reached. It will then switch to processing scan results until the `low_threshold` is reached at which point it will return to sending scans. The thread remains in this loop until the "stop\_scan" command is received.

If no `thread_ids` option is specified the command is sent to all the threads.

`sui_size_in_bytes`

This number of bytes will be allocated for the SUI which will be used to construct a frame descriptor. Initially the entire SUI is filled with the period character ".".

`pattern_data`

A literal string of characters that will be copied in the SUI.

`low_threshold`

The specified threads process scan results until the number of outstanding scans reduces to this amount.

`high_threshold`

The specified threads generate scan requests until the number of outstanding requests reaches this count: at which point the thread starts to process scan results.

`use_compound_frame`

A frame descriptor can be either be contiguous or a compound frame. If `use_compound_frame` is 0 then a contiguous frame format is used, otherwise a compound frame format is used. The compound frame output frame is of zero size bytes.

#### OPTIONS

`thread_ids`

Identifies which core affined thread shall receive this command. The `thread_id` has the same value as the `core_ids` indicated in [pme\\_loopback\\_test](#) on page 1020. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
```

where: `first_thread < last_thread`

### EXAMPLES

The following example does the following:

- prepare the SUI in all threads with SUI size of 10 bytes
- the SUI pattern being sent is abcdefghij
- a `low_threshold` in flight SUI count of 15 and a `high_threshold` in flight SUI count of 30
- `compound_frame` is not used (i.e. contiguous is used)

```
>prep_scan_2 10 abcdefghij 15 30 0
```

### 7.4.12.3.6 start\_scan

Instruct the specified threads to commence sending pme scan requests and processing the results.

#### Synopsis

```
start_scan [thread_ids]
```

#### Description

Each specified thread will enter the following loop:

```

sending scan loop
do
 invoke the pme_ctx_scan
 increment in_flight_scans by one
 while (in flight scans < high_threshold)
 processing scan response loop
 do
 qman_poll_dqrr(16)
 while (in_flight_scans >= low_threshold)

```

The `qman_poll_dqrr()` api will invoke the function callback specified in the `pme_ctx` object. This callback will decrement the `in_flight_scans` counter and also update some statistics such as number of notifications and truncations received.

The threads will remain in this loop until the `stop_scan` command is received.

#### OPTIONS

`thread_ids`

Identifies which core affine thread shall receive this command. The `thread_id` has the same value as the `core_ids` indicated in [pme\\_loopback\\_test](#) on page 1020. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
```

where: `first_thread < last_thread`

### EXAMPLES

To instruct all threads to start their scanning loop:

```
>start_scan
```

To instruct threads 0 to 4 to start its scanning loop:

```
>start_scan 0..4
```

To instruct thread 2 to start its scanning loop:

```
>start_scan 2
```

### 7.4.12.3.7 stop\_scan

Instruct the specified threads to stop sending pme scan requests and complete processing all remaining scan results.

#### Synopsis

```
stop_scan [thread_ids]
```

#### Description

Each specified thread will stop sending scan requests and will process all scan results until the `in_flight_scan` counter reaches zero. The following performance statistics are displayed:

- Total units scanned
- Total number of SUIs sent to the PME device.
- Total time
- The time in seconds from the moment the first scan unit is sent until the last scan response is processed.
- Scan Units per second
- Total units scanned / Total time
- Bandwidth

The number of SUIs send and corresponding responses processed measured in Mbps.

i.e. 9512 Mbps

#### OPTIONS

```
thread_ids
```

Identifies which core affine thread shall receive this command. The `thread_id` has the same value as the `core_ids` indicated in [pme\\_loopback\\_test](#) on page 1020. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
```

where: `first_thread < last_thread`

#### EXAMPLES

To instruct all threads to stop their scanning loop:

```
>stop_scan Total units scanned: 24622356 Total time: 31.007910 sec Scan Units per second: 794066
Bandwidth: 9528 Mbps
```

### 7.4.12.3.8 free\_mem

Instruct the specified threads to free the memory previously allocated via the `prep_scan` or `prep_scan_2` command.

#### Synopsis

```
free_mem [thread_ids]
```

#### Description

Each specified thread will free their memory previously allocated during the `prep_scan` and `prep_scan_2` command. This command needs to be run to undo any previous `prep_scan` or `prep_scan_2` commands.

#### OPTIONS

```
thread_ids
```

Identifies which core affine thread shall receive this command. The `thread_id` has the same value as the `core_ids` indicated in [pme\\_loopback\\_test](#) on page 1020. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
```

where:

```
first_thread < last_thread
```

## EXAMPLES

To instruct all threads to free their allocated memory:

```
>free_mem
```

### 7.4.12.3.9 delete\_ctx

Instruct the specified threads to disable and "finish" their previously initialized pme\_ctx object.

#### Synopsis

```
delete_ctx [thread_ids]
```

#### Description

Each specified thread will invoked the pme\_ctx\_disabled() API on their pme\_ctx object. Once completed they will invoked the pme\_ctx\_finish() API. These APIs free any internal resources used (such as frame queues) and render the pme\_ctx object in an un-initialized state once again.

#### OPTIONS

```
thread_ids
```

Identifies which core affine thread shall receive this command. The thread\_id has the same value as the core\_ids indicated in [pme\\_loopback\\_test](#) on page 1020. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
```

where: first\_thread < last\_thread

## EXAMPLES

To instruct all threads to disable and finish their pme\_ctx object:

```
>delete_ctx
```

### 7.4.12.3.10 rm

Remove a pme USDPAA core affined thread

#### Synopsis

```
rm thread_ids
```

#### Description

Each specified thread will be destroyed by the application. The application will display that the thread on a specific cpu has been killed.

```
thread_ids
```

Identifies which core affine thread shall receive this command. The thread\_id has the same value as the core\_ids indicated in [pme\\_loopback\\_test](#) on page 1020. The format is as follows:

```
single_thread | first_thread..last_thread
```

where: first\_thread < last\_thread

## EXAMPLES

To instruct threads 0 to 7 to be removed:

```
>rm 0..7 Thread killed on cpu 0 Thread killed on cpu 1 Thread killed on cpu 2 Thread killed on cpu 3
Thread killed on cpu 4 Thread killed on cpu 5 Thread killed on cpu 6 Thread killed on cpu 7
```

### 7.4.12.3.11 add

Add a pme USDPAA core affined thread

#### Synopsis

```
add core_ids
```

#### Description

Each specified core will have a pme USDPAA thread created by the application.

```
core_ids
```

Identifies which cores shall have an affine USDPAA thread created. The format is as follows:

```
single_thread | first_thread..last_thread
```

where: first\_thread < last\_thread

#### EXAMPLES

To instruct the application to create threads on cores 1 to 2:

```
>add 1.2 Thread alive on cpu 1 Thread alive on cpu 2
```

### 7.4.12.3.12 list

Display a list of pme USDPAA core affine threads.

#### Synopsis

```
list
```

#### Description

A command is sent to each thread. In response the thread will display on which core it is running on.

#### EXAMPLES

```
>list Thread alive on cpu 0 Thread alive on cpu 1 Thread alive on cpu 2 Thread alive on cpu 3 Thread
alive on cpu 4 Thread alive on cpu 5 Thread alive on cpu 6 Thread alive on cpu 7
```

### 7.4.12.3.13 display\_stats

Instruct the specified threads to display their internal statistics

#### Synopsis

```
display_stats [thread_ids]
```

#### Description

Each specified thread will display their internal statistics.

#### OPTIONS

```
thread_ids
```

Identifies which core affine thread shall receive this command. The thread\_id has the same value as the core\_ids indicated in [pme\\_loopback\\_test](#) on page 1020. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
```

where: first\_thread < last\_thread

#### EXAMPLES

To instruct thread 0 to display its internal statistics:

```
>display_stats 0 in_flight = 0 rx_packets = 0 total_notifs = 0 num_queue_empty = 0 num_erns = 0
num_truncates = 0 full_fifo = 0 Time: 0.000000 secs
```

### 7.4.12.3.14 clear\_stats

Instruct the specified threads to clear their internal statistics

#### Synopsis

```
clear_stats [thread_ids]
```

#### Description

Each specified thread will clear their internal statistics.

#### OPTIONS

```
thread_ids
```

Identifies which core affine thread shall receive this command. The thread\_id has the same value as the core\_ids indicated in [pme\\_loopback\\_test](#) on page 1020. If not specified, all threads receive this command. The format is as follows:

```
single_thread | first_thread..last_thread
```

where: first\_thread < last\_thread

#### EXAMPLES

To instruct all threads to clear their internal statistics:

```
>clear_stats
```

### 7.4.12.3.15 help

Display the list of pme\_loopback commands

#### Synopsis

```
help
```

Description

Prints out all pme\_loopback commands

#### OPTIONS

#### EXAMPLES

To display the list of command:

```
>help
```

Available commands:

```
help add list rm create_ctx_flow_mode create_ctx_direct_mode delete_ctx prep_scan prep_scan_2
start_scan stop_scan free_mem display_stats clear_stats
```

### 7.4.12.3.16 quit

Quit the pme\_loopback application

#### Synopsis

```
quit
```

#### Description

This will shutdown the pme\_loopback application. All threads will first be removed and then the application will exit.

## 7.4.12.4 Running pme\_loopback

Log in to the p4080 DS environment as "root".

```
login: root
Password:
[root@p4080 root]#
```

Clear the pme database via pmm application:

```
[root@p4080 root]#pmm
Successfully created the PMM DB.
pmm> commit
Successfully committed changes made to the data base of expressions.
Command execution time: 00:00:00 [hour:min:sec].
pmm> quit
Terminating the PMM application.
[root@p4080 root]#
```

Start the pme\_loopback application with threads on cores 0 to 7:

```
/usr/bin/pme_loopback_test 0..7
Qman: FQID allocator includes range 512:128
Bman: BPID allocator includes range 56:8
FSL dma_mem device mapped (phys=0xf8000000,virt=0x70000000,sz=0x4000000)
Thread alive on cpu 0
Thread alive on cpu 1
Thread alive on cpu 2
Thread alive on cpu 3
Thread alive on cpu 4
Thread alive on cpu 5
Thread alive on cpu 6
Thread alive on cpu 7
>
```

A CLI (Command-Line Interface) is presented in order to allow you to enter the next pme\_loopback commands.

Create a pme context in direct mode on each pme USDPAA thread.

```
> create_ctx_direct_mode
```

Prepare the data for scanning.

```
> prep_scan 1024 0 15 30 0
```

Start the scanning

Once the command to start scanning is executed it will stay in this mode scanning data until the stop\_scan command is entered.

```
> start_scan
```

[ ... wait 30 seconds (though as low as 10 seconds can also be used) before typing in stop command below. To allow enough data to be processed by the PME to make the initial command generation overhead amortized over adequate scans. (i.e. the result is a steady state performance result)]

Command to stop Scan

```
> stop_scan
Total units scanned: 34608812
```

```
Total time: 29.804104 sec
Scan Units per second: 1161209
Bandwidth: 9512 Mbps
```

Need to free scan memory which was allocated

```
> free_mem
```

Tear down application, need to delete contexts

```
> delete_ctx
```

Quit application

```
> quit
```

## 7.4.13 USDPAA IPFwd Longest Prefix Match User Manual

### 7.4.13.1 NXP P4080/P5020/P3041 USDPAA IPFwd Longest Prefix Match User Manual

#### 7.4.13.1.1 Introduction

This user manual describes USDPAA IPFwd based upon Longest Prefix Match methodology. This IPFwd application is different from the other route cache based IPFwd. The User Space Datapath Acceleration Architecture (USDPAA) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document provides the following:

- A summary of the USDPAA LPM IPFwd application
- A summary of usage of Shared MAC
- A summary of usage of MAC-less interface
- Execution steps for USDPAA LPM IPFwd application from the NXP SDK package on the P4080 DS/P3041DS/P5020DS

This document describes the USDPAA application which demonstrates:

1. IP Forwarding application using Longest prefix match as route decision algorithm
2. MAC less communication between USDPAA application and kernel
3. Sharing of same physical Ethernet port using shared-mac mode

#### 7.4.13.1.2 Overview

The USDPAA LPM based IPv4 forwarding application is a multi-threaded application that routes IPv4 packets from one ethernet interface to another on all QorIQ platforms. The LPM routing algorithm uses the prefix for destination ip address to do the route look up. Any combination of the cores can run a LPM based USDPAA IPv4 Forwarding application thread. The packets that reach the USDPAA application can be forwarded to destination IP address using LPM route algorithm.

LPM IPFwd packet-processing:

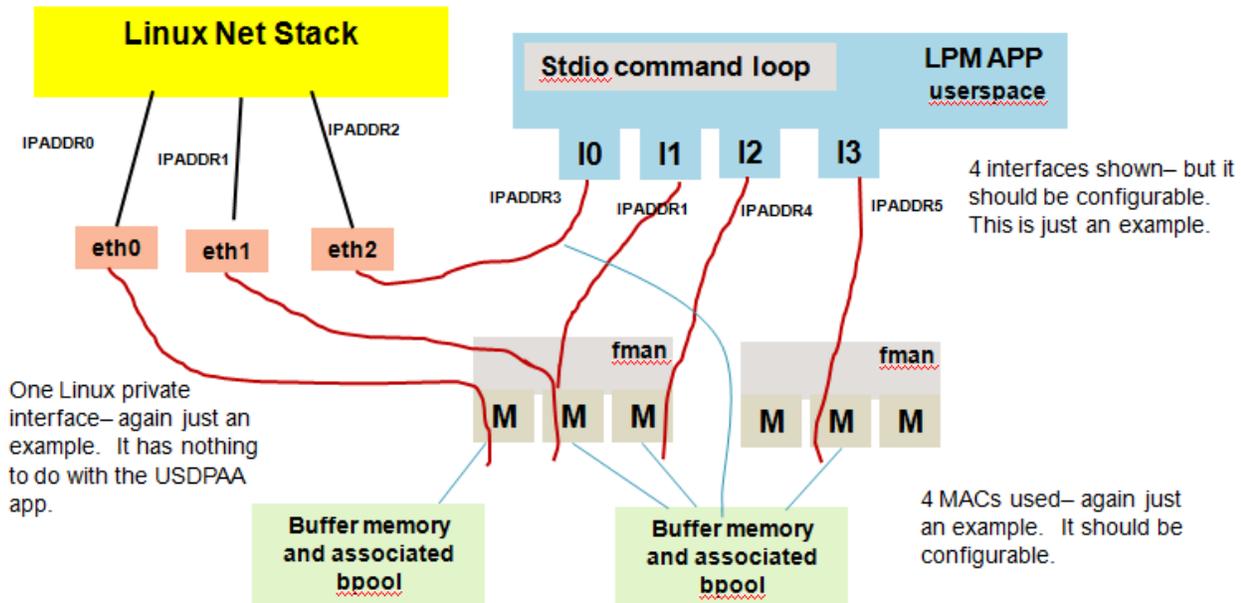
- Receives ethernet frames on ethernet interfaces.
- On the basis of defined PCD rules in FMAN's PCD table, IPv4 traffic would be sent to USDPAA application through PCD Frame queue range.

For IPv4 frames, processing takes place as defined in section [Overview of packet flow](#): on page 1038

This application also demonstrates the following features:

- how the traffic coming from a common MAC port can be split in between kernel and USDPAA application on the basis of defined PCD rules. The packets that reach the USDPAA application can be forwarded to destination IP address using LPM route algorithm.

- ping between linux and USDPAA using MAC-less interface.



IPADDR2 and IPADDR3 are on the same network. Note that IPADDR1 is used twice.

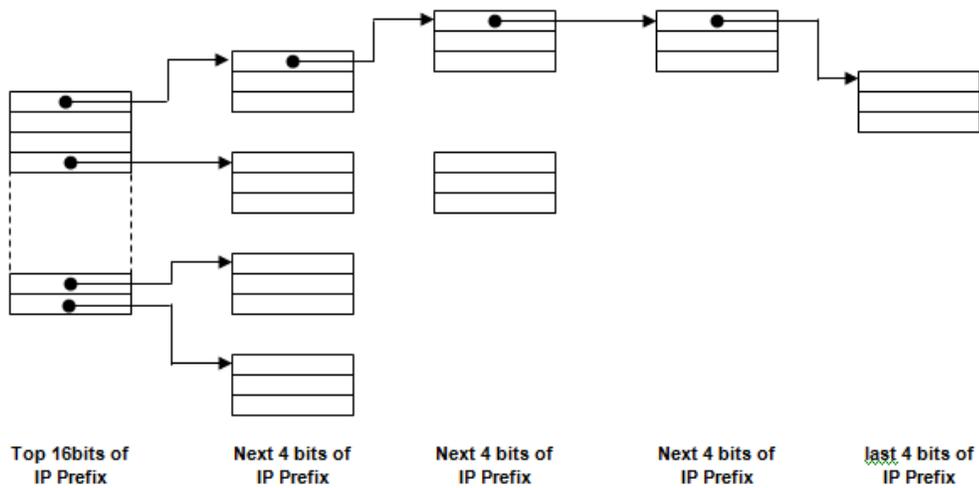
### 7.4.13.13 How is it different from existing Route cache based IPFwd?

This USDPAA application uses Longest Prefix Match algorithm for doing route lookup by using prefix for destination IP address in contrast to the existing route cache based IPFwd which takes route decision based upon hash results calculated by FMAN using source IP address and destination IP address in the frame.

The user will have to use new commands for route addition in LPM table (check the command section)

### 7.4.13.14 Longest Prefix Match algorithm

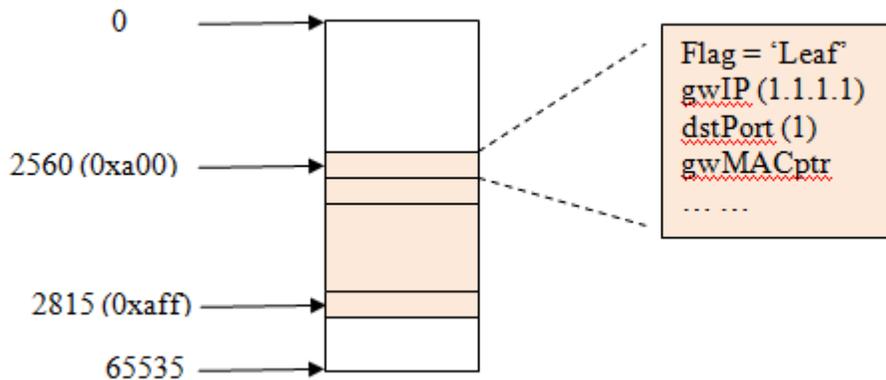
Instead of using traditional Radix-Trie algorithm, here we choose to use one simpler LPM algorithm. It uses 5 level tables: First level table – 65536 entries array, indexed by the top 16 bits of IP address. Second level table – 32 entries array, indexed by bit12~15 of IP address. Third level table – 32 entries array, indexed by bit8~11 of IP address. Fourth level table – 32 entries array, indexed by bit4~7 of IP address. Fifth level table – 32 entries array, indexed by bit0~3 of IP address. The 2nd level to 5th level tables are only created when its first level table entry has valid value. See below figure:



At init time, only the first level (i.e. top16 bits) array is created which contains 65536 null entries. While adding route entries to the FIB table, the 2nd level to 5th level arrays will be created accordingly. This is a typical ASIC design algorithm of LPM which is fast and simple to search while costs far more memory. The worst case is to index and compare 5 times when searching an IP address, but it's still fast enough.

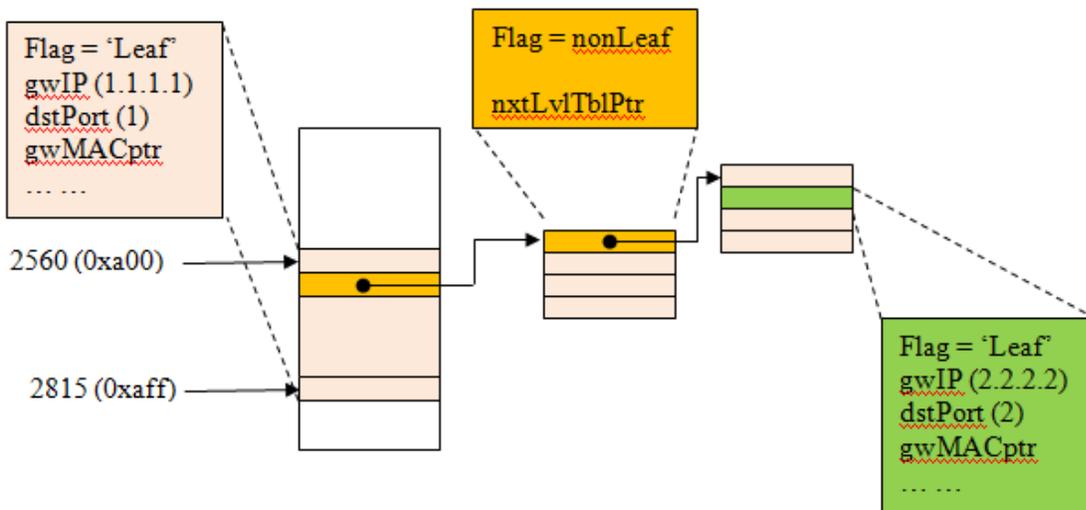
Examples:

1) Add one ClassA route 10.0.0.0/8 to the route table. (gateway is 1.1.1.1, destination port is 1) The first 16bit of 10.0.0.0 (0x0a000000) is 0x0a00 (2560). And the mask is 8 bit which is smaller than the 1st level bit-length (16b), so below entries (from 0x0a00 to 0x0aff) will be created in the FIB table:



From entry No.2560 to No.2815 (total 255 entries) are filled with same content (flag, gwIP, dstPort, ptr ...). Now if a packet with DIP of 10.1.1.1 comes in, its first 16 bit value is 0x0a01 (2561). So, the No.2561 entry of the 1st level table will be checked. If it's a 'leaf' node (now it is), then the best-match is found. And the packet will be forwarded to the 'dstPort' after replacing the SMAC and DMAC. And, any DIP of 10.x.x.x will all be forwarded to port 1 with gwIP 1.1.1.1 based on above table.

2) Now, a new route 10.1.1.0/24 is added to the FIB table. The table will be like this:



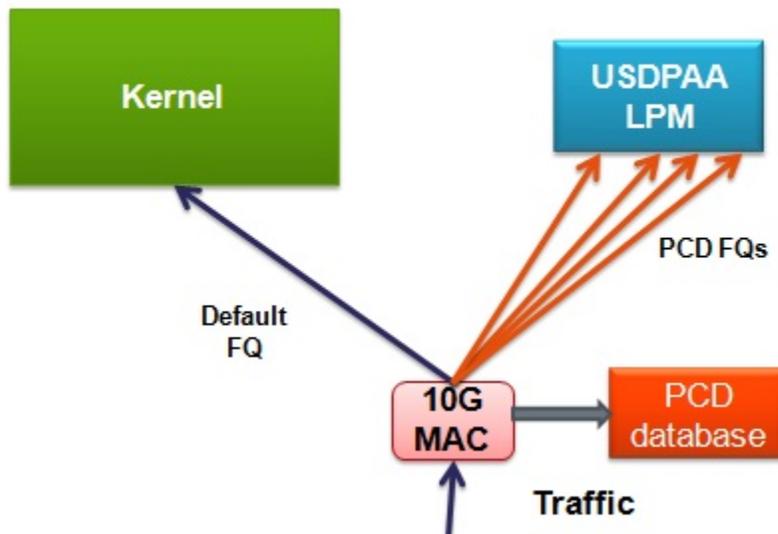
The new route will overwrite the No.2561 (0x0a01) entry with 'flag' of 'nonLeaf' and a pointer to 'next-level-table'. Now a 16-entry-block memory will be allocated as the 'next-level-table' because the 2nd level is 4bit indexed. And the base address of this new block will be set to the 'nxtLvlTblPtr' of No.2561. Because the next 4 bit of the new route is 0 (bit16 to bit19 of 0x0a010100), so the 1st entry in the 2nd level table is used as another 'nonLeaf' entry. While all the other entries in the 2nd level table should be filled with the same value of its 'parent' route (10.0.0.0/8). The netmask is 24bit which is larger than 16+4, so the 3rd level table should also be allocated (16 entries). And the next 4bit of the new route is 1 (bit20 to bit23 of 0x0a010100), so the 2nd entry will be used for the new route. And because the netmask (24bit) is no-larger-than 16+4+4, so this entry will be the 'Leaf' entry of this new route (see above figure in green). And the according values (gwIP, dstPort, etc.) will be filled in that entry. Now a frame with DIP of 10.1.1.100 comes. There will have 3 lookups to get the final result:

- Index with first 16bit of DIP, whose value is 0x0a01. 'Non-leaf' means to continue the next-level lookup.
- Index with the next 4bit of DIP, whose value is 0. Then 'Non-leaf' again.
- Index with the next 4bit of DIP, whose value is 1. Then the 'leaf' node is found and the lookup reaches an end.

Now a frame with DIP of 10.1.192.10 comes. You can see it will find the 'leaf' node in the 2nd level table and get the route of net-address 10.0.0.0/8. And a frame with DIP of 10.1.10.10 will find its 'leaf' node in the 3rd level table and also get the route of net-address 10.0.0.0/8 as we expected. The multi-branch trie algorithm provides a very fast way of route-lookup but a relatively complicated way of route-add/deletion.

### 7.4.13.1.5 Shared MAC Overview

The kernel and USDPAA should be able to receive traffic of their interest from a shared Ethernet port. On the basis of defined PCD rules in FMAN's PCD table, IPv4 traffic ( for non-owned IP addresses) would be sent to USDPAA application through PCD Frame queue range and rest of the traffic ( i.e. ICMP etc) would be sent to kernel through default Frame queue.



### 7.4.13.1.6 How to run shared MAC interface ?

1. Make sure to use the corresponding dtb to "p4080ds-usdpaa-shared-interfaces.dts" file from kernel source. In this example dts file ethernet@9 depicts shared MAC node.

**NOTE**

For p3-p5, use p3041ds-usdpaa-shared-interfaces.dts and p5020ds-usdpaa-shared-interfaces.dts respectively. ethernet@5 depicts shared MAC node.

**NOTE**

For B4, use b4860qds-usdpaa-shared-interfaces.dts where ethernet@9 depicts shared MAC node. And for T4, use t4240qds-usdpaa-shared-interfaces.dts where ethernet@15 depicts shared MAC node.

2. When linux comes up, check the output of "ifconfig -a".

```
root@p4080ds:~# ifconfig -a
fm2-10g Link encap:Ethernet HWaddr 00:e0:0c:00:96:09
inet6 addr: fe80::2e0:cff:fe00:9609/64 Scope:Link
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:5 errors:1 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:378 (378.0 B)
Memory:fe5f0000-fe5f0fff
```

3. Run fmc

```
$cd /usr/etc
fmc -c usdpaa_config_p4_serdes_0xe.xml -p usdpaa_policy_hash_shared_mac_ipv4.xml -a
For p3-p5 use this command:
```

```
fmc -c usdpaa_config_p3_p5_serdes_0x36_shared_mac.xml -p usdpaa_policy_hash_shared_mac_ipv4.xml -a
```

For B4 use this command:

```
fmc -c usdpaa_config_b4_serdes_0x2a_0x98.xml -p usdpaa_policy_hash_shared_mac_ipv4.xml -a
```

For T4 use this command:

```
fmc -c usdpaa_config_t4_serdes_1_1_6_6.xml -p usdpaa_policy_hash_shared_mac_ipv4.xml -a
```

4. Run lpm-ipfwd application

```
$lpm_ipfwd_app -d 0x10000000 -b 1600:1600:1600 -i fm2-10g
```

For p3-p5

```
$lpm_ipfwd_app -d 0x10000000 -b 1600:1600:1600 -i fm1-10g
```

For B4

```
$lpm_ipfwd_app -d 0x10000000 -b 1600:1600:1600 -c usdpaa_config_b4_serdes_0x2a_0x98.xml -p
usdpaa_policy_hash_shared_mac_ipv4.xml -i fm1-mac10
```

For T4

```
$lpm_ipfwd_app -d 0x10000000 -b 1600:1600:1600 -c usdpaa_config_t4_serdes_1_1_6_6.xml -p
usdpaa_policy_hash_shared_mac_ipv4.xml -i fm2-mac10
```

5. Assign ip address to USDPAAs side as well as kernel side as per script "usdpaa\_policy\_hash\_shared\_mac\_ipv4.xml".

In this script, the coarse classification "ip\_dest\_cls" corresponding to this port uses "192.168.44.3" for USDPAAs and "192.168.44.4" for kernel.

---

**NOTE**

Check ip address for p3-p5 as per scripts usdpaa\_config\_p3\_p5\_serdes\_0x36\_shared\_mac.xml and usdpaa\_policy\_hash\_shared\_mac\_ipv4.xml.

---

---

**NOTE**

For B4/T4, use "192.168.44.3" for USDPAAs and "192.168.44.4" for kernel as defined in usdpaa\_policy\_hash\_shared\_mac\_ipv4.xml.

---

ssh to p4080 and give these commands:

```
$lpm_ipfwd_config -E -a true
```

output:

FMAN Interface number: 11

, PortID=1:5 is FMan interface node with MAC Address 00:e0:0c:00:96:09

---

**NOTE**

Check pid from application print "Message queue to send: /mq\_snd\_2536"

---

```
$ lpm_ipfwd_config -P 2536 -F -a 192.168.44.1 -i 11
```

```
$ lpm_ipfwd_config -P 2536 -G -s 192.168.44.3 -m 02:00:c0:a8:a0:02 -r true
```

```
$ lpm_ipfwd_config -P 2536 -B -c 1 -d 192.168.44.3 -n 16 -g 192.168.44.3
```

```
$ifconfig fm2-10g 192.168.44.4
```

6. Now run traffic on fm2-10g using the above ip addresses and can see traffic splitting amongst kernel and USDPAAs.

### 74.13.1.7 MAC-less use case

This section describes how MAC-less interface is being used in this application. The user space configuration commands and USDPAA application can communicate with each other through the management interface which can be a MAC interface (SGMII, RGMII etc) or MAC-less interface. In this application we are just using MAC-less interface. The user can do such a communication with USDPAA LPM application by using `lpm_ipfwd_config` binary. For command reference, please check section [IPv4 forward application Configuration command](#) on page 936.

### 74.13.1.8 How to ping MAC-less interface ?

. Make sure to use the corresponding dts to "p4080ds-usdpaa-shared-interfaces.dts" file from kernel source. In this example dts file `ethernet@10` depicts MAC-less node.

. NOTE: For B4, use `b4860qds-usdpaa-shared-interfaces.dts` and for T4, use `t4240qds-usdpaa-shared-interfaces.dts`. In both cases, `ethernet@16` is the macless node.

. When linux comes up, check the output of "ifconfig -a". The interface containing similar mac address as given under `ethernet@10` node in device tree, is the MAC-less interface.

```
root@p4080ds:~# ifconfig -a
eth0 Link encap:Ethernet HWaddr 00:15:17:1e:22:9e
UP BROADCAST MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
Interrupt:42 Memory:40000000-40020000

eth3 Link encap:Ethernet HWaddr 00:11:22:33:44:55
BROADCAST MULTICAST MTU:1500 Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)
```

The above output shows that "eth3" is the MAC-less node.

. Run `lpm-ipfwd` application

```
$cd /usr/etc
```

```
$lpm_ipfwd_app -d 0x10000000 -b 1600:1600:1600 -i eth3:[66-22-33-44-55-66]
```

The mac address "66-22-33-44-55-66" is the mac that will be assigned as source mac address to USDPAA side of MAC-less interface

. Assign ip address to USDPAA side as well as kernel side.

ssh to p4080 and give these commands:

```
$lpm_ipfwd_config -E -a true
```

Output:

```
MACLESS Interface:
```

```
name : eth3
```

```
$lpm_ipfwd_config -F -a 192.168.55.6 -n eth3
$ifconfig eth3 192.168.55.2
$ping 192.168.55.6
PING 192.168.55.6 (192.168.55.6): 56 data bytes
64 bytes from 192.168.55.6: icmp_seq=0 ttl=64 time=12.413 ms
64 bytes from 192.168.55.6: icmp_seq=1 ttl=64 time=12.431 ms
```

### 7.4.13.1.9 USDPAAs LPM based IPv4 forwarding application flow

The LPM based IPFwd application has two main phases. There is an initial configuration phase and a subsequent packet processing phase.

The configuration phase executes when the application starts. Application threads are created and global initialization of resources is done which is the part of PPAC (see USDPAAs PPAC User Guide for more details). The configuration phase also includes PPAM (i.e. IPFwd) related initialization. Once the configuration phase is completed the IPFwd application moves to the packet processing phase. This application provides a command-line interface to enable users to add and remove routing table and ARP cache entries. For each user input, the appropriate information is communicated to the IPFwd application via standard Posix IPC. Messages are placed onto the message queue till they are received by the IPFwd application. Note that this application does not dynamically resolve ARP – missing ARP entries will result in the application dropping the packet.

In the packet processing phase, the loop migrates from polling mode to a blocking “IRQ mode” whenever LPM IPFwd has looped a certain number of times without any forward progress. For more information on IRQ mode please refer to “USDPAAs PPAC User Guide”. In polling mode – the application constantly looks for data to process on its dedicated QMan portal . Network traffic is classified and distributed by the FMan to frame queues based on source and destination IP address in the packet. There is an associated handler that processes the packets arriving on each frame queue.

### 7.4.13.1.10 Overview of packet flow:

1. Static Route table entries are populated using the user space configuration commands.
2. If a packet received by the FMan is an IPv4 packet it uses 2-tuple (Src IP & Dest IP) to hash the packet to a Rx frame queue. Otherwise if packet is ICMP etc it would be sent through default Frame queue where the buffer is freed.
3. The packet enqueued to PCD FQ to reach USDPAAs LPM based IPFwd application thread running on one of the cores is subjected to LPM based route lookup.
4. All of the threads enter the processing loop where they call Qman\_poll till a frame is received.
5. If a frame is received, the application checks whether the destination IP address exist in the FIB table. For this it calls ip\_route\_lookup(), which does the route look up using LPM algorithm. The LPM IPFWD application does not change its FIB table in response to seeing the first packet in a flow. Instead the FIB table is set only by commands, as mentioned in section [Syntax](#) on page 936.
6. If the route entry for this frame is not present in the FIB table, the frame is dropped. It then continues with Qman\_poll.
7. If the route entry for this frame exists in the FIB table, the frame is sent for forwarding.
8. If the frame is to be forwarded, it is checked if ARP entry exists in ARP table for destination IP address. LPM IPFWD application will not dynamically resolve the ARP. So if sending packets to forward using a regular computer, the user will have to create static ARP entries. On host computer, ARP table can be updated by sending the ARP request. LPM IPFWD will respond to external ARP requests
9. If ARP entry exists, TTL is decremented in L3 header.
10. Finally, the L2 header is updated, which includes changing the dst MAC address in L2 header. The frame is then enqueued to the TX FQ.

### 7.4.13.1.11 Overview of PPAC

The source code to LPM-IPFwd has been reorganized into two parts; the “PPAC” (Packet-Processing Application Core) and a “PPAM” (Packet-Processing Application Module). The PPAM portion implements the LPM-IPFWD application specific logic of processing the packet using longest prefix match and forward it. On the other hand, the PPAC component represents the common infrastructure to support PPAM; initializing devices, handling flow-control, implementing a CLI (Command-Line Interface), managing threads and buffers. PPAC details can be found in the document “USDPAAs PPAC User Guide”.

### 7.4.13.1.12 Compile-time configuration

PPAC-based application are compiled using a certain set of options that are currently defined in the header located at apps/include/ppac.h. The following describes the most useful options for modification if alternative application behaviour is desired.

### 7.4.13.1.13 Order Preservation in LPM-IPFWD

This section describes how user can enable Order Preservation in LPM-IPFWD application. By default Order Preservation is disabled in LPM-IPFWD application and in order to enable it the user will have to re-compile the binary by making following changes to the source code.

In file, usdpaa/apps/include/ppac.h you can find these two lines.

```
/* Application options */
#undef PPAC_HOLDACTIVE /* Process each FQ on one portal at a time */
#undef PPAC_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */
```

Change the above to

```
#define PPAC_HOLDACTIVE /* Process each FQ on one portal at a time */
#define PPAC_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */
```

And then compile usdpaa once again. Now run the LPM-IPFWD application with Order Preservation.

### 7.4.13.1.14 Order Restoration in LPM IPFWD

Order restoration is the functionality of QMan software portal interface which restores the relative temporal order of a flow of frames (sequence of frames) to that observed before transmitting to the destination Frame Queue and Order Definition Point (ODP) takes note of the correct order of packets before start processing by using the sequence number. Use of “HOLDACTIVE” is mutually exclusive with another QMan option “AVOIDBLOCK”, which is selected by default in PPAC. To enable order-restoration, the user will have to re-compile the binary by making following changes to the source code.

```
/* Application options */
#undef PPAC_HOLDACTIVE /* Process each FQ on one portal at a time */
#undef PPAC_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */
#undef PPAC_2FWD_ORDER_RESTORATION /* Use ORP */
#define PPAC_AVOIDBLOCK /* No full-DQRR blocking of FQs */
```

Change the above to

```
#undef PPAC_HOLDACTIVE /* Process each FQ on one portal at a time */
#undef PPAC_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */
#define PPAC_ORDER_RESTORATION /* Use ORP */
#define PPAC_AVOIDBLOCK /* No full-DQRR blocking of FQs */
```

And then compile usdpaa once again. Now run the LPM-IPFWD application with Order Restoration. Implementation note: Order restoration has been implemented such that each PCD Frame queue has a corresponding ORP (order restoration point) frame queue associated with it. Each ORP is configured with default window settings as seen below

```
#define PPAC_ORP_WINDOW_SIZE 7 /* 0->32, 1->64, 2->128, ... 7->4096 */
#define PPAC_ORP_AUTO_ADVANCE 1 /* boolean */
#define PPAC_ORP_ACCEPT_LATE 3 /* 0->no, 3->yes (for 1 & 2->see RM) */
```

Here the ORP window size is set to be 4K, auto advance window size as 4K and accept late arrival window size as 8K. This ensures that no traffic is getting dropped but are always accepted below and at Zero loss throughput. Beyond zero loss throughput, as usual packets would be dropped and thus you can see mis-ordering.

ORP FQ descriptor attributes settings:

- Prefer in cache
- No "HOLDACTIVE"
- No "AVOIDBLOCK"
- ORP enabled

Assumption: To see the effect of Order Restoration in LPM-IPFwd application the user must use separate streamblocks as a source of traffic. If not done so, mis-ordering would be seen.

Key observation: It has been observed in LPM-IPFwd application that use of "HOLDACTIVE" with traffic generated using separate streamblocks, all the packets are IN sequence. Therefore, it is recommended that if user wants to see the real effect of Order restoration in LPM-IPFwd application he should use "AVOIDBLOCK" with "RESTORATION" and not "HOLDACTIVE" with "RESTORATION"

## 7.4.13.15 Monitoring Rx/Tx fill-levels and flow-control via CGR

If CGR-based monitoring is enabled, then two Congestion Group Records will be configured, with all Rx FQs for all interfaces being subscribed to one, and all Tx FQs being subscribed to the other. This simply allows the user to monitor the overall fill-level of frame queues in the system, in particular to determine whether build-up is occurring before or after the software-processing phase. Refer to section "Monitoring Rx/Tx fill-levels via CGR" of USDPAA PPAC User Guide for more details. To enable this feature, in ppac.h change;

```
#undef PPAC_CGR
to;
#define PPAC_CGR
```

The CGRs can also be configured to perform flow-control using Congestion state tail drop by setting CSTD\_EN bits. Each congestion group record can be configured to track either byte counts or frame counts in all frame queues in the Congestion Group. When the threshold set for each CGR is exceeded, the CS bit is set in the CGR, and the congestion group is said to have entered congestion. At this point the incoming frames are marked for discard and QMAN will generate enqueue rejections to the producer. When the group's I\_BCNT returns below the threshold (minus approximately 1/8 of the threshold to provide hysteresis), the CS bit is cleared, and the congestion group's state exits congestion. To enable tail drop, in ppac.h change;

```
#undef PPAC_CGR /* Track rx and tx fill-levels via CGR */
#undef PPAC_CSTD /* CGR tail-drop */
#undef PPAC_CSCN /* Log CGR state-change notifications */
to
#define PPAC_CGR /* Track rx and tx fill-levels via CGR */
#define PPAC_CSTD /* CGR tail-drop */
```

```
#undef PPAC_CSCN /* Log CGR state-change notifications */
```

And then compile usdpaa once again. Now run the IPFWD application with CGR tail drop enabled. To test this feature PPAC CLI provides a command “cgr” which will query and display all the fields of both CGRs. On pumping the traffic to IPFWD application at full line rate, the instantaneous group byte count value I\_BCNT(Instantaneous frame/byte count) must be maintained lesser than the CGR threshold set for each congestion group. Here is one such cgr command output:

```
> cgr
Rx CGR ID: 10, selected fields;
cscn_en: 0
cscn_targ: 0x00800000
cstd_en: 1
cs: 0
cs_thresh: 0x00_0000_1000
mode: 1
i_bcnt: 0x00_0000_0e1e
a_bcnt: 0x00_0000_0e1e
Tx CGR ID: 11, selected fields;
cscn_en: 0
cscn_targ: 0x00800000
cstd_en: 1
cs: 0
cs_thresh: 0x00_0000_0200
mode: 1
i_bcnt: 0x00_0000_0002
a_bcnt: 0x00_0000_0004
```

On the other hand if this feature of Congestion Group tail drop is disabled in IPFWD application I\_BCNT is never maintained below CGR threshold value with traffic at full line-rate. This can be checked by compiling the IPFWD application with;

```
#define PPAC_CGR /* Track rx and tx fill-levels via CGR */
#undef PPAC_CSTD /* CGR tail-drop */
#undef PPAC_CSCN /* Log CGR state-change notifications */
```

Now on pumping traffic at full line-rate, atleast one of the CGRs must go into congestion state and its I\_BCNT should be above CGR threshold value. Here is the sample output:

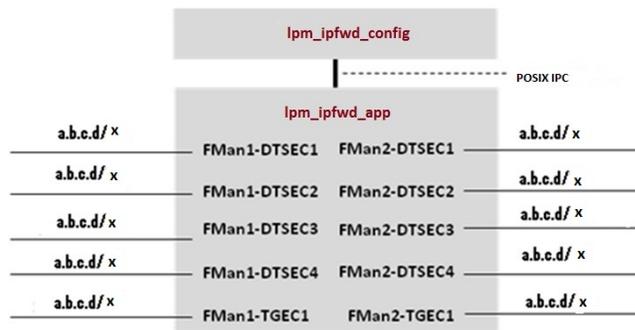
```
> cgr
Rx CGR ID: 10, selected fields;
cscn_en: 1
cscn_targ: 0x00800000
cstd_en: 0
cs: 0
cs_thresh: 0x00_0000_1000
mode: 1
```

Linux User Space  
USDPAAs Applications

i\_bcnt: 0x00\_0000\_0006  
a\_bcnt: 0x00\_0000\_0004  
Tx CGR ID: 11, selected fields;  
cscn\_en: 1  
cscn\_targ: 0x00800000  
cstd\_en: 0  
cs: 1  
cs\_thresh: 0x00\_0000\_0200  
mode: 1  
i\_bcnt: 0x00\_0000\_5fb7  
a\_bcnt: 0x00\_0000\_5fb8

Thus, the above test showcases the flow control achieved by enabling congestion group tail drop in IPFWD application.

### 7.4.13.16 LPM IPFWD Application Suite



The figure above shows the structure of the LPM IPFWD USDPAAs application suite. Its purpose is to forward IPv4 packets between the interfaces shown. No IP address is assigned to any of the interfaces by default. Using ipfwd\_config command as mentioned in section [Assign IP address to interfaces](#) on page 1056 IP address can be assigned to all these interfaces. Each interface can have a variable netmask. The MAC addresses of these interfaces are determined by u-boot environment variables ethaddr, eth1addr, eth2addr, etc.

The ipfwd application will respond to ARP requests on these interfaces. However, the application will not generate ARP requests to determine the destination MAC address of forwarded packets. An ipfwd\_config command should be used to set these MAC addresses. On host side, ARP table can be updated by sending ARP requests to this application.

It is important to understand that the application can use only a subset of these interfaces at any one time. This is due to pin multiplexing rules on the P4080 SoC. The set of available interfaces is determined by a number of factors:

1. The interface set made available by the selected SerDes Protocol and u-boot variable "hwconfig". See the SDK document "NXP DPAA SDK X.Y: Selecting Ethernet Interfaces". This document is distributed with the DPAA SDK.
2. The Linux device tree determines which subset of the available interfaces is for use by USDPAAs applications. See the USDPAAs User Guide for more information.
3. Finally, the fmc configuration file passed by command line argument to lpm\_ipfwd\_app determines the subset of these interfaces that the application will attempt to initialize.

In the default SerDes 0xe example, the interfaces used by lpm\_ipfwd\_app are:

- FMan1-TGEC1
- FMan2-DTSEC3

- FMan2-DTSEC4
- FMan2-TGEC1

Running the lpm ipfwd application suite involves four steps:

1. Run lpm\_ipfwd\_app
2. Run lpm\_ipfwd\_config repeatedly to add routes to LPM FIB table.
3. Run the fmc application to configure the FMan hardware instances.

Specific examples showing these steps are provided in other sections of this document.

### 7.4.13.1.17 Possible configuration scenario for LPM based IPFWD

LPM based IPfwd application can run in different configuration scenario. The table below shows the configuration files and corresponding sample shell script existing in the repository.

| xml file                                     | Sample shell script       | Number of routes (as per sample shell script) | Netmask (as per sample shell script) |
|----------------------------------------------|---------------------------|-----------------------------------------------|--------------------------------------|
| usdpaa_config_p4_serdes_0xe.xml (2x10G+2x1G) | lpm_ipfwd_20G.sh          | 1024 routes (2x10G)                           | 16                                   |
|                                              | lpm_ipfwd_22G.sh          | 1024 routes (2x10G+2x1G)                      | 16                                   |
|                                              | lpm_ipfwd_20G_1Mroutes.sh | 1M/256 (4K) routes                            | 24                                   |
| usdpaa_config_p2_p3_p5_14g.xml(4x1G+10G)     | lpm_ipfwd_14G.sh          | 1020 routes (4x1G+10G)                        | 16                                   |



Figure 1: usdpaa\_config\_p4\_serdes\_0xe.xml

The figure above shows the ethernet interfaces available as per configuration file “ usdpaa\_config\_p4\_serdes\_0xe.xml”. It contains the 1 RGMII port (FMAN1, DTSEC 2), 2 SGMII ports ( FMan2, DTSECs 3- 4 ) and 2 XAUI ports (FMAN1, TGEC1 and FMAN2-TGEC2). It is the user’s wish to use any combination of ports available with the configuration file. The above table shows the sample shell scripts that user can use for this configuration. If user uses 2x10G, following is the flow configuration.

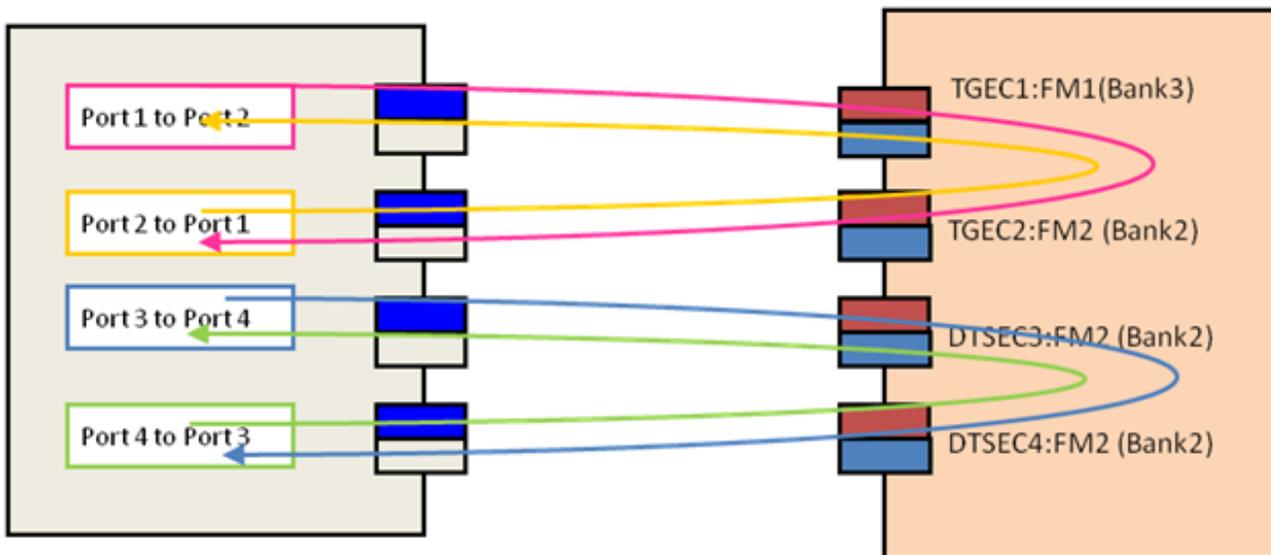
Test Center <-> P4080



Port 2 to Port 1 (512 flows) :  
 Dst : 190.0.24.2 - 190.255.24.2  
 Dst : 191.0.24.2 - 191.255.24.2  
 Gw : 192.168.60.2

Port 1 to Port 2 (512 flows) :  
 Dst : 192.0.24.2 - 192.255.24.2  
 Dst : 193.0.24.2 - 193.255.24.2  
 Gw : 192.168.160.2

Figure 2 : Flow Configuration for 2x10G on P4080DS



**Port 2 to Port 1 (256 flows):**  
**Dst : 190.0.24.2 - 190.255.24.2**  
**Gw : 192.168.60.2**

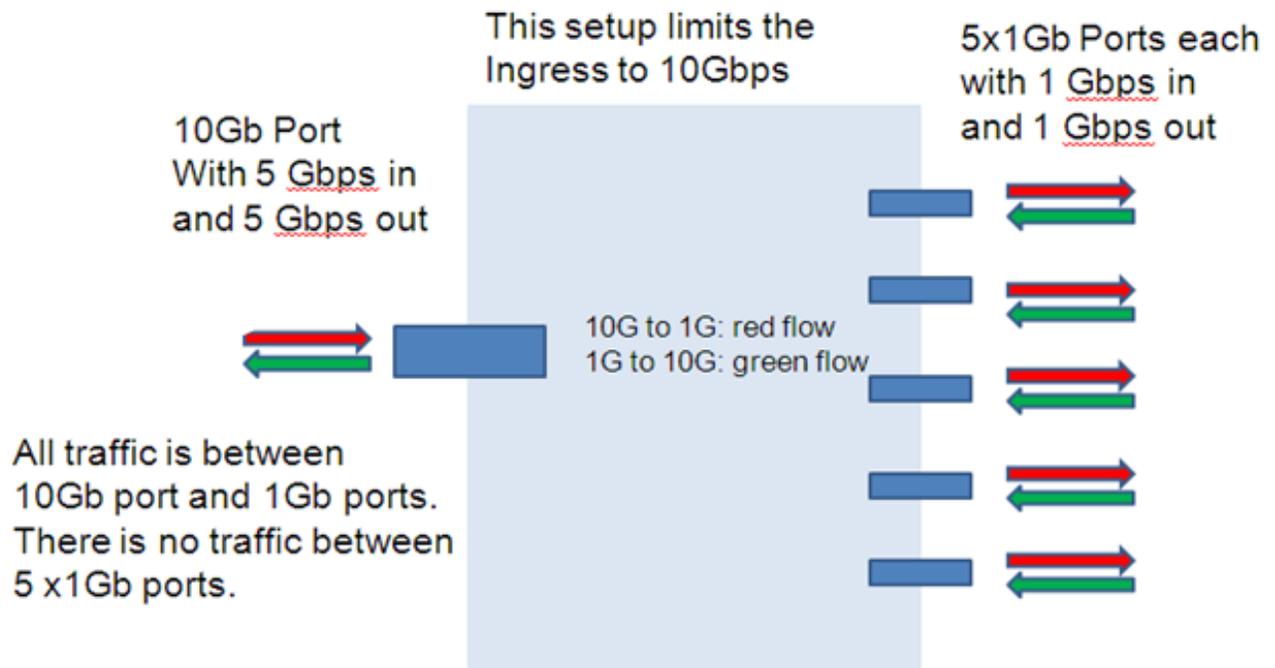
**Port 1 to Port 2 (256 flows):**  
**Dst : 193.0.24.2 - 193.255.24.2**  
**Gw : 192.168.160.2**

**Port 4 to Port 3 (256 flows):**  
**Dst : 191.0.24.2 - 191.255.24.2**  
**Gw : 192.168.130.2**

**Port 3 to Port 4 (256 flows):**  
**Dst : 192.0.24.2 - 192.255.24.2**  
**Gw : 192.168.140.2**

Figure 3 : configuration for 2x10G and 2x1G on P4080

For running LPM IPFwd on P3041DS/P5020DS, the user can use "usdpaa\_config\_p2\_p3\_p5\_14g.xml". Following is the flow configuration.



Left: 10G port (port6: fm1-10g)  
Right: top-to-bottom: 1G port (port1-port6)

Figure 4 : Flow Configuration for 10G on P5020DS and P3041DS

|                                                                                    |                                                                                    |
|------------------------------------------------------------------------------------|------------------------------------------------------------------------------------|
| Port 1 to Port 6 (100 flows):<br>Dst: 195.0.24.2 – 195.99.24.2<br>Gw: 192.168.60.2 | Port 6 to Port 1 (100 flows):<br>Dst: 190.0.24.2 – 190.99.24.2<br>Gw: 192.168.10.2 |
| Port 2 to Port 6 (100 flows):<br>Dst: 196.0.24.2 – 196.99.24.2<br>Gw: 192.168.60.2 | Port 6 to Port 2 (100 flows):<br>Dst: 191.0.24.2 – 191.99.24.2<br>Gw: 192.168.20.2 |
| Port 3 to Port 6 (100 flows):<br>Dst: 197.0.24.2 – 197.99.24.2<br>Gw: 192.168.60.2 | Port 6 to Port 3 (100 flows):<br>Dst: 192.0.24.2 – 192.99.24.2<br>Gw: 192.168.30.2 |
| Port 4 to Port 6 (100 flows):<br>Dst: 198.0.24.2 – 198.99.24.2<br>Gw: 192.168.60.2 | Port 6 to Port 4 (100 flows):<br>Dst: 193.0.24.2 – 193.99.24.2<br>Gw: 192.168.40.2 |
| Port 5 to Port 6 (100 flows):<br>Dst: 199.0.24.2 – 199.99.24.2<br>Gw: 192.168.60.2 | Port 6 to Port 5 (100 flows):<br>Dst: 194.0.24.2 – 194.99.24.2<br>Gw: 192.168.50.2 |

#### How to run if user has no XAUI but only SGMII riser card?

Assume user does not have a XAUI riser card and wants to run LPM IPFwd using only the SGMII ports as shown below.



This configuration contains the 1 RGMII port (FMAN1, DTSEC 2), 4 SGMII ports ( FMan2, DTSECs 1- 4 ). The user can modify the configuration file and sample shell scripts as per requirement. The configuration file would contain the following

```
<cfgdata>
<config>
<engine name="fm1">
<port type="1G" number="0" policy="hash_ipsec_src_dst_spi_policy6"/>
<port type="1G" number="1" policy="hash_ipsec_src_dst_spi_policy7"/>
<port type="1G" number="2" policy="hash_ipsec_src_dst_spi_policy8"/>
<port type="1G" number="3" policy="hash_ipsec_src_dst_spi_policy9"/>
</engine>
</config>
</cfgdata>
```

User can create a new shell script which would make routes between the 4x1G. Here is the content of the script.

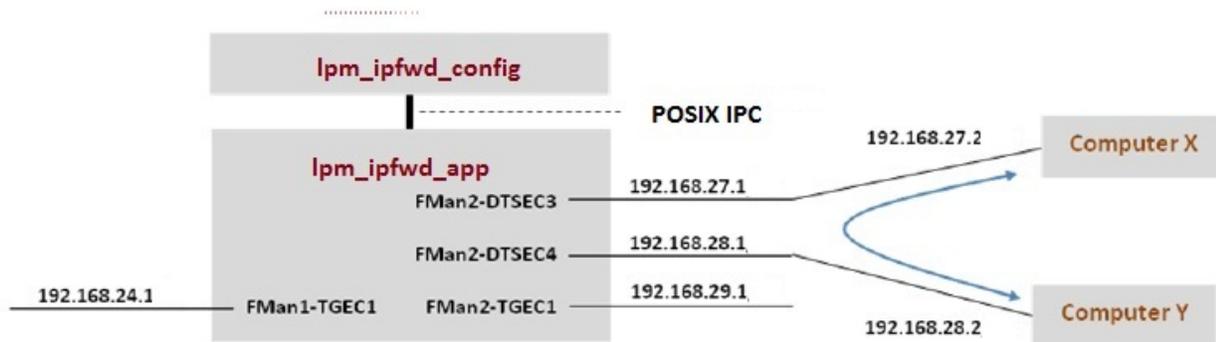
```
net_pair_routes()
{
net=0
while ["$net" -le $5]
do
lpm_ipfwd_config -P $pid -B -c $2 -d $1.$net.24.2 -n $3 -g \
192.168.$4.2
net=`expr $net + 1`
done
}
lpm_ipfwd_config -P $pid -F -a 192.168.110.1 -i 6
lpm_ipfwd_config -P $pid -F -a 192.168.120.1 -i 7
```

```
lpm_ipfwd_config -P $pid -F -a 192.168.130.1 -i 8
lpm_ipfwd_config -P $pid -F -a 192.168.140.1 -i 9
lpm_ipfwd_config -P $pid -G -s 192.168.110.2 -m 02:00:c0:a8:6e:02 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.120.2 -m 02:00:c0:a8:78:02 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.130.2 -m 02:00:c0:a8:82:02 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.140.2 -m 02:00:c0:a8:8c:02 -r true
1024

net_pair_routes 190 1 16 110 255 # 256
net_pair_routes 191 1 16 120 255 # 256
net_pair_routes 192 1 16 130 255 # 256
net_pair_routes 193 1 16 140 255 # 256
```

### 7.4.13.18 Using Two Computers to Test the IPFWD Application Suite

This section describes how to configure the IPFWD application suite to forward packets between two ordinary computers as shown in the following figure. It is assumed that the two 1 Gbps Ethernet interfaces provided by SerDes 0xe are used.



Assign IP addresses to all the interfaces. The IP addresses used here for Computer X and Computer Y are examples. Their MAC addresses must be known as they will be needed in the commands below.

Keep in mind that `ipfwd_app` has no default IP address assigned to the interfaces. This means that you must first assign IP addresses to the interfaces and then use IP addresses for Computer X and Computer Y that are on the subnets shown in the figure. Please be careful with the network parameters. Any mistake will prevent packets from flowing from end to end.

Follow these steps on Computer X:

1. Set the ip address for eth0 interface

```
ifconfig eth0 192.168.27.2 up
```

Set the default gateway for subnet 192.168.27.1

```
route add default gw 192.168.27.1 eth0
```

2. No need to add arp on host side. The application can handle external arp requests

Follow these steps on Computer Y:

1. Set the ip address for eth0 interface

```
ifconfig eth0 192.168.28.2 up
```

Set the default gateway for subnet 192.168.28.1

```
route add default gw 192.168.28.1 eth0
```

2. No need to add arp on host side. The application can handle external arp requests

The commands to perform on P4080 are:

# Boot the P4080 and login as root.

*Assign IP address to fm1-gb1*

ifconfig fm1-gb1 <IPADD> up

# Assume use of cores 1 - 7 lpm\_ipfwd\_app 1..7 -d 0x10000000 -b 0:0:1728 -i fm1-10g,fm2-gb2,fm2-gb3,fm2-10g

# Now ssh to P4080 linux on other terminal

ssh root@<IPADD>

give IP address as assigned to fm1-gb1 in the beginning

# Now assign ip address to the interfaces

# First run command to check what all enabled interfaces are available

Note: check pid from application print "Message queue to send: /mq\_snd\_2536"

lpm\_ipfwd\_config -P 2536 -E -a true

Interface number: 11

PortID=1:5 is FMan interface node

with MAC Address

02:00:c0:a8:65:fe

Interface number: 9

PortID=1:3 is FMan interface node

with MAC Address

02:00:c0:a8:5b:fe

Interface number: 8

PortID=1:2 is FMan interface node

with MAC Address

02:00:c0:a8:51:fe

Interface number: 5

PortID=0:5 is FMan interface node

with MAC Address

02:00:c0:a8:33:fe

Are all the Enabled Interfaces

# Assign IP address to the interfaces. Use the same interface number displayed as an output on giving the above command

lpm\_ipfwd\_config -P 2536 -F -a 192.168.24.1 -i 5

lpm\_ipfwd\_config -P 2536 -F -a 192.168.28.1 -i 9

lpm\_ipfwd\_config -P 2536 -F -a 192.168.27.1 -i 8

lpm\_ipfwd\_config -P 2536 -F -a 192.168.29.1 -i 11

# Now enter routes and MAC addresses. Format of a MAC address is

# aa:bb:cc:dd:ee:ff where the letters are hexadecimal digits.

```
lpm_ipfwd_config -P 2536 -B -d 192.168.28.2 -g 192.168.28.2 -n 24 -c 1
```

```
lpm_ipfwd_config -P 2536 -G -s 192.168.28.2 -m COMPUTER_Y_MAC_ADDRESS -r true
```

```
lpm_ipfwd_config -P 2536 -B -d 192.168.27.2 -g 192.168.27.2 -n 24 -c 1
```

```
lpm_ipfwd_config -P 2536 -G -s 192.168.27.2 -m COMPUTER_X_MAC_ADDRESS -r true
```

```
Run fmc command cd /usr/etc
```

```
fmc -c usdpaa_config_p4_serdes_0xe.xml -p usdpaa_policy_hash_lpm_ipv4.xml -a
```

```
Now, traffic can pass between Computer X and Computer Y. For example, on Computer X
```

```
enter:
```

```
ping 192.168.28.2
```

### 7.4.13.19 Running LPM IPv4 forwarding on P4080DS board

The instructions below describe how to run LPM-IPFWD. Traffic should only be directed to the P4080DS once the application is running and configuration via the ipfwd\_config application is completed.

- On linux prompt, assign IP address to fm1-gb1

```
$ ifconfig fm1-gb1 <IPADD> up
```

Now run the FMC command

```
$ cd /usr/etc
```

*To setup the FMan to distribute traffic to 32 ingress frame queues per port:*

```
$ fmc -c usdpaa_config_p4_serdes_0xe.xml -p usdpaa_policy_hash_lpm_ipv4.xml -a
```

- Run LPM-IPFWD application

*The main LPM-IPFwd application binary is called **lpm\_ipfwd\_app**. The application can run on multiple cores as specified by the first parameter to the application. To run it to handle traffic distributed over 32 ingress frame queues per port:*

```
$ lpm_ipfwd_app <m..n>
```

By default lpm\_ipfwd\_app uses usdpaa\_config\_p4\_serdes\_0xe.xml and usdpaa\_policy\_hash\_lpm\_ipv4.xml files.

#### **LPM-IPFWD application command syntax:**

```
[root@p4080 etc]# lpm_ipfwd_app --usage
```

```
Usage: lpm_ipfwd_app [-n?V] [-b x:y:z] [-c FILE] [-d SIZE] [-i FILE] [-p FILE]
```

```
[-buffers=x:y:z] [--fm-config=FILE] [--dma-mem=SIZE]
```

```
[-fm-interfaces=FILE] [--non-interactive] [--fm-pcd=FILE]
```

```
 [--cpu-range] [--help] [--usage] [--version] [cpu-range]
```

LPM-IPFWD application run command:

```
[root@p4080 root]# cd /usr/etc
```

```
[root@p4080 etc]# lpm_ipfwd_app 1..7 -d 0x4000000 -b 0:0:1728 -i fm1-10g,fm2-gb2,fm2-gb3,fm2-10g
```

To use dma region size as 64M in USDPAAs for lpm-ipfwd application, make sure to set the same or greater size in bootargs.  
E.g. usdpaa\_mem=256M

```
[1] 5363
```

```
[root@p4080 etc]# Found /fsl,dpaa/dpa-fman0-oh@1, Tx Channel = 46, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@4, Tx Channel = 40, FMAN = 0, Port ID = 0
Found /fsl,dpaa/ethernet@7, Tx Channel = 63, FMAN = 1, Port ID = 2
Found /fsl,dpaa/ethernet@8, Tx Channel = 64, FMAN = 1, Port ID = 3
Found /fsl,dpaa/ethernet@9, Tx Channel = 60, FMAN = 1, Port ID = 0
Configuring for 4 network interfaces
Allocated DMA region size 0x10000000
lpm_ipfwd_app starting
IPv4 FIB table init now... Done!
Message queue to send: /mq_snd_2536
Message queue to receive: /mq_rcv_2536
Thread uid:0 alive (on cpu 1)
Release 0 bufs to BPID 7
Release 0 bufs to BPID 8
Release 1600 bufs to BPID 9

Thread uid:1 alive (on cpu 2)
Thread uid:2 alive (on cpu 3)
Thread uid:3 alive (on cpu 4)
Thread uid:4 alive (on cpu 5)
Thread uid:5 alive (on cpu 6)
Thread uid:6 alive (on cpu 7)
```

If, in the run application command, cpu-range is given i.e. "lpm\_ipfwd\_app <m..n>" LPM-IPFWD application starts threads on cpu-range m..n. The main thread (by default on CPU 1) then does global initialization needed by the application, including starting other application threads.

If, on the other hand, run application command is given without any cpu-range i.e. "lpm\_ipfwd\_app" LPM-IPFWD application starts up with a single thread running on CPU 1 by default, which does global initialization needed by the application and enables all the network interfaces.

The CLI (Command-Line Interface) allows you to add and remove additional threads to enable the use of multiple CPUs, with the only restriction being that the primary thread on CPU 1 cannot be removed (except by shutting down the application).

To add a thread on a single CPU (e.g. CPU 2):

```
> add 2
```

To add threads on a range of CPUs:

```
> add 3..6
```

To list the threads (this also queries each thread, verifying that they aren't blocked):

```
> list
```

```
Thread alive on cpu 1
Thread alive on cpu 2
Thread alive on cpu 3
Thread alive on cpu 4
```

Thread alive on cpu 5

Thread alive on cpu 6

To enable all interfaces

```
> macs on
```

To disable all interfaces

```
> macs off
```

To perform a controlled shutdown of ipfwd (this includes disabling the network ports):

```
> quit
```

- Once the application starts, it can receive the configuration commands. Run application configuration script.

For creating route entries, the binary `lpm_ipfwd_config` is run. This binary processes the configuration request from the user (using the command line) and populates the configuration via Linux standard posix IPC messages to the LPM-IPFwd application.

The shell script mentioned below contains sample commands to add route entries. Detailed description of all `lpm_ipfwd_config` commands is provided in section [Syntax](#) on page 936.

SSH to p4080 linux on another terminal:

```
$ ssh root@<IPADD> (give the IP address as assigned to fm1-gb1 in the beginning)
```

*Run the shell script:*

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq\_snd\_2536 "

```
lpm_ipfwd_20G.sh "pid"
```

```
$ lpm_ipfwd_20G.sh 2536
```

There are example shell scripts available. They can assign IP addresses to the interfaces, add an ARP entry and can use variable netmask while creating route entries. The following table summarizes the settings done by these scripts. Check section [Possible configuration scenario for LPM based IPFWD](#) on page 1043 for more details.

For the LPM-IPFwd application to forward traffic successfully, traffic destined for the P4080DS ports must have the appropriate destination IP addresses.

Console messages are printed for each entry added to the routing table. Once all configuration is completed, the application moves to the packet processing phase - the message "LPM-IPFwd Route Creation completed" is printed on the console.

At this point, traffic can be sent to the ethernet interfaces, IPv4 packets would be processed by the LPM application on the cpu-range specified by the user on the application command-line.

### 7.4.13.1.20 Running LPM IPv4 forwarding on P3041/P5020 board

The instructions below describe how to run LPM-IPFWD on P3041/P5020. Traffic should only be directed to P3041/P5020 once the application is running and configuration via the `lpm_ipfwd_config` application is completed. On linux prompt, assign IP address to eth0

```
$ ifconfig eth0 <IPADD> up
```

Configure FMan PCD using fmc with the XML files in /usr/etc:

```
$ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
$ fmc -c usdpaa_config_p2_p3_p5_14g.xml -p usdpaa_policy_hash_lpm_ipv4.xml -a
```

Run lpm-IPFWD

```
$ lpm_ipfwd_app <m..n> -c usdpaa_config_p2_p3_p5_14g.xml -p usdpaa_policy_hash_lpm_ipv4.xml -d 0x4000000 -b 0:0:1728 -i fm1-gb0, fm1-gb1, fm1-gb3, fm1-gb4, fm1-10g
```

For P3041, m..n can be 0..3. For P5020, m..n can be 0..1.

SSH to board (linux) on another terminal:

```
$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)
```

Run the shell script:

```
$ lpm_ipfwd_14G.sh
```

There is an example shell script available named as lpm\_ipfwd\_14G.sh creates routes for only the 4 x 1G and 1x10G interfaces. It can assign ip addresses to the interfaces, add an ARP entry . Check section [Possible configuration scenario for LPM based IPFWD](#) on page 1043 for more details. Now traffic can be run as per the routes created.

### 7.4.13.1.21 USDPAA LPM IP Fwd performance gap between 6 core and 8 core

USDPAA LPM IPfwd performance for 8 core is less than 6 core on e6500 series. USDPAA LPM IP Fwd application need to run using "-s" option to bridge the gap between two configuration.

### 7.4.13.1.22 PPAC (and IPFwd) CLI commands

The following commands are illustrated in the context of IPFwd, but the commands are common to all PPAC-based applications.

To add a thread on a single CPU (e.g. CPU 2):

```
> add 2
```

To add threads on a range of CPUs:

```
> add 3..6
```

To list the threads (this also queries each thread, verifying that they aren't blocked):

```
> list
```

```
Thread uid:0 alive (on cpu 1)
```

```
Thread uid:1 alive (on cpu 2)
```

```
Thread uid:2 alive (on cpu 3)
```

```
Thread uid:3 alive (on cpu 4)
```

```
Thread uid:4 alive (on cpu 5)
```

```
Thread uid:5 alive (on cpu 6)
```

```
Thread uid:6 alive (on cpu 7)
```

To remove a thread by its UID:

```
> rm uid:2
```

```
Thread uid:2 killed (cpu 3)
```

To remove a thread running on a given CPU:

```
> rm 5 Thread uid:4 killed (cpu 5)
```

To enable all interfaces:

```
> macs on
```

To disable all interfaces:

> macs off

To perform a controlled shutdown of ipfwd (this includes disabling the network ports):

> quit

To query cgr

> cgr

Rx CGR ID: 10, selected fields;

cscn\_en: 0

cscn\_targ: 0x00800000

cstd\_en: 1

cs: 0

cs\_thresh: 0x00\_0000\_1000

mode: 1

i\_bcnc: 0x00\_0000\_0e1e

a\_bcnc: 0x00\_0000\_0e1e

Tx CGR ID: 11, selected fields;

cscn\_en: 0

cscn\_targ: 0x00800000

cstd\_en: 1

cs: 0

cs\_thresh: 0x00\_0000\_0200

mode: 1

i\_bcnc: 0x00\_0000\_0002

a\_bcnc: 0x00\_0000\_0004

### 7.4.13.1.23 Syntax

The syntax is as follows:

```
[$ [root@p4080 bin]# ipfwd_config --help
```

Usage: ipfwd\_config [OPTION...]

- B, --routeadd=TYPE adding a route
- C, --routedel=TYPE deleting a route
- E, --showintf=TYPE show interfaces
- F, --intfconf=TYPE change intf config
- G, --arpadd=TYPE adding a arp entry
- H, --arpdel=TYPE deleting a arp entry
- , --help Give this help list
- usage Give a short usage message
- V, --version Print program version

### 74.13.1.24 Command to show all enabled interfaces and their interface numbers

The command to show all the enabled interfaces while running IPv4 forward is as follows:

```
lpfwd_config -P pid -E -a true
```

**Table 163. Field Description (show all enabled interfaces)**

Parameter	Description	Mandatory	Format/ Value
-a	Show all the interfaces	Yes	true

Command to show all enabled interfaces

Note: check pid from application print "Message queue to send: /mq\_snd\_2536"

```
[root@p4080 bin]# lpfwd_config -P 2536 -E -a true
```

Interface number: 11

PortID=1:5 is FMan interface node

with MAC Address

02:00:c0:a8:65:fe

Interface number: 9

PortID=1:3 is FMan interface node

with MAC Address

02:00:c0:a8:5b:fe

Interface number: 8

PortID=1:2 is FMan interface node

with MAC Address

02:00:c0:a8:51:fe

Interface number: 5

PortID=0:5 is FMan interface node

with MAC Address

02:00:c0:a8:33:fe

Are all the Enabled Interfaces [root@p4080 bin]#

### 74.13.1.25 Help for show all enabled interfaces command

The command to obtain help for show all enabled interfaces command is as follows:

```
lpm_ipfwd_config -E -help
```

Help for show all enabled interfaces

```
[root@p4080 etc]# lpm_ipfwd_config -E --help
```

Usage: -E [OPTION...]

-a, --a=ALL All interfaces

- ?, --help Give this help list
- usage Give a short usage message
- V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

### 7.4.13.1.26 Assign IP address to interfaces

The command to assign IP address to shared or private interfaces while running IPv4 forward is as follows:

```
lpm_ipfwd_config -P pid -F -a 192.168.60.1 -i <Interface number>
```

The command to assign IP address to MAC-less interfaces while running IPv4 forward is as follows:

```
lpm_ipfwd_config -P pid -F -a 192.168.60.1 -n <MAC-less interface name>
```

**NOTE**

Note: The interface name(MAC-less) or interface number to be used here must be one of the names/numbers that got displayed as the output of "show all enabled interfaces command" in section [Command to show all enabled interfaces and their interface numbers](#) on page 937.

**Table 164. Field description (assign IP address to shared or private MAC interfaces)**

Parameter	Description	Mandatory	Format/ Value
-a	IP Address	Yes	a.b.c.d
-i	Interface number	Yes	0-11  (Choose this number from "show all enabled interfaces" command output)

**Table 165. Field description (assign IP address to MAC-less interfaces)**

Parameter	Description	Mandatory	Format/ Value
-a	IP Address	Yes	a.b.c.d
-n	MAC-less Interface name	Yes	eth3  (Choose this name in case of MAC-less from "show all enabled interfaces" command output)

Assign IP address to interfaces

Note: check pid from application print "Message queue to send: /mq\_snd\_2536"

. Command to assign IP address to private or shared MAC interfaces

```
lpm_ipfwd_config -P 2536 -F -a 192.168.60.1 -i 5
```

IPADDR assigned = 0xc0a83c01 to interface num 5

Intf Configuration Changed successfully

. Command to assign IP address to MAC-less interface

```
lpm_ipfwd_config -P 2536 -F -a 192.168.55.6 -n eth3
```

IPADDR assigned = 0xc0a88506 to MACLESS intf eth3

Intf Configuration Changed successfully

### 7.4.13.1.27 Help for assign IP address to interfaces

The command to obtain help for assign IP address to interfaces command is as follows:

```
lpm_ipfwd_config -F --help
```

Help for assign IP address to interfaces

```
[root@p4080 etc]# lpm_ipfwd_config -F --help
```

Usage: -F [OPTION...]

-a, --a=IPADDR IP Address

-i, --i=IFNUM If Number

-n, --n=IFNAME MACLESS Interface Name

-, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

### 7.4.13.1.28 Adding a Route Entry

The command to add a route while running IPv4 forward is as follows:

```
lpm_ipfwd_config -P pid -B -d b.c.d.e -g c.d.e.f -n maskbits -c numentry
```

**Table 166. Field Description (Adding a Route Entry)**

Parameter	Description	Mandatory	Format/ Value
-d	Destination IP Address	Yes	a.b.c.d
-g	Gateway IP Address	Yes	a.b.c.d
-n	netmask length to be used by LPM	Yes	upto 32 bits
-c	number of fib entries	Yes	1- valid count

Adding a Route Entry

Note: check pid from application print "Message queue to send: /mq\_snd\_2536"

```
[root@p4080 bin]# lpm_ipfwd_config -P 2536 -B -d 192.168.24.2 -g 192.168.24.2 -n 24 -c 1024
```

Route Entry Added successfully

```
[root@p4080 bin]#
```

### 7.4.13.129 Help for Route Entry Addition

The command to obtain help for route entry addition is as follows:

```
lpm_ipfwd_config -B --help
```

Help for Adding a Route Entry

```
[root@p4080 bin]# lpm_ipfwd_config -B --help
```

Usage: -B [OPTION]

-d, --d=DESTIP Destination IP

-g, --g=GWIP Gateway IP

-n, --n=MASKBITS Netmask length

-c, --c=NUMENTRY Number of fib entries

-, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

### 7.4.13.130 Deleting a Route Entry

NOT IMPLEMENTED

### 7.4.13.131 Help for Deleting a Route Entry

Delete route command is not implemented

### 7.4.13.132 Adding an ARP Entry

The command to add an ARP entry while running IPv4 forward is as follows:

```
lpm_ipfwd_config -P pid -G -s a.b.c.d -m aa:bb:cc:dd:ee [-r true]
```

**Table 167. Field Description (Adding an ARP Entry)**

Parameter	Description	Mandatory	Format/ Value
-s	Gateway IP Address	Yes	a.b.c.d
-m	Mac Address	Yes	aa:bb:cc:dd:ee
-r	Replace existing entry	No	true/ false {Default: false}

Adding an ARP Entry

Note: check pid from application print "Message queue to send: /mq\_snd\_2536"

```
[root@p4080 bin]# lpm_ipfwd_config -P 2536 -G -s 192.168.24.2 -m 02:00:c0:a8:33:fd -r true
```

ARP Entry Added successfully

### 7.4.13.133 Help for ARP Entry Addition

The command to obtain help for ARP entry addition is as follows:

```
lpm-ipfwd_config -G --help
```

Help for Adding an ARP Entry

```
[root@p4080 etc]# lpm_ipfwd_config -G --help
```

Usage: -G [OPTION...]

-m, --m=MACADDR MAC Address

-r, --r=Replace Replace Existing Entry - true/ false {Default: false}

-s, --s=IPADDR IP Address

-, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

### 7.4.13.134 Deleting an ARP Entry

The command to delete an ARP while running the IPv4 forward is as follows:

```
lpm_ipfwd_config -P pid -H -s a.b.c.d
```

**Table 168. Field Description (Deleting an ARP Entry)**

Parameter	Description	Mandatory	Format/ Value
-s	Gateway IP Address	Yes	a.b.c.d

Deleting an ARP Entry

Note: check pid from application print "Message queue to send: /mq\_snd\_2536"

```
[root@p4080 bin]# lpm_ipfwd_config -P 2536 -H -s 192.168.24.2
```

Arp Entry Deleted successfully

```
[root@p4080 bin]#
```

### 7.4.13.135 Help for Deleting an ARP Entry

The command to obtain help for ARP entry deletion is as follows:

```
lpm_ipfwd_config -H --help
```

Help for Deleting an ARP Entry

```
[root@p4080 etc]# lpm_ipfwd_config -H --help
```

Usage: -H [OPTION...]

-s, --s=IPADDR IP Address

-, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

## 7.4.13.136 References

1. USDPAA PPAC User Guide
2. QMan/BMan API Guide

## 7.4.14 NXP USDPAA FRA Configuration User Manual

### 7.4.14.1 Introduction

#### 7.4.14.1.1 Purpose

This document is intended to introduce how to configure the NXP RMan Application (FRA).

#### 7.4.14.2 FRA Configuration

The NXP RMan Application is a multi-thread USDPAA application based on RMan driver. The RapidIO message manager (RMan) supports a message passing programming model for inter-processor and inter-device communication. The FRA configuration file is to configure the behavior of passing messages. This file follows standard XML rules and is composed of several elements. Each element begins with a start tag and can contain attributes and/or child elements. If the element contains child elements, it must have a corresponding end-tag after them. An element without child elements must end with a slash (/).

Note that element and attribute names are always case sensitive.

The FRA configuration file always begins with the <fra\_cfg> root element. As such, the end-tag of the fra\_cfg element must appear at the end of the file.

Attributes: No required attributes

For example:

```
<fra_cfg>
 <rman_cfg>
 <defcfg file="/usr/etc/rman_config.xml"/>
 </rman_cfg>

 <network_cfg>
 <defcfg file="/usr/etc/network_config.xml"/>
 </network_cfg>

 <trans_cfg>
 <defcfg file="/usr/etc/transactions_config.xml"/>
 </trans_cfg>

 <dists_cfg>
 <defcfg file="/usr/etc/distributions_config.xml"/>
 </dists_cfg>

 <policies_cfg>
 <defcfg file="/usr/etc/policies_config.xml"/>
 <policy name="processing2" enable="yes"/>
 </policies_cfg>
</fra_cfg>
```

```
</policies_cfg>
</fra_cfg>
```

This configuration file contains the following top elements:

- rman\_cfg
- network\_cfg
- trans\_cfg
- dists\_cfg
- policies\_cfg

Each top element mentioned above may includes 'defcfg' element to include the default configuration file. Following 'defcfg', the different settings can be included to change the default value.

For example, in order to use sRIO port 2 to send messages, sRIO port number should be changed from default value 0 to 1. The configuration is as follows:

```
<dists_cfg>
 <defcfg file="/usr/etc/distributions_config.xml"/>

 <distribution name="fman_to_rman_dtsec0">
 <rio_port number="1"/>
 </distribution>
</dists_cfg>
```

### 7.4.14.2.1 Rman\_cfg Element

This element is responsible for RMan general settings. It contains some child elements: fqbits, md\_create, osid, bpid and sgbpid.

For example:

```
<rman_cfg>
 <fqbits type="Data-streaming" value="2"/>
 <fqbits type="Mailbox" value="2"/>
 <md_create mode="yes"/>

 <osid value="no"/>
 <bpid type="Data-streaming" value="11"/>
 <bpid type="Doorbell" value="10"/>
 <bpid type="Mailbox" value="11"/>
 <sgbpid value="12"/>
</rman_cfg>
```

#### OSID Element

Outbound Segmentation Interleaving Disable. Segmentation interleaving allows the message manager to interleave segments from two or more transactions with the same destination device ID (regardless of type). This may increase performance for transmission but may also consume additional reassembly resources at the destination.

A non-NXP target device may not complete Type 9 reassembly correctly when segmentation interleaving is performed on two PDUs with the same flow but different CoS. Workaround: disable segmentation interleaving by setting <osid value="yes">

Attributes:

value - (required) string, the valid value is "yes" or "no". "yes" means disable interleaving, "no" means enable interleaving.

#### EFQ Element

EFQ element is used to define error frame queue status. If enable EFQ, all the error information will be enqueued to a error frame queue.

Attributes:

value - (required) string, the valid value is "yes" or "no".

### 7.4.14.2.1.1 Fqbits Element

RMan supports algorithmic frame queue generation mode. This mode allows for steering transaction to a frame queue ID based on the RapidIO header attributes when wildcards are used with a classification rule. For The detailed information please refer to relevant manual. The current version of FRA application only support using ltr attribute of mailbox or stream ID attribute of data streaming transaction to dynamically generate the frame queue ID. The fqbits element indicates the number of rule mask bits to include in the frame queue ID.

Attributes:

value - (required) string; Defines the number of rule mask bits to include in the frame queue ID. For mailbox type 10 the valid value is 1 or 2. For data streaming is 1, 2, 3 or 4.

type - (required) string; Defines the transaction type, the valid value are 9 or 11.

### 7.4.14.2.1.2 Md\_create Element

Md\_create element is used to enable or disable RMan to write the inbound message descriptor. A performance improvement may be achieved by not writing the message descriptor.

Attributes:

mode - (required) string, the valid value is "yes" or "no". "yes" means do write the message descriptor, "no" means do not write the message descriptor.

### 7.4.14.2.1.3 BPID Element

BPID element is used to specify the buffer pool ID which RMan uses to store received transaction messages.

Attributes:

value - (required) string; Defines the buffer pool ID.

type - (required) string; Defines the transaction type, the valid value are 9-11.

### 7.4.14.2.1.4 SGBPID Element

SGBPID element is used to define the buffer pool ID which RMan uses to store the scatter/gather frame header.

Attributes:

value - (required) string; Defines the buffer pool ID.

## 7.4.14.2.2 Network\_cfg Element

This element describes FMan ports that FRA will use. It has one child element: port

### 7.4.14.2.2.1 Port Element

Port Element is used to represent a FMan port.

Attributes:

name - (required) string; Defines the port name

type - (required) string; must be "MAC"

fm - (required) string: Define the FMan engine name, valid value is "0" or "1"

number - (required) string: Define the MAC index

### 7.4.14.2.3 Transaction Element

This element describes the RapidIO type 9/10/11 transaction information. Each transaction must be defined separately within its own elements. Each transaction is used for outbound and inbound operations.

Attributes:

name - (required) string; Defines a unique name for each transaction.

type - (required) string; Defines the transaction type, the valid value are Doorbell, Mailbox, Data-streaming.

#### 7.4.14.2.3.1 Doorbell (type10) Transaction

A type10 outbound doorbell operation enables a producer to send a small amount of software-defined information across the interconnect fabric to a consumer's doorbell unit. It is the responsibility of the processor receiving the doorbell to determine the action to undertake. A small data payload of 2 bytes is used for doorbells.

This transaction has 1 child element: flowlvl

##### 7.4.14.2.3.1.1 Flowlvl Element (type 10)

This element defines transaction flow level.

Attributes:

value - (required) string; Defines transaction flow level. 0: lowest flow level, 5: highest flow level

mask - (required) string; Defines flow conditional match.

0 Exact match on flow level

1 Match on less than or equal to flow level value

2 Match on greater than or equal to flow level value

3 Reserved

For example:

```
<transaction name="dbell-peer" type="Doorbell">
 <flowlvl value="5" mask="1"/>
</transaction>
```

#### 7.4.14.2.3.2 Mailbox (type11) Transaction

A Type11 outbound RMan enables a producer to send a message across the interconnect fabric to a consumer's message hardware, called a mailbox. The receiving mailbox hardware places a message descriptor onto an inbound frame queue. A message may consist of one to sixteen segments. Messages can be queued for transmission in the producer's memory and the message hardware then processes them sequentially. Messages can also be queued in the consumer's memory while software processes them sequentially. The depths of the queues at the producer and consumer are configurable by software.

This transaction has 3 child elements: flowlvl, mbox, ltr, msglen.

##### 7.4.14.2.3.2.1 Flowlvl Element (type 11)

This element defines transaction flow level.

Attributes:

value - (required) string; Defines transaction flow level. 0: lowest flow level, 5: highest flow level

mask - (required) string; Defines flow conditional match.

0 Exact match on flow level

- 1 Match on less than or equal to flow level value
- 2 Match on greater than or equal to flow level value
- 3 Reserved

#### 7.4.14.2.3.2.2 Mbox Element (type 11)

This element defines mailbox field from packet header of mailbox transaction.

Attributes:

value - (required) string; Defines mailbox field, the valid value are 0,1,2,3

mask - (required) string; Defines bit mask for this field. Each bit set indicates don't care value for the inbound port-write header attribute.

#### 7.4.14.2.3.2.3 Ltr Element (type 11)

This element defines letter field from packet header of mailbox transaction. This field allows a sending processing element to concurrently send up to four messages to the same mailbox on the same processing element.

Attributes:

value - (required) string; Defines letter field. The valid value is 0-3.

mask - (required) string; Defines bit mask for this field. Each bit set indicates don't care value for the inbound port-write header attribute.

#### 7.4.14.2.3.2.4 Msglen Element (type 11)

This element defines message length field from packet header of mailbox transaction.

Attributes:

value - (required) string; Defines message length, and represents total number of packets comprising this message operation. The valid value is 0-15. A value of 0 indicates a single-packet message. A value of 15 (0xF) indicates a 16-packet message, etc.

mask - (required) string; Defines message length conditional match. Each bit set indicates don't care value for the inbound port-write header attribute.

- 0 Exact match on message length
- 1 Match on less than or equal to message length
- 2 Match on greater than or equal to message length
- 3 Reserved

For example:

```
<transaction name="mbox-10gec" type="Mailbox">
 <flowlvl value="0" mask="2"/>
 <mbox value="1" mask="0"/>
 <ltr value="0" mask="0" />
 <msglen value="6" mask="1"/>
</transaction>
```

### 7.4.14.2.3.3 Data streaming (type9) Transaction

A Type9 outbound segmentation operation enables a producer to send a packetized data unit (PDU) across the interconnect fabric to a consumer's reassembly hardware. The receiving reassembly hardware places the PDU descriptor onto an inbound frame queue. A PDU may consist of multiple segments supporting a total size of 64 Kbytes.

This transaction has 3 child elements: flowlvl, cos, streamid

#### 7.4.14.2.3.3.1 Flowlvl Element (type 9)

This element defines transaction flow level.

Attributes:

value - (required) string; Defines transaction flow level. 0: lowest flow level, 5: highest flow level

mask - (required) string; Defines flow conditional match.

0 Exact match on flow level

1 Match on less than or equal to flow level value

2 Match on greater than or equal to flow level value

3 Reserved

#### 7.4.14.2.3.3.2 CoS Element (type 9):

This element defines class-of-service field from the start or single packet header of data streaming transaction.

Attributes:

value - (required) string; Defines cos field, the valid value are 0-0xff

mask - (required) string; Defines bit mask for this field. Each bit set indicates don't care value for the inbound port-write header attribute.

#### 7.4.14.2.3.3.3 Streamid Element (type 9):

This element defines traffic stream identifier field from packet header of data streaming transaction. This is an end to end (producer to consumer) traffic stream identifier.

Attributes:

value - (required) string; Defines streamid field, the valid values is 0-0xffff

mask - (required) string; Defines bit mask for this field. Each bit set indicates don't care value for the inbound port-write header attribute.

For example:

```
<transaction name="dstr-10gec" type="Data-streaming">
 <flowlvl value="0" mask="2"/>
 <cos value="15" mask="0"/>
 <streamid value="0" mask="0x1f"/>
</transaction>
```

### 7.4.14.2.4 Distribution Element

This element describes the approach of RMan/FMan processing message, it includes six types of distribution: RMAN\_RX, RMAN\_TX, FMAN\_RX, FMAN\_TX, RMAN\_TO\_FMAN, FMAN\_TO\_RMAN.

Attributes:

name - (required) string; Defines a unique name for each distribution.

type - (required) string; Defines the distribution type.

#### 7.4.14.2.4.1 RMAN\_RX Distribution

The RMAN\_RX distribution describes RMan how to process the received message from RapidIO end point. Each enabled RMAN\_RX distribution will be configured to a IBCU (Inbound Block Classification Unit). RMan has a total of 32 IBCU. If a

message matches an IBCU setting, it will be put to the corresponding queue. This type element contains four child elements: `rio_port`, `sid`, `queue`, `transactionref`

For example:

```
<distribution name="rman_to_dtsec0" type="RMAN_RX">
 <rio_port number="0" mask="1"/>
 <sid value="0" mask="0xff"/>
 <queue base="0x2500" mode="algorithmic" wq="0"/>
 <transactionref name="dstr-dtsec0"/>
</distribution>
```

#### 7.4.14.2.4.1.1 *RX Rio\_port Element*

This element defines the inbound messages come from which RapidIO port.

Attributes:

`number` - (required) string; Defines port number. The valid value is 1 or 2.

`mask` - (required) string; Defines bit mask for this field. If this value is 1, RMAN\_RX/RMAN\_TO\_FMAN distribution will accepted messages from port 1 and port 2. This setting may be used for port 1 and port 2 loop-back mode.

#### 7.4.14.2.4.1.2 *RX Sid Element*

This element defines the accepted messages sent by which device.

Attributes:

`value` - (required) string; Defines source device id,

`mask` - (required) string; Defines bit mask for this field. Each bit set indicates don't care value for the inbound port-write header attribute.

#### 7.4.14.2.4.1.3 *RX Queue Element*

This element defines the frame queue which the accepted messages will be put to. A frame queue is enqueued onto a Work Queue, and a work queue is grouped into a Channel. RMan supports direct mode and algorithmic mode to generate frame queue ID.

Attributes:

`base` - (required) string; Defines the basic queue id.

`mode` - (required) string; Defines the frame queue mode the valid value is "direct" or "algorithmic".

`wq` - (required) string; Defines the work queue

Transactionref element

This element refers to a transaction element by its name.

Attributes:

`name` - (required) string; Defines name of the transaction referred

#### 7.4.14.2.4.1.4 *Transactionref Element*

This element refers to a transaction element by its name.

Attributes:

`name` - (required) string; Defines name of the transaction referred

### 7.4.14.2.4.2 RMAN\_TX Distribution

This distribution describes RMan how to transmit a message. It contains four child elements: rio\_port, did, queue, transactionref.

For example:

```
<distribution name="rman_to_peer_dtsec0" type="RMAN_TX">
 <rio_port number="0"/>
 <did value="1"/>
 <queue base="0x6000" count="4" wq="0"/>
 <transactionref name="dstr-dtsec0"/>
</distribution>
```

#### 7.4.14.2.4.2.1 TX Rio\_port Element:

This element defines RapidIO port number which will be used to send the messages.

Attributes:

number - (required) string; Defines RapidIO port number.

#### 7.4.14.2.4.2.2 TX Did Element

This element defines device id that the message will send to.

Attributes:

value - (required) string; Defines destination device id.

#### 7.4.14.2.4.2.3 TX Queue Element

This element defines the frame queue which the messages will be put to. The frame queue is enqueued onto a Work Queue. FRA supports up to 4 transmission frame queue.

Attributes:

base - (required) string; Defines the basic queue id.

count - (required) string; Defines the number of frame queue.

wq - (required) string; Defines the work queue

#### 7.4.14.2.4.2.4 Transactionref Element

This element refers to a transaction element by its name.

Attributes:

name - (required) string; Defines name of the transaction referred

### 7.4.14.2.4.3 FMAN\_RX Distribution

FMAN\_RX distribution describes FMan how to process the inbound message. This distribution contains two child elements: fman\_port and queue.

For example:

```
<distribution name="dtsec0_to_rman" type="FMAN_RX">
 <fman_port name="dtsec0"/>
 <queue wq="4"/>
</distribution>
```

#### 7.4.14.2.4.3.1 *Fman\_port Element*

This element define the accepted messages come from which Fman port.

**Attributes:**

name - (required) string; Defines FMan port name. This is a reference to FMan port which defined in network\_cfg element.

#### 7.4.14.2.4.3.2 *FMAN\_RX Queue Element*

This element defines the frame queue which the inbound messages will be put to. The FQID is defined in FMan policy file.

**Attributes:**

wq - (required) string; Defines the work queue.

### 7.4.14.2.4.4 **FMAN\_TX Distribution**

FMAN\_TX distribution describes FMan how to transmit the message.  
This distribution contains two child elements: fman\_port and queue.

For example:

```
<distribution name="dtsec0_to_network" type="FMAN_TX">
 <fman_port name="dtsec0"/>
 <queue count="2" wq="4"/>
</distribution>
```

#### 7.4.14.2.4.4.1 *Fman\_port Element*

This element define the accepted messages come from which Fman port.

**Attributes:**

name - (required) string; Defines FMan port name. This is a reference to FMan port which defined in network\_cfg element.

#### 7.4.14.2.4.4.2 *[FMAN\_TX Queue Element]*

This element defines the frame queue which the messages will be put to. The frame queue is enqueued to a Work Queue. FRA supports up to 4 transmission frame queue.

**Attributes:**

count - (required) string; Defines the number of frame queue.

wq - (required) string; Defines the work queue

### 7.4.14.2.4.5 **RMAN\_TO\_FMAN Distribution**

The RMAN\_TO\_FMAN distribution describes RMan how to transfer the inbound message from RapidIO port to FMan port. It contains five child elements: rio\_port, sid, queue, transactionref and fman\_port

For example:

```
<distribution name="rman_to_fman0_dtsec0" type="RMAN_TO_FMAN">
 <rio_port number="0" mask="1"/>
 <sid value="0" mask="0xff"/>
 <queue base="0x1000" mode="algorithmic" wq="0"/>
 <transactionref name="mbox-dtsec0"/>
```

```
<fman_port name="dtsec0"/>
</distribution>
```

#### 7.4.14.2.4.5.1 *RX Rio\_port Element*

This element defines the inbound messages come from which RapidIO port.

Attributes:

number - (required) string; Defines port number. The valid value is 1 or 2.

mask - (required) string; Defines bit mask for this field. If this value is 1, RMAN\_RX/RMAN\_TO\_FMAN distribution will accepted messages from port 1 and port 2. This setting may be used for port 1 and port 2 loop-back mode.

#### 7.4.14.2.4.5.2 *RX Sid Element*

This element defines the accepted messages sent by which device.

Attributes:

value - (required) string; Defines source device id,

mask - (required) string; Defines bit mask for this field. Each bit set indicates don't care value for the inbound port-write header attribute.

#### 7.4.14.2.4.5.3 *RX Queue Element*

This element defines the frame queue which the accepted messages will be put to. A frame queue is enqueued onto a Work Queue, and a work queue is grouped into a Channel. RMan supports direct mode and algorithmic mode to generate frame queue ID.

Attributes:

base - (required) string; Defines the basic queue id.

mode - (required) string; Defines the frame queue mode the valid value is "direct" or "algorithmic".

wq - (required) string; Defines the work queue

Transactionref element

This element refers to a transaction element by its name.

Attributes:

name - (required) string; Defines name of the transaction referred

#### 7.4.14.2.4.5.4 *Transactionref Element*

This element refers to a transaction element by its name.

Attributes:

name - (required) string; Defines name of the transaction referred

#### 7.4.14.2.4.5.5 *Fman\_port Element*

This element define the accepted messages come from which Fman port.

**Attributes:**

name - (required) string; Defines FMan port name. This is a reference to FMan port which defined in network\_cfg element.

### 7.4.14.2.4.6 FMAN\_TO\_RMAN Distribution

The FMAN\_TO\_RMAN distribution describes FMan how to transfer the inbound message from network port to RMan and RpiadI/O port. It contains five child elements: rio\_port, did, queue, transactionref and fman\_port

For example:

```
<distribution name="fman_to_rman_dtsec0" type="FMAN_TO_RMAN">
 <fman_port name="dtsec0"/>
 <queue wq="0"/>
 <rio_port number="0"/>
 <did value="1"/>
 <transactionref name="mbox-dtsec0"/>
</distribution>
```

#### 7.4.14.2.4.6.1 TX Rio\_port Element:

This element defines RapidI/O port number which will be used to send the messages.

Attributes:

number - (required) string; Defines RapidI/O port number.

#### 7.4.14.2.4.6.2 TX Did Element

This element defines device id that the message will send to.

Attributes:

value - (required) string; Defines destination device id.

#### 7.4.14.2.4.6.3 TX Queue Element

This element defines the frame queue which the messages will be put to. The frame queue is enqueued onto a Work Queue. FRA supports up to 4 transmission frame queue.

Attributes:

base - (required) string; Defines the basic queue id.

count - (required) string; Defines the number of frame queue.

wq - (required) string; Defines the work queue

#### 7.4.14.2.4.6.4 Transactionref Element

This element refers to a transaction element by its name.

Attributes:

name - (required) string; Defines name of the transaction referred

#### 7.4.14.2.4.6.5 Fman\_port Element

This element define the accepted messages come from which Fman port.

Attributes:

name - (required) string; Defines FMan port name. This is a reference to FMan port which defined in network\_cfg element.

### 7.4.14.2.5 Policy Element

This element describes the FRA's behavior. It was composed by one or more distribution order elements.

Attributes:

name - (required) string; Defines a unique name for each policy configuration.

enable - (required) string; Defines the policy status. The valid values are "yes" or "no"

### 7.4.14.2.5.1 Distribution Order Element

The `dist_order` element is a container for a list of distribution references. The distribution reference list contained within `dist_order` element is looked up sequentially and the first conforming record is the record to follow to. Thus, the order of distribution references is important.

#### 7.4.14.2.5.1.1 Distributionref Element

The `distributionref` element refers to a distribution element by its name.

Attributes:

name - (required) string; Defines name of a distribution referred.

For example:

```
<policy name="processing1" enable="no">
 <!-- 10gec packets processing -->
 <dist_order>
 <distributionref name="10gec_to_rman"/>
 <distributionref name="rman_to_peer_10gec"/>
 </dist_order>
 <dist_order>
 <distributionref name="rman_to_10gec"/>
 <distributionref name="10gec_to_network"/>
 </dist_order>
</policy>
```

## 7.4.14.3 Revision History

Table 169. Revision History

Version	Updates
1.1	<ul style="list-style-type: none"> <li>- Change distribution rx, tx, fwd to RMAN_RX, RMAN_TX, FMAN_RX, FMAN_TX respectively</li> <li>- Add network element to describe the FMan ports</li> <li>- Add FMAN_TO_RMAN and RMAN_TO_FMAN distribution to describe the messages transmission between FMan and RMan without core involvement.</li> </ul>
1.0	<ul style="list-style-type: none"> <li>Creation of FRA Configuration UM for SDK 1.1 release</li> <li>- RMan configuration introduction</li> <li>- RapidIO transaction configuration introduction</li> <li>- FRA distribution configuration introduction</li> <li>- FRA policy configuration introduction</li> </ul>

## 7.4.15 NXP USDPAAs FRA User Manual

## 7.4.15.1 Overview

This User Manual describes the NXP RMan Application (FRA) and explains how to install and configure and run FRA.

This Manual provides the following:

- A summary of the FRA application.
- Installing FRA based on SDK.
- Running and testing FRA steps

## 7.4.15.2 Introduction

### 7.4.15.2.1 Purpose

FRA NXP RMan Application is a Linux user space software to transfer packets from SRIO port to FMan port. This document describes the FRA installation, configuration, running and test.

### 7.4.15.2.2 Definitions and Acronyms

- FMan - Frame Manager
- BMan - Buffer Manager
- QMan - Queue Manager
- RMan - RapidIO Message Manager
- FRA - NXP RMan Application
- USDPAAs - User Space Data Path Acceleration Architecture
- DPAA - Data path acceleration architecture

## 7.4.15.3 Overview of FRA

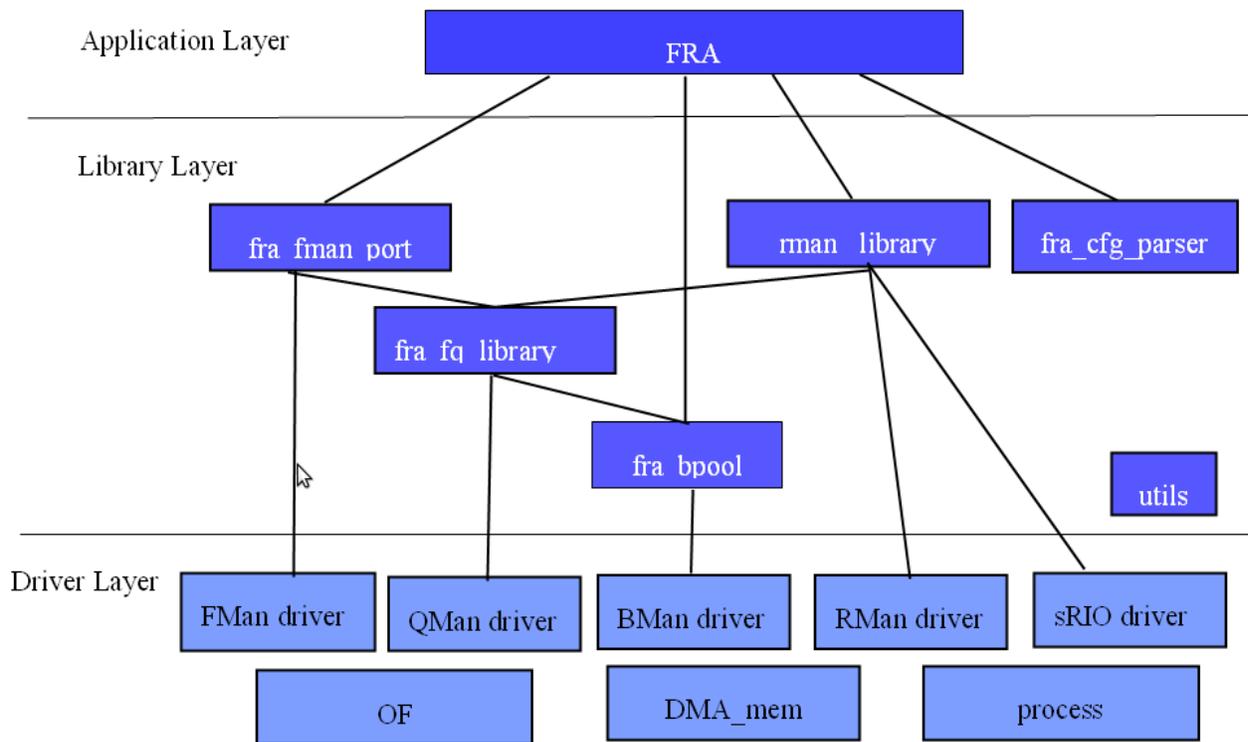
The USDPAAs FRA application is a multi-thread USDPAAs application that runs simultaneously on one or more boards which are connected with RapidIO cable. FRA application forwards IPv4 packets from one Ethernet interface to another boards using RapidIO transaction.

### Overview of USDPAAs

The NXP Data Path Acceleration Architecture comprises a set of hardware components which are integrated via a hardware queue manager and use a common hardware buffer manager. Software accesses the DPAA via hardware components called "software portals". These directly provide queue and buffer manager operations such as enqueues, dequeues, buffer allocations, and buffer releases and indirectly provide access to all of the other DPAA hardware components via the queue manager. USDPAAs is a software framework that permits Linux user space applications to directly access the DPAA queue and buffer manager software portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as RMan and FMan

### FRA Architecture

FRA is based on USDPAAs. But comparing the normal USDPAAs application, in order to reuse the frame queue and buffer pool codes between processing network packets and RapidIO packets, FRA's architecture is adjusted.



**Figure 156. FRA Architecture**

As Fig 1 shown, FRA is split to three layers, driver layer, library layer and application layer.

### Driver Layer

Driver layer contains FMan, QMan, BMan, RMan, sRIO, DMA\_mem, of, and process drivers. All the driver code files are located in “usdpaa/driver” folder, which are used by all the USDPAAs applications. For FMan, QMan, BMan, DMA\_mem, of, process driver information please refer to other USDPAAs manuals.

### sRIO driver

The serial RapidIO interface provides a RapidIO port to communicate with other RapidIO devices. sRIO driver manages RapidIO ports hardware and provides some interfaces to configure port’s attributes and RapidIO protocol configuration such as “accept all”, “target id”, “traffic mode”. It also provides a few interface to handle interrupt events.

### RMan driver

RapidIO message manager supports the message passing programming model for inter-processor and inter-device communication. RMan has a total of 32 inbound traffic classification units. A block is a group of eight classification units. The units are managed through a set of runtime registers. There are 4 outbound segmentation units. RMan driver is used to manage RMan global registers and inbound blocks. Users can call `rman_dev_config()` to configure global registers. This function supports setting inbound message descriptor write mode and the frame queue assembly rule of the doorbell mailbox and data-streaming transaction. RMan driver provides enable/disable/clear/status interfaces to handle interrupt and provides a few functions to request/release, enable/disable configure classification unit.

### Library Layer

Library layer contains utils library, FMan port library, frame queue library, buffer pool library, RMan library, FRA configure parser library. All the libraries code files are located in “usdpaa/app/fra/lib” folder.

## Utils Library

Utils library is at the bottom of the layer. It provides the common interface for all the other libraries. At present, utils mainly provides `fra_dbg()` function to print the debug information and `cli_cmd()` function to define a command.

## Buffer Pool Library

Buffer pool library is intended to manage buffer pools. FRA supports up to 64 buffer pools. Each pool has buffer pool id, buffer number and buffer size three attributes which are grouped to a structure, as follows:

```
struct bpool_config {
 int bpid;
 uint32_t num;
 uint32_t sz;
};
```

Users can pass the array of the structure `bpool_config` to function `bpools_init()` to initialize a series of buffer pools. Buffer pool library will initialize the buffer pools and allocate DMA memory and release them to the pools. Buffer pool interface also provides acquire/free buffer functions to allocate and release buffers.

## Frame queue Library

Frame queues that used by FMan and RMan can be separated into three types: nonpcd, pcd and tx.

Nonpcd frame queues are used to receive the error or complete status frame which are generally enqueued by FMan or RMan. Those frame queues normally has a higher priority and uses dynamically frame queue id, and only support `QM_FQCTRL_CTXASTASHING` option.

Pcd frame queues are used to store inbound data frame, such as network packets frame or RapidIO transaction message frames. Those frame queues support more control option such as `QM_FQCTRL_AVOIDBLOCK`, `QM_FQCTRL_PREFERINCACHE` and `QM_FQCTRL_CTXASTASHING`.

Tx frame queues are used to send outbound data frame which Fman or Rman dequeues and send out. These frame queues need to set context A and B according hardware specification.

This library supports send/free frame via calling `fra_send_frame()` and `fra_drop_frame()` respectively.

## FMan Port Library

Each FMan port is separated into rx and tx functional modules. Rx module includes pcd rx sockets and nonpcd rx sockets. Pcd rx sockets is used to receive the Ipv4/Ipv6 real data frames. Nonpcd rx sockets is used to receive the frame that describe error status. The tx module is similar to rx module.

## RMan Library

Rman library performs the initialization of the RMan and sRIO ports. There is a series of A interfaces provided to receive and transmit messages.

The main interfaces as follows:

### 1. RMan rx API

RMan rx socket contains a set of frame queues to receive the RapidIO messages. Each RMan rx socket corresponds to a classification unit.

#### 1.1 rman\_rx\_init

This function requests a RMan hardware resource-classification unit and then creates the rx frame queues. Returns the pointer of `rman_rx` on success or NULL on failure.

#### 1.2 rman\_rx\_listen

Configure classification unit to receive the specific RapidIO messages which come from specified sid did and port.

#### 1.3 rman\_rx\_enable

Enable `rman_rx` to receive messages.

#### 1.4 rman\_rx\_disable

Stop rman\_rx receiving messages, but don't release ibcu resource.

#### 1.5 rman\_rx\_finish

Stop classification unit receiving, release rx frame queues ibcu resource and rman\_rx socket.

#### 1.6 rman\_rx\_get\_fqs\_num

RMan supports two mode direct and algorithmic to generate frame queue ID, if direct mode returns 1, if algorithmic mode, calculates and returns the number of frame queues according to algorithmic rule and transaction configuration.

#### 1.7 rman\_rx\_get\_ibcu

Return the classification unit index which is assigned to the specific RMan rx socket

#### 1.8 rman\_rx\_get\_opt

Structure hash\_opt is used to describe which frame queue the received frame will be enqueued to Each RMan rx may include one or some rx frame queues, each rx frame queue has a hash opt attribute.

### 2. RMan tx API

RMan tx socket contains a set of frame queues to send the RapidIO messages using the same transaction.

#### 2.1 rman\_tx\_init

Create tx frame queues and tx status frame queue

#### 2.2 rman\_tx\_status\_listen

Set rman\_tx to receive the completed or/and error status frame

#### 2.3 rman\_tx\_connect

Connect rman\_tx socket to the destination device

#### 2.4 rman\_tx\_finish

Release RMan tx and tx status frame queues.

#### 2.5 opt\_bindto\_rman\_tx

Bind the opt to the specific rman tx

### FRA Configuration Parser Library

Fra configuration parser interface is to parse the fra configuration file which contains values of the RapidIO transaction header fields and distribution settings and policy setting.

### Application Layer

FRA application is similar with normal USDPAA application. It also supports add/rm/list commands to manage thread, supports q or quit to exit. FRA adds status command to print the fra information including Rman configuration, RapidIO ports information and distributions information.

Fra application supports a few options defined by macro in fra\_cfg.h file.

ENABLE\_FRA\_DEBUG - to print debug information

FRA\_CORE\_COPY\_MD – to use processor to copy the RMan descriptor

FRA\_MBOX\_MULTICAST – to support multi-cast mode

FRA\_VIRTUAL\_MULTI\_DID – to enable virtual multi-did.

The frame queue options are also defined in this files. For the detailed information please refer to USDPAA user manual.

This file also contains buffer pool settings.

bp 10 is used to store doorbell messages

bp 11 is used to store data-streaming/mailbox messages

bp 12 is used to store s/g tables.

```
#define DMA_MEM_BP4_BPID10
#define DMA_MEM_BP4_SIZE80
#define DMA_MEM_BP4_NUM0x100 /* 0x100*80==20480 (20KB) */
#define DMA_MEM_BP5_BPID11
#define DMA_MEM_BP5_SIZE1600
#define DMA_MEM_BP5_NUM0x2000 /* 0x2000*1600==13107200 (12.5M) */
#define DMA_MEM_BP6_BPID12
#define DMA_MEM_BP6_SIZE64
#define DMA_MEM_BP6_NUM0x2000 /* 0x2000*64==524288 (0.5MB) */
```

```
/* DMA memory size */
#define FRA_DMA_MAP_SIZE0x4000000 /* 64MB */
```

## FRA Configuration

Fra configuration includes RMan global configuration, transaction settings distribution setting and policy setting. FRA Configuration User Manual detailedly describes each element

## RMan Configuration

This element is responsible for RMan general settings. It contains a few child elements: fqbits and md\_create and bpid settings.

For example:

```
<rman_cfg>
 <fqbits type="Data-streaming" value="2"/>
 <fqbits type="Mailbox" value="2"/>
 <md_create mode="yes"/>
 <bpid type="Data-streaming" value="11"/>
 <bpid type="Doorbell" value="10"/>
 <bpid type="Mailbox" value="11"/>
 <sgbpid value="12"/>
</rman_cfg>
```

## Transaction Configuration

FRA configuration supports doorbell mailbox and data-streaming transaction. Almost all the values of the header fields of each transaction can be defined in this element.

For example:

```
<transaction name="dstr-10gec" type="Data-streaming">
 <flowlvl value="0" mask="2"/>
 <cos value="15" mask="0"/>
 <streamid value="0" mask="0x1f"/>
</transaction>
```

## Distribution Configuration

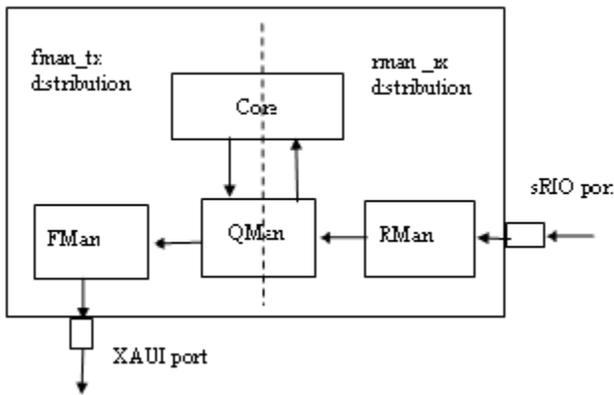
This element describes the approach of processing message of RMan or FMan, it includes six types of distribution: rman\_rx, rman\_tx, fman\_rx, fman\_tx, fman\_to\_rman, rman\_to\_fman.

## Policy Configuration

A distribution describes the one device module FMan or RMan how to process a series of specified packets. A distribution order element containing a sequence of distribution describes the board how to process specified packets. Policy element including one or more distribution order elements describes the one or two boards how to process all the packets. The typical dist order elements and the packets processing flows are as follows:

1. rman\_rx and fman\_tx distribution forms a packets processing flow

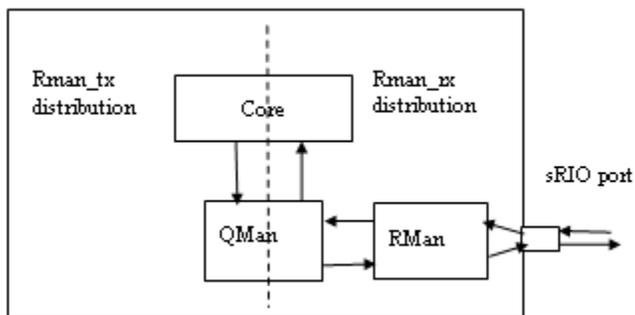
```
<dist_order>
 <distributionref name=" rman_to_10gec "/>
 <distributionref name="10gec_to_network "/>
</dist_order>
```



**Figure 157. RMan-core-FMan Processing Packet Flow**

2. rman\_rx and rman\_tx distributions forms a loopback processing flow

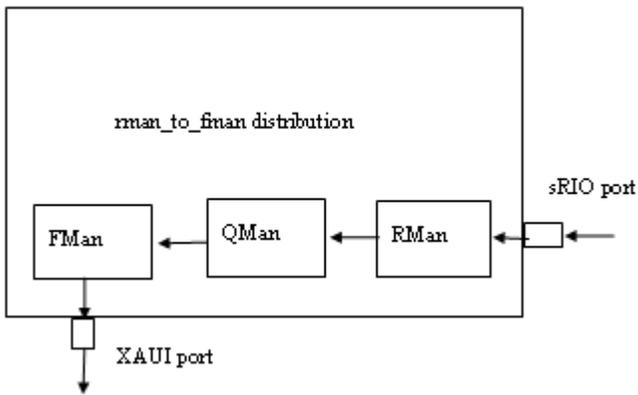
```
<dist_order>
 <distributionref name="rman_to_10gec"/>
 <distributionref name="rman_to_peer_10gec"/>
</dist_order>
```



**Figure 158. RMan-core-RMan Processing Packet Flow**

3. rman\_to\_fman distribution forms a packets processing flow without core involvement.

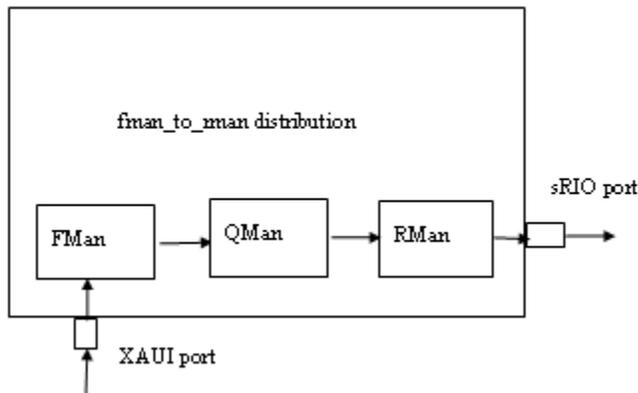
```
<dist_order>
 <distributionref name="rman_to_fman0_10gec"/>
</dist_order>
```



**Figure 159. RMan-FMan Processing Packet Flow**

4. fman\_to\_rman distribution forms a packets processing flow without core involvement.

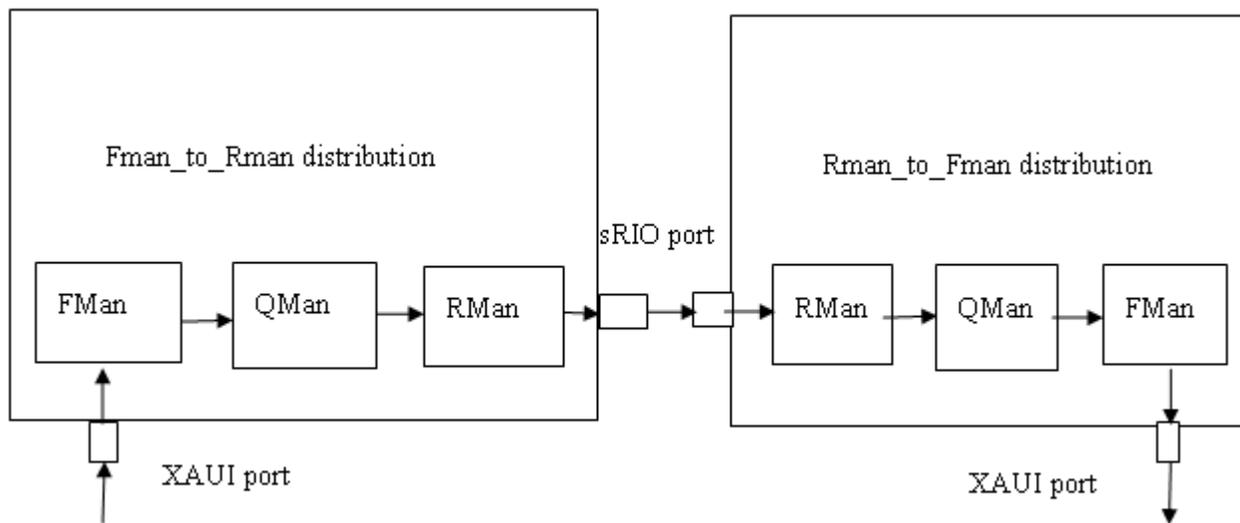
```
<dist_order>
 <distributionref name="fman_to_rman0_10gec"/>
</dist_order>
```



**Figure 160. FMan-RMan Processing Packet Flow**

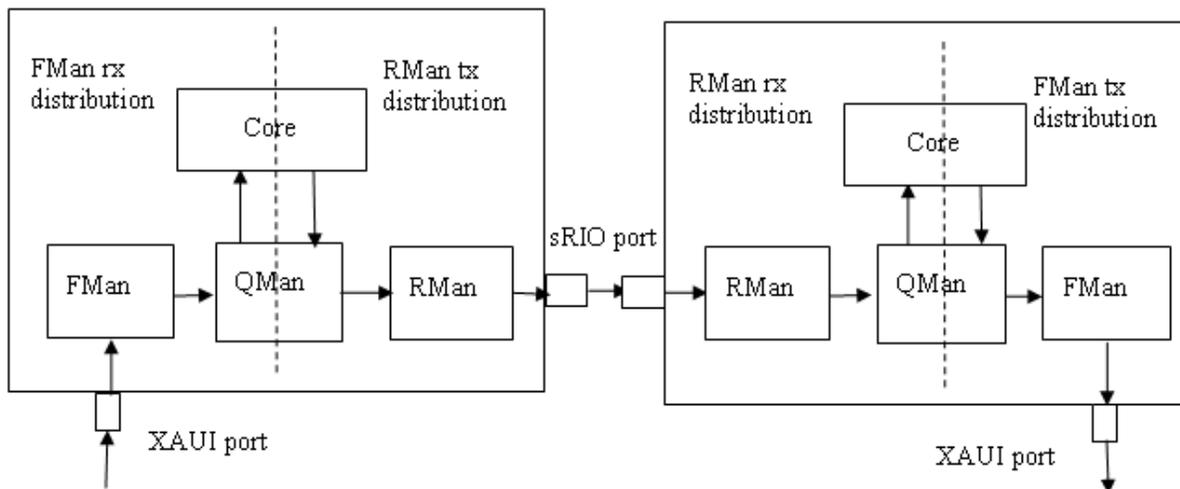
**Packet Flow**

The FRA application can run on two boards at the same time, the two boards host and agent are connected with RapidIO cable. FRA first parses the configuration, and then creates threads to do packet processing. FRA provides two packets processing flows – processing1 and processing2.



**Figure 161. Processing1 Packet Flow**

As the Fig6 shown, the processing1 is that RMan directly enqueues the packets to FMan dedicated transmission channel, FMan directly enqueues the packets to RMan dedicated transmission channel, and the core does not need to participate in the processing.



**Figure 162. Processing2 Packet Flow**

As the Fig7 shown, the processing 2 is that FMan and RMan enqueues the packets to a frame queue of pool channel, and the core will dequeue and process the packets, and then enqueue them to the FMan or RMan dedicated transmission channel to send out.

### FRA Configuration Files

FRA configuration files are located in “usdpaa/app/fra/app\_config” folder. There are five files.  
 fra\_config\_dstr\_processing1.xml – do packets processing1 using data-streaming transaction  
 fra\_config\_dstr\_processing2.xml - do packets processing2 using data-streaming transaction  
 fra\_config\_mbox\_processing1.xml - do packets processing1 using mailbox transaction  
 fra\_config\_mbox\_processing2.xml - do packets processing1 using mailbox transaction

fra\_config\_dstr\_port1\_port2\_loopback.xml – do packets loopback processing using data-streaming transaction

fra\_config\_port\_write\_test.xml - do Port-Write testing

## Features

The FRA features are as follows:

- Supports all the network interfaces settings.
- Supports sRIO ports and RMan error interrupt handler.
- Supports two sRIO ports loopback mode on one board.
- Supports setting RMan writing the inbound message descriptor mode
- Supports Doorbell transaction with the following features:
  - Supports setting flow level using configuration file
- Supports Mailbox transaction with the following features:
  - Supports setting flow level, mailbox, letter, message length fields using configuration file
  - Supports setting the frame queue mode “direct” or “algorithmic.”
  - Supports multi-cast mode
- Supports Data streaming transaction with the following features:
  - Supports setting flow level, cos, streamid fields using configuration file
  - Supports setting the frame queue mode “direct” or “algorithmic.”
- Supports setting the behavior of each network port.
  
- Supports Port-Write transaction

## 7.4.15.4 Running FRA on Two Boards

### 7.4.15.4.1 Installation

The DPAA SDK has included the FRA code. So just need to install SDK.

### 7.4.15.4.2 Compilation

Please follow the SDK build guide to build the software. This will perform a default build of all files needed to boot Linux and run the FRA software. These files include dtb, FMan ucode image, u-boot image, kernel image, and rootfs image.

When build kernel, make sure these options are selected.

```
Device Drivers --->
 <*> Userspace I/O drivers --->
 <*> Freescale Serial RapidIO support
 [*] Staging drivers --->
 [*] Freescale RapidIO Message Manager support
```

### 7.4.15.4.3 Configuring FRA

The FRA application uses three configuration files. One is to configure how FRA to process packets, the default name is fra\_config\_dstr\_processing1.xml. Currently, default packet flow is processing 1. If you want to test the processing 2, you should run FRA with fra\_config\_dstr\_processing2.xml configuration file. Similarly, if we want to use mailbox transaction to transmit the IP packets, using fra\_config\_mbox\_processing1.xml or fra\_config\_mbox\_processing2.xml. All the FRA configuration files located in /usr/etc folder. More detailed configuration refers to NXP FRA Configuration User's Manual.

The other two files are to configure FMD action, default names are usdpaa\_config\_p3\_p5\_serdes\_0x33.xml and usdpaa\_policy\_hash\_ipv4.xml. these two file we can refer to frame manager configuration manual.

If we want to specify the configuration file, use the following command on the both boards:

```
fra -c usdpaa_config_p3_p5_serdes_0x33.xml -p usdpaa_policy_hash_ipv4.xml -f
fra_config_dstr_processing1.xml
```

### 7.4.15.4.3.1 Selecting Ethernet interfaces for FRA

The following device tree snippet shows a Linux private interface and also an interface used privately by FRA.

```
ethernet@0 {
 compatible = "fsl,p3041-dpa-ethernet-init", "fsl,dpa-ethernet-init";
 fsl,bman-buffer-pools = <&bp7 &bp8 &bp9>;
 fsl,qman-frame-queues-rx = <0x50 1 0x51 1>;
 fsl,qman-frame-queues-tx = <0x70 1 0x71 1>;
};
ethernet@1 {
 compatible = "fsl,p3041-dpa-ethernet", "fsl,dpa-ethernet";
 fsl,fman-mac = <&enet0>;
};
```

The first Ethernet is used by FRA. The second is used by the Linux Ethernet driver.

The following list shows the correspondence between Ethernets in the device tree and physical Ethernet MACs on FMan hardware instances on P3041.

**Table 170. Correspondence between Ethernets in DTS and physical Ethernet**

SW Interface #	FMAN1
0	DTSEC 0
1	DTSEC 1
2	DTSEC 2
3	DTSEC 3
4	DTSEC 4
5	10GEC

### 7.4.15.4.4 Prepare the Hardware

The FRA application needs two PC or one PC with multiple Ethernet interfaces, two p3041DS or P5020DS boards, when using 0x33 RCW each board should be inserted a RapidIO card to slot6. Check board's switch, sw2 is 0b00100001 (slot7->PEX1, slot6->SRIO, slot4->PEX3, slot2->XAUI), Sw5 is 0b00010100 (Serdes reference clock for bank1 is 100MHz, bank2 is 125MHz bank3 is 125MHz). For P5020DS board sw2 is also 0b00100001. For P2041RDB, using 0x02 RCW, RapidIO card should be inserted to slot1. and the following command should be run

```
cpld lane_mux 6 0;
cpld lane_mux a 0;
cpld lane_mux c 0;
cpld lane_mux d 0;
cpld reset altbank;
```

FRA supports 15G network interfaces. But 0x33 RCW only provides 12G network interfaces: RGMII1 (DTSEC3), RGMII2 (DTSEC4) and XAUI. As the Fig8 shown, we can use DTSEC3 to receive and transmit the IP packets. The host machine should connect to host board RGMII1 port, and agent machine should connect to agent board RGMII 1 port too.

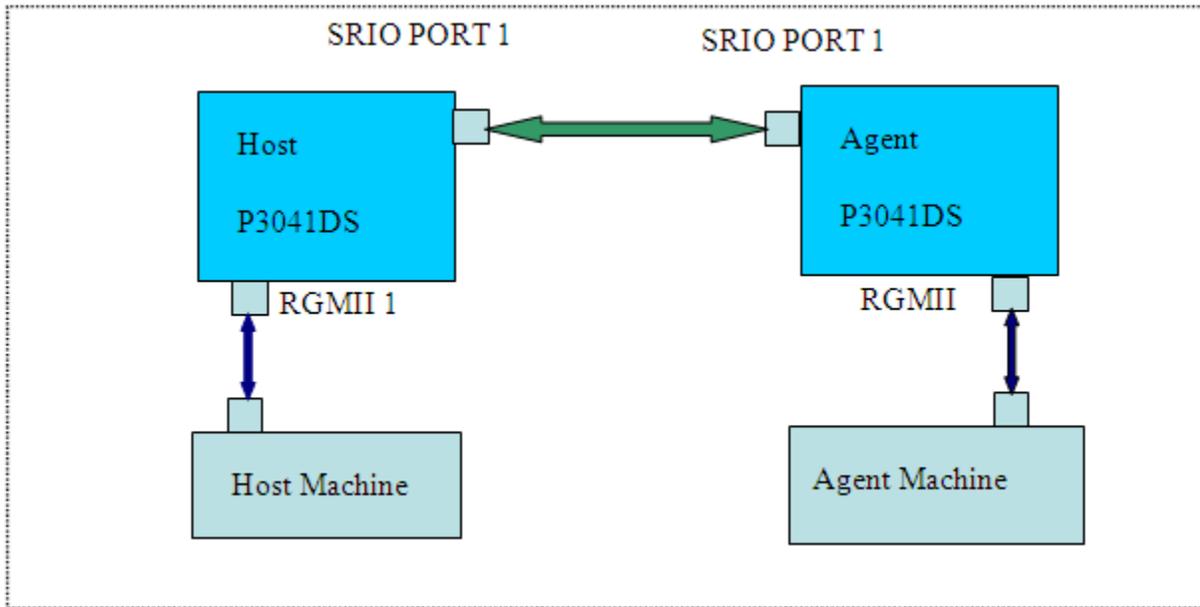


Figure 163. Serdes 0x33 protocol hardware setup

#### 7.4.15.4.5 Managing RCW and U-boot Image

The RCW, FMan microcode, and u-boot binary files must be programmed into the P3041DS NOR-flash We can use Processor Expert (PEX) to create serdes 0x33 rcw file and then convert to rcw image.

```
serdes 0x33 RCW of P3041DS is :
00000000: AA55 AA55 010E 0100 1260 0000 0000 0000
00000010: 241C 0000 0000 0000 CC98 4A00 0300 2000
00000020: FE80 0000 4100 0000 0000 0000 0000 0000
00000030: 0000 0000 1007 0000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 CF8E 2F88
```

```
serdes 0x33 RCW of P5020DS is :
00000000: AA55 AA55 010E 0100 0C54 0000 0000 0000
00000010: 1E12 0000 0000 0000 CC98 4A00 0300 2000
00000020: FE80 0000 4100 0000 0000 0000 0000 0000
00000030: 0000 0000 1007 0000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 1AC8 E13F
```

```
serdes 0x02 RCW of P2041RDB is:
00000000: AA55 AA55 010E 0100 1260 0000 0000 0000
00000010: 241C 0000 0000 0000 0899 30C0 C7C0 2000
00000020: FE80 0000 4000 0000 0000 0000 0000 0000
00000030: 0000 0000 D003 0F07 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 57B0 F7D9
```

Convert rcw xxd format to image

```
$ xxd -r rcw.xxd > rcw-0x33.bin
```

The P3041DS divides the flash into multiple banks. This permits multiple copies of RCW and u-boot files to be kept in flash. It is not required, but NXP suggests leaving a functional u-boot in bank0 (which is used when the board is powered on) and program new rcw and u-boot files into bank 4. This method ensures that a working u-boot (in bank 0) is retained in case of a mistake in programming bank 4. To program bank 4, first do a power-on reset to boot from bank 0. Use the u-boot in bank 0 to program the files into bank 4. Networking parameters need to be set for the u-boot in bank 0 so that it can tftp. U-boot uses environment variables to control network settings. Variables are given values with the `isetenv` command. The choices for these settings must match your local network. Note in particular `serverip`. A machine with that IP address is assumed to be running a tftp server containing the files to be transferred via tftp. The major U-boot networking variables are:

```
=> setenv ethact <active Ethernet interface for u-boot use. Should be FM1@DTSEC5 if using the
recommended SerDes protocol number 0xe>
=> setenv ethaddr <MAC address 0>
=> setenv ethNaddr <MAC address N=1 to 5>
...
=> setenv ipaddr <your ip address>
=> setenv netmask <your netmask>
=> setenv gatewayip <your default gateway ip address>
=> setenv serverip <the address of your TFTP server>
=> sav
```

The following instructions show how to reprogram bank 4. Remember to do this only when booted from bank 0.

```
Be booted from bank 0!!!!
=> tftp 0x02000000 u-boot.bin
=> protect off 0xebf80000 +$filesize
=> erase 0xebf80000 +$filesize
=> cp.b 0x02000000 0xebf80000 $filesize
=> tftp 0x02000000 rcw_0x33.bin
=> protect off 0xec000000 +$filesize
=> erase 0xec000000 +$filesize
=> cp.b 0x02000000 0xec000000 $filesize
=> tftp 0x02000000 fsl_fman_ucode.bin
=> protect off 0xeb000000 +$filesize
=> erase 0xeb000000 +$filesize
=> cp.b 0x02000000 0xeb000000 $filesize
Reset to bank 4:
=>pix altbank
```

U-boot in bank 4 will then run and one can proceed to boot Linux and run the FRA software.

## 74.15.4.6 Booting Linux

Set following environment settings on board's bank4, Setup networking variables for tftp in bank4:

```
=> setenv ethact <active Ethernet interface for u-boot use>
=> setenv ethaddr <MAC address 0>
=> setenv ethNaddr <MAC address N=1 to 5>
...
=> setenv ipaddr <your ip address>
=> setenv netmask <your netmask>
=> setenv gatewayip <your default gateway ip address >
=> setenv serverip <the address of your TFTP server>
=> sav
```

Setup and save other uboot variables to tftp the USDPAA device tree, root file system and Linux image and boot Linux.

```
=>setenv bootargs "root=/dev/ram rw console=ttyS0,115200"
=>setenv fraboot "tftp 1000000 uImage-p3041ds.bin; tftp 2000000 fsl-image-core-p3041ds.ext2.gz.u-
```

```
boot; tftp c00000 uImage-p3041ds-usdpaa.dtb; bootm 1000000 2000000 c00000"
=>run fraboot
```

Following are some booting information points:

u-boot boot log:

```
Flash: 128 MiB
L2: 128 KB enabled
Corenet Platform Cache: 1024 KB enabled
SRI01: enabled
SRI02: disabled
NAND: 1024 MiB
MMC: FSL_ESDHC: 0
```

Linux kernel boot log:

```
fsl-of-srio ffe0c0000.rapidio: Rapidio UIO driver initialized
...
fsl-of-rman ffe1e0000.rman: Of-device full name /soc@ffe000000/rman@1e0000 initialized
fsl-of-rman ffe1e0000.rman: RMan inbound block0 initialized.
fsl-of-rman ffe1e0000.rman: RMan inbound block1 initialized.
fsl-of-rman ffe1e0000.rman: RMan inbound block2 initialized.
fsl-of-rman ffe1e0000.rman: RMan inbound block3 initialized.
```

## 74.15.4.7 Running FRA

Logging into Linux

At the Linux prompt, login as "root".

```
Yocto (Built by Poky 6.0) 1.1 p3041ds ttyS0

p3041ds login: root
root@p3041ds:~#
```

Configure FMan PCD using fmc with the XML files in /usr/etc:

```
[root@p3041 root]# cd /usr/etc
[root@p3041 root]# fmc -c usdpaa_config_p3_p5_serdes_0x33.xml -p usdpaa_policy_hash_ipv4.xml -a
```

Run the FRA application:

```
root@p3041ds:~# fra
Found /fsl,dpaa/dpa-fman0-oh@1, Tx Channel = 47, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@3, Tx Channel = 44, FMAN = 0, Port ID = 3
Found /fsl,dpaa/ethernet@4, Tx Channel = 45, FMAN = 0, Port ID = 4
Found /fsl,dpaa/ethernet@5, Tx Channel = 40, FMAN = 0, Port ID = 0
Configuring for 3 network interfaces
fra: BPOOL: Release 8192 bufs to BPID 9
fra: BPOOL: Release 256 bufs to BPID 10
fra: BPOOL: Release 8192 bufs to BPID 11
fra: BPOOL: Release 8192 bufs to BPID 12
fra: RMan inbound block0 initialized
fra: RMan inbound block1 initialized
fra: RMan inbound block2 initialized
fra: RMan inbound block3 initialized
fra: can not find fman port dtsec0
fra: can not find fman port dtsec1
```

```
fra: can not find fman port dtsec2
Start dist(rman_to_fman0_10gec)
Start dist(fman_to_rman_10gec)
Start dist(rman_to_fman0_dtsec4)
Start dist(fman_to_rman_dtsec4)
Start dist(rman_to_fman0_dtsec3)
Start dist(fman_to_rman_dtsec3)
Thread uid:0 alive (on cpu 1)
fra>
```

Additional FRA threads can be started on other CPUs by entering commands at the FRA command prompt. Running threads can also be queried and the application can also be shutdown.

- add a FRA thread on a single cpu (e.g. cpu 2)

```
fra> add 2
```

- add FRA threads on a range of cpus

```
fra> add 2..3
```

- list the cpus currently enabled (by querying them, i.e. this also verifies that they aren't blocked)

```
fra> list
```

Thread alive on cpu 1

Thread alive on cpu 2

Thread alive on cpu 3

- display FRA configuration and status

```
fra> status
RMan configuration:
Create inbound message descriptor: yes
The algorithmic frame queue bits info:
data streaming:2 mailbox:2
BPID info:
data streaming:11 mailbox:11 doorbell:10 sg:12
Use SRIO port 0: using lane 0
Create 3 RX sockets and 12 frame queues
Create 3 TX sockets and 96 frame queues
distribution order-1:
distribution-1-RMAN_TO_FMAN: rman_to_fman0_10gec
rio port:0 - 10gec
sid:0 mask:255 queue mode:algorithmic
rio_tran:dstr-10gec type:Data-streaming
base FQID:0x1500 count:4 configured to IBCU 0

distribution order-2:
distribution-1-FMAN_TO_RMAN: fman_to_rman_10gec
FMan:10gec - rio port 0
did:1
rio_tran:dstr-10gec type:Data-streaming

distribution order-3:
distribution-1-RMAN_TO_FMAN: rman_to_fman0_dtsec4
rio port:0 - dtsec4
sid:0 mask:255 queue mode:algorithmic
rio_tran:dstr-dtsec4 type:Data-streaming
```

```
base FQID:0x1400 count:4 configured to IBCU 1

distribution order-4:
distribution-1-FMAN_TO_RMAN: fman_to_rman_dtsec4
FMan:dtsec4 - rio port 0
did:1
rio_tran:dstr-dtsec4 type:Data-streaming

distribution order-5:
distribution-1-RMAN_TO_FMAN: rman_to_fman0_dtsec3
rio port:0 - dtsec3
sid:0 mask:255 queue mode:algorithmic
rio_tran:dstr-dtsec3 type:Data-streaming
base FQID:0x1300 count:4 configured to IBCU 2

distribution order-6:
distribution-1-FMAN_TO_RMAN: fman_to_rman_dtsec3
FMan:dtsec3 - rio port 0
did:1
rio_tran:dstr-dtsec3 type:Data-streaming
fra>
```

- Enable or disable debug information output

Note: this function is effective only when compiling with ENABLE\_FRA\_DEBUG definition

```
fra> debug [on/off]
```

- If you want to quit FRA , should use the following command, FRA will release the related resource.

```
fra> q
```

## 7.4.15.4.8 Testing FRA

In order to test FRA, we need to send IP packets to host board's Ethernet interface, since FRA does not support ARP protocol, we should do IP and MAC binding on host PC. Following are the test FRA steps on P3041DS boards.

1. Setup the hardware as Fig9 and run the FRA application on both host and agent as described in previous section.
2. Find the host board's Ethernet interface Mac address

This case, FRA will use the port 3 to receive the IP packets; on host board console Linux startup log has the MAC information:

```
cpu0: fsl_mac: FSL FMan MAC API based driver ()
cpu0: fsl_mac: ffe4e6000.ethernet: FMan dtSEC version: 0x08240101
cpu0: fsl_mac: ffe4e6000.ethernet: FMan MAC address: 00:e0:0c:00:d7:03
cpu0: fsl_mac: ffe4e8000.ethernet: FMan dtSEC version: 0x08240101
cpu0: fsl_mac: ffe4e8000.ethernet: FMan MAC address: 00:e0:0c:00:d7:04
cpu0/0: Applying 10G tx-ecc error workaround (10GMAC-A004) ...
cpu0/0: done.
cpu0: fsl_mac: ffe4f0000.ethernet: FMan XGEC version: 0x00010330
cpu0: fsl_mac: ffe4f0000.ethernet: FMan MAC address: 00:e0:0c:00:d1:05
```

So we find DTSEC3's Mac address is 00:e0:0c:00:d7:03.

3. If we has configure the Ethernet card IP address 192.168.2.1, so on host machine create a ARP entry in subnet range of the host pc network port connected to host board's DTSEC3, by using the following command:

```
$ sudo arp -s 192.168.2.2 00:e0:0c:00:d7:03
```

- On host machine ping host board's DTSEC3

```
$ ping 192.168.2.2
```

- Run Wireshark on agent machine and to capture the network port connected to the agent board's DTSEC3 network port.

```
$ sudo wireshark
```

As the figure below shows, we should see the ping packets sent from host machine.

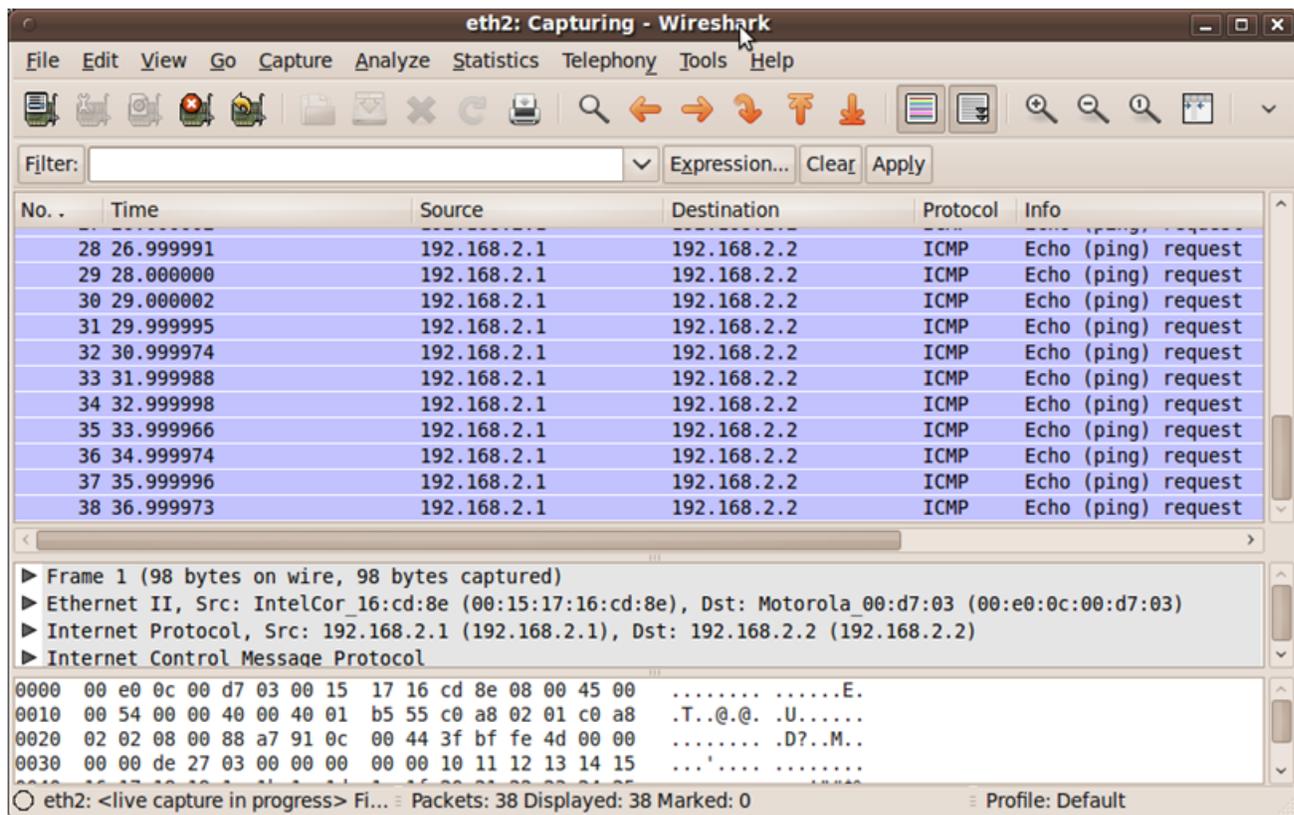


Figure 164. Wireshark Capturing Packets

### 74.15.4.9 Debugging FRA

We may need to debug FRA with detailed output log. FRA provides debug command to enable or disable those debug information output. In order to support debug mode, we should insert the following sentence to fra.h file line 46 and then re-compile.

```
#define ENABLE_FRA_DEBUG
```

Use the following command to enable or disable the debug information output.

```
fra> debug [on/off]
```

## 7.4.15.4.10 Testing Port Write

A Type8 port-write is intended as an error reporting mechanism from an end point-less device to a control processor or other system host. The RMan also supports generation of outbound port-writes for testability and debug. Inbound port-writes have dedicated hardware for guaranteed delivery.

Run the FRA application with Port-Write test configuration file

```
root@t4240qds:~# fra -r -f fra_config_port_write_test.xml
fra: BPOOL: Release 8192 bufs to BPID 9
fra: BPOOL: Release 256 bufs to BPID 10
fra: BPOOL: Release 8192 bufs to BPID 11
fra: BPOOL: Release 8192 bufs to BPID 12
Start dist(pw_to_peer)
fra: RMan inbound block0 is initialized
Start dist(pw_from_peer)
Thread uid:0 alive (on cpu 1)
fra>
```

Send Port-Write message using "pw" command

```
fra> pw 1234
```

Receive the Port-Write message on the peer

```
fra> Port Write: get 4 bytes data:1234
```

- Note: RapidIO supports a maintenance port-write type that contains a data payload 4 8 16 32 or 64 bytes.

## 7.4.15.5 Running FRA with flow control

The chapter is intended to introduce how to run FRA with flow control.

The operations of installation and testing are similar to Chapter 3, So those sections please refer to above description.

### 7.4.15.5.1 Booting Linux

Set following environment settings on board's bank4, Setup networking variables for tftp in bank4:

```
=> setenv ethact <active Ethernet interface for u-boot use>
=> setenv ethaddr <MAC address 0>
=> setenv ethNaddr <MAC address N=1 to 5>
...
=> setenv ipaddr <your ip address>
=> setenv netmask <your netmask>
=> setenv gatewayip <your default gateway ip address >
=> setenv serverip <the address of your TFTP server>
=> sav
```

Setup srio device ID on left board.

```
=> mm 0xffe0d0100
fe0d0100: 00000000 ? 808b0000
fe0d0104: 00000000 ? q
=>
```

Setup srio device ID on right board.

```
=> mm 0xffe0d0100
fe0d0100: 00000000 ? 808d0000
fe0d0104: 00000000 ? q
=>
```

Setup and save other uboot variables to tftp the USDPAAs device tree, root file system and Linux image and boot Linux.

```
=>setenv bootargs "root=/dev/ram rw console=ttyS0,115200 usdpaa_mem=0x4000000"
=>setenv fraboot "tftp 1000000 uImage-t4240qds.bin; tftp 2000000 fsl-image-core-
t4240qds.ext2.gz.u-boot; tftp c00000 uImage-t4240qds-usdpaa.dtb; bootm 1000000 2000000 c00000"
=>run fraboot
```

Following are some booting information points:

u-boot boot log:

```
Flash: 128 MiB
L2: 128 KB enabled
Corenet Platform Cache: 1024 KB enabled
SRI01: enabled
SRI02: disabled
NAND: 1024 MiB
MMC: FSL_ESDHC: 0
```

Linux kernel boot log:

```
fsl-of-srio ffe0c0000.rapidio: Rapidio UIO driver initialized
...
fsl-of-rman ffe1e0000.rman: Of-device full name /soc@ffe000000/rman@1e0000 initialized
fsl-of-rman ffe1e0000.rman: RMan inbound block0 initialized.
fsl-of-rman ffe1e0000.rman: RMan inbound block1 initialized.
fsl-of-rman ffe1e0000.rman: RMan inbound block2 initialized.
fsl-of-rman ffe1e0000.rman: RMan inbound block3 initialized.
```

## 74.15.5.2 Compilation FRA with flow control

Please follow the SDK build guide to build the software. This will perform a default build of all files needed to boot Linux and run the FRA software. These files include dtb, FMan ucode image, u-boot image, kernel image, and rootfs image.

When build kernel, make sure these options are selected.

```
Device Drivers --->
 <*> Userspace I/O drivers --->
 <*> Freescale Serial RapidIO support
 [*] Staging drivers --->
 [*] Freescale RapidIO Message Manager support
```

When build rootfs, make sure to define FRA\_FC in fra\_cfg.h.

```
#define FRA_FC
```

## 74.15.5.3 Running FRA

Logging into Linux

At the Linux prompt, login as "root".

```
Yocto (Built by Poky 6.0) 1.1 t4240qds ttyS0

t4240qds login: root
root@t4240qds:~#
```

Configure FMan PCD using fmc with the XML files in /usr/etc:

```
[root@t4240 root]# cd /usr/etc
[root@t4240 root]# fmc -c usdpaa_config_t4_serdes_1_1_6_6.xml -p fra_dstr_fc_policy.xml -a
```

Run the FRA application on left board:

```
root@t4240qds:~# fra -c usdpaa_config_t4_serdes_1_1_6_6.xml -p fra_dstr_fc_policy.xml -f
fra_config_dstr_fc_processing1_left.xml
```

Run the FRA application on right board:

```
root@t4240qds:~# fra -c usdpaa_config_t4_serdes_1_1_6_6.xml -p fra_dstr_fc_policy.xml -f
fra_config_dstr_fc_processing1_right.xml
```

Additional FRA threads can be started on other CPUs by entering commands at the FRA command prompt. Running threads can also be queried and the application can also be shutdown.

- add a FRA thread on a single cpu (e.g. cpu 2)

```
fra> add 2
```

- add FRA threads on a range of cpus

```
fra> add 2..3
```

- list the cpus currently enabled (by querying them, i.e. this also verifies that they aren't blocked)

```
fra> list
Thread alive on cpu 1
Thread alive on cpu 2
Thread alive on cpu 3
```

- display FRA congestion group status

```
fra> cgr
```

- display FRA configuration and status

```
fra> status
RMan configuration:
Create inbound message descriptor: yes
The algorithmic frame queue bits info:
data streaming:2 mailbox:2
BPID info:
data streaming:11 mailbox:11 doorbell:10 sg:12
Use SRIO port 0: using lane 0
Create 3 RX sockets and 12 frame queues
Create 3 TX sockets and 96 frame queues
distribution order-1:
distribution-1-RMAN_TO_FMAN: rman_to_fman0_10gec
rio port:0 - 10gec
```

```

sid:0 mask:255 queue mode:algorithmic
rio_tran:dstr-10gec type:Data-streaming
base FQID:0x1500 count:4 configured to IBCU 0

distribution order-2:
distribution-1-FMAN_TO_RMAN: fman_to_rman_10gec
FMan:10gec - rio port 0
did:1
rio_tran:dstr-10gec type:Data-streaming

distribution order-3:
distribution-1-RMAN_TO_FMAN: rman_to_fman0_dtsec4
rio port:0 - dtsec4
sid:0 mask:255 queue mode:algorithmic
rio_tran:dstr-dtsec4 type:Data-streaming
base FQID:0x1400 count:4 configured to IBCU 1

distribution order-4:
distribution-1-FMAN_TO_RMAN: fman_to_rman_dtsec4
FMan:dtsec4 - rio port 0
did:1
rio_tran:dstr-dtsec4 type:Data-streaming

distribution order-5:
distribution-1-RMAN_TO_FMAN: rman_to_fman0_dtsec3
rio port:0 - dtsec3
sid:0 mask:255 queue mode:algorithmic
rio_tran:dstr-dtsec3 type:Data-streaming
base FQID:0x1300 count:4 configured to IBCU 2

distribution order-6:
distribution-1-FMAN_TO_RMAN: fman_to_rman_dtsec3
FMan:dtsec3 - rio port 0
did:1
rio_tran:dstr-dtsec3 type:Data-streaming
fra>

```

- Enable or disable debug information output

Note: this function is effective only when compiling with ENABLE\_FRA\_DEBUG definition

```
fra> debug [on/off]
```

- If you want to quit FRA , should use the following command, FRA will release the related resource.

```
fra> q
```

## 7.4.15.6 Running FRA on One Board

The chapter is intended to introduce how to run FRA on one board. The operations of installation and compilation are similar to Chapter 3, So those sections please refer to above description.

### 7.4.15.6.1 Prepare the Hardware (one board)

FRA can run on one board P3041DS or P5020DS. As the Fig10 shown, when FRA application run on one board, needs two PC or one PC with two Ethernet interfaces. Two pc's Ethernet interfaces are connected with two RGMI interface of the board respectively. The board should been inserted two RapidIO card to slot6 and slot7. Check board's switch, sw2 is 0b10100101(slot7->SRIO2, slot6->SRIO1, slot3->PEX2, slot2->SGMI), Sw5 is 0b00010100(Serdes reference clock for bank1 is 100MHz, bank2 is 125MHz bank3 is 125MHz).

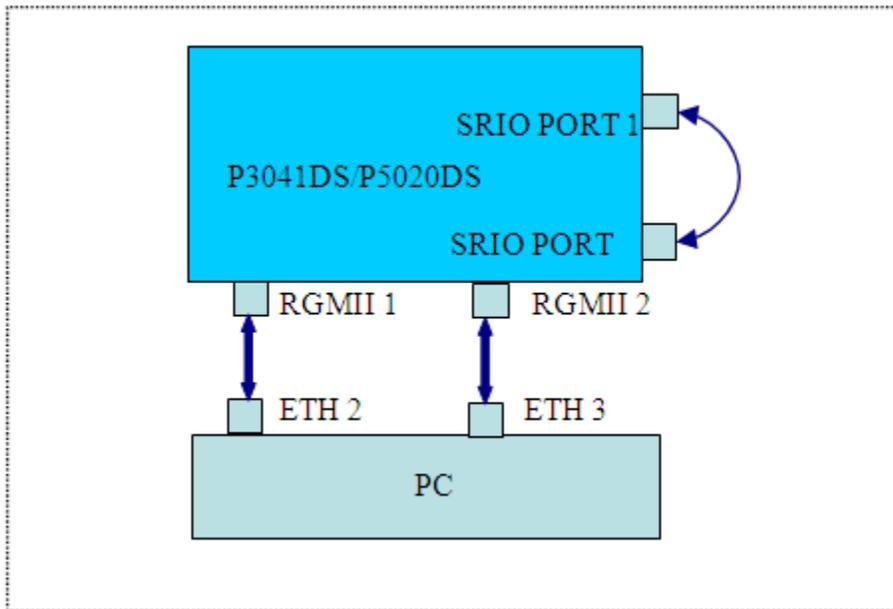


Figure 165. Serdes 0x04 protocol hardware setup

### 7.4.15.6.2 Configuring FRA (one board)

In this case FRA default configuration file is fra\_config\_dstr\_port1\_port2\_loopback.xml. Under This configuration the packet flow is shown as Fig5. Board receives the IP packets from RGMII 1, adds RMan message descriptor to the headroom of each packet according to configuration, and then directly sends them from SRIO port 1; the packets are looped back to SRIO port2. RMan receives those packets, and transfers them to RGMII2. So PC can receive packets which it sends out.

The other configuration files are also valid. The difference with the default configuration is that board will use the same network port for receiving and sending.

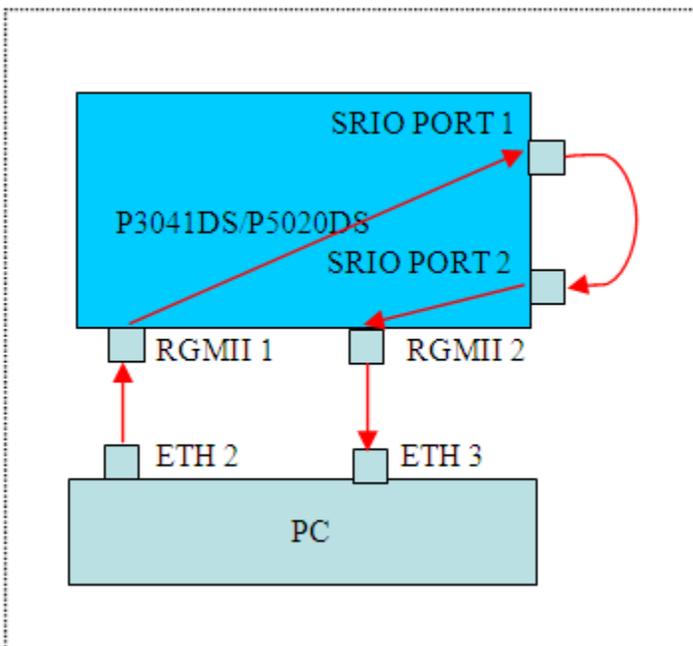


Figure 166. packet flow under port1-port2 loop back mode

When we run FRA on one board, may use the following command:

```
fra -f /usr/etc/fra_config_dstr_port1_port2_loopback.xml
```

### 7.4.15.6.3 Managing RCW

In order to enable board's two SRIO ports, we should select the appropriate RCW setting, As the example this section provides a valid RCW of Serdes 0x04 type for P3041Ds and P5020DS .

RCW for P3041DS is :

```
00000000: AA55 AA55 010E 0100 1260 0000 0000 0000
00000010: 241C 0000 0000 0000 1081 6400 2400 2000
00000020: FE80 0000 4100 0000 0000 0000 0000 0000
00000030: 0000 0000 1007 0000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 2F4C A9FA
```

RCW for P5020DS is :

```
00000000: AA55 AA55 010E 0100 0C54 0000 0000 0000
00000010: 1E12 0000 0000 0000 1081 6400 2400 2000
00000020: FE80 0000 4100 0000 0000 0000 0000 0000
00000030: 0000 0000 1007 0000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 FA0A 674D
```

Save the above RCW data to rcw-0x04.xxd file and then convert to RCW image

```
$ xxd -r rcw-0x04.xxd > rcw-0x04.bin
```

### 7.4.15.6.4 Running FRA

According to Chapter 3 description, apply the related patches; build the software and burn images to board.

Following are some u-boot boot log:

```
Flash: 128 MiB
L2: 128 KB enabled
Corenet Platform Cache: 1024 KB enabled
SRIO1: enabled
SRIO2: enabled
NAND: 1024 MiB
MMC: FSL_ESDHC: 0
```

The above log has printed out SRIO status, if SRIO1 or SRIO2 is not enabled, we should check RCW image and uboot image.

The other output log is same with Chapter 3.

Configure FMan PCD using fmc with the XML files located in /usr/etc:

```
[root@p3041 root]# cd /usr/etc
[root@p3041 root]# fmc -c usdpaa_config_p3_p5_serdes_0x04.xml -p
usdpaa_policy_hash_ipv4.xml -a
```

Run the FRA application:

```
[root@p3041 root]# fra -f /usr/etc/fra_config_dstr_port1_port2_loopback.xml
```

We can use status command to check the FRA configuration:

### 7.4.15.6.5 Testing FRA (one board)

We can configure ETH2's IP address is 192.168.2.1, ETH3's IP address is 192.168.3.1. In order to test FRA, we need to send IP packets to a RGMII Ethernet interface, since FRA does not support ARP protocol, we should do IP and MAC binding on PC. On board console Linux startup log has the MAC information.

```
cpu0: fsl_mac: FSL FMan MAC API based driver ()
cpu0: fsl_mac: ffe4e2000.ethernet: FMan dTSEC version: 0x08240101
cpu0: fsl_mac: ffe4e2000.ethernet: FMan MAC address: 00:e0:0c:00:d7:01
cpu0: fsl_mac: ffe4e4000.ethernet: FMan dTSEC version: 0x08240101
cpu0: fsl_mac: ffe4e4000.ethernet: FMan MAC address: 00:e0:0c:00:d7:02
cpu0: fsl_mac: ffe4e6000.ethernet: FMan dTSEC version: 0x08240101
cpu0: fsl_mac: ffe4e6000.ethernet: FMan MAC address: 00:e0:0c:00:d7:03
cpu0: fsl_mac: ffe4e8000.ethernet: FMan dTSEC version: 0x08240101
cpu0: fsl_mac: ffe4e8000.ethernet: FMan MAC address: 00:e0:0c:00:d7:04
cpu0/0: Applying 10G tx-ecc error workaround (10GMAC-A004) ...
cpu0/0: done.
cpu0: fsl_mac: ffe4f0000.ethernet: FMan XGEC version: 0x00010330
cpu0: fsl_mac: ffe4f0000.ethernet: FMan MAC address: 00:e0:0c:00:d7:05
```

For 0x04 RCW, RGMII1 corresponds to DTSEC3, RGMII2 corresponds to DTSEC4. So we find the RGMII1 port Mac address is 00:e0:0c:00:d7:03, RGMII2 port Mac address is 00:e0:0c:00:d7:04.

Create an ARP entry on pc by using the following command:

```
$ sudo arp -s 192.168.2.2 00:e0:0c:00:d7:03
$ sudo arp -s 192.168.3.2 00:e0:0c:00:d7:04
```

On PC ping host board's DTSEC3

```
$ ping 192.168.2.2
```

Run Wireshark on PC to capture the ETH3 which connected to the board's RGMII2 port.

```
$ sudo wireshark
```

As the figure below shows, we should see the ping packets sent from PC.

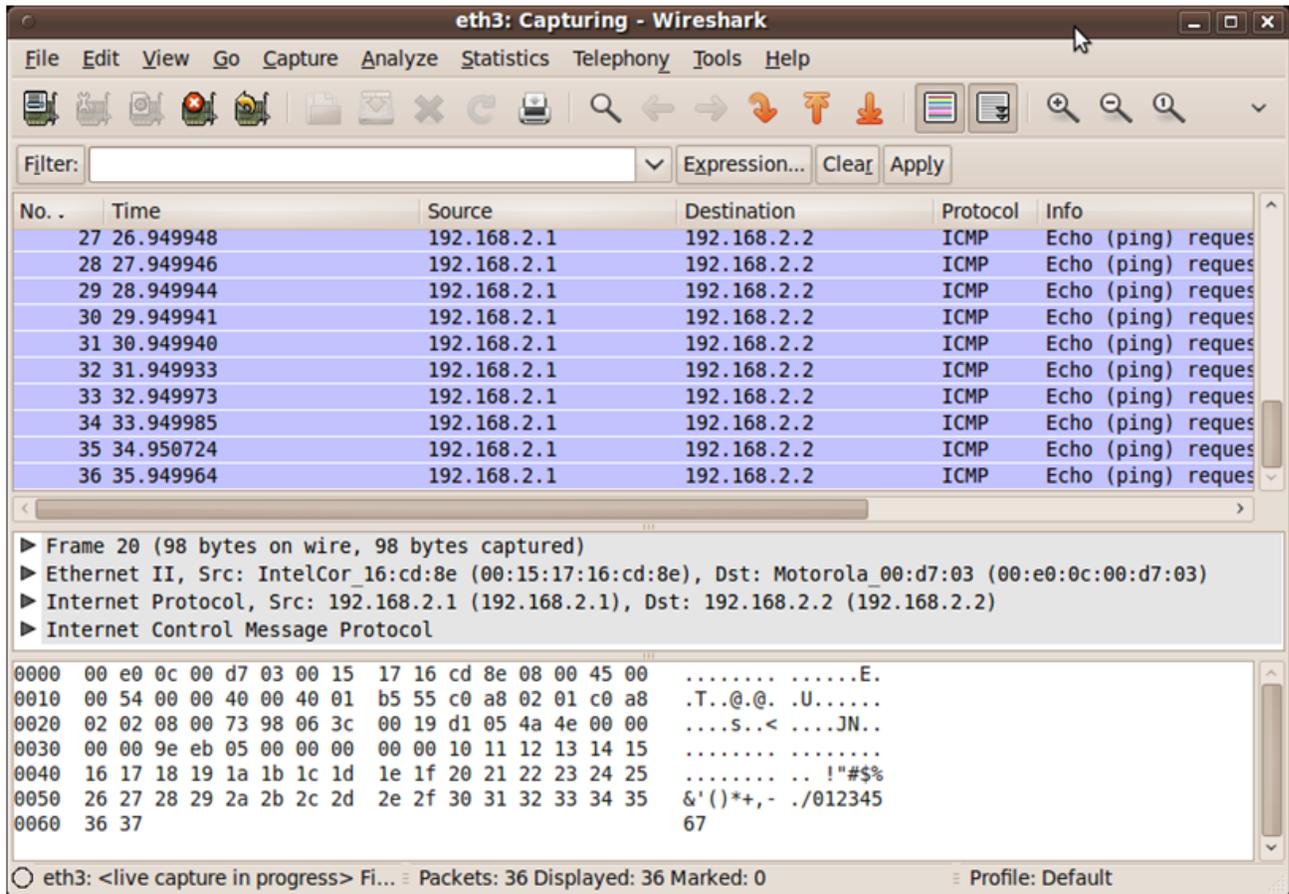


Figure 167. Wireshark Capturing Packets

## 7.4.15.7 Revision History

Table 171. Revision History

Version	Updates
1.1	Add chapter: Overview of FRA Update FRA log
1.0	Creation of FRA UM for sdk 1.1 release. - Basic application flow - Commands to use - Testing FRA application

## 7.4.16 NXP USDPAAs SRA User Manual

### 7.4.16.1 Serial RapidIO application

This User Manual describes the NXP P3041DS/P4080DS/P5020DS/T4240QDS/B4860QDS Serial RapidIO linux user space driver. This user manual provides you with instructions about how to use SRIO (Serial RapidIO) user space driver.

## 7.4.16.1.1 Overview

NXP Serial RapidIO user space driver is based on NXP Linux SDK. It is composed of Linux UIO driver in Linux kernel space and SRIO driver/application in Linux user space. There are dma and srio UIO drivers which map dma/srio registers and SRIO window to linux user space for user space driver usage. User space dma and srio driver provide driver interfaces for application, and timer driver provide means to measure the performance. Application can use driver interface to accomplish the final srio function. Currently, the srio driver support two rapidio ports, and the dma driver supports two controllers and eight channels based on basic direct mode and basic chain mode . The demo sra application can implement SRIO SWRITE, NWRITE, NWRITE\_R, NREAD, ATOMIC\_INC, ATOMIC\_DEC, ATOMIC\_CLR, ATOMIC\_SET type protocols based on window/segment/subsegment, and give user srio performance data to evaluate NXP rapidio IP block. Currently, it can run on P3041DS/P4080DS/P5020DS/T4240QDS/B4860QDS board.

## 7.4.16.1.2 SRA environment setup

### 7.4.16.1.2.1 Hardware environment

To run SRIO application demo, have two P3041DS/P4080DS/P5020DS/T4240QDS/B4860QDS boards connected with each other via SRIO cable through their RapidIO ports, as shown in Figure 1 (SRIO riser card located in slot6 on P3041DS/ P5020DS, and located in slot3 on P4080DS, in slot6 on T4240QDS, in additional AMC2PEX-2S card on B4860QDS). Or using one board, connecting its port1 with port2 via SRIO cable, as shown in Figure 2 (SRIO riser card located in slot6 and slot7 on single P3041DS/ P5020DS). Then burn the updated rcw, u-boot, ucode for Fman to flash to setup board environment.

For a two SRIO port operation on single board, burn rcw-0x04 file (support 4 lanes for each rapidio port) on P3041DS/ P5020DS. For two board SRIO connection, burn rcw-0x33 file on P3041DS/ P5020DS (support 4 lanes for SRIO1), and rcw-0x16 on P4080DS (support 4 lanes for each rapidio port, but only SRIO1 can be connected between two P4080DS with 4 lanes). These RCW can be found in the SDK. P4080DS can also use rcw-0x1d to accomplish two board connection via two RapidIO ports with 1 lane for each port. Since this RCW is not included in SDK, generate it following the later RCW generation guide. For T4240QDS and B4860QDS platforms, you should create the new rcw files using QCS based on your specific environment.

For P3041DS/P5020DS, when using rcw-0x33 for two board connection via SRIO, check the board's switch: sw2 is "0b00100001"(slot7->PEX1, slot6->SRIO, slot4->PEX3, slot2->XAUI), sw5 is "0b00010100" (Serdes reference clock for bank1 is 100MHz, bank2 is 125MHz bank3 is 125MHz). For P3041DS/P5020DS, when using rcw-0x04 for two SRIO port connections on one board, board's switch should be: sw2 is "0b10101001"(slot7->SRIO2, slot6->SRIO1, slot3->PEX2, slot2->SGMI), Sw5 is "0b00010100" (Serdes reference clock for bank1 is 100MHz, bank2 is 125MHz bank3 is 125MHz). For P4080DS, when using rcw-0x16 for two SRIO port connections between two boards, board's switch should be: sw3 is "0b01001100" (Serdes reference clock for bank1 is 125MHz, bank2 is 125MHz, bank3 is 125MHz).

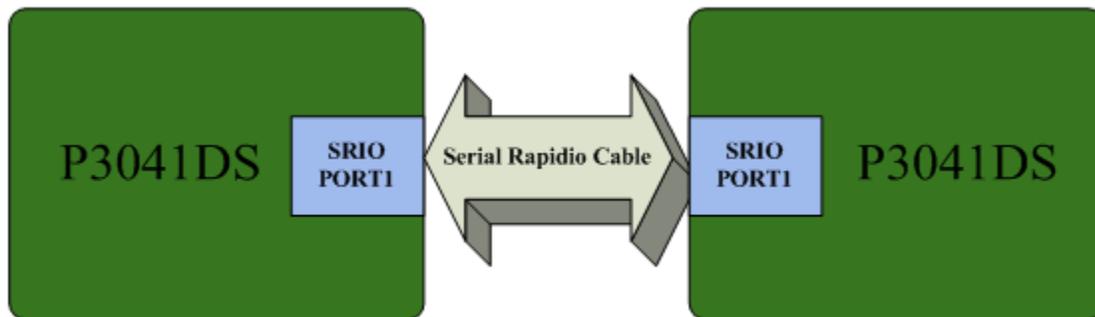


Figure 168. SRIO Hardware Connection between Two Boards

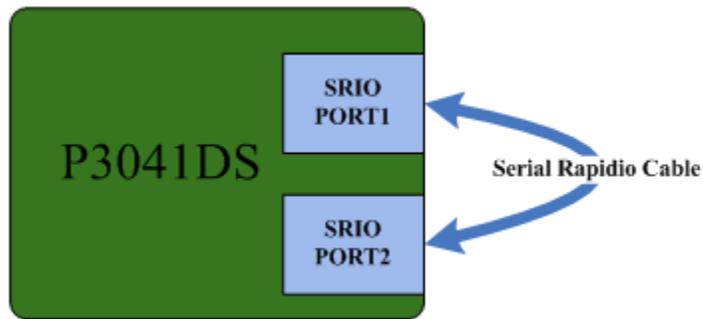


Figure 169. SRIO Hardware Connection between two Ports on One Board

### 7.4.16.1.2.2 SDK Installation

SRA is based on NXP Linux SDK. So to use SRA, make sure NXP Linux SDK is well installed.

### 7.4.16.1.2.3 RCW Generation

For P3041DS and P5020DS platforms, user can use NXP PBL tools to create SerDes 0x04 RCW file (support srio two ports) or SerDes 0x33 RCW file (support srio one port) and then convert to rcw image.

SerDes 0x33 RCW of P3041DS is :

```
0000000: aa55 aa55 010e 0100 1260 0000 0000 0000
0000010: 241c 0000 0000 0000 cc98 4a00 0300 2000
0000020: fe80 0000 4100 0000 0000 0000 0000 0000
0000030: 0000 0000 1007 0000 0000 0000 0000 0000
0000040: 0000 0000 0000 0000 0813 8040 cf8e 2f88
```

SerDes 0x33 RCW of P5020DS is :

```
0000000: aa55 aa55 010e 0100 0c54 0000 0000 0000
0000010: 1e12 0000 0000 0000 cc98 4a00 0300 2000
0000020: fe80 0000 4100 0000 0000 0000 0000 0000
0000030: 0000 0000 1007 0000 0000 0000 0000 0000
0000040: 0000 0000 0000 0000 0813 8040 1ac8 e13f
```

Save the above RCW data to rcw-0x33.xxd file and then convert to RCW image

```
$ xxd -r rcw-0x33.xxd > rcw-0x33.bin
```

SerDes 0x04 RCW for P3041DS is :

```
00000000: aa55 aa55 010e 0100 1260 0000 0000 0000
00000010: 241c 0000 0000 0000 1081 6400 2400 2000
00000020: fe80 0000 4100 0000 0000 0000 0000 0000
00000030: 0000 0000 1007 0000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 2f4c a9fa
```

SerDes 0x04 RCW for P5020DS is :

```
00000000: aa55 aa55 010e 0100 0c54 0000 0000 0000
00000010: 1e12 0000 0000 0000 1081 6400 2400 2000
00000020: fe80 0000 4100 0000 0000 0000 0000 0000
```

Linux User Space  
USDPAA Applications

```
00000030: 0000 0000 1007 0000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 fa0a 674d
```

Save the above RCW data to rcw-0x04.xxd file and then convert to RCW image

```
$ xxd -r rcw-0x04.xxd > rcw-0x04.bin
```

For P4080DS platform, SerDes 0x16 RCW file supports srio two X4 ports; SerDes 0x1d RCW file supports srio two X1 ports.

SerDes 0x16 RCW for P4080DS is :

```
00000000: aa55 aa55 010e 0100 105a 0000 0000 0000
00000010: 1e1e 181e 0000 cccc 5840 0000 3c3c 2000
00000020: fe80 0000 e100 0000 0000 0000 0000 0000
00000030: 0000 0000 008b 6000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 532a bb17
```

Save the above RCW data to rcw-0x16.xxd file and then convert to RCW image

```
$ xxd -r rcw-0x16.xxd > rcw-0x16.bin
```

SerDes 0x1d RCW for P4080DS is :

```
00000000: aa55 aa55 010e 0100 0c58 0000 0000 0000
00000010: 1818 1818 0000 8888 7440 4000 0000 2000
00000020: fe80 0000 0100 0000 0000 0000 0000 0000
00000030: 0000 0000 0083 0000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 0129 56be
```

Save the above RCW data to rcw-0x1d.xxd file and then convert to RCW image

```
$ xxd -r rcw-0x1d.xxd > rcw-0x1d.bin
```

For P2041RDB platform, SerDes 0x02 RCW file supports srio one X4 ports. RapidIO card should be inserted to slot1. and the following commands should be ran first:

```
cpld lane_mux 6 0;
cpld lane_mux a 0;
cpld lane_mux c 0;
cpld lane_mux d 0;
cpld reset altbank;
```

SerDes 0x02 RCW for P2041RDB is :

```
00000000: AA55 AA55 010E 0100 1260 0000 0000 0000
00000010: 241C 0000 0000 0000 0899 30C0 C7C0 2000
00000020: FE80 0000 4000 0000 0000 0000 0000 0000
00000030: 0000 0000 D003 0F07 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 57B0 F7D9
```

Save the above RCW data to rcw-0x02.xxd file and then convert to RCW image

```
$ xxd -r rcw-0x02.xxd > rcw-0x02.bin
```

For T4240QDS and B4860QDS platforms, please select the right RCW files from the released SDK.

u-boot/kernel/sra generation:

Please follow the SDK build guide.

### 74.16.1.2.4 Build Configuration

When build kernel, make sure these options are selected.

```
Device Drivers --->
 <*> Userspace I/O drivers --->
 <*> Freescale Serial RapidIO support
 <*> Freescale DMA support
```

### 74.16.1.3 Boot

To run the USDPAA software, one must first boot Linux with the correct files. These files are including u-boot.bin, ulmage, p3041ds/p4080ds/p5020ds/t4240qds/b4860qds-usdpaa.dtb and initramfs.cpio.gz.uboot. Please use the ucode and rcw file in the SDK package.

Following are some booting information points:

u-boot boot log (rcw-0x04):

```
Flash: 128 MiB
L2: 128 KB enabled
Corenet Platform Cache: 1024 KB enabled
SRIO1: enabled
SRIO2: enabled
NAND: 1024 MiB
```

Linux kernel boot log:

```
cpu2: fsl_oh: FSL FMan Offline Parsing port driver ()
cpu2: fsl_oh: dpa-fman0-oh.32: Found OH node handle compatible with fsl,dpa-oh.
cpu2: fsl_oh: dpa-fman0-oh.32: OH port /soc@ffe000000/fman@400000/port@82000 enabled.
fsl-of-srio ffe0c0000.rapidio: Rapidio UIO driver initialized
fsl-of-dma ffe100300.dma: dma channel dma-uio0-0 initialized
fsl-of-dma ffe100300.dma: dma channel dma-uio0-1 initialized
fsl-of-dma ffe100300.dma: dma channel dma-uio0-2 initialized
fsl-of-dma ffe100300.dma: dma channel dma-uio0-3 initialized
fsl-of-dma ffe101300.dma: dma channel dma-uio1-0 initialized
fsl-of-dma ffe101300.dma: dma channel dma-uio1-1 initialized
fsl-of-dma ffe101300.dma: dma channel dma-uio1-2 initialized
fsl-of-dma ffe101300.dma: dma channel dma-uio1-3 initialized
ehci_hcd: USB 2.0 'Enhanced' Host Controller (EHCI) Driver
ohci_hcd: USB 1.1 'Open' Host Controller (OHCI) Driver
```

### 74.16.1.4 SRA Demo

SRA is a demo application to implement SRIO functions, via dma and srio driver interface. It supports SRIO SWRITE, NWRITE, NWRITE\_R, NREAD, ATOMIC\_INC, ATOMIC\_DEC, ATOMIC\_CLR, ATOMIC\_SET type protocols based on window/segment/subsegment, using command input. It can read data from the other board memory, and write data to the other board memory, then display the memory. It can also evaluate the srio performance, and print out the performance result. These functions are done by dma direct mode operation or core. It also supports dma basic chain mode test and basic direct mode performance test. Currently, it supports two rapidio ports. And you can assign a port to implement rapidio performance test. It will measure the srio performance under different transmission protocol type, with different transmission data size, and with different dma BWC (bandwidth control).

As shown in Figure 3, SRA defines two large areas for two rapidio ports in dma pool. Each large area is 8M bytes. And the large area is divided into four small areas, each of which is 2M bytes. They are:

map space - match data written from other rapidio ports come into this area.

read data space - when read data from other ports, the data is saved in this area.

writing preparing space - this area is preparing data for writing to other rapidio ports.

reserved space - unused.

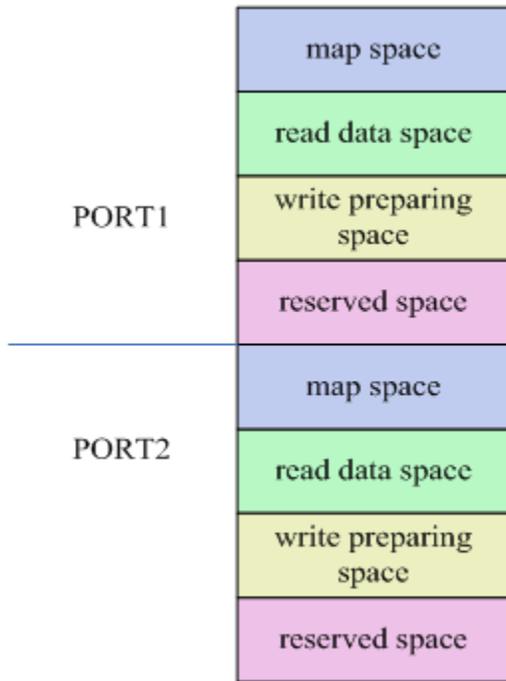


Figure 170. SRA DMA Pool Partition

### 7.4.16.1.5 SRA Command Description

There are sra command formats as followings:

```
sra -attr [port_id] [fun] [fun_id] ([write_attr] [read_attr])

sra -attr port1/2 device_id [id]
sra -attr port1/2 target_id [id]
sra -attr port1/2 seg_num [num]
sra -attr port1/2 subseg_num [num]
sra -attr port1/2 subseg_tdid [seg_id] [tdid]
sra -attr port1/2 accept_all [id]
sra -attr port1/2 irq [id]
sra -attr port1/2 win_attr [id] [write_attr] [read_attr]
sra -attr port1/2 seg_attr [id] [write_attr] [read_attr]
```

Notes: This is sra attribute command serial, which set rapidio attribute.

-attr: It indicates this is an attribute setting command.

port\_id: It indicates which port is set currently. It can be these characters: port1, port2.

fun: It indicates which function this command sets. The function names are listed above.

fun\_id: It indicates which operation this function uses. In most case, it is a number.

write\_attr: This is writing attribute when sra do writing operation. It can be these characters: swrite, nwrite, nwrite, nwrite\_r which are consistent with rapidio standard protocol.

`read_attr`: This is reading attribute when sra do reading operation. It can be these characters: `nread`, `atomic_inc`, `atomic_dec`, `atomic_clr`, `atomic_set`, which are consistent with rapidio standard protocol.

```
sra -op [port_id] [win_id] [seg_id] [subseg_id] [operation] [data_len]
sra -op port1/2 [win_id] [seg_id] [subseg_id] w/r/s/p [data_len]
```

`-op`: It indicates this is an operation command.

`port_id`: It indicates which port is set currently. It can be these characters: `port1`, `port2`.

`win_id`: It indicates which window this operation works on. Currently, application uses window 1 as the only window for operation.

`seg_id`: It indicates which segment this operation works on.

`subseg_id`: It indicates which subsegment this operation works on.

`operation`: It indicates what operation should do.

`s(set)`: It sets 0/1/2/3/4/5/6/7 in different area to local memory and ignore the `data_len` value.

`p(print)`: It prints definite parts of local memory and ignore the `data_len` value.

`w(write)`: It sends data in current port write preparing space to the other board map space in memory according to `data_len`.

`r(read)`: It reads the other board port map space data, and saves them in its read data space in local memory according to `data_len`.

`data_len`: This is the data transmission length. It can be any number which is smaller than 2M. And for ATOMIC operation, the `data_len` should be 1/2/4. Other number using for ATOMIC reading operation will cause transmission fault. The `data_len` will be ignored for the `s(set)` and `p(print)` commands, but when run the `s(set)` and `p(print)` commands you also should give the `data_len` parameter, any value will be ok.

```
sra -test [case] [port] [task] [srio] [number]
sra -test srio port1/2 dma/core [srio_type] payload_size
 [srio_type] should be swrite/nwrite/nwrite_r/nread
 payload_size should be less than 2M bytes
sra -test dma_chain
sra -test show_perf port1/2 times
sra -test show_task
sra -test free_task port1/2
```

`-test`: It indicates sra will do test or create a srio transmission task.

`case`: The type of the test, can be `srio`, `dma_chain`, `show_perf`, `show_task`, or `free_task`. There should be different following parameters for different test case. And also there are some dependent relationships between the cases.

`port`: Indicates which SRIO port will be used for the test, can be `port1` or `port2`.

`task`: Determines the mode of the test, can be `dma` or `core`. If you select the core mode, the transmission will be implemented with `memcpy` function, so you'd better very clear about the effect of the cache. The testing memory space for the srio was configured to be cached by default, so the results of the core test may don't make any sense for the normal requirements.

`srio`: Indicates the type of the srio transmission, can be `swrite`, `nwrite`, `nwrite_r`, or `nread`.

`number`: The payload size of the srio transmission task or the times for a performance calculation.

## 7.4.16.1.6 SRA command Usage

### •SRA Help

```
#sra
sra> sra
```

Then sra help will display. Error command will cause sra to print help information. Followings are the help information:

```
-----SRIO APP CMD FORMAT-----
Set window attribute
sra -attr [port_id] [fun] [fun_id] ([write_attr] [read_attr])
sra -attr port1/2 device_id [id]
sra -attr port1/2 target_id [id]
sra -attr port1/2 seg_num [num]
sra -attr port1/2 subseg_num [num]
sra -attr port1/2 subseg_tdid [seg_id] [tdid]
sra -attr port1/2 accept_all [id]
sra -attr port1/2 irq [id]
sra -attr port1/2 win_attr [id] [write_attr] [read_attr]
sra -attr port1/2 seg_attr [id] [write_attr] [read_attr]

Notes:
 [id] for command accept_all: 0 - disable; 1 - enable
 [id] for irq: 0 - disable; 1 - enable
 [write_attr] : swrite/nwrite/nwrite_r
 [read_attr] :
 nread/atomic_inc/atomic_dec/atomic_set/atomic_clr

Do sra operation
sra -op [port_id] [win_id] [seg_id] [subseg_id] [operation] [data_len]
sra -op port1/2 [win_id] [seg_id] [subseg_id] w/r/s/p [data_len]
 [data_len] : should be less than the window/segment/subsegment's size, max size is 2M
 data_len should be 1/2/4 for ATOMIC operation

Do SRIO test and print performance result
sra -test [case] [port] [task] [srio] [number]
 sra -test srio port1/2 dma/core [srio_type] payload_size
 [srio_type] should be swrite/nwrite/nwrite_r/nread
 payload_size should be less than 2M bytes
 sra -test dma_chain
 sra -test show_perf port1/2 times
 sra -test show_task
 sra -test free_task port1/2

```

#### •SRA Set SRIO Port Device Id

This command sets SRIO port device id. Any package from other srio device will be accepted only when the sender's target id matches the acceptor's device id.(When "accept all" functions is closed)

```
sra> sra -attr port1 device_id 0x55
```

#### •SRA Set SRIO Port Target Id

This command sets SRIO port1 target id.

```
sra> sra -attr port1 target_id 0x33
```

#### •SRA Set SRIO Port Segment Number

This command sets the segment number in a window of a srio port. The number can be 1/2/4.

```
sra> sra -attr port1 seg_num 4
```

#### •SRA Set SRIO Port Sub Segment Number

This command sets the sub segment number in a segment of a srio port. The number can be 1/2/4/8. This command needs the segment to be set firstly.

```
sra> sra -attr port1 subseg_num 4
```

#### •SRA Set SRIO Port Sub Segment base target id

This command sets the sub segment base target id for subsegment serial of a srio port. This command needs srio segment/subsegment number be set firstly. [seg\_id]: It indicates the segment which is used for subsegment setting. [tdid]: subsegment base target id. It should be aligned with the subsegment number.

```
sra> sra -attr port1 subseg_tdid 1 0x40
```

#### •SRA Set SRIO Port enable or disable accept all function

This command can enable or disable srio port accept all package feature. [id]: 0 - disable the feature; 1 - enable the feature.

```
sra> sra -attr port1 accept_all 1
```

#### •SRA Set SRIO enable or disable irq

This command can enable or disable srio irq. [id]: 0 - disable irq; 1 - enable irq.

```
sra> sra -attr port1 irq 1
```

#### •SRA Set SRIO Port Window Attributes

This command sets SRIO port outbound window attributes. The outbound window attribute for write can be nwrite, swrite, nwrite\_r, and for read can be nread, atomic\_inc, atomic\_dec, atomic\_set, atomic\_clr. [id]: indicates window id. Currently, this parameter should 1, indicating sra setting window 1.

```
sra> sra -attr port1 1 nwrite nread
```

#### •SRA Set SRIO Port Window1 Segment Attributes

This command sets SRIO port1/2 window1 segment attribute. The outbound window segment attribute for write can be nwrite, swrite, nwrite\_r, and for read can be nread, atomic\_inc, atomic\_dec, atomic\_set, atomic\_clr. [id]: indicates window1 segment id. Currently, this parameter should be compatible with segment number. Ex. if segment number is 4, the segment id can be 1/2/3/4.

#### •SRA Setting and Printing Operation

This command sets 0/1/2/3/4/5/6/7 to the eight areas in local memory. [win\_id] indicates the window id. [seg\_id] indicates the segment id. [subseg\_id] indicates the subsegment id.

```
sra> sra -op port1 1 0 0 s 0x100000
```

This command prints parts of local memory data.

```
sra> sra -op port1 1 0 0 p 0x100000
```

#### •SRA Writing Operation

This command sends 1M bytes data in port writing preparing space to other board memory, using port attribute set in attribute command.

```
sra> sra -op port1 1 0 0 w 0x100000
```

#### •SRA Reading Operation

This command uses port attribute to read 1M byte data from other port, and stores the data in port read data space in local memory.

Notes: ATOMIC operation requires data length should be 1/2/4.

```
sra> sra -op port2 1 0 0 r 0x100000
```

### •SRA Test Operation

This command uses srio port1 or port2 to implement rapidio performance test, or do dma basic chain mode test based on the [case] parameter input. In rapidio performance test, if you just give the parameters [case], [port], [task], it will measure the srio performance under different transmission protocol type, with different transmission data size, and with different dma BWC (bandwidth control), and print out the performance result. If you give all the parameters [case], [port], [task], [srio], [number], it will create a task for the specific srio transmission performance test. When you want to get the performance result from the task, you can start the calculation by the other command. In dma basic chain mode test, sra will copy lower region data to the upper region.

### •SRA Test under dma mode

It will measure the srio port1 performance using dma under different transmission protocol type with different transmission data size, and with different dma BWC (bandwidth control). It will print out the performance result finally.

```
sra> sra -test srio port1 dma
```

### •SRA Test under core mode

It will measure the srio port1 performance using core under different transmission protocol type with different transmission data size. It will print out the performance result finally.

Note: The cache will impact the results of the performance.

```
sra> sra -test srio port1 core
```

### •SRA Test to create a transmission task with dma mode

It will create a srio transmission task with the port1, dma mode, the swrite protocol type, and the 4096 bytes payload size. It can print out the performance result for the specific transmission when start the performance calculation with the other command.

Note: The payload size should be less than 2M bytes.

```
sra> sra -test srio port1 dma swrite 4096
```

### •SRA Test to create a transmission task with core mode

It will create a srio transmission task with the port2, core mode, the nread protocol type, and the 2048 bytes payload size. It can print out the performance result for the specific transmission when start the performance calculation with the other command.

Note: The payload size should be less than 2M bytes, and the cache will impact the results of the performance.

```
sra> sra -test srio port2 core nread 2048
```

### •SRA Test for dma chain

The sra will implement the dma chain mode test.

```
sra> sra -test dma_chain
```

### •SRA Test to start the task's performance calculation

When there is a srio task on port1, this command can start the performance calculation with the 10000 transmission times. And it will print out the performance result finally.

```
sra> sra -test show_perf port1 10000
```

●SRA Test to show the existing tasks

When there are srio tasks, this command can show the all tasks on port1 and port2, and print out the task's parameters.

```
sra> sra -test show_task
```

●SRA Test to free the task

When there is a srio task on port1, this command can free the task.

```
sra> sra -test free_task port1
```

### 7.4.16.1.7 Run SRA Demo

Example 1: Board A sends 1M byte data to board B memory via rapidio NWRITE protocol. Two boards are connected via board A port1 and board B port1.

The figure below describes the two board memory map. Board A copies data from its port1 write preparing space to outbound window via dma. And board A outbound window is mapped to board B port1 map space via rapidio system. So writing to board A outbound window just like writing to board B ddr. Following is the command operation steps.

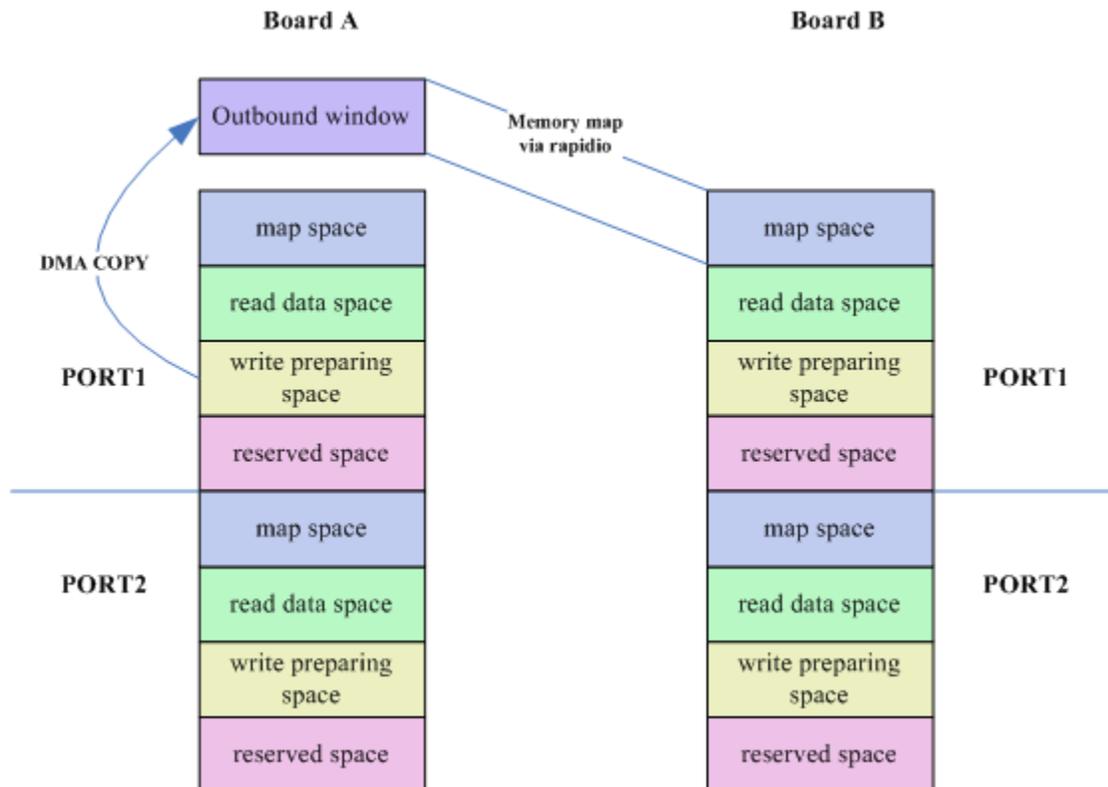


Figure 171. SRIO NWRITE Operation with Two Boards

```
Step1: Boot up two boards.
Step2:
```

```

Board B:
sra> sra -attr port1 win_attr 1 nwrite nread (set board B port1 window 1 attribute to nwrite,
nread)
sra> sra -op port1 1 0 0 s 0x100000 (set local memory to predefined data)
sra> sra -op port1 1 0 0 p 0x100000 (to see the memory data)
Step3:
Board A:
sra> sra -attr port1 win_attr 1 nwrite nread (set board A port1 outbound window with nwrite,
nread attribute)
sra> sra -op port1 1 0 0 s 0x100000 (set local memory to predefined data)
sra> sra -op port1 1 0 0 p 0x100000 (to see whether memory is set)
sra> sra -op port1 1 0 0 w 0x100000 (send 1M data in writing preparing space to board B map space)
Step4:
Board B:
sra> sra -op port1 1 0 0 p 0x100000 (To see whether board A write preparing data are written to
board B map space)

```

Example 2: Board A NREAD 1M byte data from Board B memory. Two boards are connected via board A port1 and board B port2.

The figure below describes the two board memory maps. Board A copies data from its port1 outbound window to its read data space via dma. And board A port1 outbound window is mapped to board B port2 map space via rapidio system. So reading from board A outbound window just like reading from board B ddr. Following is the command operation steps.

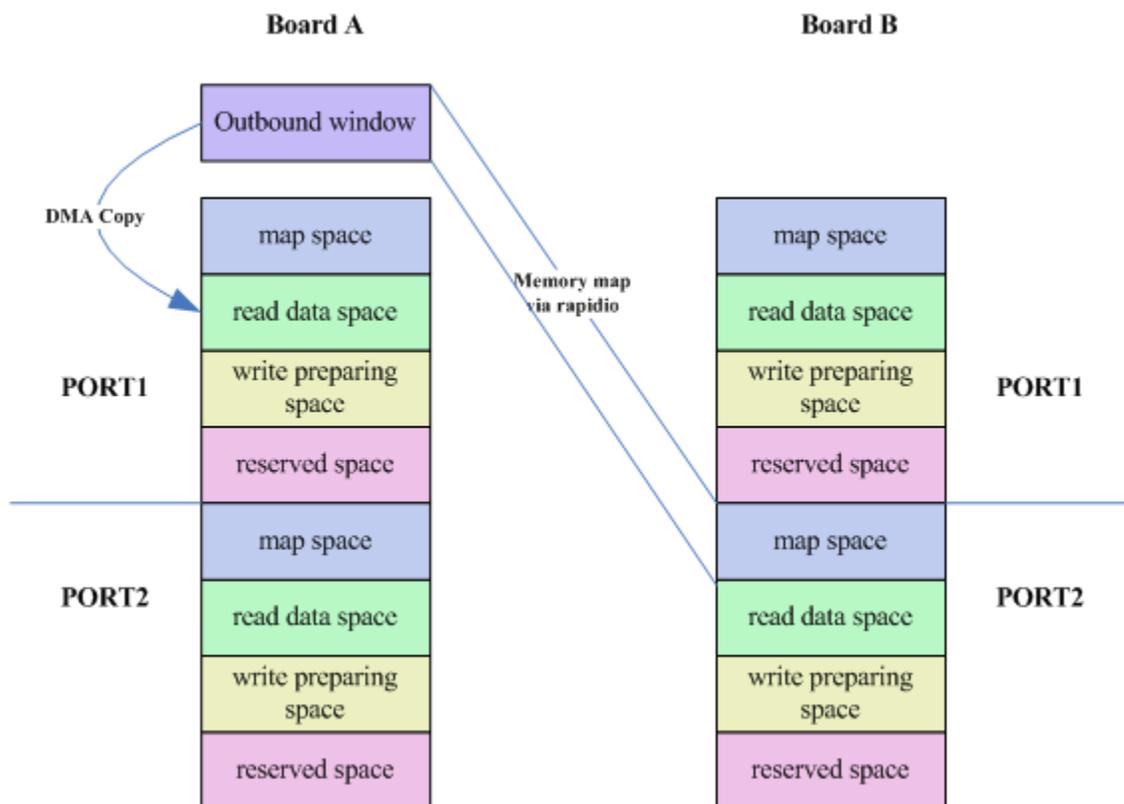


Figure 172. SRIO NREAD Operation with Two Boards

```

Step1: Boot up the two boards.
Step2:
Board B:
sra> sra -attr port2 win_attr 1 nwrite nread (set board B port2 outbound window attribute to

```

```
nwrite, nread)
sra> sra -op port2 1 0 0 s 0x100000 (set local memory to predefined data)
sra> sra -op port2 1 0 0 p 0x100000 (to see the memory data)
Step3:
Board A:
sra> sra -attr port1 win_attr 1 nwrite nread (set board A port1 outbound window attribute to
nwrite, nread)
sra> sra -op port1 1 0 0 s 0x100000 (set local memory to predefined data)
sra> sra -op port1 1 0 0 p 0x100000 (to see whether memory is set)
sra> sra -op port1 1 0 0 r 0x100000 (read 1M data from board B map space, and save them in its
read data space)
sra> sra -op port1 1 0 0 p 0x100000 (to see whether board B map space data is saved in its port1
read data space)
```

**Example 3: Board A ATOMIC\_INC read 4 byte from Board B memory. Two boards are connected via board A port1 and board B port1.**

```
Step1: Boot up the two boards.
Step2:
Board B:
sra> sra -attr port1 win_attr 1 nwrite nread (set board B port1 outbound window attribute to
nwrite, nread)
sra> sra -op port1 1 0 0 s 0x100000 (set local memory to predefined data)
sra> sra -op port1 1 0 0 p 0x100000 (to see the memory data)
Step3:
Board A:
sra> sra -attr port1 win_attr 1 nwrite atomic_inc (set board A port1 outbound window attribute to
nwrite, atomic_inc)
sra> sra -op port1 1 0 0 s 0x100000 (set local memory to predefined data)
sra> sra -op port1 1 0 0 p 0x100000 (to see whether the data is set)
sra> sra -op port1 1 0 0 r 0x4 (read 4 byte data from board B map space, and save them in its
read data space)
sra> sra -op port1 1 0 0 p 0x100000 (to see whether read data is correct)
Step4:
Board B:
sra> sra -op port1 1 0 0 p 0x100000 (to see whether Board B data is added by 1)
```

**Example 4: As shown in the figure below, on one board, using NREAD protocol to read 1M byte data from its port2 map space, and then store data into its port1 read data space. In this scenario, user can test srio two port functions on one board. It needs connecting two srio ports on P3041 board via srio cable.**

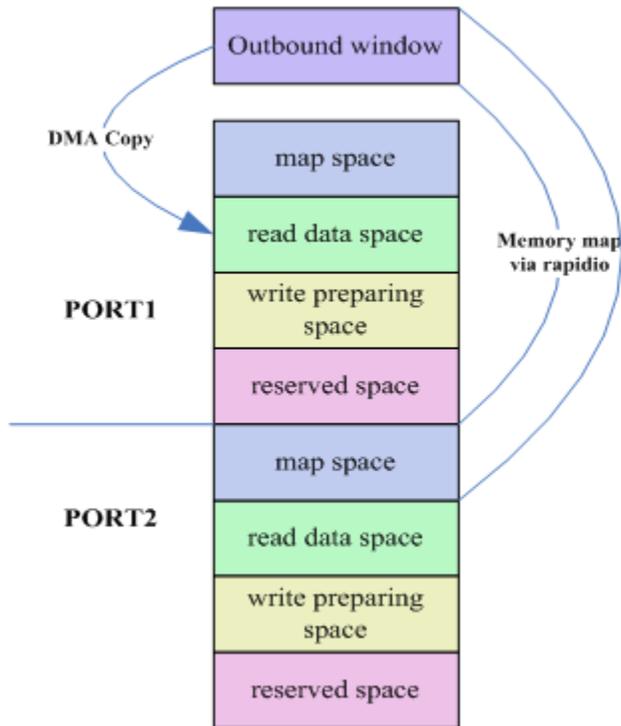


Figure 173. SRIO NREAD Operation with Two Ports on One Board

```

Step1: Boot up the board.
Step2:
sra> sra -attr port1 win_attr 1 nwrite nread (set port1 outbound window attribute to nwrite,
nread)
sra> sra -attr port2 win_attr 1 nwrite nread (set port2 outbound window attribute to nwrite,
nread)
sra> sra -op port1 1 0 0 s 0x100000 (set local memory to predefined data)
sra> sra -op port1 1 0 0 p 0x100000 (to see the memory data)
sra> sra -op port1 1 0 0 r 0x100000 (port1 reads 1M byte data from port2 map space, and save them
in port1's read data space)
sra> sra -op port1 1 0 0 p 0x100000 (to see whether read data is correct)

```

**NOTE**

Currently, SRA is used to demonstrate IO related transaction functionalities, not include functionalities of Message unit and RMan, and not support Doorbell Mailbox Data-streaming transactions.

## 7.4.16.2 Revision History

Document revision history.

Table 172. Revision History

Version	Author	Description

## 7.4.17 USDPAA RMU User Manual

### 7.4.17.1 RapidIO Message Unit Application

This User Manual describes the NXP RapidIO Message Unit Linux user space driver, provides you with instructions about how to use RMU user space application. This user manual provides you with instructions about how to use RMU user space application.

### 7.4.17.2 Overview

NXP RapidIO Message Unit user space driver is implemented based on NXP Linux SDK. It is composed of Linux UIO driver in Linux kernel space and RMU driver/application in Linux user space. There is RMU UIO driver which maps RMU registers to Linux user space for user space driver. User space RMU driver provides interfaces for application, and application accomplishes the final RMU function. Currently, the RMU driver support two message units and one doorbell unit. The demo RMU application can implement RapidIO message and doorbell type protocols, and give performance data to evaluate NXP RMU IP block.

### 7.4.17.3 RMU Environment Setup

To run the RMU application, user needs to build an appropriate environment based on the requirements of hardware, software, RCW and other configurations.

#### 7.4.17.3.1 Hardware Environment

To run RMU application demo, user needs two P4080DS boards connected with each other via SRIO cable as shown in Figure 1 (SRIO card locates in slot3 on P4080DS). Then update the RCW, U-Boot, Ucode to setup board environment.

For the RCW, user should burn RCW-0x16 on P4080DS (support 4 lanes for each RapidIO port, but only SRIO1 can be connected between two P4080DS boards with 4 lanes). The RCW can be found in SDK. P4080DS can also use RCW-0x1d to accomplish two boards connection via two rapidio 1x ports. This RCW is not included in SDK, user needs to generate it following the later RCW generation guide.

For P4080DS, when using RCW-0x16 for two SRIO ports connection between two boards, board's switch should be: sw3 is "0b01001100" (Serdes reference clock for bank1 is 125MHz, bank2 is 125MHz, bank3 is 125MHz).

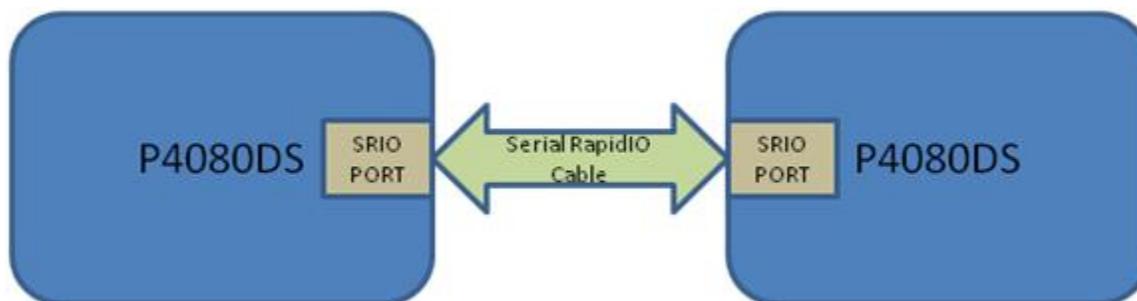


Figure 174. SRIO Hardware Connection between Two Boards

#### 7.4.17.3.2 SDK Installation

RMU is based on NXP Linux SDK. So to use RMU, make sure NXP Linux SDK is well installed.

#### 7.4.17.3.3 RCW Generation

User can use the released RCW binaries in SDK or to create the specific RCW by NXP PBL tool.

serdes 0x16 RCW for P4080DS is :

```
00000000: aa55 aa55 010e 0100 105a 0000 0000 0000
00000010: 1e1e 181e 0000 cccc 5840 0000 3c3c 2000
00000020: fe80 0000 e100 0000 0000 0000 0000 0000
00000030: 0000 0000 008b 6000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 532a bb17
```

Save the above RCW data to rcw-0x16.xxd file and then convert to RCW image

```
$ xxd -r rcw-0x16.xxd > rcw-0x16.bin
```

serdes 0x1d RCW for P4080DS is :

```
00000000: aa55 aa55 010e 0100 0c58 0000 0000 0000
00000010: 1818 1818 0000 8888 7440 4000 0000 2000
00000020: fe80 0000 0100 0000 0000 0000 0000 0000
00000030: 0000 0000 0083 0000 0000 0000 0000 0000
00000040: 0000 0000 0000 0000 0813 8040 0129 56be
```

Save the above RCW data to rcw-0x1d.xxd file and then convert to RCW image

```
$ xxd -r rcw-0x1d.xxd > rcw-0x1d.bin
```

## 74.17.3.4 Kernel Building Configuration

When building kernel, make sure these options are selected.

```
Device Drivers --->
 <*> Userspace I/O drivers --->
 <*> Freescale Rapidio Message Unit support
```

## 74.17.4 Boot

To run the RMU application, one must first boot Linux with the correct files. They include u-boot.bin, ulmage, p4080ds-usdpaa.dtb and initramfs.cpio.gz.uboot. And please use the ucode and RCW file in the SDK package.

Following are some booting information points:

u-boot boot log:

```
POST memory PASSED
Flash: 128 MiB
L2: 128 KB enabled
Corenet Platform Cache: 2048 KB enabled
SRIO1: enabled
SRIO2: enabled
MMC: FSL_SDHC: 0
```

Linux kernel boot log:

```
fsl-of-rmu ffe0d3000.rmu: rmu unit rmu-uio-msg0 initialized
fsl-of-rmu ffe0d3000.rmu: rmu unit rmu-uio-msg1 initialized
fsl-of-rmu ffe0d3000.rmu: rmu unit rmu-uio-doorbell initialized
```

## 7.4.17.5 RMU Demo

RMU is a demo application to implement NXP RapidIO message and doorbell functions. It can send message or doorbell package to the partner board, then the partner can display the received package. It can also evaluate the message and doorbell unit performance, and print out the result. Currently, it supports two message units and one doorbell unit, and uses message unit 0 and SRIO port 0 to implement performance tests. It will measure the message unit performance with different transmission data size.

As shown in Figure 1, RMU defines three areas for each message unit in DMA pool. They are:

Message Tx description map – all 32 entries for the message unit Tx description ring.

Message Tx buffer map – each buffer of the 32 entries loads the Tx data of message package.

Message Rx buffer map – every received message package will be saved to each of the 32 entries buffer.

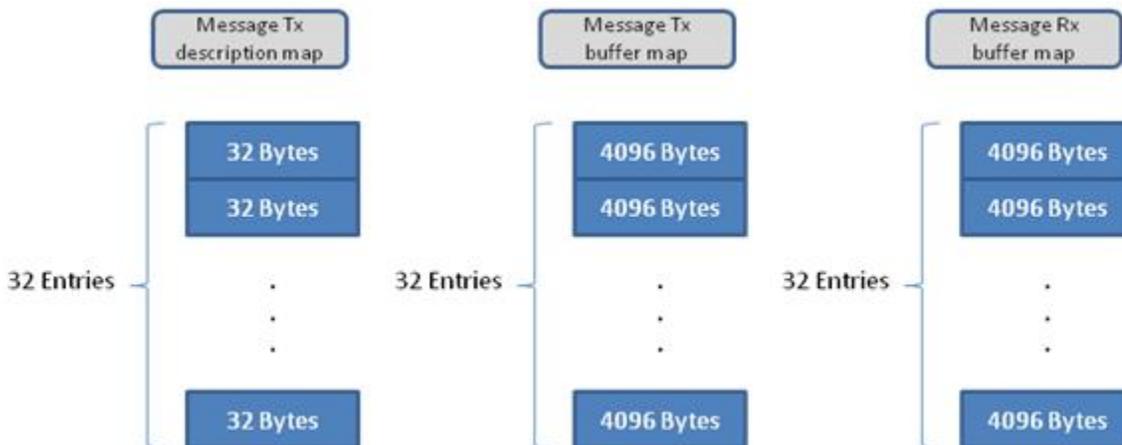


Figure 175. RMU each message unit DMA Pool Partition

The doorbell unit just needs one area in DMA pool for the received package, as shown in Figure 2.

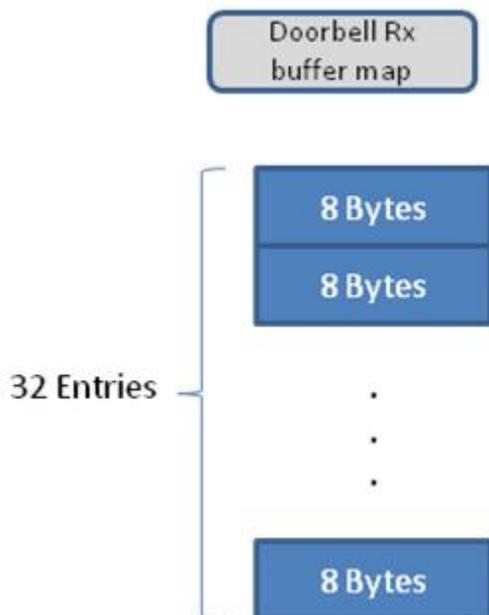


Figure 176. RMU doorbell unit DMA Pool Partition

## 7.4.17.6 RMU Commands

There are RMU command formats as followings. It will also display when you give the command “rmu help” or give a wrong command.

```
-----RMU APP CMD FORMAT-----
--Set Rmu Attribute--
rmu -attr [unit_name] [fun] [slot]
rmu -attr msg0/msg1/dbell txdesc [slot]
rmu -attr msg0/msg1/dbell txbuff [slot]
rmu -attr msg0/msg1/dbell rxbuff [slot]
 -[unit_name]: msg0/msg1/dbell
 -[slot]: the slot of the buffer in the ring

--Do Rmu Operation--
rmu -op [unit_name] [operation] [port_id] [dest_id] [dest_mbox] [priority] [data] [data_len]
rmu -op msg0/msg1/dbell t/r/ar/s/p [port_id] [dest_id] [dest_mbox] [priority] [data] [data_len]
 -[operation]: t---tx a message or doorbell
 r---rx a message or doorbell and print
 ar---create a rx thread for the unit and print
 s---set next tx message buffer
 p---print next tx local buffer
 -[data]: for message should be u8 size
 for dbell should be u16 size
 -[data_len]: for message should be 8~4096 bytes
 for dbell should be equal 2 bytes

--Do Rmu Test and Print Performance Result--
rmu -test [case_name]
 -[case_name]: msg/dbell

```

As shown in the above print information, RMU commands consist of three formats:

**-attr: Attribute printing serial. Print the buffers of the message or doorbell unit.**

```
rmu -attr [unit_name] [fun] [slot]

 unit_name: which message or doorbell unit will be operated
 valid characters: msg0, msg1 or dbell

 fun: which buffer will be printed
 valid characters: txdesc, txbuff, rxbuff
 Note: The doorbell unit just has rxbuff.

 slot: which entry of the buffer in the 32 entries ring will be printed
 valid value: 0 ~ 31. Over 31 will print all the 32 entries

Example:

● Print the message unit 0 Tx description slot 0.

Command:
rmu> rmu -attr msg0 txdesc 0

Display:
ATTR: Msg0 Tx Desc
 cell size:32bytes; entries:32
 base virt addr:1000000; base phy addr:27000000
```

```
desc 0; virt addr:1000000; phy addr:27000000
01000000: 00000000 00000000 00000000 00000000
01000010: 00000000 00000000 00000000 00000000
```

**-op: Operate command serial. Accomplish sending, receiving, setting and other operations.**

```
rmu -op [unit_name] [operation] [port_id] [dest_id] [dest_mbox] [priority] [data] [data_len]
```

unit\_name: which message or doorbell unit will be operated  
valid characters: msg0, msg1 or dbell

operation: which operation will be implemented  
valid characters:  
t - send a message or doorbell package  
r - print the valid received package  
ar - create a receive thread, it can receive and print the packages

continually

s - set next local Tx message buffer with a specific value  
p - print next local Tx message buffer to verify the sending value

Note: For each of the message units and doorbell unit, it has a 32 entries Rx buffer ring for all the received packages. Once received a package, it will occupy a Rx buffer in the ring, the buffer will be released and can be re-used after the reading command. If no "r" or "ar" commands to release the used Rx buffers, the message or doorbell unit can just receive 32 packages because of the 32 entries of the Rx buffer ring.

port\_id: which SRIO port the package will be sent through  
valid value: 0, 1  
Note: The used port must be enabled in RCW.

dest\_id: the device ID of the partner. Generally set it to 0

dest\_mbox: which mailbox of the partner will receive this package  
Valid value: 0, 1

priority: the priority of the sending package  
valid value: 0, 1, 2  
Note: 0 is the lowest and the 2 is the highest.

data: the value should be set or sent  
valid value: for message should be u8 size, for dbell should be u16 size

data\_len: how many bytes will be set or sent for the message package  
Note: Doorbell operations don't need this parameter.

Note: Different operation command needs different number of parameters! Followings are all the "-op" commands based on the msg0 and dbell.

Example:

- Send a message with msg0 to the partner's mailbox.

Command:

```
rmu> rmu -op msg0 t 0 0 0 0 128
```

-op: operation command  
msg0: [unit\_name] - operate message unit 0  
t: [operation] - send a message package

```
0: [port_id] - through the local SRIO port 0
0: [dest_id] - the destination device ID is 0
0: [dest_mbox] - send this package to the partner's mailbox 0
0: [priority] - set the sending package's priority to 0
128: [data_len] - The package payloads 128 bytes. It must be 8 ~ 4096 size.
```

Note: This command will send the data in the next msg0 Tx buffer. So if want to send a specific value, you need to accomplish "s" command first to set the Tx buffer.

- Print the next valid received message package for the msg0.

Command:

```
rmu> rmu -op msg0 r
```

```
-op: operation command
msg0: [unit_name] - operate message unit 0
r: [operation] - Find the next available received message package and print
```

Note: If there is no available package in the Rx buffer ring, following information will display:

```
OP: next available rx message for msg0:
 message fetch failed!
```

- Create a receive thread for the msg0.

Command:

```
rmu> rmu -op msg0 ar 1
```

```
-op: operation command
msg0: [unit_name] - operate message unit 0
ar: [operation] - Create a receive thread for the msg0
1: [data] - 1: create the receive thread
 0: release the thread
```

Note: If create the receive thread successfully, you can see the following information:

```
OP: msg0 rx thread ...create success!
```

- Set next local Tx buffer for the msg0.

Command:

```
rmu> rmu -op msg0 s 0x5a 4096
```

```
-op: operation command
msg0: [unit_name] - operate message unit 0
s: [operation] - Set a specific value to the next local Tx buffer of the msg0
0x5a: [data] - the value to be set
4096: [data_len] - set 4096 bytes
```

Note: For the message Tx buffer, [data] must be one byte size. The size of the Tx buffer is 4096 bytes, so the max length you can set is 4096. There will be the following information after the command:

```
OP: msg0 NEXT tx buffer SET, slot:0
 txbuff 0; virt addr:1020000; phy addr:27020000
 Slot 0 tx buff set down.
```

- Print the next local Tx buffer of the msg0 to verify the sending value.

Command:

```
rmu> rmu -op msg0 p 64
```

```
-op: operation command
msg0: [unit_name] - operate message unit 0
p: [operation] - print the next local Tx buffer with specific printing size.
64: [data_len] - just print 64 bytes of the Tx buffer.
```

Note: The max size of the Tx buffer is 4096, so the [data\_len] should be equal or smaller than 4096. The following information will display after the command:

```
OP: msg0 NEXT tx buffer PRINT, slot:1
 txbuff 1; virt addr:1021000; phy addr:27021000
 01021000: 00000000 00000000 00000000 00000000
 01021010: 00000000 00000000 00000000 00000000
 01021020: 00000000 00000000 00000000 00000000
 01021030: 00000000 00000000 00000000 00000000
```

- Send a doorbell package to the partner.

Command:

```
rmu> rmu -op dbell t 0 0 0 0x5a5a
```

```
-op: operation command
dbell: [unit_name] - operate doorbell unit
t: [operation] - send a doorbell package
0: [port_id] - through the local SRIO port 0
0: [dest_id] - the destination device ID is 0
0: [priority] - set the sending package's priority to 0
0x5a5a: [data] - the value to be doorbell package.
```

Note: The payload of the doorbell package must be 2 bytes.

- Print the next valid received doorbell package for the dbell.

Command:

```
rmu> rmu -op dbell r
```

- Create a receive thread for the dbell.

Command:

```
rmu> rmu -op dbell ar 1
```

Note: Dbell has no the "s" and "p" commands because it does not need the Tx buffers.

### -test: Message unit or doorbell unit performance test commands.

```
rmu -test [case_name]
```

```
case_name: which performance test will be done
valid characters: msg, dbell
```

Note: It will use the msg0 for the message unit test. In order to avoid running out of the partner's Rx buffer, it's recommended to open the Rx thread in partner's board using the "ar" command.

Example:

● Message unit performance test.

Command:

```
rmu> rmu -test msg
```

The following information will display:

TEST: msg0 performance test .....

txbuff 0; virt addr:1020000; phy addr:27020000

Slot 0 tx buff set down.

length(byte): 8	time(us): 2.034187	avg Gb/s: 0.031462	max Gb/s:
0.032282			
length(byte): 16	time(us): 1.994464	avg Gb/s: 0.064178	max Gb/s:
0.064267			
length(byte): 32	time(us): 2.004464	avg Gb/s: 0.127715	max Gb/s:
0.127892			
length(byte): 64	time(us): 2.305023	avg Gb/s: 0.222124	max Gb/s:
0.222445			
length(byte): 128	time(us): 2.610026	avg Gb/s: 0.392333	max Gb/s:
0.392584			
length(byte): 256	time(us): 3.219476	avg Gb/s: 0.636128	max Gb/s:
0.636348			
length(byte): 512	time(us): 4.136708	avg Gb/s: 0.990159	max Gb/s:
0.990558			
length(byte): 1024	time(us): 6.567287	avg Gb/s: 1.247395	max Gb/s:
1.247817			
length(byte): 2048	time(us): 10.818163	avg Gb/s: 1.514490	max Gb/s:
1.514918			
length(byte): 4096	time(us): 19.619639	avg Gb/s: 1.670163	max Gb/s:
1.670400			

● Doorbell unit performance test.

Command:

```
rmu> rmu -test dbell
```

The following information will display:

TEST: dbell performance test .....

length(byte): 2	time(us): 1.361402	avg Gb/s: 0.011753	max Gb/s:
0.012175			

## 7.4.17.7 Run RMU Demo

To run the RMU application, user needs to construct the environment based on the above information.

After the board get the RMU application binary, user should first run the RMU to initialize the two message units and one doorbell unit. If success, the following information will be printed:

```
root@p4080ds:~# ./rmu
RMU: msg0 uio initialized.
 -msg0desc ...
 cell size:32bytes; entries:32
 base virt addr:1000000; base phy addr:27000000
 -msg0txbuff ...
 cell size:4096bytes; entries:32
 base virt addr:1020000; base phy addr:27020000
 -msg0rxbuff ...
```

```
 cell size:4096bytes; entries:32
 base virt addr:1040000; base phy addr:27040000
RMU: msg1 uio initialized.
-msg1desc ...
 cell size:32bytes; entries:32
 base virt addr:1000400; base phy addr:27000400
-msg1txbuff ...
 cell size:4096bytes; entries:32
 base virt addr:1060000; base phy addr:27060000
-msg1rxbuff ...
 cell size:4096bytes; entries:32
 base virt addr:1080000; base phy addr:27080000
RMU: doorbell uio initialized.
-dbellrxbuff ...
 cell size:8bytes; entries:32
 base virt addr:1000800; base phy addr:27000800
rmu>
```

Then you can implement the RMU application commands under the “rmu>” prompt.

## 7.4.18 USDPAA SRIO IPsec Offload User Manual

### 7.4.18.1 Introduction

This document describes the usage of "srio\_ipsec\_offload" application which is built on USDPAA PPAC architecture. User can experience the functionality of using SRIO (Serial RapidIO) and RMAN with this application. In this application, it shows how to use DPAA technology to implement high performance autonomous ipsec with sRIO-Ethernet connection.

The concept of RMAN is beyond scope of this document. The user should refer to RMAN user manual for this part.

### 7.4.18.2 Overview of srio\_ipsec\_offload demo

This application is based on ipsec\_offload(eth-eth) to support srio-eth connection. In this way, it shows how to use DPAA technology to implement high performance autonomous ipsec between srio and ethernet. It mainly supports below configuration:

- IPv4 autonomous /non-autonomous
- ESP tunnel mode
- Fragmentation & reassembly
- SRIO Msg Type9

### 7.4.18.2.1 Srio\_IPSec\_offload outbound flows

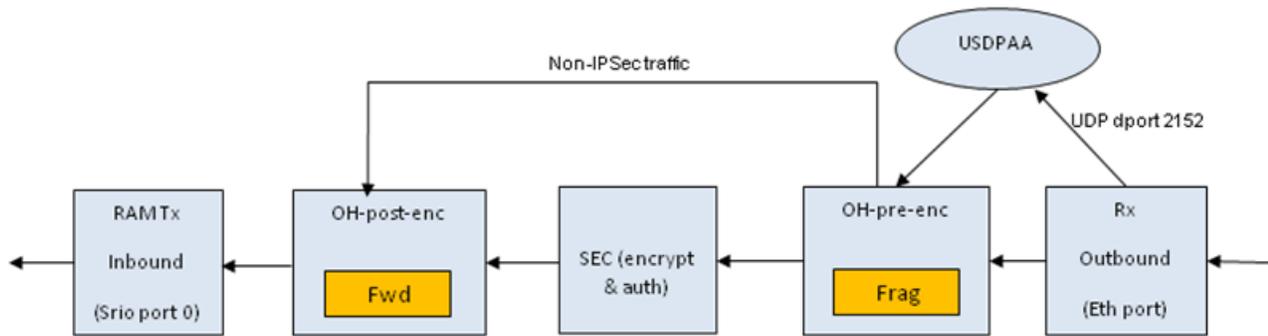


Figure 1 Outbound processing

Figure 177. Outbound processing

As srio\_ipsec\_offload application is based on ipsec\_offload, regarding outbound processing, Please refer to IPsec\_offload outbound flows. The main difference in srio\_ipsec\_offload is that encrypted packets from post-encryption OH port will be directly sent to RMAN which adds RMan message descriptor to the headroom of each packet according to configuration (Message Type 9), and then directly sends them to SRIO port 0.

### 7.4.18.2.2 Srio\_IPSec\_offload inbound flows

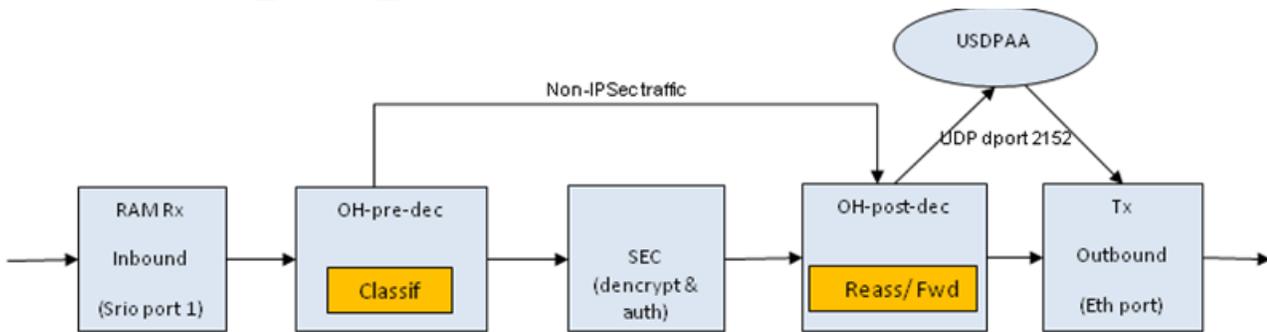


Figure 2 Inbound processing

Figure 178. Inbound processing

IPsec traffic (IPv4/IPv6 ESP and UDP-encap ESP) received on the inbound port srio 1 through RMAN is directly sent to pre-decryption OH port where the traffic is classified according to offloaded security associations. Frames for which a security association is found are sent to the appropriate SEC engine input queues for decryption/decapsulation; traffic for which a security association is not found is dropped. Non-IPsec traffic is directly sent to post-decryption OH port. Decrypted and non-IPsec traffic is reassembled if fragmented, and further a post IPsec classification which sends frames matching UDP destination port 2152 to USDPAA frame processing threads. Frames not matching UDP destination port 2152 have their Ethernet source and destination MAC addresses updated with Tx port MAC address and next hop MAC address respectively.

### 7.4.18.2.3 Limitations

The traffic load in outbound direction is not higher than 750Mbps.

This application doesn't support for restarting functionality.

## 7.4.18.3 Running srio\_ipsec\_offload

Before running this demo, please make sure some kernel options are enabled and the device tree used for this application is compiled. For more details, please refer to “Compiling the device tree and enabling kernel options”.

### 7.4.18.3.1 Application environment specifications

To simplify the test setup, it uses the similar environment as ipsec\_offload, please refer to Application environment specifications in ipsec\_offload. The application forwards traffic between one Ethernet port and two sRio ports. The one Ethernet is used for receiving clear packets, the other two srio ports process encrypted packets. Here two srio ports are connected with external line. So one srio port(srio 0) works as TX port, the other one(srio 1) as RX port. Like ipsec\_offload , one port is configured as the protected interface-BP(eth port), the others as unprotected-BH(sRio port).Now it only supports on B4860QDS Platform.

### 7.4.18.3.2 Running srio\_ipsec\_offload

Srio\_ipsec\_offload only supports b4860qds platform at present. The rcw named N\_RRSS\_0x2A\_0x7A is dedicated to using for this application. Before running this application, need to update this rcw in uboot. Use the steps described in Compiling the device tree for B4860 to generate a device tree binary file. Boot the board with the compiled kernel (arch/powerpc/boot/ulmage) and the DTB file. To reserve memory for USDPAA and enable the scatter-gather support in the DPAA Ethernet driver add the following to the boot arguments: setenv othbootargs “fsl\_fm\_max\_frm=9600” For setting up the USDPAA network configuration and PCD resources used by the application run the following commands:

```
export DEF_CFG_PATH="/usr/etc/srio_ipsec_offload_config_b4860.xml"
export DEF_PCD_PATH="/usr/etc/srio_ipsec_offload_pcd_b4.xml"
export DEF_SWP_PATH="/usr/etc/srio_ipsec_offload_swp.xml"
export DEF_PDL_PATH="/etc/fmc/config/hxs_pdl_v3.xml"
Start the application with the following commands:
/usr/bin/srio_ipsec_offload \
-c /usr/etc/srio_ipsec_offload_config_b4860.xml \
-p /usr/etc/srio_ipsec_offload_policy.xml \
--vif macless0 --vof eth2 --vipsec eth3 -y -z --mtu-pre-enc 500
```

Notes: Interfaces vif, vof and vipsec may change due to different hardware configuration.

To disable ECN tunneling in outbound and inbound direction, following parameters are used:

```
-y Disable inbound ECN tunneling
```

```
-z Disable outbound ECN tunneling
```

The following arguments specify two Linux virtual interfaces associated with inbound and outbound Ethernet ports. These interfaces are used for adding neighboring entries used for Ethernet header update before transmission:

```
--vif virtual inbound interface index
```

```
--vof virtual outbound interface index
```

The following argument configures MTU for pre encryption fragmentation. Frames larger than this value will be fragmented prior encryption:

```
--mtu-pre-enc pre encryption MTU
```

### 7.4.18.3.3 Application configuration for IPsec

IPSec can be configured for offloading using setkey tool. To add an SA and two corresponding SPs add the following lines in a setkey.conf file:

```
flush;
spdflush;
add 192.168.100.1 192.168.200.1 esp 0x201
-E 3des-cbc "abcdefghijklmnopqrstuvwxyabcde"
-A hmac-sha1 "abcdefghijklmnopqrstuwxya";
spdadd 172.16.0.1/32[any] 172.17.0.1/32[any] udp
-P out ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
spdadd 172.17.0.1/32[any] 172.16.0.1/32[any] udp
-P in ipsec esp/tunnel/192.168.100.1-192.168.200.1/require;
Now run the command:
setkey -f setkey.conf
```

These commands create an ESP tunnel with the following endpoints: IP src 192.168.100.1 – IP dst 192.168.200.1, SPI 0x201, 3DES-CBC encryption and HMAC-SHA1 authentication. The UDP traffic coming from 172.16.0.1 going to 172.17.0.1 will be tunneled using the defined tunnel.

To add neighboring entries for Ethernet header update run the following commands: ip neigh add 192.168.0.200 lladdr <inbound port MAC address> dev macless0 ip neigh add 172.17.0.1 lladdr <next hop MAC address> dev eth2

### 7.4.18.3.4 Running Traffic

Assuming that the traffic is generated with a directly connected Linux box, you need to configure the Linux box to output frames for 172.16.0.1 on the connected interface (ethX):ifconfig <ethX> 172.16.0.2 netmask 255.255.0.0 up

You can use the hping tool to generate UDP packets and tcpdump to capture traffic on interface ethX. Each sent frame should be returned by the application. hping -2 -s 2150 -p 2150 -c 1 172.17.0.1

At any point you can use the following commands to show packet and byte statistics

- sa\_stats <sa-id>

This command shows the statistics for a particular SA

- ipsec\_stats 0

This command shows the global IPsec statistics. This includes the miss counters for the inbound and outbound direction.

## 7.4.18.4 Compiling the device tree and enabling kernel options

### 7.4.18.4.1 Compiling the device tree for B4860

Go to your Linux kernel source code directory.

```
cp drivers/staging/fsl_dpa_offload/dts/b4860si-pre.dtsi
arch/powerpc/boot/dts/fsl
cp drivers/staging/fsl_dpa_offload/dts/b4860si-chosen-offfld.dtsi
arch/powerpc/boot/dts/fsl/b4860si-chosen.dtsi
cp drivers/staging/fsl_dpa_offload/dts/b4860qds-usdpaa-srio-eth-shared-interfaces.dts
arch/powerpc/boot/dts
scripts/dtc/dtc -f -b 0 -p 1024 -I dts -O dtb -o b4860qds-usdpaa-srio_eth_shared-mac.dtb
arch/powerpc/boot/dts/b4860qds-usdpaa-srio-eth-shared-interfaces.dts
```

The DTB file b4860qds-usdpaa-srio\_eth\_shared-mac.dtb will be built in the current directory.

## 7.4.18.4.2 Enabling DPA Offloading and RMAN Drivers in the Linux Kernel

Regarding `srio_ipsec_offload`, it need two srio ports, some options related to them should be selected as below.

```
Device Drivers --->
 <*> Userspace I/O drivers --->
 <*> Freescale Serial RapidIO support
 [*] Staging drivers --->
 [*] Freescale RapidIO Message Manager support
 [*] Freescale Datapath Offloading Driver
```

# Chapter 8

## Boot Loaders

### 8.1 Primary Protected Application (PPA) User's Guide

#### 8.1.1 Introduction

##### 8.1.1.1 Rationale and Scope

This document is the Specification and Users Guide for a loadable secure services firmware component running in TrustZone. This component, called the Primary Protected Application (PPA), has the following characteristics:

- Is loaded into the secure side of an ARM core early in the boot process
- Remains resident after boot
- Provides secure services for boot software and runtime software
- Contains a Secure Monitor, which controls access to/from the secure world
- Implements services behind a std abstract interface (published by ARM)
- Has the secure world exception vectors and handlers
- Is the focal point for implementing the Platform Security Policy

Each of these will be discussed in detail in the sections that follow. Although secure boot and other boot components such as bootloaders and bootrom will be mentioned here, this specification is not intended to exhaustively cover secure boot, bootloaders (such as UEFI and U-boot), or bootrom code.

The phrase “secure world”, used heavily in this doc, refers to ARM TrustZone, *and vice-versa*. The phrase “secure monitor” refers to the ARM v8 definition of a software entity that runs at EL3 and controls access to/from the secure world. Do not confuse “secure monitor” in this context with “security monitor”, which is a hardware component of the QorIQ Trust Architecture.

There are a number of compelling reasons for having a resident secure services layer:

1. The secure services layer is first-and-foremost a focal point for implementation of a Platform Security Policy.
2. ARM cores come out of reset executing in the secure world.
3. The non-secure world needs an agent to perform tasks in the secure world
4. The PSCI interface, which is ARM's abstract power mgmt interface, runs in the secure world.
5. A resident secure firmware can streamline bootloaders, making it easier to support multiple bootloaders.
6. The PPA is the foundation upon which a deeper TrustZone software stack can be built.

##### 8.1.1.2 References

[1] LS1043A SoC Architecture Specification v0.3.0

[2] SMC Calling Convention, ARM Ltd, 2013

[3] ARM Architecture Reference Manual, Armv8 Edition, beta

[4] LS1043 PRL, Revision 0.7

[5] QorIQ Chassis Architecture Specification, Generation 2.1, Revision 0.8

[6] LS1 Trust Architecture, Chapter 10

- [7] Layerscape Chassis Architecture Specification, Generation 3, v0.9
- [8] ARM Trusted Firmware Design
- [10] LS2 Boot Interfaces Programmers Guide, v1.5
- [11] Power State Coordination Interface, ARM Ltd, 2012-2013
- [12] QorIQ LS1046A Reference Manual, Rev C, 06/2016

### 8.1.1.3 Definitions

AP – Application Processor, same as GPP

ATF – ARM Trusted Firmware

Bootloader – FW that loads the OS kernel (uboot, uefi)

ESBC – External Secure Boot Code, image validation code in the bootloader

GPP – General Purpose Processor

ISBC – Internal Secure Boot Code, image validation code in the bootrom

OCRAM – On-Chip RAM

PPA – Primary Protected Application, the secure monitor and associated functions that comprise the base EL3 sw foundation

Protected – Higher-privilege sw, such as a hypervisor

PSCI – Power State Coordination Interface, an ARM std interface

Secure – SW or components that are isolated by the TrustZone architecture

Secure Monitor – the SW running at EL3 that controls the gateway from the non-secure world to the secure world

Security Monitor – a HW feature of the QorIQ Trust Architecture

SCP

SMC – an ARM instruction which generates an exception, *and* an ARM std interface based on that exception call

SP – Service Processor, an auxiliary core that performs initial boot functions on the SoC

TPM – Trusted Platform Module, a specification of the Trusted Computing Group

Trusted Architecture (TA) – a security architecture found in the QorIQ family of SoCs, including the ARM-based QorIQ products

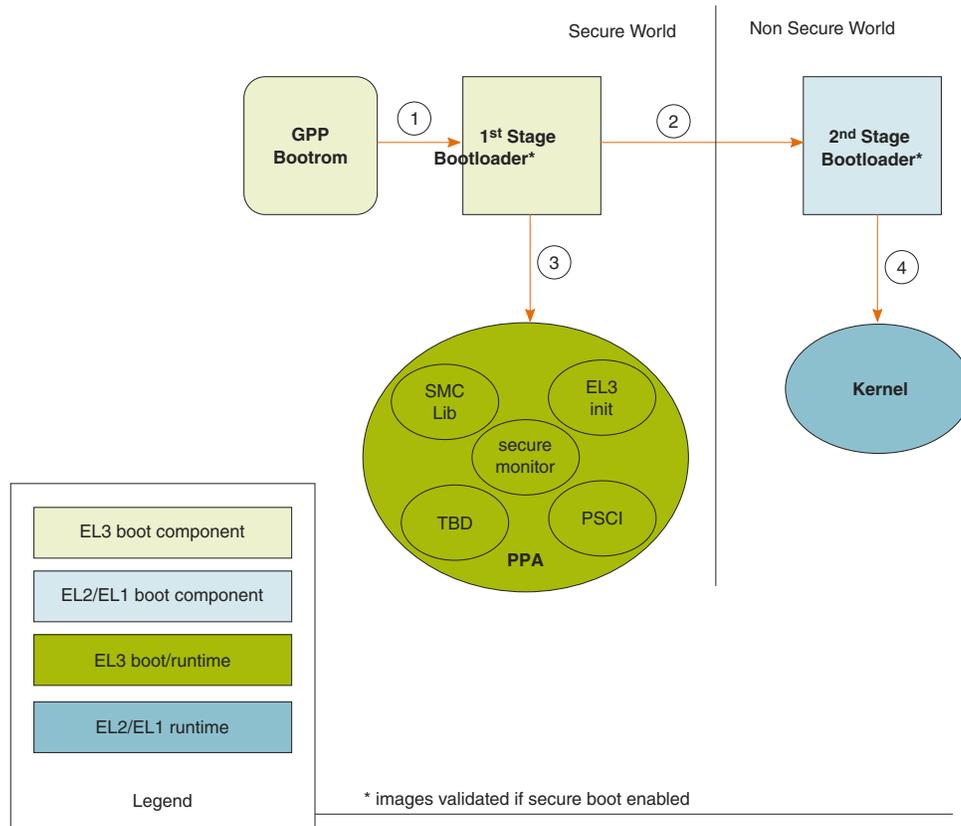
TrustZone (TZ) – an isolation context provided as part of the ARM architecture; an infrastructure for building secure subsystems

## 8.1.2 Boot Flow Architecture

### 8.1.2.1 LS1046A Boot Flow

#### Component Load Sequence

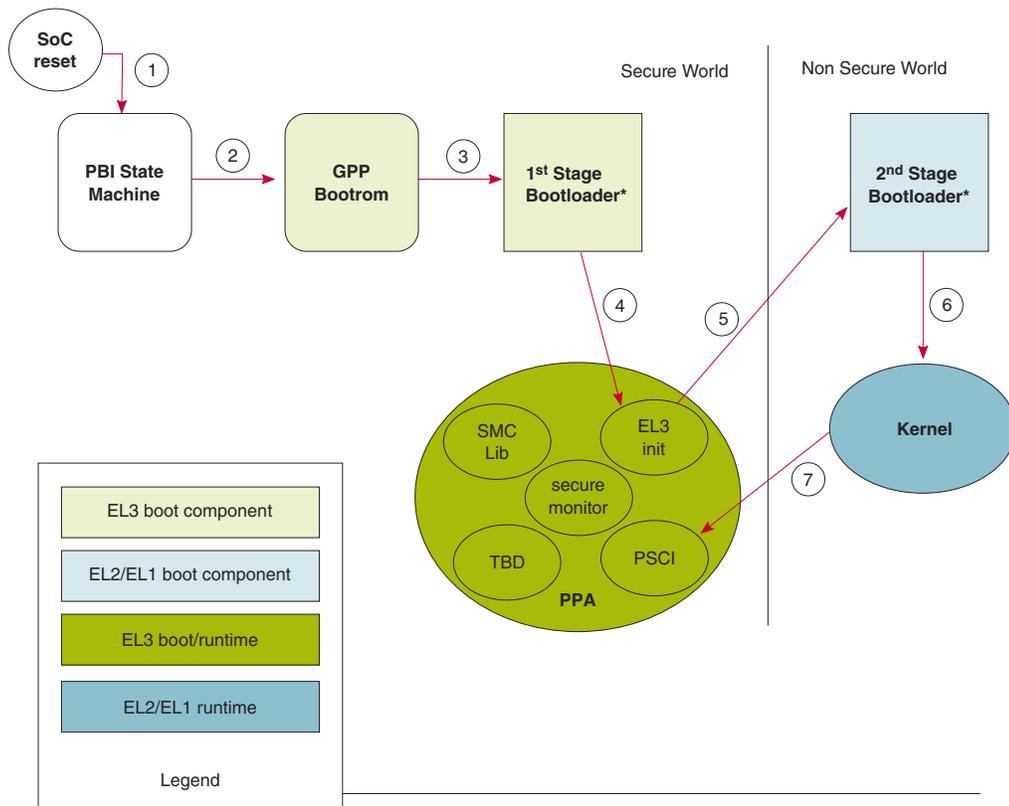
1. GPP Bootrom loads/validates\* 1st stage bootloader
2. 1st stage bootloader loads/validates\* 2nd stage bootloader
3. 1st stage bootloader loads/validates\* PPA
4. 2nd stage bootloader loads/validates\* kernel



**Figure 179. Component Load Sequence**

**Boot execution Order**

1. Execution begins in the PBI State Machine when the SoC comes out of reset
2. After PBI, execution starts with bootcore in GPP bootrom
3. Bootcore branches to 1st stage bootloader running in EL3
4. Bootcore in 1st stage bootloader branches to EL3 init code in PPA
5. When bootcore completes EL3 init, it branches to 2nd stage bootloader in EL2
6. Bootcore in 2nd stage bootloader branches to Linux kernel in EL1



**Figure 180. Boot Execution Order**

**Secondary core execution path**

1. Execution starts in the GPP bootrom when secondary core released from reset
2. If core is marked to be disabled, core enters power-down sequence in bootrom
3. Cores not disabled branch to EL3 init code in PPA
4. Upon completion of EL3 init, cores branch to start address at EL2 in kernel

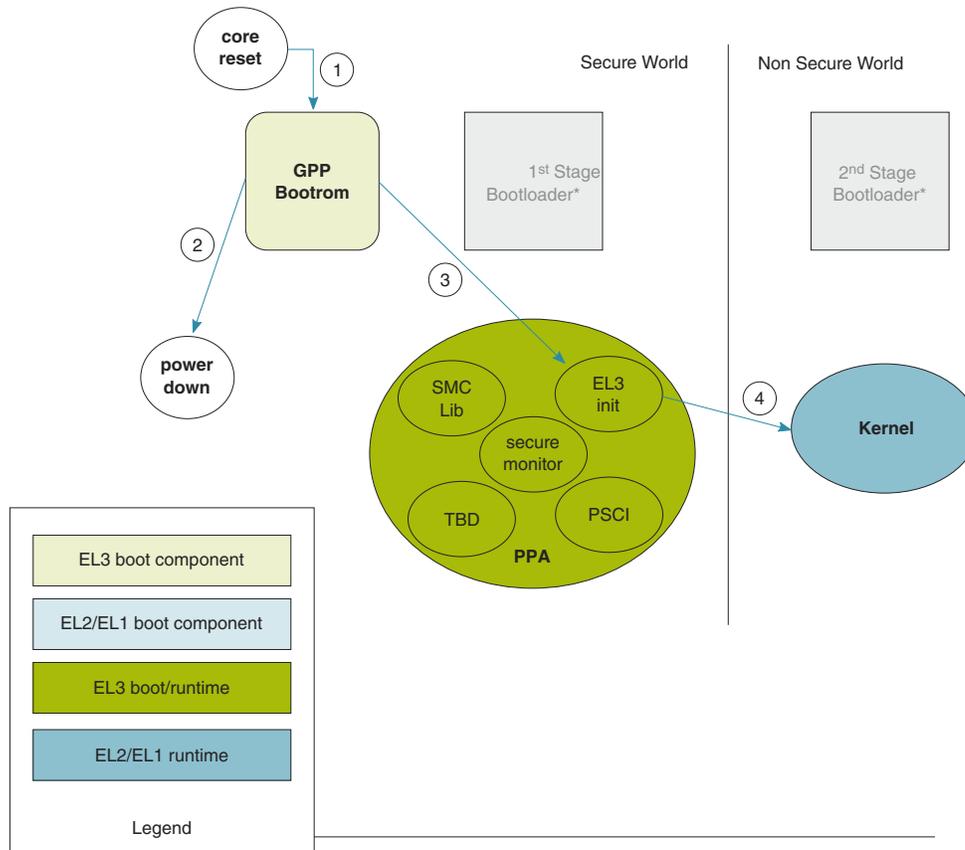


Figure 181. Secondary Core Execution Path

### 8.1.3 Loading and Initializing the PPA

- The PPA must be loaded to a 64Kb boundary
- Copy the binary image to the load address – the component installing the PPA MUST be executing at EL3
- PPA should be loaded to an address in secure memory - recommend a 2MB secure region in DDR (but PPA can be tested in non-secure DDR)
- After copying the image file to DDR, clean the data cache by VA (all virtual address ranges affected by PPA load) , and invalidate the instruction cache
- The PPA initialization runs at EL3 – but the PPA transfers control back to the address loaded in BOOTLOCPTR **at EL2** – you must write the start address of the EL2 portion of your bootloader into BOOTLOCPTR *before* initializing the PPA
- After writing the EL2 start address in BOOTLOCPTR, initialize the PPA by branching to its start address

### 8.1.4 How to Call SMC/PSCI functions

- SMC functions obey the ARM ABI
- SMC functions treat registers x0-x12 as volatile, all others are preserved
- To call an SMC/PSCI function, load the registers according to the table below, then execute an “SMC 0x0” instruction
- If the function specifies a return value, it will be found in register x0

- Return values, including error returns, are returned in x0
- The “SMC 0x0” instruction generates an exception – after the exception is processed in the secure monitor (when the SMC function has completed), control will return to the instruction following the “SMC 0x0”
- Please refer to [2] for more details of the SMC calling convention
- Note: currently, only SMC “fast calls” are implemented (including PSCI). This means that while the calling core is in the secure world, interrupts to the core are masked.

SMC input parameters:

Register	Meaning
X0	<i>Function ID (see Section 5, 6)</i>
X1	<i>First optional parameter</i>
X2	<i>Second optional parameter</i>
X3	<i>Third optional parameter</i>

SMC return values:

Register	Meaning
X0	<i>Return value, Error return code</i>

## 8.1.5 PSCI Function List

Please see [11] for details of the PSCI function interface. Keep in mind that the PSCI interface is a subset of the SMC Calling Convention [2].

### 8.1.5.1 PSCI\_VERSION

Get the Version number of this PSCI implementation.

Function ID: 0x8400\_0000

Input parameters:

Register	Meaning
X0	<i>Function ID</i>
<i>No other inputs</i>	

Return values:

Register	Meaning
X0 bits[31:16]	<i>Major Version Number</i>
X0 bits [15:0]	<i>Minor Version Number</i>

Currently, the PSCI v0.2 specification is implemented. Thus, the *Major Version Number* returned is 0x0, and the *Minor Version Number* returned is 0x2.

## 8.1.5.2 CPU\_ON

Release a secondary core from reset, or from the CPU\_OFF state.

Function ID: 0xC400\_0000

Input Parameters:

Register	Meaning
X0	<i>Function ID</i>
X1	<i>Target CPU, in MPIDR format (see [11])</i>
X2	<i>Start address (Physical)</i>
X3	<i>Context ID</i>

Return Values:

Register	Return Code (see 5.8)
X0	SUCCESS
"	INVALID_PARAMETERS
"	ALREADY_ON
"	ON_PENDING
"	INTERNAL_FAILURE

### NOTE

When cores are delivered to the *Start Address*, they will be executing at EL2.

## 8.1.5.3 CPU\_OFF

Power down the calling core.

Function ID: 0x8400\_0002

Input Parameters:

Register	Meaning
X0	<i>Function ID</i>

Return Values:

Register	Return Code (see 5.8)
<i>Function does not return if successful</i>	
X0	DENIED

Note that this function is called on the core that is to be powered down. There is no mechanism to power down one core from another core. By definition, a power-down state is a state without retention, so the caller must save whatever state is needed when the core resumes execution.

The only way to restart a core after a call to *CPU\_OFF* is with a call to *CPU\_ON*.

If successful, this function does not return.

### 8.1.5.4 CPU\_SUSPEND

Put the calling core/cluster/system into a low-power state.

Function ID: 0xC400\_0001

Input Parameters (see [11]):

Register	Meaning
X0	<i>Function ID</i>
X1	<i>Power_State</i>
X2	<i>Start_Address (Physical)</i>
X3	<i>Context_Id</i>

Return Values:

Register	Return Code (see 5.8)
X0	SUCCESS
"	INVALID_PARAMETERS

Note that this function is called on the core/cluster/system that is to be suspended. There is no mechanism to suspend one core from another core. There are two available power states, *Standby* and *Power-Down* (see [11]).

For cluster low-power states, all cores of the cluster except this final core must already be in the requested power state. The function checks to see if this is the "last core standing" of the cluster – if it is the last core, the core is suspended along with the cluster. If this function is called when there is more than one active core in the cluster, the function will return with the INVALID\_PARAMETERS value in x0.

Likewise for system power-down, the function checks to see if this is the last active gpp core in the SoC. If it is the last active core, then the core is suspended along with the SoC. If it is not the last active core, then the function returns with the error value INVALID\_PARAMETERS in x0.

The input parameters to this function describe a complex interface. Please see [11] for details of how to use this function call.

If successful, this function does not return.

### 8.1.5.5 AFFINITY\_INFO

Get information about a specific affinity level.

Function ID: 0xC400\_0004

Input Parameters (see [11]):

Register	Meaning
X0	<i>Function ID</i>
X1	<i>Target_Affinity</i>
X2	<i>Lowest_Affinity</i>

Return Values:

Register	Return Codes (see 5.8, 5.8.1)
X0	ON_PENDING
"	OFF
"	ON
"	INVALID_PARAMETERS
"	NOT_PRESENT
"	DISABLED

### 8.1.5.6 SYSTEM\_OFF

Power down the entire system.

Function ID: 0x8400\_0008

Input Parameters:

Register	Meaning
X0	Function ID

Return Values:

*The function does not return.*

### 8.1.5.7 SYSTEM\_RESET

Perform a hard reset on the entire system.

Function ID: 0x8400\_0009

Input Parameters:

Register	Meaning
X0	Function ID

Return Values:

*The function does not return.*

### 8.1.5.8 PSCI Return Code Values

Mnemonic	Value
SUCCESS	0
NOT_SUPPORTED	-1
INVALID_PARAMETERS	-2
DENIED	-3

*Table continues on the next page...*

Table continued from the previous page...

ALREADY_ON	-4
ON_PENDING	-5
INTERNAL_FAILURE	-6
NOT_PRESENT	-7
DISABLED	-8

**PSCI Return Codes Specific to Affinity\_Info**

Mnemonic	Value
ON	0
OFF	1
ON_PENDING	2

**8.1.5.9 PSCI Functions Implemented, by SoC**

	LS1046A
CPU_ON	
CPU_OFF	
AFFINITY_INFO	
CPU_SUSPEND	
PSCI_VERSION	
SYSTEM_RESET	
SYSTEM_OFF	X

**8.1.6 SMC Function List**

Please see [2] for details of the SMC function interface.

**8.1.6.1 Function Count - SMC64**

Return the number of functions implemented by the smc64 interface, including this function and PSCI functions using this interface.

Uses smc64 interface.

Function ID: 0xC200\_FF00

Input Parameters:

Register	Meaning
X0	Function ID

Return Values:

Register	Value
X0	<i>Smc64 function count</i>

### 8.1.6.2 Function Count - SMC32

Return the number of functions implemented by the smc32 interface, including *this* function and PSCI functions using this interface.

Uses smc32 interface.

Function ID: 0x8200\_FF00

Input Parameters:

Register	Meaning
X0	<i>Function ID</i>

Return Values:

Register	Value
X0	<i>Smc32 function count</i>

### 8.1.6.3 Get UUID

Return the 128-bit UUID uniquely identifying this SMC implementation.

Uses the smc32 interface.

Function ID: 0x8200\_FF01

Input Parameters:

Register	Meaning
X0	<i>Function ID</i>

Return Values:

Register	Value
X0	Bytes [3:0] of UUID
X1	Bytes [7:4] of UUID
X2	Bytes [11:8] of UUID
X3	Bytes [15:12] of UUID

### 8.1.6.4 Get Revision

Return the major and minor revision numbers of the SIP portion of this SMC implementation.

Uses smc32 interface.

Function ID: 0x8200\_FF03

Input Parameters:

Register	Meaning
X0	Function ID

Return Values:

Register	Value
X0	Major revision number of sip-smc
X1	Minor revision number of sip-smc

## 8.1.7 Building the PPA

In the SDK, the PPA image will be built as part of the Yocto recipe.

## 8.1.8 System Considerations When Calling SMC & PSCI Functions

There is always the possibility that malware will attempt to execute an SMC. The affects of this may be mitigated with three methods:

1. Insure that calling an smc/psci function can *never* corrupt the machine
2. Insure that calling an smc/psci function can *never* lower the security stance of the machine
3. Provide *only* one place in the system, a kernel driver, from which *all* smc calls are made

Items (1) & (2) mean that even if malware successfully makes an smc call, the effects of that call will not open the system up to an attack. Take careful note of item (2) – this means that any EL3 configuration that lowers the security of the system must be set in **the PPA initialization phase**, and *not* as a dynamic smc function.

Item (3) allows the PPA to determine if an unauthorized access has been attempted. Since the smc call generates an exception, the register ELR\_EL3 is loaded, by hw, with the return address of the caller. If there is only one place in the system where smc calls are made from, then the PPA can be configured to register the return address. Malware, attempting to make an smc call, will have a different return address. The secure monitor in the PPA can detect this different return address, and reject the request. In addition, the secure monitor can return to the authorized return address with a security violation error. To enable this capability in later revisions of the PPA, the kernel developer should implement a kernel driver where all smc calls are made from.

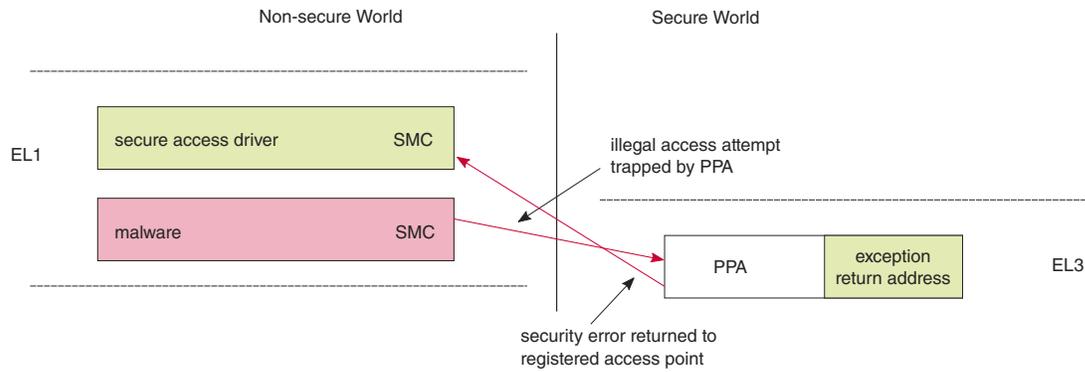


Figure 182. System Considerations When Calling SMC and PSCI Functions

## 8.2 Secure Boot: PBL Based Platforms

### 8.2.1 Introduction

This document is intended for end-users to demonstrate the image validation process. The image validation can be split into stages, where each stage performs a specific function and validates the subsequent stage before passing control to that stage. In the example, the ESBC is Freescale provided reference code referred to as ESBC uboot.

**Chain of Trust** ESBC uboot performs minimal SoC configuration before validating the Next Executable using the same CSF header format as the ISBC used to validate ESBC Uboot. The CSF Header and signature are added to the Next Executable using the Freescale Code Signing Tool.

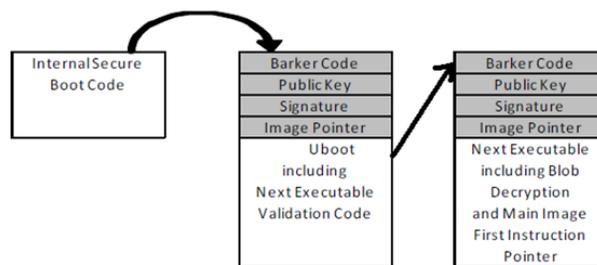
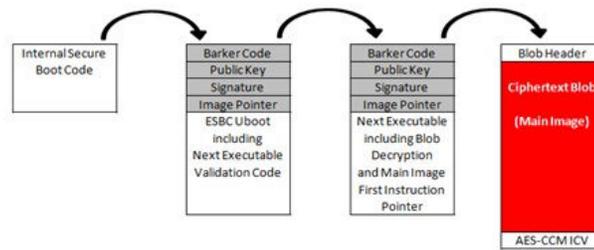


Figure 183. Chain of Trust

**Chain of Trust with confidentiality**

The validated ESBC uboot image is allowed to use the One Time Programmable Master Key to decrypt system secrets. Cryptographic blob mechanism is used to establish Chain of trust with confidentiality.



**Figure 184. Chain of Trust with confidentiality**

This document provides more details on the secure boot flow, ISBC, ESBC and Freescale Code signing tool.

## 8.2.2 Secure boot Process

Secure boot process uses a digital signature validation routine already present in INTERNAL BOOT ROM. This routine performs validation using HW bound RSA public key to decrypt the signed hash and compare it to a freshly calculated hash over the same system image. If the comparison passes, the image can be considered as authentic.

The complete process can be broken down into following phases:

- Pre Boot Phase
  1. PBL
  2. SFP
- ISBC
- ESBC

The Complete Secure boot Process is shown in the Figure below.

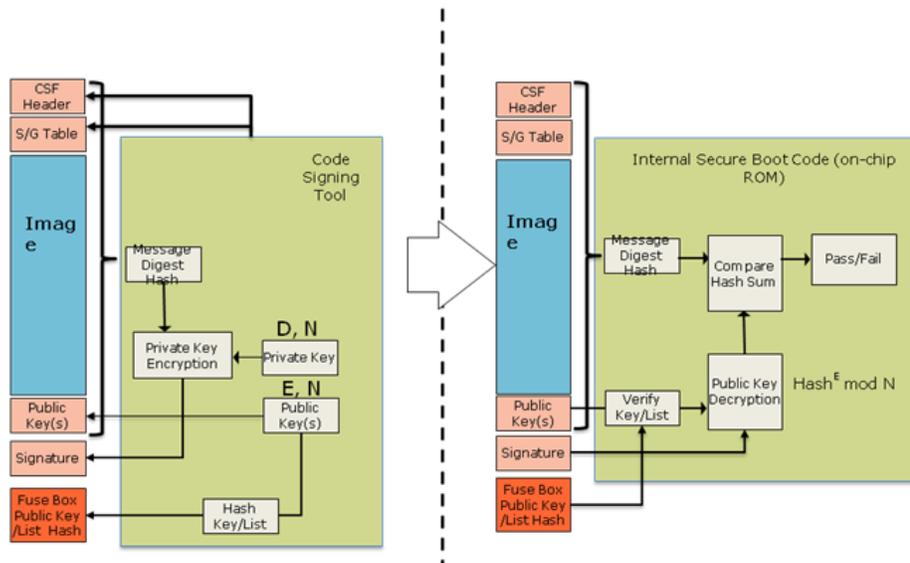


Figure 185. Secure Boot Process

## 8.2.3 Pre-Boot Phase

When the processor is powered on, reset control logic blocks all device activity (including scan and debug activity) until fuse values can be accurately sensed. The most important fuse value at this stage of operation is the 'Intent to Secure' (ITS) bit. When an OEM sets ITS, they intend for the system to operate in a secure and trusted manner.

The two main components involved during this process are :

**The security fuse processor (SFP)** has two roles. The first is to physically burn fuses during device provisioning. The second is to use these provisioned values to enforce security policy in the pre-boot phase, and to securely pass provisioned keys and other secret values to other hardware blocks when the system is in a trusted/secure state.

**PreBoot Loader (PBL)** is the micro-sequencer that can simplify system boot by configuring the DDR memory controllers to more optimal settings and copying code and data from low speed memory into DDR. This allows subsequent phases of boot to operate at higher speed. The setting of ITS determines where the PBL is allowed to read and write. The use of the PBL is mandatory when performing secure boot. At a minimum, the PBL must read a command file from a location determined by the Reset Configuration Word (RCW) and perform a store of a value to the ESBC Pointer Register within the SoC. If the PBL doesn't perform this operation (or sets the ESBC pointer to the wrong value), the ISBC will fail to validate the ESBC. Once the PBL has completed any operations defined by its command file, the PBL is disabled until the next Power on Reset and the Boot Phase begins.

Some example PBI commands used in the demo are given below. The commands are embedded in the RCW's mentioned in the [SDK Images required for the demo](#)

## NOR SECURE BOOT

### • P3/P4/P5

```
#LAW for ESBC
 09000cd0 00000000
 09138000 00000000 (Flush command)
 09000cd4 c0000000
 09138000 00000000 (Flush command)
 09000cd8 81f0001d
 09138000 00000000(FLUSH command)
Scratch Register
 090e0200 c0b00000
```

### • T1/T2/T4/B4

```
#LAW for ESBC
 09000c10 00000000
 09000c14 c0000000
 09000c18 81f0001b
LAW for CPC/SRAM
 09000d00 00000000
 09000d04 bff00000
 09000d08 81000013
Scratch Registers
 090e0200 c0b00000
 090e0208 c0c00000
CPC SRAM
 09010100 00000000
 09010104 bff00009
CPC Configuration
 09010f00 08000000
 09010000 80000000
```

## NAND SECURE BOOT

### • P3/P5

```
SCRATCH REGISTER
 090e0200 bff00000
 09138000 00000000 (Flush Command)
CPC1 SRAM
 09010000 00200400
 09010100 00000000
 09010104 bff0000b
 09010f00 08000000
 09010000 80000000
 09138000 00000000 (Flush Command)
LAW for CPC/SRAM
 09000d00 00000000
 09000d04 bff00000
 09000d08 81000013
 09138000 00000000 (Flush Command)
Alternate Configuration Space Configuration
 09000010 00000000
 09000014 bf000000
 09000018 81000000
 09138000 00000000 (Flush Command)
CPC2 Cache
 09110000 80000403
```

```

09110020 2d170008
09110024 00100008
09110028 00100008
0911002c 00100008
09138000 00000000 (Flush Command)

```

```

/* hdr_uboot.out and u-boot.bin must also be loaded on NAND
 * ALT_CONFIG_WRITE command must be used for the same.
 * Starting offset for ALT_CONFIG_WRITE command would be
 * hdr_uboot.out - 0xf00000
 * u-boot.bin - 0xf40000
 */

```

The ISBC is capable of reading from NOR flash connected to the Local Bus, on-chip memory configured as SRAM, or main memory. Unless the ESBC is stored in NOR flash, the developer is required to create a PBL Image that copies the image to be validated from NVRAM to main memory or internal SRAM prior to writing the SCRATCHRW1 Register and executing the ISBC code.

To assist with the creation of PBL Images (for both normal and Trust systems), Freescale offers a PBL Image Tool.

Note that it is possible for an attacker to modify the board to direct the PBL to the wrong non-volatile memory interface, or change the PBL Image and CSF Header pointer, however this will result in a secure boot failure and the system remaining in an idle loop indefinitely.

## 8.2.4 ISBC Phase

### 8.2.4.1 Flow in the ISBC Code

With the PBL disabled and all external masters blocked by the PAMUs, CPU 0 is released from boot hold-off and begins executing instructions from a hardwired location within the Internal BootROM. The instructions inside the Internal BootROM are Freescale developed code known as the Internal Secure Boot Code (ISBC). The ISBC leads CPU 0 to perform the following actions:

1. **Who am I check?** - CPU 0 reads its Processor ID Register, and if it finds any value besides physical CPU 0, the CPU enters a loop. This insures that only CPU 0 executes the ISBC.
2. **Sec\_Mon check** - CPU 0 confirms that the Sec\_Mon is in the Check state. If not, it writes a 'fail' bit in a Sec\_Mon control register, leading to a state transition.
3. **ESBC pointer read** - CPU 0 reads the ESBC Pointer Register, and then reads the word at the indicated address, which is the first word of the Command Sequence File Header which precedes the ESBC itself. If the contents of the word don't match a hard coded preamble value, the ISBC takes this to mean it has not found a valid CSF and cannot proceed. This leads to a fail, as described in #2 above.
4. **CSF parsing and public key check** - If CPU 0 finds a valid CSF header, it parses the CSF header to locate the public key to be used to validate the code. There can be a single public key or a table of 4 public keys present in the header. The Secure Fuse Processor doesn't actually store a public key, it stores a SHA-256 hash of the public key/table of 4 keys. This is done to allow support for up to 4096b keys without an excessively large fuse block. If the hash of the public key fails to match the stored hash, secure boot fails.
5. **Signature validation** - With the validated public key, CPU 0 decrypts the digital signature stored with the CSF header. The ISBC then uses the ESBC lengths and pointer fields in the CSF header to calculate a hash over the code. The ISBC checks that the CSF header is included in the address range to be hashed. Option flags in the CSF header tell the ISBC whether the Freescale Unique ID and the OEM Unique ID (in the Secure Fuse Processor) are included in the hash calculation. Including these IDs allows the image to be bound to a single platform. If the decrypted hash and generated hash don't match, secure boot fails.
6. **ESBC First Instruction Pointer check** - One final check is performed by the ISBC. This check confirms that the First Instruction Pointer in the CSF header falls within the range of the addresses included in the previous hash. If the pointer

is valid, the ISBC writes a 'PASS' bit in a Sec\_Mon command register, the state machine transitions to 'Trusted', and the OTPMK is made available to the SEC.

7. In case of failure, for Trust v2.0 devices, secondary flag is checked in the CSF header. If set, ISBC reads the CSF header pointer from SCRATCHRW3 location and repeats from step 4.

There are many reasons the ISBC could fail to validate the ESBC. Technicians with debug access can check the SCRATCHRW2 Register to obtain an error code. For a list of error codes refer ISBC Validation Error Codes.

## 8.2.4.2 Super Root keys (SRKs) and signing keys

These are RSA public and private key pairs. Private keys are used to sign the images and public keys are used to validate the image during ISBC and ESBC phase.

Public keys are embedded in the header and the hash of srk table is fused in SRKH register of SFP.

These are Hardware Bound Keys, once the hash is fused the public private key pair can't be modified.

Keys of sizes 1k, 2k and 4k are supported in FSL Secure Boot Process.

It is the OEM's responsibility to tightly control access to the RSA private signature key. If this key is ever exposed, attackers will be able to generate alternate images that will pass secure boot.

If this key is ever lost, the OEM will be unable to update the image.

## 8.2.4.3 Key Revocation

Trust Architecture 2.0 introduces support for revoking the RSA public keys used by the ISBC to verify the ESBC. The RSA public keys used for this purpose are called super root keys.

OEM can use either a single key or a list of upto 4 super root keys in the Trust Arch v2.0 devices.

In the Freescale Code Signing Tool (CST), the OEM defines whether the device uses a single super root key, or offers a list of super root keys. If using a single super root key, a new flag bit in the CSF header will indicate "Key", otherwise the flag will indicate "Key List". Assuming key list, the OEM can populate a list of up to 4 super root keys for trust arch v2.0 onwards platforms. And calculates a SHA-256 hash over the list. This hash is written to the SRKH registers in the SFP.

As part of code signing, the OEM defines which key in the key list is to be used for validating the image. This key number is included as a new field in the CSF header.

During secure boot, the ISBC determines whether a key list is in use. If the key list is valid, the ISBC checks the key number indicated in the CSF header against the revocation fuses in the SFP's OEM Security Policy Register (SFP\_OSPPR). If the key is revoked, the image validation fails.

### NOTE

In order to prevent unauthorized revocation of keys, SFP provides a bit (Write Disable). If the bit is set, the Key revocation bits cannot be written to.

In regular operation, the ESBC (early Trusted S/W) needs to set the SFP Write Disable bit. When circumstances call for revoking a key, the OEM will use an ESBC image with "Write Disable" bit not set. So, the SFP will be in a state in which key revocation fuses can be set.

Logically after revoking the required key(s), the OEM would then load a new signed ESBC image with code to set the "Write Disable" bit, with new CSF header indicating which of the remaining non-revoked key to use.

So, only the possessor of a legitimate RSA private key can enable key revocation.

One possible motivation for an OEM to revoke a super root key is the loss of the associated RSA private key to an attacker. If the attacker has gained access to a legitimate RSA private key, and the attacker can turn on power to the fuse programming circuitry, then the attacker could maliciously revoke keys. To prevent this from being used to permanently disable the system, one super root key does not have an associated revocation fuse.

### 8.2.4.4 Alternate Image Support

Trust 2.0 onwards will support a primary and alternate image, where failure to find a valid image at the Primary location will cause the ISBC to check a configured alternate location.

To execute, the alternate image must be validated using a non-revoked public key as defined by its CSF Header. A valid alternate image has same rights and privileges as a valid primary image.

This feature helps to reduce risk of corrupting single valid image during firmware update or as a result of Flash block wear-out.

To enable this feature, create PBI with pointers for both Primary and Alternate Images (HW PBL uses SCRATCHRW1 & SCRATCHRW3).

### 8.2.4.5 ESBC with CSF Header

ESBC is the generic name for the code that the ISBC validates. A few ESBC scenarios are described in later sections.

The figure below provides an example of an ESBC with CSF (Command Sequence File) Header. The CSF Header includes lengths and offset which allow the ISBC to locate the operands used in ESBC image validation, as well as describe the size and location of the ESBC image itself.

Note: CSF Header and ESBC Header may be used synonymously in this and other Freescale Trust Architecture documentation.

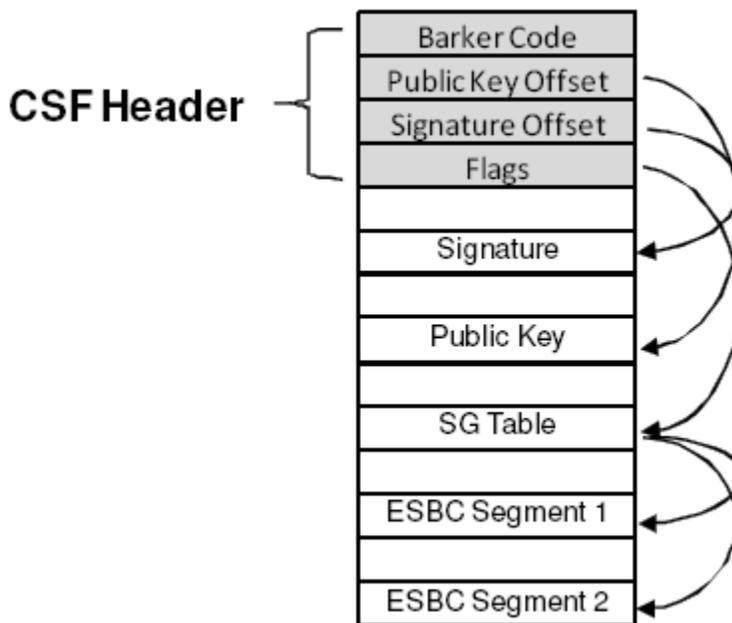


Figure 186. ESBC with CSF Header

### 8.2.5 ESBC Phase

Unlike the ISBC, which is in an internal ROM and therefore unchangeable, the ESBC is Freescale-supplied reference code, and can be changed by OEMs. The remainder of this section is the description of a reasonable secure boot chain of trust based on Freescale's reference software for secure boot. Depending on the requirement, ESBC can be a monolithic image - including uboot, device trees, boot firmware, drivers along with the OS and applications or can be mini-uboot.

Freescale provided ESBC consists of standard u-boot which has been signed using a private key. U-boot reserves a small space for storing environment variables. This space is typically one sector above or below the u-boot and is stored on persistent storage devices like NOR flash if macro CONFIG\_ENV\_IS\_IN\_FLASH is used. In case of secure boot, macro

CONFIG\_ENV\_IS\_NOWHERE is used and so, environment is compiled in uboot image and is called default environment. This default environment can't be stored on flash devices. User won't be able to edit this environment also as he can't reach to uboot prompt in case of secure boot. There is default boot command for secure boot in this default environment which executes on autoboot.

ESBC validates a file called boot script and on successful validation execute the commands in the boot script.

There are many reasons ESBC could fail to validate Client images or boot script. The error status message along with the code is printed on the u-boot console. For a list of error codes refer ESBC Validation Error Codes.

Users are free to use Freescale ESBC as it is provided or to use it as reference to modify their own secure boot system.

---

**NOTE**

On Soc's with ARMv8 core (eg:- LS1043, LS1046), during ISBC phase in Internal Boot ROM, SMMU (which by default is in by-pass mode) is configured to allow only secure transactions from CAAM.

The security policy w.r.t. SMMU in ESBC phase must be decided by the user/customer. So, currently in ESBC (U-Boot), SMMU is configured back to by-pass mode allowing all transactions (secure as well as non-secure).

---

## 8.2.5.1 Boot script

Bootscrip is a U-Boot script image which contains u-boot commands. ESBC would validate this boot script before executing commands in it.

---

**NOTE**

1. Boot script can have any commands which u-boot supports. No checking on the allowed commands in boot script. Since it is validated image, assumption is that commands in boot script would be correct.
  2. If some basic scripting error done in boot script like unknown command, missing arguments, the required usage of that command and core is put in infinite loop.
  3. After execution of commands in boot script, if control reaches back in u-boot, error message would be printed on u-boot console and core would be put in spin loop by command `esbc_halt`.
  4. Scatter gather images not supported with validate command.
  5. If ITS fuse is blown, any error in verification of the image would result in system reset. The error would be printed on console before system goes for a reset.
- 

### 8.2.5.1.1 Where to place the boot script?

Freescale's ESBC u-boot expects the boot script to be loaded in flash as specified in [Address map used for the demo](#). ESBC u-boot code assumes that the public/private key pair used to sign the boot script is same as that was used while signing the u-boot image. If user used different key pair to sign the image, hash of the N and E component of the key pair should be defined in macro:

**CONFIG\_BOOTSCRIPT\_KEY\_HASH.**

Note - The hash defined should be hex value, 256 bits long.

Both the above macros can be defined or changed in the configuration file `secure_boot.h` at the following location in u-boot code:

```
u-boot/arch/powerpc/include/asm/fsl_secure_boot.h
```

Two new commands called `esbc_validate` and `esbc_halt` have been added in Freescale ESBC u-boot.

### 8.2.5.1.2 Chain of Trust

Boot script contains information about the next level of images, e.g. Linux, HV, etc. ESBC validates these images as per their public keys and then executes bootm command to pass-on the control to next image.

Users are free to use Freescale ESBC as it is provided or to use it as reference to modify their own secure boot system.

Figure below shows the Chain of trust established for Validation with this ESBC u-boot.

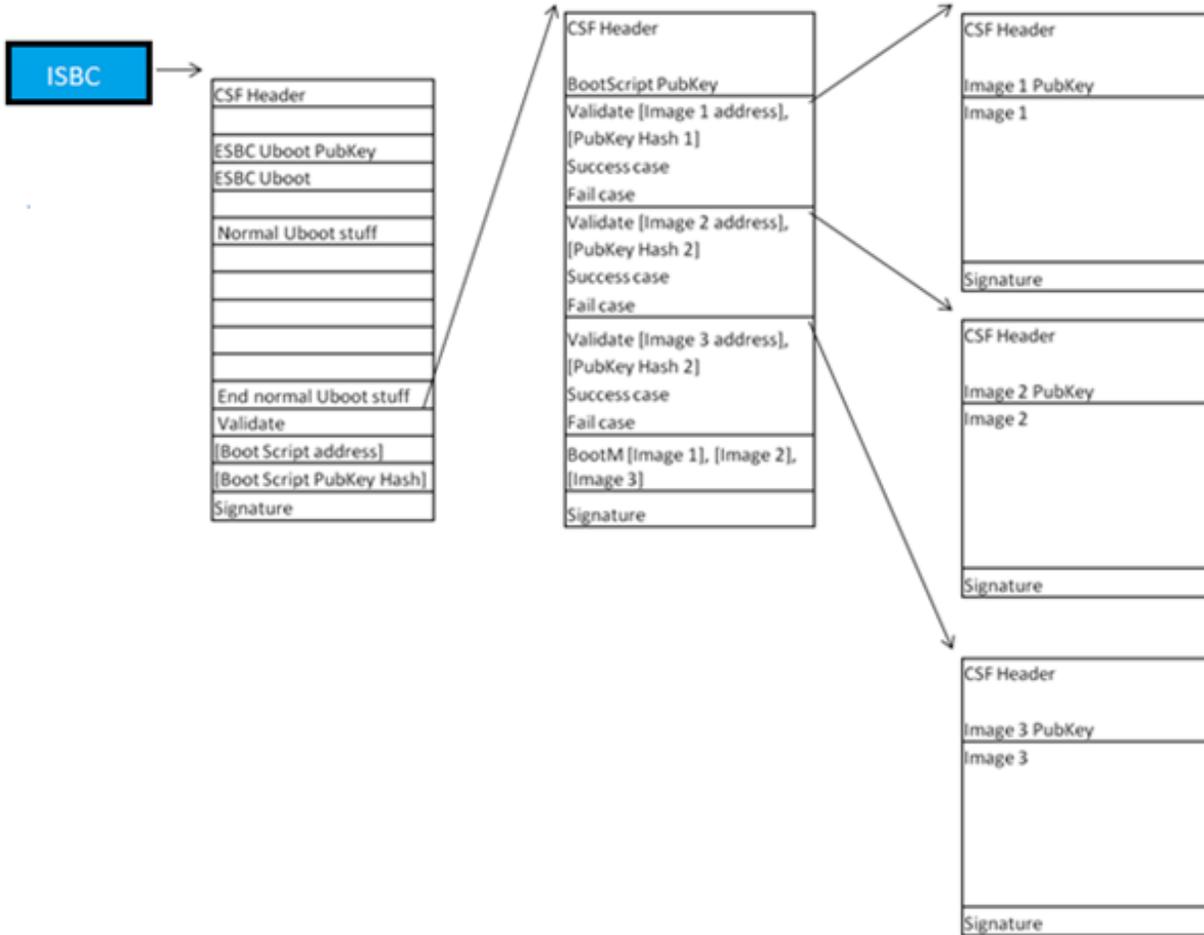


Figure 187. Secure boot flow (Chain of Trust)

#### 8.2.5.1.2.1 Sample Boot Script

A sample boot script would look like:

```

...
esbc_validate <Img1 header addr> <pub_key hash>
esbc_validate <Img2 header addr> <pub_key hash>
esbc_validate <Img3 header addr> <pub_key hash>
...
bootm <img1 addr> <img2 addr> <img3 addr>

```

##### 8.2.5.1.2.1.1 esbc\_validate command

esbc\_validate img\_hdr [pub\_key\_hash]

**Input arguments:**

`img_hdr` - Location of CSF Header of the image to be validated

`pub_key_hash` - hash of the public key used to verify the image. This is optional parameter. If not provided, code makes the assumption that the key pair used to sign the image is same as that used with ISBC. So the hash of the key in the header is checked against the hash available in SRK fuse for verification.

**Description:**

The command would do the following:

- Perform CSF header validation on the address passed in the image header. During parsing of the header, image address is stored in an environment variable which is later used in source command in default secure boot command.
- Signature checks on the image

### 8.2.5.1.2.1.2 *esbc\_halt command*

`esbc_halt` (no arguments)

**Description:**

The command would do the following:

This command puts core in spin loop.

After successful validation of images, `bootm` command in `bootscript` should execute and control should never reach back to `u-boot`. If somehow, control reaches back to `u-boot` (eg. `bootm` not present in `bootscript`), core should just spin.

## 8.2.5.1.3 Chain of Trust with Confidentiality

To establish chain of trust with confidentiality, cryptographic blob mechanism can be used. In this chain of trust, validated image is allowed to use the One Time Programmable Master Key to decrypt system secrets.

Two `bootscripts` are to be used. First `encap bootscripts` is used which creates a blob of the LINUX images and saves them. After this the system is booted after replacing the `encap bootscript` with `decap bootscript` which decapsulates the blobs and boot the LINUX with the images.

Figures below show the Chain of trust with confidentiality (Encapsulation and Decapsulation).

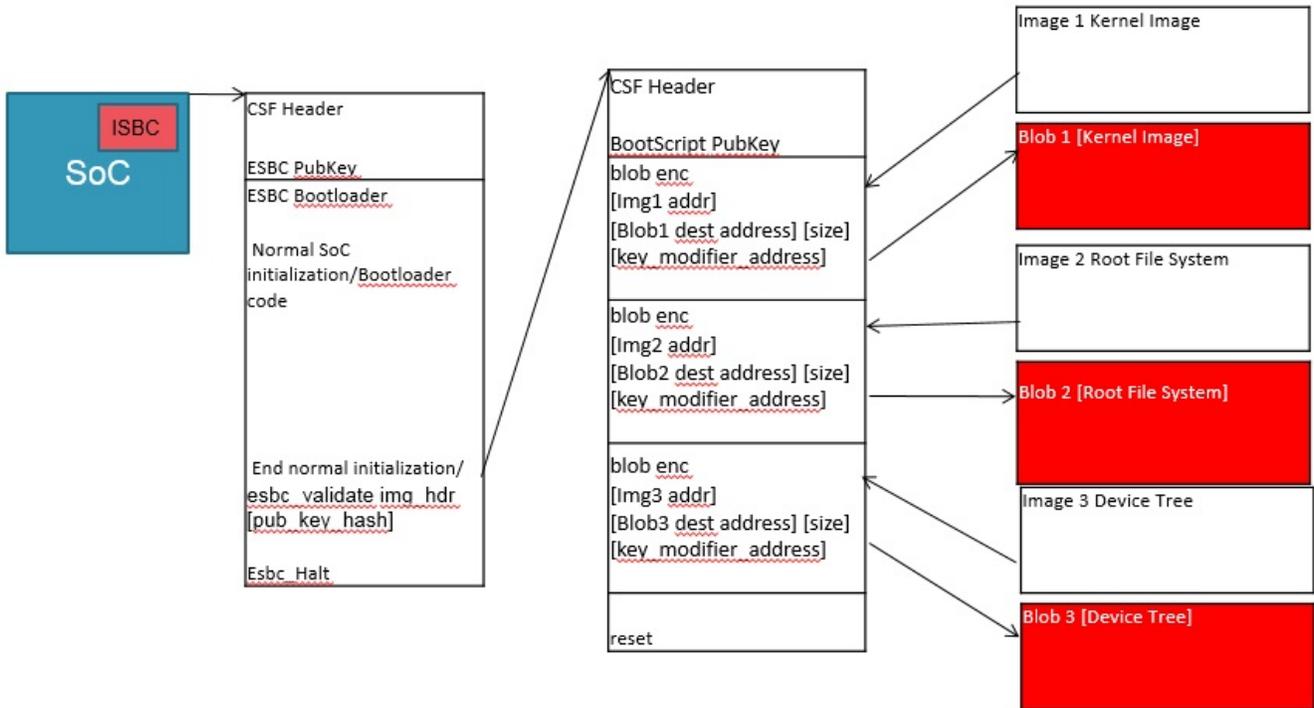


Figure 188. Chain of Trust with Confidentiality (Encapsulation)

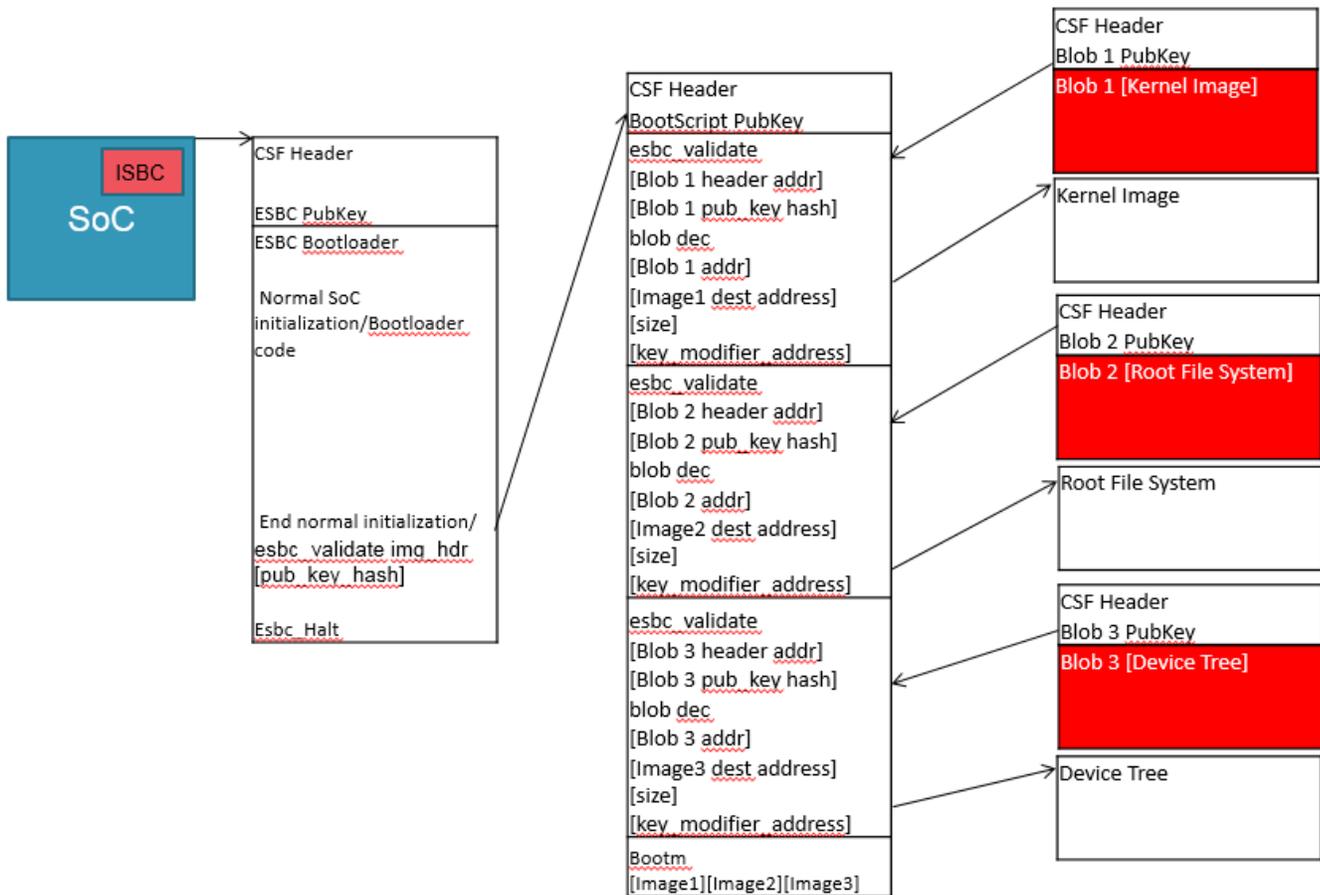


Figure 189. Chain of Trust with Confidentiality (Decapsulation)

### 8.2.5.1.3.1 Sample Encap Boot Script

A sample encap boot script would look like:

```

...
blob enc <Img1 addr> <Img1 dest addr> <Img1 size> <key_modifier address>
erase <encap Img1 addr> +<encap Img1 size>
cp.b <Img1 dest addr> <encap Img1 addr> <encap Img1 size>

blob enc <Img2 addr> <Img2 dest addr> <Img2 size> <key_modifier address>
erase <encap Img2 addr> +<encap Img2 size>
cp.b <Img2 dest addr> <encap Img2 addr> <encap Img2 size>

blob enc <Img3 addr> <Img3 dest addr> <Img3 size> <key_modifier address>
erase <encap Img3 addr> +<encap Img3 size>
cp.b <Img3 dest addr> <encap Img3 addr> <encap Img3 size>

...

```

#### 8.2.5.1.3.1.1 blob enc command

`blob enc <src location> <dst location> <length> <key_modifier address>`

#### Input arguments:

`src location` - Address of the image to be encapsulated

`dst location` - Address where the blob will be created

`length` - Size of the image to be encapsulated

`key_modifier address` - Address where a random number 16 bytes long(key modifier) is placed

### Description:

The command would do the following:

- Create a cryptographic blob of the image placed at `src location` and place the blob at `dst location`.

## 8.2.5.1.3.2 Sample Decap Boot Script

A sample decap boot script would look like:

```
...
blob dec <Img1 blob addr> <Img1 dest addr> <expected Img1 size> <key_modifier address>
blob dec <Img2 blob addr> <Img2 dest addr> <expected Img2 size> <key_modifier address>
blob dec <Img3 blob addr> <Img3 dest addr> <expected Img3 size> <key_modifier address>
...
bootm <Img1 dest addr> <Img2 dest addr> <Img3 dest addr>
```

### 8.2.5.1.3.2.1 blob dec command

`blob dec <src location> <dst location> <length> <key_modifier address>`

#### Input arguments:

`src location` - Address of the image blob to be decapsulated

`dst location` - Address where the decapsulated image will be placed

`length` - Expected Size of the image after decapsulation.

`key_modifier address` - Address where key modifier (Same as that used for Encapsulation) is placed

#### Description:

The command would do the following:

- Decapsulate the blob placed at `src location` and place the decapsulated data of expected size at `dst location`.

## 8.2.6 Next Executable (Linux Phase)

The bootloader (ESBC) finishes the platform initialization and passed control to the Linux image. The boot-chain can be further extended to be able to sign application which would be running on Linux prompt. Further RTIC can be integrated to verify memory regions using Security Engine (SEC) during run time.

## 8.2.7 CST Tool

### 8.2.7.1 KEY GENERATION

#### 8.2.7.1.1 gen\_keys

This utility generates a RSA public and private key pair using OPENSSL APIs. The key pair consists of 3 parts: N, E and D.

N – Modulus

E – Encryption exponent

D – Decryption exponent

**Public Key** - It is a combination of E and N components.

**Private Key** - It is a combination of D and N components.

It is the OEM's responsibility to tightly control access to the RSA private signature key. If this key is ever exposed, attackers will be able to generate alternate images that will pass secure boot. If this key is ever lost, the OEM will be unable to update the image.

### Features

- The application allows the user to generate 3 sizes keys. The sizes allowed are - 1024 bits, 2048 bits and 4096 bits.
- It generates RSA key pairs in PEM format.
- Keys are generated and stored in the files. User can provide filenames through command line option.

### Usage

`./gen_keys [OPTION] SIZE`

SIZE refers to size of public key in bits. (Modulus size).

Sizes supported -- 1024, 2048, 4096. The generated keys would be in PEM format.

Options:

`-h,--help` Usage of the command

`-k,--pubkey` File where Public key would be stored in PEM format(default = `srk.pub`)

`-p,--privkey` File where Private key would be stored in PEM format(default = `srk.priv`)

### Usage Example

```
$./gen_keys 1024

#-----#
#-----#
#----- CST (Code Signing Tool) Version 2.0 -----#
#-----#
#-----#

=====
This product includes software developed by the OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)
This product includes cryptographic software written by
Eric Young (eay@cryptsoft.com)
=====

Generated SRK pair stored in :
 PUBLIC KEY srk.pub
 PRIVATE KEY srk.priv
```

```
$./gen_keys 4096 -k my.pub -p my.pri

#-----#
#-----#
#----- CST (Code Signing Tool) Version 2.0 -----#
#-----#
#-----#

=====
This product includes software developed by the OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)
This product includes cryptographic software written by
Eric Young (eay@cryptsoft.com)
=====
```

```
Generated SRK pair stored in :
PUBLIC KEY my.pub
PRIVATE KEY my.pri
```

### 8.2.7.1.2 gen\_drv\_drbg

This utility in the Code Signing Tool inserts hamming code in a user defined 64b hexadecimal string, or generate a 64b hexadecimal random number and inserts the hamming code in it which can be used as Debug Response Value.

**NOTE**

For random number generation, Hash\_DRBG library is used. The Hash\_DRBG is an implementation of the NIST approved DRBG(Deterministic Random Bit Generator), specified in SP800-90A. The entropy source is the Linux /dev/random.

**Features:**

- Generates random numbers, which can be used if user defined string is not provided, to generate Debug Response value.
- Calculates and embeds the hamming code in the hexadecimal string.

**Usage:**

`./gen_drv_drbg <Hamming_algo> [string]`

Hamming\_algo : Platforms

A1 : T10xx, T20xx, T4xxx, P4080rev1, B4xxx

A2 : LSx

B : P10xx, P20xx, P30xx, P4080rev2, P4080rev3, P50xx, BSC913x, C29x

string : 8 byte string

In case string is not specified, the utility generates an 8 byte random number and embeds hamming code in it.

**Usage Example:**

```
$./gen_drv_drbg A2

#-----#
#-----#
#----- CST (Code Signing Tool) Version 2.0 -----#
#-----#
#-----#

Input string not provided
Generating a random string

* Hash_DRBG library invoked
* Seed being taken from /dev/random

Random Key Generated is:
f4bfc65e16284dbb
DRV[63:0] after Hamming Code is:
f4bfc65f16294daf
NAME | BITS | VALUE
-----|-----|-----
```

```
DRV 0 | 63 - 32 | f4bfc65f
DRV 1 | 31 - 0 | 16294daf
```

```
$./gen_drv_drbg A2 1652afe595631dec
```

```
#-----#
#-----#
#----- CST (Code Signing Tool) Version 2.0 -----#
#-----#
#-----#
```

DRV[63:0] after Hamming Code is:

1652afe495631cea

NAME	BITS	VALUE
DRV 0	63 - 32	1652afe4
DRV 1	31 - 0	95631cea

### 8.2.7.1.3 gen\_otpmk\_drbg

This utility in the Code Signing Tool inserts hamming code in a user defined 256b hexadecimal string, or generate a 256b hexadecimal random number and inserts the hamming code in it which can be used as OTPMK value.

#### NOTE

For random number generation, Hash\_DRBG library is used. The Hash\_DRBG is an implementation of the NIST approved DRBG(Deterministic Random Bit Generator), specified in SP800-90A. The entropy source is the Linux /dev/random.

#### Features:

- Generates random numbers, which can be used if user defined string is not provided, to generate OTPMK value.
- Calculates and embeds the hamming code in the hexadecimal string.

#### Usage:

```
./gen_otpmk_drbg <bit_order> [string]
```

<bit\_order> : (1 or 2) OTPMK Bit Ordering Scheme in SFP

1 : BSC913x, P1010, P3, P4, P5, C29x

2 : T1, T2, T4, B4, LSx

<string> : 32 byte string

In case string is not specified, the utility generates a 32 bytes random number and embeds hamming code in it.

#### Usage Example:

```
$./gen_otpmk_drbg 2
```

```
#-----#
#-----#
#----- CST (Code Signing Tool) Version 2.0 -----#
#-----#
#-----#
```

```
Input string not provided
Generating a random string
```

```
* Hash_DRBG library invoked
```

## Boot Loaders

### Secure Boot: PBL Based Platforms

```
* Seed being taken from /dev/random
```

```

OTPMK[255:0] is:
```

```
3feac02ce3583ad9077ab70f3a398cd71955f8bffa3191428cb25bb6bffb3113
```

NAME	BITS	VALUE
OTPMKR 0	255-224	3feac02c
OTPMKR 1	223-192	e3583ad9
OTPMKR 2	191-160	077ab70f
OTPMKR 3	159-128	3a398cd7
OTPMKR 4	127- 96	1955f8bf
OTPMKR 5	95- 64	fa319142
OTPMKR 6	63- 32	8cb25bb6
OTPMKR 7	31- 0	bffb3113

```
$./gen_otpmk_drbg 2 1234567856485626a6f6e6174858583847720673534a8958c983774b848438fe
```

```
#-----#
#-----#
#----- CST (Code Signing Tool) Version 2.0 -----#
#-----#
#-----#
```

```
OTPMK[255:0] is:
```

```
ce3c563856584664a6b66617485a5e3d46720673534a8958c983774b848438fe
```

NAME	BITS	VALUE
OTPMKR 0	255-224	ce3c5638
OTPMKR 1	223-192	56584664
OTPMKR 2	191-160	a6b66617
OTPMKR 3	159-128	485a5e3d
OTPMKR 4	127- 96	46720673
OTPMKR 5	95- 64	534a8958
OTPMKR 6	63- 32	c983774b
OTPMKR 7	31- 0	848438fe

## 8.2.7.2 CSF Header Generation

uni\_sign tool can be used for the following functions :

- CSF header generation along with signature for both ISBC and ESBC phase
- CSF header generation without signature if private key is not provided

### Usage:

**If INPUT file does not have ESBC = 1, uni\_sign invokes create\_hdr\_isbc else it will invoke create\_hdr\_esbc**

To view usage of tool:

```
$./uni_sign --help
```

```
#-----#
#-----#
#----- CST (Code Signing Tool) Version 2.0 -----#
#-----#
#-----#
```

Correct Usage of Tool is:

```
./create_hdr_isbc [options] <input_file>
 --verbose Display header Info after Creation
 --hash Print the SRK(Public key) hash.
 --img_hash Header is generated without Signature.
 Image Hash is stored in a separate file.
 --help Show the Help for Tool Usage.
```

<input\_file> Contains all information required by tool

```

* uni_sign is a wrapper script over the TOOL
* Correct Usage (Description as specified above):
*
* ./uni_sign [options] <input_file>
*

```

## 8.2.7.2.1 Default Usage

When uni\_sign is executed without any option i.e. only providing the input file as the argument, it parses the required fields from the input file and creates the CSF header as described in 5.2 along with the Public Key/ SRK Hash, Digital Signature and SG Table to create a combined binary.

Usage :: \$./uni\_sign <input\_file>

### Example

```
$./uni_sign input_uboot_secure

=====
This product includes software developed by the OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)
This product includes cryptographic software written by
Eric Young (eay@cryptsoft.com)
=====

Key Hash :
9288723a0253229000a70bcbaa9d3aa1acb70f6369e1e81c9225319d9a364e2a

HEADER file hdr_uboot.out created
```

### 8.2.7.2.1.1 Sample Input File and Output

Sample input file to generate CSF Header is as follows –

```

Specify the platform. [Mandatory]
Choose Platform - 1010/1040/2041/3041/4080/5020/5040/9131/9132/9164/4240/C290
PLATFORM=4240
ESBC Flag. Specify ESBC=0 to sign u-boot and ESBC=1 to sign ESBC images.(default is 0)
ESBC=0

Entry Point/Image start address field in the header.[Mandatory]
(default=ADDRESS of first file specified in images)
ENTRY_POINT=cffffffc

```

## Boot Loaders

### Secure Boot: PBL Based Platforms

```
Specify the file name of the keys seperated by comma.
The number of files and key select should lie between 1 and 4 for 1040 and C290.
For rest of the platforms only one key is required and key select should not be provided.

USAGE (for 4080/5020/5040/3041/2041/1010/913x): PRI_KEY = <key1.pri>
USAGE (for 1040/C290/9164/4240): PRI_KEY = <key1.pri>, <key2.pri>, <key3.pri>, <key4.pri>

PRI_KEY (Default private key :srk.pri) - [Optional]
PRI_KEY=srk.pri
PUB_KEY (Default public key :srk.pub) - [Optional]
PUB_KEY=srk.pub
Please provide KEY_SELECT(between 1 to 4) (Required for 1040/C290/9164/4240 only) - [Optional]
KEY_SELECT=

Specify SG table address, only for (2041/3041/4080/5020/5040) with ESBC=0 - [Optional]
SG_TABLE_ADDR=

Specify the target where image will be loaded. (Default is NOR_16B) - [Optional]
Only required for Non-PBL Devices (1010/1040/9131/9132i/C290)
Select from - NOR_8B/NOR_16B/NAND_8B_512/NAND_8B_2K/NAND_8B_4K/NAND_16B_512/NAND_16B_2K/
NAND_16B_4K/SD/MMC/SPI
IMAGE_TARGET=

Specify IMAGE, Max 8 images are possible. DST_ADDR is required only for Non-PBL Platform.
[Mandatory]
USAGE : IMAGE_NO = {IMAGE_NAME, SRC_ADDR, DST_ADDR}
IMAGE_1={u-boot.bin,cff40000,ffffffff}
IMAGE_2={,,}
IMAGE_3={,,}
IMAGE_4={,,}
IMAGE_5={,,}
IMAGE_6={,,}
IMAGE_7={,,}
IMAGE_8={,,}

Specify OEM AND FSL ID to be populated in header. [Optional]
e.g FSL_UID=11111111
FSL_UID=
OEM_UID=

Specify the file names of csf header and sg table. (Default :hdr.out) [Optional]
OUTPUT_HDR_FILENAME=hdr_uboot.out

Specify the file names of hash file and sign file.
HASH_FILENAME=img_hash.out
INPUT_SIGN_FILENAME=sign.out

Specify the signature size.It is mandatory when neither public key nor private key is specified.
Signature size would be [0x80 for 1k key, 0x100 for 2k key, and 0x200 for 4k key].
SIGN_SIZE=

Specify the output file name of sg table. (Default :sg_table.out). [Optional]
Please note that OUTPUT SG BIN is only required for 2041/3041/4080/5020/5040 when ESBC flag is
not set.
OUTPUT_SG_BIN=

Following fields are Required for 4240/9164/1040/C290 only

Specify House keeping Area
Required for 4240/9164/1040/C290 only when ESBC flag is not set. [Mandatory]
```

```

HK_AREA_POINTER=bff00000
HK_AREA_SIZE=00010000

Following field Required for 4240/9164/1040/C290 only
Specify Secondary Image Flag. (0 or 1) - [Optional]
(Default is 0)
SEC_IMAGE=

```

**Table 173. Description of fields.**

Field	Field Description
PLATFORM	To identify the platform/SoC for which CF Header needs to be created.
ESBC	Don't set this flag when code signing is being performed on the image directly verified by the ISBC. For later images in the chain of trust, set this flag.
ENTRY_POINT	Entry Point address / Image start address field in the header.
PRI_KEY	Private key filename to be used for signing the image. (File has to be in PEM format) (default = srk.pri generated by gen_keys command) FILE1 [,FILE2, FILE3, FILE4]. Multiple key support for Trust Arch v2.x devices only.
PUB_KEY	Public key filename in PEM format. (default = srk.pub generated by gen_keys) FILE1 [,FILE2, FILE3, FILE4]. Multiple key support for Trust Arch v2.x devices only.
KEY_SELECT	Specify the key to be used in signature generation when more than one key has been given as input. (Default=1, first key will be selected)
IMAGE_1 - IMAGE_8	Create Entries for SG Table in the format { IMAGE_NAME, SRC_ADDR, DST_ADDR }
OEM_UID	OEM UID to be populated in the header.
OEM_UID_1	OEM UID 1 to be populated in the header. Required Only for ls1
FSL_UID	FSL UID to be populated in header.
FSL_UID_1	FSL UID 1 to be populated in header.Required Only for ls1
HK_AREA_POINTER	House Keeping Area Starting Pointer Required by Sec (Required for Trust Arch v2.x devices only when esbc option is not provided)
HKAREA_SIZE	House Keeping Area Size (Required for Trust Arch v2.x devices only when esbc option is not provided)
OUTPUT_HDR_FILENAME	Name of the combined header binary to be created by tool

*Table continues on the next page...*

**Table 173. Description of fields. (continued)**

Field	Field Description
SG_TABLE_ADDR	Specify SG_TABLE Address where Scatter Gather table is present for 2041/3041/4080/5020/5040 when ESBC=0.
OUTPUT_SG_BIN	Specify the output file name of sg table.
IMAGE_TARGET	Specify the target where image will be loaded. Ex:NOR_8B/NOR_16B/NAND_8B_512/NAND_8B_2K/ NAND_8B_4K/ NAND_16B_512/NAND_16B_2K/ NAND_16B_4K/SD/MMC/SPI
SEC_IMG	Flag for Secondary Image. Required for Trust Arch v2.x devices only
MP_FLAG	Specify Manufacturing Protection Flag. Available for LS1 only.
VERBOSE	Specify Verbose option. Contents of header generated will be printed.

### 8.2.7.2.2 Verbose Mode (--verbose)

Verbose mode can be used to display extra information while creating the header. If selected, along with header creation, the tool will also display information about Output header and SG\_TABLE entries..

Usage :: `./uni_sign --verbose <input_file>`

#### Example

```

$./uni_sign --verbose input_uboot_secure

=====
This product includes software developed by the OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)
This product includes cryptographic software written by
Eric Young (eay@cryptsoft.com)
=====

Key Hash :
9288723a0253229000a70bcbaa9d3aa1acb70f6369e1e81c9225319d9a364e2a

Image Hash :6c0048c92079b5e44152634a16d2463b808b5fd6ae23e7c42dcd30f49efafa8e
***** HEADER *****
barker:0x68392781
srk_table_offset 200
srk_table_flag(8) : 1
srk_sel(8) : 1
num_srk_entries(16) : 3
psign 1410, length 128
uid_flag 0
sfp_wp(8) : 0
sec_image_flag(8) : 0
uid_flag(16) : 0
psgtable 1400 num_entries 1
img start cfffffff
FSL UID 0
OEM UID 0

```

```
sg_flag 1
hkptr bff00000
hksize 10000
***** SG TABLE *****
no of entries 1
entry 0 len 786432 ptr cff40000
SIGNATURE file sign.out created
HEADER file hdr_uboot.out created
```

### 8.2.7.2.3 Public Key/ SRK Hash Generation Only (--hash)

The Hash of the Public Key or SRK Table as selected by user in the input file while signing the images needs to be fused in the SFP block. So if user wants to get the value of SRK Hash, this option can be used.

Usage :: `./uni_sign --hash <input_file>`

#### Example

```
./uni_sign --hash input_uboot_secure

=====
This product includes software developed by the OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)
This product includes cryptographic software written by
Eric Young (eay@cryptsoft.com)
=====

Key Hash :
478c14568d8f76e822a2d483489a5ed9752c7c453b1fe5351f085a57fae2a30f
```

### 8.2.7.2.4 ISBC Key Extension (IE)

#### 8.2.7.2.4.1 Introduction

The ISBC Key Extension feature allows the user to extend the ISBC and the number of keys available for signature validation. The ISBC uses a key directly bound to the silicon via the SRKH, the ISBC extension code (added to downstream images in a chain of trust) use IE\_Keys, which are validated by the ISBC.

#### 8.2.7.2.4.2 How it works

If IE feature is enabled in input file, the CST signs the image along with a number of public keys. Logically, it will be used when signing Boot 1 (bootloader), so that the bootloader and downstream images in the chain of trust can use keys which aren't directly bound to the silicon via the SRKH. Decoupling the chain of trust from the hardware super root keys minimizes the need to perform hardware key revocation.

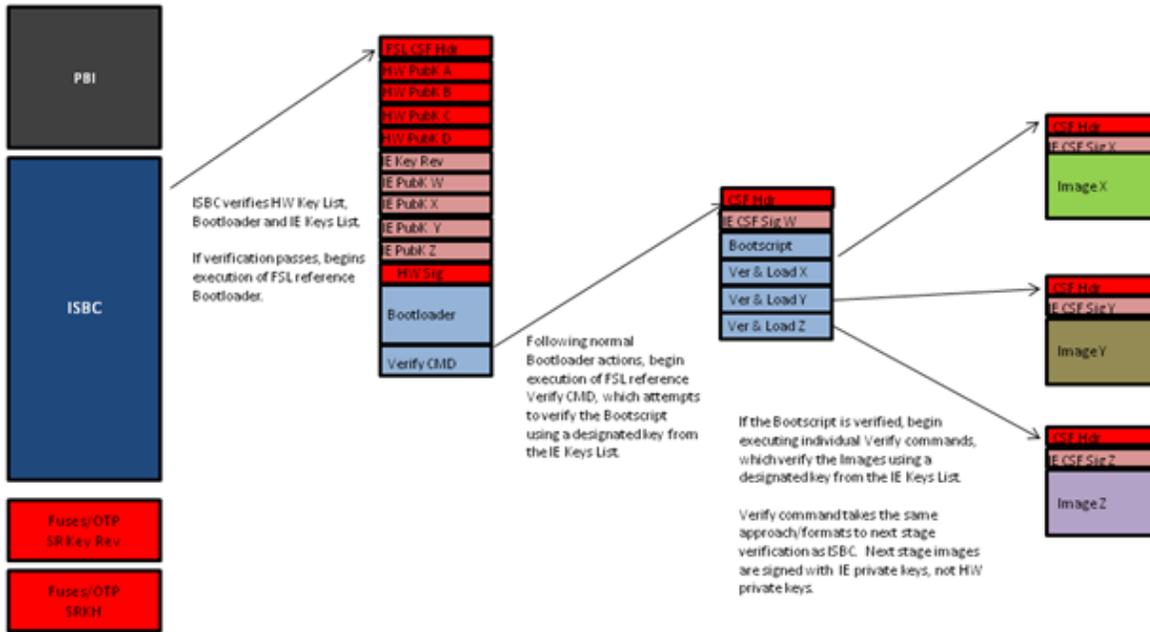


Figure 190. Execution and Verification of Images using Key\_Ext feature.

**NOTE**

Next stage images are signed with corresponding pair of Extension private keys list, not HW private keys.

Key Extension feature is applicable only for NOR secure Boot. It is not applicable for RAMBOOT (where data has to be copied onto RAM, eg:- NAND, SD, SPI)

### 8.2.7.2.4.3 IE Key Structure

Table 174. IE Key Structure which is embedded in header and placed in memory.

Offset	Data Bits [0:31]
0x00-0x03	This 32 bit word can be used to represent which keys from the table below have been revoked and are no longer available for use. Each bit represents 1 Key, Bit 0 represents Key 1 in the table ....Bit 31 is the 32nd key in the table
0x04-0x07	Total number of keys (Max N = 32 as 32 bit key revocation field is provided)
0x08-0x0b	Key 1 length.
0x0c-0x40b	Key 1 value.
0x40c-0x40f	Key 2 length.
0x410-0x80f	Key 2 value.

Table continues on the next page...

**Table 174. IE Key Structure which is embedded in header and placed in memory. (continued)**

Offset	Data Bits [0:31]
-	-
-	Key N value

### 8.2.7.2.4.4 Sample Input File and Output

This file is same as file described above in <link to 4.1.2> except fields required for IE Key extension highlighted in red.

```

Specify the platform. [Mandatory]
Choose Platform - 1040/2080/2041/3041/4080/5020/5040/4860/4240/LS1
PLATFORM=1040
ESBC Flag. Specify ESBC=0 to sign u-boot and ESBC=1 to sign ESBC images.(default is 0)
ESBC=0
ESBC Header address. It contains address where ESBC header is loaded in memory.
ESBC_HDRADDR=c0b00000

Entry Point/Image start address field in the header.[Mandatory]
(default=ADDRESS of first file specified in images)
ENTRY_POINT=cffffffc

Specify the file name of the keys seperated by comma.
The number of files and key select should lie between 1 and 4 for 1040/2080 and C290.
For rest of the platforms only one key is required and key select should not be provided.

USAGE (for 4080/5020/5040/3041/2041/1010/913x): PRI_KEY = <key1.pri>
USAGE (for 1040/2080/C290/4860/4240): PRI_KEY = <key1.pri>,<key2.pri>,<key3.pri>,<key4.pri>

PRI_KEY (Default private key :srk.pri) - [Optional]
PRI_KEY=srk.pri
PUB_KEY (Default public key :srk.pub) - [Optional]
PUB_KEY=srk.pub
Please provide KEY_SELECT(between 1 to 4) (Required for 1040/2080/C290/4860/4240 only) -
[Optional]
KEY_SELECT=

Specify the file name of the extension keys seperated by comma.
USAGE : IE_KEY = <key1.pub>,<key2.pub>,<key3.pub>,<key4.pub>,<key5.pub>
IE_KEY=<iekey1k_1.pub>,<iekey1k_2.pub>,<iekey1k_3.pub>,<iekey2k_1.pub>,<iekey2k_2.pub>,<iekey2k_3.
pub>,<iekey4k_1.pub>,<iekey4k_2.pub>

Please provide Revoke keys. - [Optional]
Provide key numbers from available ie keys to be revoked. Max n-1 keys can be revoked. n is
total number of IE keys.
Lsb represents key0 and MSb represents key 31. So total 32 keys are supported.
IE_REVOC=1,7

Specify SG table address, only for (2041/3041/4080/5020/5040) with ESBC=0 - [Optional]
SG_TABLE_ADDR=

Specify the target where image will be loaded. (Default is NOR_16B) - [Optional]
Only required for Non-PBL Devices (1010/9131/9132/C290)
Select from - NOR_8B/NOR_16B/NAND_8B_512/NAND_8B_2K/NAND_8B_4K/NAND_16B_512/NAND_16B_2K/
NAND_16B_4K/SD/MMC/SPI

```

Boot Loaders

Secure Boot: PBL Based Platforms

```

IMAGE_TARGET=

Specify IMAGE, Max 8 images are possible. DST_ADDR is required only for Non-PBL Platform.
[Mandatory]
In case using IE_KEY, Max 7 images are possible. [Mandatory]
USAGE : IMAGE_NO = {IMAGE_NAME, SRC_ADDR, DST_ADDR}
IMAGE_1={u-boot.bin,cff40000,ffffffff}
IMAGE_2={,,}
IMAGE_3={,,}
IMAGE_4={,,}
IMAGE_5={,,}
IMAGE_6={,,}
IMAGE_7={,,}

Specify OEM AND FSL ID to be populated in header. [Optional]
e.g FSL_UID=11111111
FSL_UID=
OEM_UID=

Specify the file names of csf header and sg table. (Default :hdr.out) [Optional]
OUTPUT_HDR_FILENAME=hdr_uboot.out

Specify the file names of hash file and sign file.
HASH_FILENAME=img_hash.out
INPUT_SIGN_FILENAME=sign.out

Specify the signature size.It is mandatory when neither public key nor private key is specified.
Signature size would be [0x80 for 1k key, 0x100 for 2k key, and 0x200 for 4k key].
SIGN_SIZE=

Specify the output file name of sg table. (Default :sg_table.out). [Optional]
Please note that OUTPUT SG BIN is only required for 2041/3041/4080/5020/5040 when ESBC flag is
not set.
OUTPUT_SG_BIN=

Following fields are Required for 4240/4860/1040/2080/C290 only

Specify House keeping Area
Required for 4240/4860/1040/2080/C290 only when ESBC flag is not set. [Mandatory]
HK_AREA_POINTER=bff00000
HK_AREA_SIZE=00010000

Following field Required for 4240/4860/1040/2080/C290 only
Specify Secondary Image Flag. (0 or 1) - [Optional]
(Default is 0)
SEC_IMAGE=

```

**Table 175. Description of new fields introduced.**

Field	Field Description
ESBC_HDRADDR	ESBC Header address. It contains location of ESBC header in the memory
<i>Table continues on the next page...</i>	

**Table 175. Description of new fields introduced. (continued)**

Field	Field Description
IE_KEY	Extension Public key filenames to be used by further level images. (File has to be in PEM format) FILE1 [,FILE2, FILE3, FILE4].
IE_REVOC	Revoked keys numbers from available ie keys. If a key is compromised then this feature helps to avoid that key usage. Max n-1 keys can be revoked. n is total number of IE keys and less than equal to 32.Ex.[1,3,5]

**OUTPUT**

```

 Size of IE Key Structure Memory address of IE structure
 ↗ ↘
0001400: 0000 2028 0000 000f c8b0 1500 ffff ffff
0001410: 000c 0000 0000 000f cff4 0000 ffff ffff
0001420: 0000 0000 0000 0000 0000 0000 0000 0000
0001430: 0000 0000 0000 0000 0000 0000 0000 0000
0001440: 0000 0000 0000 0000 0000 0000 0000 0000
0001450: 0000 0000 0000 0000 0000 0000 0000 0000
0001460: 0000 0000 0000 0000 0000 0000 0000 0000
0001470: 0000 0000 0000 0000 0000 0000 0000 0000
0001480: 0000 0000 0000 0000 0000 0000 0000 0000
0001490: 0000 0000 0000 0000 0000 0000 0000 0000
00014a0: 0000 0000 0000 0000 0000 0000 0000 0000
00014b0: 0000 0000 0000 0000 0000 0000 0000 0000
00014c0: 0000 0000 0000 0000 0000 0000 0000 0000
00014d0: 0000 0000 0000 0000 0000 0000 0000 0000
00014e0: 0000 0000 0000 0000 0000 0000 0000 0000
00014f0: 0000 0000 0000 0000 0000 0000 0000 0000
0001500: 0000 0001 0000 0008 0000 0100 b277 ef63
0001510: bde1 50c5 29a7 d2ab abe7 52d5 3fcb 8c00
0001520: 02b7 cc0b bdc8 dbba f966 6b9d a9c2 d8e2
0001530: e05d 2313 99f8 4b1a 9198 aa76 68d1 b452
0001540: 9b23 6d46 5ac0 4554 cf01 6b40 827d 12ac
0001550: 9b7f 9d25 13a6 c5ca 8f8c af58 d29b 865f
0001560: 0969 33cc d5b0 d90a f5ee 170c e896 3b1d
0001570: 086e 9f45 31f5 c843 4038 2137 c37f 4fab
0001580: 9e78 f8e8 f7f8 22f5 5759 deab 0000 0000

```

Highlighted fields shows IE structure is embedded in the CSF header.

### 8.2.7.2.4.5 Generate Header for Next Level Images (bootscript, rootfs, dtb, linux).

IE key table generated in previous is embedded along with the CSF header for u-boot. Boot ROM code verifies these keys along with the bootloader. For the rest of the images in the chain of trust, user can use the keys in the IE key table. The IE

## Boot Loaders

### Secure Boot: PBL Based Platforms

Key Table is in the memory already, the sample input file needs to have the IE Key number to be used.(IE\_KEY\_SEL). The corresponding private key of the file needs to be provided for signature to be generated (PRI\_KEY).

This sample file is same as file described above in <link to 4.1.2> except fields required for IE Key extension highlighted in red.

### CSF Header for bootscript

```

Specify the platform. [Mandatory]
Choose Platform - 1040/2080/2041/3041/4080/5020/5040/4860/4240/LS1
PLATFORM=1040
ESBC Flag. Specify ESBC=0 to sign u-boot and ESBC=1 to sign ESBC images.(default is 0)
ESBC=1

Entry Point/Image start address field in the header.[Mandatory]
(default=ADDRESS of first file specified in images)
ENTRY_POINT=e8a00000

Specify the file name of the keys seperated by comma.
The number of files and key select should lie between 1 and 4 for 1040/2080 and C290.
For rest of the platforms only one key is required and key select should not be provided.

USAGE (for 4080/5020/5040/3041/2041/1010/913x): PRI_KEY = <key1.pri>
USAGE (for 1040/2080/C290/4860/4240): PRI_KEY = <key1.pri>, <key2.pri>, <key3.pri>, <key4.pri>

PRI_KEY (Default private key :srk.pri) - [Optional]
PRI_KEY=iekey4k_2.pri
PUB_KEY (Default public key :srk.pub) - [Optional]
PUB_KEY=
Please provide KEY_SELECT(between 1 to 4) (Required for 1040/2080/C290/9164/4240 only) -
[Optional]
KEY_SELECT=

Specify SG table address, only for (2041/3041/4080/5020/5040) with ESBC=0 - [Optional]
SG_TABLE_ADDR=

Specify IE_KEY to be used for signature verification. [Mandatory]
IE_KEY_SEL=8

Specify the target where image will be loaded. (Default is NOR_16B) - [Optional]
Only required for Non-PBL Devices (1010/9131/9132/C290)
Select from - NOR_8B/NOR_16B/NAND_8B_512/NAND_8B_2K/NAND_8B_4K/NAND_16B_512/NAND_16B_2K/
NAND_16B_4K/SD/MMC/SPI
IMAGE_TARGET=

Specify IMAGE, Max 8 images are possible. DST_ADDR is required only for Non-PBL Platform.
[Mandatory]
In case using IE_KEY, Max 1 image is possible. [Mandatory]
USAGE : IMAGE_NO = {IMAGE_NAME, SRC_ADDR, DST_ADDR}
IMAGE_1={bootscript,e8a00000,ffffffff}
IMAGE_2={,,}
IMAGE_3={,,}
IMAGE_4={,,}
IMAGE_5={,,}
IMAGE_6={,,}
IMAGE_7={,,}
IMAGE_8={,,}

```

```
Specify OEM AND FSL ID to be populated in header. [Optional]
e.g FSL_UID=11111111
FSL_UID=
OEM_UID=

Specify the file names of csf header. (Default :hdr.out) [Optional]
OUTPUT_HDR_FILENAME=hdr_bs.out

Specify the file names of hash file and sign file.
HASH_FILENAME=img_hash.out
INPUT_SIGN_FILENAME=sign.out

Specify the signature size.It is mandatory when neither public key nor private key is specified.
Signature size would be [0x80 for 1k key, 0x100 for 2k key, and 0x200 for 4k key].
SIGN_SIZE=0x200

Specify the output file name of sg table. (Default :sg_table.out). [Optional]
Please note that OUTPUT SG BIN is only required for 2041/3041/4080/5020/5040 when ESBC flag is
not set.
OUTPUT_SG_BIN=

Following fields are Required for 4240/9164/1040/2080/C290 only

Specify House keeping Area
Required for 4240/9164/1040/2080/C290 only when ESBC flag is not set. [Mandatory]
HK_AREA_POINTER=
HK_AREA_SIZE=

Following field Required for 4240/9164/1040/2080/C290 only
Specify Secondary Image Flag. (0 or 1) - [Optional]
(Default is 0)
SEC_IMAGE=

```

**Table 176. Description of new fields introduced.**

Field	Field Description
IE_KEY_SEL	IE_KEY number for public key in IE Key table to be used for signature verification of ESBC image.

## OUTPUT

Given below is a snapshot of header generated in which highlighted fields indicates IE flag is ON and IE KEY SELECT i.e. key to be used to verify image is embedded in header.

```
000000: 6839 2781 0000 0000 0000 0000 0000 1400 h9'.....
000010: 0000 0100 e8a0 0000 0000 00ae e8a0 0000
000020: 0000 0000 0000 0000 0000 0000 0000 0000
000030: 0000 0001 0000 0002 0000 0000 0000 0000
```

Highlighted fields shows IE key select in CSF header.

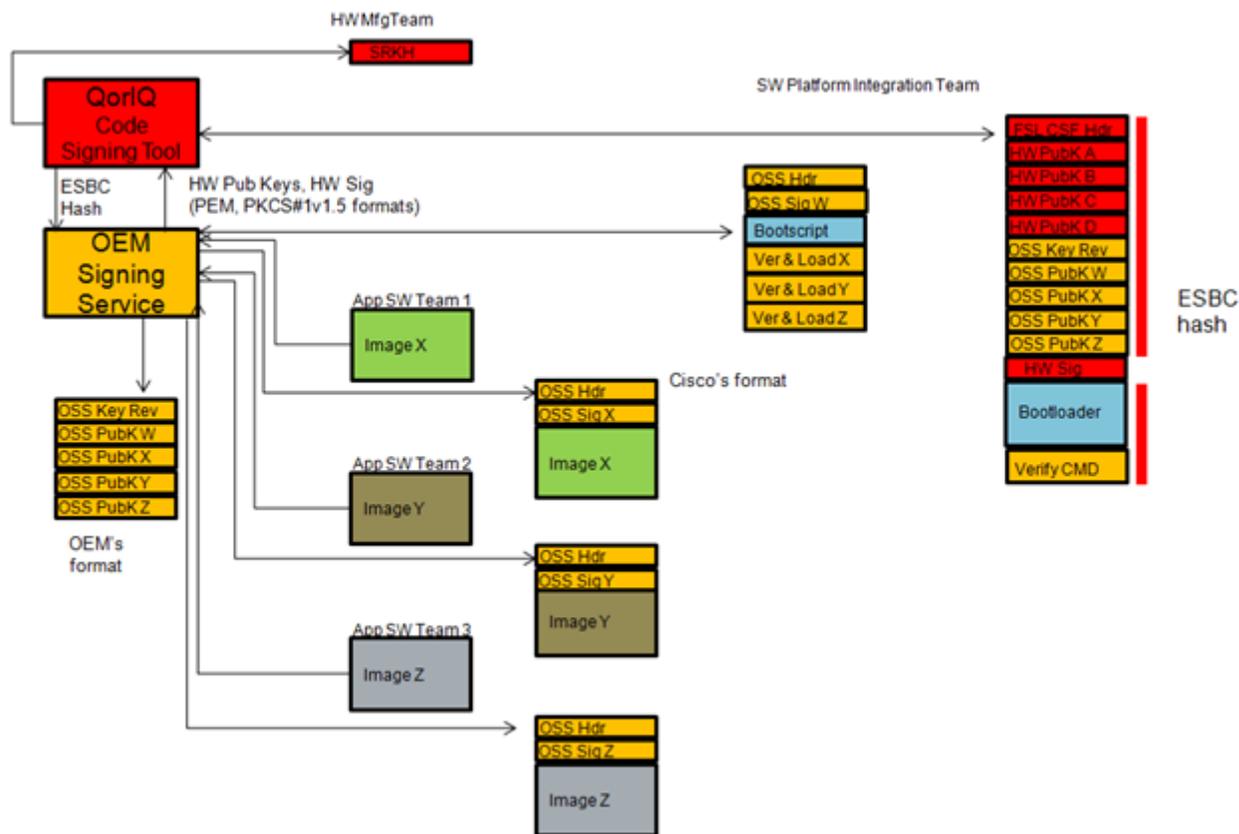
## 8.2.7.2.5 Image Hash Generation (--img\_hash)

### 8.2.7.2.5.1 Introduction

The -img\_hash generation feature provides OEMs with the ability to perform code signing in a secure environment which does not run the FSL Code Signing Tool.

When used in conjunction with the IE Key List feature, the user generates the IE key list and the list of hardware public keys (those bound to the silicon with the SRKH), and passes them to the CST for inclusion in the ESBC image hash calculation. The CST generates the appropriate CSF header, S/G table, and key lists, then calculates and exports the SHA256 hash. The OEM then RSA encrypts the hash with one of the private keys associated with the public key provided to verify the signature.

The signature, which must be in PKCS#1v1.5 format, is then appended to the ESBC. See section 4.7 for more information on appending.



### 8.2.7.2.5.2 Features

- Generates hash file in binary format which contains SHA256 hash of CSF header along with keys(SRK table, IE keys), SG table and its entries.
- Generates output header binary file based on the fields specified in input file.
- Output header binary file doesn't contain signature.
- Provides flexibility to manually append signature at the end of output header file. User's can use their own custom tool to generate the signature. The signature offset chosen in the header is such that the signature can be appended at the end of the header file.
- This option does not require private key to be provided. But the corresponding Public key from the public/private key pair must be provided to calculate correct SHA256 hash.

### Usage Example:

```
./uni_sign --img_hash input_uboot_nor_secure

=====
This product includes software developed by the OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)
This product includes cryptographic software written by
Eric Young (eay@cryptsoft.com)
=====

Key Hash :
f1f18e7eae28cee50a30f5ba5d270b3e71b2c7c8382507ca8e7e4c110547eb2

HASH file hash.out created
HEADER file esbc_hdr.out created
```

### 8.2.7.2.6 Help (--help)

It prints help menu describing various options available.

### 8.2.7.3 Code Signing Tool Walkthrough

#### Step-1)

Yocto installs the cst package at the following location:

```
tmp/sysroots/x86_64-linux/usr/bin/cst
```

OR

In the Yocto environment, the user can use below commands to rebuild cst:

1. bitbake cst-native -c cleanall
2. bitbake cst-native

#### Step-2)

```
cd tmp/sysroots/x86_64-linux/usr/bin/cst
```

gen\_keys and uni\_sign binaries are available in cst.

Note : LD\_LIBRARY\_PATH should be set to the library path in yocto workspace. <project\_folder\_path>/tmp/sysroots/x86\_64-linux/usr/lib

#### Step-3)

Generate private key public key pair -

```
./gen_keys 1024
```

#### NOTE

- Here, 1024 refers to the size of public key Modulus in bits.
- Other allowed sizes are - 2048 bits, 4096 bits.
- See help -

```
bash-2.05a$./gen_keys -h
```

#### Step-4)

Put all the images (limited by number 8) you want to sign using OPENSSL RSA APIs in current directory.

**Step-5)**

Execute the binary uni\_cfsign to generate signature over CSF header and ESBC images.

**CSF Header Generation**

Example taken for B4860:

```

$./uni_sign input_files/uni_sign/b4860/input_uboot_nor_secure

=====
This product includes software developed by the OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)
This product includes cryptographic software written by
Eric Young (eay@cryptsoft.com)
=====

Key Hash :
a9bb28a23c641e13b58c19a7fc48dfd8660a29895ebbc8bd0beba432e04c0785

HEADER file hdr_uboot.out created

```

**The header would look like this:**

```

00000000 68 39 27 81 00 00 02 00 00 00 01 00 00 00 14 00 |h9'.....|
00000010 00 00 00 80 00 00 16 00 00 00 00 01 11 07 f0 00 |.....|
00000020 00 00 00 01 00 00 00 01 11 11 11 11 99 99 99 99 |.....|
00000030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00000200 cc fe 4d fc 20 c1 6d ba 77 42 51 c2 4d c4 5b 45 |..M. .m.wBQ.M.[E|
00000210 41 e2 88 a9 55 d0 49 7b 86 fe 5a 85 89 68 b7 db |A...U.I{..Z..h..|
00000220 89 ef b7 2d 2a 1f 5b 74 4d 9c 7a c7 54 a9 b0 ff |...-*. [tM.z.T...|
00000230 cf a6 1c ed 3d f3 de 8d cc 91 ae 5f 60 b4 88 ab |...=....._`...|
00000240 a5 70 0b 20 73 30 75 38 5b 1b 51 22 e7 2f fd a6 |.p. s0u8[.Q"/..|
00000250 65 00 07 4a 78 5d 1e ee 81 b8 a6 c4 81 e5 bc be |e..Jx].....|
00000260 dc 64 09 c0 07 91 7a 36 ab 7c 0c e0 ab b1 01 bb |.d....z6.|.....|
00000270 de a0 e2 56 65 0a 29 73 67 57 d3 ba 1f 52 7a 5f |...Ve.)sgW...Rz_|
00000280 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
000002f0 00 00 00 00 00 00 00 00 00 00 00 00 01 00 01 |.....|
00000300 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00001400 b3 6e 9e d5 3a 47 c6 44 4e 09 ff 29 0d a5 a1 c3 |.n...:G.DN..)....|
00001410 32 f3 b5 50 c6 42 f0 5b 59 29 f6 7d 57 0d 0a f9 |2..P.B.[Y].)W...|
00001420 22 d6 d8 68 57 85 2a e9 dd 15 18 c1 eb d3 03 d6 |".hW.*.....|
00001430 8f 79 27 60 fa 4b 8c 1c 3e 7c db e6 3e 72 fd 8d |.y'`.K.>|...>r..|
00001440 50 25 d9 ee 0f 30 5a 3a cf 7e d4 3a dc 98 bc c9 |P%...OZ:~.:....|
00001450 34 b3 8f 13 35 2e 55 1a f5 92 98 32 71 9c 8d 5b |4...5.U...2q..[|
00001460 8c f0 80 d2 1c 38 d5 a1 77 07 38 49 7c 7d 01 2f |.....8..w.8I|}./|
00001470 a1 c4 08 43 f5 af 67 7f d2 eb b9 e4 84 6c e1 77 |...C..g.....l.w|
00001480 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00001600 00 08 00 00 00 00 00 08 00 00 40 00 11 00 00 00 |.....@.....|

```

## 8.2.8 Product execution

This section presents the steps needed to be followed in order to properly run the software product according to its intended use and functionalities.

### 8.2.8.1 Getting started

The example below demonstrates the secure-boot flow with all the images loaded in NOR Flash.

Steps in the demo would be:

1. ISBC code would validate the ESBC code.
2. On successful validation, ESBC code would run, which would then validate the boot script.
3. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to validate next level images, i.e rootfs, linux ulmage and device tree.
5. Once all the images are validated, bootm command in boot script would be executed which would pass control to linux.

#### NOTE

For PowerPC SoC's, ISBC expects the code to be validated i.e. ESBC code to be within 0 - 3.5G address map. In the demo we map the flash to address 0xc0000000 for the ISBC code to validate ESBC in NOR Flash using PBI commands. Once the control reaches the ESBC code the earlier mapping of flash is removed and flash is mapped to address 0xe0000000.

#### 8.2.8.1.1 Environment for Secure Boot

There are 2 ways in which secure boot can be initiated:

- Set SB\_EN bit in RCW to 1.
- Programming the ITS fuse.

In a manufacturing environment, it is recommended that all fuses be programmed at once, including the ITS and OEM Section Write Protect bits. In a prototyping environment, it may be preferable to leave ITS and Write Protect unprogrammed (relying on RCW to initiate secure boot) until the developer has confidence in the secure boot process.

Two different RCW's are provided for the demo purpose:

1. The RCW which has SB\_EN bit set as 0 (sben0) and can be used when ITS = 1 i.e user wants to initiate secure boot flow using fuse.
2. The RCW which has the SB\_EN bit set as 1 (sben1) and can be used when user wants to initiate secure boot using RCW.

#### 8.2.8.1.2 SDK/ Images required for the demo

Given below are the images required for the demo which are built with Yocto as part of the SDK:

1. RCW with PBI commands
2. ESBC (U-Boot)
3. ulmage (Linux Image) \*
4. rootfs Image \*
5. Device tree \*

Please refer to User Manual QorIQ DPAA SDK for detailed description on how to run Yocto Build. Once the build process finishes, all the binaries would be present at the following location:

***build\_<platform>\_release/tmp/deploy/images***

The images will be created with the following names:

<b>u-boot-&lt;platform&gt;.bin</b>	U-Boot binary image for Secure Boot
<b>ulmage-&lt;platform&gt;.bin</b>	kernel image that can be loaded with U-Boot
<b>fsl-image-core-&lt;platform&gt;.ext2.gz.u-boot</b>	ramdisk filesystem image that can be loaded with U-Boot
<b>ulmage-&lt;platform&gt;.dtb</b>	device tree binary(dtb) for kernel bootup

RCW files will be present in the path **build\_<platform>\_release/tmp/ deploy/images/rcw**

---

**NOTE**

\* Some platforms like LS1043 have support for LINUX boot using a single kernel FIT image instead of 3 separate images (ulmage, rootfs and DTB)

---

## 8.2.8.2 Chain of Trust

This section presents the steps needed to be followed in order to execute Chain of Trust.

Steps in the demo would be:

1. ISBC code would validate the ESBC code.
2. On successful validation, ESBC code would run, which would then validate the boot script.
3. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to validate next level images, i.e rootfs, linux ulmage and device tree.
5. Once all the images are validated, bootm command in boot script would be executed which would pass control to linux.

### 8.2.8.2.1 Other images required for the demo

Apart from SDK images described above, the following images are also required:

1. CSF Header of the ESBC u-boot image
2. CSF Header of the ulmage \*
3. CSF Header of the rootfs image \*
4. CSF Header of the device tree \*
5. Boot Script
6. CSF Header of the boot script

The following section describes how to create the CSF headers and boot script.

---

**NOTE**

\* Some platforms like LS1043 have support for LINUX boot using a single kernel FIT image instead of 3 separate images (ulmage, rootfs and DTB). For these platforms a single CSF Header is required.

---

### 8.2.8.2.2 Boot Script and Signing the images

User can sign all the images with same public/private key pair or can use different key pairs to sign the images. Section below describes both the processes.

CST tool used for signing the images is provided as a package with yocto and is built for host. It can be run from your host machine.

Install path for CST binaries in yocto:

```
tmp/sysroots/x86_64-linux/usr/bin/cst/
```

CST uses openssl libraries, version 0.9.8.

In the Yocto environment, the user needs to use below commands to rebuild cst:

1. bitbake cst-native -c cleanall
2. bitbake cst-native
3. Modify the CST source code if needed.
4. bitbake cst-native -c cleanall
5. bitbake cst-native -c patch

Note: after step5, CST binary will be put to build\_<platform\_release/tmp/sysroots/x86\_64-linux/usr/bin/cst/ directory.

**Table 177. Platforms supported in SDK for Secure Boot**

Soc	<platform> supported in SDK	<platform> supported in CST
B4860	b4860qds	b4860
P2041	p2041rdb	p3_p4_p5
P3041	p3041ds	p3_p4_p5
P4080	p4080ds	p3_p4_p5
P5020	p5020ds	p3_p4_p5
P5040	p5040ds	p3_p4_p5
T1024	t1024rdb	t1_t2_t4
T104x	t1040rdb, t1042rdb	t1_t2_t4
T2080	t2080qds, t2080rdb	t1_t2_t4
T4240	t4240qds	t1_t2_t4
LS1021	ls1021aqds	ls1
LS1021	ls1021atwr	ls1
LS1043	ls1043ardb	ls1043
LS1043	ls1043aqds	ls1043
LS1046	ls1046aqds	ls1046
LS1046	ls0146ardb	ls0146

**NOTE**

Some platforms with ARMv8 core have support for LINUX boot using a single kernel FIT image instead of 3 separate images (ulmage, rootfs and DTB).

For such platforms, in CST as well instead of 3 different input files for signing ulmage, rootfs and dtb, there is a single input file named **input\_kernel\_secure** which can be used to sign the single kernel FIT image.

**This applies to all subsequent sections below.**

### 8.2.8.2.2.1 Signing the images using same key pair

CSF header needs to be generated for all the images. More details on the commands provided by CST can be found in Section .

## Boot Loaders

### Secure Boot: PBL Based Platforms

1. Generate the key pair to be used for signing the image

```
./gen_keys 1024
```

Key pair - public key file - srk.pub and private key in srk.priv would be generated.

2. Obtain hash string of the key pair generated to be programmed in SFP

```
./uni_sign --hash input_files/uni_sign/<platform>/input_uboot_secure
```

This would provide you the 256 bit hash in form of string of the key pair generated in the previous step. The hash has to be programmed in the SRK hash Fuse.

3. Create CSF header for u-boot Image.

```
./uni_sign input_files/uni_sign/<platform>/input_uboot_secure
```

The input fields are specified in input\_uboot\_secure file. Please ensure that the filename mentioned in the input\_uboot\_secure is same as copied in the cst directory.

4. Create CSF header for Linux ulmage

```
./uni_sign input_files/uni_sign/<platform>/input_uimage_secure
```

ulmage.bin would be validated form u-boot. The flash address used here is according to the address map of u-boot. Please ensure that filename mentioned in the input\_uimage\_secure is same as copied in the cst directory.

5. Create CSF header for rootfs

```
./uni_sign input_files/uni_sign/<platform>/input_rootfs_secure
```

Please make sure that filename mentioned in the input\_rootfs\_secure is same as copied in the cst directory

6. Create CSF Header for hardware device tree

```
./uni_sign input_files/uni_sign/<platform>/input_dtb_secure
```

Please make sure that filename mentioned in the input\_dtb\_secure is same as copied in the cst directory

7. Create Boot script

Bootscrip is a U-Boot script image. Steps to create bootscrip are given below :

- a. Create a text file bootscrip.txt with following commands.

```
esbc_validate <uImage CSF Header address>

esbc_validate <dtb CSF Header address>

esbc_validate <rootfs CSF Header address>

bootm <uImage Address> <rootfs address> <dtb address>
```

- b. Then you will have to use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

```
powerpc arch:: /tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d
bootscrip.txt bootscrip
```

```
arm arch:: /tmp/sysroots/x86_64-linux/usr/bin/mkimage -A arm -T script -a 0 -e 0x40 -d
bootscrip.txt bootscrip
```

8. Generate CSF hdr for the boot script

```
./uni_sign input_files/uni_sign/<platform>/input_bootscrip_secure
```

The fields can be changed in the input files for the images based on the requirement.

## 8.2.8.2.2 Signing the images using different key pair

If boot script is also signed with a different key, remember to define the macro "**CONFIG\_BOOTSCRIPT\_KEY\_HASH**" with the hash of the key used to sign the boot script in file *arch/powerpc/asm/include/fsl\_secure\_boot.h*. *ESBC u-boot would have to be recompiled if any change in this file is made.*

1. Generate the key pair to be used for signing the image

```
./gen_keys 1024 -p u-boot.priv -k u-boot.pub
```

Key pair - public key file - u-boot.pub and private key in u-boot.priv would be generated.

2. Obtain hash string of the key pair generated to be programmed in SFP

```
./uni_sign --hash input_files/uni_sign/<platform>/input_uboot_secure
```

This would provide you the 256 bit hash in form of string of the key pair generated in the previous step. The hash has to be programmed in the SRK hash Fuse.

3. Create CSF header for u-boot Image.

Open `input_files/uni_sign/<platform>/input_uboot_secure` and change `PRI_KEY` and `PUB_KEY` to `u-boot.priv` and `u-boot.pub` respectively and run the following command.

```
./uni_sign input_files/uni_sign/<platform>/input_uboot_secure
```

4. Create CSF header for Linux ulmage using different key pair.

Repeat step 1 to generate another key pair.

```
./gen_keys 1024 -p lnx.priv -k lnx.pub
```

Open `input_files/uni_sign/<platform>/input_uimage_secure` and change `PRI_KEY` and `PUB_KEY` to `lnx.priv` and `lnx.pub` respectively and run the following command. **./uni\_sign input\_files/uni\_sign/<platform>/input\_uimage\_secure**

Remember the "Key Hash" printed as it would be required in `esbc_validate` command in boot script. Say the hash of the key is `<lnx_key_hash>`

5. Create CSF header for rootfs

```
./gen_keys 1024 -p rootfs.priv -k rootfs.pub
```

Open `input_files/uni_sign/<platform>/input_rootfs_secure` and change `PRI_KEY` and `PUB_KEY` to `rootfs.priv` and `rootfs.pub` respectively and run the following command.

```
./uni_sign input_files/uni_sign/<platform>/input_rootfs_secure
```

Remember the "Key Hash" printed as it would be required in `esbc_validate` command in boot script. Say the hash of the key is `<rootfs_key_hash>`

6. Create CSF Header for hardware device tree

```
./gen_keys 1024 -p dtb.priv -k dtb.pub
```

Open `input_files/uni_sign/<platform>/input_dtb_secure` and change `PRI_KEY` and `PUB_KEY` to `dtb.priv` and `dtb.pub` respectively and run the following command. **./uni\_sign input\_files/uni\_sign/<platform>/input\_dtb\_secure**

Remember the "Key Hash" printed as it would be required in `esbc_validate` command in boot script. Say the hash of the key is `<dtb_key_hash>`

7. Write Boot script

Bootscrip is a U-Boot script image. Steps to create bootscrip are given below :

- a. Create a text file bootscript.txt with following commands.

```
esbc_validate <uImage CSF Header address> <lnx_key_hash>
esbc_validate <dtb CSF Header address> <dtb_key_hash>
esbc_validate <rootfs CSF Header address> <rootfs_key_hahs>
bootm <uImage Address> <rootfs address> <dtb address>
```

**NOTE**

Hashes would be the 256 bit string hash. These are the hashes of the key used to sign the respective images.

- b. Generate header over bootscript.txt which will be consumed by uboot command source
 

```
powerpc arch :: tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d bootscript.txt bootscript
arm arch :: tmp/sysroots/x86_64-linux/usr/bin/mkimage -A arm -T script -a 0 -e 0x40 -d bootscript.txt bootscript
```

8. Generate CSF hdr for the boot script

```
./gen_keys 1024 -p bs.priv -k bs.pub
```

Open `input_files/uni_sign/<platform>/input_dtb_secure` and change `PRI_KEY` and `PUB_KEY` to `bs.priv` and `bs.pub` respectively and run the following command.

```
./uni_sign input_files/uni_sign/<platform>/input_dtb_secure
```

### 8.2.8.2.3 Running secure boot (Chain of Trust)

1. Setup the board for secure boot flow. You can choose any if the flows mentioned below.

- a. **Flow A**

Program the ITS fuse. Use RCW with `SB_EN=0`

Or

- b. **Flow B**

For prototyping phase, don't blow the ITS fuse, but use rcw with `SB_EN = 1`.

**Note: For P3/P4/P5, if ITS fuse is blown, then ITF fuse must also be blown. (The value of ITS and ITF fuse must be identical.)**

2. Blow other required fuses on the board. (OTPMK and SRK hash<sup>[13]</sup>) For more details regarding fuse blowing, CCS and Boot Hold Off, refer to Platform reference manual and Trust Architecture User Guide.

**NOTE**

SRK hash in the fuse should be same as the hash of the key pair being used to sign the ESBC u-boot. Step 2 of [Signing the images using same key pair](#) on page 1167

For testing purpose, the SRK Hash can be written in the mirror registers.

`gen_otpmk_drbg` utility in `cst` can be used to generate otpmk key.

3. Flash all the generated images at locations as described in the address map ().

[13] Blowing of **OTPMK** is essential to run secure boot for both Production (Flow A) and Prototyping/Development (Flow B).

For **SRK Hash**, in Development Mode (Flow B), there is a workaround to avoid blowing fuses. For this use RCW with `BOOT_HO = 1`. This will put the core in Boot Hold off stage. Then a CCS can be connected via JTAG.

Write the SRK Hash value in SFP mirror registers and then release the core out of Boot Hold off by writing to Core Release Register in DCFG.

- a. **Flow A** - All the images would have to be flashed at the current bank addresses. Once ITS fuse is blown, the control would automatically shift to ISBC on power on.
  - b. If you are using **Flow B**, you can use alternate bank for demo purpose. This would mean flashing the images on alternate bank addresses from Bank0 and then switching to Bank4.
4. Give a power on cycle to the board.
- a. For **Flow A** and **Flow B** (*Secure boot Images flashed on default Bank*)
    - On power on, ISBC code would get control, validate the ESBC image.
    - ESBC image would further validate the signed linux, rootfs and dtb images
    - Linux would come up
  - b. **Flow B** (*Secure boot Images flashed on alternate Bank*)
    - On power on cycle, u-boot prompt on bank 0 would come up.
    - On switching to alternate bank, the secure boot flow as mentioned above would execute.

### 8.2.8.3 Chain of Trust with Confidentiality

This section presents the steps needed to be followed in order to execute Chain of Trust with confidentiality.

The demo would be divided into two parts:

1. Creating /encrypting images in form of blobs.
2. Decrypting the images, and booting from decrypted images.

Steps in the demo would be:

#### Step 1: Creating blobs

1. ISBC code would validate the ESBC code.
2. On successful validation, ESBC code would run, which would then validate the boot script.
3. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to encapsulate next level images, i.e rootfs, linux ulmage and device tree.

blob encapsulation command::

**blob enc src dst len km** - Encapsulate and create blob of data

\$len - Number of bytes to be encapsulated.

\$src - The address where image to be encapsulated is present.

\$dst - The address where encapsulated image will be stored.

\$km - It is the address where the key modifier is stored. The modifier is required and used as key for cryptographic operation. Key modifier should be 16 bytes long.

#### Step 2: Decrypting blob and booting

1. ISBC code would validate the ESBC code.
2. On successful validation, ESBC code would run, which would then validate the boot script.
3. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to decapsulate/decrypt next level images, i.e rootfs, linux ulmage and device tree.
5. After decryption, bootm command would be executed in boot script to pass control to Linux.

blob decapsulation command::

**blob dec src dst len km** - Decapsulate the image and recover the data

\$len - Number of bytes to be decapsulated.

\$src - The address where encapsulated image is present.

\$dst - The address where decapsulated image will be stored.

\$km - It is the address where the key modifier is stored. The modifier is required and used as key for cryptographic operation. Key modifier should be 16 bytes long. It should be same as passed while encapsulating the image.

### 8.2.8.3.1 Other images required for the demo

Apart from SDK images described above, the following images are also required:

1. Encap Boot script
2. Decap Boot script
3. CSF header for ESBC u-boot Image
4. CSF Header of the encap boot script
5. CSF Header of the decap boot script

The following section describes how to create the CSF headers and boot script.

### 8.2.8.3.2 Encap Bootscript

1. Create a bootscript\_en.txt file with following commands:

```
blob enc <uImage address> 0x10000000 <uImage size> <key_modifier address>

erase <encapsulated uImage address> +<encapsulated uImage size>
cp.b 0x10000000 <encapsulated uImage address> <encapsulated uImage size>

blob enc <rootfs address> 0x20000000 <rootfs size> <key_modifier address>

erase <encapsulated rootfs address> +<encapsulated rootfs size>
cp.b 0x20000000 <encapsulated rootfs address> <encapsulated rootfs size>

blob enc <dtb address> 0x1000000 <dtb size> <key_modifier address>

erase <encapsulated dtb address> +<encapsulated dtb size>
cp.b 0x1000000 <encapsulated dtb address> <encapsulated dtb size>
```

2. Use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

```
/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d bootscript_en.txt
bootscript_encap
```

### 8.2.8.3.3 Decap Bootscript

1. Create a bootscript\_de.txt file with following commands:

```
blob dec <encapsulated uImage address> 0x10000000 <uImage size + 0x30> <key_modifier address>

blob dec <encapsulated rootfs address> 0x20000000 <rootfs size + 0x30> <key_modifier address>

blob dec <encapsulated dtb address> 0x1000000 <dtb size + 0x30> <key_modifier address>

bootm 0x10000000 0x20000000 0x1000000
```

The script decapsulates/decrypts the blob created by earlier boot script and boots using them.

**NOTE**

0x30(48 bytes) should be added in the size of encapsulated images while decapsulating them. Always 48B are added at the end of the encapsulated image which needs to be added while providing the size of image to be decapsulated in blob dec command.

2. Use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

```
/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d bootscript_de.txt
bootscript_decap
```

### 8.2.8.3.4 Creating CSF Headers

- **CSF Header for ESBC**

Use the command given below to generate the hdr for u-boot binary.

```
./uni_sign input_files/uni_sign/<platform>/<input file for uboot>
```

Please change the binary name as per your uboot binary in “IMAGE\_1”

- **CSF Header for bootscript\_encap and bootscript\_decap**

Use the command given below to generate the headers for bootscripts

```
./uni_sign input_files/uni_sign/<platform>/<input file for bootscript>
```

Please change the binary name as per your bootscript in “IMAGE\_1”

### 8.2.8.3.5 Running secure boot (Chain of Trust with Confidentiality)

1. Setup the board for secure boot flow. You can choose any if the flows mentioned below.
  - a. **Flow A**  
Program the ITS fuse. Use RCW with SB\_EN=0  
Or
  - b. **Flow B**  
For prototyping phase, don't blow the ITS fuse, but use rcw with SB\_EN = 1.  
**Note: For P3/P4/P5, if ITS fuse is blown, then ITF fuse must also be blown. (The value of ITS and ITF fuse must be identical.)**
2. Blow other required fuses on the board. (OTPMK and SRK hash<sup>[14]</sup>) For more details regarding fuse blowing, CCS and Boot Hold Off, refer to Platform reference manual and Trust Architecture User Guide.

**NOTE**

SRK hash in the fuse should be same as the hash of the key pair being used to sign the ESBC u-boot.

For testing purpose, the SRK Hash can be written in the mirror registers.

gen\_otpmk\_drbg utility in cst can be used to generate otpmk key.

[14] Blowing of **OTPMK** is essential to run secure boot for both Production (Flow A) and Prototyping/Development (Flow B).

For **SRK Hash**, in Development Mode (Flow B), there is a workaround to avoid blowing fuses. For this use RCW with BOOT\_HO = 1. This will put the core in Boot Hold off stage. Then a CCS can be connected via JTAG.

Write the SRK Hash value in SFP mirror registers and then release the core out of Boot Hold off by writing to Core Release Register in DCFG.

3. Flash all the generated images at locations as described in the address map ()
  - Uboot binary
  - CSF Header of uboot
  - Linux ulmage
  - Rootfs
  - Device tree
  - bootscript\_encap
  - CSF Header for bootscript\_encap
  - a. **Flow A** - All the images would have to be flashed at the current bank addresses. Once ITS fuse is blown, the control would automatically shift to ISBC on power on.
  - b. If you are using **Flow B**, you can use alternate bank for demo purpose. This would mean flashing the images on alternate bank addresses from Bank0 and then switching to Bank4.
4. Give a power on cycle to the board.
  - a. For **Flow A** and **Flow B** (*Secure boot Images flashed on default Bank*)
    - On power on, ISBC code would get control, validate the ESBC image.
    - ESBC image would further validate the bootscript.
    - Bootscript would encapsulate the Linux, rootfs and device tree and store the blobs at the desired locations.
  - b. **Flow B** (*Secure boot Images flashed on alternate Bank*)
    - On power on cycle, u-boot prompt on bank 0 would come up.
    - On switching to alternate bank, the secure boot flow as mentioned above would execute.
5. Give a power on cycle to the board.
6. Replace the encap bootscript and its CSF header with decap bootscript and the CSF header of the decap bootscript respectively.
7. Give a power on cycle to the board.
  - a. For **Flow A** and **Flow B** (*Secure boot Images flashed on default Bank*)
    - On power on, ISBC code would get control, validate the ESBC image.
    - ESBC image would further validate the bootscript.
    - Bootscript would decapsulate the Linux, rootfs and device tree blob and store them on DDR
    - Bootm command in bootscript would execute on successful decapsulation
    - Linux prompt would come up .
  - b. **Flow B** (*Secure boot Images flashed on alternate Bank*)
    - On power on cycle, u-boot prompt on bank 0 would come up.
    - On switching to alternate bank, the secure boot flow as mentioned above would execute.

### 8.2.8.4 NAND Secure Boot (Chain of Trust)

This section presents the steps and images needed for running Secure Boot Chain of Trust from NAND on **P3/P5**.

The procedure for running Secure boot from NAND is same as Secure Boot from NOR. The only difference is that in case of NOR, image is not required to be copied from NOR while in case of NAND, images have to be copied from NAND to SRAM/DDR before validation.

#### Images Required for Demo

1. PBL.bin
2. ulmage (Linux Image)
3. rootfs
4. dtb (Device Tree)
5. CSF Header of the ulmage
6. CSF Header of the rootfs image
7. CSF Header of the device tree
8. Boot Script
9. CSF Header of the Boot Script

### **Boot Script**

The sample bootscript.txt would have the following commands:

```
Read uImage & Header
nand read <uImage DDR> <uImage NAND> <uImage size>
nand read <uImage Header DDR> <uImage Header NAND> <uImage Header size>

Read rootfs & Header
nand read <rootfs DDR> <rootfs NAND> <rootfs size>
nand read <rootfs Header DDR> <rootfs Header NAND> <rootfs Header size>

Read dtb & Header
nand read <dtb DDR> <dtb NAND> <dtb size>
nand read <dtb Header DDR> <dtb Header NAND> <dtb Header size>

Validate and Boot
esbc_validate <uImage Header DDR>
esbc_validate <DTB Header DDR>
esbc_validate <rootfs Header DDR>
bootm <uImage DDR> <rootfs DDR> <dtb DDR>
```

### **Image Signing**

The image signing process will remain same as in case of NOR [Boot Script and Signing the images](#) on page 1166

<platform> will be p3\_p4\_p5/nand

#### **NOTE**

ISBC Key Extension Feature is not applicable for Secure Boot from NAND.

## **8.2.8.4.1 Running Secure Boot Chain of Trust (from NAND)**

1. Setup the board for secure boot flow. You can choose any if the flows mentioned below.
  - a. **Flow A**  
Program the ITS fuse. Use RCW with SB\_EN=0  
Or
  - b. **Flow B**  
For prototyping phase, don't blow the ITS fuse, but use rcw with SB\_EN = 1.

**Note: For P3/P5, if ITS fuse is blown, then ITF fuse must also be blown. (The value of ITS and ITF fuse must be identical.)**

- Blow other required fuses on the board. (OTPMK and SRK hash<sup>[15]</sup>) For more details regarding fuse blowing, CCS and Boot Hold Off, refer to Platform reference manual and Trust Architecture User Guide.

#### NOTE

SRK hash in the fuse should be same as the hash of the key pair being used to sign the ESBC u-boot. Step 2 of [Signing the images using same key pair](#) on page 1167

For testing purpose, the SRK Hash can be written in the mirror registers.

gen\_otpmk\_drbg utility in cst can be used to generate otpmk key.

- Flash all the generated images on NAND Flash at locations as described in the address map ().
- Switch to NAND Boot.

#### a. FLOW A

Change the Switch Settings to change the RCW\_SRC to NAND and power on the board.

#### b. FLOW B

Power on the board to bring up Non-Secure U-Boot on NOR and from U-Boot prompt issue the following command.

```
mw.b 0xffdf0020 0x48;mw.b 0xffdf0021 0x78;mw.b 0xffdf002c 0x90;mw.b 0xffdf002d 0xf0;mw.b
0xffdf0010 0; mw.b 0xffdf0010 1
```

- The PBL would configure CPC as SRAM, update the SCRATCH register and copy the Header and U-boot (ESBC) on CPC configured as SRAM.
- ESBC code would get control, validate the ESBC image.
- ESBC image would further copy the Boot Script Header and Boot Script from NAND to DDR, validate the boot script and execute it.
- The Boot Script has commands to copy the linux images and their respective headers from NAND to DDR, validate the signed linux, rootfs and dtb images.
- Linux would be booted.

## 8.2.8.5 NAND Secure Boot (Chain of Trust with Confidentiality)

This section presents the steps and images needed for running Secure Boot Chain of Trust with Confidentiality from NAND on **P3/P5**

The procedure for running Secure boot from NAND is same as Secure Boot from NAND. The only difference is that in case of NOR, image is not required to be copied from NOR while in case of NAND, images have to be copied from NAND to SRAM/DDR before validation.

### Images Required for Demo

- PBL.bin

The PBL.bin is generated using QCVS Tool. It creates the RCW along with PBI commands. ESBC (U-boot) and CSF Header for U-Boot are added using ACS\_WRITE PBI commands. (For details/screenshots refer [Using QCVS Tool \(Secure Boot From NAND\)](#) on page 1205)

[15] Blowing of **OTPMK** is essential to run secure boot for both Production (Flow A) and Prototyping/Development (Flow B).

For **SRK Hash**, in Development Mode (Flow B), there is a workaround to avoid blowing fuses. For this use RCW with BOOT\_HO = 1. This will put the core in Boot Hold off stage. Then a CCS can be connected via JTAG.

Write the SRK Hash value in SFP mirror registers and then release the core out of Boot Hold off by writing to Core Release Register in DCFG.

2. uImage (Linux Image)
3. rootfs
4. dtb (Device Tree)
5. Encap Boot Script
6. CSF Header of the Encap Boot Script
7. Decap Boot Script
8. CSH Header for Decap Boot Script

### Encap Boot Script

```
uImage
nand read <uImage DDR> <uImage NAND> <uImage size>
esbc_blob_encap <uImage DDR> 0x10000000 <uImage size> 0x11223344556677889900aabbccddeeff
nand erase <encapsulated uImage NAND> <encapsulated uImage size>
nand write 0x10000000 <encapsulated uImage NAND> <encapsulated uImage size>

rootfs
nand read <rootfs DDR> <rootfs NAND> <rootfs size>
esbc_blob_encap <rootfs DDR> 0x10000000 <rootfs size> 0x11223344556677889900aabbccddeeff
nand erase <encapsulated rootfs NAND> <encapsulated rootfs size>
nand write 0x10000000 <encapsulated rootfs NAND> <encapsulated rootfs size>

dtb
nand read <dtb DDR> <dtb NAND> <dtb size>
esbc_blob_encap <dtb DDR> 0x10000000 <dtb size> 0x11223344556677889900aabbccddeeff
nand erase <encapsulated dtb NAND> <encapsulated dtb size>
nand write 0x10000000 <encapsulated dtb NAND> <encapsulated dtb size>
```

### Decap Boot Script

```
nand read <encapsulated uImage DDR> <encapsulated uImage NAND> <encapsulated uImage size>
esbc_blob_decap <encapsulated uImage DDR> 0x10000000 <uImage size>
0x11223344556677889900aabbccddeeff

nand read <encapsulated rootfs DDR> <encapsulated rootfs NAND> <encapsulated rootfs size>
esbc_blob_decap <encapsulated rootfs DDR> 0x20000000 <rootfs size>
0x11223344556677889900aabbccddeeff

nand read <encapsulated dtb DDR> <encapsulated dtb NAND> <encapsulated dtb size>
esbc_blob_decap <encapsulated dtb DDR> 0x10000000 <dtb size> 0x11223344556677889900aabbccddeeff

bootm 0x10000000 0x20000000 0x10000000
```

### Image Signing

The image signing process will remain same as in case of NOR [Boot Script and Signing the images](#) on page 1166  
<platform> will be p3\_p4\_p5/nand

## **8.2.8.5.1 Running Secure Boot Chain of Trust with Confidentiality (from NAND)**

1. Setup the board for secure boot flow. You can choose any if the flows mentioned below.

- a. **Flow A**

Program the ITS fuse. Use RCW with SB\_EN=0

Or

b. **Flow B**

For prototyping phase, don't blow the ITS fuse, but use rcw with SB\_EN = 1.

**Note: For P3/P5, if ITS fuse is blown, then ITF fuse must also be blown. (The value of ITS and ITF fuse must be identical.)**

2. Blow other required fuses on the board. (OTPMK and SRK hash<sup>[16]</sup>) For more details regarding fuse blowing, CCS and Boot Hold Off, refer to Platform reference manual and Trust Architecture User Guide.

---

**NOTE**

SRK hash in the fuse should be same as the hash of the key pair being used to sign the ESBC u-boot. Step 2 of [Signing the images using same key pair](#) on page 1167

For testing purpose, the SRK Hash can be written in the mirror registers.

gen\_otpmk\_drbg utility in cst can be used to generate otpmk key.

---

3. Flash all the generated images on NAND Flash.

- a. PBL.bin
- b. LINUX Images (ulmage, dtb, rootfs)
- c. CSF Header for bootscript\_encap
- d. bootscript\_encap

4. Switch to NAND Boot.

a. **FLOW A**

Change the Switch Settings to change the RCW\_SRC to NAND and power on the board.

b. **FLOW B**

Power on the board to bring up Non-Secure U-Boot on NOR and from U-Boot prompt issue the following command.

```
mw.b 0xffdf0020 0x48;mw.b 0xffdf0021 0x78;mw.b 0xffdf002c 0x90;mw.b 0xffdf002d 0xf0;mw.b
0xffdf0010 0; mw.b 0xffdf0010 1
```

- The PBL would configure CPC as SRAM, update the SCRATCH register and copy the Header and U-boot (ESBC) on CPC configured as SRAM.
  - ISBC code would get control, validate the ESBC image.
  - ESBC image would further copy the Boot Script Header and Boot Script from NAND to DDR, validate the boot script and execute it.
  - Bootscript would encapsulate the Linux, rootfs and device tree and store the blobs at the desired locations.
5. Revert the switch settings to earlier RCW\_SRC and power on the board. Replace the encap bootscript and its CSF header with decap bootscript and the CSF header of the decap bootscript respectively.
  6. Switch to NAND Boot.

a. **FLOW A**

Change the Switch Settings to change the RCW\_SRC to NAND and power on the board.

---

[16] Blowing of **OTPMK** is essential to run secure boot for both Production (Flow A) and Prototyping/Development (Flow B).

For **SRK Hash**, in Development Mode (Flow B), there is a workaround to avoid blowing fuses. For this use RCW with BOOT\_HO = 1. This will put the core in Boot Hold off stage. Then a CCS can be connected via JTAG.

Write the SRK Hash value in SFP mirror registers and then release the core out of Boot Hold off by writing to Core Release Register in DCFG.

**b. FLOW B**

Power on the board to bring up Non-Secure U-Boot on NOR and from U-Boot prompt issue the following command.

```
mw.b 0xffdf0020 0x48;mw.b 0xffdf0021 0x78;mw.b 0xffdf002c 0x90;mw.b 0xffdf002d 0xf0;mw.b 0xffdf0010 0; mw.b 0xffdf0010 1
```

- The PBL would configure CPC as SRAM, update the SCRATCH register and copy the Header and U-boot (ESBC) on CPC configured as SRAM.
- ISBC code would get control, validate the ESBC image.
- ESBC image would further copy the Boot Script Header and Boot Script from NAND to DDR, validate the boot script and execute it.
- Bootscript would copy the Linux, rootfs and device tree blobs on DDR and then decapsulate them on DDR.
- Bootm commnd in bootscript would execute on successful decapsulation.
- Linux prompt would come up.

## 8.2.9 Troubleshooting

**Table 178. Troubleshooting**

	Symptoms	Reasons and/or Recommended actions
1.	No print on UART console.	<ul style="list-style-type: none"> <li>• Check the status register of sec mon block (location 0xfe314014). Refer to the details of the register from the Reference Manual. Bits OTPMK_ZERO, OTPMK_SYNDROME and PE should be 0 otherwise there is some error in the OTPMK fuse blown by you.</li> <li>• If OTPMK fuse is correct (see Step 1), check the SCRATCHRW2 register for errors. Refer to Section for error codes.</li> <li>• If <b>Error code = 0</b> then check the Security Monitor state in HPSR register of Sec Mon.</li> </ul> <p><b>Sec Mon in Check State (0x9)</b></p> <p>If ITS fuse = 1, then it means ISBC code has reset the board. This may be due to the following reasons:</p> <p>Hash of the public key used to sign the ESBC u-boot doesn't match with the value in SRK Hash Fuse</p> <p>Or</p> <p>Signature verification of the image failed</p> <p><b>Sec Mon in Trusted State (0xd) or Non Secure State (0xb)</b></p> <p>Check the entry point field in the ESBC header. It should be 0xcfffffc for the demo described in Section 4.</p> <p>If entry point is correct, ensure that u-boot image has been compiled with the required secure boot configuration.</p>

*Table continues on the next page...*

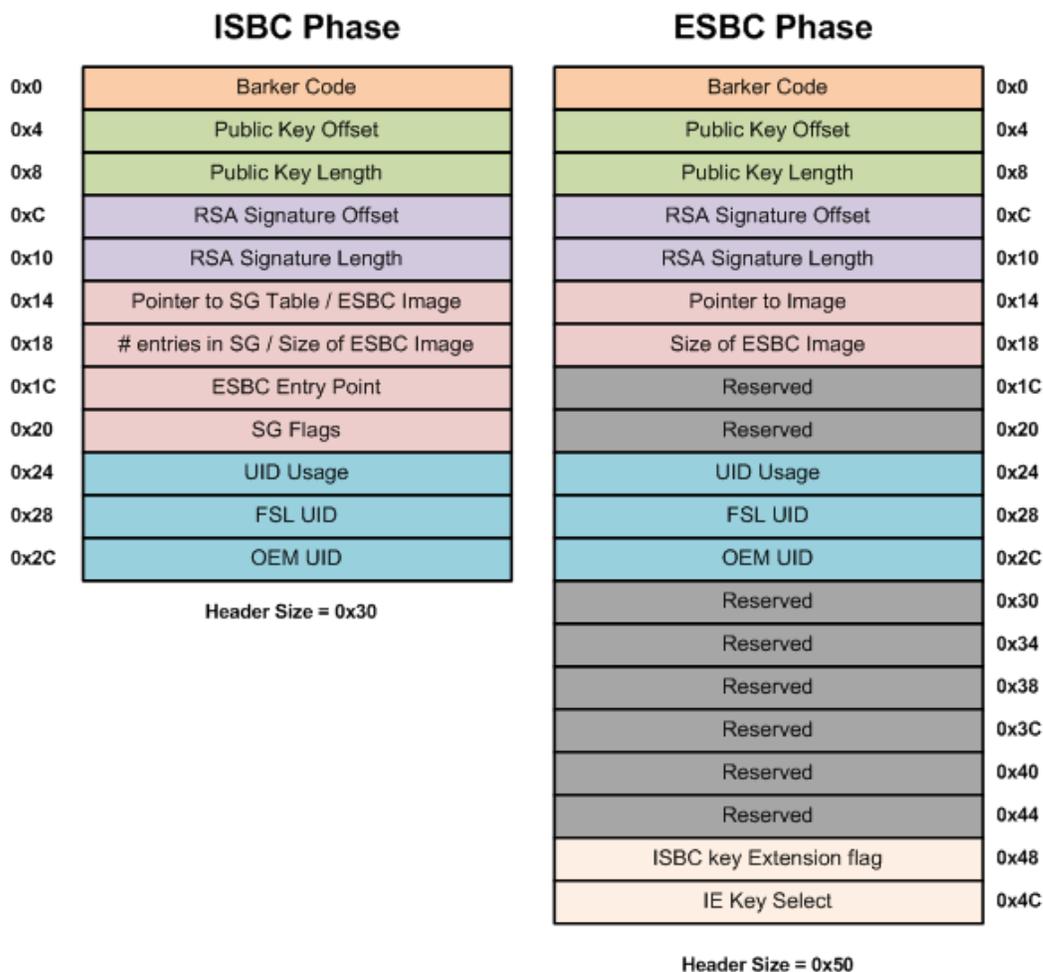
**Table 178. Troubleshooting (continued)**

	Symptoms	Reasons and/or Recommended actions
2.	Instead of linux prompt, you get a u-boot command prompt instead of linux prompt.	You have not booted in secure boot mode. You never get a u-boot prompt in secure boot flow. Check Step 1 in <a href="#">Running secure boot (Chain of Trust)</a> on page 1170. You would reach this stage if ITS = 0 and you are using rcw where sben0 is present in its name.
3	u-boot hangs or board resets	Some validation failure occurred in ESBC u-boot. Error code and description would be printed on u-boot console. Refer to for more details on errors.

## 8.2.10 CSF Header Data Structure

The CSF Header provides the ISBC with most of the information needed to validate the image.

### P3/P4/P5 Platforms



**Figure 191. CSF Header for P3/P4/P5 (ISBC and ESBC Phase)**

**Table 179. CSF Header Format (P3/P4/P5 Platforms)**

Offset	Data Bits [0:31]
0x00-0x03	<b>Barker code.</b> This location should contain the value: 0x68392781. The ISBC code searches for this Barker code. If the value in this location does not match the Barker code, the ISBC stops execution and reports error.
0x04-0x07	<b>Public key offset.</b> This location contains an offset in bytes of the public key from the start of CSF header. Using this offset and the public key length, the public key is read.
0x08-0x0b	<b>Public key length</b> in bytes. (Value populated here should be twice of Modulus size). Supported sizes are 256, 512 or 1024 bytes (2 * 1024, 2 * 2048 , 2 * 4096 bits).
0x0c-0x0f	<b>RSA Signature offset.</b> This location contains an offset(in bytes) of the RSA signature from the start of CSF header. Using this offset and the Signature length, the RSA signature is read. The RSA signature is calculated over CSF Header, Scatter Gather table and ESBC images.
0x10-0x13	<b>RSA Signature length</b> in bytes.
0x14-0x17	For ISBC Phase: Based on the Scatter Gather flag in CSF header, this location can either be treated as <b>Pointer to Scatter Gather table or the address of ESBC image.</b> For ESBC Phase: This location is treated as <b>address of image</b> (linux/bootscript/rootfs/dtb) to be validated.
0x18-0x1b	For ISBC Phase: Based on the Scatter gather flag in CSF Header, this location can either be treated as <b>number of entries in SG table or ESBC image size</b> in bytes. For ESBC Phase: Size of image to be validated
0x1c-0x1f	For ISBC Phase: <b>ESBC entry point.</b> ISBC transfers control to this location upon successful validation of ESBC image(s). For ESBC Phase: Reserved.
0x20-0x23	For ISBC Phase: <b>Scatter Gather flag.</b> 0x0000 - 0x14-0x17 is a pointer to the ESBC image 0x0001 - 0x14-0x17 is a pointer to a scatter/gather table. For ESBC Phase: Reserved
<i>Table continues on the next page...</i>	

**Table 179. CSF Header Format (P3/P4/P5 Platforms) (continued)**

Offset	Data Bits [0:31]
0x24-0x27	<p><b>Unique ID Usage.</b></p> <p>UIDs present in the CSF Header are compared to the corresponding UIDs in the SFP, and are included in the ESBC validation.</p> <p>0x0000 - No UIDs are present in CSF header</p> <p>0x0001 - FSL_UID and OEM_UID are present in CSF header</p> <p>0x0002 - Only OEM_UID present in CSF header</p> <p>0x0004 - Only FSL_UID present in CSF header</p>
0x28-0x2b	<p><b>Freescale unique ID.</b></p> <p>A unique 32 bit value, which is specific to Freescale. This value is compared with the FSL ID in Secure Fuse Processor 's FSL-ID registers</p>
0x2c-0x2f	<p><b>OEM unique ID.</b></p> <p>A unique 32 bit value, which is specific to OEM. This value is compared with the OEM ID in Secure Fuse Processor 's OEM-ID registers</p>
0x30-0x47	<p>For ISBC Phase: Not Applicable</p> <p>For ESBC Phase: Reserved</p>
0x48-0x4b	<p>For ISBC Phase: Not Applicable</p> <p>For ESBC Phase:</p> <p><b>ISBC key Extension flag.</b></p> <p>If this flag is set, key to be used for validation needs to be picked up from IE Key table.</p>
0x4c-0x4f	<p>For ISBC Phase: Not Applicable</p> <p>For ESBC Phase:</p> <p><b>IE Key Select.</b></p> <p>Key Number to be used from the IE Key Table if IE flag is set.</p>

**Table 180. Scatter Gather Table Format (P3/P4/P5 Platforms)**

Offset	Data Bits [0:31]
0x00-0x03	Length in bytes of the first segment of the ESBC image.
0x04-0x07	Pointer to first segment of ESBC image.
0x08-0x0b	Length in bytes of the second segment of the ESBC image.
0x0c-0x0f	Pointer to second segment of ESBC image.
<i>Table continues on the next page...</i>	

**Table 180. Scatter Gather Table Format (P3/P4/P5 Platforms) (continued)**

Offset	Data Bits [0:31]
0xww-0xxx	Length in bytes of the nth segment of the ESBC image.
0xyy-0xzz	Pointer to nth segment of ESBC image

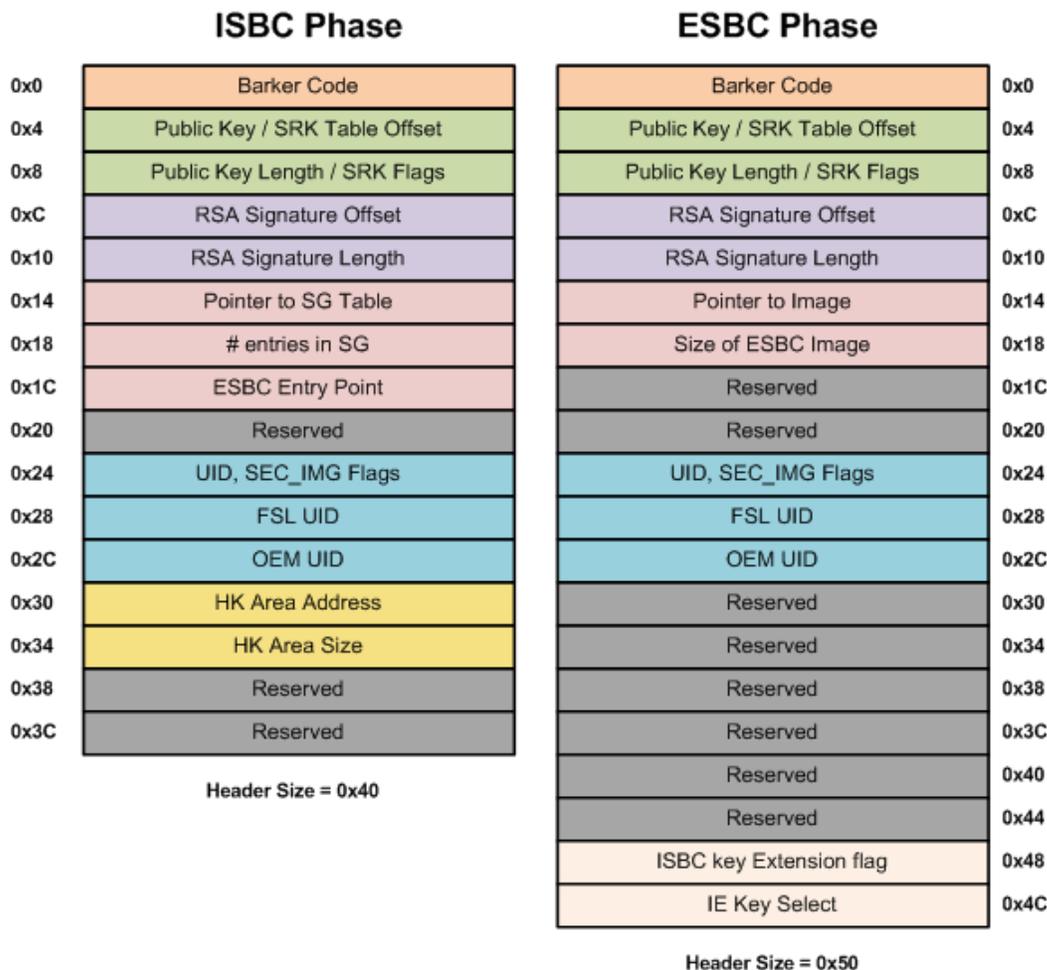
**Table 181. Signature (P3/P4/P5 Platforms)**

Offset	Data Bits [0:31]
0x00-size	The RSA signature calculated over CSF Header, Scatter Gather table and ESBC image(s).

**Table 182. Public key (P3/P4/P5 Platforms)**

Offset	Data Bits [0:31]
0x00-size	Public Key Value. The hash of this public key is compared with the hash stored in Secure Fuse Processor SRKH registers.

**B4/T1/T2/T4 Platforms**



**Figure 192. CSF Header for B4/T1/T2/T4 (ISBC and ESBC Phase)**

**Table 183. CSF Header Format (B4/T1/T2/T4 Platforms)**

Offset	Data Bits [0:31]
0x00-0x03	<b>Barker code.</b> This location should contain the value: 0x68392781. The ISBC code searches for this Barker code. If the value in this location does not match the Barker code, the ISBC stops execution and reports error.
0x04-0x07	If the srk_table_flag is not set : <ul style="list-style-type: none"> <li>• <b>Public key offset:</b> This location contains an address which is the offset of the public key from the start of CSF header. Using this offset and the public key length, the public key is read.</li> </ul> If srk_table_flag is set: <ul style="list-style-type: none"> <li>• <b>Srk table offset:</b> This location contains an address which is the offset of the srk table from the start of CSF header. Using this offset and the number of entries is SRK Table, the SRK table is read.</li> </ul>
<i>Table continues on the next page...</i>	

**Table 183. CSF Header Format (B4/T1/T2/T4 Platforms) (continued)**

Offset	Data Bits [0:31]
0x08	<b>Srk table flag.</b> This flag indicates whether hash burnt in srk fuse is of a single key or of srk table.
0x09-0x0b	If the srk_table_flag is not set : <ul style="list-style-type: none"> <li>• <b>Public key length:</b> This location contains the length of the public key in bytes.</li> </ul> If srk_table_flag is set: <ul style="list-style-type: none"> <li>• 0x09 – <b>Key Number from srk table</b> which is to be used for verification.</li> <li>• 0x0a-0x0b – <b>Number of entries in srk table.</b> Minimum number of entries in table = 1, Maximum = 4.</li> </ul>
0x0c-0x0f	<b>RSA Signature offset.</b> This location contains an offset(in bytes) of the RSA signature from the start of CSF header. Using this offset and the Signature length, the RSA signature is read. The RSA signature is calculated over CSF Header, Scatter Gather table and ESBC images.
0x10-0x13	<b>RSA Signature length</b> in bytes.
0x14-0x17	For ISBC Phase: <b>SG Table offset</b> This location contains an address which is the offset of the SG table from the start of CSF header. Using this offset and the number of entries in SG Table, the SG table is read.  For ESBC Phase: Address of the image to be validated.
0x18-0x1b	For ISBC Phase: <b>Number of entries in SG Table</b> (Earlier ,Based on the Scatter gather flag in CSF Header, this location can either be treated as number of entries in SG table or ESBC image size in bytes.).  For ESBC Phase: <b>Size of Image</b> to be validated
0x1c-0x1f	For ISBC Phase: <b>ESBC entry point.</b> ISBC transfers control to this location upon successful validation of ESBC image(s).  For ESBC Phase: Reserved
0x20-0x23	<b>Reserved</b> .(Earlier this field was SG Flag. SG flag is always assumed to be 1 in unified implementation.)
0x24	For ISBC Phase: Reserved  For ESBC Phase: Reserved

*Table continues on the next page...*

**Table 183. CSF Header Format (B4/T1/T2/T4 Platforms) (continued)**

Offset	Data Bits [0:31]
0x25	<p>For ISBC Phase:</p> <p><b>Secondary Image flag</b></p> <p>Indicates if user has a secondary image available in case of failures in validating primary image. Valid in case of primary Images's Header.</p> <p>For ESBC Phase: Reserved</p>
0x26-0x27	<p><b>Unique ID Usage</b></p> <p>This location contains a flag which specifies one of these possibilities</p> <ul style="list-style-type: none"> <li>• 0x00 - No UID's present</li> <li>• 0x01 - FSL UID and OEM UID are present</li> <li>• 0x02 - Only FSL UID is present</li> <li>• 0x04 - Only OEM UID is present</li> </ul>
0x28-0x2b	<p><b>Freescale unique ID.</b></p> <p>A unique 32 bit value, which is specific to Freescale. This value is compared with the FSL ID in Secure Fuse Processor 's FSL-ID registers</p>
0x2c-0x2f	<p><b>OEM unique ID.</b></p> <p>A unique 32 bit value, which is specific to OEM. This value is compared with the OEM ID in Secure Fuse Processor 's OEM-ID registers</p>
0x30-0x33	<p>For ISBC Phase:</p> <p><b>Housekeeping area address</b></p> <p>This is address of start of a memory which can be accessed by devices on SOC bus. (DDR, L3 cache configured as SRAM ). The area should have been pre-configured by user through PBL commands or configuration header</p> <p>For ESBC Phase: Reserved</p>
0x34-0x37	<p>For ISBC Phase:</p> <p><b>Size of the housekeeping area</b></p> <p>Size of the pre-configured memory which can be used by Boot Rom Code.</p> <p>For ESBC Phase: Reserved</p>
0x38-0x3f	Reserved
0x40-0x47	<p>For ISBC Phase: Not Applicable</p> <p>For ESBC Phase: Reserved</p>

*Table continues on the next page...*

**Table 183. CSF Header Format (B4/T1/T2/T4 Platforms) (continued)**

Offset	Data Bits [0:31]
0x48-0x4b	For ISBC Phase: Not Applicable For ESBC Phase: <b>ISBC key Extension flag</b> If this flag is set, key to be used for validation needs to be picked up from IE Key table.
0x4c-0x4f	For ISBC Phase: Not Applicable For ESBC Phase: <b>IE Key Select</b> Key Number to be used from the IE Key Table if IE flag is set.

**Table 184. Scatter Gather Table Format (B4/T1/T2/T4 Platforms)**

Offset	Data Bits [0:31]
0x00-0x03	Length. This location specifies the length in bytes of the ESBC image 1.
0x04-0x07	Target where the ESBC Image 1 can be found. This field is ignored in case of PBL based SOC's.
0x08-0x0b	Source Address of ESBC Image 1
0x0c-0x0f	Destination Address of ESBC Image 1 If the target address is 0xffffffff, the image is not copied to the target. This field is ignored in case of PBL based SOC's.
0x10-0x13	Length. This location specifies the length in bytes of the ESBC image 2.
0x14-0x17	Target where the ESBC Image 2 can be found. This field is ignored in case of PBL based SOC's.
0x18-0x1b	Source Address of ESBC Image 2
0x1c-0x1f	Destination Address of ESBC Image 2 If the target address is 0xffffffff, the image is not copied to the target. This field is ignored in case of PBL based SOC's.

**Table 185. Signature (B4/T1/T2/T4 Platforms)**

Offset	Data Bits [0:31]
0x00-size	The RSA signature calculated over CSF Header, Scatter Gather table and ESBC image(s).

**Table 186. Public key (B4/T1/T2/T4 Platforms)**

Offset	Data Bits [0:31]
0x00-size	Public Key Value. The hash of this public key is compared with the hash stored in Secure Fuse Processor SRKH registers.

**Table 187. SRK Table (B4/T1/T2/T4 Platforms)**

Offset	Data Bits [0:31]
0x00-0x03	Key 1 length
0x04-0x403	Key 1 value. (Remaining bytes will be padded with zero)
0x404-0x407	Key 2 length
0x408-0x807	Key 2 value. (Remaining bytes will be padded with zero)
0x808-0x80b	Key 3 length
0x80c-0xb0b	Key 3 value. (Remaining bytes will be padded with zero)
0xb0c-0xb0f	Key 4 length
0xb10-0xe10	Key 4 value. (Remaining bytes will be padded with zero)

LS1 Platform

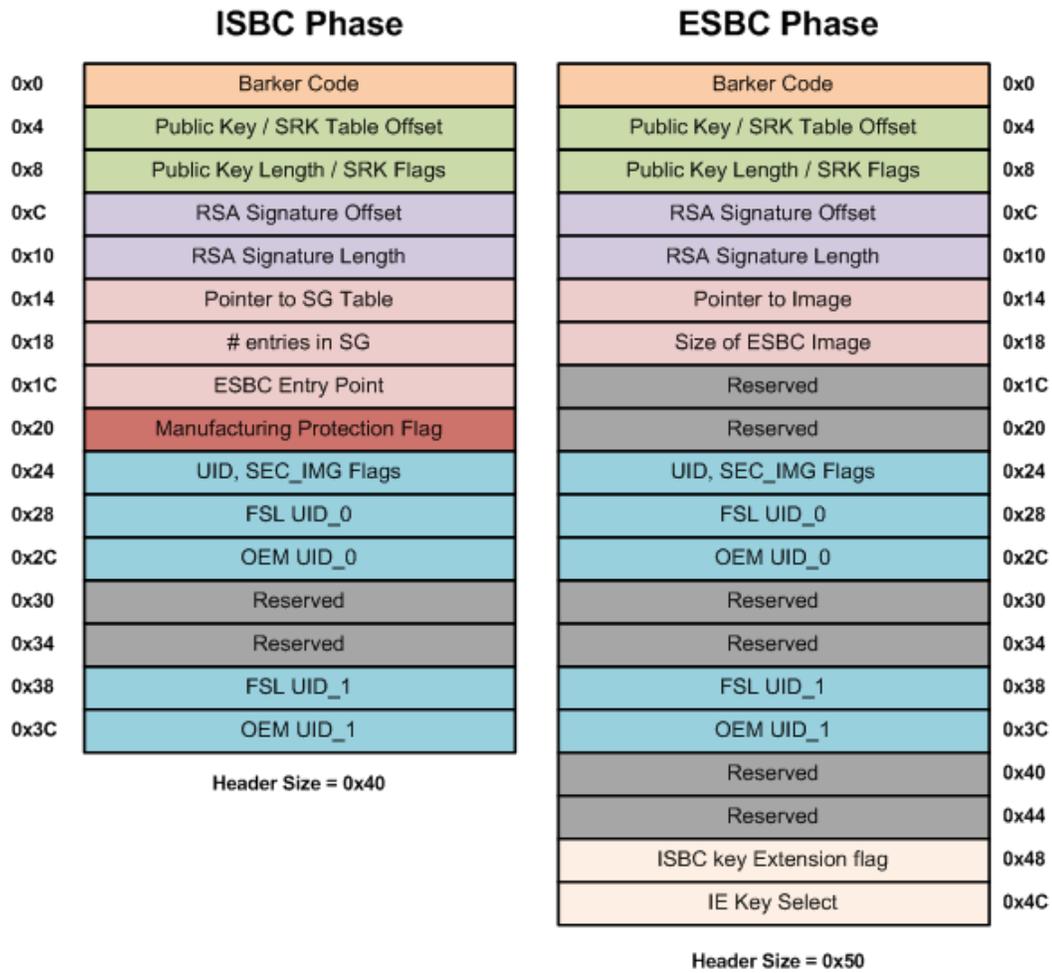


Figure 193. CSF Header for LS1 (ISBC and ESBC Phase)

Table 188. CSF Header Format (LS1 Platform)

Offset	Data Bits [0:31]
0x00-0x03	<b>Barker code.</b> This location should contain the value: 0x68392781. The ISBC code searches for this Barker code. If the value in this location does not match the Barker code, the ISBC stops execution and reports error.
0x07-0x04	If the srk_table_flag is not set : <ul style="list-style-type: none"> <li>• <b>Public key offset:</b> This location contains an address which is the offset of the public key from the start of CSF header. Using this offset and the public key length, the public key is read.</li> </ul> If srk_table_flag is set: <ul style="list-style-type: none"> <li>• <b>Srk table offset:</b> This location contains an address which is the offset of the srk table from the start of CSF header. Using this offset and the number of entries is SRK Table, the SRK table is read.</li> </ul>

*Table continues on the next page...*

**Table 188. CSF Header Format (LS1 Platform) (continued)**

Offset	Data Bits [0:31]
0x08	<p><b>Srk table flag.</b></p> <p>This flag indicates whether hash burnt in srk fuse is of a single key or of srk table.</p>
0x0b-0x09	<p>If the srk_table_flag is not set :</p> <ul style="list-style-type: none"> <li>• <b>0x0b-0x9 -- Public key length:</b> This location contains the length of the public key in bytes.</li> </ul> <p>If srk_table_flag is set:</p> <ul style="list-style-type: none"> <li>• <b>0x09 – Key Number from srk table</b> which is to be used for verification.</li> <li>• <b>0x0b-0x0a – Number of entries in srk table.</b> Minimum number of entries in table = 1, Maximum = 4.</li> </ul>
0x0f-0x0c	<p><b>RSA Signature offset.</b></p> <p>This location contains an offset(in bytes) of the RSA signature from the start of CSF header. Using this offset and the Signature length, the RSA signature is read. The RSA signature is calculated over CSF Header, Scatter Gather table and ESBC images.</p>
0x13-0x10	<p><b>RSA Signature length</b> in bytes.</p>
0x17-0x14	<p><b>For ISBC Phase:</b></p> <p><b>SG Table offset</b></p> <p>This location contains an address which is the offset of the SG table from the start of CSF header. Using this offset and the number of entries in SG Table, the SG table is read.</p> <p><b>For ESBC Phase:</b></p> <p>Address of the image to be validated.</p>
0x1b-0x18	<p><b>For ISBC Phase:</b></p> <p><b>Number of entries in SG Table</b> (Earlier ,Based on the Scatter gather flag in CSF Header, this location can either be treated as number of entries in SG table or ESBC image size in bytes.).</p> <p><b>For ESBC Phase</b></p> <p><b>Size of image</b> to be validated</p>
0x1f-0x1c	<p><b>For ISBC Phase:</b></p> <p><b>ESBC entry point.</b></p> <p>ISBC transfers control to this location upon successful validation of ESBC image(s).</p> <p><b>For ESBC Phase:</b> Reserved</p>
0x21-0x20	<p><b>Manufacturing Protection Flag</b></p> <p>Indicates if manufacturing protection has to be enabled or not in ISBC.</p>
0x23-0x22	<p><b>Reserved</b> .(Earlier this field was SG Flag. SG flag is always assumed to be 1 in unified implementation.)</p>
<i>Table continues on the next page...</i>	

**Table 188. CSF Header Format (LS1 Platform) (continued)**

Offset	Data Bits [0:31]
0x24	For ISBC Phase: Reserved For ESBC Phase: Reserved
0x25	<b>For ISBC Phase</b> <b>Secondary Image flag</b> Indicates if user has a secondary image available in case of failures in validating primary image. Valid in case of primary Images's Header. <b>For ESBC Phase:Reserved</b>
0x27-0x26	<b>Unique ID Usage</b> This location contains a flag which specifies one of these possibilities <ul style="list-style-type: none"> <li>• 0x00 - No UID's present</li> <li>• 0x01 - FSL UID and OEM UID are present</li> <li>• 0x02 - Only FSL UID is present</li> <li>• 0x04 - Only OEM UID is present</li> </ul>
0x2b-0x28	<b>Freescale unique ID 0</b> Upper 32 bits of a unique 64 bit value, which is specific to Freescale. This value is compared with the FSL ID 1 in Secure Fuse Processor 's FSL-ID registers
0x2f-0x2c	<b>OEM unique ID 0</b> Upper 32 bits of a unique 64 bit value, which is specific to OEM. This value is compared with the OEM ID 0 in Secure Fuse Processor 's OEM-ID registers
0x37-0x30	Reserved
0x3b-0x38	<b>Freescale unique ID 1</b> Lower 32 bits of a unique 64 bit value, which is specific to Freescale. This value is compared with the FSL ID 1 in Secure Fuse Processor 's FSL-ID registers
0x3f-0x3c	<b>OEM unique ID 1</b> Lower 32 bits of a unique 32 bit value, which is specific to OEM. This value is compared with the OEM ID 1 in Secure Fuse Processor 's OEM-ID registers
0x40-0x47	For ISBC Phase: Not Applicable For ESBC Phase: Reserved
0x48-0x4b	<b>For ISBC Phase: Not Applicable</b> <b>For ESBC Phase:</b> <b>ISBC key Extension flag</b> If this flag is set, key to be used for validation needs to be picked up from IE Key table.
<i>Table continues on the next page...</i>	

**Table 188. CSF Header Format (LS1 Platform) (continued)**

Offset	Data Bits [0:31]
0x4c-0x4f	<p><b>For ISBC Phase:</b> Not Applicable</p> <p><b>For ESBC Phase:</b></p> <p><b>IE Key Select</b></p> <p>Key Number to be used from the IE Key Table if IE flag is set.</p>

**Table 189. Scatter Gather Table Format (LS1 Platform)**

Offset	Data Bits [0:31]
0x00-0x03	Length. This location specifies the length in bytes of the ESBC image 1.
0x04-0x07	Target where the ESBC Image 1 can be found. This field is ignored in case of PBL based SOC's.
0x08-0x0b	Source Address of ESBC Image 1
0x0c-0x0f	<p>Destination Address of ESBC Image 1</p> <p>If the target address is 0xffffffff, the image is not copied to the target. This field is ignored in case of PBL based SOC's.</p>
0x10-0x13	Length. This location specifies the length in bytes of the ESBC image 2.
0x14-0x17	Target where the ESBC Image 2 can be found. This field is ignored in case of PBL based SOC's.
0x18-0x1b	Source Address of ESBC Image 2
0x1c-0x1f	<p>Destination Address of ESBC Image 2</p> <p>If the target address is 0xffffffff, the image is not copied to the target. This field is ignored in case of PBL based SOC's.</p>

**Table 190. Signature (LS1 Platform)**

Offset	Data Bits [0:31]
0x00-size	The RSA signature calculated over CSF Header, Scatter Gather table and ESBC image(s).

**Table 191. Public key (LS1 Platform)**

Offset	Data Bits [0:31]
0x00-size	Public Key Value. The hash of this public key is compared with the hash stored in Secure Fuse Processor SRKH registers.

**Table 192. SRK Table (LS1 Platform)**

<b>Offset</b>	<b>Data Bits [0:31]</b>
0x00-0x03	Key 1 length
0x04-0x403	Key 1 value. (Remaining bytes will be padded with zero)
0x404-0x407	Key 2 length
0x408-0x807	Key 2 value. (Remaining bytes will be padded with zero)
0x808-0x80b	Key 3 length
0x80c-0xb0b	Key 3 value. (Remaining bytes will be padded with zero)
0xb0c-0xb0f	Key 4 length
0xb10-0xe10	Key 4 value. (Remaining bytes will be padded with zero)

LS1043/LS1012 Platforms

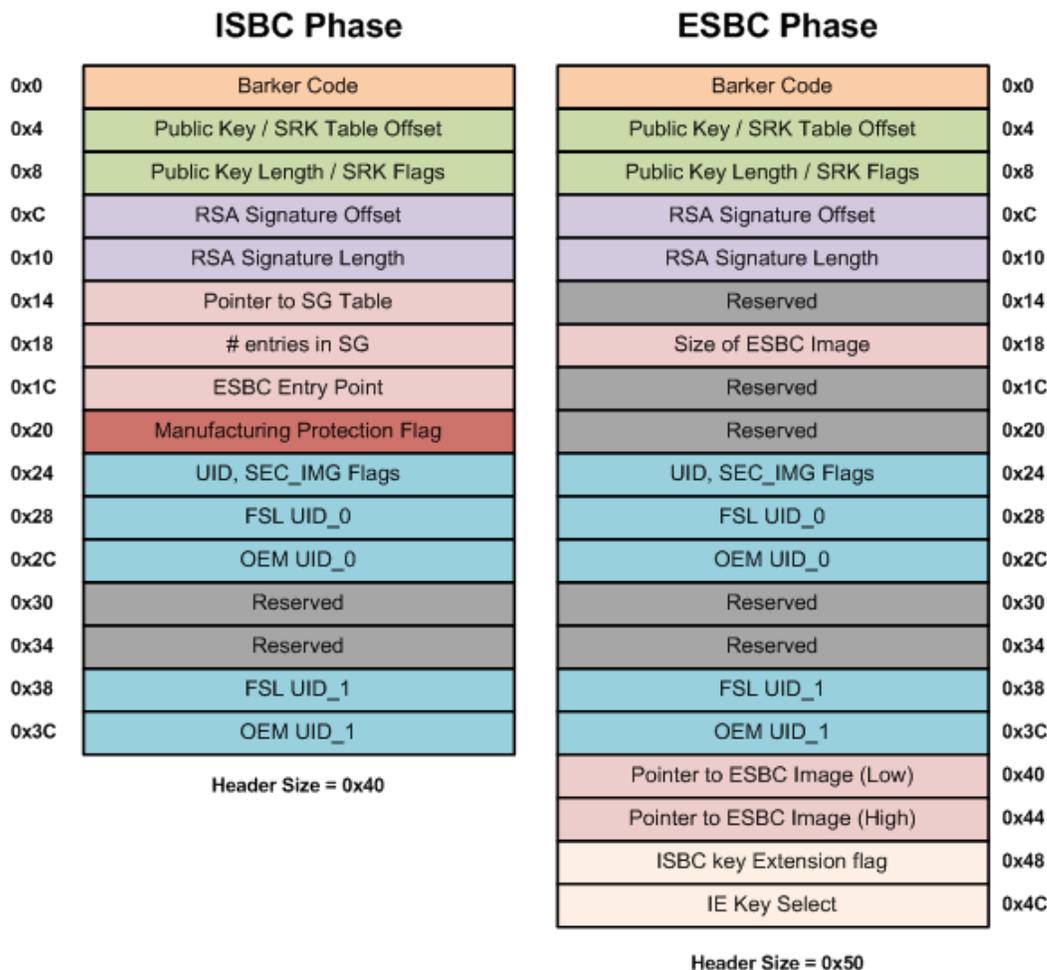


Figure 194. CSF Header for LS1043/LS1012/LS1046 (ISBC and ESBC Phase)

Table 193. CSF Header Format (LS1043/LS1012/LS1046 Platforms)

Offset	Data Bits [0:31]
0x00-0x03	<p><b>Barker code.</b></p> <p>This location should contain the value: 0x68392781. The ISBC code searches for this Barker code. If the value in this location does not match the Barker code, the ISBC stops execution and reports error.</p>
0x07-0x04	<p>If the srk_table_flag is not set :</p> <ul style="list-style-type: none"> <li>• <b>Public key offset:</b> This location contains an address which is the offset of the public key from the start of CSF header. Using this offset and the public key length, the public key is read.</li> </ul> <p>If srk_table_flag is set:</p> <ul style="list-style-type: none"> <li>• <b>Srk table offset:</b> This location contains an address which is the offset of the srk table from the start of CSF header. Using this offset and the number of entries is SRK Table, the SRK table is read.</li> </ul>

Table continues on the next page...

**Table 193. CSF Header Format (LS1043/LS1012/LS1046 Platforms) (continued)**

Offset	Data Bits [0:31]
0x08	<p><b>Srk table flag.</b></p> <p>This flag indicates whether hash burnt in srk fuse is of a single key or of srk table.</p>
0x0b-0x09	<p>If the srk_table_flag is not set :</p> <ul style="list-style-type: none"> <li>• <b>0x0b-0x9</b> -- <b>Public key length:</b> This location contains the length of the public key in bytes.</li> </ul> <p>If srk_table_flag is set:</p> <ul style="list-style-type: none"> <li>• <b>0x09</b> – <b>Key Number from srk table</b> which is to be used for verification.</li> <li>• <b>0x0b-0x0a</b> – <b>Number of entries in srk table.</b> Minimum number of entries in table = 1, Maximum = 4.</li> </ul>
0x0f-0x0c	<p><b>RSA Signature offset.</b></p> <p>This location contains an offset(in bytes) of the RSA signature from the start of CSF header. Using this offset and the Signature length, the RSA signature is read. The RSA signature is calculated over CSF Header, Scatter Gather table and ESBC images.</p>
0x13-0x10	<p><b>RSA Signature length</b> in bytes.</p>
0x17-0x14	<p><b>For ISBC Phase:</b></p> <p><b>SG Table offset</b></p> <p>This location contains an address which is the offset of the SG table from the start of CSF header. Using this offset and the number of entries in SG Table, the SG table is read.</p> <p><b>For ESBC Phase:</b></p> <p>Reserved</p>
0x1b-0x18	<p><b>For ISBC Phase:</b></p> <p><b>Number of entries in SG Table</b> (Earlier ,Based on the Scatter gather flag in CSF Header, this location can either be treated as number of entries in SG table or ESBC image size in bytes.).</p> <p><b>For ESBC Phase</b></p> <p><b>Size of image</b> to be validated</p>
0x1f-0x1c	<p><b>For ISBC Phase:</b></p> <p><b>ESBC entry point.</b></p> <p>ISBC transfers control to this location upon successful validation of ESBC image(s).</p> <p><b>For ESBC Phase:</b> Reserved</p>
0x21-0x20	<p><b>Manufacturing Protection Flag</b></p> <p>Indicates if manufacturing protection has to be enabled or not in ISBC.</p>
0x23-0x22	<p><b>Reserved</b> .(Earlier this field was SG Flag. SG flag is always assumed to be 1 in unified implementation.)</p>
<p><i>Table continues on the next page...</i></p>	

**Table 193. CSF Header Format (LS1043/LS1012/LS1046 Platforms) (continued)**

Offset	Data Bits [0:31]
0x24	For ISBC Phase: Reserved For ESBC Phase: Reserved
0x25	<b>For ISBC Phase</b> <b>Secondary Image flag</b> Indicates if user has a secondary image available in case of failures in validating primary image. Valid in case of primary Images's Header. <b>For ESBC Phase:Reserved</b>
0x27-0x26	<b>Unique ID Usage</b> This location contains a flag which specifies one of these possibilities <ul style="list-style-type: none"> <li>• 0x00 - No UID's present</li> <li>• 0x01 - FSL UID and OEM UID are present</li> <li>• 0x02 - Only FSL UID is present</li> <li>• 0x04 - Only OEM UID is present</li> </ul>
0x2b-0x28	<b>Freescale unique ID 0</b> Upper 32 bits of a unique 64 bit value, which is specific to Freescale. This value is compared with the FSL ID 1 in Secure Fuse Processor 's FSL-ID registers
0x2f-0x2c	<b>OEM unique ID 0</b> Upper 32 bits of a unique 64 bit value, which is specific to OEM. This value is compared with the OEM ID 0 in Secure Fuse Processor 's OEM-ID registers
0x37-0x30	Reserved
0x3b-0x38	<b>Freescale unique ID 1</b> Lower 32 bits of a unique 64 bit value, which is specific to Freescale. This value is compared with the FSL ID 1 in Secure Fuse Processor 's FSL-ID registers
0x3f-0x3c	<b>OEM unique ID 1</b> Lower 32 bits of a unique 32 bit value, which is specific to OEM. This value is compared with the OEM ID 1 in Secure Fuse Processor 's OEM-ID registers
0x40-0x47	<b>For ISBC Phase:</b> Not Applicable <b>For ESBC Phase:</b> 64 bit pointer to ESBC image
0x48-0x4b	<b>For ISBC Phase:</b> Not Applicable <b>For ESBC Phase:</b> <b>ISBC key Extension flag</b> If this flag is set, key to be used for validation needs to be picked up from IE Key table.
<i>Table continues on the next page...</i>	

**Table 193. CSF Header Format (LS1043/LS1012/LS1046 Platforms) (continued)**

Offset	Data Bits [0:31]
0x4c-0x4f	<p><b>For ISBC Phase:</b> Not Applicable</p> <p><b>For ESBC Phase:</b></p> <p><b>IE Key Select</b></p> <p>Key Number to be used from the IE Key Table if IE flag is set.</p>

**Table 194. Scatter Gather Table Format (LS1043/LS1012/LS1046 Platforms)**

Offset	Data Bits [0:31]
0x00-0x03	Length. This location specifies the length in bytes of the ESBC image 1.
0x04-0x07	Target where the ESBC Image 1 can be found. This field is ignored in case of PBL based SOC's.
0x08-0x0b	Source Address of ESBC Image 1
0x0c-0x0f	<p>Destination Address of ESBC Image 1</p> <p>If the target address is 0xffffffff, the image is not copied to the target. This field is ignored in case of PBL based SOC's.</p>
0x10-0x13	Length. This location specifies the length in bytes of the ESBC image 2.
0x14-0x17	Target where the ESBC Image 2 can be found. This field is ignored in case of PBL based SOC's.
0x18-0x1b	Source Address of ESBC Image 2
0x1c-0x1f	<p>Destination Address of ESBC Image 2</p> <p>If the target address is 0xffffffff, the image is not copied to the target. This field is ignored in case of PBL based SOC's.</p>

**Table 195. Signature (LS1043/LS1012/LS1046 Platforms)**

Offset	Data Bits [0:31]
0x00-size	The RSA signature calculated over CSF Header, Scatter Gather table and ESBC image(s).

**Table 196. Public key (LS1043/LS1012/LS1046 Platforms)**

Offset	Data Bits [0:31]
0x00-size	Public Key Value. The hash of this public key is compared with the hash stored in Secure Fuse Processor SRKH registers.

**Table 197. SRK Table (LS1043/LS1012/LS1046 Platforms)**

Offset	Data Bits [0:31]
0x00-0x03	Key 1 length
0x04-0x403	Key 1 value. (Remaining bytes will be padded with zero)
0x404-0x407	Key 2 length
0x408-0x807	Key 2 value. (Remaining bytes will be padded with zero)
0x808-0x80b	Key 3 length
0x80c-0xb0b	Key 3 value. (Remaining bytes will be padded with zero)
0xb0c-0xb0f	Key 4 length
0xb10-0xe10	Key 4 value. (Remaining bytes will be padded with zero)

## 8.2.11 ISBC Validation Error Codes

### P3/P4/P5 platforms

**Table 198. ISBC Validation Failures (P3/P4/P5 platforms)**

Value	Code	Definition
0x1	CPUID_NO_MATCH	ISBC is not running on CPU0
0x2	ESBC_HDR_LOC	ESBC header location is not in 3.5G space
0x4	ESBC_HEADER_BARKER	Barker code in the header is incorrect.
0x8	ESBC_HEADER_KEY_LEN	Length of public key in header is not one of the supported values.
0x10	ESBC_HEADER_SIGN_LEN	Length of RSA signature in header is not one of the supported values.
0x20	ESBC_HEADER_KEY_LEN_NOT_TWICE_SIG_LEN	Public key is not twice the length of the RSA signature
0x40	ESBC_HEADER_SG_TABLE_ADDR_NULL	SG table/ESBC image address (0x14-0x17 in CSF Header) is null
0x80	ESBC_HEADER_SG_TABLE_ADDR_NOT_IN_3_5G	SG table/ESBC image address (0x14-0x17 in CSF Header) is beyond 3.5G
0x100	ESBC_HEADER_KEY_MOD_1	Most significant bit of modulus in header is zero.

*Table continues on the next page...*

**Table 198. ISBC Validation Failures (P3/P4/P5 platforms) (continued)**

Value	Code	Definition
0x200	ESBC_HEADER_KEY_MOD_2	Modulus in header is even number
0x400	ESBC_HEADER_SIG_KEY_MOD	Signature value is greater than modulus in header
0x800	ESBC_HEADER_SG_ENTRIES_NUL	SG Table contains zero entries
0x1000	ESBC_HEADER_SG_ENTRIES_NOT_IN_3_5G	Address in SG entry is not in 3.5G
0x2000	ESBC_HEADER_SG_ESBC_EP	ESBC entry point in header not within ESBC address range
0x4000	HASH_COMPARE_KEY	Super Root Key Hash Comparison failure. Mismatch in the hash of the public key as present in the header with the value in the SRK HASH fuse.
0x8000	HASH_COMPARE_EM	RSA signature check failure. Signature provided by you in the header doesn't match with the signature of the ESBC image generated by ISBC. The ESBC image loaded by you may be different than the image used while generating the signature(using CST)
0x10000	SSM_CHECKSTS	SEC_MON State Machine not in CHECK state at start of ISBC. Some Security violation could have occurred. Details can be found in P4080 Reference Manual.
0x20000	SSM_TRUSTSTS	SEC_MON State Machine not in TRUSTED state at end of ISBC.
0x40000	FSL_UID	FSL_UID in ESBC Header did not match the FSL_UID in SFP
0x80000	OEM_UID	OEM_UID in ESBC Header did not match the OEM_UID in SFP
0x100000	BAD_ADDRESS	A Data / Instruction TLB Exception occurred during ISBC execution
0x200000	MISC	E500mc exception other than TLB
0x400000	ESBC_HEADER_SG_ENTRIES_BAD	SG Table too large (too many entries)

**NOTE**

For error codes 0x2 - 0x2000 i.e errors in the ESBC Header, check the value of that particular field by dumping the header.

**B4/T1/T2/T4/LS1/LS1043/LS1012 platforms**

Errors in the system can be of following types:

1. Core Exceptions
2. System State Failures

- 3. Header Checking Failures
  - a. General Failures
  - b. Key/Signature/UID related errors
- 4. Verification Failures
- 5. SEC/PAMU errors

**Table 199. Core Exceptions (LS1 platform)**

Value	Code	Definition
0x1	ERROR_UNDEFINED_INSTRUCTION	Occurs if neither the processor nor any attached co-processor recognizes the currently executing instruction.
0x2	ERROR_SWI	Software Interrupt is a user-defined interrupt instruction. It allows a program running in User mode, for example, to request privileged operations that run in Supervisor mode.
0x3	ERROR_PREFETCH_ABORT	Occurs when the processor attempts to execute an instruction that has been prefetched from an illegal address.
0x4	ERROR_DATA_ABORT	Occurs when a data transfer instruction attempts to load or store data at an illegal address.
0x5	ERROR_IRQ	Occurs when the processor external interrupt request pin is asserted (LOW) and IRQ interrupts are enabled.
0x6	ERROR_FIQ	Occurs when the processor external fast interrupt request pin is asserted (LOW) and FIQ interrupts are enabled.

**Table 200. Core Exceptions (LS1043/LS1012/LS1046 platforms)**

Error Code	Value
<b>Current EL with SP0</b>	
ERROR_EXCEPTION_SYNC_SP0	0x01
ERROR_EXCEPTION_IRQ_SP0	0x02
ERROR_EXCEPTION_FIQ_SP0	0x03
ERROR_EXCEPTION_SERROR_SP0	0x04
<b>Current EL with SPx</b>	
ERROR_EXCEPTION_SYNC_SPX	0x05
ERROR_EXCEPTION_IRQ_SPX	0x06
ERROR_EXCEPTION_FIQ_SPX	0x07
ERROR_EXCEPTION_SERROR_SPX	0x08
<b>Lower EL using AArch64</b>	
ERROR_EXCEPTION_SYNC_L64	0x11
ERROR_EXCEPTION_IRQ_L64	0x12
<i>Table continues on the next page...</i>	

**Table 200. Core Exceptions (LS1043/LS1012/LS1046 platforms) (continued)**

ERROR_EXCEPTION_FIQ_L64	0x13
ERROR_EXCEPTION_SERROR_L64	0x14
<b>Lower EL using AArch32</b>	
ERROR_EXCEPTION_SYNC_L32	0x15
ERROR_EXCEPTION_IRQ_L32	0x16
ERROR_EXCEPTION_FIQ_L32	0x17
ERROR_EXCEPTION_SERROR_L32	0x18

**Table 201. Core Exceptions (B4/T1/T2/T4 platforms)**

Value	Code	Definition
0x1	ERROR_MACHINECHECK	Machine check Exception
0x2	ERROR_DSI	DSI Exception
0x3	ERROR_ISI	ISI Exception
0x4	ERROR_CRITICAL	Critical Exception
0x5	ERROR_ALIGN	Alignment Exception
0x6	ERROR_PROG	Program Exception
0x13	ERROR_DATA_TLB	Data TLB Miss
0x14	ERROR_INST_TLB	Instruction TLb Miss
0x20	ERROR_MISC	Any other exception

**Table 202. System State Failures (B4/T1/T2/T4/LS1/LS1043/LS1012/LS1046 platforms)**

Value	Code	Definition
0x100	ERROR_CORE_NON_ZERO	ISBC is not running on CPU0
0x101	ERROR_STATE_NOT_CHECK	SEC_MON State Machine not in CHECK state at start of ISBC. Some Security violation could have occurred.
0x102	ERROR2_STATE_NOT_CHECK	SEC_MON State Machine not in CHECK state, when trying to transition it to Trusted/Non Secure/Soft Fail state
0x103	ERROR_SSM_TRUSTSTS	SEC_MON State Machine not in TRUSTED state at end of ISBC.

**Table 203. General Header Checking Failures (B4/T1/T2/T4/LS1/LS1043/LS1012/LS1046 platforms)**

Value	Code	Definition
0x301	ERROR_ESBC_HDR_LOC	ESBC header location is not in 3.5G space
0x302	ERROR_ESBC_HEADER_BARKER	Barker code in the header is incorrect.
0x303	ERROR_ESBC_HEADER_SG_ENTRIES_NOT_IN_3_5G	SG table/ESBC image address (header address + image offset in sg table) is beyond 3.5G
0x303	ERROR_ESBC_HEADER_SG_ENTRIES_ON_OCRAM	One Entry in the SG table is on OCRAM
0x304	ERROR_ESBC_HEADER_SG_ESBC_EP	ESBC entry point in header not within ESBC address range
0x305	ERROR_SGL_ENTIRES_NOT_SUPPORTED	Number of entries in SG table exceeds maximum limit i.e 8
0x306	ERROR_ESBC_HEADER_HKAREA_LEN_ZERO	Houskeeping area not provided in header
0x307	ERROR_ESBC_HEADER_HKAREA_NOT_IN_3_5G	House keeping area not in 3.5G boundary
0x308	ERROR_ESBC_HEADER_HKAREA_LEN_INSUFFICIENT	Housekeeping area length provided is not sufficient.
0x309	ERROR_SG_TABLE_NOT_IN_3_5	SG Table is not in 3.5G boundary
0x309	ERROR_SG_TABLE_ON_OCRAM	SG table is on OCRAM
0x310	ERROR_ESBC_HEADER_HKAREA_NOT_4K_ALIGNED	House keeping area is not aligned to 4K boundary
0x311	ERROR_SGL_ENTRIES_SIZE_ZERO	SG table has entry with size zero.

**Table 204. Key/Signature/UID related errors (B4/T1/T2/T4/LS1/LS1043/LS1012/LS1046 platforms)**

Value	Code	Definition
0x320	ERROR_ESBC_HEADER_KEY_LEN	Length of public key in header is not one of the supported values.
0x321	ERROR_ESBC_HEADER_KEY_LEN_NOT_TWICE_SIG_LEN	Public key is not twice the length of the RSA signature
0x322	ERROR_ESBC_HEADER_KEY_MOD_1	Most significant bit of modulus in header is zero.
0x323	ERROR_ESBC_HEADER_KEY_MOD_2	Modulus in header is even number
0x324	ERROR_ESBC_HEADER_SIG_KEY_MODAL	Signature value is greater than modulus in header
0x325	ERROR_FSL_UID	FSL_UID in ESBC Header did not match the FSL_UID in SFP if fsl uid flag is 1
0x326	ERROR_OEM_UID	OEM_UID in ESBC Header did not match the OEM_UID in SFP if oem uid flag is 1.

*Table continues on the next page...*

**Table 204. Key/Signature/UID related errors (B4/T1/T2/T4/LS1/LS1043/LS1012/LS1046 platforms)  
(continued)**

Value	Code	Definition
0x327	ERROR_INVALID_SRK_NUM_ENTRY	Number of entries field in CSF Header is > 4(This is when srk_flag in header is 1)
0x328	ERROR_INVALID_KEY_NUM	Key number to be used from srk table is not present in table. ( This is when srk_flag in header is 1)
0x329	ERROR_KEY_REVOKED	Key selected from srk table has been revoked(This is when srk_flag in header is 1)
0x32a	ERROR_INVALID_SRK_ENTRY_KEYLEN	Key length specified in one of the entries in srk table is not one of the supported values (This is when srk_flag in header is 1)
0x32b	ERROR_SRK_TBL_NOT_IN_3_5	SRK Table is not in 3.5G boundary (This is when srk_flag in header is 1)
0x32b	ERROR_SRK_TBL_ON_OCRAM	SRK Table is on OCRAM
0x32c	ERROR_KEY_NOT_IN_3_5G	Key is not in 3.5G boundary
0x32c	ERROR_KEY_ON_OCRAM	Key on OCRAM

**Table 205. Verification Failures (B4/T1/T2/T4/LS1/LS1043/LS1012/LS1046 platforms)**

Value	Code	Definition
0x340	ERROR_HASH_COMPARE_KEY	Super Root Key Hash Comparison failure. Mismatch in the hash of the public key/srk table as present in the header with the value in the SRK HASH fuse.
0x341	ERROR_HASH_COMPARE_EM	RSA signature check failure. Signature provided by you in the header doesn't match with the signature of the ESBC image generated by ISBC. The ESBC image loaded by you may be different than the image used while generating the signature(using CST)

**Table 206. SEC/PAMU Failures (B4/T1/T2/T4/LS1/LS1043/LS1012/LS1046 platforms)**

Value	Code	Definition
0x700	ERROR_SEC_ENQ	Error when enqueueing to SEC
0x701	ERROR_SEC_DEQ	Sec Block returned some error when dequeuing from it.
0x702	ERROR_SEC_DEQ_TO	Timeout when trying to deq from SEC
0x800	ERROR_PAMU	Error while programming PAACT/SPAACT tables in PAMU (For PowerPC platforms only)

## 8.2.12 ESBC Validation Error Codes

For trust arch version 1.x and 2.x.

**Table 207. ESBC Validation Failures**

Value	Code	Definition
0x4	ERROR_ESBC_CLIENT_HEADER_BARKER	Wrong barker code in header
0x8	ERROR_ESBC_CLIENT_HEADER_KEY_LEN	Wrong public key length in header
0x10	ERROR_ESBC_CLIENT_HEADER_SIG_LEN	Wrong signature length in header
0x20	ERROR_ESBC_CLIENT_HEADER_KEY_LEN_NOT_TWICE_SIG_LEN	Public key length not twice of signature length
0x40	ERROR_ESBC_CLIENT_HEADER_KEY_MOD_1	Public key Modulus most significant bit not set
0x80	ERROR_ESBC_CLIENT_HEADER_KEY_MOD_2	Public key Modulus in header not odd
0x100	ERROR_ESBC_CLIENT_HEADER_SIG_KEY_MOD	Signature not less than modulus
0x400	ERROR_ESBC_CLIENT_HASH_COMPARE_KEY	Public key hash comparison failed
0x800	ERROR_ESBC_CLIENT_HASH_COMPARE_EM	RSA verification failed
0x10000	ERROR_ESBC_CLIENT_HEADER_SG	No SG support
0x20000	ERROR_ESBC_WRONG_CMD	Failure in command/Unknown command/Wrong arguments of boot script.
0x40000	ERROR_ESBC_MISSING_BOOTM	Bootm command missing from boot script.

## 8.2.13 Trust Architecture and SFP Information

SoC	Trust Arch. Version	SFP Version	POVDD	DRVR		OTPMK		SNVS/SFP Register to check Hamming Error
				Algo (CST)	Register to check Hamming Error	Algo (CST)	Register to check Hamming Error	
P4080 rev1	1	1	1.5 V	A	None/ Simulation	1	SNVS	SecMon_HP Status (HPSR)
P4080 rev2	1	1.1	1.5 V	B	None/ Simulation	1	SNVS	

*Table continues on the next page...*

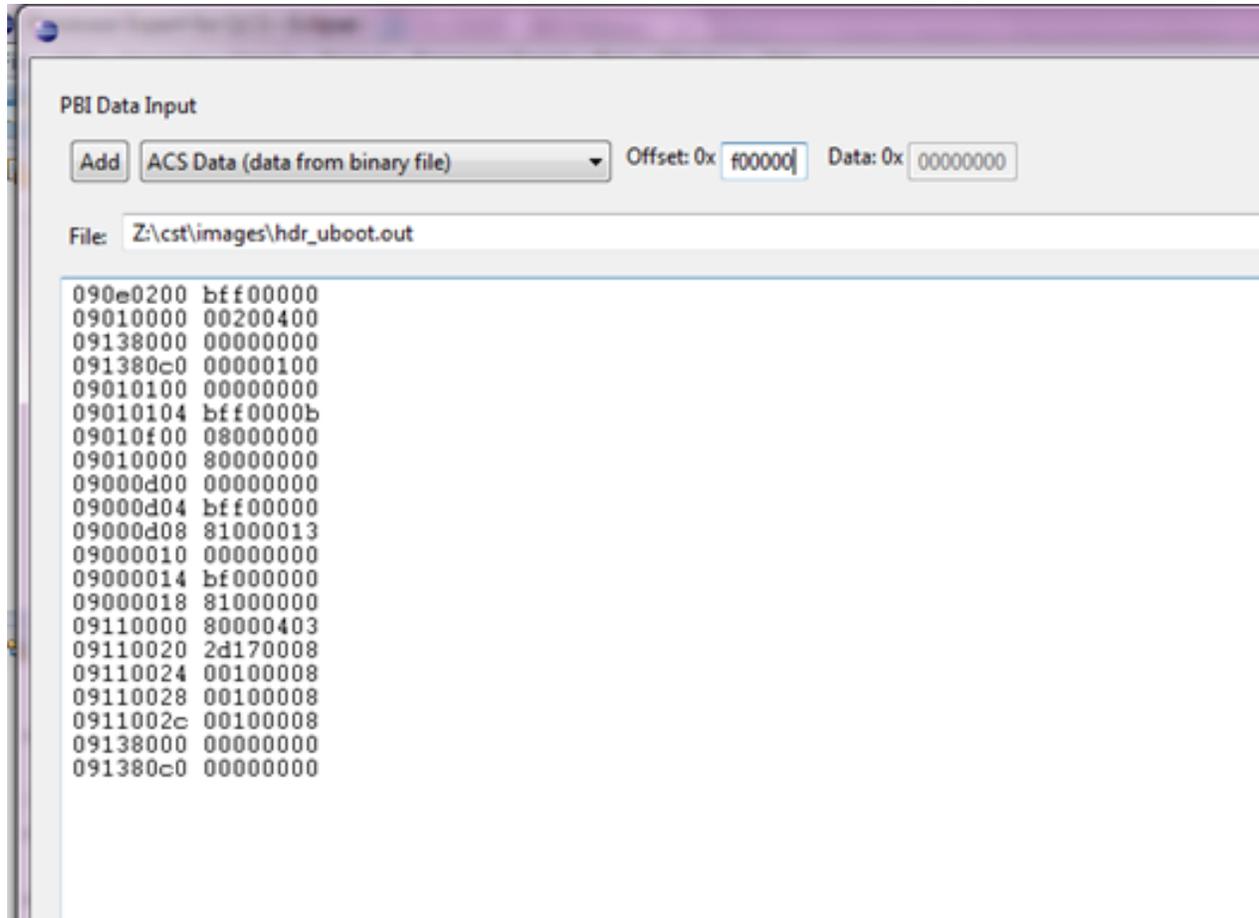
Table continued from the previous page...

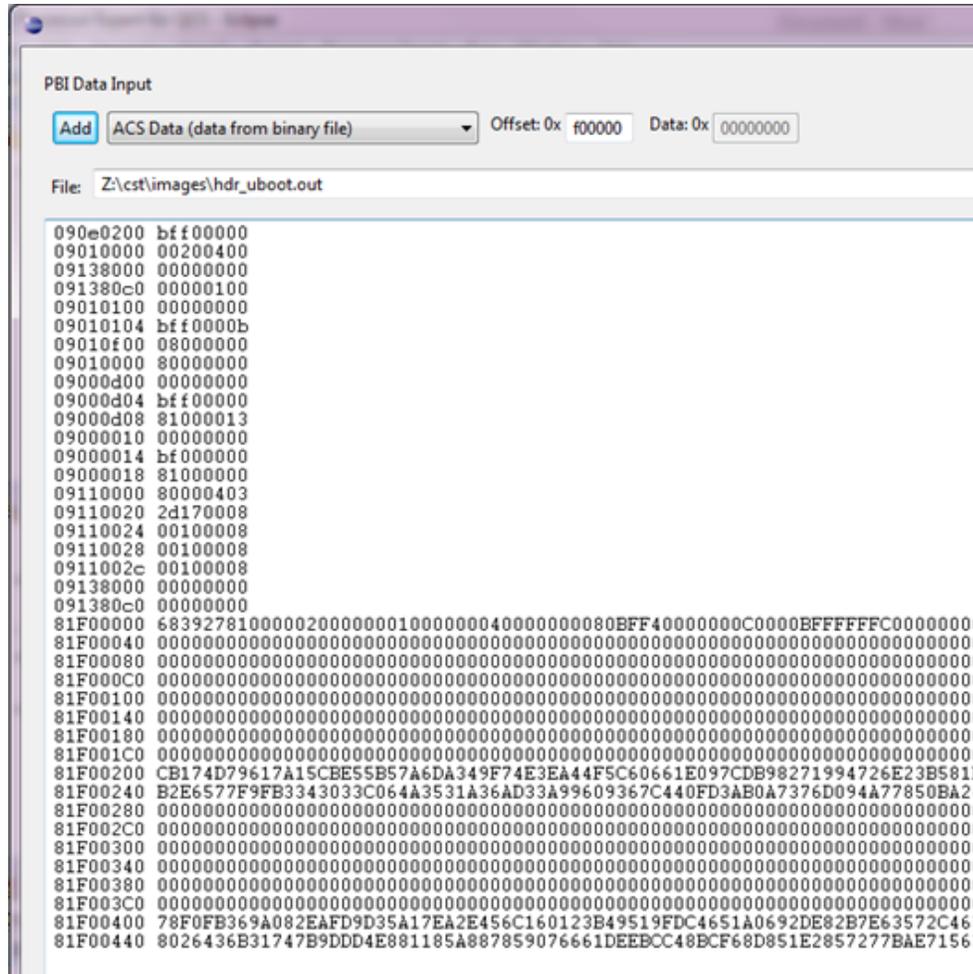
P4080 rev3	1	2.2	1.5 V	B	None/ Simulation	1	SNVS	
P3041	1.1	2	1.5 V	B	None/ Simulation	1	SNVS	
P5020	1.1	2	1.5 V	B	None/ Simulation	1	SNVS	
P5040	1.1	2.2	1.5 V	B	None/ Simulation	1	SNVS	
P5021	1.1	2.2	1.5 V	B	None/ Simulation	1	SNVS	
T4240 rev1	2	3.1	1.89 V	A	SFP	2	SFP	SFP Secret Value Hamming Error Status Register (SFP_SVHE SR)
T4240 rev2	2	3.2	1.89 V	A	SFP	2	SFP	
B4860 rev1	2	3.1	1.89 V	A	SFP	2	SFP	
B4860 rev2	2	3.2	1.89 V	A	SFP	2	SFP	
T2080	2	3.2	1.89 V	A	SFP	2	SFP	
T1040	2	3.2	1.89 V	A	SFP	2	SFP	
T1023	2	3.2	1.89 V	A	SFP	2	SFP	
LS1020A	2.1	3.3	1.89 V	A	SFP	2	SFP	
LS1043A	2.1	3.3	1.89 V	A	SFP	2	SFP	
LS1046A	2.1	3.3	1.89 V	A	SFP	2	SFP	

## 8.2.14 Using QCVS Tool (Secure Boot From NAND)

Use Freescale's QCVS tool for adding the `hdr_uboot.out` and `u-boot.bin` in terms of `ALT_CONFIG_WRITE` PBI commands at required addresses. The below screenshots describe the usage of QCVS Tool.

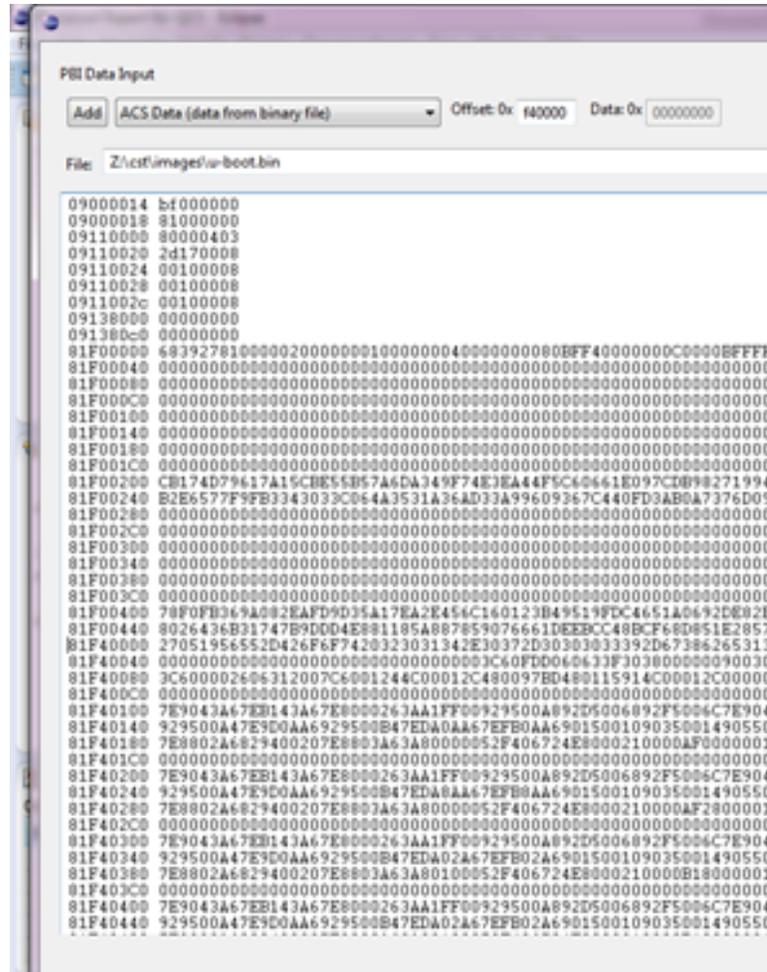
1. Import `rcw.bin` from SDK in QCVS Tool.
2. Add ACS Data `hdr_uboot.out` @ `0xF00000`.





3. Add ACS Data u-boot.bin @0xF4000.

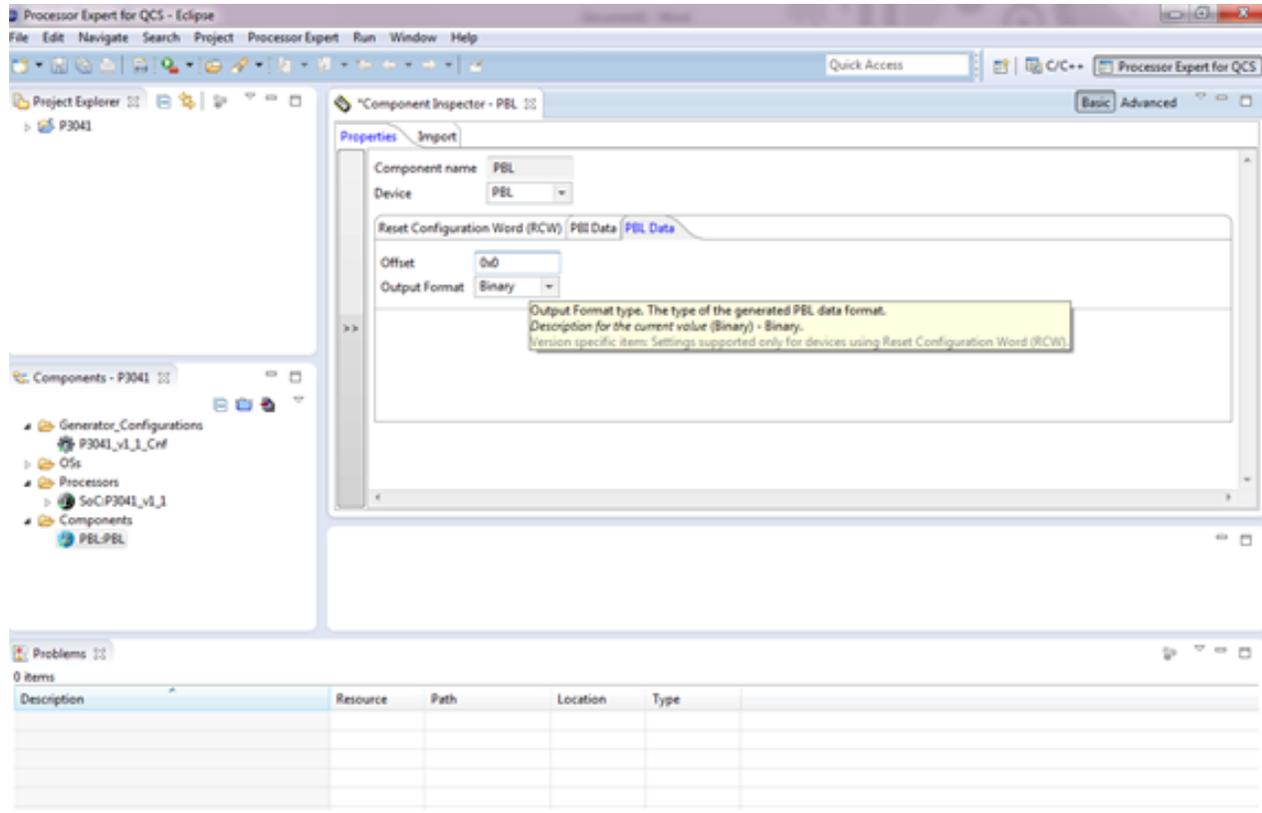




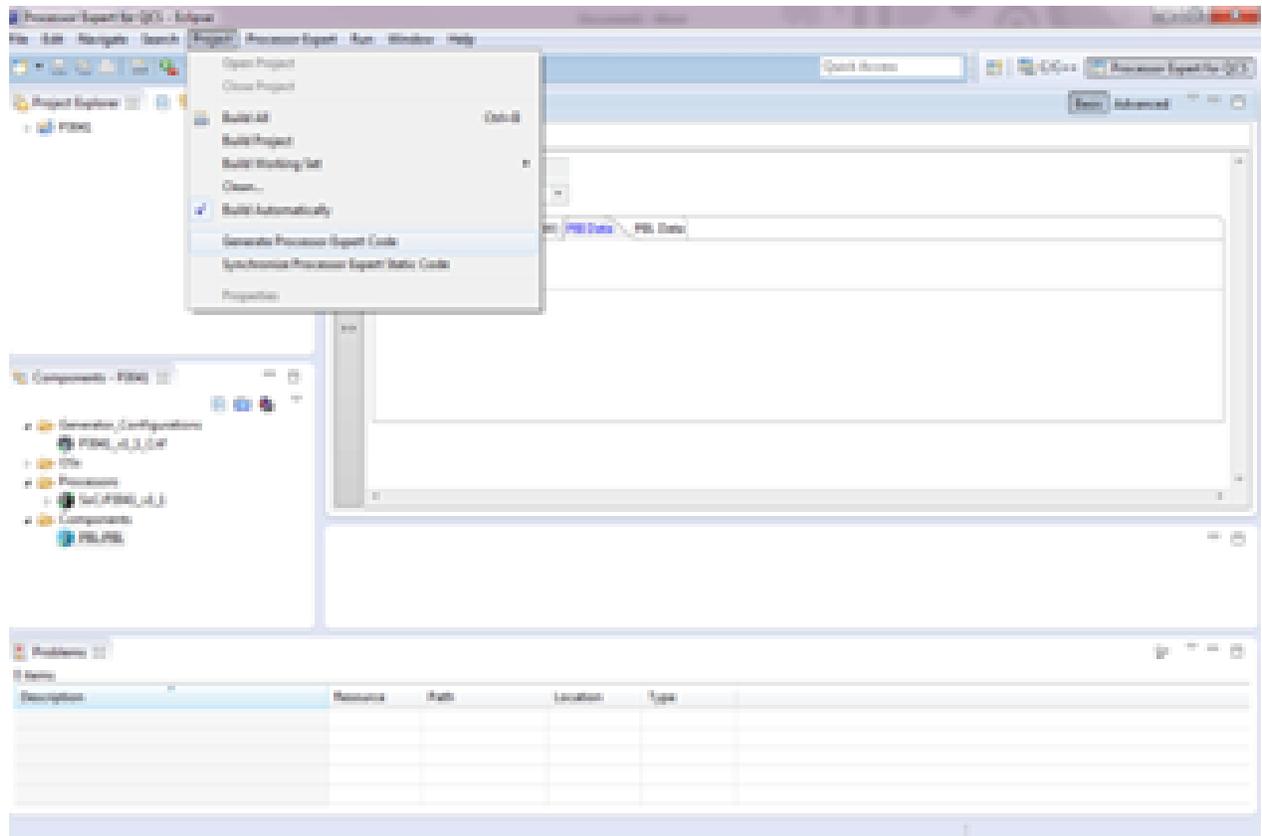
4. Make Sure that the output format is Binary.

## Boot Loaders

### Secure Boot: PBL Based Platforms



#### 5. Generate Processor Expert Code to get PBL.bin.



## 8.2.15 Appendix LS1046 Secure Boot demo

The addresses below are effective addresses as mapped by u-boot

**Boot Source: NOR**

**Table 208. Memory Map for LS1046 Platforms**

Address NOR(vBank 0)	Address (NOR Alternate Bank)	Definition (Chain of Trust)	Size Reserved (KB)
60000000	64000000	RCW	128
60060000	64060000	Bootscript	128
60080000	64080000	ESBC U-Boot HEADER	128
600A0000	640A0000	Bootscript Header	128
600C0000	640C0000	PPA Header	128
60100000	64100000	ESBC U-Boot	1024
60500000	64500000	PPA FIT Image	2048
60A00000*	64A00000*	Kernel FIT Image	54272
63F40000	64F40000	kernel Header	128

**NOTE**

For LS1046 Bootsript, kernel image must be copied to DDR address 0x81000000 before issuing esbc\_validate command.

### Chain Of Trust Boot Script Used as per Address Map

```
#Copy the Kernel Image from Flash to DDRcp.b
0x60A00000 0x81000000 0x3500000
#Validate the Kernel Image (The header has Image address as 0x81000000)
esbc_validate 0x63F40000
#Boot the validated Kernel FIT Image.
setenv bootargs "console=ttyS0,115200 root=/dev/ram0
earlycon=uart8250,0x21c0500";
setenv fdt_high "0xffffffffffffffff";
setenv initrd_high "0xffffffffffffffff";
bootm
$img_addr
```

### Useful U-Boot and CCS Commands

```
protect off all;
setenv path <tftp_path>
tftp 80000000 $path/rcw.bin;erase 64000000 +$filesize;cp.b 80000000 64000000 $filesize;
tftp 80000000 $path/hdr_uboot.out;erase 64080000 +$filesize;cp.b 80000000 64080000 $filesize;
tftp 80000000 $path/u-boot.bin;erase 64100000 +$filesize;cp.b 80000000 64100000 $filesize;
tftp 80000000 $path/hdr_bs.out;erase 640A0000 +$filesize;cp.b 80000000 640A0000 $filesize;
tftp 80000000 $path/bootsript;erase 64060000 +$filesize;cp.b 80000000 64060000 $filesize;
```

## Boot Loaders

### Secure Boot: PBL Based Platforms

```
tftp 80000000 $path/hdr_kernel.out;erase 67F40000 +$filesize;cp.b 80000000 67F40000 $filesize;
tftp 80000000 $path/kernel.itb;erase 64a00000 +$filesize;cp.b 80000000 64a00000 $filesize;
tftp 80000000 $path/hdr_ppa.out;erase 640C0000 +$filesize;cp.b 80000000 640C0000 $filesize;
tftp 80000000 $path/ppa.itb;erase 64500000 +$filesize;cp.b 80000000 64500000 $filesize;
```

```
Connect to CCS and configure Config Chain
ccs::config_server 0 10000
ccs::config_chain {ls1043a dap sap2}
display ccs::get_config_chain
#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem <dap chain pos> 0x1e90014 4 0 4
ccs::display_mem <dap chain pos> 0x1ee0200 4 0 4
#Write the SRK Hash Value in Mirror Registers
ccs::write_mem <dap chain pos> 0x1e80254 4 0 <SRKH1>
ccs::write_mem <dap chain pos> 0x1e80258 4 0 <SRKH2>
ccs::write_mem <dap chain pos> 0x1e8025c 4 0 <SRKH3>
ccs::write_mem <dap chain pos> 0x1e80260 4 0 <SRKH4>
ccs::write_mem <dap chain pos> 0x1e80264 4 0 <SRKH5>
ccs::write_mem <dap chain pos> 0x1e80268 4 0 <SRKH6>
ccs::write_mem <dap chain pos> 0x1e8026c 4 0 <SRKH7>
ccs::write_mem <dap chain pos> 0x1e80270 4 0 <SRKH8>
#Get the Core Out of Boot Hold-Off
ccs::write_mem <dap chain pos> 0x1ee00e4 4 0 0x00000001
```

### **Boot Source: QSPI**

**Table 209. Memory Map for LS1046 Platform**

Address QSPI <sup>[17]</sup>	Definition (Chain of Trust)	Size Reserved (KB)
40000000	RCW	128
40060000	Bootscript	128
40080000	ESBC U-Boot HEADER	128
400C0000	Bootscript Header	128
40100000	PPA Header	128
40480000	ESBC U-Boot	1024
40500000	PPA FIT Image	2048
40A00000 <sup>[18]</sup>	Kernel FIT Image	54272
43200000	kernel Header	128

### **Chain Of Trust Boot Script Used as per Address Map**

```
Copy the Kernel Image from Flash to DDR
cp.b 0x40A00000 0x81000000 0x2800000
#Validate the Kernel Image (The header has Image address as 0x81000000)
esbc_validate 0x43200000
```

[17] QSPI by default works in 64bit Big Endian as XIP memory. So the data has to be 64 bit byte swapped before loading on QSPI.

[18] The Kernel Image and its CSF Header may be placed on any other memory as well. The same must be copied to DDR before validation and Booting.

```
#Boot the validated Kernel FIT Image.
setenv bootargs "console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,0x21c0500";
setenv fdt_high "0xffffffffffffffff";
setenv initrd_high "0xffffffffffffffff";
bootm $img_addr
```

### Chain of Trust Boot Script Used as per Address Map (For Encapsulation)

```
sf probe 0:0
blob enc 0x41000000 0x80000000 0x2000000 0x40000000
sf erase 0x1000000 +2020000;
sf write 0x80000000 0x1000000 2020000;
```

### Chain of Trust Boot Script Used as per Address Map (For Decapsulation)

```
sf probe 0:0
sf read 0x84000000 0x1000000 2020000
blob dec 0x84000000 0x81000000 0x2000000 0x40000000
esbc_validate 0x407c0000
Boot the validated Kernel FIT Image.
setenv bootargs "console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500";
setenv fdt_high "0xffffffffffffffff";
setenv initrd_high "0xffffffffffffffff";
bootm $img_addr
```

### Useful U-Boot and CCS Commands

```
protect off all;
setenv path <tftp_path>
sf probe 0:1
tftp 0x80000000 $path/rcw_800_sben_swap.bin; sf erase 0x0 20000; sf write 0x80000000 0x0 20000
tftp 0x80000000 $path/hdr_swap_uboot.out; sf erase 0x80000 20000; sf write 0x80000000 0x80000
20000
tftp 0x80000000 $path/u-boot_swap.bin; sf erase 0x100000 80000; sf write 0x80000000 0x100000 80000
tftp 0x80000000 $path/hdr_swap_bs.out; sf erase 0xc0000 20000; sf write 0x80000000 0xc0000 20000
tftp 0x80000000 $path/bootscrip_swap; sf erase 0x60000 20000; sf write 0x80000000 0x60000 20000
tftp 0x80000000 $path/hdr_swap_kernel.out; sf erase 0x320000 20000; sf write 0x80000000
0x320000 20000
tftp 0x80000000 $path/kernel_swap.itb; sf erase 0xa00000 0x2800000; sf write 0x80000000 0xa00000
0x2800000
tftp 0x80000000 $path/hdr_swap_ppa.out; sf erase 0x480000 20000; sf write 0x80000000 0x480000
20000
tftp 0x80000000 $path/ppa_swap.itb; sf erase 0x500000 40000; sf write 0x80000000 0x500000 40000
```

```
Connect to CCS and configure Config Chain
ccs::config_server 0 10000
ccs::config_chain {ls1043a dap sap2}
display ccs::get_config_chain
#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem <dap chain pos> 0x1e90014 4 0 4
ccs::display_mem <dap chain pos> 0x1ee0200 4 0 4
#Write the SRK Hash Value in Mirror Registers
ccs::write_mem <dap chain pos> 0x1e80254 4 0 <SRKH1>
ccs::write_mem <dap chain pos> 0x1e80258 4 0 <SRKH2>
ccs::write_mem <dap chain pos> 0x1e8025c 4 0 <SRKH3>
ccs::write_mem <dap chain pos> 0x1e80260 4 0 <SRKH4>
ccs::write_mem <dap chain pos> 0x1e80264 4 0 <SRKH5>
ccs::write_mem <dap chain pos> 0x1e80268 4 0 <SRKH6>
ccs::write_mem <dap chain pos> 0x1e8026c 4 0 <SRKH7>
```

## Boot Loaders

### Secure Boot: PBL Based Platforms

```
ccs::write_mem <dap chain pos> 0x1e80270 4 0 <SRKH8>
#Get the Core Out of Boot Hold-Off
ccs::write_mem <dap chain pos> 0x1ee00e4 4 0 0x00000001
```

# Chapter 9

## Virtualization

### 9.1 KVM/QEMU

#### 9.1.1 KVM/QEMU Release Notes

This document describes current limitations in the release of KVM and QEMU for NXP SoCs. Copyright (C) 2013-2016 Freescale Semiconductor, Inc.

Freescale KVM/QEMU Release Notes 09/14/2016

Overview

-----

This document describes new features, current limitations, and known issues in KVM and QEMU for NXP QorIQ releases.

New Features

-----

Linux and QEMU versions:

- KVM is based on the current release Linux kernel
- QEMU is based on QEMU 2.4.0

Limitations

-----

The following items describe known limitations with this release of KVM/QEMU for ARM based platforms.

- VFIO-PCI is not supported
- PMU counters are not supported in the guest

#### 9.1.2 KVM for ARM Architecture Users Guide and Reference

##### 9.1.2.1 Introduction to KVM and QEMU

###### 9.1.2.1.1 Overview

This document is a guide and tutorial to building and using KVM (Kernel-based Virtual Machine) on NXP QorIQ SoCs.

Virtualization provides an environment that enables running multiple operating systems on a single computer system. Virtualization uses hardware and software technologies together to enable this by providing an abstraction layer between system hardware and the OS. The isolated environment in which OSes run is known as a *virtual machine* (or VM). The abstraction layer that manages all this is referred to as a *hypervisor or virtual machine manager*. The hypervisor layer operates at a privilege level higher than that of the operating systems, thus enabling it to enforce system security, ensure that virtual machines cannot interfere with each other, and transparently provide other services such as I/O sharing to the VM.

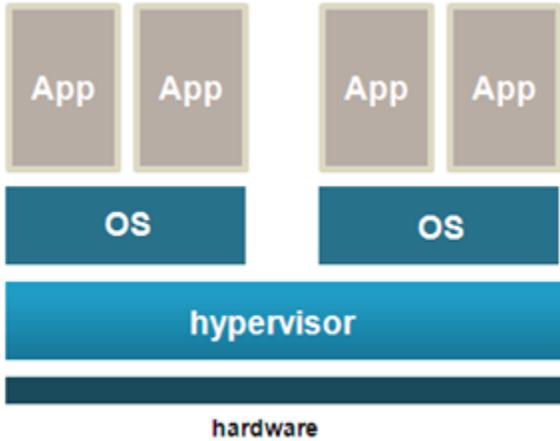


Figure 195.

KVM is a Linux kernel driver that together with QEMU, an open source machine emulator, provides an open source virtualization platform based on Linux. KVM and QEMU together act as a virtual machine manager that can boot and run operating systems in virtual machines. See Figure below.

In this document the term *host* kernel refers to the underlying instance of Linux with the KVM driver that acts as the hypervisor. The term *guest* refers to the operating system, such as Linux, that runs in a virtual machine. A virtual machine will be referred to as a "VM".

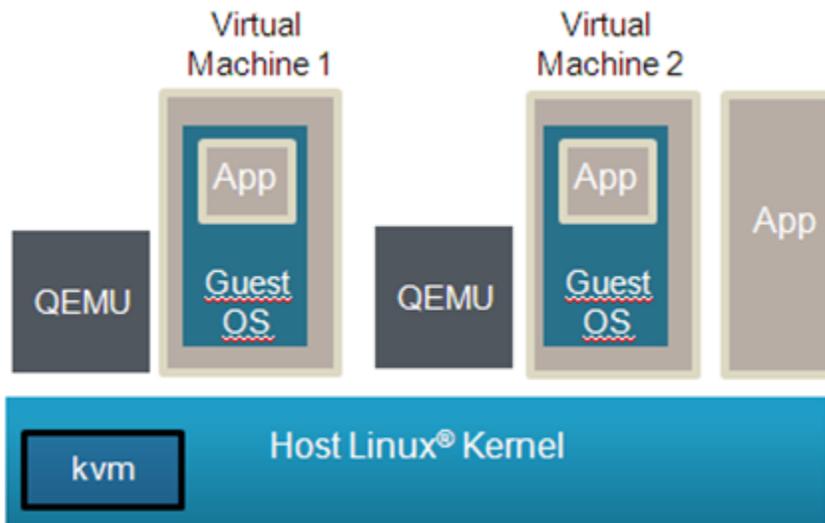


Figure 196.

NXP QorIQ SoCs based on ARM v7 and ARM v8 CPUs are supported.

### 9.1.2.1.2 Organization of this Document

This document is organized as follows:

- [Introduction to KVM and QEMU](#) on page 1215 provides an introduction to KVM/QEMU, including overview information and references.
- [Building QEMU and KVM](#) on page 1220 provides information on how to build QEMU and the Linux kernel with KVM.

- [Using QEMU and KVM](#) on page 1225 describes how to use KVM/QEMU, including how to invoke QEMU to start virtual machines and how to set up virtual I/O and passthrough I/O devices.
- [Virtual machine reference](#) on page 1229 provides a reference for virtual machines-- details about initial VM state, virtual CPUs, and virtual I/O devices. This information is relevant when porting an OS or device driver to a KVM-based virtual machine.
- [Debugging virtual machines](#) on page 1232 describes facilities available for debugging software running in a virtual machine.
- [KVM/QEMU How-to's](#) on page 1234 provides a set of examples for common tasks.

### 9.1.2.1.3 Virtual Machine Overview

A guest OS running in a KVM/QEMU virtual machine "sees" a hardware environment similar to running on a physical board. The guest sees CPUs, memory, and a number of I/O devices. Some aspects of this environment are virtualized (emulated in software by KVM/QEMU) but this virtualization is mostly transparent to the guest, and changes to the guest are typically not required to run in a virtual machine.

The number of virtual machines that can be run simultaneously is only limited by the amount of available resources (like any other application on Linux).

KVM/QEMU implements a generic virt machine which is described completely by the device tree. The virtual machine contains the following resources:

- one or more ARMv7/ARMv8 virtual CPUs
- memory
- virtual console based on an emulated PL011
- virtio over PCI (used for virtual devices such as block and network devices)
- ARM Virtual Generic Interrupt Controller
- ARM virtual timer and counter

### 9.1.2.1.4 Introduction to KVM and QEMU

QEMU (pronounced KYOO-em-yoo) is a software-based machine emulator that emulates a variety of CPUs and hardware systems. KVM is a Linux kernel device driver that provides virtual CPU services to QEMU. The two software components work together as a virtual machine manager.

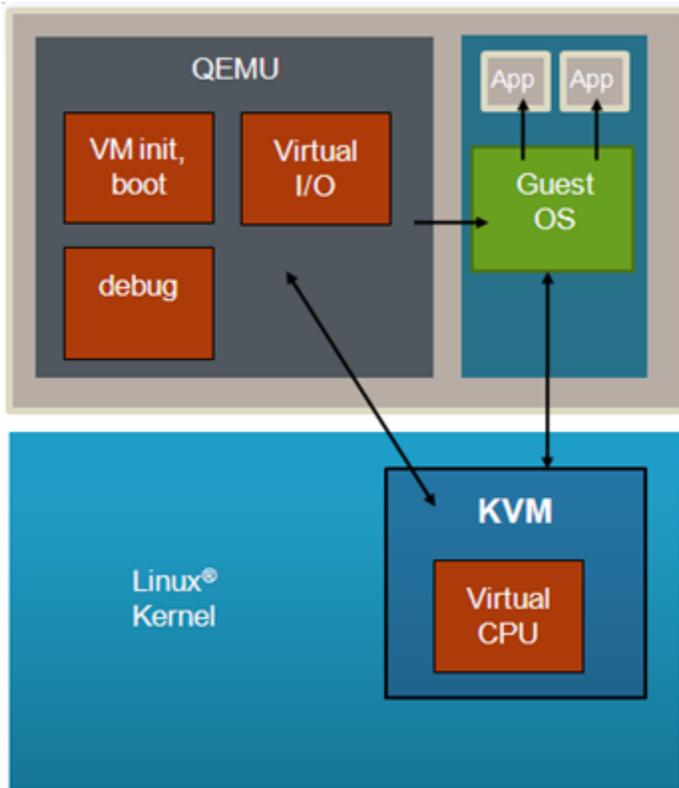


Figure 197.

QEMU is a Linux user-space application that runs on the host Linux instance and is used to start and manage a virtual machine. QEMU provides the following:

- A command line interface that provides extensive customization and configuration of a virtual machine when it is started-- e.g. type of VM, which images to load, and how virtual devices are configured
- Loading of all images needed by the guest-- e.g kernel images, root filesystem, guest device tree
- Setting the initial state of the VM and booting the guest
- Virtual I/O services, such as virtual network interfaces and virtual disks
- Debug services-which provide the capability to debug a guest OS using GDB (similar to a virtual JTAG)

KVM is a device driver in the Linux kernel whose key role in the VM architecture is to provide virtual CPU services. These services involve two aspects:

1. First, KVM provides an API set that QEMU uses to set and get the state of virtual CPUs and run them. For example, QEMU sets the initial values of the CPU's registers before starting the VM.
2. Second, after KVM starts a guest OS, certain operations (such as privileged instructions) performed by the OS cause an exception (or exit) into the host Linux kernel that must be handled and processed by KVM. This handling of traps is referred to as "emulation". These traps are transparent to the guest.

The KVM API is documented in the Linux kernel-- Documentation/virtual/kvm/api.txt.

KVM/QEMU supports virtual I/O which allows sharing of physical I/O devices by multiple VMs. Virtual network and block I/O are supported. See [For More Information](#) on page 1220 for references that provide additional information on virtio.

### 9.1.2.1.5 Device Tree Overview

A device tree is a data structure that describes hardware resources such as CPUs, memory, and I/O devices. An device tree aware OS is passed a device tree which it reads to determine what hardware resources are available.

The host Linux kernel is booted first, typically by u-boot (an open source bootloader). U-boot passes the kernel a **hardware** device tree that lists and describes all system hardware resources available to the host kernel (CPUs/cores, memory, interrupt controller and I/O).

Similarly, when a guest OS is booted in a KVM/QEMU virtual machine, QEMU passes it a **guest** device tree that describes all the hardware resources in the VM. See Figure below.

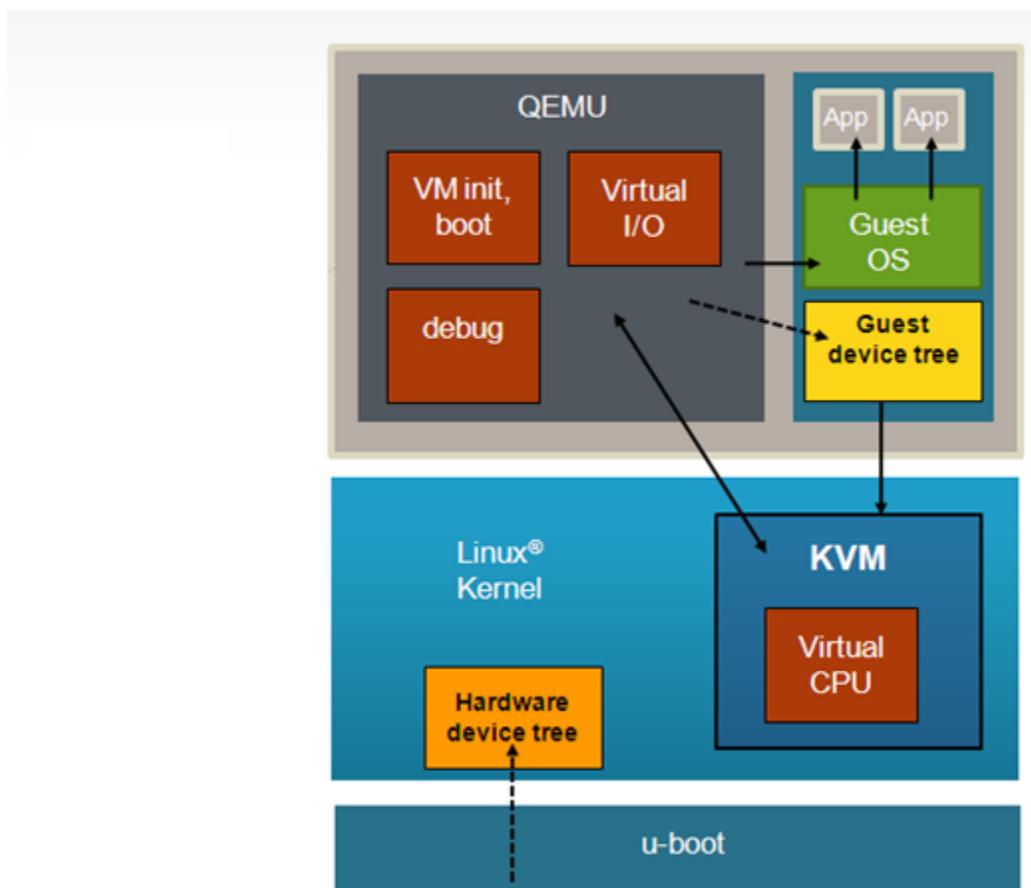


Figure 198.

The guest device tree is generated by QEMU and is used to define the resources a virtual machine will see. The guest device tree defines CPUs, memory, and I/O devices. QEMU places the guest device tree in the virtual machine's memory prior to starting the virtual machine.

### 9.1.2.1.6 References

- [1] QEMU Emulator User Documentation: <http://qemu.weilnetz.de/qemu-doc.html>
- [2] The Linux usage model for device tree data: <https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt>
- [3] Specification for virtio devices: <http://docs.oasis-open.org/virtio/virtio/v1.0/csprd01/virtio-v1.0-csprd01.pdf>

## 9.1.2.1.7 For More Information

### KVM

- KVM website: <http://www.linux-kvm.org>
- ARM VM specification: <http://lwn.net/Articles/589122/>
- Supporting KVM on ARM architecture: <http://lwn.net/Articles/557132/>

### QEMU

- QEMU website: <http://www.qemu.org/>

### Device Trees

- devicetree.org website: <http://devicetree.org>
- DTC, the device tree compiler is available at: <http://git.jdl.com> . DTC also includes a library called libfdt which can be used by software to parse device trees.

Virtio-- a framework for doing virtual I/O using KVM/QEMU

- <http://www.ibm.com/developerworks/linux/library/l-virtio/>
- <http://ozlabs.org/~rusty/virtio-spec/virtio-paper.pdf>
- [http://www.linux-kvm.org/wiki/images/d/dd/KvmForum2007%24kvm\\_pv\\_drv.pdf](http://www.linux-kvm.org/wiki/images/d/dd/KvmForum2007%24kvm_pv_drv.pdf)
- <http://docs.oasis-open.org/virtio/virtio/v1.0/csprd01/virtio-v1.0-csprd01.pdf>

Virtual Networking with QEMU

- <http://wiki.qemu.org/Documentation/Networking>
- <http://www.linux-kvm.org/page/Networking>

## 9.1.2.2 Building QEMU and KVM

### 9.1.2.2.1 Overview

Linux with KVM enabled and QEMU can be built as part of the standard build process used to build the NXP SDK using Yocto.

They can also be built in a standalone manner outside of Yocto.

The build instructions in the sections that follow assume a working understanding of how to use Yocto to build the NXP SDK. Please refer to the Yocto documentation in the SDK.

## 9.1.2.2.2 Building Linux with KVM

### 9.1.2.2.2.1 Overview

KVM is a component in the Linux kernel. KVM is not enabled in the default kernel configuration in the NXP SDK and KVM features must be enabled using the kernel's menuconfig configuration utility prior to building the kernel.

In the sections that follow configuration options are described for both the host and guest Linux kernel. The host and guest kernels can be built separately, but it is possible to build a single Linux kernel image that can be used for both the host and the guest.

The kernel configuration options described below would be the same if building the kernel standalone (outside of Yocto).

The following sections provide high level build information:

- Running menuconfig with Yocto - describes how to configure the kernel under Yocto
- Quick Start - Recommended Configuration Options - in a single step shows all the recommended configuration options to enable to build a kernel with virtual I/O enabled with the same kernel image serving as both host and guest.

The following sections provide more detailed information on each KVM-related configuration option for host and guest:

- Host Kernel: Enabling KVM - describes the configuration options to enable KVM in the host kernel.
- Host Kernel: Enabling Virtual Networking - describes how to enable bridging and tun/tap in the host kernel which enables virtual networking.
- Guest Kernel: Enabling Network and Block Virtual I/O - describes how to enable virtual I/O in the guest kernel.
- Guest kernel: Enabling console - describes how to enable the console for the guest kernel

### 9.1.2.2.2 Running menuconfig with Yocto

The prerequisite and starting point for building the Linux kernel with KVM enabled is performing a standard kernel build with Yocto.

```
$ bitbake virtual/kernel
```

To change the kernel configuration options use the Linux standard menuconfig utility. To invoke menuconfig under Yocto do the following from the Yocto build environment:

```
$ bitbake -c menuconfig virtual/kernel
```

**Note:** Depending on what build steps may have been done previously, it may be necessary to invoke the command 'bitbake -c clean virtual/kernel' prior to running the menuconfig command.

The result will be an xterm that appears that displays the menuconfig screen:

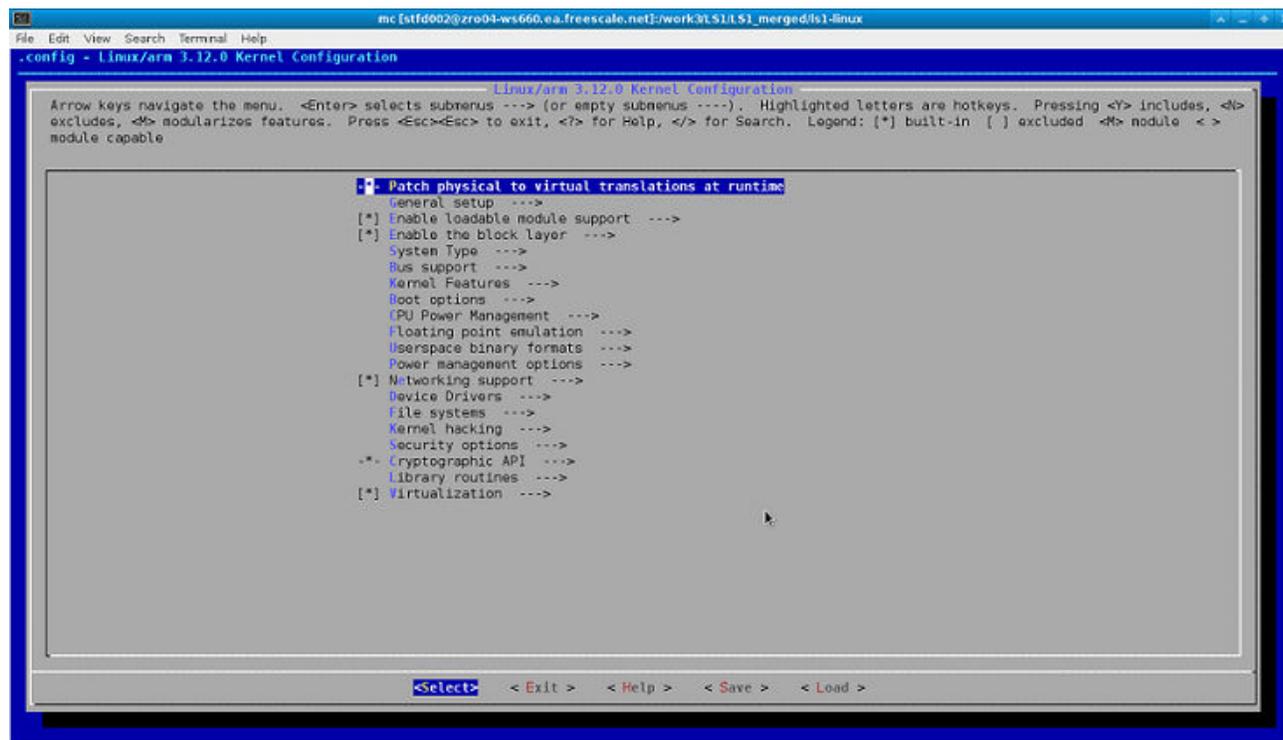


Figure 199.

### 9.1.2.2.3 Quick Start - Recommended Configuration Options

The steps below show all the recommended configuration options to enable to build a kernel with virtual I/O enabled with the same kernel image serving as both host and guest. The sections that follow explain these options in further detail.

Virtualization  
KVM/QEMU

**Note:** The configuration options are enabled by default in the kernel configuration. However, they are listed here for reference.

1. From the main menuconfig window enable virtualization:

```
[*] Virtualization
```

2. In the virtualization menu enable the following options:

```
[*] Kernel-based Virtual Machine (KVM) support
```

3. Enable network bridging

```
Networking support --->
 Networking options --->
 <*> 802.1d Ethernet Bridging
```

4. Enable virtio PCI

```
Device Drivers --->
 Virtio drivers --->
 <*> PCI driver for virtio devices
```

5. Enable virtio for block devices

```
Device Drivers --->
 [*] Block devices --->
 <*> Virtio block driver
```

6. Enable virtio for network devices

```
[*] Network core driver support
 <*> Universal TUN/TAP device driver support
 <*> Virtio network driver
```

7. Enable vhost for virtio network devices

```
[*] Virtualization
 <*> Host kernel accelerator for virtio net
```

8. Enable Huge TLB file support

```
File Systems --->
 Pseudo filesystems --->
 [*] Huge TLB file system support
```

9. Enable guest serial support

```
Device Drivers --->
 Character devices --->
 Serial drivers --->
 <*> ARM AMBA PL011 serial port support
 [*] Support for console on AMBA serial port
```

#### 9.1.2.2.4 Host Kernel: Enabling KVM

This section describes the core, basic options needed to enable KVM in the host kernel. KVM is enabled in the host kernel under the virtualization menu of the main kernel menuconfig window.

```
[*] Virtualization
```

Core KVM support is enabled as follows:

```
[*] Kernel-based Virtual Machine (KVM) support
```

#### 9.1.2.2.5 Host Kernel: Enabling Virtual Networking

[Virtual network interfaces](#) on page 1228 describes how virtual networking can be used to give each VMs a virtual network interface which share physical network interfaces in Linux.

One common approach to configuring virtual networking is for QEMU to use a tun/tap interface bridged to a physical network interface. To do this Ethernet bridging and the kernel's tun/tap features must be enabled in the host kernel:

```
Networking support --->
 Networking options --->
 <*> 802.1d Ethernet Bridging
Device Drivers --->
 [*] Network core driver support
 <*> Universal TUN/TAP device driver support
```

In order to enable vhost-net, the following config option should be enabled:

```
[*] Virtualization
 <*> Host kernel accelerator for virtio net
```

#### 9.1.2.2.6 Guest Kernel: Enabling Network and Block Virtual I/O

Virtio is a framework for doing paravirtualized I/O using QEMU/KVM. In order to support communication between guest and hypervisor virtio uses a PCI transport protocol.

Below the kernel configuration options are shown to enable virtio-pci:

```
Device Drivers --->
 Virtio drivers --->
 <*> PCI driver for virtio devices
```

Below the kernel configuration options are shown to enable virtio drivers in the Linux kernel to support networking I/O and block (disk) I/O.

```
Device Drivers --->
 [*] Network device support --->
 <*> Virtio network driver
Device Drivers --->
 [*] Block devices --->
 <*> Virtio block driver
```

### 9.1.2.2.2.7 Guest kernel: Enabling console

QEMU emulates an AMBA/PL011 console.

Below the kernel configuration options are shown to enable console:

```
Device Drivers --->
 Character devices ---->
 Serial drivers ---->
 <*> ARM AMBA PL011 serial port support
 [*] Support for console on AMBA serial port
```

### 9.1.2.2.3 Building QEMU

QEMU is a standard package in the QorIQ SDK and will be built for the following Yocto build configurations:

- fsl-image-virt
- fsl-image-full

To add QEMU to another Yocto build configuration, edit the conf/local.conf file and add the IMAGE\_INSTALL\_append variable:

```
IMAGE_INSTALL_append = " qemu"
```

After a Yocto build is complete, the target root filesystem will contain the following QEMU components that are required for using KVM/QEMU:

```
/usr/bin/qemu-system-arm # QEMU executable for ARMv7 platforms
/usr/bin/qemu-system-aarch64 # QEMU executable for ARMv8 platforms
```

QEMU can be built as an individual component in the Yocto environment as well, its package name is "qemu".

To clean and rebuild QEMU from the Yocto build environment:

```
$ bitbake -c clean qemu
$ bitbake qemu
```

## 9.1.2.2.4 Creating a host Linux root filesystem

### 9.1.2.2.4.1 Overview

Creating a Linux root filesystem is out of the scope of this document. Please reference the NXP QorIQ SDK for more information on how to create root filesystems with Yocto. This section describes the software components needed on a root filesystem to use KVM/QEMU.

The host root filesystem is the filesystem booted by the host kernel. The host rootfs is distinct from a guest root filesystem which may be needed by certain guest such as Linux.

A host root filesystem capable of running Linux as a guest needs the following components:

- Guest Linux kernel image (Image or zImage). *Note:* Only zImage is supported for ARMv7 platforms.
- QEMU executable
- Guest root filesystem
- Dynamic libraries needed by QEMU (libfdt, libz, glib2.0). These libraries are standard components in a Yocto-created rootfs.

Example host root filesystem layout with the required components to boot a Linux guest (excluding shared libraries):

```
/root/zImage # guest Linux kernel
/root/rootfs.ext2.gz # guest rootfs
```

```
/usr/bin/qemu-system-arm # QEMU for ARMv7 platforms
/usr/bin/qemu-system-aarch64 # QEMU for ARMv8 platforms
```

### 9.1.2.2.4.2 Adding Images to a Root Filesystem with Yocto

If using Yocto, as described in [Building QEMU](#) on page 1224, the root filesystem produced by the build process will contain QEMU and the example guest device trees provided by the SDK.

A feature is also available with Yocto and the SDK to add arbitrary additional images to the root filesystem. This is done using the merge-files component in Yocto.

Any files and directories copied to the "merge" directory (see path below) will be copied to the root filesystem created by Yocto:

```
meta-freescale/recipes-extended/merge-files/merge-files/merge
```

After populating the merge directory with the desired files, clean and rebuild the rootfs. See example below for the fsl-image-core image type:

```
$ bitbake -c install -f merge-files
$ bitbake merge-files
$ bitbake fsl-image-core
```

See the how-to article [Quick-start Steps to Build and Deploy KVM Using Yocto](#) on page 1234 for a more detailed example.

## 9.1.2.3 Using QEMU and KVM

### 9.1.2.3.1 Overview of Using QEMU

QEMU is used to start virtual machines and is built and included in the rootfs created by Yocto. The QEMU application is named **qemu-system-arm** (for 32 bit platforms) or **qemu-system-aarch64** (for 64 bit platforms).

In addition to the QEMU executable itself, the following is a list of the minimum components that must be available on the target system to launch a virtual machine using QEMU:

- The host Linux kernel on the target must be built with virtualization support for KVM enabled as described in [Building Linux with KVM](#) on page 1220.
- A guest OS kernel image (e.g. zImage or Image for Linux)
- A guest root filesystem (If needed by the guest OS. For example, a Linux guest requires a rootfs.)
- Recommended: A working network interface (to interface to the guest's console and the QEMU monitor)

See the article [Quick-start Steps to Run KVM Using Hugelbfs](#) on page 1236 for an example of how to boot a virtual machine with a rootfs created by Yocto.

The QEMU Emulator User Documentation [1] (see [References](#) on page 1219) contains complete documentation for all QEMU command line arguments. The Table below summarizes some of the flags and arguments for basic operation.

**Table 210.**

Argument	Descriptions
-enable-kvm	Specifies that the Linux KVM should be used for the virtual machine's CPUs
<i>Table continues on the next page...</i>	

**Table 210. (continued)**

Argument	Descriptions
-nographic	Disables graphical output-console will be on emulated serial port.
-M <i>machine</i>	Specifies the type of virtual machine. One value is supported: <ul style="list-style-type: none"> <li>• virt</li> </ul>
-smp <i>cpu_count</i>	Specifies the number of CPUs for the virtual machine. The number of virtual CPUs allowed is the same as the value of the CONFIG_NR_CPUS config option in the host Linux kernel. To see this value issue the following command from Linux on the target board: <pre>zcat /proc/config.gz   grep NR_CPUS</pre>
-kernel <i>file</i>	Specifies the guest OS image. The supported image types are in <i>Image</i> format (the generic Linux kernel binary image file) and <i>zImage</i> (a compressed version of the Linux kernel image)
-initrd <i>file</i>	Specifies a root filesystem image
-append <i>cmdline</i>	Use <i>cmdline</i> as the guest OS kernel command line (passed in the bootargs property of the /chosen node in the guest device tree)
-serial <i>dev</i>	Redirects the virtual serial port to the host device <i>dev</i> . QEMU supports many possible host devices. Please refer to the QEMU User Documentation [1] (see <a href="#">References</a> on page 1219) for complete details. Note: if using a tcp device with the <b>server</b> option QEMU will wait for a connection to the device before continuing unless the <b>nowait</b> option is used.
-m <i>megs</i>	Specifies the size of the VM's RAM in megabytes. This option is ignored if using direct mapped memory. .
-mem-path <i>path</i>	Specifies the path to a file from which to allocate memory for the virtual machine. This option should be used to allocate memory from hugetlbfs.
-monitor <i>dev</i>	Redirects the QEMU monitor to the host device <i>dev</i> . QEMU supports many possible host devices. Please refer to the QEMU User Documentation [1] (see <a href="#">References</a> on page 1219) for complete details. Note: if using a tcp device with the <b>server</b> option QEMU will wait for a connection to the device before continuing unless the <b>nowait</b> option is used.
-S	Do not start CPU at startup (you must type 'c' in the monitor). This can be useful if debugging.
-gdb <i>dev</i>	Wait for gdb connection on device <i>dev</i>

*Table continues on the next page...*

Table 210. (continued)

Argument	Descriptions
-drive [args]	Used to create a virtual disk in a virtual machine.
-netdev [args] -device virtio-net-device [args]	The -netdev and -device virtio-net-device arguments specify the network backend and front end for creating virtual network devices in virtual machines.
-cpu model	Select CPU model. Only one model is supported: <ul style="list-style-type: none"> <li>• host</li> </ul>

Below is an example command line a user would run from the host Linux to start virt virtual machine booting a Linux guest:

ARMv7:

```
qemu-system-arm -enable-kvm -m 512 -nographic -cpu host -machine type=virt -kernel /boot/zImage -
serial tcp::4446,server,telnet -initrd /boot/guest.rootfs.ext2.gz -append 'root=/dev/ram0 rw
console=ttyAMA0 rootwait earlyprintk' -monitor stdio
```

ARMv8:

```
qemu-system-aarch64 -enable-kvm -m 512 -nographic -cpu host -machine type=virt -kernel /boot/Image
-serial tcp::4446,server,telnet -initrd /boot/guest.rootfs.ext2.gz -append 'root=/dev/ram0 rw
console=ttyAMA0 rootwait earlyprintk' -monitor stdio
```

### 9.1.2.3.2 Virtual Machine Memory

QEMU allocates and loads images into a VM's memory prior to starting the VM. The amount of memory needed for a virtual machine will be dependent on the workload to be run in the VM. There are two ways to allocate memory:

#### 1. Allocation via hugetlbfs

Hugetlbfs is a Linux mechanism that allows applications to allocate memory backed large physically contiguous regions of memory. QEMU can take advantage of hugetlbfs for allocation of memory for virtual machines, which can provide a significant performance improvement over malloc allocated memory. Hugetlbfs allocated memory provides the flexibility of memory that can be allocated and freed with performance comparable to direct mapped memory.

The 32 bit ARMv7 implementation in Linux supports 2MB sized huge pages. The 64 bit ARMv8 implementation in Linux supports 2MB and 1GB sized huge pages (for 4K granularity page size).

The **-mem-path** argument to QEMU specifies the path to the hugetlbfs mount point where the huge pages should be allocated from.

The **-m** argument to QEMU specifies the amount of memory to allocate to the virtual machine. There are no constraints on the size passed to this argument other than that the amount of memory must fit within the constraints of the system and be enough for the workload in the VM.

See the how-to article [Quick-start Steps to Run KVM Using Hugetlbfs](#) on page 1236 for an example of how to use hugetlbfs.

#### 2. Allocation via malloc

The default for QEMU is to allocate guest memory by the standard malloc facility available to user space applications in Linux. The amount of memory is specified with the -m command line argument. Malloc'ed memory has the flexibility of being allocated and freed by QEMU as needed. However, malloc'ed memory is backed by 4KB physical pages that are

not contiguous and emulation is required by KVM to present a contiguous guest physical memory region to the VM. This approach is discouraged since the emulation can result in a substantial performance penalty for certain workloads.

The guest device tree generated by QEMU will contain a memory node that specifies the total amount of memory.

#### NOTE

A virtual machine's memory is part of the address space of the QEMU process. This means that the amount of memory allocated to a VM is limited by the standard limits that exist for Linux processes. A 32-bit host kernel has a 2GiB virtual address space used for stack, text, and other data, and this limits the amount of memory that can be allocated to a VM.

### 9.1.2.3.3 Virtual network interfaces

QEMU provides a number of options for creating virtual network interfaces in virtual machines. Virtual network interfaces are specified using the QEMU command line and guest software sees them as memory mapped devices.

There are two aspects of virtual network interfaces with QEMU:

1. The network "front-end", which is the network card as seen by the guest. This is specified with the **-device** QEMU argument. The argument to specify a virtio network front end would look like: **-device virtio-net-pci**
2. The network "backend", which connects the network card to some network. Network backend options include user mode networking, a host TAP interface, sockets, or virtual distributed Ethernet. The network backend is specified using the **-netdev** command line argument of QEMU. Note: It is possible to connect two virtual machines using virtual network interfaces. Normally QEMU userspace process emulates I/O accesses from the guest. However, there is an in-kernel implementation: *vhost-net* which puts the data plane emulation code into the kernel.

For example, to use a virtio NIC card with a TAP interface back-end the QEMU command line argument would look like:

```
-netdev tap,id=tap0,script=/root/qemu-ifup -device virtio-net-pci,netdev=tap0
```

The script "/root/qemu-ifup" is a script that QEMU invokes and passes the TAP interface name as an argument. For example, the script could add the TAP interface to an Ethernet bridge.

See the QEMU Users Manual [1] (see [References](#) on page 1219) for detailed information about command line options and the types of network interfaces and backends. For best performance, the virtio front-end is recommended.

For additional information about QEMU networking see the references in [For More Information](#) on page 1220.

For a detailed example, see the how-to article [How to Use Virtual Network Interfaces Using Virtio](#) on page 1238 .

### 9.1.2.3.4 VMs and the Linux Scheduler

Each virtual machine appears to the host Linux as a process with each virtual CPU in the VM implemented as a thread. A VM appears as an instance of QEMU when looking at Linux processes as can be seen in the example below:

```
$ ps -ef
 ○
 ○
root 1333 1 0 Oct01 ttyS0 00:00:00 -sh
root 1336 2 0 08:24 ? 00:00:00 [kworker/u4:2]
root 1372 1333 18 08:27 ttyS0 00:00:17 qemu-system-arm -enable-kvm -m
root 1361 1304 0 08:28 ? 00:00:00 sshd: root@pts/0
root 1363 1361 0 08:28 pts/0 00:00:00 -sh
 ○
 ○
```

CPUs appear as threads. To see thread IDs use the `info cpus` command in the QEMU monitor. Example of a VM with 8 virtual CPUs:

```
(qemu) info cpus
* CPU #0: thread_id=1984
 CPU #1: (halted) thread_id=1985
 CPU #2: (halted) thread_id=1986
 CPU #3: (halted) thread_id=1987
 CPU #4: (halted) thread_id=1988
 CPU #5: (halted) thread_id=1989
 CPU #6: (halted) thread_id=1990
 CPU #7: (halted) thread_id=1991
```

To see the QEMU threads using the `ps` command:

```
root@ls2080ardb:~# ps -eL | grep qemu
1981 1981 ttyS1 00:00:00 qemu-system-aar
1981 1982 ttyS1 00:00:00 qemu-system-aar
1981 1983 ttyS1 00:00:00 qemu-system-aar
1981 1984 ttyS1 00:00:00 qemu-system-aar
1981 1985 ttyS1 00:00:00 qemu-system-aar
1981 1986 ttyS1 00:00:00 qemu-system-aar
1981 1987 ttyS1 00:00:00 qemu-system-aar
1981 1988 ttyS1 00:00:00 qemu-system-aar
1981 1989 ttyS1 00:00:00 qemu-system-aar
1981 1990 ttyS1 00:00:00 qemu-system-aar
1981 1991 ttyS1 00:00:00 qemu-system-aar
```

Being a Linux thread means that standard Linux mechanisms can be used to control aspects of how the threads are scheduled relative to other threads/processes. These mechanisms include:

- process priority
- CPU affinity
- `isolcpus`
- `cgroups`

## 9.1.2.4 Virtual machine reference

### 9.1.2.4.1 VM Overview

In general the architecture of KVM/QEMU is such that few changes should be needed to guest software to run in a VM-- i.e. a full virtualization approach is used, which means that virtual CPUs and virtual I/O devices behave like the physical hardware they are emulating.

However, there are some differences between virtual machines and native hardware that should be considered when targeting an OS to a KVM virtual machine. These differences can be divided into 2 general categories that will be discussed in further detail in this section:

1. Initial state and boot
2. CPUs

### 9.1.2.4.2 Memory Map of Virtual I/O Devices

The virt virtual machine contains a small subset of the devices found on an SoC. The available devices will be represented in the device tree passed to the guest at boot. See the table below for a summary of the virtual I/O devices in the virt VM:

Table 211.

Virtual I/O Devices for virt machine	
virt VM Address, size	Descriptions
0,128MB	space for a flash device (this allows running bootrom code)
0x08000000, 0x20000	Virtual CPU peripherals (including GIC distributor and CPU peripheral space)
0x08000000, 0x10000	Virtual GIC distributor
0x08010000, 0x10000	Virtual GIC CPU interface
0x08020000, 0x10000	Virtul GICv2m controller
0x09000000, 0x10000	Virtual UART
0x09010000, 0x10000	Virtual RTC
0x0a000000..0x0a0001ff	Virtual MMIO
...	Virtual MMIO
0x0c000000, 0x02000000	Virtual platform bus
0x10000000, 0x30000000	virtual PCIE
0x40000000, 30G	guest RAM

### 9.1.2.4.3 Virtual machine state at initialization

#### 9.1.2.4.3.1 Initial State and Boot

When booting the Host, kernel is entered into the HYP mode for ARMv7 respectively EL2 privilege level for ARMv8. After the boot the kernel uses a stub to install KVM and switches back to SVC, respectively EL1. The virtual machine has no virtualization extensions available, so the guest kernel will be entered in SVC mode (ARMv7) respectively EL1 (ARMv8).

In case of a real hardware the boot program will provide some services before giving control to the OS. The necessary steps needed to be done by the bootloader are described in the kernel documentation: *Documentation/arm/Booting/* (ARMv7), *Documentation/arm64/booting.txt* (ARMv8). In case of virtualization, KVM/QEMU makes the necessary actions to put hardware into the initial state (as seen by the guest) and also will take the role of the bootloader and makes the necessary settings. QEMU also installs a very simple bootloader which just set some registers and after that it jumps to the kernel.

It is recommended that a guest OS be minimally device tree aware. The libfdt library (available with the DTC tool) provides a full range of APIs to parse and manipulate device trees and will make the process of adding device tree awareness to an OS straightforward.

#### 9.1.2.4.3.2 Initial State of Virtual CPUs

In a VM with multiple virtual CPUs, CPU #0 is the boot CPU and all other vcpus in the partition are considered secondary. The boot method for the secondary CPUs is PSCI.

The virtual CPU entry conditions comply with the entry conditions specified by *linux/Documentation/arm/Booting* (ARMv7) or *Documentation/ arm64/booting.txt* (ARMv8)

The virtual CPU state is summarized below:

ARMv7:

- R0 = 0

- R1 = machine type number
- R2 = physical address of device tree block (DTB) in system RAM
- MMU off
- data cache off
- CPSR: 0x000001d3 (SVC mode, asynchronous abort, IRQ and FIQ masked)

ARMv8:

- x0 = physical address of device tree blob (dtb) in system RAM.
- x1 = 0
- x2 = 0
- x3 = 0
- MMU off

## 9.1.2.4.4 Virtual CPUs

### 9.1.2.4.4.1 Virtual CPU Specification

Software running in a virtual machine sees a virtual CPU that emulates an ARMv7/ARMv8 core without virtualization extensions.

The virtual CPU type will match that of the host hardware platform.

### 9.1.2.4.4.2 Time in the Virtual CPU

ARM architecture has an optional extension, the generic timers, which provides:

- a counter (*physical counter*) that measures passing of time in real time
- a timer (*physical timer*) for each CPU. The timer is programmed to raise an interrupt to the CPU after a certain amount of time has passed.

The generic timers include virtualization support by introducing:

- a new counter, the *virtual counter*
- a new timer, the *virtual timer*.

This allows the virtual machine to have direct access to reading (virtual) counters and programming (virtual) timers without trapping.

KVM uses the physical timers in the host, the virtual machine access to the physical timers being disabled.

The virtual machine accesses the virtual timer and can, in this way, directly access the timer hardware without trapping to the hypervisor. However, the virtual timers do not raise virtual interrupts, but hardware interrupts which trap to the hypervisor. KVM injects a corresponding virtual interrupt into the VM when it detects that the virtual timer expired.

### 9.1.2.4.5 VGIC

The ARM Generic Interrupt Controller (GIC) provides hardware support for virtualization. The guest is able to mask, acknowledge and EOI interrupts without trapping to the hypervisor. However, there is a central part of the GIC called distributor which is responsible for interrupt prioritization and distribution to each CPU which does not provide virtualization extensions and for this part KVM provides an in-kernel emulation. Also, all the physical interrupts cannot be directly received by the guest. Instead, the KVM will program a virtual interrupt which will be raised in the guest. But, with the virtualization support in the GIC controller, when the guest is ACK-ing and EOI-ing the virtual interrupt, there is no need to trap into KVM.

## 9.1.2.5 Debugging virtual machines

### 9.1.2.5.1 QEMU Monitor

When starting QEMU, a monitor shell is available that can be used to control and see the state of VM. By default this monitor is started in the Linux shell where QEMU is invoked.

See example below of the output when starting QEMU. The user can interact with the monitor at the (qemu) prompt.

```
QEMU 2.4.0 monitor - type 'help' for more information
(qemu) QEMU waiting for connection on: disconnected:telnet::4446,server
```

The monitor can also be exposed over a network port by using the `-monitor dev` command line option. See [Overview of Using QEMU](#) on page 1225 and the QEMU user's manual [1] (see [References](#) on page 1219).

Refer to the QEMU user's manual [1] for a complete listing of the monitor commands available. Below is a list of some useful commands supported in the NXP SDK implementation of QEMU:

- **help** - lists all the available commands with usage information
- **info cpus** - displays the state and thread ID of all virtual CPUs
- **info registers** - displays the contents of the default vcpu's registers
- **cpu cpu\_number** - sets the default vcpu number
- **system\_reset** - resets the VM
- **x/fmt addr** -- virtual memory dump starting at 'addr'
- **xp/fmt addr** -- physical memory dump starting at 'addr'

### 9.1.2.5.2 QEMU GDB Stub

QEMU supports debugging of a VM using gdb. QEMU contains a gdb stub that can be attached to from a host system and allows standard source level debugging capabilities to examine the state of the VM and do run control.

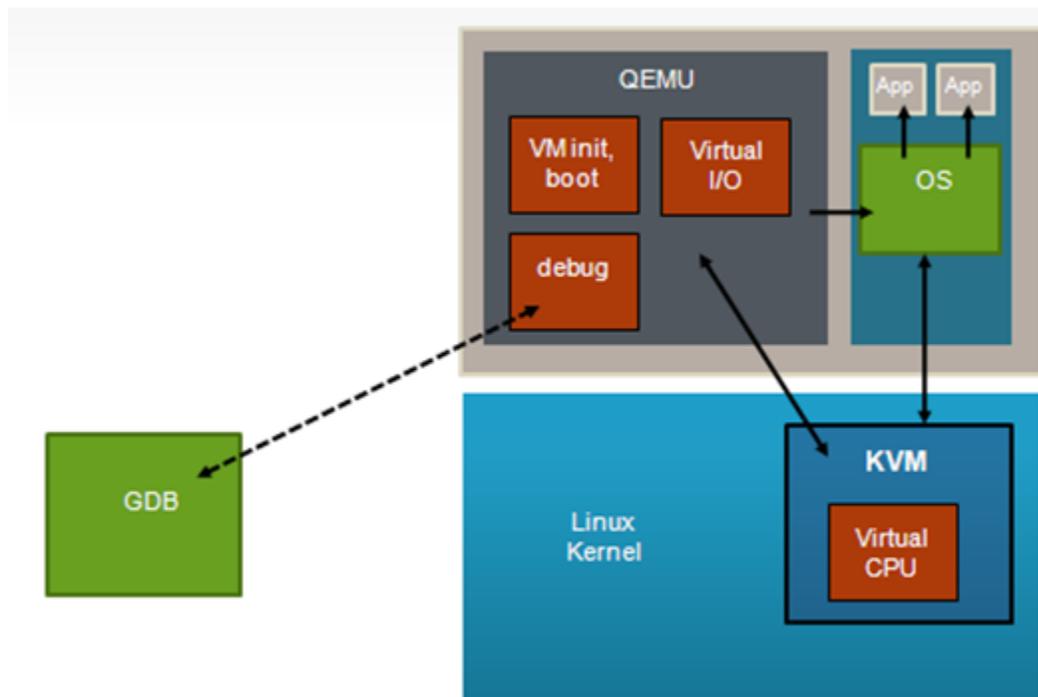


Figure 200.

To use the gdb stub, start QEMU with the `-gdb dev` option where `dev` specifies the type of connection to be used. See the QEMU user's manual [1] (see [References](#) on page 1219) for details.

One useful option when debugging is the `-S` argument to QEMU which causes QEMU to wait to start the first instruction of the guest until told to start using the monitor (**continue** command).

In the example below the `tcp` device type is used. A gdb stub will be active on port 4445 of the host Linux kernel when starting QEMU:

```
$ qemu-system-arm -enable-kvm -m 512 -mem-path /var/lib/lugetlbfs/pagesize-2MB -nographic -cpu
host -machine type=virt -kernel /boot/zImage -serial tcp::4446,server,telnet -initrd /boot/fsl-
image-core-ls1021atwr.ext2.gz -append 'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' -
monitor stdio -gdb tcp::4445
```

```
QEMU 2.4.0 monitor - type 'help' for more information
(qemu) QEMU waiting for connection on: telnet:0.0.0.0:4446,server
```

After the guest has been started normally, gdb can be used to connect to the VM (in this example the host kernel has an ip address of 192.168.3.30):

```
(gdb) target remote 192.168.3.30:4445
Remote debugging using 192.168.3.30:4445
0x80024f44 in ?? ()
```

Debugging with gdb can then proceed normally:

```
(gdb) p/x $r15
$2 = 0x80024f44
(gdb)
```

## 9.1.2.6 KVM/QEMU How-to's

### 9.1.2.6.1 Quick-start Steps to Build and Deploy KVM Using Yocto

The following steps show how to build host and guest root filesystems, Linux kernel, and QEMU in the Yocto environment.

There are two possibilities to deploy KVM using Yocto:

- Using the preconfigured **fsl-image-virt** Yocto image as the base for host root filesystem. This image type will generate a host root filesystem which contains: QEMU, guest root filesystem and guest kernel image.
- Using other Yocto images as the base for host rootfilesystem . If the user wants to use other Yocto images to enable KVM there will be additional steps needed in order to add all the needed pieces into the host root filesystem: QEMU, guest root filesystem, guest kernel image.

#### 9.1.2.6.1.1 Deploy KVM using fsl-image-virt Image

The host roots is based on the **fsl-image-virt** image type and the guest rootfs is based on **fsl-image-core**.

The steps outlined below assume that the Yocto build environment is configured so that the **bitbake** command can be run.

1. Build the base host root filesystem using the **fsl-image-virt** image type.

```
$ bitbake fsl-image-virt
```

2. Build a kernel with KVM enabled. In this case the same kernel image will be used for both host and guest.

Configure the Linux kernel to enable KVM-related features (if not enabled by default in the kernel config)

```
$ bitbake -c menuconfig virtual/kernel
```

Follow the steps described in section [Quick Start - Recommended Configuration Options](#) on page 1221 to enable KVM in the Linux kernel.

Then rebuild the kernel based on the new configuration options:

```
$ bitbake virtual/kernel
```

3. Re-build the fsl-image-virt image (if nothing was done at step 2, this step may be skipped)

```
bitbake fsl-image-virt
```

4. Create the kernel.itb file (if necessary)

U-boot may use both ulmage or FIT image format to load the kernel image. For ARMv7 platforms an ulmage is used and for these platforms this step is not needed. For ARMv8, a FIT image is used and the FIT image must be generated. The following steps need to be taken:

- Update the *fsl-image-kernelitb* to include the rootfs generated by the *fsl-image-virt* (by default the recipee would use the rootfs generated by *fsl-image-core*). In order to change the rootfs type in the generated itb file there are two possible solutions:
  - Redefine the *ROOTFS\_IMAGE* variable in *meta-freescale/recipes-fsl/images/fsl-image-kernelitb.bb* directly:

```
-ROOTFS_IMAGE ?= "fsl-image-core"
+ROOTFS_IMAGE ?= "fsl-image-virt"
```

- Add the following line to *build\_<machine\_release>/conf/local.conf*:

```
ROOTFS_IMAGE = "fsl-image-virt"
```

- Generate the kernel itb:

```
bitbake fsl-image-kernelitb
```

The resulting host rootfs will contain:

- Linux kernel image. Currently for ARMv7 platforms the zImage file will be included and for ARMv8 platforms the Image file. The kernel image will be located in */boot*
- Guest root filesystem: based on **fsl-image-core**. The guest root file system will be located in */boot*
- QEMU

For steps to run QEMU see the article [Quick-start Steps to Run KVM Using Hugetlbfs](#) on page 1236.

### 9.1.2.6.1.2 Deploy KVM using fsl-image-core Image

The host rootfs is based on the **fsl-image-core** image type and the guest rootfs is based on **fsl-image-minimal**.

The steps outlined below assume that the Yocto build environment is configured so that the **bitbake** command can be run.

1. Build the base host root filesystem using the **fsl-image-core** image type.

```
$ bitbake fsl-image-core
```

2. Build a host kernel with KVM-enabled (if not enabled by default in the kernel config)

Configure the Linux kernel to enable KVM-related features.

```
$ bitbake -c menuconfig virtual/kernel
```

Follow the steps described in section [Quick Start - Recommended Configuration Options](#) on page 1221 to enable KVM in the Linux kernel.

Then rebuild the kernel based on the new configuration options:

```
$ bitbake virtual/kernel
```

3. Add QEMU to the packages built by **fsl-image-core**.

Edit the `conf/local.conf` file and append the following line which adds the QEMU package:

```
IMAGE_INSTALL_append = " qemu"
```

4. Build a guest root filesystem and add it to the host rootfs.

**A.** Build the guest rootfs using the **fsl-image-minimal** image type. This creates a small rootfs sufficient for booting a Linux guest:

```
$ bitbake fsl-image-minimal
```

This command results in the minimal rootfs being created in `tmp/deploy/images`. In this example the minimal rootfs built is named: `fsl-image-minimal.rootfs.ext2.gz`

**B.** Use the `merge-files` feature of Yocto (see [Adding Images to a Root Filesystem with Yocto](#) on page 1225) to copy the guest rootfs to the host rootfs.

```
$ mkdir -p meta-freescale/recipes-extended/merge-files/merge-files/merge/home/root
```

```
$ cp tmp/deploy/images/machine/fsl-image-minimal.rootfs.ext2.gz meta-freescale/recipes-extended/merge-files/merge-files/merge/home/root/guest.rootfs.ext2.gz

$ bitbake -c install -f merge-files

$ bitbake merge-files
```

5. Re-build the fsl-image-core image.

```
bitbake fsl-image-core
```

6. Create the kernel.itb file (if necessary)

U-boot may use both ulmage or FIT image format to load the kernel image. For ARMv7 platforms an ulmage is used and for those platforms this step is not needed. For ARMv8, a FIT image is used and the FIT image must be generated.

```
bitbake fsl-image-kernelitb
```

The resulting host rootfs will contain:

- Linux kernel image
- Guest root filesystem
- QEMU, including the example guest device trees

For steps to run QEMU see the article [Quick-start Steps to Run KVM Using Hugetlbfs](#) on page 1236.

## 9.1.2.6.2 Quick-start Steps to Run KVM Using Hugetlbfs

The pre-requisite to this example is completing the steps in the article [Quick-start Steps to Build and Deploy KVM Using Yocto](#) on page 1234.

This example assumes that the host Linux kernel is booted, has a working network interface, and the following images are present in the host root filesystem:

- Guest kernel image (*/boot/zImage* or */boot/Image*)
- Guest root filesystem (*/boot/guest.rootfs.ext2.gz*)
- QEMU (*/usr/bin/qemu-system-arm* or */usr/bin/qemu-system-aarch64*)

There are a number of mechanisms for allocating huge pages and making them accessible via a mount point. Refer to the SDK documentation for details. This example assume allocating pages using the hugeadm command. Create a 512MB pool of 2MB huge pages, which can be used by QEMU for allocating VM memory:

```
$ hugeadm --pool-pages-min 2M:256

hugeadm --pool-list
Size Minimum Current Maximum Default
2097152 256 256 256 256 *
```

Create a mount point to access the huge pages:

```
$ hugeadm --create-mounts

$ ls -l /var/lib/hugetlbfs/
pagesize-2MB
```

Start QEMU specifying the 2MB huge page pool as the file from which to allocate memory. In this example 512MB of memory is allocated to the VM:

## 32 bit ARMv7:

```
qemu-system-arm -enable-kvm -m 512 -mem-path /var/lib/lugetlbfs/pagesize-2MB -nographic -cpu host -
machine type=virt -kernel /boot/zImage -serial tcp::4446,server,telnet -initrd /boot/
guest.rootfs.ext2.gz -append 'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' -monitor stdio
```

```
QEMU waiting for connection on: telnet::0.0.0.04446,server
```

## 64bit ARMv8:

```
qemu-system-aarch64 -enable-kvm -m 512 -mem-path /var/lib/lugetlbfs/pagesize-2MB -nographic -cpu
host -machine type=virt -kernel /boot/Image -serial tcp::4446,server,telnet -initrd /boot/
guest.rootfs.ext2.gz -append 'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' -monitor
stdio
```

```
QEMU waiting for connection on: telnet::0.0.0.04446,server
```

## Explanation of the command line options:

- **-enable-kvm** : specifies that KVM should be used
- **-m 512** : the amount of memory for the VM
- **-mem-path /var/lib/lugetlbfs/pagesize-2MB** : allocates from hugetlbfs based memory
- **-nographic** : don't instantiate a graphics card, this is the only option supported for the SDK
- **-cpu host** : the type of the CPU. In this case it is the same as the host CPU
- **-machine type=virt** : the type of virtual machine
- **-kernel /boot/zImage** : name of guest Linux kernel
- **-initrd ./boot/guest.rootfs.ext2.gz** : name of guest roofs
- **-append "root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk"** : guest Linux bootargs
- **-serial tcp::4446,server,telne** : provide an emulated serial port (telnet server) on port 4444 on the host Linux system. Default behavior will be for QEMU to wait until the user connects to this port before booting the VM.
- **-monitor stdio** : start QEMU monitor

At this point QEMU is waiting for a telnet connection to the virtual machine's console (port 4446 of the target board) prior to starting the virtual machine.

Connect to QEMU via telnet to start the virtual machine booting. In this example the target board has IP address 192.168.3.30.

```
$ telnet 192.168.3.30 4446
[0.000000] Booting Linux on physical CPU 0x0
[0.000000] Initializing cgroup subsys cpu
[0.000000] Linux version 4.1.8-rt8+gf42311c (gcc version 4.9.3 20150311 (prerelease) (Linaro
GCC 4.9-2015.03)) #1 SMP Tue Apr 26 15:40:46 EEST 2016
[0.000000] CPU: AArch64 Processor [411fd071] revision 1
[0.000000] Detected PIPT I-cache on CPU0
[0.000000] alternatives: enabling workaround for ARM erratum 832075
[0.000000] alternatives: enabling workaround for ARM erratum 834220
[0.000000] efi: Getting EFI parameters from FDT:
.....
Starting system log daemon...0
Starting kernel log daemon...0
Starting internet superserver: xinetd.

QorIQ SDK (FSL Reference Distro) 2.0 ls2080ardb /dev/ttyAMA0
```

```
ls2080ardb login:
```

### 9.1.2.6.3 How to Use Virtual Network Interfaces Using Virtio

As discussed in [Virtual network interfaces](#) on page 1228, there are two aspects of virtual network interfaces-- 1) the "front end" (the device as seen by the guest OS) and 2) the "backend" (the means by the virtual device is connected to the network).

This example uses a "virtio" model NIC card and a tap network backend. The virtual network interface is bridged via a TAP interface to the physical network. The guest OS is Linux.

When starting QEMU we will add the following arguments to create the virtual network interface:

```
-netdev tap,id=tap0,script=/home/root/qemu-ifup,downscript=no,ifname="tap0" -device virtio-net-pci,netdev=tap0
```

Perform the following steps:

1. Enable virtio networking in the host and guest Linux kernels (see [Host Kernel: Enabling Virtual Networking](#) on page 1223 and [Guest Kernel: Enabling Network and Block Virtual I/O](#) on page 1223).
2. On the host Linux create a bridge to the physical network interface to be used by the virtual network interface in the virtual machine using the **brctl** command. In this example the physical interface being used is eth2:

```
brctl addbr br0
ifconfig br0 192.168.3.30 netmask 255.255.248.0
ifconfig eth2 0.0.0.0
brctl addif br0 eth2
```

3. Create a `qemu-ifup` script on the host Linux system. For the TAP backend type, when QEMU creates the virtual network interface it invokes a user-created script that allows customization of how the TAP interface is to be handled. The name of the TAP interface created by QEMU is passed as an argument. In this example we will bridge the the TAP interface to the bridge created in step #2. See the example `qemu-ifup` script below:

```
#!/bin/sh
TAP interface will be passed in $1
bridge=br0
guest_device=$1
ifconfig $guest_device 0.0.0.0 up
brctl addif $bridge $guest_device
```

4. When starting QEMU specify that the network device type is "virtio" and specify the path to the script created in step #3:

```
qemu-system-aarch64 -smp 8 -enable-kvm -m 4096 -nographic -cpu host -machine type=virt -kernel /boot/Image -serial tcp::4446,server,telnet -initrd /boot/fsl-image-core-ls2080ardb.ext2.gz -append 'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' -monitor stdio -netdev tap,id=tap0,script=qemu-ifup,downscript=no,ifname="tap0" -device virtio-net-pci,netdev=tap0
```

5. In the guest OS the virtual network interface will appear and can be brought up and assigned an IP address in the normal way. In the example below (the commands are run from the guest command shell) the virtio interface is eth0.

```
$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default
 link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
 inet 127.0.0.1/8 scope host lo
 valid_lft forever preferred_lft forever
```

```

inet6 ::1/128 scope host
 valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
 link/ether 52:54:00:12:34:56 brd ff:ff:ff:ff:ff:ff
3: sit0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN group default
 link/sit 0.0.0.0 brd 0.0.0.0

root@ls2080ardb:~# ethtool -i eth0
driver: virtio_net
version: 1.0.0
firmware-version:
expansion-rom-version:
bus-info: 0000:00:01.0
supports-statistics: no
supports-test: no
supports-eeprom-access: no
supports-register-dump: no
supports-priv-flags: no

$ ifconfig eth0 192.168.3.31 netmask 255.255.248.0

```

#### 9.1.2.6.4 How to use vhost-net with virtio

vhost-net is a character device that can be used to reduce the number of system calls involved in virtio networking. vhost-net moves network packets between the guest and the host system using the Linux kernel, bypassing QEMU.

In order to use vhost-net perform the following steps:

1. Enable virtio networking and vhost-net in the host and guest Linux kernels (see [Host Kernel: Enabling Virtual Networking](#) on page 1223 and [Guest Kernel: Enabling Network and Block Virtual I/O](#) on page 1223).
2. On the host Linux create a bridge to the physical network interface to be used by the virtual network interface in the virtual machine using the **brctl** command. In this example the physical interface being used is eth2:

```

brctl addbr br0
ifconfig br0 192.168.3.30 netmask 255.255.248.0
ifconfig eth2 0.0.0.0
brctl addif br0 eth2

```

3. Create a qemu-ifup script on the host Linux system. For the TAP backend type, when QEMU creates the virtual network interface it invokes a user-created script that allows customization of how the TAP interface is to be handled. The name of the TAP interface created by QEMU is passed as an argument. In this example we will bridge the the TAP interface to the bridge created in step #2. See the example qemu-ifup script below:

```

#!/bin/sh
TAP interface will be passed in $1
bridge=br0
guest_device=$1
ifconfig $guest_device 0.0.0.0 up
brctl addif $bridge $guest_device

```

4. When starting QEMU specify that the network device type is "virtio" and that vhost-net (**vhost=on** parameter) is used:

```

qemu-system-aarch64 -smp 8 -enable-kvm -m 8192 -mem-path /var/lib/lugetlbf/pagesize-1GB -
nographic -cpu host -machine type=virt -kernel /boot/Image -serial tcp::4446,server,telnet -
initrd /boot/fsl-image-core-ls2080ardb.ext2.gz -append 'root=/dev/ram0 rw console=ttyAMA0
rootwait earlyprintk ramdisk_size=307200' -monitor stdio -netdev tap,id=tap0,script=qemu-
ifup,downscript=no,ifname="tap0",vhost=on -device virtio-net-pci,netdev=tap0

```

5. In the guest the virtual interface will come up as described in [How to Use Virtual Network Interfaces Using Virtio](#) on page 1238. In the Host kernel the vhost thread can be seen consuming CPU:

```
 2078 root 20 0 0 0 0 R 93.7 0.0 0:07.09
vhost-2066
 2066 root 20 0 9192660 511632 7532 S 82.0 3.3 0:12.70 qemu-system-
aar
 2091 root 20 0 159636 1092 960 S 68.0 0.0 0:05.50 iperf
```

### 9.1.2.6.5 Debugging: How to Examine Initial Virtual Machine State with QEMU

It can be helpful when debugging to examine the state of the virtual machine prior to executing the first instruction of the guest OS.

To do this, start QEMU with the -S option.

Example:

```
qemu-system-aarch64 -enable-kvm -m 512 -nographic -cpu host -machine type=virt -kernel /boot/
Image -serial tcp::4446,server,telnet -initrd /boot/fsl-image-core-ls2080ar.db.ext2.gz -append
'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' -monitor stdio -S
```

The console was started with the "-serial tcp::4446,server,telnet" option so QEMU waits for a connection prior to starting initialization. Use telnet to connect to port 4446 of the target.

At this point QEMU initializes the VM, but does not execute the entry point to the guest OS. The monitor prompt can now be used to examine initial state:

```
QEMU 2.4.0 monitor - type 'help' for more information
(qemu) QEMU waiting for connection on: telnet:0.0.0.0:4446,server
(qemu)
```

To see where boot images are loaded and placed by QEMU use the **info roms** command:

```
(qemu) info roms
addr=0000000040000000 size=0x000028 mem=ram name="bootloader"
addr=0000000040080000 size=0xb9a800 mem=ram name="/boot/Image"
addr=0000000048000000 size=0x1517eef mem=ram name="/boot/fsl-image-core-ls2080ar.db.ext2.gz"
addr=0000000049600000 size=0x010000 mem=ram name="dtb"
/rom@etc/acpi/tables size=0x200000 name="etc/acpi/tables"
/rom@etc/table-loader size=0x000880 name="etc/table-loader"
/rom@etc/acpi/rsdp size=0x000024 name="etc/acpi/rsdp"
```

A trivial bootloader is loaded at the start of guest memory at 0x40000000

The kernel image (zImage) is loaded at 0x40080000. The ramdisk is loaded at 0x48000000.

To examine the initial state of registers use the **info registers** command:

```
(qemu) info registers
PC=0000000040000000 SP=0000000000000000
X00=0000000000000000 X01=0000000000000000 X02=0000000000000000 X03=0000000000000000
X04=0000000000000000 X05=0000000000000000 X06=0000000000000000 X07=0000000000000000
X08=0000000000000000 X09=0000000000000000 X10=0000000000000000 X11=0000000000000000
X12=0000000000000000 X13=0000000000000000 X14=0000000000000000 X15=0000000000000000
X16=0000000000000000 X17=0000000000000000 X18=0000000000000000 X19=0000000000000000
X20=0000000000000000 X21=0000000000000000 X22=0000000000000000 X23=0000000000000000
```

```

X24=0000000000000000 X25=0000000000000000 X26=0000000000000000 X27=0000000000000000
X28=0000000000000000 X29=0000000000000000 X30=0000000000000000 PSTATE=400003c5 (flags -Z--)

q00=0000000000000000:0000000000000000 q01=0000000000000000:0000000000000000
q02=0000000000000000:0000000000000000 q03=0000000000000000:0000000000000000
q04=0000000000000000:0000000000000000 q05=0000000000000000:0000000000000000
q06=0000000000000000:0000000000000000 q07=0000000000000000:0000000000000000
q08=0000000000000000:0000000000000000 q09=0000000000000000:0000000000000000
q10=0000000000000000:0000000000000000 q11=0000000000000000:0000000000000000
q12=0000000000000000:0000000000000000 q13=0000000000000000:0000000000000000
q14=0000000000000000:0000000000000000 q15=0000000000000000:0000000000000000
q16=0000000000000000:0000000000000000 q17=0000000000000000:0000000000000000
q18=0000000000000000:0000000000000000 q19=0000000000000000:0000000000000000
q20=0000000000000000:0000000000000000 q21=0000000000000000:0000000000000000
q22=0000000000000000:0000000000000000 q23=0000000000000000:0000000000000000
q24=0000000000000000:0000000000000000 q25=0000000000000000:0000000000000000
q26=0000000000000000:0000000000000000 q27=0000000000000000:0000000000000000
q28=0000000000000000:0000000000000000 q29=0000000000000000:0000000000000000
q30=0000000000000000:0000000000000000 q31=0000000000000000:0000000000000000
FPCR: 00000000 FPSR: 00000000

```

The program counter is set to 0x40000000 which is the effective address of the entry point of the kernel.

### 9.1.2.6.6 Debugging: How to Profile Virtualization Overhead with KVM

Running software in a virtual machine can cause additional overhead that affects performance. The virtualization overhead is directly related to the number of times the hypervisor (KVM) is invoked to handle exception conditions that may occur in the virtual machine. These exception handling events are referred to as 'exits', because guest context is exited.

Examples of exits include things such the guest executing a privileged instruction, access a privileged CPU register, accessing a virtual I/O device, or a hardware interrupt such as a decremter interrupt.

The type and number of exits that occur is workload dependent.

KVM implements a mechanism in which different events are logged. These events are actually tracepoint events, and perf nicely integrates with them. You have to compile the host kernel with the following options:

```

Kernel hacking --->
 [*] Tracers --->
 [*] Trace process context switches and events

```

#### Counting Events

A count of a subset of KVM events that occur can be seen under debugfs. To see this first mount debugfs:

```
mount -t debugfs none /sys/kernel/debug
```

The statistics can be seen using perf tool:

```

perf stat -e "kvm:*" -p 1395
^C
Performance counter stats for process id '1395':

 5678 kvm:kvm_entry
 5678 kvm:kvm_exit
 3121 kvm:kvm_guest_fault
 2278 kvm:kvm_irq_line
 0 kvm:kvm_mmio_emulate

```

```
0 kvm:kvm_emulate_cp15_imp
2438 kvm:kvm_wfi
0 kvm:kvm_unmap_hva
2 kvm:kvm_unmap_hva_range
0 kvm:kvm_set_spte_hva
0 kvm:kvm_hvc
3119 kvm:kvm_userspace_exit
0 kvm:kvm_set_irq
0 kvm:kvm_ack_irq
4068 kvm:kvm_mmio
0 kvm:kvm_fpu
0 kvm:kvm_age_page
```

59.316709040 seconds time elapsed

## Tracing events

Detailed traced can be generated using ftrace:

```
[enable ftrace in kernel: events and system calls]
$echo 1 > /sys/kernel/debug/tracing/events/kvm/enable
$cat /sys/kernel/debug/tracing/trace_pipe
```

```
qemu-system-arm-1366 [000] ... 716.115891: kvm_guest_fault: ipa 0x9000000, hsr 0x93430046,
hxfar 0xa084c030, pc 0x80266a9c
qemu-system-arm-1366 [000] ... 716.115892: kvm_mmio: mmio write len 2 gpa 0x9000030 val 0xf01
qemu-system-arm-1366 [000] ... 716.115895: kvm_userspace_exit: reason KVM_EXIT_MMIO (6)
qemu-system-arm-1366 [000] d... 716.115907: kvm_entry: PC: 0x80266aa0
qemu-system-arm-1366 [000] d... 716.116234: kvm_exit: PC: 0x800cf508
qemu-system-arm-1366 [000] d... 716.118274: kvm_entry: PC: 0x800cf508
qemu-system-arm-1366 [000] d... 716.118704: kvm_exit: PC: 0x0000981c
qemu-system-arm-1366 [000] d... 716.120737: kvm_entry: PC: 0x0000981c
qemu-system-arm-1366 [000] d... 716.121159: kvm_exit: PC: 0x800bb104
qemu-system-arm-1366 [000] d... 716.123197: kvm_entry: PC: 0x800bb104
qemu-system-arm-1366 [000] d... 716.123620: kvm_exit: PC: 0x8009cae0
qemu-system-arm-1366 [000] d... 716.125696: kvm_entry: PC: 0x8009cae0
qemu-system-arm-1366 [000] d... 716.126091: kvm_exit: PC: 0x800c90f4
qemu-system-arm-1366 [000] d... 716.128130: kvm_entry: PC: 0x800c90f4
qemu-system-arm-1366 [000] d... 716.128561: kvm_exit: PC: 0x801f37f4
qemu-system-arm-1366 [000] d... 716.130594: kvm_entry: PC: 0x801f37f4
qemu-system-arm-1366 [000] d... 716.130623: kvm_exit: PC: 0x8020576c
qemu-system-arm-1366 [000] d... 716.130635: kvm_entry: PC: 0x8020576c
qemu-system-arm-1366 [000] d... 716.131018: kvm_exit: PC: 0x43014750
qemu-system-arm-1366 [000] d... 716.133053: kvm_entry: PC: 0x43014750
qemu-system-arm-1366 [000] d... 716.133478: kvm_exit: PC: 0x80205778
qemu-system-arm-1366 [000] d... 716.135555: kvm_entry: PC: 0x80205778
```

## 9.2 Libvirt Users Guide

### 9.2.1 Introduction to libvirt

#### 9.2.1.1 Overview

This document is a guide and tutorial to using libvirt on NXP SoCs.

Libvirt is an open source toolkit that enables the management of Linux-based virtualization technologies such as KVM/QEMU virtual machines and Linux containers.

The goal of the libvirt project (see <http://libvirt.org>) is to provide a stable, standard, hypervisor-agnostic interface for managing virtualization "domains" such as virtual machines and containers. Domains can be remote and libvirt provides full security for managing remote domains over a network. Libvirt is a layer intended to be used as a building block for higher level management tools and applications.

Libvirt provides:

- An interface to remotely manage the lifecycle of virtualization domains-- provisioning, start/stop, monitoring
- Support for a variety of hypervisors-- KVM/QEMU and Linux Containers are supported in the NXP SDK
- **libvirtd** -- a Linux daemon that runs on a target node/system and allows a libvirt management tool to manage virtualization domains on the node
- **virsh** -- a basic command shell for managing libvirt domains
- A standard XML format for defining domains

### 9.2.1.2 For Further Information

Libvirt is an open source project and a great deal of technical and usage information is available on the [libvirt.org](http://libvirt.org) website:

- <http://libvirt.org/index.html>

Additional references:

- Architecture: <http://libvirt.org/intro.html>
- Deployment: <http://libvirt.org/deployment.html>
- XML Format: <http://libvirt.org/format.html>
- virsh command reference: <http://linux.die.net/man/1/virsh>
- The libvirt wiki has user generated content: [http://wiki.libvirt.org/page/Main\\_Page](http://wiki.libvirt.org/page/Main_Page)

#### Mailing Lists

There are three libvirt mailing lists available which can be subscribed to. Archives of the lists are also available.

<https://www.redhat.com/archives/libvir-list>

<https://www.redhat.com/archives/libvirt-users>

<https://www.redhat.com/archives/libvirt-announce>

### 9.2.1.3 Libvirt in the NXP QorIQ SDK -- Supported Features

The libvirt packages provides a huge number of capabilities and features. This section describes the features tested in the NXP QorIQ SDK release.

The SDK supports QEMU/KVM and LXC and thus supports URIs for QEMU and LXC:

- `qemu:///`
- `lxc:///`

The following virsh commands are supported:

- Domain Management
  - *attach-device* - Attach a device from an XML file. To use --config option, then it will effect after the active domain restarted.
  - *attach-disk* - attach disk device
  - *attach-interface* - attach network interface

- *autostart* - configure a domain to be automatically started at boot
- *blkdeviotune* - set or query a block device I/O tuning parameters.
- *console* - connect to the console of a domain
- *cpu-stats* - show domain cpu statistics (need mount /cgroup/cpuacct).
- *create* - creates a transient domain from an XML file and starts it
- *define* - define a new persistent domain from an XML file
- *desc* - show or modify the description and title of a domain
- *destroy* - For persistent domains, stops the domain. For transient domains, the domain is destroyed. This command does not gracefully stop the domain.
- *detach-device* - detach a device from an XML file. To use --config option, then it will effect after the active domain restarted.
- *detach-disk* - detach disk device
- *detach-interface* - detach network interface
- *domid* - convert a domain name or UUID to domain id.
- *domif-setlink* - set link state of a virtual interface.
- *domiftune* - get/set parameters of a virtual interface.
- *domname* - convert a domain id or UUID to domain name.
- *domuuid* - convert a domain name or id to domain UUID.
- *domxml-from-native* - convert a QEMU command line to libvirt XML
- *domxml-to-native* - convert a libvirt XML file (for QEMU) to a native QEMU command line. Useful for debugging.
- *dumpxml* - output the XML for the specified domain
- *edit* - edit XML configuration for a domain.
- *maxvcpus* - show connection vcpu maximum (for QEMU)
- *memtune* - get or set memory parameters.
- *qemu-monitor-command* - for QEMU/KVM domains allows sending commands to the QEMU monitor
- *reset* - reset a domain
- *restore* - restore a domain from a saved state in a file
- *resume* - resume a suspended domain. After *resume* the domain is in the "running" state.
- *save* - save a domain state to a file
- *schedinfo* - show/set scheduler parameters (need mount cgroup CPU controller).
- *setmaxmem* - change maximum memory limit.
- *setmem* - change memory allocation.
- *start* - start a domain
- *suspend* - suspend a running domain. After *suspend* the domain is the "paused" state.
- *ttyconsole* - show tty console.
- *undefine* - remove a domain (undo the effects of *define*)
- *vcpucount* - show domain vcpu counts.
- *vcpuinfo* - show detailed domain vcpu information (for QEMU).
- *vcpupin* - control or query domain vcpu affinity (for QEMU).

- *emulatorpin* - control or query domain emulator affinity.
- Domain Monitoring
  - *domblkerror* - show errors on block devices (for QEMU).
  - *domblkinfo* - show domain block device size information.
  - *domblklist* - list all domain blocks.
  - *domblkstat* - get device block stats for a domain.
  - *domcontrol* - show domain control interface state.
  - *domif-getlink* - get link state of a virtual interface.
  - *domiflist* - list all domain virtual interfaces.
  - *domifstat* - get network interface stats for a domain.
  - *dominfo* - show domain information.
  - *dommemstat* - get memory statistics for a domain.
  - *domstate* - show domain state.
  - *list* - show the status of all domains
- Host and Hypervisor
  - *capabilities* - show capabilities.
  - *hostname* - print the hypervisor hostname.
  - *nodecpumap* - show node cpu map.
  - *nodecpustats* - print cpu stats of the node.
  - *nodeinfo* - show node information.
  - *nodememstats* - print memory stats of the node.
  - *sysinfo* - print the hypervisor sysinfo.
  - *uri* - print the hypervisor canonical URI.
  - *version* - show version.
- Snapshot
  - *snapshot-create* - Create a snapshot from XML
  - *snapshot-create-as* - Create a snapshot from a set of args
  - *snapshot-current* - Get or set the current snapshot
  - *snapshot-delete* - Delete a domain snapshot
  - *snapshot-dumpxml* - Dump XML for a domain snapshot
  - *snapshot-edit* - edit XML for a snapshot
  - *snapshot-info* - snapshot information
  - *snapshot-list* - List snapshots for a domain
  - *snapshot-parent* - Get the name of the parent of a snapshot
  - *snapshot-revert* - Revert a domain to a snapshot
- Virsh itself
  - *cd* - change the current directory.
  - *connect* - (re)connect to hypervisor.

- *echo* - echo arguments.
- *exit* - quit this interactive terminal.
- *help* - print help.
- *pwd* - print the current directory.
- *quit* - quit this interactive terminal.

Other virsh commands may operate correctly, but have not been specifically validated in the QorIQ SDK.

## 9.2.2 Build, Installation, and Configuration

### 9.2.2.1 Building Libvirt with Yocto

Libvirt is a Linux user space package that can easily be added to a root filesystem using the Yocto build system.

In the NXP SDK, Libvirt and all pre-requisite user space packages are included when building the "virt" and "full" image type:

```
bitbake fsl-image-virt
bitbake fsl-image-full
```

Libvirt can be easily added to any rootfs image by updating the IMAGE\_INSTALL\_append variable in the **conf/local.conf** file in the Yocto build environment. For example, append the following line to **local.conf**:

```
IMAGE_INSTALL_append = " libvirt libvirt-libvirtd libvirt-virsh"
```

After libvirt is included in a root filesystem the libvirtd daemon will be automatically started by the system init scripts.

### 9.2.2.2 Running libvirtd

#### Running libvirtd

The libvirtd daemon is installed as part of a libvirt packages installation. By default the target system init scripts should start libvirtd.

Running libvirtd on the target system is a pre-requisite to running any management tools such as virsh.

The libvirtd daemon can be manually started like this:

```
$ /etc/init.d/libvirtd start
```

In some circumstances the daemon may need to be restarted such as after mounting cgroups or hugetlbfs. Daemon restart can be done like this:

```
$ /etc/init.d/libvirtd restart
```

The libvirtd daemon can be configured in `/etc/libvirt/libvirtd.conf`. The file is self-documented and has detailed comments on the configuration options available.

#### The libvirtd Daemon and Logging

The libvirt daemon logs data to `/var/log/libvirt/`

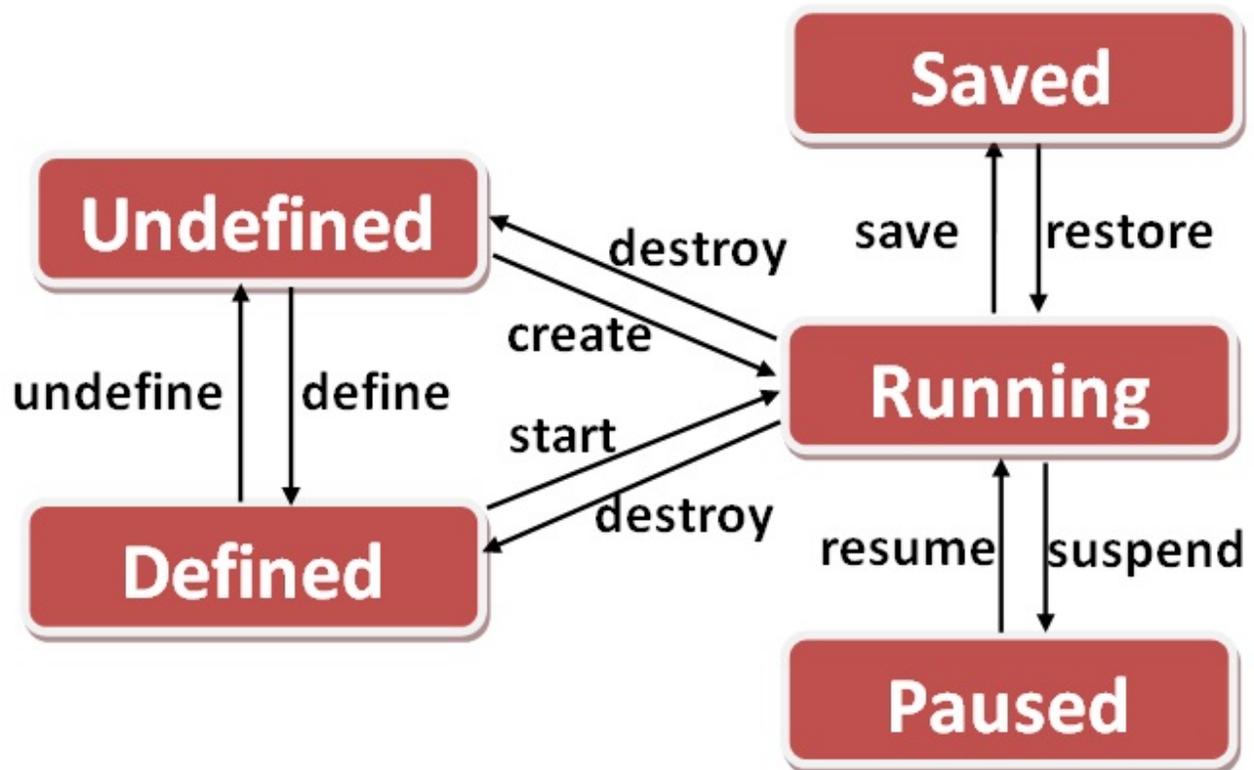
- General libvirtd log messages are in: `/var/log/libvirt/libvirtd.log`
- QEMU/KVM domain logs are in: `/var/log/libvirt/qemu/[domain-name].log`
- LXC domains logs are in: `/var/log/libvirt/lxc/[domain-name].log`

The verbosity of logging can be controlled in `/etc/libvirt/libvirtd.conf`.

### 9.2.2.3 Libvirt Domain Lifecycle

Two types of libvirt domains are supported in the QorIQ SDK-- KVM/QEMU virtual machines and Linux containers.

The following state diagram illustrates the lifecycle of a domain, the states that domains can be in and the **virsh** commands that move the domain between states.



#### Domain States

##### 1. Undefined

There are two types of domains-- persistent and transient domains. All domains begin in the "undefined" state where they are defined in XML definition file, and libvirt is unaware of them.

##### 2. Defined

Persistent domains begin with being "defined". This adds the domain to libvirt, but it is not running. This state can also be conceptually thought of as "stopped". The output of **virsh list --all** shows the domain as being "shut off".

##### 3. Running

The "running" state is the normal state of an active domain after it has been started. The **start** command is used to move persistent domains into this state. Transient domains go from being undefined to "running" through the **create** command.

##### 4. Paused

The domain execution has been suspended. The domain is unaware of being in this state.

##### 5. Saved

The domain state has been saved and could be restored again.

See the [Basic Example](#) on page 1252 article for an example of a container domain lifecycle.

## 9.2.2.4 Libvirt URIs

Because libvirt supports managing multiple types of virtualization domains (possibly remote) it uses uniform resource identifiers (URIs) to describes the target "node" to manage and the type of domain being managed.

A URI is specified when tools such as **virsh** makes a connection to a target node running **libvirtd**.

Two types of URIs are supported in the QorIQ SDK-- QEMU/KVM and LXC.

QEMU/KVM URIs are in the form:

- For a local node: *qemu:///system*
- For a remote node: *qemu[+transport]://[hostname]/system*

For Linux containers:

- For a local node: *lxc:///*
- For a remote node: *lxc[+transport]://[hostname]/*

A default URI can be specified in the environment (LIBVIRT\_DEFAULT\_URI) or in the /etc/libvirt/libvirtd.conf config file.

For further information about URIs:

- <http://libvirt.org/uri.html>
- [http://libvirt.org/remote.html#Remote\\_URI\\_reference](http://libvirt.org/remote.html#Remote_URI_reference)

## 9.2.2.5 virsh

The virsh command is a command line tool provided with the libvirt package for managing libvirt domains. It can be used to create, start, pause, shutdown domains. The general command format is:

```
virsh [OPTION]... <command> <domain> [ARG]...
```

## 9.2.2.6 Libvirt xml

The libvirt XML format is defined at: <http://libvirt.org/format.html>.

## 9.2.3 Examples

### 9.2.3.1 KVM Examples

#### 9.2.3.1.1 Libvirt KVM/QEMU Example (ARM Architecture)

The following example shows the lifecycle of a simple KVM/QEMU libvirt domain called **kvm**. In this example the default URI is *qemu:///system*, and because of this default an explicit URI is not used in the virsh commands

Libvirt has the possibility to convert qemu command line arguments in xml. However, the current version of libvirt does not have full support for all qemu arguments. It can be used to generate a basic xml file, but there is currently a problem that it generates a default USB node which should be removed.

1. We begin with a simple QEMU command line in a text file named **kvm.args**:

- 32 bit ARMv7:

```
echo "/usr/bin/qemu-system-arm -name kvm -smp 2 -enable-kvm -m 512 -nographic -cpu host -machine
type=virt -kernel /boot/zImage -serial pty -initrd /boot/fsl-image-core-ls1021atwr.ext2.gz -
append 'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' " > kvm.args
```

- 64 bit ARMv8:

```
echo "/usr/bin/qemu-system-aarch64 -name kvm -smp 2 -enable-kvm -m 512 -nographic -cpu host -
machine type=virt -kernel /boot/Image -serial pty -initrd /boot/guest.rootfs.ext2.gz -append
'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' " > kvm.args
```

**NOTE**

The serial console is a tty, not a telnet server. The -name option is required and specifies the name of the virtual machine.

2. Before defining the domain the QEMU command line must be converted to libvirt XML format:

```
virsh domxml-from-native qemu-argv kvm.args > kvm.xml
```

**NOTE**

For ARMv7 platforms the generated xml file has to be manually changed to remove the USB node.

The content of the newly created domain XML file is shown below:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
 <name>kvm</name>
 <uuid>12a3391e-9cd4-43dd-b8b7-27b1fb193378</uuid>
 <memory unit='KiB'>524288</memory>
 <currentMemory unit='KiB'>524288</currentMemory>
 <vcpu placement='static'>2</vcpu>
 <os>
 <type machine='virt'>hvm</type>
 <kernel>/boot/zImage</kernel>
 <initrd>/boot/fsl-image-core-ls1021atwr.ext2.gz</initrd>
 <cmdline>root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk</cmdline>
 </os>
 <cpu mode='custom' match='exact'>
 <model fallback='allow'>host</model>
 </cpu>
 <clock offset='utc'/>
 <on_poweroff>destroy</on_poweroff>
 <on_reboot>restart</on_reboot>
 <on_crash>destroy</on_crash>
 <devices>
 <emulator>/usr/bin/qemu-system-arm</emulator>
 <controller type='usb' index='0'/>
 <serial type='pty'>
 <target port='0'/>
 </serial>
 <console type='pty'>
 <target type='serial' port='0'/>
 </console>
 </devices>
</domain>
```

A more complex example including devices can be found below.

The example contains two devices:

- a virtio network interface
- a virtio block device (disk)

**NOTE**

This example will use MMIO as transport for virtio. Currently libvirt has no support for PCI transport, but it can be used using passthrough QEMU command line arguments (see the next example)

Device example (using virtio with MMIO as transport):

```
<domain type='kvm' xmlns:gemu='http://libvirt.org/schemas/domain/qemu/1.0'>
 <name>kvm</name>
 <uuid>12a3391e-9cd4-43dd-b8b7-27b1fb193378</uuid>
 <memory unit='KiB'>524288</memory>
 <currentMemory unit='KiB'>524288</currentMemory>
 <vcpu placement='static'>2</vcpu>
 <os>
 <type machine='virt'>hvm</type>
 <kernel>/boot/zImage</kernel>
 <initrd>/boot/fsl-image-core-ls1021atwr.ext2.gz</initrd>
 <cmdline>root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk</cmdline>
 </os>
 <cpu mode='custom' match='exact'>
 <model fallback='allow'>host</model>
 </cpu>
 <clock offset='utc'/>
 <on_poweroff>destroy</on_poweroff>
 <on_reboot>restart</on_reboot>
 <on_crash>destroy</on_crash>
 <devices>
 <emulator>/usr/bin/qemu-system-arm</emulator>
 <serial type='pty'>
 <target port='0'/>
 </serial>
 <console type='pty'>
 <target type='serial' port='0'/>
 </console>

 <interface type='ethernet'>
 <mac address='52:54:00:b0:39:28'/>
 <script path='/home/root/qemu-ifup'/>
 </interface>
 <disk type='file' device='disk'>
 <driver name='qemu' type='raw' cache='none'/>
 <source file='/home/root/my_guest_disk'/>
 <target dev='vda' bus='virtio'/>
 </disk>

 </devices>
 <qemu:commandline>
 <qemu:arg value='-mem-path'/>
 <qemu:arg value='/var/lib/lugetlbfs/pagesize-2MB'/>
 </qemu:commandline>
</domain>
```

Device example (using virtio with PCI as transport):

```
<domain type='kvm' xmlns:gemu='http://libvirt.org/schemas/domain/qemu/1.0'>
 <name>kvm</name>
 <uuid>faa89ddc-e156-46c0-9d0f-029c322f9bf7</uuid>
 <memory unit='KiB'>1048576</memory>
 <currentMemory unit='KiB'>1048576</currentMemory>
```

```

<vcpu placement='static'>4</vcpu>
<os>
 <type arch='aarch64' machine='virt'>hvm</type>
 <kernel>/boot/Image</kernel>
 <initrd>/images/fsl-image-core-ls1043ar.db.ext2.gz</initrd>
 <cmdline>root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk</cmdline>
</os>
<cpu mode='custom' match='exact'>
 <model fallback='allow'>host</model>
</cpu>
<clock offset='utc' />
<on_poweroff>destroy</on_poweroff>
<on_reboot>restart</on_reboot>
<on_crash>destroy</on_crash>
<devices>
 <emulator>/usr/bin/qemu-system-aarch64</emulator>
 <serial type='pty'>
 <target port='0' />
 </serial>
 <console type='pty'>
 <target type='serial' port='0' />
 </console>
 <memballoon model='none' />
</devices>
<qemu:commandline>

 <qemu:arg value='--netdev' />
 <qemu:arg value='tap,id=tap0,script=/home/root/qemu-ifup,downscript=no,ifname=tap0' />
 <qemu:arg value='-device' />
 <qemu:arg value='virtio-net-pci,netdev=tap0' />

</qemu:commandline>
</domain>

```

### 3. Now the domain can be defined:

```

virsh define kvm.xml
Domain kvm defined from kvm.xml

virsh list --all
 Id Name State
 ---- --- -
 - kvm shut off

```

### 4. Next start the domain. This starts the VM and boots the guest Linux.

```

virsh start kvm
Domain kvm started

virsh list
 Id Name State
 ---- --- -
 3 kvm running

```

5. The **virsh console** command can be used to connect to the console of the running Linux domain.

```
virsh console kvm
Connected to domain kvm
Escape character is ^]

Poky (Yocto Project Reference Distro) 1.5 ls1021aqds /dev/ttyAMA0

ls1021aqds login: root
```

6. To stop the domain use the destroy command:

```
virsh destroy kvm
Domain kvm destroyed

root@p4080ds:~# virsh list --all
 Id Name State

 - kvm shut off
```

7. To remove the domain from libvirt, use the undefine command:

```
virsh undefine kvm
Domain kvm has been undefined

root@p4080ds:~# virsh list --all
 Id Name State

```

## 9.2.3.2 Libvirt\_lxc Examples

### 9.2.3.2.1 Basic Example

The following example shows the lifecycle of a simple LXC libvirt domain called *container1*. The virsh tool is used for managing the lxc domain lifecycle.

**1. Confirm the host Linux configuration.** Begin by confirming that the host kernel is configured correctly and that rootfs setup such as mounting cgroups has been done. This can be done with the lxc-checkconfig command.

```
lxc-checkconfig
--- Namespaces ---
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
Pid namespace: enabled
User namespace: missing
Network namespace: enabled
Multiple /dev/pts instances: enabled

--- Control groups ---
Cgroup: enabled
Cgroup clone_children flag: enabled
Cgroup device: enabled
Cgroup sched: enabled
Cgroup cpu account: enabled
Cgroup memory controller: enabled
```

```
Cgroup cpuset: enabled

--- Misc ---
Veth pair device: enabled
Macvlan: enabled
Vlan: enabled
File capabilities: enabled
```

**2. Create a libvirt XML file defining the container.** The example below shows a very simple container defined in `container1.xml` that runs the command `/bin/sh` and has a console:

```
cat container1.xml
<domain type='lxc'>
 <name>container1</name>
 <memory>500000</memory>
 <os>
 <type>exe</type>
 <init>/bin/sh</init>
 </os>
 <devices>
 <console type='pty' />
 </devices>
</domain>
```

**3. Define the container.** The `virsh define` command processes the XML and makes creates the new libvirt domain.

```
virsh -c lxc:/// define container1.xml
Domain container1 defined from container1.xml

virsh -c lxc:/// list --all
 Id Name State

- container1 shut off
```

**4. Start the container.**

```
virsh -c lxc:/// start container1
Domain container1 started

virsh -c lxc:/// list
 Id Name State

3196 container1 running
```

**5. Connect to the console.**

```
virsh -c lxc:/// console container1
Connected to domain container1
Escape character is ^]
sh-4.2#
sh-4.2# ps -ef
UID PID PPID C STIME TTY TIME CMD
root 1 0 0 18:25 pts/2 00:00:00 /bin/sh
root 3 1 0 18:36 pts/2 00:00:00 ps -ef

sh-4.2# ifconfig br0
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
```

```
inet 10.171.73.123 netmask 255.255.254.0 broadcast 10.171.73.255
inet6 fe80::a00:27ff:fe01:fe07 prefixlen 64 scopeid 0x20<link>
ether 08:00:27:01:fe:07 txqueuelen 0 (Ethernet)
RX packets 865838 bytes 104029354 (99.2 MiB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 104446 bytes 43998714 (41.9 MiB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

```
sh-4.2#
```

Press CTRL + ] to exit the console.

The following aspects must be noted:

- the processes inside the container are running in a separate namespace, hence the different process hierarchy
- since no network configuration for the domain is explicitly specified, all networking interfaces are shared with the host (all the other interfaces are present too - br0 is mentioned as an example)
- since no filesystem configuration is specified for the domain, the filesystem is shared with the host-- all host mounts are present in the container as well.

6. To stop the container use the destroy command:

```
virsh -c lxc:/// destroy container1
Domain container1 destroyed

virsh -c lxc:/// list --all
 Id Name State

- container1 shut off
```

7. To remove the domain from libvirt, use the undefine command.

```
virsh -c lxc:/// undefine container1
Domain container1 has been undefined
```

## 9.2.3.2.2 Custom Container Filesystem

The libvirt documentation (<http://libvirt.org/formatdomain.html#elementsFilesystems>) details the flavors and usage of the **filesystem** tag in order to assign particular types of filesystem mounts to the domain.

### Mounts

The <mount> tag specifies private root filesystem, available on host in a specific directory. It will be the rootfs of the container. The filesystem can be handcrafted, installed from media, debootstrapped, etc. Below is an example snippet of the XML that assigns a filesystem to a container:

```
<domain type='lxc'>
 [...]
 <devices>
 [...]
 <filesystem type='mount'>
 <source dir='/var/lib/lxc/foo/rootfs'/>
 <target dir='/'/>
 </filesystem>
 </devices>
</domain>
```

The result is that the domain is started with the root mounted at `/var/lib/lxc/foo/rootfs`.

For ease of use, the example that follows will use the standard LXC command `lxc-create` to build a container Busybox rootfs. However, the default rootfs created by `lxc-create` will not work with libvirt tools as-is, and some additional terminal setup must be done. This will be detailed in the next example: [Container Terminal Setup](#) on page 1255.

**Please note** that, if you're planning to use this LXC built rootfs with libvirt containers, you will also have to bind-mount the host library directories. These vary depending on whether the libraries are 32bit, 64 bit or both. Basically, you will have to bind-mount all the available library dirs among `/lib`, `/usr/lib`, `/lib64`, `/usr/lib64`, using entries like the one below:

```
<domain type='lxc'>
 [...]
 <devices>
 [...]
 <filesystem type='mount'>
 <source dir='/lib'/>
 <target dir='/lib'/>
 <readonly />
 </filesystem>
 </devices>
</domain>
```

### 9.2.3.2.3 Container Terminal Setup

Each LXC domain needs at least one console device defined in the XML. By default, libvirt will link this console to the process and make it available with **virsh console** command.

Libvirt also offers support for multiple consoles in a container. This section provides an example describing how libvirt can be used to start four processes inside the container and assign each one of these a private terminal. This will be done by using:

- a Busybox container filesystem, built with `lxc-create`
- a modified `inittab`
- the container XML configuration

Libvirt will mount a `tmpfs` on `/dev` and a `devpts` on `/dev/pts` and it will create all the device nodes itself inside the domain. The domain definition will be altered so that it will start the `busybox-init` as the `init` process.

The `init` process reads the `/etc/inittab` file inside the rootfs and will determine additional processes to start, and the terminals to link them to. Here is an example `inittab` that specifies the four processes to start and the separate `tty` device assigned to each:

```
::sysinit:/etc/init.d/rcS
tty1::askfirst:/bin/sh
tty2::respawn:/bin/getty -L tty2 115200 vt100
tty3::respawn:/bin/getty -L tty3 115200 vt100
tty4::respawn:/bin/getty -L tty4 115200 vt100
```

The domain XML configuration that shows the `/sbin/init` as the initial program to run and describes the four `tty` devices looks like this:

```
<domain type='lxc'>
 [...]
 <os>
 <type>exe</type>
 <init>/sbin/init</init>
 </os>
 [...]
```

```

<devices>
 [...]
 <console type='pty'>
 <target type='serial' port='0'/>
 </console>
 <console type='pty'>
 <target type='serial' port='1'/>
 </console>
 <console type='pty'>
 <target type='serial' port='2'/>
 </console>
 <console type='pty'>
 <target type='serial' port='3'/>
 </console>
</devices>
</domain>

```

Using the inittab above and XML configs, **virsh start** will start the following process hierarchy:

```

ps axf
[...]
18450 ? Ss 0:00 /usr/libexec/libvirt_lxc --name foo --console 24 --console 25 --console 26
--console 27 --security=selinux --handshake 30 --background
18451 pts/0 Ss+ 0:00 _ init
18454 ? Ss 0:00 _ /bin/syslogd
18459 ? Ss 0:00 _ /bin/sh
18460 pts/1 Ss+ 0:00 _ /bin/getty -L tty2 115200 vt100
18461 pts/2 Ss+ 0:00 _ /bin/getty -L tty3 115200 vt100
18462 pts/3 Ss+ 0:00 _ /bin/getty -L tty4 115200 vt100

```

By running **virsh -c lxc:/// dumpxml foo**, we can see what alias libvirt has assigned to each console device:

```

<domain type='lxc' id='18450'>
 [...]
 <devices>
 [...]
 <console type='pty' tty='/dev/pts/3'>
 <source path='/dev/pts/3'/>
 <target type='serial' port='0'/>
 <alias name='console0'/>
 </console>
 <console type='pty'>
 <source path='/dev/pts/4'/>
 <target type='serial' port='1'/>
 <alias name='console1'/>
 </console>
 <console type='pty'>
 <source path='/dev/pts/5'/>
 <target type='serial' port='2'/>
 <alias name='console2'/>
 </console>
 <console type='pty'>
 <source path='/dev/pts/6'/>
 <target type='serial' port='3'/>
 <alias name='console3'/>
 </console>
 </devices>
</domain>

```

To connect to a particular console, one must run

```
virsh -c lxc:/// console --devname <console_alias> <domain_name>
```

```
[root@everest][~]# virsh -c lxc:/// console --devname console2 foo
Connected to domain foo
Escape character is ^]

everest.ea.freescale.net login: root
#
#
#
```

## 9.2.3.2.4 Networking Examples

Libvirt offers extensive support when it comes to networking. The purpose of this section is to provide some insight of how standard LXC networking scenarios can be achieved with Libvirt - to provide the same functionality. In its further versions, this document could include details regarding other, more advanced networking options as well.

### 9.2.3.2.4.1 Shared Networking

#### Shared Networking

By default libvirt containers share network interfaces and the network namespace with the host. To share the hosts network interfaces, simply do not define any network interfaces in the domain XML.

When at least one interface is defined, the container will be started in a new network namespace, with the defined interface and a loopback.

#### No Networking

In order to remove all access to the host's network interfaces, start the container in a new network namespace, without specifying any interface. To do this use the <privnet> XML tag as seen in the example below. This will a loopback interface in the new namespace, but the host network interfaces are not visible in the container.

```
<domain type='lxc'>
 [...]
 <features>
 <privnet/>
 </features>
</domain>
```

### 9.2.3.2.4.2 Ethernet Bridging

This is the recommended configuration for general guest activity on hosts with static wired networking configurations. It relies on 802.1d Ethernet Bridging, and it provides a bridge from the VM directly into the LAN. This assumes there is a bridge device on the host with one or more of the host's physical NICs enslaved to it. The guest VM will have an associated tun device created with a name of vnetN, which can also be overridden with the **target** element in the XML config file. This tun device will also be enslaved to the bridge. The IP range / network configuration is whatever is used on the LAN. This provides the guest VM full incoming and outgoing net access just like a physical machine. The bridge normally is a Linux bridge - but this can be configured to be an open vSwitch as well, if it is supported on the host. This is done by adding some further parameters in the config file.

#### Host Configuration

The physical interface **fm1-gb1** is added to a bridge device - **br0**:

```
ifconfig fm1-gb1 0.0.0.0 up
brctl addbr br0
ifconfig br0 192.168.1.141 up
brctl addif br0
```

### XML Description of the Interface

Only the relevant part is described here - defining the interface of the guest in XML:

```
<domain type='lxc'>
 [...]
 <devices>
 [...]
 <interface type="bridge">
 <source bridge="br0" />
 </interface>
 </devices>
</domain>
```

### Guest Configuration and Testing

After booting the container the network interface can be managed normally:

```
~ # ifconfig
eth0 Link encap:Ethernet HWaddr 52:54:00:83:78:F4
 inet6 addr: fe80::5054:ff:fe83:78f4/64 Scope:Link
 UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
 RX packets:22 errors:0 dropped:0 overruns:0 frame:0
 TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:1000
 RX bytes:3104 (3.0 KiB) TX bytes:636 (636.0 B)

lo Link encap:Local Loopback
 inet addr:127.0.0.1 Mask:255.0.0.0
 inet6 addr: ::1/128 Scope:Host
 UP LOOPBACK RUNNING MTU:16436 Metric:1
 RX packets:0 errors:0 dropped:0 overruns:0 frame:0
 TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:0
 RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

~ # ifconfig eth0 192.168.1.143
~ # ping -c 3 192.168.1.1 # gateway
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: seq=0 ttl=64 time=3.557 ms
64 bytes from 192.168.1.1: seq=1 ttl=64 time=0.220 ms
64 bytes from 192.168.1.1: seq=2 ttl=64 time=0.227 ms

--- 192.168.1.1 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.220/1.334/3.557 ms
~ # ping -c 3 192.168.1.141 # host
PING 192.168.1.141 (192.168.1.141): 56 data bytes
64 bytes from 192.168.1.141: seq=0 ttl=64 time=3.493 ms
64 bytes from 192.168.1.141: seq=1 ttl=64 time=0.050 ms
64 bytes from 192.168.1.141: seq=2 ttl=64 time=0.036 ms

--- 192.168.1.141 ping statistics ---
```

```
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.036/1.193/3.493 ms
~ #
```

### 9.2.3.2.4.3 MACVLAN

Macvlan is a relatively new Linux kernel technology used to ease the task of networking in virtual machines. See the following article for some useful overview information: <http://seravo.fi/2012/virtualized-bridged-networking-with-macvtap>.

Macvlan is device driver consisting of 2 components:

- the **macvlan** driver - which makes it possible to create **virtual network interfaces** that "cling on" a physical network interface. Each virtual interface has its own MAC address distinct from the physical interface's MAC address. Frames sent to or from the virtual interfaces are mapped to the physical interface, which is called **the lower interface**.
- the **tap** interface - a software-only interface, using to pass Ethernet frames. Instead of passing frames to and from a physical Ethernet card, the frames are read and written by a userspace program. The kernel makes the Tap interface available via the `/dev/tapN` device file, where N is the index of the network interface.

Libvirt uses the macvtap device technology to attach virtual network interfaces to physical ones. This has no impact on the functionality of the physical interfaces on the host. The macvtap device has a different approach than the bridge. While the latter provides connectivity from the host to the virtual devices, the former isolates them. The bridge unifies the physical interface with the virtual ones, thus providing a common addressing space and connectivity between each 2 endpoints. The macvtap device will put the virtual interfaces in separate MAC-based VLAN's, isolated from the host.

The macvtap device can function in three different modes: **vepa**, **bridge** and **private** - libvirt supports all three of them. In order to configure networking using a macvtap device, the type attribute of the interface is set to "direct" which can be seen in the following XML example:

```
<domain type='lxc'>
 [...]
 <devices>
 [...]
 <interface type="direct">
 <source dev="p7p1" mode="vepa" />
 </interface>
 </devices>
</domain>
```

In the above example, **p7p1** is a physical network interface on the host. The **mode** tag specifies the mode of the macvtap device - and it can be one of the following:

- **vepa** (Virtual Ethernet Port Aggregator) - in this mode, which is the default, data between endpoints on the same lower device are sent via the lower device (physical Ethernet card), to the physical switch the lower device is connected to. This mode requires that the switch supports *Reflective Relay* mode, also known as *Hairpin* mode. Reflective Relay means that the switch can send back a frame on the same port it received it on. The reason this mode exists is to leverage the switching computation to an external switch (and thus freeing the host).
- **bridge** - in this mode, the endpoints can communicate directly without sending the data out via the lower device. There is no isolation between endpoints on the same lower device, but there is isolation between them and the lower device itself.
- **private** - the nodes on the same Macvtap device can never talk to each other. This is used when you want to isolate the virtual machines connected to the endpoints from each other, but not from the outside network.

### 9.2.3.2.4.4 Direct Assignment

This options enables a container to have private access to a host interface directly. It is similar to the **lxc-phys** networking configuration option. Technically, this option will move a host network interface from the host network namespace to the

container's. Once the container is stopped (destroyed), the interface will be assigned back to the host network namespace. While the container is running, the interface will not be available from the host.

### XML Description of the Interface

Assuming the host has interface **fm2-gb0**, this is the XML snippet that assigns it to the container:

```
<domain type='lxc'>
 [...]
 <devices>
 [...]
 <hostdev mode='capabilities' type='net'>
 <source>
 <interface>fm2-gb0</interface>
 </source>
 </hostdev>
 </devices>
</domain>
```

### Guest Configuration and Testing

Once the container is started, the interface must be configured with IP, netmask, etc. Then it can be used just like it would have been on the host.

```
~ # ifconfig fm2-gb0
fm2-gb0 Link encap:Ethernet HWaddr 00:04:9f:00:02:05
 BROADCAST MULTICAST MTU:1500 Metric:1
 RX packets:18 errors:0 dropped:0 overruns:0 frame:0
 TX packets:56 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:1000
 RX bytes:1094 (1.0 KiB) TX bytes:4068 (3.9 KiB)
 Memory:fe5e0000-fe5e0fff

~ # ifconfig fm2-gb0 20.0.0.3 netmask 255.0.0.0
~ # ping -c 3 20.0.0.1
PING 20.0.0.1 (20.0.0.1) 56(84) bytes of data:
64 bytes from 20.0.0.1: icmp_req=1 ttl=64 time=0.377 ms
64 bytes from 20.0.0.1: icmp_req=2 ttl=64 time=0.186 ms
64 bytes from 20.0.0.1: icmp_req=3 ttl=64 time=0.198 ms

--- 20.0.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.186/0.253/0.377/0.089 ms
~ #
```

When returning to the host network namespace, the interface loses its configuration and it is down.

## 9.2.3.2.4.5 VLAN

Libvirt does not directly provide VLAN configuration support-- there is no equivalent to the capability in user space LXC where a container can be configured to have a specific VLAN assigned only to itself (see [LXC: How to configure networking using a VLAN \(lxc-vlan.conf\)](#) on page 1282).

Libvirt containers does not offer equivalent support, but VLANs can be used with libvirt containers using traditional VLAN tools such as vconfig, and the **bridge** or **macvtap** network sharing modes.

This leads to the following different configuration scenarios for using VLAN based network interfaces:

1. Configuring a VLAN on the host:

```
vconfig add eth0 2
ifconfig eth0.2 192.168.20.2
```

then exposing the **eth0.2** sub-interface in the containers, using **bridging** or direct attachment through a **macvtap** device. These would provide the same functionality as if **eth0.2** were a normal physical interface on the host, but it will only provide connectivity inside the VLAN.

2. Configuring VLAN in container shared with macvtap - in this scenario we assume that the domain has been configured to attach directly to the eth0 interface, and we create the VLAN inside the container. Note that this scenario works only if eth0 does not have a sub-interface in the same VLAN defined on the host. As stated in the Macvtap section, ping will work only between virtual endpoints and the outside network.
3. Configuring VLAN in container shared with bridge - in this scenario we assume that the domain has been configured to attach to a host bridge. This bridge previously had a physical interface attached to it. When starting the domain, virtual endpoints will be created and attached to this bridge inside the domain. We create the VLAN sub-interfaces inside the created containers. Note that this scenario works only if there has not been configured a sub-interface on the physical interface with the same VLAN id.

## 9.3 Linux Containers (LXC) for NXP QorIQ User's Guide

### 9.3.1 Introduction to Linux Containers

#### 9.3.1.1 NXP LXC Release Notes

This document describes current limitations in the release of LXC for NXP SoCs.

```
Copyright (C) 2015 NXP Semiconductors, Inc.
```

```
NXP LXC Release Notes
04/27/2016
```

```
Overview

```

```
This document describes new features, current limitations, and
working demos in Linux Containers (LXC) for NXP QorIQ SDK 2.0.
```

```
Fixes

```

- o Seccomp support on ARMv8 platforms.
- o Unprivileged containers support on ARMv8 platforms.

```
SDK Demo List

```

- o Basic container usage flow and management commands
- o Container networking setups
  - o Shared networking
  - o Private NICs
  - o Ethernet bridge
  - o MACVLAN
  - o VLAN
- o Adjusting container capabilities
- o Tuning container resource usage

- o Running application containers
- o Isolating USDPAA applications in LXC containers. This has been tested using the USDPAA reflector app in a Multiple Instance Scenario on a DPAA board. After partitioning the board resources in order to support multiple reflector instances, these have been further isolated in container environments.
- o Running an unprivileged container linked to a host bridge.
- o Running containers with Seccomp protection.

### 9.3.1.2 Overview

This document is a guide and tutorial to using Linux Containers on NXP e500-based, ARMv7 and ARMv8-based SoCs.

Linux Containers is a lightweight virtualization technology that allows the creation of environments in Linux called "**containers**" in which Linux applications can be run in isolation from the rest of the system and with fine grained control over resources allocated to the container (e.g. CPU, memory, network).

There are 2 implementations of containers in the QorIQ SDK:

- LXC. LXC is a user space package that provides a set of commands to create and manage containers and uses existing Linux kernel features to accomplish the desired isolation and control.
- Libvirt. The libvirt package is a virtualization toolkit that provides a set of management tools for managing virtual machines and Linux containers. See the **Libvirt Users Guide** chapter for general information regarding libvirt. The libvirt driver for containers is called "lxc", but the libvirt "lxc" driver is distinct from the user space LXC package.

Applications in a container run in a "sandbox" and can be restricted in what they can do and what visibility they have. In a container:

- An application "sees" only other processes that are in the container.
- An application has access only to network resources granted to the container.
- If configured as such, an application "sees" only a container-specific root filesystem. In addition to limiting access to data in the system's host rootfs, by limiting the `/dev` entries that exist in the containers rootfs this limits the devices that the container can access.
- The file POSIX capabilities available to programs are controlled and configured by the system administrator.
- The container's processes run in what is known as a "control group" which the system administrator can use to monitor and control the container's resources.

Why are containers useful? Below are a few examples of container use cases:

- **Application partitioning** -- control CPU utilization between high priority and low priority applications, control what resources applications can access.
- **Virtual private server** -- boot multiple instances of user space, each which effectively looks like a private instance of a server. This approach is commonly used in website infrastructure.
- **Software upgrade** -- run Linux user space in a container, when it becomes necessary to upgrade applications in the system, create and test upgraded software in a new container. The old container can be stopped and the new container can be started as desired.
- **Terminal servers** -- user accesses the system with a thin client, with containers on the server providing applications. Each user gets a private, sandboxed workspace.

There are two general usage models for containers:

- **application containers**: Running a single application in a container. In this scenario, a single executable program is started in the container.
- **system containers**: Booting an instance of user space in a container. Booting multiple system containers allows multiple isolated instances of user space to run at the same time.

Containers are conceptually different than virtual machine technologies such as QEMU/KVM. Virtual machines emulate a hardware platform and are capable of booting an operating system kernel. A container is a mechanism to isolate Linux applications. In a system using containers there is only one Linux kernel running -- the host Linux kernel.

### 9.3.1.3 Comparing LXC and Libvirt

LXC and the `lxc` driver in `libvirt` provide similar capabilities and use the same kernel mechanisms to create containers. This section highlights some of the differences between the two tools.

#### LXC

- Container management is done with local LXC package commands. No remote support.
- Container creation done with `lxc-create`. LXC config file and template govern the creation of the template and the container's rootfs.

#### libvirt

- `libvirt` abstracts the container and thus a variety of tools can be used to manage containers.
- Remote management is supported.
- Container configuration defined in `libvirt` XML file.
- No tools to facilitate container creation.
- Same tools can be used to manage containers and KVM/QEMU virtual machines.

### 9.3.1.4 For Further Information

Linux containers is an approach to virtualization similar to OS virtualization solutions such as Linux VServer and OpenVZ that are widely used for virtual private servers. Documentation for these projects has helpful and relevant information:

- <http://linux-vserver.org/Overview>
- [http://wiki.openvz.org/Main\\_Page](http://wiki.openvz.org/Main_Page)

The LXC package is an open source project and much information is available online.

See the chapter *Libvirt Users Guide* for general information about `libvirt`.

#### Web

- `libvirt` LXC driver: <http://libvirt.org/drvlxc.html>
- Getting started with LXC using `libvirt` : <https://www.berrange.com/posts/2011/09/27/getting-started-with-lxc-using-libvirt/>
- LXC: Official web page for the LXC project: <https://linuxcontainers.org/>
- LXC: Overview article on LXC on IBM developerWorks (2009): <http://www.ibm.com/developerworks/linux/library/l-lxc-containers/>
- Article on POSIX file capabilities: <http://www.friedhoff.org/posixfilecaps.html>
- SUSE LXC tutorial: [https://www.suse.com/documentation/sles11/singlehtml/lxc\\_quickstart/lxc\\_quickstart.html](https://www.suse.com/documentation/sles11/singlehtml/lxc_quickstart/lxc_quickstart.html)
- LXC Linux Containers, presentation: <http://www.slideshare.net/samof76/lxc-17456998>
- Stephane Graber's LXC 1.0 blog posts: <https://www.stgraber.org/2013/12/20/lxc-1-0-blog-post-series/>
- Linux Plumbers 2013 videos: <https://www.youtube.com/channel/UCIxsmRWj3-795FMlrsikd3A/videos>

#### Containers and Security

If using containers to sandbox untrusted applications, a thorough understanding is needed of the capabilities granted to a container and the security vulnerabilities they may imply. The following references are helpful for understanding container security:

- Ubuntu's security issues and mitigations with LXC, <https://wiki.ubuntu.com/LxcSecurity>
- Emeric Nasi, Exploiting capabilities, [http://www.sevagas.com/IMG/pdf/exploiting\\_capabilities\\_the\\_dark\\_side.pdf](http://www.sevagas.com/IMG/pdf/exploiting_capabilities_the_dark_side.pdf)
- Secure containers with SELinux and Smack, <http://www.ibm.com/developerworks/linux/library/l-lxc-security/index.html>
- Seccomp and sandboxing, <http://lwn.net/Articles/332974/>

### Mailing Lists

For LXC, there are two mailing lists available which can be subscribed to. Archives of the lists are also available.

```
https://lists.linuxcontainers.org/listinfo/lxc-devel
```

```
https://lists.linuxcontainers.org/listinfo/lxc-users
```

## 9.3.2 LXC: Build, Installation, Configuration

### 9.3.2.1 Summary

To prepare a root filesystem with the needed components for using LXC-based or libvirt containers the following steps are required:

1. Build and include the LXC and/or libvirt packages in the rootfs. For LXC see: [LXC: Building with Yocto](#) on page 1264). For libvirt see the chapter *Libvirt Users Guide* .
2. Update the Linux kernel configuration and build to included features needed to support containers (see [Building the Linux Kernel](#) on page 1265).
3. Update the rootfs so the system is ready to support containers (see [Host Root Filesystem Configuration for Linux Containers](#) on page 1267).

### 9.3.2.2 LXC: Building with Yocto

LXC is a Linux user space package that can easily be added to a rootfs using the Yocto build system.

In the NXP SDK, LXC and all pre-requisite user space packages are included when building the "full" and "virt" image types:

```
bitbake fsl-image-full
bitbake fsl-image-virt
```

LXC can be easily added to any rootfs image by updating the IMAGE\_INSTALL\_append variable in the **conf/local.conf** file in the Yocto build environment. For example, append the following line to **local.conf**:

```
IMAGE_INSTALL_append = " lxc"
```

If you are building for ARM64 platforms, you need to perform an additional step. You need to update the rootfs target in the board image - by default it is fsl-image-core. If you plan to update this rootfs to, say, fsl-image-virt, you need to add the following line in to build\_<machine\_release>/conf/local.conf:

```
ROOTFS_IMAGE = "fsl-image-virt"
```

After enabling the required Linux options mentioned in the following chapter, generate the kernel itb:

```
bitbake fsl-image-kernelitb
```

### 9.3.2.3 Building the Linux Kernel

In order to use LXC the Linux kernel must be configured with options to enable cgroups, namespaces, POSIX file capabilities, and options to support networking in containers.

These options can be enabled automatically by building the Linux kernel with an additional config fragment. In order to do this, add the following line in the **conf/local.conf** file in the Yocto build environment:

```
DELTA_KERNEL_DEFCONFIG_append = " <sdk-devel>/sources/meta-freescale/recipes-kernel/linux/files/containers.config"
```

Alternatively, you can enable the options manually. To configure and build the Linux kernel:

```
bitbake virtual/kernel -c cleansstate
bitbake virtual/kernel -c menuconfig
bitbake virtual/kernel
```

Make sure the following configuration options are enabled:

Kernel Configuration Options	Description
<pre> General setup ---&gt;   [*] Control Group support --&gt;       [*] Example debug cgroup subsystem       [*] Freezer cgroup subsystem       [*] Device controller for cgroups       [*] Cpuset support       [*] Simple CPU accounting cgroup subsystem       [*] Resource counters       [*]     Memory Resource Controller for Control Groups       [*]     Memory Resource Controller Swap Extension       [*]     Memory Resource Controller Swap Extension enabled by default (NEW)   [*]     Memory Resource Controller Kernel Memory accounting (EXPERIMENTAL)   [*]     HugeTLB Resource Controller for Control Groups   [*] Enable perf_event per-cpu per-container group (cgroup) monitoring   [*] Group CPU scheduler   [*] Block IO controller           </pre>	Control Group settings

Kernel Configuration Options	Description
<pre> General setup ---&gt;   [*] Namespaces support ---&gt;       [*] UTS namespace       [*] IPC namespace       [*] User namespace       [*] PID Namespaces       [*] Network namespace           </pre>	Namespaces settings

Kernel Configuration Options	Description
<pre>Device Drivers ---&gt; [*] Network device support ---&gt;     &lt;*&gt; MAC-VLAN support (EXPERIMENTAL)     &lt;*&gt;     MAC-VLAN based tap driver (EXPERIMENTAL)     &lt;*&gt; Virtual ethernet pair device</pre>	Network Device Drivers settings

Kernel Configuration Options	Description
<pre>Device Drivers ---&gt; [*] Character devices ---&gt;     [*] Unix98 PTY support     [*]     Support multiple instances of devpts</pre>	Character Device Drivers settings

Kernel Configuration Options	Description
<pre>[*] Networking support ---&gt;     Networking options ---&gt;         &lt;*&gt; 802.1d Ethernet Bridging         [*]     IGMP/MLD snooping (NEW)         &lt;*&gt; 802.1Q VLAN Support         [*]     GVRP (GARP VLAN Registration Protocol) support</pre>	Networking support settings

Kernel Configuration Options	Description
<pre>File systems ---&gt;     &lt;*&gt; Second extended fs support     [*]   Ext2 extended attributes     [*]   Ext2 POSIX Access Control Lists     [*]   Ext2 Security Labels     &lt;*&gt; Ext3 journalling file system support     [*]   Ext3 extended attributes     [*]   Ext3 POSIX Access Control Lists     [*]   Ext3 Security Labels     &lt;*&gt; The Extended 4 (ext4) filesystem     [*]   Ext4 POSIX Access Control Lists     [*]   Ext4 Security Labels</pre>	File System settings

Kernel Configuration Options	Description
<pre>[*] Enable the block layer ---&gt;     [*] Block layer bio throttling support</pre>	Block layer settings

Kernel Configuration Options	Description
<pre>IO Schedulers ---&gt;     &lt;*&gt; CFQ I/O scheduler     [*]  CFQ Group Scheduling support</pre>	

Kernel Configuration Options	Description
<pre>Kernel Features ---&gt;     [*] Enable seccomp to safely compute untrusted bytecode</pre>	Seccomp kernel support

### 9.3.2.4 Host Root Filesystem Configuration for Linux Containers

In order to use containers, mount the 'cgroup' pseudo-filesystem. When booting kernels compiled with cgroups support, there is a default directory for mounting them - /sys/fs/cgroup. If they are not already mounted, we will use this to mount our cgroup controllers.

```
mount -t cgroup cgroups /sys/fs/cgroup
```

## 9.3.3 More Details

### 9.3.3.1 LXC: Command Reference

This section contains links to available open source documentation for the commands in the LXC user space package.

For a description of the libvirt commands for managing containers see the chapter **Libvirt Users Guide**.

**Table 212.**

LXC man page	Description	Man Page Link
lxc	lxc overview	<a href="#">click here</a>
lxc-attach	start a process inside a running container	<a href="#">click here</a>
lxc-autostart	start/stop/kill auto-started containers	<a href="#">click here</a>
lxc-cgroup	manage the control group associated with a container	<a href="#">click here</a>
lxc-checkconfig	check the current kernel for lxc support	<a href="#">click here</a>
lxc-clone	clone a new container from an existing one	<a href="#">click here</a>
lxc-config	query LXC system configuration	<a href="#">click here</a>
lxc.conf	a description of all configuration options available	<a href="#">LXC Configuration File Reference on page 1294</a>
lxc-console	launch a console for the specified container	<a href="#">click here</a>
lxc-create	creates a container	<a href="#">click here</a>

*Table continues on the next page...*

**Table 212. (continued)**

LXC man page	Description	Man Page Link
lxc-destroy	destroy a container previously created with lxc-create	<a href="#">click here</a>
lxc-execute	run the specified command inside a container	<a href="#">click here</a>
lxc-freeze	freeze (suspend) all the container's processes	<a href="#">click here</a>
lxc-info	query information about a container	<a href="#">click here</a>
lxc-ls	list the containers existing on the system	<a href="#">click here</a>
lxc-monitor	monitor the container state	<a href="#">click here</a>
lxc-snapshot	snapshot an existing container	<a href="#">click here</a>
lxc-start	starts a container previously created with lxc-create	<a href="#">click here</a>
lxc-stop	stop a container	<a href="#">click here</a>
lxc-unfreeze	resumes a containers processes suspended previously with lxc-freeze	<a href="#">click here</a>
lxc-unshare	run a task in a new set of namespaces	<a href="#">click here</a>
lxc-usernsexec	run task as root in a new user namespace	<a href="#">click here</a>
lxc-wait	wait for a specific container state	<a href="#">click here</a>

The following LXC commands are not supported:

- lxc-usernsexec

### 9.3.3.2 LXC: Configuration Files

**NOTE**

This section is applicable to LXC only, not to libvirt.

For LXC, configuration files are used to configure aspects of a container at the time it is created. The configuration file defines what resources are private to the container and what is shared. By default the following resources are private to a container:

- process IDs
- sysv ipc mechanisms
- mount points

This means for example, that by default the container will share network resources and the filesystem with the host system, but will have it's own private process IDs.

The container configuration file allows additional isolation to be specified through configuration in the following areas:

- network
- console
- mount points and the backing store for the root filesystem
- control groups (cgroups)
- POSIX capabilities

See the [LXC Configuration File Reference](#) on page 1294 for details on each configuration option.

When a container is created a new directory with the container's name is created in `/var/lib/lxc`. The configuration file for the container is stored in:

```
/var/lib/lxc/[container-name]/config
```

Below is an example of the contents of a minimal configuration file for a container named "foo", which has no networking:

```
$ cat /var/lib/lxc/foo/config
Container with non-virtualized network
lxc.utsname = foo
lxc.tty = 1
lxc.pts = 1
lxc.rootfs = /var/lib/lxc/foo/rootfs
lxc.mount.entry=/lib /var/lib/lxc/foo/rootfs/lib none ro,bind 0 0
lxc.mount.entry=/usr/lib /var/lib/lxc/foo/rootfs/usr/lib none ro,bind 0 0
```

See the [LXC: Getting Started \(with a Busybox System Container\)](#) on page 1273 how-to article for an introduction to the container lifecycle and how configuration files are used when creating containers.

Several example configuration files are provided with LXC:

```
/usr/share/doc/lxc/examples/lxc-empty-netns.conf
/usr/share/doc/lxc/examples/lxc-complex.conf
/usr/share/doc/lxc/examples/lxc-no-netns.conf
/usr/share/doc/lxc/examples/lxc-vlan.conf
/usr/share/doc/lxc/examples/lxc-macvlan.conf
/usr/share/doc/lxc/examples/lxc-veth.conf
/usr/share/doc/lxc/examples/lxc-phys.conf
```

### 9.3.3.3 LXC: Templates

#### NOTE

This section is applicable to LXC only, not to libvirt.

For LXC, When a container is "created" a directory for the container (which has the same name as the container) is created under `/var/lib/lxc`. This is where the container's configuration file is stored and can be edited.

For system containers (containers created with **lxc-create**), the default is for the root filesystem structure of the container to be stored here as well.

Creating containers is simplified by the use of example "templates" provided with the LXC. Template examples are provided for a number of different Linux distributions. A template is a script invoked by **lxc-create** that creates the root filesystem structure and sets up the container's config file.

The following example templates are provided with LXC and can be referred to for the expected template structure:

```
/usr/share/lxc/templates/lxc-alpine
/usr/share/lxc/templates/lxc-altlinux
/usr/share/lxc/templates/lxc-archlinux
/usr/share/lxc/templates/lxc-busybox
/usr/share/lxc/templates/lxc-centos
/usr/share/lxc/templates/lxc-cirros
/usr/share/lxc/templates/lxc-debian
/usr/share/lxc/templates/lxc-download
/usr/share/lxc/templates/lxc-fedora
/usr/share/lxc/templates/lxc-gentoo
/usr/share/lxc/templates/lxc-openmandriva
/usr/share/lxc/templates/lxc-opensuse
```

```

/usr/share/lxc/templates/lxc-oracle
/usr/share/lxc/templates/lxc-plamo
/usr/share/lxc/templates/lxc-sshd
/usr/share/lxc/templates/lxc-ubuntu
/usr/share/lxc/templates/lxc-ubuntu-cloud

```

For the NXP Linux SDK for QorIQ the busybox template is recommended and has been tested with Yocto-created root filesystems.

The how-to examples provided in this user guide that create system containers use the busybox template.

### 9.3.3.4 Containers with Libvirt

This section provides an overview to using libvirt-based containers.

For an general introduction to libvirt, please see the chapter *Libvirt Users Guide*. Also, see the container information available on the libvirt website: <http://libvirt.org/drvlxc.html>.

With libvirt, a container "domain" is specified in an XML file. The XML is used to "define" the container, which then allows the container to be managed with the standard libvirt domain lifecycle.

#### Libvirt XML

The XML for the simplest functional container would look like the example below:

```

<domain type='lxc'>
 <name>container1</name>
 <memory>500000</memory>
 <os>
 <type>exe</type>
 <init>/bin/sh</init>
 </os>
 <devices>
 <console type='pty' />
 </devices>
</domain>

```

Refer to the XML reference information available on the libvirt website for detailed reference information: <http://libvirt.org/formatdomain.html>

The <domain> element must specify a type attribute of "lxc" for a container/lxc domain. There are 4 additional sub-nodes required:

- <name> - specifies the name of the container
- <memory> - specifies the maximum memory the container may use
- <os> - identifies the initial program to run. In the example this is /bin/sh. For an application based container this is the name of the application. If booting an instance of Linux user space this would typically be /sbin/init.
- <devices> - specifies any devices, in the above example there is just a console

To see a working example using this XML, see the how to article: [Basic Example](#) on page 1252.

#### Filesystem mounts (from <http://libvirt.org/drvlxc.html>)

In the absence of any explicit configuration, the container will inherit the host OS filesystem mounts. A number of mount points will be made read only, or re-mounted with new instances to provide container specific data. The following special mounts are setup by libvirt:

- /dev a new "tmpfs" pre-populated with authorized device nodes

- /dev/pts a new private "devpts" instance for console devices
- /sys the host "sysfs" instance remounted read-only
- /proc a new instance of the "proc" filesystem
- /proc/sys the host "/proc/sys" bind-mounted read-only
- /sys/fs/selinux the host "selinux" instance remounted read-only
- /sys/fs/cgroup/NNNN the host cgroups controllers bind-mounted to only expose the sub-tree associated with the container
- /proc/meminfo a FUSE backed file reflecting memory limits of the container

Additional filesystem mounts can be created using the <filesystem> node under the <devices> node. See the [libvirt.org](http://libvirt.org) documentation referenced above for further details.

**Device nodes** from <http://libvirt.org/drvlxc.html>

The container init process will be started with CAP\_MKNOD capability removed and blocked from re-acquiring it. As such it will not be able to create any device nodes in /dev or anywhere else in its filesystems. Libvirt itself will take care of pre-populating the /dev filesystem with any devices that the container is authorized to use. The current devices that will be made available to all containers are:

- /dev/zero
- /dev/null
- /dev/full
- /dev/random
- /dev/urandom
- /dev/stdin symlinked to /proc/self/fd/0
- /dev/stdout symlinked to /proc/self/fd/1
- /dev/stderr symlinked to /proc/self/fd/2
- /dev/fd symlinked to /proc/self/fd
- /dev/ptmx symlinked to /dev/pts/ptmx
- /dev/console symlinked to /dev/pts/0

### 9.3.3.5 Linux Control Groups (cgroups)

Linux control groups (or cgroups) is a feature of the Linux kernel that allows the allocation, prioritization, control, and monitoring of resources such as CPU time, memory, network bandwidth among groups of Linux processes.

Cgroups is one of the underlying Linux kernel features that LXC is built upon. LXC automatically creates a cgroup for each container when it is started. A pre-requisite for using LXC is mounting the cgroup virtual filesystem. Mounting the cgroup filesystem is presented in section [Host Root Filesystem Configuration for Linux Containers](#) on page 1267.

Cgroups encompass a number of different subsystems or "controllers" that are used for managing and controlling different resources. The following subsystems/controllers are supported:

- cpu - controls CPU allocation for tasks in a cgroup;
- cpuset - assigns individual CPUs and memory nodes to tasks in a cgroup;
- cpuacct - generates automatic reports on CPU resources used by the tasks in a cgroup;
- memory - isolates the memory behavior of a group of tasks from the rest of the system;
- devices - allows or denies access to devices by tasks in a cgroup;
- freezer - suspends or resumes tasks in a cgroup;

- `net_cls` - tags packets with a class identifier that allows the Linux traffic controller to identify packets originating from a particular cgroup;
- `net_prio` - provides a way to dynamically set the priority of network traffic per each network interface for applications within various cgroups;
- `blkio` - controls and monitors access to I/O on block devices by tasks in cgroups.

For an overview of cgroups, see the Linux kernel documentation overview here: [Documentation/cgroups/cgroups.txt](#) on page 1310.

You may also check out the Red Hat documentation on cgroups here: [https://access.redhat.com/site/documentation/en-US/Red\\_Hat\\_Enterprise\\_Linux/6/html/Resource\\_Management\\_Guide/ch-Subsystems\\_and\\_Tunable\\_Parameters.html](https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch-Subsystems_and_Tunable_Parameters.html).

Cgroup subsystems can be configured within the configuration file used when creating a container. The configuration file accepts cgroup configuration in the following form:

```
lxc.cgroup.[subsystem name] = <value>
```

See the [LXC Configuration File Reference](#) on page 1294 for further details.

Cgroup subsystems can also be displayed or updated while a container is running using the `lxc-cgroup` command:

```
lxc-cgroup -n [container-name] [cgroup-subsystem] [value]
```

For some examples of how to use cgroups to control container configuration, see the article: [LXC: How to use cgroups to manage and control a containers resources](#) on page 1285.

### 9.3.3.6 Linux Namespaces

Linux namespaces is a feature in the Linux kernel that allows one to unshare and isolate a processes' resources like UTS, PID, IPC, file system mount and network from their parent. To achieve this the kernel places the resources in different namespaces.

When LXC spawns the container's main process it unshares all these resources except the network. The network is controlled from the configuration file and is shared by default.

A network namespace provides an isolated view of the networking stack (network device interfaces, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, the `/proc/net` and `/sys/class/net` directory trees, sockets, etc.). A physical network device can live in exactly one network namespace. A virtual network device ("veth") pair provides a pipe-like abstraction that can be used to create tunnels between network namespaces, and can be used to create a bridge to a physical network device in another namespace. When a network namespace is freed (i.e., when the last process in the namespace terminates), its physical network devices are moved back to the initial network namespace (not to the parent of the process).

Each namespace is documented in the Linux `clone` man page. See: [clone \(2\)](#)

### 9.3.3.7 POSIX Capabilities

Linux supports the concept of file "capabilities" which provides fine grained control over what executable programs are permitted to do. Instead of the "all or nothing" paradigm where a super-user or "root" has the power to perform all operations, capabilities provide a mechanism to grant a specific program specific capabilities.

LXC uses this feature of the kernel to implement containers. By default processes running in a container will have **all** capabilities, but this can be configured. Capabilities can be dropped in the container's configuration file. See [LXC: Configuration Files](#) on page 1268.

For example, to drop the CAP\_SYS\_MODULE, CAP\_MKNOD, CAP\_SETUID, and CAP\_NET\_RAW capabilities, the following configuration file options would be specified:

```
lxc.cap.drop = sys_module mknod setuid net_raw
```

Each capability is documented in the Linux **capabilities** man page. See: [capabilities \(7\)](#)

In order to fully isolate a container, the capabilities to be dropped must be carefully considered. The Linux Vserver project considers only the following capabilities as **safe** for virtual private servers:

```
CAP_CHOWN
CAP_DAC_OVERRIDE
CAP_DAC_READ_SEARCH
CAP_FOWNER
CAP_FSETID
CAP_KILL
CAP_SETGID
CAP_SETUID
CAP_NET_BIND_SERVICE
CAP_SYS_CHROOT
CAP_SYS_PTRACE
CAP_SYS_BOOT
CAP_SYS_TTY_CONFIG
CAP_LEASE
```

(see: [http://linux-vserver.org/Paper#Secure\\_Capabilities](http://linux-vserver.org/Paper#Secure_Capabilities))

## 9.3.4 LXC: How To's

### 9.3.4.1 LXC: Getting Started (with a Busybox System Container)

The following article describes steps to run a simple container example. All the command below are issued from a host Linux command prompt.

1. Confirm that your kernel environment is configured correctly using **lxc-checkconfig**. All options should show as 'enabled'.

```
lxc-checkconfig
--- Namespaces ---
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
Pid namespace: enabled
User namespace: missing
Network namespace: enabled
Multiple /dev/pts instances: enabled

--- Control groups ---
Cgroup: enabled
Cgroup clone_children flag: enabled
Cgroup device: enabled
Cgroup sched: enabled
Cgroup cpu account: enabled
Cgroup memory controller: enabled
Cgroup cpuset: enabled

--- Misc ---
Veth pair device: enabled
Macvlan: enabled
```

```
Vlan: enabled
File capabilities: enabled

Note : Before booting a new kernel, you can check its configuration
usage : CONFIG=/path/to/config /usr/bin/lxc-checkconfig
```

If the cgroup namespace option shows as required:

```
Cgroup namespace: required
```

...most likely the /cgroup directory needs to be created and or mounted. See [Host Root Filesystem Configuration for Linux Containers](#) on page 1267.

NOTE: the User namespace is reported as missing. Although initial support for User Namespaces has been enabled in Linux 3.8, a lot of work still had to be done after this release and the USER\_NS config flag could not be enabled. This has no impact on the functionality of containers, since User namespace support was not implemented - just the flag was there.

## 2. Create a container

Create a system container using `lxc-create` and specify the busybox template and `lxc-empty-netns.conf` config file. `lxc-empty-netns.conf` is a simple config file with no networking:

```
lxc-create -n foo -t busybox -f /usr/share/doc/lxc/examples/lxc-empty-netns.conf
setting root password to "root"
Password for 'root' changed
#
```

By default, LXC will try to install the dropbear ssh utility, if it's available on the host system. The Busybox template also has support for installing OpenSSH (assuming it's installed on the host Linux) in the container. This needs to be passed explicitly using a command line parameter:

```
lxc-create -n foo -t busybox -f /usr/share/doc/lxc/examples/lxc-empty-netns.conf -- -s openssh
setting root password to "root"
Password for 'root' changed
'OpenSSH' ssh utility installed
#
```

## 3. List containers that exist

```
lxc-ls -l
drwxr-xr-x 3 root root 1024 May 30 15:37 foo
```

## 4. From a shell on the host Linux, start the container. When prompted, press 'Enter'.

```
lxc-start -n foo -F

Please press Enter to activate this console.

/ #

/ #
```

Note that the shell is now running within the container. Normal Linux commands can be executed.

**Important notice:** while this mode starts the container and directly connects to one of its terminals, there is a minor caveat: the terminal will be stuck in this container console until the container is halted (either from here, by running **halt**, or from another terminal by running **lxc-stop**). In order to avoid this, there is also the possibility to start the container as a daemon and connect to it using **lxc-console** (this is the default mode). This provides better terminal capabilities and the user is not forced to stop the container from another terminal. On the other hand, there is no indication that after

running **lxc-start** the container has actually started - no errors are reported. You must check if the container is running yourself, using **lxc-info** - see below.

```
lxc-start -n foo
lxc-console -n foo

Type <Ctrl+a q> to exit the console, <Ctrl+a Ctrl+a> to enter Ctrl+a itself

foo login: root
Password: (root)
~ #
~ #
~ #
~ # (Ctrl+a q)
#
```

This will be the preferred mode of starting and connecting to containers.

5. List processes in the container.

From in the container shell use the **ps** command to list processes:

```
~ # ps
 PID USER VSZ STAT COMMAND
 1 root 2384 S init
 4 root 2384 S /bin/syslogd
 6 root 2388 S -sh
 7 root 2384 S init
 8 root 2388 R ps
```

Note process IDs have a number-space unique to the container.

6. Show the status of the foo container (from a host shell):

```
lxc-info -n foo
Name: foo
State: RUNNING
PID: 4544
CPU use: 0.01 seconds
Memory use: 472.00 KiB
KMem use: 0 bytes
```

7. Look at the files/directories in **/var/lib/lxc** related to the container

```
ls -l /var/lib/lxc/foo
total 2
-rw-r--r-- 1 root root 675 May 30 15:37 config
drwxr-xr-x 16 root root 1024 May 30 15:44 rootfs
```

This shows the containers config file and rootfs backing store.

Look at the contents of the config file:

```
cat /var/lib/lxc/foo/config
Template used to create this container: /usr/share/lxc/templates/lxc-busybox
Parameters passed to the template:
For additional config options, please look at lxc.conf(5)
lxc.utsname = omega
lxc.network.type = empty
lxc.network.flags = up
lxc.rootfs = /var/lib/lxc/foo/rootfs
```

```

lxc.haltsignal = SIGUSR1
lxc.utsname = foo
lxc.tty = 1
lxc.pts = 1
lxc.cap.drop = sys_module mac_admin mac_override sys_time

When using LXC with apparmor, uncomment the next line to run unconfined:
#lxc_aa_profile = unconfined
lxc.mount.entry = /lib lib none ro,bind 0 0
lxc.mount.entry = /usr/lib usr/lib none ro,bind 0 0
lxc.mount.entry = /sys/kernel/security sys/kernel/security none ro,bind,optional 0 0
lxc.mount.auto = proc:mixed sys

```

8. Start a process inside the container using `lxc-attach`. This command will run the process inside the system container's isolated environment. The container has to be running already.

```

lxc-attach -n foo -- /bin/sh
root@foo:/# ps
PID USER TIME COMMAND
 1 root 0:00 init
 6 root 0:00 /bin/syslogd
 8 root 0:00 /bin/getty -L tty1 115200 vt100
 9 root 0:00 init
 10 root 0:00 /bin/sh
 11 root 0:00 ps
root@foo:/# ls -l /dev
total 0
crw-rw-rw- 1 root 5 136, 1 May 26 13:13 console
lrwxrwxrwx 1 root root 13 May 26 13:12 fd -> /proc/self/fd
lrwxrwxrwx 1 root root 7 May 26 13:13 kmsg -> console
srw-rw-rw- 1 root root 0 May 26 13:13 log
crw-rw-rw- 1 root root 1, 3 May 26 13:10 null
lrwxrwxrwx 1 root root 13 May 26 13:12 ptmx -> /dev/pts/ptmx
drwxr-xr-x 2 root root 0 May 26 13:13 pts
brw----- 1 root root 1, 0 May 26 13:10 ram0
drwxrwxrwt 2 root root 40 May 26 13:13 shm
lrwxrwxrwx 1 root root 15 May 26 13:12 stderr -> /proc/self/fd/2
lrwxrwxrwx 1 root root 15 May 26 13:12 stdin -> /proc/self/fd/0
lrwxrwxrwx 1 root root 15 May 26 13:12 stdout -> /proc/self/fd/1
crw-rw-rw- 1 root root 5, 0 May 26 13:10 tty
crw-rw-rw- 1 root root 4, 0 May 26 13:10 tty0
crw--w---- 1 root root 136, 0 May 26 13:13 tty1
crw-rw-rw- 1 root root 4, 0 May 26 13:10 tty5
crw-rw-rw- 1 root root 1, 9 May 26 13:10 urandom
crw-rw-rw- 1 root root 1, 5 May 26 13:10 zero
root@foo:/#

```

9. Stop the container (from a host shell)

```

lxc-stop -n foo
#
lxc-info -n foo
Name: foo
State: STOPPED

```

10. Destroy the container. This removes the containers config file and backing store.

```

lxc-destroy -n foo
#

```

### 9.3.4.2 LXC: How to configure non-virtualized networking (lxc-no-netns.conf)

One approach to providing networking capability to a container is to simply allow the container to use existing host network interfaces. To accomplish this, a configuration file is created with no networking setup (i.e. the **lxc.network.type** property is not set) and the default will be to allow the container to access the host's networking interfaces.

With this approach no network namespace is created for the container.

An example config is provided:

```
/usr/share/doc/lxc/examples/lxc-no-netns.conf
```

The contents of lxc-no-netns.conf look like this:

```
Container with non-virtualized network
lxc.network.type = none
lxc.utsname = delta
```

The example below shows starting an application container (running bash) with this config file and shows that the host network interface fm2-mac5 is inherited and accessible by the container:

```
lxc-execute -n mytest -f /usr/share/doc/lxc/examples/lxc-no-netns.conf -- /bin/bash
bash-4.3# ifconfig
fm2-mac5 Link encap:Ethernet HWaddr 00:04:9f:02:7a:3b
 inet addr:10.0.0.1 Bcast:10.255.255.255 Mask:255.0.0.0
 inet6 addr: fe80::204:9fff:fe02:7a3b/64 Scope:Link
 UP BROADCAST MULTICAST MTU:1500 Metric:1
 RX packets:0 errors:0 dropped:0 overruns:0 frame:0
 TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:1000
 RX bytes:0 (0.0 B) TX bytes:508 (508.0 B)
 Memory:fe5e8000-fe5e8fff

lo Link encap:Local Loopback
 inet addr:127.0.0.1 Mask:255.0.0.0
 inet6 addr: ::1/128 Scope:Host
 UP LOOPBACK RUNNING MTU:65536 Metric:1
 RX packets:4 errors:0 dropped:0 overruns:0 frame:0
 TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:0
 RX bytes:272 (272.0 B) TX bytes:272 (272.0 B)

bash-4.2#
```

### 9.3.4.3 LXC: How to assign a physical network interface to a container (lxc-phys.conf)

One approach to providing networking capability to a container is to directly assign an available, unused network interface to the container. The interface is not shared, it becomes the private resource of the container.

An example LXC configuration file is provided to configure this type of networking:

```
/usr/share/doc/lxc/examples/lxc-phys.conf
```

The contents of the default `lxc-phys.conf` example are show below:

```
Container with network virtualized using a physical network device with name
'eth0'
lxc.utsname = gamma
lxc.network.type = phys
lxc.network.flags = up
lxc.network.link = eth0
lxc.network.hwaddr = 4a:49:43:49:79:ff
lxc.network.ipv4 = 10.2.3.6/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3297
```

**Note:** The network type is set to: **phys**. Make a copy of the example config file and update it with the name of the Ethernet interface to be assigned, an appropriate IP address, and any other appropriate changes (e.g. mac address). For example, the change (in universal diff format) to set the interface `fm2-gb0` and IP address `192.168.10.3` would look like:

```
--- /usr/share/doc/lxc/examples/lxc-phys.conf
+++ lxc-phys.conf
@@ -3,7 +3,6 @@
 lxc.utsname = gamma
 lxc.network.type = phys
 lxc.network.flags = up
- lxc.network.link = eth0
- lxc.network.hwaddr = 4a:49:43:49:79:ff
- lxc.network.ipv4 = 10.2.3.6/24
- lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3297
+ lxc.network.link = fm2-mac5
+ lxc.network.hwaddr = 00:e0:0c:00:93:05
+ lxc.network.ipv4 = 192.168.10.3/24
```

A simple way to test the new config file and the network interface is to run `/bin/bash` as a command with `lxc-execute`, which will provide a shell running in the container:

```
lxc-execute -n mytest -f lxc-phys.conf -- /bin/bash
bash-4.2#
```

In the container, use the `fm1-gb4` interface normally:

```
bash-4.3# ifconfig
fm2-mac5 Link encap:Ethernet HWaddr 00:e0:0c:00:93:05
 inet addr:192.168.10.3 Bcast:192.168.10.255 Mask:255.255.255.0
 UP BROADCAST MULTICAST MTU:1500 Metric:1
 RX packets:0 errors:0 dropped:0 overruns:0 frame:0
 TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:1000
 RX bytes:0 (0.0 B) TX bytes:508 (508.0 B)
 Memory:fe5e8000-fe5e8fff

lo Link encap:Local Loopback
 inet addr:127.0.0.1 Mask:255.0.0.0
 inet6 addr: ::1/128 Scope:Host
 UP LOOPBACK RUNNING MTU:65536 Metric:1
 RX packets:0 errors:0 dropped:0 overruns:0 frame:0
 TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:0
 RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

bash-4.2# ping -c 3 192.168.10.1
```

```
PING 192.168.10.1 (192.168.10.1) 56(84) bytes of data.
64 bytes from 192.168.10.1: icmp_req=1 ttl=64 time=0.385 ms
64 bytes from 192.168.10.1: icmp_req=2 ttl=64 time=0.207 ms
64 bytes from 192.168.10.1: icmp_req=3 ttl=64 time=0.187 ms

--- 192.168.10.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.187/0.259/0.385/0.090 ms
```

### 9.3.4.4 LXC: How to configure networking with virtual Ethernet pairs (lxc-veth.conf)

One approach to providing a virtual network interface to a container is using the "Virtual ethernet pair device" feature of the Linux kernel in conjunction with a network bridge.

See the veth description in [LXC Configuration File Reference](#) on page 1294 for additional details on this approach to networking.

With this approach LXC creates a new network namespace for the container.

The example configuration file `lxc-veth.conf` demonstrates this approach:

```
/usr/share/doc/lxc/examples/lxc-veth.conf
```

The contents of the default `lxc-veth.conf` example are show below:

```
Container with network virtualized using a pre-configured bridge named br0 and
veth pair virtual network devices
lxc.utsname = beta
lxc.network.type = veth
lxc.network.flags = up
lxc.network.link = br0
lxc.network.hwaddr = 4a:49:43:49:79:bf
lxc.network.ipv4 = 10.2.3.5/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3597
```

Note, the network type value is: **veth** and the link property value is **br0**.

First, create a network bridge which is attached to a physical network interface and assign the bridge an IP address. The bridge becomes one endpoint In the example below the bridge `br0` is created, interface `fm2-gb1` is added to it, and the bridge is assigned an IP address of `192.168.20.2`.

```
brctl addbr br0
ifconfig br0 192.168.20.2 up
ifconfig fm2-mac5 up
brctl addif br0 fm2-mac5
```

Make a copy of the example config file and update it with an appropriate IP address and any other appropriate changes (e.g. mac address). For example, the change (in universal diff format) to update the IP address to `192.168.20.3` would look like:

```
--- /usr/share/doc/lxc/examples/lxc-veth.conf
+++ lxc-veth.conf
@@ -5,5 +5,5 @@
 lxc.network.flags = up
 lxc.network.link = br0
 lxc.network.hwaddr = 4a:49:43:49:79:bf
-lxc.network.ipv4 = 10.2.3.5/24
```

```
+lxc.network.ipv4 = 192.168.20.3/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3597
```

A simple way to test the new config file and the network interface is to run `/bin/bash` as a command with `lxc-execute`, which will provide a shell running in the container:

```
lxc-execute -n mytest -f lxc-veth.conf -- /bin/bash
bash-4.2#
```

In the container, use the virtual network interface (`eth0` in this example) normally:

```
bash-4.2# ifconfig
eth0 Link encap:Ethernet HWaddr 4a:49:43:49:79:bf
 inet addr:192.168.20.3 Bcast:192.168.20.255 Mask:255.255.255.0
 inet6 addr: fe80::4849:43ff:fe49:79bf/64 Scope:Link
 inet6 addr: 2003:db8:1:0:214:1234:fe0b:3597/64 Scope:Global
 UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
 RX packets:6 errors:0 dropped:0 overruns:0 frame:0
 TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:1000
 RX bytes:468 (468.0 B) TX bytes:586 (586.0 B)

lo Link encap:Local Loopback
 inet addr:127.0.0.1 Mask:255.0.0.0
 inet6 addr: ::1/128 Scope:Host
 UP LOOPBACK RUNNING MTU:65536 Metric:1
 RX packets:0 errors:0 dropped:0 overruns:0 frame:0
 TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:0
 RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

bash-4.2# ping -c 3 192.168.20.1
PING 192.168.20.1 (192.168.20.1) 56(84) bytes of data.
64 bytes from 192.168.20.1: icmp_req=1 ttl=64 time=0.433 ms
64 bytes from 192.168.20.1: icmp_req=2 ttl=64 time=0.221 ms
64 bytes from 192.168.20.1: icmp_req=3 ttl=64 time=0.228 ms

--- 192.168.20.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.221/0.294/0.433/0.098 ms
```

### 9.3.4.5 LXC: How to configure networking with macvlan (lxc-macvlan.conf)

An LXC container can be provided with a virtual network interface using the "MAC-VLAN" feature of the Linux kernel (see kernel config option `CONFIG_MACVLAN`). MAC-VLAN allows virtual interfaces to be created that route packets to or from a MAC address to a physical network interface.

See the macvlan description in [LXC Configuration File Reference](#) on page 1294 for some additional details on this approach to networking.

The example configuration file `lxc-veth.conf` demonstrates this approach:

```
/usr/share/doc/lxc/examples/lxc-macvlan.conf
```

The contents of the provided lxc-phys.conf example configuration file are show below:

```
Container with network virtualized using the macvlan device driver
lxc.utsname = alpha
lxc.network.type = macvlan
lxc.network.flags = up
lxc.network.link = eth0
lxc.network.hwaddr = 4a:49:43:49:79:bd
lxc.network.ipv4 = 10.2.3.4/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
```

Make a copy of the example config file and update it with the physical network interface to be used, an appropriate IP address, and any other appropriate changes (e.g. mac address). For example, the change (in universal diff format) to specify the fm1-gb4 interface and update the IP address to 192.168.1.24 would look like:

```
--- /usr/share/doc/lxc/examples/lxc-macvlan.conf
+++ lxc-macvlan.conf
@@ -2,7 +2,7 @@
 lxc.utsname = alpha
 lxc.network.type = macvlan
 lxc.network.flags = up
- lxc.network.link = eth0
+ lxc.network.link = fm2-mac5
 lxc.network.hwaddr = 4a:49:43:49:79:bd
- lxc.network.ipv4 = 10.2.3.4/24
+ lxc.network.ipv4 = 192.168.10.3/24
 lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
```

Put the network interface in promiscuous mode:

```
ifconfig fm2-mac5 promisc
ifconfig fm2-mac5
fm2-gb0 Link encap:Ethernet HWaddr 00:e0:0c:00:93:05
 inet addr:192.168.10.2 Bcast:192.168.10.255 Mask:255.255.255.0
 inet6 addr: fe80::2e0:cff:fe00:9305/64 Scope:Link
 UP BROADCAST RUNNING PROMISC MULTICAST MTU:1500 Metric:1
 RX packets:5 errors:0 dropped:0 overruns:0 frame:0
 TX packets:17 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:1000
 RX bytes:344 (344.0 B) TX bytes:1314 (1.2 KiB)
 Memory:fe5e0000-fe5e0fff
```

Test the MAC-VLAN interface by starting an application container running /bin/bash:

```
lxc-execute -n mytest -f lxc-macvlan.conf -- /bin/bash
bash-4.2#
```

Note: the shell prompt above ("bash-4.2") is in the container.

Test the interface in the now running container:

```
bash-4.2# ifconfig
eth0 Link encap:Ethernet HWaddr 4a:49:43:49:79:bd
 inet addr:192.168.10.3 Bcast:192.168.10.255 Mask:255.255.255.0
 inet6 addr: fe80::4849:43ff:fe49:79bd/64 Scope:Link
 inet6 addr: 2003:db8:1:0:214:1234:fe0b:3596/64 Scope:Global
 UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
 RX packets:0 errors:0 dropped:0 overruns:0 frame:0
```

```

TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:0 (0.0 B) TX bytes:586 (586.0 B)

lo Link encap:Local Loopback
 inet addr:127.0.0.1 Mask:255.0.0.0
 inet6 addr: ::1/128 Scope:Host
 UP LOOPBACK RUNNING MTU:65536 Metric:1
 RX packets:0 errors:0 dropped:0 overruns:0 frame:0
 TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
 collisions:0 txqueuelen:0
 RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

bash-4.2# ping -c 3 192.168.10.1
PING 192.168.10.1 (192.168.10.1) 56(84) bytes of data.
64 bytes from 192.168.10.1: icmp_req=1 ttl=64 time=0.380 ms
64 bytes from 192.168.10.1: icmp_req=2 ttl=64 time=0.204 ms
64 bytes from 192.168.10.1: icmp_req=3 ttl=64 time=0.201 ms

--- 192.168.10.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.201/0.261/0.380/0.085 ms

```

### 9.3.4.6 LXC: How to configure networking using a VLAN (lxc-vlan.conf)

A container can be provided with a virtual network interface using VLANs.

See the vlan description in [LXC Configuration File Reference](#) on page 1294 for some additional details on this approach to networking.

The example configuration file lxc-veth.conf demonstrates this approach:

```
/usr/share/doc/lxc/examples/lxc-vlan.conf
```

The contents of the provided lxc-vlan.conf example configuration file are show below:

```

Container with network virtualized using the vlan device driver
lxc.utsname = alpha
lxc.network.type = vlan
lxc.network.vlan.id = 1234
lxc.network.flags = up
lxc.network.link = eth0
lxc.network.hwaddr = 4a:49:43:49:79:bd
lxc.network.ipv4 = 10.2.3.4/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596

```

Make a copy of the example config file and update it with the physical network interface to be used and the vlan ID, an appropriate IP address, and any other appropriate changes. For example, the change (in universal diff format) to specify the fm2-gb0 interface, a VLAN id of 2, and an IP address of 192.168.30.2 would look like:

```

--- /usr/share/doc/lxc/examples/lxc-vlan.conf 2013-05-30 14:22:14.980406375 +0300
+++ lxc-vlan.conf 2013-06-03 13:26:38.477580000 +0300
@@ -1,9 +1,9 @@
Container with network virtualized using the vlan device driver
lxc.utsname = alpha
lxc.network.type = vlan

```

```
-lxc.network.vlan.id = 1234
+lxc.network.vlan.id = 2
 lxc.network.flags = up
-lxc.network.link = eth0
+lxc.network.link = fm2-mac5
 lxc.network.hwaddr = 4a:49:43:49:79:bd
-lxc.network.ipv4 = 10.2.3.4/24
+lxc.network.ipv4 = 192.168.30.2/24
 lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
```

In this setup, the host is connected to a test machine through physical interface fm2-gb0. On the test machine, the following commands have been issued (interface p7p1 on this machine has physical link to fm2-gb0):

```
[root@everest][~]# modprobe 8021q
[root@everest][~]# lsmod | grep 8021q
8021q 23476 0
garp 13763 1 8021q
[root@everest][~]# vconfig add p7p1 2
Added VLAN with VID == 2 to IF -:p7p1:-
[root@everest][~]# ifconfig p7p1.2 192.168.30.1 up
```

Test the VLAN interface by starting an application container running /bin/bash:

```
lxc-execute -n mytest -f lxc-vlan.conf -- /bin/bash
bash-4.2#
```

Test the interface in the now running container:

```
bash-4.2# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
 inet 192.168.30.2 netmask 255.255.255.0 broadcast 192.168.30.255
 inet6 fe80::21e:c9ff:fe49:bb93 prefixlen 64 scopeid 0x20<link>
 ether 00:1e:c9:49:bb:93 txqueuelen 0 (Ethernet)
 RX packets 0 bytes 0 (0.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 6 bytes 468 (468.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

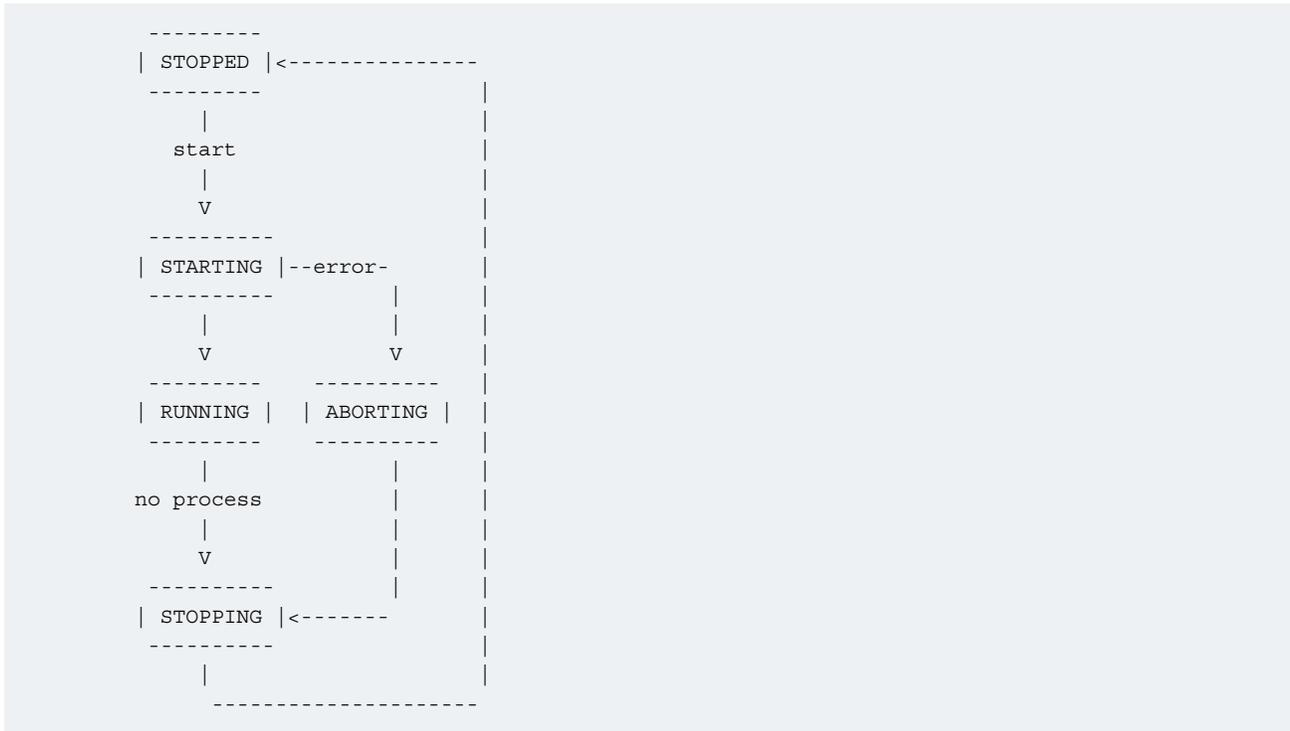
lo: flags=73<UP,LOOPBACK,RUNNING> mtu 16436
 inet 127.0.0.1 netmask 255.0.0.0
 inet6 ::1 prefixlen 128 scopeid 0x10<host>
 loop txqueuelen 0 (Local Loopback)
 RX packets 4 bytes 200 (200.0 B)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 4 bytes 200 (200.0 B)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

bash-4.2# ping -c 3 192.168.30.1
PING 192.168.30.1 (192.168.30.1) 56(84) bytes of data.
64 bytes from 192.168.30.1: icmp_req=1 ttl=64 time=0.338 ms
64 bytes from 192.168.30.1: icmp_req=2 ttl=64 time=0.372 ms
64 bytes from 192.168.30.1: icmp_req=3 ttl=64 time=0.355 ms

--- 192.168.30.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.338/0.355/0.372/0.013 ms
```

### 9.3.4.7 LXC: How to monitor containers

Containers transition through a set of well defined states. After a container is created it is in the "stopped" state.



A number of commands are available in LXC to monitor the state of a container. The following examples provide an introduction and demonstrate the capabilities of these commands.

#### 1. lxc-info

The `lxc-info` command shows the current state of the container.

In the example below, a container called "foo" has already been created but not started and the container is stopped:

```
lxc-info -n foo
Name: foo
State: STOPPED
After the container is started lxc-info shows the container in the running state:
```

```
lxc-start -n foo
lxc-info -n foo
Name: foo
State: RUNNING
PID: 5075
CPU use: 0.01 seconds
Memory use: 508.00 KiB
KMem use: 0 bytes
```

#### 2. lxc-monitor

The `lxc-monitor` command can monitor the state of one or more containers, the command continues to run until it is killed.

In this example `lxc-monitor` monitors the state of a container named "foo":

```
lxc-monitor -n foo
```

In a separate shell, start and then stop the container foo:

```
lxc-start -n foo
lxc-stop -n foo
```

The running **lxc-monitor** command displays the state changes as they occur:

```
'foo' changed state to [STARTING]
'foo' changed state to [RUNNING]
'foo' changed state to [STOPPING]
'foo' changed state to [STOPPED]
```

### 3. lxc-wait

The **lxc-wait** command will wait for a container state change and then exit. This can be useful for scripting and synchronizing the start or exit of a container.

For example, to wait until the container named "foo" stops:

```
lxc-wait -n foo -s STOPPED
```

## 9.3.4.8 LXC: How to modify the capabilities of a container to provide additional isolation

As described in [POSIX Capabilities](#) on page 1272 ,by default processes running in a container will have all capabilities. And the configuration for a container can further restrict these capabilities.

This example shows how to remove the ability for a container to issue the **mknod** command.

By default a container can issue the **mknod** command:

```
~ # mknod zero c 1 5
~ # ls -l zero
crw-r--r-- 1 root root 1, 5 Jun 3 17:08 zero
```

In this example we modify the config file of a container named "foo" (`/var/lib/lxc/foo/config`) and specify in the **lxc.cap.drop** property that the **mknod** capability (`CAP_MKNOD`) should be removed:

```
@@ -5,6 +5,7 @@
lxc.utsname = foo
lxc.tty = 1
lxc.pts = 1
+lxc.cap.drop = mknod
lxc.rootfs = /var/lib/lxc/foo/rootfs
lxc.mount.entry=/lib /var/lib/lxc/foo/rootfs/lib none ro,bind 0 0
lxc.mount.entry=/usr/lib /var/lib/lxc/foo/rootfs/usr/lib none ro,bind 0 0
```

Now restart the container and the **mknod** operation is no longer permitted:

```
~ # mknod zero c 1 5
mknod: zero: Operation not permitted
```

## 9.3.4.9 LXC: How to use cgroups to manage and control a containers resources

This example demonstrates how to use control croups to control which CPU's a container is scheduled on and the percentage of CPU time allocated to a container.

In this example we'll examine and change:

- the **cpuset** subsystem's **cpus** parameter which controls which physical CPUs the container's processes will run on
- the **cpu** subsystem's **shares** parameter which controls the percentage of the CPU to be allocated to the container

1. Start two application containers each running `/bin/bash`:

First container:

```
lxc-execute -n foo1 -f lxc-no-netns.conf -- /bin/bash
bash-4.2#
```

Second container:

```
lxc-execute -n foo2 -f lxc-no-netns.conf -- /bin/bash
bash-4.2#
```

2. In both containers start a process that will put a 100% load on the CPUs:

```
(while true; do true; done) &
```

3. The **cpuset.cpus** subsystem/value specifies which physical CPUs the container's processes run on. From a host shell, examine this with the **lxc-cgroup** command:

```
lxc-cgroup -n foo1 cpuset.cpus
0-7
```

In this example the host system has 4 CPUs.

This can also be seen directly through the `/cgroup` filesystem:

```
cat /cgroup/cpuset/lxc/foo1/cpuset.cpus
0-7
```

4. Change both containers to run only on CPU 2:

```
lxc-cgroup -n foo1 cpuset.cpus 2
lxc-cgroup -n foo2 cpuset.cpus 2
#
```

The `top` command now shows CPU 2 with 100% utilization. The `bash` commands running in each container, each have about 50% of the CPU:

```
top - 17:14:41 up 10 min, 4 users, load average: 1.64, 0.61, 0.23
Tasks: 100 total, 3 running, 97 sleeping, 0 stopped, 0 zombie
Cpu0 : 0.0%us, 0.3%sy, 0.0%ni, 99.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu1 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu2 :100.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu3 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu4 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu5 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu6 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Cpu7 : 0.0%us, 0.0%sy, 0.0%ni,100.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 3996400k total, 189836k used, 3806564k free, 1652k buffers
Swap: 0k total, 0k used, 0k free, 26180k cached

 PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
 2875 root 20 0 3624 416 164 R 50 0.0 1:28.12 bash
 2874 root 20 0 3624 424 168 R 50 0.0 1:31.06 bash
```

5. The `cpu.shares` subsystem/value specifies the percentage of the CPU allocated to the cgroup/container. By default each container has a shares value of 1024:

```
lxc-cgroup -n foo1 cpu.shares
1024
lxc-cgroup -n foo2 cpu.shares
1024
```

6. Change container "foo2" to have about 10% of the CPU:

```
lxc-cgroup -n foo2 cpu.shares 100
lxc-cgroup -n foo1 cpu.shares 900
```

Now the `top` command output reflects the new CPU allocation:

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2874	root	20	0	3624	424	168	R	90	0.0	2:53.63	bash
2875	root	20	0	3624	416	164	R	10	0.0	2:11.36	bash

7. Stop the containers

```
lxc-stop -n foo1 -k
lxc-stop -n foo2 -k
#
```

### 9.3.4.10 LXC: How to run an application in a container with `lxc-execute`

The **`lxc-execute`** command allows a single application to be run in a container (as contrasted with a system container which boots an instance of Linux user space starting with System V style `init`).

In the example below an instance of a QEMU/KVM virtual machine is started in a container called `foo`.

Note, it is not required to explicitly create (and destroy) a container when running application containers with `lxc-execute`. The containers will automatically created and destroyed.

1. Start QEMU in the container with `lxc-execute`:

```
lxc-execute -n foo -f lxc-no-netns.conf -- qemu-system-ppc -enable-kvm -smp 2 -m 256M -nographic
-M ppce500 -kernel uImage -initrd rootfs.ext2.gz -append "root=/dev/ram rw console=ttyS0,115200" -
serial tcp::4445,server,telnet
```

NOTE: For 64bit platforms, please replace `qemu-system-ppc` with `qemu-system-ppc64`.

Some notes:

- The QEMU command line follows the double dash ("`--`") specified on the `lxc-execute` command line and distinguishes argument to `lxc-execute` from arguments to `qemu-system-ppc`.
- Using the specified configuration file, QEMU will run in the network namespace of the host system, meaning the TCP ports for serial and the monitor (ports 4445 and 4446) can be accessed from the host. However, `lxc-execute` will accept a configuration file as an argument allowing customization of the degree of isolation of the container.
- In this example there are 2 virtual cpus specified, which results in a total of 3 QEMU processes/threads. So we expect to see 3 QEMU processes in the container.

2. Examine the state of the container with `lxc-ls` and `lxc-info`:

```
lxc-ls --active
foo

lxc-info -n foo
Name: foo
State: RUNNING
PID: 3205
IP: 192.168.2.80
CPU use: 3.96 seconds
Memory use: 140.46 MiB
KMem use: 0 bytes
```

## 3. In the QEMU console look at the CPU status which shows the process IDs for the two virtual CPUs in in the virtual machine:

```
(qemu) info cpus
* CPU #0: nip=0x00000000c001450c thread_id=4
 CPU #1: nip=0x00000000c001450c thread_id=5
(qemu)
```

Note that the process/thread IDs as viewed from within the container (thread IDs 4 and 5) are different than from the host, since they are in a different namespace.

## 5. Using the container's cgroup restrict the physical CPUs on which the virtual machine is allowed to run.

By default all 4 CPUs can be used by the container :

```
by default all 4 CPUs can be used by the container
cat /cgroup/lxc/foo/cpuset.cpus
0-3
```

Restrict the containers processes to CPUs 2 and 3:

```
echo 2-3 > /cgroup/lxc/foo/cpuset.cpus
cat /cgroup/lxc/foo/cpuset.cpus
2-3
```

### 9.3.4.11 LXC: How to run an unprivileged container

With the addition of the user namespace in the Linux kernel, a normal user on a Linux host can create and run container instances. This feature has been integrated in the LXC package, starting from version 1.0.

The steps below detail the necessary steps required in order to configure and manage an unprivileged container.

**NOTE:** Before running these steps, make sure that the host is properly configured for container use, by running `lxc-checkconfig` (cgroups, namespaces, etc.). If some of the options are missing, please refer to the guidelines in [LXC: Build, Installation, Configuration](#) on page 1264.

1. Create the `/etc/subuid` and `/etc/subgid` file on the Linux host. These will be used to store the unprivileged user's subordinate UIDs and GIDs. The unprivileged user has the ability to manage users on his own in his user namespace, and their IDs will be mapped to corresponding ranges on IDs on the host system. The subordinate IDs will correspond to the ranges defined in these files.

```
for file in '/etc/subuid' '/etc/subgid'; do
 touch $file
 chown root:root $file
```

```
chmod 644 $file
done
```

2. Add a user in the system - **lxc-user**.

```
useradd lxc-user -p $(echo test | openssl passwd -1 -stdin)
```

3. Check the contents of **/etc/subuid** and **/etc/subgid**. If they contain the following entries, the user has been automatically assigned a default set of subordinate IDs.

```
root@t4240qds:~# cat /etc/sub*
lxc-user:100000:65536
lxc-user:100000:65536
root@t4240qds:~#
```

If the files are empty, you need to manually assign a set of subordinate IDs to the user.

```
usermod --add-subuids 100000-165536 lxc-user
usermod --add-subgids 100000-165536 lxc-user
```

4. The container will have a virtual interface linked to a bridge on the host. Use the following command to create the bridge.

```
brctl addbr br0 && ifconfig br0 10.0.0.1
```

5. You must create and edit the **/etc/lxc/lxc-usernet** file. This file specifies how many interfaces the lxc-user will be allowed to have linked in this bridge.

```
echo "lxc-user veth br0 10" > /etc/lxc/lxc-usernet
```

6. Create the **/home/lxc-user/.config/lxc** directory on the host. This will hold the default configuration for unprivileged containers belonging to the lxc-user.

```
mkdir -p /home/lxc-user/.config/lxc
```

7. **Create** the default container configuration file, **/home/lxc-user/.config/lxc/default.conf**, and paste the following lines.

```
lxc.network.type = veth
lxc.network.link = br0
lxc.network.flags = up
lxc.id_map = u 0 100000 65536
lxc.id_map = g 0 100000 65536
```

8. Change the ownership of the newly created files and folders to lxc-user.

```
chown -R lxc-user:lxc-user /home/lxc-user/.config
```

9. For each of the mounted cgroup controllers, created a directory in the top called **lxc-user**, and change its ownership to lxc-user. Be sure to enable the **cgroup.clone\_children** and **memory.use\_hierarchy** flags.

```
echo 1 > /sys/fs/cgroup/memory/memory.use_hierarchy

for c in `ls /sys/fs/cgroup/`; do
 echo 1 > /sys/fs/cgroup/$c/cgroup.clone_children
 mkdir /sys/fs/cgroup/$c/lxc-user
 chown -R lxc-user:lxc-user /sys/fs/cgroup/$c/lxc-user
done
```

**10. Login as the new user in a new console.**

```
t4240qds login: lxc-user
Password:
t4240qds:~$
```

**11. Copy the shell PID in the lxc-user cgroups.**

```
for c in `ls /sys/fs/cgroup/`; do
 echo $$ > /sys/fs/cgroup/$c/lxc-user/tasks
done
```

**12. From the same shell as before, create a Busybox container. You can pass it a custom config file using the `-f` cmdline parameter. Otherwise, it will pick the default config from `/home/lxc-user.config/default.conf`.**

```
t4240qds:~$ lxc-create -n foo -t busybox
setting root password to "root"
Password for 'root' changed
t4240qds:~$
```

**13. Start the container.**

```
t4240qds:~$ lxc-start -n foo -F

Please press Enter to activate this console.
/ #
/ #
/ # whoami
root
/ #
```

Now you can interact with the container as you would with one created by root. **Make sure that all container commands are run as lxc-user.**

## 9.3.4.12 LXC: How to run containers with Seccomp protection

A large number of system calls are exposed to every userland process with many of them going unused for the entire lifetime of the process. As system calls change and mature, bugs are found and eradicated. A certain subset of userland applications benefit by having a reduced set of available system calls. The resulting set reduces the total kernel surface exposed to the application. System call filtering is meant for use with those applications.

Seccomp (short for *secure compute*) is a system call filtering mechanism present in the kernel. **Initially** it has been thought to be a sandboxing mechanism that would allow userspace processes to issue a very limited set of system calls - `read()`, `write()`, `exit()` and `sigreturn()`. This has been further known to be **seccomp mode 1**, and while it is strong on the security side, it doesn't leave much room for flexibility.

The next addition to seccomp was to allow **filtering** (or **seccomp mode 2**) based on the kernel **BPF** (Berkeley Packet Filter) infrastructure. This allows the system administrator to define complex and granular policies, per system call and its arguments. This is an extension to the BPF mechanism, that allows filtering to apply to system call numbers and their arguments, besides its original purpose (socket packets). The defined filter results in a seccomp policy which is attached to the userspace process in the form of a BPF program. Each time the process issues a system call, it is checked against this policy in order to determine how it will be handled:

- **SECCOMP\_RET\_KILL** - the task exits immediately without executing the system call.
- **SECCOMP\_RET\_TRAP** - the kernel sends a SIGSYS to the triggering task without executing the system call.
- **SECCOMP\_RET\_ERRNO** - a custom errno is returned to userspace without executing the system call.

- **SECCOMP\_RET\_TRACE** - causes the kernel to attempt to notify a ptrace-based tracer prior to executing the system call. The tracer can skip the system call or change it to a valid syscall number.
- **SECCOMP\_RET\_ALLOW** - results in the system call being executed.

In order to make the secure computing facility more userspace-friendly, the [libseccomp](#) library has been developed, which is meant to make it easier for applications to take advantage of the packet-filter-based seccomp mode. Prior to this, userspace applications had to [define the BPF filter themselves](#). libseccomp restructures this approach into [a simple and straightforward API](#) which userspace applications can use. [The latest version of libseccomp](#) adds support for Python bindings as well, and is designed to work on multiple architectures (ARM, MIPS). [PowerPC support has also been merged](#) on a separate branch, and is expected to be included in future releases.

### Using seccomp with LXC containers

Please refer to [Building LXC](#) for information on how to build LXC with seccomp support in the SDK.

**Note:** Currently LXC seccomp support is not available for ARM64 architectures.

Seccomp filtering integrates well with processes sandboxed as containers, as they can be assigned to untrusted users and exposed with a limited set of allowed system calls. This is a portable and granular low-level security mechanism which can be used to increase container security. The seccomp policy file needs to be applied only to the init process in the container, and will be inherited by all its children.

The seccomp policy for the container is specified using the [container configuration file](#), in the form of a single line containing:

```
lxc.seccomp = /var/lib/lxc/lxc_seccomp.conf
```

An example lxc\_seccomp policy file can look as follows:

```
2
blacklist
[ppc64]
mknod errno 120
sched_setscheduler trap
fchmodat kill
[ppc]
mknod
```

The elements in the policy file represent the following:

1. Version number (1/2) - a single integer containing a single number, 1 or 2. Version 1 only allows to define a set of system calls which are allowed (whitelisted) in the container, specified by syscall number. This version is limited in configurability and portability, since it's only used to specify allowed syscall numbers, which may differ from arch to arch. Version 2 allows the policies to be either a whitelist (default deny, except mentioned syscalls) or a blacklist (default allow, except mentioned syscalls), and the syscalls can be expressed by name.
2. Policy type (whitelist/blacklist) - with an option of a default policy action (errno #, trap, kill, allow). The policy type is per seccomp context, and can be either whitelist or blacklist, not both.
3. Architecture tag [optional] - mentions that the following set of system calls will only be applied to a specific architecture. There can be multiple architecture tags and associated syscalls. These tags allow the same seccomp policy file to be used on multiple platforms, treating each one differently with respect to the set of system calls.
4. System calls - which can be expressed by number (in version 1) or name (in version 2). Optionally, an action can be expressed after the system call (errno #, trap, kill, allow), specifying the desired seccomp behavior. If this is omitted, the default rule action of the policy will be applied (allow for whitelist policies, kill for blacklist policies).

When running a container with the previous policy file on a PowerPC 64-bit platform, the mknod, sched\_setscheduler (chrt) and fchmodat (chmod) system calls will be denied, with mentioned behaviors: mknod will return errno 120 without executing,

chrt will trap and chmod will result in the process executing it being killed. On PowerPC platforms, only mknod will be denied, resulting in the process being killed. All other system calls will be allowed.

#### Notes:

- Containers can still be started without loading a seccomp policy file, simply by omitting the lxc.seccomp line in the config file. No seccomp policy is loaded by default.
- If a container process has a seccomp policy loaded, this can be seen in /proc/PID/status, on the seccomp line. This line will contain "Seccomp: 2" when using seccomp filter (mode 2). "Seccomp: 0" means there is no seccomp policy in effect.
- Seccomp policies of a process are automatically inherited by its children.
- Currently LXC supports only system call based filtering, with no support for system call arguments.
- The performance degradation of the processes running with a seccomp policy applied is directly proportional with the policy file size: normally, the system calls are listed as rules in the BPF filter program, and they all need to be parsed and matched at each system call. The longer the list, the more time this will take.
- The LXC package comes shipped with a set of example policy files which can be found at /share/doc/lxc/examples/seccomp-\*. There's also a policy file, [common.seccomp](#), which denies common security syscall threats in the container, such as kernel module manipulation, kexec and open\_by\_handle\_at (the vector for the [Shocker exploit](#)).

## 9.3.5 Libvirt How To's

### 9.3.5.1 Basic Example

The following example shows the lifecycle of a simple LXC libvirt domain called *container1*. The virsh tool is used for managing the lxc domain lifecycle.

**1. Confirm the host Linux configuration.** Begin by confirming that the host kernel is configured correctly and that rootfs setup such as mounting cgroups has been done. This can be done with the lxc-checkconfig command.

```
lxc-checkconfig
--- Namespaces ---
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
Pid namespace: enabled
User namespace: missing
Network namespace: enabled
Multiple /dev/pts instances: enabled

--- Control groups ---
Cgroup: enabled
Cgroup clone_children flag: enabled
Cgroup device: enabled
Cgroup sched: enabled
Cgroup cpu account: enabled
Cgroup memory controller: enabled
Cgroup cpuset: enabled

--- Misc ---
Veth pair device: enabled
Macvlan: enabled
Vlan: enabled
File capabilities: enabled
```

2. **Create a libvirt XML file defining the container.** The example below shows a very simple container defined in `container1.xml` that runs the command `/bin/sh` and has a console:

```
cat container1.xml
<domain type='lxc'>
 <name>container1</name>
 <memory>500000</memory>
 <os>
 <type>exe</type>
 <init>/bin/sh</init>
 </os>
 <devices>
 <console type='pty' />
 </devices>
</domain>
```

3. **Define the container.** The `virsh define` command processes the XML and makes creates the new libvirt domain.

```
virsh -c lxc:/// define container1.xml
Domain container1 defined from container1.xml
```

```
virsh -c lxc:/// list --all
 Id Name State

 - container1 shut off
```

4. **Start the container.**

```
virsh -c lxc:/// start container1
Domain container1 started
```

```
virsh -c lxc:/// list
 Id Name State

 3196 container1 running
```

5. **Connect to the console.**

```
virsh -c lxc:/// console container1
Connected to domain container1
Escape character is ^]
sh-4.2#
sh-4.2# ps -ef
UID PID PPID C STIME TTY TIME CMD
root 1 0 0 18:25 pts/2 00:00:00 /bin/sh
root 3 1 0 18:36 pts/2 00:00:00 ps -ef

sh-4.2# ifconfig br0
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
 inet 10.171.73.123 netmask 255.255.254.0 broadcast 10.171.73.255
 inet6 fe80::a00:27ff:fe01:fe07 prefixlen 64 scopeid 0x20<link>
 ether 08:00:27:01:fe:07 txqueuelen 0 (Ethernet)
 RX packets 865838 bytes 104029354 (99.2 MiB)
 RX errors 0 dropped 0 overruns 0 frame 0
 TX packets 104446 bytes 43998714 (41.9 MiB)
 TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

```
sh-4.2#
```

Press CTRL + ] to exit the console.

The following aspects must be noted:

- the processes inside the container are running in a separate namespace, hence the different process hierarchy
- since no network configuration for the domain is explicitly specified, all networking interfaces are shared with the host (all the other interfaces are present too - br0 is mentioned as an example)
- since no filesystem configuration is specified for the domain, the filesystem is shared with the host-- all host mounts are present in the container as well.

6. To stop the container use the destroy command:

```
virsh -c lxc:/// destroy container1
Domain container1 destroyed

virsh -c lxc:/// list --all
 Id Name State

 - container1 shut off
```

7. To remove the domain from libvirt, use the undefine command.

```
virsh -c lxc:/// undefine container1
Domain container1 has been undefined
```

## 9.3.6 Appendix

### 9.3.6.1 LXC Configuration File Reference

The text below comes from the lxc.conf.5 man page:

```
LXC.CONF(5)
NAME

 lxc.conf - Configuration files for LXC.
DESCRIPTION

 LXC configuration is split in two parts. Container configuration and
 system configuration.

CONTAINER CONFIGURATION

 The container configuration is held in the config stored in the
 container's directory.

 A basic configuration is generated at container creation time with
 the default's recommended for the chosen template as well as extra
 default keys coming from the default.conf file.

 That default.conf file is either located at
 /usr/local/etc/lxc/default.conf or for unprivileged containers at
 ~/.config/lxc/default.conf.
```

Details about the syntax of this file can be found in:  
lxc.container.conf(5)

#### SYSTEM CONFIGURATION

The system configuration is located at /usr/local/etc/lxc/lxc.conf or  
~/.config/lxc/lxc.conf for unprivileged containers.

This configuration file is used to set values such as default lookup  
paths and storage backend settings for LXC.

Details about the syntax of this file can be found in:  
lxc.system.conf(5)

#### SEE ALSO

lxc(1), lxc.container.conf(5), lxc.system.conf(5), lxc-usernet(5)

#### AUTHOR

Stéphane Graber <stgraber@ubuntu.com>

#### COLOPHON

This page is part of the lxc (Linux containers) project. Information  
about the project can be found at <http://linuxcontainers.org/>. If  
you have a bug report for this manual page, send it to  
lxc-devel@lists.linuxcontainers.org. This page was obtained from the  
project's upstream Git repository ([git://github.com/lxc/lxc](https://github.com/lxc/lxc)) on  
2014-05-21. If you discover any rendering problems in this HTML  
version of the page, or you believe there is a better or more up-to-  
date source for the page, or you have corrections or improvements to  
the information in this COLOPHON (which is not part of the original  
manual page), send a mail to [man-pages@man7.org](mailto:man-pages@man7.org)

Wed May 21 10:30:15 CEST 2014

LXC.CONF(5)

The text below comes from the lxc.container.conf.5 man page:

LXC.CONTAINER.CONF(5)

LXC.CONTAINER.CONF(5)

#### NAME

lxc.container.conf - LXC container configuration file

#### DESCRIPTION

The linux containers (lxc) are always created before being used. This  
creation defines a set of system resources to be virtualized /  
isolated when a process is using the container. By default, the pids,  
sysv ipc and mount points are virtualized and isolated. The other  
system resources are shared across containers, until they are  
explicitly defined in the configuration file. For example, if there  
is no network configuration, the network will be shared between the  
creator of the container and the container itself, but if the network  
is specified, a new network stack is created for the container and  
the container can no longer use the network of its ancestor.

The configuration file defines the different system resources to be  
assigned for the container. At present, the utsname, the network, the  
mount points, the root file system, the user namespace, and the  
control groups are supported.

Each option in the configuration file has the form key = value  
fitting in one line. The '#' character means the line is a comment.

**CONFIGURATION**

In order to ease administration of multiple related containers, it is possible to have a container configuration file cause another file to be loaded. For instance, network configuration can be defined in one common file which is included by multiple containers. Then, if the containers are moved to another host, only one file may need to be updated.

**lxc.include**

Specify the file to be included. The included file must be in the same valid lxc configuration file format.

**ARCHITECTURE**

Allows one to set the architecture for the container. For example, set a 32bits architecture for a container running 32bits binaries on a 64bits host. This fixes the container scripts which rely on the architecture to do some work like downloading the packages.

**lxc.arch**

Specify the architecture for the container.

Valid options are x86, i686, x86\_64, amd64

**HOSTNAME**

The utsname section defines the hostname to be set for the container. That means the container can set its own hostname without changing the one from the system. That makes the hostname private for the container.

**lxc.utsname**

specify the hostname for the container

**HALT SIGNAL**

Allows one to specify signal name or number, sent by lxc-stop to the container's init process to cleanly shutdown the container. Different init systems could use different signals to perform clean shutdown sequence. This option allows the signal to be specified in kill(1) fashion, e.g. SIGPWR, SIGRTMIN+14, SIGRTMAX-10 or plain number. The default signal is SIGPWR.

**lxc.haltsignal**

specify the signal used to halt the container

**STOP SIGNAL**

Allows one to specify signal name or number, sent by lxc-stop to forcibly shutdown the container. This option allows signal to be specified in kill(1) fashion, e.g. SIGKILL, SIGRTMIN+14, SIGRTMAX-10 or plain number. The default signal is SIGKILL.

**lxc.stopsignal**

specify the signal used to stop the container

**NETWORK**

The network section defines how the network is virtualized in the container. The network virtualization acts at layer two. In order to use the network virtualization, parameters must be specified to define the network interfaces of the container. Several virtual interfaces can be assigned and used in a container even if the system has only one physical network interface.

#### `lxc.network.type`

specify what kind of network virtualization to be used for the container. Each time a `lxc.network.type` field is found a new round of network configuration begins. In this way, several network virtualization types can be specified for the same container, as well as assigning several network interfaces for one container. The different virtualization types can be:

`none`: will cause the container to share the host's network namespace. This means the host network devices are usable in the container. It also means that if both the container and host have `upstart` as `init`, `'halt'` in a container (for instance) will shut down the host.

`empty`: will create only the loopback interface.

`veth`: a peer network device is created with one side assigned to the container and the other side is attached to a bridge specified by the `lxc.network.link`. If the bridge is not specified, then the veth pair device will be created but not attached to any bridge. Otherwise, the bridge has to be setup before on the system, `lxc` won't handle any configuration outside of the container. By default `lxc` choose a name for the network device belonging to the outside of the container, this name is handled by `lxc`, but if you wish to handle this name yourself, you can tell `lxc` to set a specific name with the `lxc.network.veth.pair` option.

`vlan`: a vlan interface is linked with the interface specified by the `lxc.network.link` and assigned to the container. The vlan identifier is specified with the option `lxc.network.vlan.id`.

`macvlan`: a macvlan interface is linked with the interface specified by the `lxc.network.link` and assigned to the container. `lxc.network.macvlan.mode` specifies the mode the macvlan will use to communicate between different macvlan on the same upper device. The accepted modes are `private`, the device never communicates with any other device on the same `upper_dev` (default), `vepa`, the new Virtual Ethernet Port Aggregator (VEPA) mode, it assumes that the adjacent bridge returns all frames where both source and destination are local to the macvlan port, i.e. the bridge is set up as a reflective relay. Broadcast frames coming in from the `upper_dev` get flooded to all macvlan interfaces in VEPA mode, local frames are not delivered locally, or `bridge`, it provides the behavior of a simple bridge between different macvlan interfaces on the same port. Frames from one interface to another one get delivered directly and are not sent out externally. Broadcast frames get flooded to all other bridge ports and to the external interface, but when they come back from a reflective relay, we don't deliver them again. Since we know all the MAC addresses, the macvlan bridge mode does not require learning or STP like the bridge module does.

`phys`: an already existing interface specified by the `lxc.network.link` is assigned to the container.

#### `lxc.network.flags`

specify an action to do for the network.

up: activates the interface.

`lxc.network.link`

specify the interface to be used for real network traffic.

`lxc.network.mtu`

specify the maximum transfer unit for this interface.

`lxc.network.name`

the interface name is dynamically allocated, but if another name is needed because the configuration files being used by the container use a generic name, eg. eth0, this option will rename the interface in the container.

`lxc.network.hwaddr`

the interface mac address is dynamically allocated by default to the virtual interface, but in some cases, this is needed to resolve a mac address conflict or to always have the same link-local ipv6 address. Any "x" in address will be replaced by random value, this allows setting hwaddr templates.

`lxc.network.ipv4`

specify the ipv4 address to assign to the virtualized interface. Several lines specify several ipv4 addresses. The address is in format x.y.z.t/m, eg. 192.168.1.123/24. The broadcast address should be specified on the same line, right after the ipv4 address.

`lxc.network.ipv4.gateway`

specify the ipv4 address to use as the gateway inside the container. The address is in format x.y.z.t, eg. 192.168.1.123. Can also have the special value auto, which means to take the primary address from the bridge interface (as specified by the `lxc.network.link` option) and use that as the gateway. auto is only available when using the veth and macvlan network types.

`lxc.network.ipv6`

specify the ipv6 address to assign to the virtualized interface. Several lines specify several ipv6 addresses. The address is in format x::y/m, eg.  
2003:db8:1:0:214:1234:fe0b:3596/64

`lxc.network.ipv6.gateway`

specify the ipv6 address to use as the gateway inside the container. The address is in format x::y, eg. 2003:db8:1:0::1  
Can also have the special value auto, which means to take the primary address from the bridge interface (as specified by the `lxc.network.link` option) and use that as the gateway. auto is only available when using the veth and macvlan network types.

`lxc.network.script.up`

add a configuration option to specify a script to be executed after creating and configuring the network used from the host side. The following arguments are passed to the script:  
container name and config section name (net) Additional arguments depend on the config section employing a script hook; the following are used by the network system: execution

context (up), network type (empty/veth/macvlan/phys), Depending on the network type, other arguments may be passed: veth/macvlan/phys. And finally (host-sided) device name.

Standard output from the script is logged at debug level. Standard error is not logged, but can be captured by the hook redirecting its standard error to standard output.

#### `lxc.network.script.down`

add a configuration option to specify a script to be executed before destroying the network used from the host side. The following arguments are passed to the script: container name and config section name (net) Additional arguments depend on the config section employing a script hook; the following are used by the network system: execution context (down), network type (empty/veth/macvlan/phys), Depending on the network type, other arguments may be passed: veth/macvlan/phys. And finally (host-sided) device name.

Standard output from the script is logged at debug level. Standard error is not logged, but can be captured by the hook redirecting its standard error to standard output.

#### NEW PSEUDO TTY INSTANCE (DEVPTS)

For stricter isolation the container can have its own private instance of the pseudo tty.

#### `lxc.pts`

If set, the container will have a new pseudo tty instance, making this private to it. The value specifies the maximum number of pseudo ttys allowed for a pts instance (this limitation is not implemented yet).

#### CONTAINER SYSTEM CONSOLE

If the container is configured with a root filesystem and the inittab file is setup to use the console, you may want to specify where the output of this console goes.

#### `lxc.console`

Specify a path to a file where the console output will be written. The keyword 'none' will simply disable the console. This is dangerous once if have a rootfs with a console device file where the application can write, the messages will fall in the host.

#### CONSOLE THROUGH THE TTYS

This option is useful if the container is configured with a root filesystem and the inittab file is setup to launch a getty on the ttys. The option specifies the number of ttys to be available for the container. The number of gettys in the inittab file of the container should not be greater than the number of ttys specified in this option, otherwise the excess getty sessions will die and respawn indefinitely giving annoying messages on the console or in /var/log/messages.

#### `lxc.tty`

Specify the number of tty to make available to the container.

#### CONSOLE DEVICES LOCATION

LXC consoles are provided through Unix98 PTYs created on the host and

bind-mounted over the expected devices in the container. By default, they are bind-mounted over `/dev/console` and `/dev/ttyN`. This can prevent package upgrades in the guest. Therefore you can specify a directory location (under `/dev` under which LXC will create the files and bind-mount over them. These will then be symbolically linked to `/dev/console` and `/dev/ttyN`. A package upgrade can then succeed as it is able to remove and replace the symbolic links.

#### `lxc.devtttydir`

Specify a directory under `/dev` under which to create the container console devices.

#### `/DEV DIRECTORY`

By default, `lxc` creates a few symbolic links (`fd,stdin,stdout,stderr`) in the container's `/dev` directory but does not automatically create device node entries. This allows the container's `/dev` to be set up as needed in the container rootfs. If `lxc.autodev` is set to 1, then after mounting the container's rootfs LXC will mount a fresh `tmpfs` under `/dev` (limited to 100k) and fill in a minimal set of initial devices. This is generally required when starting a container containing a "systemd" based "init" but may be optional at other times. Additional devices in the containers `/dev` directory may be created through the use of the `lxc.hook.autodev` hook.

#### `lxc.autodev`

Set this to 1 to have LXC mount and populate a minimal `/dev` when starting the container.

#### `ENABLE KMSG SYMLINK`

Enable creating `/dev/kmsg` as symlink to `/dev/console`. This defaults to 1.

#### `lxc.kmsg`

Set this to 0 to disable `/dev/kmsg` symlinking.

#### `MOUNT POINTS`

The mount points section specifies the different places to be mounted. These mount points will be private to the container and won't be visible by the processes running outside of the container. This is useful to mount `/etc`, `/var` or `/home` for examples.

#### `lxc.mount`

specify a file location in the `fstab` format, containing the mount information. The mount target location can and in most cases should be a relative path, which will become relative to the mounted container root. For instance,

```
proc proc proc nodev,noexec,nosuid 0 0
```

Will mount a `proc` filesystem under the container's `/proc`, regardless of where the root filesystem comes from. This is resilient to block device backed filesystems as well as container cloning.

Note that when mounting a filesystem from an image file or block device the third field (`fs_vfstype`) cannot be `auto` as with `mount(8)` but must be explicitly specified.

#### `lxc.mount.entry`

specify a mount point corresponding to a line in the `fstab`

format.

#### lxc.mount.auto

specify which standard kernel file systems should be automatically mounted. This may dramatically simplify the configuration. The file systems are:

- `proc:mixed` (or `proc`): mount `/proc` as read-write, but remount `/proc/sys` and `/proc/sysrq-trigger` read-only for security / container isolation purposes.
- `proc:rw`: mount `/proc` as read-write
- `sys:ro` (or `sys`): mount `/sys` as read-only for security / container isolation purposes.
- `sys:rw`: mount `/sys` as read-write
- `cgroup:mixed`: mount a tmpfs to `/sys/fs/cgroup`, create directories for all hierarchies to which the container is added, create subdirectories there with the name of the cgroup, and bind-mount the container's own cgroup into that directory. The container will be able to write to its own cgroup directory, but not the parents, since they will be remounted read-only
- `cgroup:ro`: similar to `cgroup:mixed`, but everything will be mounted read-only.
- `cgroup:rw`: similar to `cgroup:mixed`, but everything will be mounted read-write. Note that the paths leading up to the container's own cgroup will be writable, but will not be a cgroup filesystem but just part of the tmpfs of `/sys/fs/cgroup`
- `cgroup` (without specifier): defaults to `cgroup:rw` if the container retains the `CAP_SYS_ADMIN` capability, `cgroup:mixed` otherwise.
- `cgroup-full:mixed`: mount a tmpfs to `/sys/fs/cgroup`, create directories for all hierarchies to which the container is added, bind-mount the hierarchies from the host to the container and make everything read-only except the container's own cgroup. Note that compared to `cgroup`, where all paths leading up to the container's own cgroup are just simple directories in the underlying tmpfs, here `/sys/fs/cgroup/$hierarchy` will contain the host's full cgroup hierarchy, albeit read-only outside the container's own cgroup. This may leak quite a bit of information into the container.
- `cgroup-full:ro`: similar to `cgroup-full:mixed`, but everything will be mounted read-only.
- `cgroup-full:rw`: similar to `cgroup-full:mixed`, but everything will be mounted read-write. Note that in this case, the container may escape its own cgroup. (Note also that if the container has `CAP_SYS_ADMIN` support and can mount the cgroup filesystem itself, it may do so anyway.)

- `cgroup-full` (without specifier): defaults to `cgroup-full:rw` if the container retains the `CAP_SYS_ADMIN` capability, `cgroup-full:mixed` otherwise.

Note that if automatic mounting of the `cgroup` filesystem is enabled, the `tmpfs` under `/sys/fs/cgroup` will always be mounted read-write (but for the `:mixed` and `:ro` cases, the individual hierarchies, `/sys/fs/cgroup/$hierarchy`, will be read-only). This is in order to work around a quirk in Ubuntu's `mountall(8)` command that will cause containers to wait for user input at boot if `/sys/fs/cgroup` is mounted read-only and the container can't remount it read-write due to a lack of `CAP_SYS_ADMIN`.

Examples:

```
lxc.mount.auto = proc sys cgroup
lxc.mount.auto = proc:rw sys:rw cgroup-full:rw
```

#### ROOT FILE SYSTEM

The root file system of the container can be different than that of the host system.

##### `lxc.rootfs`

specify the root file system for the container. It can be an image file, a directory or a block device. If not specified, the container shares its root file system with the host.

For directory or simple block-device backed containers, a pathname can be used. If the `rootfs` is backed by a `nbd` device, then `nbd:file:1` specifies that file should be attached to a `nbd` device, and partition 1 should be mounted as the `rootfs`. `nbd:file` specifies that the `nbd` device itself should be mounted. `overlayfs:/lower:/upper` specifies that the `rootfs` should be an overlay with `/upper` being mounted read-write over a read-only mount of `/lower`. `aufs:/lower:/upper` does the same using `aufs` in place of `overlayfs`. `loop:/file` tells `lxc` to attach `/file` to a `loop` device and mount the `loop` device.

##### `lxc.rootfs.mount`

where to recursively bind `lxc.rootfs` before pivoting. This is to ensure success of the `pivot_root(8)` syscall. Any directory suffices, the default should generally work.

##### `lxc.rootfs.options`

extra mount options to use when mounting the `rootfs`.

##### `lxc.pivotdir`

where to pivot the original root file system under `lxc.rootfs`, specified relatively to that. The default is `mnt`. It is created if necessary, and also removed after unmounting everything from it during container setup.

#### CONTROL GROUP

The control group section contains the configuration for the different subsystem. `lxc` does not check the correctness of the subsystem name. This has the disadvantage of not detecting configuration errors until the container is started, but has the advantage of permitting any future subsystem.

```
lxc.cgroup.[subsystem name]
```

specify the control group value to be set. The subsystem name is the literal name of the control group subsystem. The permitted names and the syntax of their values is not dictated by LXC, instead it depends on the features of the Linux kernel running at the time the container is started, eg.  
lxc.cgroup.cpuset.cpus

#### CAPABILITIES

The capabilities can be dropped in the container if this one is run as root.

##### lxc.cap.drop

Specify the capability to be dropped in the container. A single line defining several capabilities with a space separation is allowed. The format is the lower case of the capability definition without the "CAP\_" prefix, eg. CAP\_SYS\_MODULE should be specified as sys\_module. See capabilities(7),

##### lxc.cap.keep

Specify the capability to be kept in the container. All other capabilities will be dropped.

#### APPARMOR PROFILE

If lxc was compiled and installed with apparmor support, and the host system has apparmor enabled, then the apparmor profile under which the container should be run can be specified in the container configuration. The default is lxc-container-default.

##### lxc.aa\_profile

Specify the apparmor profile under which the container should be run. To specify that the container should be unconfined, use

```
lxc.aa_profile = unconfined
```

#### SELINUX CONTEXT

If lxc was compiled and installed with SELinux support, and the host system has SELinux enabled, then the SELinux context under which the container should be run can be specified in the container configuration. The default is unconfined\_t, which means that lxc will not attempt to change contexts.

##### lxc.se\_context

Specify the SELinux context under which the container should be run or unconfined\_t. For example

```
lxc.se_context = unconfined_u:unconfined_r:lxc_t:s0-s0:c0.c1023
```

#### SECCOMP CONFIGURATION

A container can be started with a reduced set of available system calls by loading a seccomp profile at startup. The seccomp configuration file must begin with a version number on the first line, a policy type on the second line, followed by the configuration.

Versions 1 and 2 are currently supported. In version 1, the policy is a simple whitelist. The second line therefore must read "whitelist", with the rest of the file containing one (numeric) syscall number per line. Each syscall number is whitelisted, while every unlisted number

is blacklisted for use in the container

In version 2, the policy may be blacklist or whitelist, supports per-rule and per-policy default actions, and supports per-architecture system call resolution from textual names.

An example blacklist policy, in which all system calls are allowed except for `mknod`, which will simply do nothing and return 0 (success), looks like:

2

blacklist

`mknod` `errno` 0

`lxc.seccomp`

Specify a file containing the seccomp configuration to load before the container starts.

#### UID MAPPINGS

A container can be started in a private user namespace with user and group id mappings. For instance, you can map `userid` 0 in the container to `userid` 200000 on the host. The root user in the container will be privileged in the container, but unprivileged on the host. Normally a system container will want a range of ids, so you would map, for instance, user and group ids 0 through 20,000 in the container to the ids 200,000 through 220,000.

`lxc.id_map`

Four values must be provided. First a character, either 'u', or 'g', to specify whether user or group ids are being mapped. Next is the first `userid` as seen in the user namespace of the container. Next is the `userid` as seen on the host. Finally, a range indicating the number of consecutive ids to map.

#### CONTAINER HOOKS

Container hooks are programs or scripts which can be executed at various times in a container's lifetime.

When a container hook is executed, information is passed both as command line arguments and through environment variables. The arguments are:

- Container name.
- Section (always 'lxc').
- The hook type (i.e. 'clone' or 'pre-mount').
- Additional arguments In the case of the clone hook, any extra arguments passed to `lxc-clone` will appear as further arguments to the hook.

The following environment variables are set:

- `LXC_NAME`: is the container's name.
- `LXC_ROOTFS_MOUNT`: the path to the mounted root filesystem.
- `LXC_CONFIG_FILE`: the path to the container configuration file.

- `LXC_SRC_NAME`: in the case of the clone hook, this is the original container's name.
- `LXC_ROOTFS_PATH`: this is the `lxc.rootfs` entry for the container. Note this is likely not where the mounted rootfs is to be found, use `LXC_ROOTFS_MOUNT` for that.

Standard output from the hooks is logged at debug level. Standard error is not logged, but can be captured by the hook redirecting its standard error to standard output.

#### `lxc.hook.pre-start`

A hook to be run in the host's namespace before the container ttys, consoles, or mounts are up.

#### `lxc.hook.pre-mount`

A hook to be run in the container's fs namespace but before the rootfs has been set up. This allows for manipulation of the rootfs, i.e. to mount an encrypted filesystem. Mounts done in this hook will not be reflected on the host (apart from mounts propagation), so they will be automatically cleaned up when the container shuts down.

#### `lxc.hook.mount`

A hook to be run in the container's namespace after mounting has been done, but before the `pivot_root`.

#### `lxc.hook.autodev`

A hook to be run in the container's namespace after mounting has been done and after any mount hooks have run, but before the `pivot_root`, if `lxc.autodev == 1`. The purpose of this hook is to assist in populating the `/dev` directory of the container when using the `autodev` option for `systemd` based containers. The container's `/dev` directory is relative to the `${LXC_ROOTFS_MOUNT}` environment variable available when the hook is run.

#### `lxc.hook.start`

A hook to be run in the container's namespace immediately before executing the container's `init`. This requires the program to be available in the container.

#### `lxc.hook.post-stop`

A hook to be run in the host's namespace after the container has been shut down.

#### `lxc.hook.clone`

A hook to be run when the container is cloned to a new one. See `lxc-clone(1)` for more information.

### CONTAINER HOOKS ENVIRONMENT VARIABLES

A number of environment variables are made available to the startup hooks to provide configuration information and assist in the functioning of the hooks. Not all variables are valid in all contexts. In particular, all paths are relative to the host system and, as such, not valid during the `lxc.hook.start` hook.

#### `LXC_NAME`

The LXC name of the container. Useful for logging messages in common log environments. `[-n]`

**LXC\_CONFIG\_FILE**

Host relative path to the container configuration file. This gives the container to reference the original, top level, configuration file for the container in order to locate any additional configuration information not otherwise made available. [-f]

**LXC\_CONSOLE**

The path to the console output of the container if not NULL. [-c] [lxc.console]

**LXC\_CONSOLE\_LOGPATH**

The path to the console log output of the container if not NULL. [-L]

**LXC\_ROOTFS\_MOUNT**

The mount location to which the container is initially bound. This will be the host relative path to the container rootfs for the container instance being started and is where changes should be made for that instance. [lxc.rootfs.mount]

**LXC\_ROOTFS\_PATH**

The host relative path to the container root which has been mounted to the rootfs.mount location. [lxc.rootfs]

**LOGGING**

Logging can be configured on a per-container basis. By default, depending upon how the lxc package was compiled, container startup is logged only at the ERROR level, and logged to a file named after the container (with '.log' appended) either under the container path, or under /usr/local/var/log/lxc.

Both the default log level and the log file can be specified in the container configuration file, overriding the default behavior. Note that the configuration file entries can in turn be overridden by the command line options to lxc-start.

**lxc.loglevel**

The level at which to log. The log level is an integer in the range of 0..8 inclusive, where a lower number means more verbose debugging. In particular 0 = trace, 1 = debug, 2 = info, 3 = notice, 4 = warn, 5 = error, 6 = critical, 7 = alert, and 8 = fatal. If unspecified, the level defaults to 5 (error), so that only errors and above are logged.

Note that when a script (such as either a hook script or a network interface up or down script) is called, the script's standard output is logged at level 1, debug.

**lxc.logfile**

The file to which logging info should be written.

**AUTOSTART**

The autostart options support marking which containers should be auto-started and in what order. These options may be used by LXC tools directly or by external tooling provided by the distributions.

**lxc.start.auto**

Whether the container should be auto-started. Valid values

are 0 (off) and 1 (on).

**lxc.start.delay**

How long to wait (in seconds) after the container is started before starting the next one.

**lxc.start.order**

An integer used to sort the containers when auto-starting a series of containers at once.

**lxc.group**

A multi-value key (can be used multiple times) to put the container in a container group. Those groups can then be used (amongst other things) to start a series of related containers.

**EXAMPLES**

In addition to the few examples given below, you will find some other examples of configuration file in `/usr/local/share/doc/lxc/examples`

**NETWORK**

This configuration sets up a container to use a veth pair device with one side plugged to a bridge `br0` (which has been configured before on the system by the administrator). The virtual network device visible in the container is renamed to `eth0`.

```
lxc.utsname = myhostname
lxc.network.type = veth
lxc.network.flags = up
lxc.network.link = br0
lxc.network.name = eth0
lxc.network.hwaddr = 4a:49:43:49:79:bf
lxc.network.ipv4 = 10.2.3.5/24 10.2.3.255
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3597
```

**UID/GID MAPPING**

This configuration will map both user and group ids in the range 0-9999 in the container to the ids 100000-109999 on the host.

```
lxc.id_map = u 0 100000 10000
lxc.id_map = g 0 100000 10000
```

**CONTROL GROUP**

This configuration will setup several control groups for the application, `cpuset.cpus` restricts usage of the defined cpu, `cpus.share` prioritize the control group, `devices.allow` makes usable the specified devices.

```
lxc.cgroup.cpuset.cpus = 0,1
lxc.cgroup.cpu.shares = 1234
lxc.cgroup.devices.deny = a
lxc.cgroup.devices.allow = c 1:3 rw
lxc.cgroup.devices.allow = b 8:0 rw
```

**COMPLEX CONFIGURATION**

This example show a complex configuration making a complex network stack, using the control groups, setting a new hostname, mounting some locations and a changing root file system.

```

lxc.utsname = complex
lxc.network.type = veth
lxc.network.flags = up
lxc.network.link = br0
lxc.network.hwaddr = 4a:49:43:49:79:bf
lxc.network.ipv4 = 10.2.3.5/24 10.2.3.255
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3597
lxc.network.ipv6 = 2003:db8:1:0:214:5432:feab:3588
lxc.network.type = macvlan
lxc.network.flags = up
lxc.network.link = eth0
lxc.network.hwaddr = 4a:49:43:49:79:bd
lxc.network.ipv4 = 10.2.3.4/24
lxc.network.ipv4 = 192.168.10.125/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
lxc.network.type = phys
lxc.network.flags = up
lxc.network.link = dummy0
lxc.network.hwaddr = 4a:49:43:49:79:ff
lxc.network.ipv4 = 10.2.3.6/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3297
lxc.cgroup.cpuset.cpus = 0,1
lxc.cgroup.cpu.shares = 1234
lxc.cgroup.devices.deny = a
lxc.cgroup.devices.allow = c 1:3 rw
lxc.cgroup.devices.allow = b 8:0 rw
lxc.mount = /etc/fstab.complex
lxc.mount.entry = /lib /root/myrootfs/lib none ro,bind 0 0
lxc.rootfs = /mnt/rootfs.complex
lxc.cap.drop = sys_module mknod setuid net_raw
lxc.cap.drop = mac_override

```

## SEE ALSO

chroot(1), pivot\_root(8), fstab(5), capabilities(7)

## SEE ALSO

lxc(7), lxc-create(1), lxc-destroy(1), lxc-start(1), lxc-stop(1),  
lxc-execute(1), lxc-console(1), lxc-monitor(1), lxc-wait(1),  
lxc-cgroup(1), lxc-ls(1), lxc-info(1), lxc-freeze(1),  
lxc-unfreeze(1), lxc-attach(1), lxc.conf(5)

## AUTHOR

Daniel Lezcano <daniel.lezcano@free.fr>

## COLOPHON

This page is part of the lxc (Linux containers) project. Information about the project can be found at <http://linuxcontainers.org/>. If you have a bug report for this manual page, send it to [lxc-devel@lists.linuxcontainers.org](mailto:lxc-devel@lists.linuxcontainers.org). This page was obtained from the project's upstream Git repository ([git://github.com/lxc/lxc](https://github.com/lxc/lxc)) on 2014-05-21. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is not part of the original manual page), send a mail to [man-pages@man7.org](mailto:man-pages@man7.org)

Wed May 21 10:30:15 CEST 2014 LXC.CONTAINER.CONF(5)

The text below comes from the `lxc.system.conf.5` man page:

```
LXC.SYSTEM.CONF(5) LXC.SYSTEM.CONF(5)
NAME

 lxc.system.conf - LXC system configuration file

DESCRIPTION

 The system configuration is located at /usr/local/etc/lxc/lxc.conf or
 ~/.config/lxc/lxc.conf for unprivileged containers.

 This configuration file is used to set values such as default lookup
 paths and storage backend settings for LXC.

CONFIGURATION PATHS

 lxc.lxcpath
 The location in which all containers are stored.

 lxc.default_config
 The path to the default container configuration.

CONTROL GROUPS

 lxc.cgroup.use
 Comma separated list of cgroup controllers to setup.

 lxc.cgroup.pattern
 Format string used to generate the cgroup path (e.g. lxc/%n).

LVM

 lxc.bdev.lvm.vg
 Default LVM volume group name.

 lxc.bdev.lvm.thin_pool
 Default LVM thin pool name.

ZFS

 lxc.bdev.zfs.root
 Default ZFS root name.

 top

 lxc(1), lxc.container.conf(5), lxc.system.conf(5)

SEE ALSO

 lxc(7), lxc-create(1), lxc-destroy(1), lxc-start(1), lxc-stop(1),
 lxc-execute(1), lxc-console(1), lxc-monitor(1), lxc-wait(1),
 lxc-cgroup(1), lxc-ls(1), lxc-info(1), lxc-freeze(1),
 lxc-unfreeze(1), lxc-attach(1), lxc.conf(5)

AUTHOR

 Stéphane Graber <stgraber@ubuntu.com>

COLOPHON

 This page is part of the lxc (Linux containers) project. Information
```

about the project can be found at <http://linuxcontainers.org/>. If you have a bug report for this manual page, send it to [lxc-devel@lists.linuxcontainers.org](mailto:lxc-devel@lists.linuxcontainers.org). This page was obtained from the project's upstream Git repository ([git://github.com/lxc/lxc](https://github.com/lxc/lxc)) on 2014-05-21. If you discover any rendering problems in this HTML version of the page, or you believe there is a better or more up-to-date source for the page, or you have corrections or improvements to the information in this COLOPHON (which is not part of the original manual page), send a mail to [man-pages@man7.org](mailto:man-pages@man7.org)

Wed May 21 10:30:15 CEST 2014 LXC.SYSTEM.CONF (5)

## 9.3.6.2 Documentation/cgroups/cgroups.txt

The following documentation is from the Linux kernel (Documentation/cgroups/cgroups.txt)

CGROUPS

-----

Written by Paul Menage <[menage@google.com](mailto:menage@google.com)> based on  
Documentation/cgroups/cpusets.txt

Original copyright statements from cpusets.txt:  
Portions Copyright (C) 2004 BULL SA.  
Portions Copyright (c) 2004-2006 Silicon Graphics, Inc.  
Modified by Paul Jackson <[pj@sgi.com](mailto:pj@sgi.com)>  
Modified by Christoph Lameter <[clameter@sgi.com](mailto:clameter@sgi.com)>

CONTENTS:

=====

1. Control Groups
  - 1.1 What are cgroups ?
  - 1.2 Why are cgroups needed ?
  - 1.3 How are cgroups implemented ?
  - 1.4 What does `notify_on_release` do ?
  - 1.5 What does `clone_children` do ?
  - 1.6 How do I use cgroups ?
2. Usage Examples and Syntax
  - 2.1 Basic Usage
  - 2.2 Attaching processes
  - 2.3 Mounting hierarchies by name
3. Kernel API
  - 3.1 Overview
  - 3.2 Synchronization
  - 3.3 Subsystem API
4. Extended attributes usage
5. Questions

1. Control Groups

=====

1.1 What are cgroups ?

-----

Control Groups provide a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behaviour.

Definitions:

A *\*cgroup\** associates a set of tasks with a set of parameters for one or more subsystems.

A *\*subsystem\** is a module that makes use of the task grouping facilities provided by cgroups to treat groups of tasks in particular ways. A subsystem is typically a "resource controller" that schedules a resource or applies per-cgroup limits, but it may be anything that wants to act on a group of processes, e.g. a virtualization subsystem.

A *\*hierarchy\** is a set of cgroups arranged in a tree, such that every task in the system is in exactly one of the cgroups in the hierarchy, and a set of subsystems; each subsystem has system-specific state attached to each cgroup in the hierarchy. Each hierarchy has an instance of the cgroup virtual filesystem associated with it.

At any one time there may be multiple active hierarchies of task cgroups. Each hierarchy is a partition of all tasks in the system.

User-level code may create and destroy cgroups by name in an instance of the cgroup virtual file system, specify and query to which cgroup a task is assigned, and list the task PIDs assigned to a cgroup. Those creations and assignments only affect the hierarchy associated with that instance of the cgroup file system.

On their own, the only use for cgroups is for simple job tracking. The intention is that other subsystems hook into the generic cgroup support to provide new attributes for cgroups, such as accounting/limiting the resources which processes in a cgroup can access. For example, cpusets (see Documentation/cgroups/cpusets.txt) allow you to associate a set of CPUs and a set of memory nodes with the tasks in each cgroup.

1.2 Why are cgroups needed ?

-----

There are multiple efforts to provide process aggregations in the Linux kernel, mainly for resource-tracking purposes. Such efforts include cpusets, CKRM/ResGroups, UserBeanCounters, and virtual server namespaces. These all require the basic notion of a grouping/partitioning of processes, with newly forked processes ending up in the same group (cgroup) as their parent process.

The kernel cgroup patch provides the minimum essential kernel mechanisms required to efficiently implement such groups. It has minimal impact on the system fast paths, and provides hooks for specific subsystems such as cpusets to provide additional behaviour as desired.

Multiple hierarchy support is provided to allow for situations where the division of tasks into cgroups is distinctly different for different subsystems - having parallel hierarchies allows each hierarchy to be a natural division of tasks, without having to handle complex combinations of tasks that would be present if several unrelated subsystems needed to be forced into the same tree of cgroups.

At one extreme, each resource controller or subsystem could be in a

separate hierarchy; at the other extreme, all subsystems would be attached to the same hierarchy.

As an example of a scenario (originally proposed by vatsa@in.ibm.com) that can benefit from multiple hierarchies, consider a large university server with various users - students, professors, system tasks etc. The resource planning for this server could be along the following lines:

```
CPU : "Top cpuset"
 / \
 CPUSet1 CPUSet2
 | |
 (Professors) (Students)
```

In addition (system tasks) are attached to topcpuset (so that they can run anywhere) with a limit of 20%

Memory : Professors (50%), Students (30%), system (20%)

Disk : Professors (50%), Students (30%), system (20%)

```
Network : WWW browsing (20%), Network File System (60%), others (20%)
 / \
 Professors (15%) students (5%)
```

Browsers like Firefox/Lynx go into the WWW network class, while (k)nfsd goes into the NFS network class.

At the same time Firefox/Lynx will share an appropriate CPU/Memory class depending on who launched it (prof/student).

With the ability to classify tasks differently for different resources (by putting those resource subsystems in different hierarchies), the admin can easily set up a script which receives exec notifications and depending on who is launching the browser he can

```
echo browser_pid > /sys/fs/cgroup/<restype>/<userclass>/tasks
```

With only a single hierarchy, he now would potentially have to create a separate cgroup for every browser launched and associate it with appropriate network and other resource class. This may lead to proliferation of such cgroups.

Also let's say that the administrator would like to give enhanced network access temporarily to a student's browser (since it is night and the user wants to do online gaming :) ) OR give one of the student's simulation apps enhanced CPU power.

With ability to write PIDs directly to resource classes, it's just a matter of:

```
echo pid > /sys/fs/cgroup/network/<new_class>/tasks
(after some time)
echo pid > /sys/fs/cgroup/network/<orig_class>/tasks
```

Without this ability, the administrator would have to split the cgroup into multiple separate ones and then associate the new cgroups with the new resource classes.

### 1.3 How are cgroups implemented ?

-----

Control Groups extends the kernel as follows:

- Each task in the system has a reference-counted pointer to a `css_set`.
- A `css_set` contains a set of reference-counted pointers to `cgroup_subsys_state` objects, one for each cgroup subsystem registered in the system. There is no direct link from a task to the cgroup of which it's a member in each hierarchy, but this can be determined by following pointers through the `cgroup_subsys_state` objects. This is because accessing the subsystem state is something that's expected to happen frequently and in performance-critical code, whereas operations that require a task's actual cgroup assignments (in particular, moving between cgroups) are less common. A linked list runs through the `cg_list` field of each `task_struct` using the `css_set`, anchored at `css_set->tasks`.
- A cgroup hierarchy filesystem can be mounted for browsing and manipulation from user space.
- You can list all the tasks (by PID) attached to any cgroup.

The implementation of cgroups requires a few, simple hooks into the rest of the kernel, none in performance-critical paths:

- in `init/main.c`, to initialize the root cgroups and initial `css_set` at system boot.
- in `fork` and `exit`, to attach and detach a task from its `css_set`.

In addition, a new file system of type "cgroup" may be mounted, to enable browsing and modifying the cgroups presently known to the kernel. When mounting a cgroup hierarchy, you may specify a comma-separated list of subsystems to mount as the filesystem mount options. By default, mounting the cgroup filesystem attempts to mount a hierarchy containing all registered subsystems.

If an active hierarchy with exactly the same set of subsystems already exists, it will be reused for the new mount. If no existing hierarchy matches, and any of the requested subsystems are in use in an existing hierarchy, the mount will fail with `-EBUSY`. Otherwise, a new hierarchy is activated, associated with the requested subsystems.

It's not currently possible to bind a new subsystem to an active cgroup hierarchy, or to unbind a subsystem from an active cgroup hierarchy. This may be possible in future, but is fraught with nasty error-recovery issues.

When a cgroup filesystem is unmounted, if there are any child cgroups created below the top-level cgroup, that hierarchy will remain active even though unmounted; if there are no child cgroups then the hierarchy will be deactivated.

No new system calls are added for cgroups - all support for

querying and modifying cgroups is via this cgroup file system.

Each task under /proc has an added file named 'cgroup' displaying, for each active hierarchy, the subsystem names and the cgroup name as the path relative to the root of the cgroup file system.

Each cgroup is represented by a directory in the cgroup file system containing the following files describing that cgroup:

- tasks: list of tasks (by PID) attached to that cgroup. This list is not guaranteed to be sorted. Writing a thread ID into this file moves the thread into this cgroup.
- cgroup.procs: list of thread group IDs in the cgroup. This list is not guaranteed to be sorted or free of duplicate TGIDs, and userspace should sort/uniquify the list if this property is required. Writing a thread group ID into this file moves all threads in that group into this cgroup.
- notify\_on\_release flag: run the release agent on exit?
- release\_agent: the path to use for release notifications (this file exists in the top cgroup only)

Other subsystems such as cpusets may add additional files in each cgroup dir.

New cgroups are created using the mkdir system call or shell command. The properties of a cgroup, such as its flags, are modified by writing to the appropriate file in that cgroups directory, as listed above.

The named hierarchical structure of nested cgroups allows partitioning a large system into nested, dynamically changeable, "soft-partitions".

The attachment of each task, automatically inherited at fork by any children of that task, to a cgroup allows organizing the work load on a system into related sets of tasks. A task may be re-attached to any other cgroup, if allowed by the permissions on the necessary cgroup file system directories.

When a task is moved from one cgroup to another, it gets a new css\_set pointer - if there's an already existing css\_set with the desired collection of cgroups then that group is reused, otherwise a new css\_set is allocated. The appropriate existing css\_set is located by looking into a hash table.

To allow access from a cgroup to the css\_sets (and hence tasks) that comprise it, a set of cg\_cgroup\_link objects form a lattice; each cg\_cgroup\_link is linked into a list of cg\_cgroup\_links for a single cgroup on its cgrp\_link\_list field, and a list of cg\_cgroup\_links for a single css\_set on its cg\_link\_list.

Thus the set of tasks in a cgroup can be listed by iterating over each css\_set that references the cgroup, and sub-iterating over each css\_set's task set.

The use of a Linux virtual file system (vfs) to represent the cgroup hierarchy provides for a familiar permission and name space for cgroups, with a minimum of additional kernel code.

1.4 What does notify\_on\_release do ?

-----

If the `notify_on_release` flag is enabled (1) in a cgroup, then whenever the last task in the cgroup leaves (exits or attaches to some other cgroup) and the last child cgroup of that cgroup is removed, then the kernel runs the command specified by the contents of the `"release_agent"` file in that hierarchy's root directory, supplying the pathname (relative to the mount point of the cgroup file system) of the abandoned cgroup. This enables automatic removal of abandoned cgroups. The default value of `notify_on_release` in the root cgroup at system boot is disabled (0). The default value of other cgroups at creation is the current value of their parents' `notify_on_release` settings. The default value of a cgroup hierarchy's `release_agent` path is empty.

#### 1.5 What does `clone_children` do ? -----

This flag only affects the `cpuset` controller. If the `clone_children` flag is enabled (1) in a cgroup, a new `cpuset` cgroup will copy its configuration from the parent during initialization.

#### 1.6 How do I use cgroups ? -----

To start a new job that is to be contained within a cgroup, using the `"cpuset"` cgroup subsystem, the steps are something like:

- 1) `mount -t tmpfs cgroup_root /sys/fs/cgroup`
- 2) `mkdir /sys/fs/cgroup/cpuset`
- 3) `mount -t cgroup -ocpuset cpuset /sys/fs/cgroup/cpuset`
- 4) Create the new cgroup by doing `mkdir`'s and `write`'s (or `echo`'s) in the `/sys/fs/cgroup` virtual file system.
- 5) Start a task that will be the "founding father" of the new job.
- 6) Attach that task to the new cgroup by writing its PID to the `/sys/fs/cgroup/cpuset/tasks` file for that cgroup.
- 7) `fork`, `exec` or `clone` the job tasks from this founding father task.

For example, the following sequence of commands will setup a cgroup named "Charlie", containing just CPUs 2 and 3, and Memory Node 1, and then start a subshell 'sh' in that cgroup:

```
mount -t tmpfs cgroup_root /sys/fs/cgroup
mkdir /sys/fs/cgroup/cpuset
mount -t cgroup cpuset -ocpuset /sys/fs/cgroup/cpuset
cd /sys/fs/cgroup/cpuset
mkdir Charlie
cd Charlie
/bin/echo 2-3 > cpuset.cpus
/bin/echo 1 > cpuset.mems
/bin/echo $$ > tasks
sh
The subshell 'sh' is now running in cgroup Charlie
The next line should display '/Charlie'
cat /proc/self/cgroup
```

## 2. Usage Examples and Syntax =====

### 2.1 Basic Usage -----

Creating, modifying, using cgroups can be done through the cgroup virtual filesystem.

To mount a cgroup hierarchy with all available subsystems, type:  
# mount -t cgroup xxx /sys/fs/cgroup

The "xxx" is not interpreted by the cgroup code, but will appear in /proc/mounts so may be any useful identifying string that you like.

Note: Some subsystems do not work without some user input first. For instance, if cpusets are enabled the user will have to populate the cpus and mems files for each new cgroup created before that group can be used.

As explained in section '1.2 Why are cgroups needed?' you should create different hierarchies of cgroups for each single resource or group of resources you want to control. Therefore, you should mount a tmpfs on /sys/fs/cgroup and create directories for each cgroup resource or resource group.

```
mount -t tmpfs cgroup_root /sys/fs/cgroup
mkdir /sys/fs/cgroup/rg1
```

To mount a cgroup hierarchy with just the cpuset and memory subsystems, type:  
# mount -t cgroup -o cpuset,memory hier1 /sys/fs/cgroup/rg1

While remounting cgroups is currently supported, it is not recommended to use it. Remounting allows changing bound subsystems and release\_agent. Rebinding is hardly useful as it only works when the hierarchy is empty and release\_agent itself should be replaced with conventional fsnotify. The support for remounting will be removed in the future.

To Specify a hierarchy's release\_agent:  
# mount -t cgroup -o cpuset,release\_agent="/sbin/cpuset\_release\_agent" \ xxx /sys/fs/cgroup/rg1

Note that specifying 'release\_agent' more than once will return failure.

Note that changing the set of subsystems is currently only supported when the hierarchy consists of a single (root) cgroup. Supporting the ability to arbitrarily bind/unbind subsystems from an existing cgroup hierarchy is intended to be implemented in the future.

Then under /sys/fs/cgroup/rg1 you can find a tree that corresponds to the tree of the cgroups in the system. For instance, /sys/fs/cgroup/rg1 is the cgroup that holds the whole system.

If you want to change the value of release\_agent:  
# echo "/sbin/new\_release\_agent" > /sys/fs/cgroup/rg1/release\_agent

It can also be changed via remount.

If you want to create a new cgroup under /sys/fs/cgroup/rg1:  
# cd /sys/fs/cgroup/rg1  
# mkdir my\_cgroup

Now you want to do something with this cgroup.  
# cd my\_cgroup

In this directory you can find several files:  
# ls  
cgroup.procs notify\_on\_release tasks  
(plus whatever files added by the attached subsystems)

Now attach your shell to this cgroup:  
# /bin/echo \$\$ > tasks

You can also create cgroups inside your cgroup by using mkdir in this directory.  
# mkdir my\_sub\_cs

To remove a cgroup, just use rmdir:  
# rmdir my\_sub\_cs

This will fail if the cgroup is in use (has cgroups inside, or has processes attached, or is held alive by other subsystem-specific reference).

## 2.2 Attaching processes

-----

# /bin/echo PID > tasks

Note that it is PID, not PIDs. You can only attach ONE task at a time. If you have several tasks to attach, you have to do it one after another:

```
/bin/echo PID1 > tasks
/bin/echo PID2 > tasks
...
/bin/echo PIDn > tasks
```

You can attach the current shell task by echoing 0:

```
echo 0 > tasks
```

You can use the cgroup.procs file instead of the tasks file to move all threads in a threadgroup at once. Echoing the PID of any task in a threadgroup to cgroup.procs causes all tasks in that threadgroup to be attached to the cgroup. Writing 0 to cgroup.procs moves all tasks in the writing task's threadgroup.

Note: Since every task is always a member of exactly one cgroup in each mounted hierarchy, to remove a task from its current cgroup you must move it into a new cgroup (possibly the root cgroup) by writing to the new cgroup's tasks file.

Note: Due to some restrictions enforced by some cgroup subsystems, moving a process to another cgroup can fail.

## 2.3 Mounting hierarchies by name

-----

Passing the name=<x> option when mounting a cgroups hierarchy associates the given name with the hierarchy. This can be used when mounting a pre-existing hierarchy, in order to refer to it by name rather than by its set of active subsystems. Each hierarchy is either nameless, or has a unique name.

The name should match `[\w.-]+`

When passing a `name=<x>` option for a new hierarchy, you need to specify subsystems manually; the legacy behaviour of mounting all subsystems when none are explicitly specified is not supported when you give a subsystem a name.

The name of the subsystem appears as part of the hierarchy description in `/proc/mounts` and `/proc/<pid>/cgroups`.

### 3. Kernel API

=====

#### 3.1 Overview

-----

Each kernel subsystem that wants to hook into the generic cgroup system needs to create a `cgroup_subsys` object. This contains various methods, which are callbacks from the cgroup system, along with a subsystem ID which will be assigned by the cgroup system.

Other fields in the `cgroup_subsys` object include:

- `subsys_id`: a unique array index for the subsystem, indicating which entry in `cgroup->subsys[]` this subsystem should be managing.
- `name`: should be initialized to a unique subsystem name. Should be no longer than `MAX_CGROUP_TYPE_NAMELEN`.
- `early_init`: indicate if the subsystem needs early initialization at system boot.

Each cgroup object created by the system has an array of pointers, indexed by subsystem ID; this pointer is entirely managed by the subsystem; the generic cgroup code will never touch this pointer.

#### 3.2 Synchronization

-----

There is a global mutex, `cgroup_mutex`, used by the cgroup system. This should be taken by anything that wants to modify a cgroup. It may also be taken to prevent cgroups from being modified, but more specific locks may be more appropriate in that situation.

See `kernel/cgroup.c` for more details.

Subsystems can take/release the `cgroup_mutex` via the functions `cgroup_lock()/cgroup_unlock()`.

Accessing a task's cgroup pointer may be done in the following ways:

- while holding `cgroup_mutex`
- while holding the task's `alloc_lock` (via `task_lock()`)
- inside an `rcu_read_lock()` section via `rcu_dereference()`

#### 3.3 Subsystem API

-----

Each subsystem should:

- add an entry in linux/cgroup\_subsys.h
- define a cgroup\_subsys object called <name>\_subsys

If a subsystem can be compiled as a module, it should also have in its module initcall a call to `cgroup_load_subsys()`, and in its exitcall a call to `cgroup_unload_subsys()`. It should also set `its_subsys.module = THIS_MODULE` in its `.c` file.

Each subsystem may export the following methods. The only mandatory methods are `css_alloc/free`. Any others that are null are presumed to be successful no-ops.

```
struct cgroup_subsys_state *css_alloc(struct cgroup *cgrp)
(cgroup_mutex held by caller)
```

Called to allocate a subsystem state object for a cgroup. The subsystem should allocate its subsystem state object for the passed cgroup, returning a pointer to the new object on success or a `ERR_PTR()` value. On success, the subsystem pointer should point to a structure of type `cgroup_subsys_state` (typically embedded in a larger subsystem-specific object), which will be initialized by the cgroup system. Note that this will be called at initialization to create the root subsystem state for this subsystem; this case can be identified by the passed cgroup object having a NULL parent (since it's the root of the hierarchy) and may be an appropriate place for initialization code.

```
int css_online(struct cgroup *cgrp)
(cgroup_mutex held by caller)
```

Called after `@cgrp` successfully completed all allocations and made visible to `cgroup_for_each_child/descendant_*` iterators. The subsystem may choose to fail creation by returning `-errno`. This callback can be used to implement reliable state sharing and propagation along the hierarchy. See the comment on `cgroup_for_each_descendant_pre()` for details.

```
void css_offline(struct cgroup *cgrp);
(cgroup_mutex held by caller)
```

This is the counterpart of `css_online()` and called iff `css_online()` has succeeded on `@cgrp`. This signifies the beginning of the end of `@cgrp`. `@cgrp` is being removed and the subsystem should start dropping all references it's holding on `@cgrp`. When all references are dropped, cgroup removal will proceed to the next step - `css_free()`. After this callback, `@cgrp` should be considered dead to the subsystem.

```
void css_free(struct cgroup *cgrp)
(cgroup_mutex held by caller)
```

The cgroup system is about to free `@cgrp`; the subsystem should free its subsystem state object. By the time this method is called, `@cgrp` is completely unused; `@cgrp->parent` is still valid. (Note - can also be called for a newly-created cgroup if an error occurs after this subsystem's `create()` method has been called for the new cgroup).

```
int can_attach(struct cgroup *cgrp, struct cgroup_taskset *tset)
(cgroup_mutex held by caller)
```

Called prior to moving one or more tasks into a cgroup; if the subsystem returns an error, this will abort the attach operation. @tset contains the tasks to be attached and is guaranteed to have at least one task in it.

If there are multiple tasks in the taskset, then:

- it's guaranteed that all are from the same thread group
- @tset contains all tasks from the thread group whether or not they're switching cgroups
- the first task is the leader

Each @tset entry also contains the task's old cgroup and tasks which aren't switching cgroup can be skipped easily using the cgroup\_taskset\_for\_each() iterator. Note that this isn't called on a fork. If this method returns 0 (success) then this should remain valid while the caller holds cgroup\_mutex and it is ensured that either attach() or cancel\_attach() will be called in future.

```
void cancel_attach(struct cgroup *cgrp, struct cgroup_taskset *tset)
(cgroup_mutex held by caller)
```

Called when a task attach operation has failed after can\_attach() has succeeded. A subsystem whose can\_attach() has some side-effects should provide this function, so that the subsystem can implement a rollback. If not, not necessary. This will be called only about subsystems whose can\_attach() operation have succeeded. The parameters are identical to can\_attach().

```
void attach(struct cgroup *cgrp, struct cgroup_taskset *tset)
(cgroup_mutex held by caller)
```

Called after the task has been attached to the cgroup, to allow any post-attachment activity that requires memory allocations or blocking. The parameters are identical to can\_attach().

```
void fork(struct task_struct *task)
```

Called when a task is forked into a cgroup.

```
void exit(struct task_struct *task)
```

Called during task exit.

```
void bind(struct cgroup *root)
(cgroup_mutex held by caller)
```

Called when a cgroup subsystem is rebound to a different hierarchy and root cgroup. Currently this will only involve movement between the default hierarchy (which never has sub-cgroups) and a hierarchy that is being created/destroyed (and hence has no sub-cgroups).

#### 4. Extended attribute usage

=====

cgroup filesystem supports certain types of extended attributes in its directories and files. The current supported types are:

- Trusted (XATTR\_TRUSTED)
- Security (XATTR\_SECURITY)

Both require CAP\_SYS\_ADMIN capability to set.

Like in tmpfs, the extended attributes in cgroup filesystem are stored using kernel memory and it's advised to keep the usage at minimum. This is the reason why user defined extended attributes are not supported, since any user can do it and there's no limit in the value size.

The current known users for this feature are SELinux to limit cgroup usage in containers and systemd for assorted meta data like main PID in a cgroup (systemd creates a cgroup per service).

5. Questions  
=====

Q: what's up with this '/bin/echo' ?

A: bash's builtin 'echo' command does not check calls to write() against errors. If you use it in the cgroup file system, you won't be able to tell whether a command succeeded or failed.

Q: When I attach processes, only the first of the line gets really attached !

A: We can only return one error code per call to write(). So you should also put only ONE PID.

## 9.4 Docker Containers

### 9.4.1 Introduction to Docker Containers

#### 9.4.1.1 Overview

This section is a guide and tutorial to building and using Docker Containers. Docker Containers are only available on ARM64 platforms, with the exception of LS1043 Big Endian.

Docker is a different set of userspace tools implementing Linux containers and focusing on a different set of use cases. The highlights of this open source project are ease of use, shared contributions and fast deployment. In the Docker ecosystem, containers are application environment packages, which can be easily distributed and developed collaboratively, and are guaranteed to be reproducible on any supporting platform, from the development stage to production. Currently, Docker containers are mainly targeting cloud environments.

Docker can be viewed as a set of separate components:

- **Images** - the "build" component of Docker. These are read-only copies of container root filesystems, consisting of the designed application and its userspace dependencies. For example, an image can contain an Ubuntu application, an Apache server and a user web app. This image can be used to get a webserver running.
- **Registries** - the "distribution" component of Docker. These are public or private stores where users can upload / download images. The images are versioned, and are built from layers. When sharing images, the layers are first downloaded separately, and the image is assembled at runtime. Each layer corresponds to a specific user commit. Images can also be built using buildfiles. The most representative registry example is the [Docker Hub](#). The current Docker installation does not support registry configuration.
- **Containers** - the "run" component of Docker. These are very similar to the containers provided by the LXC package. The main difference is that Docker containers use an overlay filesystem as container support. The layers are taken as is from the image and marked read-only, with a topmost read-write layer on top. This means that no container makes any persistent changes to the image by default - these need to be explicitly committed by the user when the environment is in the desired state. Docker containers are designed to work as application containers by default.

Docker uses a [client-server architecture](#). The client takes the user commands and talks to a daemon, which does the entire container management work. A Linux host running the daemon is called a Docker Host. The client and daemon can run on the same machine, or on different ones, communicating through sockets or a RESTful API.

The [Docker official page](#) advertises a set of use cases, mostly relevant in cloud environments: continuous integration, continuous delivery, devops, big data and infrastructure optimization. These can be easily adapted to embedded distributions as well. As for the containers themselves, the [Linux Containers](#) chapter use cases apply, with a focus on ease of use, fast deployment and distributed usage.

## 9.4.2 Build and Installation

### 9.4.2.1 Building with Yocto

Docker is a Linux user space package that can easily be added to a rootfs using the Yocto build system.

In the NXP SDK, Docker and all pre-requisite user space packages are included when building the "virt" image type:

```
bitbake fsl-image-virt
```

Docker can be easily added to any rootfs image by updating the `IMAGE_INSTALL_append` variable in the `conf/local.conf` file in the Yocto build environment. For example, append the following line to `local.conf`:

```
IMAGE_INSTALL_append = " docker docker-registry"
```

If you are building for ARM64 platforms, you need to perform an additional step. You need to update the rootfs target in the board image - by default it is `fsl-image-core`. If you plan to update this rootfs to, say, `fsl-image-virt`, you need to add the following line in to `build_<machine_release>/conf/local.conf`:

```
ROOTFS_IMAGE = "fsl-image-virt"
```

After enabling the required Linux options mentioned in the following chapter, generate the kernel itb:

```
bitbake fsl-image-kernelitb
```

### 9.4.2.2 Building the Linux Kernel

In order to use Docker, the Linux kernel must be configured with the [options required by LXC containers](#), and some others that are specific to the filesystem overlay and Docker's internal networking.

These options can be enabled automatically by building the Linux kernel with an additional config fragment. In order to do this, add the following line in the `conf/local.conf` file in the Yocto build environment:

```
DELTA_KERNEL_DEFCONFIG_append = " <sdk-devel>/sources/meta-freescale/recipes-kernel/linux/files/
containers.config"
```

Alternatively, you can enable the options manually. Please make sure you have enabled the [options required by LXC containers](#) first. To configure and build the Linux kernel:

```
bitbake virtual/kernel -c cleansstate
bitbake virtual/kernel -c menuconfig
bitbake virtual/kernel
```

Make sure that the following options are also enabled (besides the ones mentioned in the LXC chapter):

```
[*] Networking support --->
 Networking options --->
 [*] TCP/IP networking
 < > The IPv6 protocol
```

```

[*] Network packet filtering framework (Netfilter) --->
Core Netfilter Configuration --->
 <*> Netfilter connection tracking support
 <*> Netfilter nf_tables support
 <*> Netfilter nf_tables conntrack module
 <*> Netfilter nf_tables rbtree set module
 <*> Netfilter nf_tables masquerade support
 <*> Netfilter nf_tables nat module
 <*> Netfilter x_tables over nf_tables module
 <*> Netfilter Xtables support (required for ip_tables)
 <*> "CONNMARK" target support
 <*> "addrtype" address type match support
 <*> "connmark" connection mark match support
 <*> "conntrack" connection tracking match support
IP: Netfilter Configuration --->
 <*> IPv4 connection tracking support (required for NAT)
 <*> IPv4 nf_tables support
 <*> IP tables support (required for filtering/masq/NAT)
 <*> Packet filtering
 <*> iptables NAT support
 <*> MASQUERADE target support
 <*> Packet mangling
<*> Ethernet Bridge nf_tables support --->
<*> Ethernet Bridge tables (ebtables) support --->
 <*> ebt: nat table support
 <*> ebt: dnat target support
 <*> ebt: snat target support
File systems --->
 <*> Overlay filesystem support
Pseudo filesystems --->
 [*] Tmpfs virtual memory file system support (former shm fs)
 [*] Tmpfs POSIX Access Control Lists
 *- Tmpfs extended attributes

```

## 9.4.3 Docker How To's

### 9.4.3.1 Running a webserver container

The following article describes the necessary steps to deploy a web server service using a Docker container. This is based on downloading a prepared image from the Docker hub and using it to start a container.

1. Make sure you have sufficient space on the underlying filesystem where `/var/lib` resides. In the test setup, the root filesystem is mounted on a small flash memory, not sufficient for downloading the Docker image. Therefore, `/var/lib` is transferred to a RAM filesystem mount.

```

Docker - temporary /var/lib mount
mkdir ~/var_lib
cp -R /var/lib/* ~/var_lib
mount -t tmpfs var_lib_tmpfs /var/lib
cp -R ~/var_lib/* /var/lib/

```

2. Start the docker daemon. Make sure that the board has Internet access - this will be required to download the image from the [Docker Hub](#). The daemon will configure a Linux bridge for the containers with a private network and NAT.

```

~:# docker daemon
ERRO[0000] Failed to built-in GetDriver graph btrfs /var/lib/docker

```

## Virtualization

### Docker Containers

```
ERRO[0000] Failed to built-in GetDriver graph devicemapper /var/lib/docker
INFO[0000] API listen on /var/run/docker.sock
INFO[0000] Default bridge (docker0) is assigned with an IP address 172.17.0.0/16. Daemon option
--bip can be used to set a preferred IP address
INFO[0000] Loading containers: start.

INFO[0000] Loading containers: done.
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon commit=7e5506d-dirty
execdriver=native-0.2 graphdriver=overlay version=1.9.0
```

3. You can search the registry for available **arm64** images, or using any other keyword.

```
~# docker search arm64
NAME DESCRIPTION
STARS OFFICIAL AUTOMATED
ericvh/arm64-ubuntu Base image for arm64 (armv8 aka aarch64) U...
5
ericvh/arm64-ubuntu-dev arm64 Ubuntu install with basic developmen...
3
ericvh/arm64-ubuntu-hpc arm64 (armv8) root file system with multia...
2
markusk/arm64-crosscompile A debian image with the necessary tools in...
1 [OK]
jefby/arm64 arm64 develop
0
mickaelguene/arm64-debian arm64 debian with umeq inside. To be use w...
0
mickaelguene/arm64-debian-dev arm64 debian development image
0
inaz2/debian-arm64 Debian arm64 with QEMU user-mode emulation
0
mickaelguene/arm64-debian-jenkins-slave This is a basic container to be used as a ...
0
bobsense/redis-arm64 A Docker Image for Redis On ARM64
0
zlim/arm64-ubuntu aarch64 base (Ubuntu 15.04)
0
alisw/arm64-builder 0
varakur/arm64-coreos hand built image of arm64 CoreOS includin...
0
djtm/syncting-scratch-arm64 syncting scratch image
0
jefby/centos-arm64 centos7 for aarch64
0
qoriq/arm64-ubuntu 0
justinzh/arm64-vivid 0
kickinz1/ubuntu-arm64 0
zlim/arm64-ubuntu-dev aarch64 development
0
vducuy/ubuntu-wily-arm64 First test with ubuntu-15.10 for ARM64
0
myejeo01/arm64-ubuntu-bench Working copy of arm aarch64 for HPC.
0
```

```

bobsense/nginx-arm64 A Docker Image for Nginx on ARM64
0
arm64el/helloworld-arm64el hello world for arm64 el platform
0 [OK]
arm64el/busybox-arm64el busybox image for arm64
0 [OK]
arm64el/unshare-arm64el unshare image for arm64el platform
0 [OK]

```

4. In this example, `goriq/arm64-ubuntu` is used. It's a standard Ubuntu compiled for ARM64, with a `lighttpd` webserver installed and with a home page configured to display some information on the LS2085A boards, processes and networking in the container. First download the image.

```

~# docker pull goriq/arm64-ubuntu
Using default tag: latest
latest: Pulling from goriq/arm64-ubuntu

0441656ea204: Pull complete
79560570ed4e: Pull complete
938b8dde536d: Pull complete
5fa49f1394a1: Pull complete
a6acd691f8a3: Pull complete
276092d595eb: Pull complete
e5a142352d65: Pull complete
bd635402e399: Pull complete
6f799e234e33: Pull complete
b114273a61d4: Pull complete
8c7ce7746811: Pull complete
93e0b6a1ed99: Pull complete
4e549f05ce07: Pull complete
ae2db5c579b6: Pull complete
12ce21189ebb: Pull complete
efe3b8f0d401: Pull complete
f579adcec7f6: Pull complete
885db99879a8: Pull complete
Digest: sha256:eaef3a08336f59155e6cfb61bf55688711214561ddf00817b5c848211ac66b00
Status: Downloaded newer image for goriq/arm64-ubuntu:latest

```

You can check the image is available using `docker images`:

```

~# docker images
REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE
goriq/arm64-ubuntu latest 885db99879a8 59 minutes ago 326.4 MB

```

5. Start a container using the following command:

```

~# docker run -d -p 30081:80 --name=sandbox1 -h sandbox1 goriq/arm64-ubuntu bash -c "lighttpd -
f /etc/lighttpd/lighttpd.conf -D"
468a1e4c474414379cd0dab810a7883d8037db4795b3287526ad051741ef2525

```

- `run` - create and start the container. Optionally, download the image if not available on the host.
- `-d` - start the container as a daemon.

## Virtualization

### Docker Containers

- `-p 30081:80` - forward port 80 in the container to port 30081 on the board.
- `--name=sandbox1` - the name of the container (as visible to Docker).
- `-h sandbox1` - the hostname inside the container.
- `qorIQ/arm64-ubuntu` - the base image for the container.
- `bash -c "lighttpd -f /etc/lighttpd/lighttpd.conf -D"` - the command to execute as PID 1 in the container.

The command will return a unique SHA for the container. You can check that the webserver is up and running by accessing `http://BOARD_IP:30081/` from a browser. You can also check the container is running using docker:

```
~# docker ps -a
CONTAINER ID IMAGE COMMAND CREATED
STATUS PORTS NAMES
468a1e4c4744 qorIQ/arm64-ubuntu "bash -c 'lighttpd -f" About a minute ago
Up About a minute 0.0.0.0:30081->80/tcp sandbox1
```

#### 6. Stopping and deleting the container are easy operations:

```
~# docker stop sandbox1
sandbox1
~# docker rm sandbox1
sandbox1
```

#### 7. A similar command can be used to delete the image from the board.

```
~# docker rmi qorIQ/arm64-ubuntu
Untagged: qorIQ/arm64-ubuntu:latest
Deleted: 885db99879a87371cf08e6fff99a89af6306cb410c013a967461c387a2d0d638
Deleted: f579adcec7f6e5589e9ded8e6d08db020e1475984c6a9c3dc72c43726c51d015
Deleted: efe3b8f0d40139a4218331d12172087b8a77feba7944c07343f994746e09e71c
Deleted: 12ce21189ebb2ccfd36c12a349dfe4ff1095873cb3896a3b7bf6eb6bfc390c7
Deleted: ae2db5c579b6c5da7c83fc8c37a2247c7b268323fdf38f97da64daec77c9cc77
Deleted: 4e549f05ce071ec8067b44848c84d037fa74ddad785a8ee3fff86fcaa8151ca2
Deleted: 93e0b6a1ed99925ad1b0c1cdb94b3fd2a35b17379dc0723931fcc2c958556ebe
Deleted: 8c7ce774681151e92ee43e908e492488e5ec2928268ab852c657058f03ec6dc6
Deleted: b114273a61d4cbd55c702df480983e96e192ce9d5acbb1f1a2b1758c4bd07286
Deleted: 6f799e234e330ab6b4aac71de0dcd95733af609fb72ac4aae0f30dc783a5fa87
Deleted: bd635402e3999f95efc850a52c9039b720628043ff8e8e9569636f6fc7476d6f
Deleted: e5a142352d65e82ee9ee37855e210853d4db3ad7778ec3207dac91197e71b8e2
Deleted: 276092d595eb32656b16717a2750bcd88d6fe92cc8d00c8f52651548f76da3e6
Deleted: a6acd691f8a3a2717bdec3d1c930c3eef7df2ecd3b41bef4bf7376a74df426c5
Deleted: 5fa49f1394a1b10ab403398ad5768187ed60ea015f9f0110f6ef03bb3aa4a575
Deleted: 938b8dde536d2ddcf236efd423170a1d827690b8f19b9c0486364e08dbd8fedf
Deleted: 79560570ed4ec7d9ae02a3f1d904cbae9427bc368580fd324c2273e6bce8b274
Deleted: 0441656ea20494181a7b0b2eaf90381628ab4b5621018bb10d4c60ef03a3b412
```



**How To Reach Us****Home Page:**

[nxp.com](http://nxp.com)

**Web Support:**

[nxp.com/support](http://nxp.com/support)

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: [nxp.com/SalesTermsandConditions](http://nxp.com/SalesTermsandConditions).

Freescale, the Freescale logo, and Kinetis are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. All other product or service names are the property of their respective owners. ARM and Cortex are registered trademarks of ARM Limited.

© 2016 Freescale Semiconductor, Inc.

QORIQLS1046ABSPV04  
Rev. A  
22 Sep 2016

