NXP Semiconductors
User's Guide

Document Number: FRWY-LS1046ABSPUG
Rev. 0.1, 18 April 2019

Layerscape FRWY-LS1046A BSP User Guide

Supports: FRWY-LS1046A BSP v0.1

Contents

Chapter 1 Introduction.......... 4
1.1 Reference documentation.......... 5

Chapter 2 Release Notes.......... 6
2.1 Summary of overall features.......... 6
2.2 Component location.......... 7
2.3 Feature Support Matrix.......... 7
2.4 Known issues.......... 8

Chapter 3 FRWY-LS1046A BSP Overview.......... 9
3.1 FRWY-LS1046A BSP Quick Start.......... 9
3.1.1 Introduction.......... 9
3.1.2 Host system requirements.......... 9
3.1.3 Download and assemble FRWY-LS1046A BSP images.......... 10
3.1.4 Deploy FRWY-LS1046A BSP images on board.......... 11
3.1.4.1 FRWY-LS1046A reference information.......... 12
3.1.4.2 Option 1: Deploy FRWY-LS1046A BSP images using removable storage device.......... 27
3.1.4.3 Option 2: Deploy BSP images directly to storage device on a board.......... 29
3.2 How to build FRWY-LS1046A BSP with Flexbuild.......... 29
3.3 Secure boot.......... 35
3.3.1 Hardware Pre-Boot Loader (PBL) based platforms.......... 35
3.3.1.1 Introduction.......... 35
3.3.1.2 Secure boot process.......... 36
3.3.1.3 Pre-boot phase.......... 37
3.3.1.4 ISBC phase.......... 38
3.3.1.4.1 Flow in the ISBC code.......... 38
3.3.1.4.2 Super Root Keys (SRKs) and signing keys.......... 38
3.3.1.4.3 Key revocation.......... 39
3.3.1.4.4 Alternate image support.......... 39
3.3.1.4.5 ESBC with CSF header.......... 39
3.3.1.5 ESBC phase.......... 40
3.3.1.5.1 Boot script.......... 41
3.3.1.6 Next executable (Linux phase).......... 45
3.3.1.7 Product execution.......... 46
3.3.1.7.1 Introduction.......... 46
3.3.1.7.2 Chain of Trust with confidentiality.......... 47
3.3.1.8 Troubleshooting.......... 51
3.3.1.9 CSF Header Data Structure.......... 51
3.3.1.10 ISBC validation error codes.......... 60
3.3.1.11 ESBC Validation Error Codes.......... 64
3.3.1.12 Trust Architecture and SFP information.......... 65
3.3.2 Code Signing Tool.......... 65
3.3.2.1 Key generation.......... 66
3.3.2.1.1 gen_keys.......... 66
3.3.2.1.2 gen_otpmk_drbg.......... 68
3.3.2.1.3 gen_drv_drbg.......... 69
3.3.2.2 Header creation.......... 70
3.3.2.2.1 uni_pbi.......... 70
3.3.2.2.2 uni_sign.......... 72

3.3.2.3 Signature generation...76
3.3.2.3.1 gen_sign...78
3.3.2.3.2 sign_embed...79
3.3.3 Procedure to Run Secure Boot...80
3.3.3.1 Prepare board for secure boot...80
3.3.3.2 Running secure boot on target platforms...81
3.3.3.3 Steps to run Chain of Trust with confidentiality...82
3.4 FRWY-LS1046A BSP memory layout...83
3.5 Build tools...84

Chapter 4 Linux kernel...86
4.1 Configuring and building Linux kernel...88
4.2 Device Drivers...90
4.2.1 Enhanced Secured Digital Host Controller (eSDHC)...90
4.2.2 Dual Universal Asynchronous Receiver/Transmitter (DUART)...92
4.2.3 Quad Serial Peripheral Interface (QSPI)...97
4.2.4 Universal Serial Bus Interfaces...99
4.2.4.1 USB 3.0 Controller (DesignWare USB3)...99
4.2.5 Real Time Clock (RTC)...101
4.2.6 PCI Express Interface Controller...104
4.2.6.1 PCIe Linux Driver...104
4.2.6.2 PCIe Advanced Error Reporting User Manual...107
4.2.6.3 PCIe Remove and Rescan User Manual...109
4.2.7 CAAM Direct Memory Access (DMA)...110
4.2.8 Networking...113
4.2.8.1 DPAA1-specific Software...113
4.2.8.1.1 DPAA Software Architecture Overview...113
4.2.8.1.2 Linux Ethernet...140
4.2.8.1.3 Queue Manager (QMan) and Buffer Manager (BMan)...173
4.2.8.1.4 Configuring DPAA Frame Queues...228
4.2.8.1.5 Frame Manager...237
4.2.8.1.6 Frame Manager Configuration Tool User Guide...298
4.2.8.1.7 Security Engine (SEC)...377
4.2.8.1.8 Decompression/Compression Acceleration (DCE)...379
4.2.9 Security Engine (SEC)...381
4.2.10 Watchdog...394

Chapter 1 Introduction

About FRWY-LS1046A board support package (BSP)

The FRWY-LS1046A BSP is meant for Layerscape FRWY-LS1046A board, which is based on NXP LS1046A SoC.

It is a *hybrid form* of a Linux distribution because it combines the following major components to make a complete Linux system:

- NXP boot loader: U-Boot, based on `denx.de` plus patches
- NXP Linux kernel, based on `kernel.org` upstream plus patches
- User space components added by NXP
- Ubuntu standard user space file set (userland), including compilers and cross compiler

The use of Ubuntu userland is what makes the FRWY-LS1046A BSP a hybrid. Because NXP Arm SoCs are based on common industry standards; therefore, programs, such as bash and thousands of other programs run without being recompiled.

The benefit of using Ubuntu userland is the easy availability of thousands of standard Linux user space packages. The FRWY-LS1046A BSP is used in the same way as Ubuntu; however, the kernel, firmware, and some special NXP packages are managed separately.

Accessing FRWY-LS1046A BSP

The FRWY-LS1046A BSP is distributed via www.nxp.com.

There are two ways to use the FRWY-LS1046A BSP, as an integration and as a source of individual components.

FRWY-LS1046A BSP as an integration

Using the above link, you can access the Flexbuild component. By cloning the Flexbuild component and running a script, you can create and install FRWY-LS1046A BSP on a mass storage device as an integration. Now, you can use the BSP on a FRWY-LS1046A board. You can build NXP components from source using a script, `flex-builder`, or install the components from their binaries using another script, `flex-installer`.

FRWY-LS1046A BSP as components

The same link shows git repositories for individual components, for example the FRWY-LS1046A BSP Linux kernel. If you clone and examine this git, you will see a conventional kernel source tree. You can compile the kernel using the `make` command as you would compile a `kernel.org` kernel. However, you can also place a custom kernel fragment configuration file in the `arch/arm64/configs` directory. See Linux kernel on page 86 for more details.

Having git access to components is helpful if you assemble your own Linux distribution or wish to form a hybrid with a userland other than Ubuntu userland.

FRWY-LS1046A BSP git tags

The FRWY-LS1046A BSP git repositories use git tags to indicate component revisions that have been release-tested together. Use the `git tag` command to examine them and choose a tag to check out.

FRWY-LS1046A BSP relies on mass storage devices

Ubuntu userland is very convenient for evaluation purposes because it allows you to add standard Ubuntu components that you need, using the `apt-get install` command. It also provides native development tools. However, this richness means that the user space file is large, too large for RAM disks. Therefore, you require to install the FRWY-LS1046A BSP on a mass storage device, such as:

- Micro-SD card
- USB mass storage drive

The FRWY-LS1046A BSP provides scripts that populate a mass storage device with the needed files. These scripts can run on a Linux PC. A micro-SD card or USB flash drive is a removable storage device that can easily be moved between a Linux PC and a FRWY-LS1046A board.

1.1 Reference documentation

The table below lists and explains the additional documents and resources that you can refer to for more information on the LS1046A SoC and FRWY-LS1046A board.

Table 1. Reference documentation

Document	Description	Link / how to access
QorIQ LS1046A Reference Manual	Provides a detailed description about the QorIQ LS1046A multicore processor and its features, such as memory map, serial interfaces, power supply, chip features, and clock information	LS1046ARM.pdf
Layerscape FRWY-LS1046A Board Getting Started Guide	Describes the FRWY-LS1046A board settings and explains steps to set up and boot the board	FRWY-LS1046AGSG.pdf
Layerscape FRWY-LS1046A Board Reference Manual	Provides a detailed description of the FRWY-LS1046A board	FRWY-LS1046ARM.pdf

Chapter 2 Release Notes

2.1 Summary of overall features

Highlights

- Processor support
  - LS1046A processor
- Board support:
  - FRWY-LS1046A board
- Frequency support:
  - Core: 1600 MHz, DDR: 2100 MT/s, platform: 600 MHz (default)
- SerDes protocol support:
  - SerDes1: 0x3040
  - SerDes2: 0x0506
- **U-Boot**
  - TF-A boot
  - Boot from QSPI NOR flash, micro-SD card
  - A72 core, DDR4, clock
  - UART, GIC, I2C, OCRAM
  - PCIe-gen3 Root-Complex
  - USB mass storage, NAND flash
  - Networking support using QSGMII
  - MDIO PHY support
- **Linux: 4.14.83**
  - A72 core, DDR4
  - SMP-boot
  - Clock, UART
  - GIC, I2C, OCRAM
  - USB mass storage, micro-SD, NAND Flash
  - QSPI access to NOR flash
  - PCIe-gen3 Root-Complex
  - Networking interfaces: QSGMII
  - SEC, QDMA, eDMA
  - MDIO PHY support
- **Userspace components**
  - Flexbuild and toolchain

- **Flexbuild**
  - — Ubuntu userland 18.04
  - — gcc: Ubuntu/Linaro 7.3.0-16ubuntu3~18.04, glibc-2.27, binutils-2.30-0, gdb-8.1

2.2 Component location

The table below lists the locations of FRWY-LS1046A BSP release components.

Table 2. Component location

Component	CAF location	CAF branch	CAF tag
linux	https://source.codeaurora.org/external/qoriq/qoriq-components/linux/	linux-4.14-ls1046afrwy-early-access	ls1046afrwy-early-access-bsp0.1
u-boot	https://source.codeaurora.org/external/qoriq/qoriq-components/u-boot/	ls1046afrwy-early-access	ls1046afrwy-early-access-bsp0.1
rcw	https://source.codeaurora.org/external/qoriq/qoriq-components/rcw/	ls1046afrwy-early-access	ls1046afrwy-early-access-bsp0.1
atf	https://source.codeaurora.org/external/qoriq/qoriq-components/atf	ls1046afrwy-early-access	ls1046afrwy-early-access-bsp0.1
cst	https://source.codeaurora.org/external/qoriq/qoriq-components/cst	integration	LSDK-18.12
eth-config	https://source.codeaurora.org/external/qoriq/qoriq-components/eth-config	integration	LSDK-19.03
dpdk	https://source.codeaurora.org/external/qoriq/qoriq-components/dpdk	ls1046afrwy-early-access	ls1046afrwy-early-access-bsp0.1

2.3 Feature Support Matrix

The following tables show the features that are supported in this release.

Feature	Description

Table continues on the next page...

Table continued from the previous page...

QSPI	• Read/write/erase • Read in AHB mode • Write in IP mode • Erase size: 128 KB (U-Boot) • Supported flash device: MT25QU512ABB8ESF-0SIT
UART	• UART1 and UART2 verified • Default frequency: 115.2 Kbit/s
DDR	• Fixed settings • Default frequency: 2100 MT/s
QSGMII	• Four 1G Ethernet ports
NAND flash	• Read/write/erase/file system support
Micro-SD	• SD high speed (50 MHz) read/write/erase/file system support

2.4 Known issues

The table below lists the known issues with the release. These are issues having no resolution currently. Workaround suggestions are provided wherever possible.

Table 3. Known issues

ID	Description
QLINUX-XXXXX	

Chapter 3
FRWY-LS1046A BSP Overview

This section provides FRWY-LS1046A board-specific information on switch settings, U-Boot environment variable settings, and supported binaries. It also provides details of memory layout.

For more information on the FRWY-LS1046A board, see *Layerscape FRWY-LS1046A Board Getting Started Guide* and *Layerscape FRWY-LS1046A Board Reference Manual*.

3.1 FRWY-LS1046A BSP Quick Start

3.1.1 Introduction

The following sections describe the procedure to download and assemble FRWY-LS1046A BSP images and then to deploy these images on the FRWY-LS1046A board. For more information on the different components of the board and on how to configure and boot the board, see *Layerscape FRWY-LS1046A Board Getting Started Guide*.

3.1.2 Host system requirements

- Ubuntu 18.04 should be installed on the host machine
- If this requirement is not fulfilled, see “Emulate Ubuntu 18.04 environment using docker container” topic below
- For root users, there is no limitation for the build. For non-root users, obtain sudo permission by running the command `sudoedit /etc/sudoers` and adding a line `<user-account-name> ALL=(ALL:ALL) NOPASSWD: ALL in /etc/sudoers`.
- To build the target Ubuntu userland for arm64/armhf arch, the user's network environment must have access to the remote Ubuntu official server

Emulate Ubuntu 18.04 environment using docker container (optional)

If a Linux distribution other than Ubuntu 18.04 is installed on the host machine, perform the following steps to create an Ubuntu 18.04 docker container to emulate the environment:

1. Install docker on the host machine. See https://docs.docker.com/engine/installation/ for information on how to install docker on the host machine.
2. To build the Ubuntu userland, the user must have *sudo* permission for Docker commands or the user must be added to a group called “docker” as specified below:

   a. Change current group to “docker”:

   ```
   $ sudo newgrp - docker
   ```

   NOTE

   User can run the command `cut -d: -f1 /etc/group | sort` to check if "docker" group is included in the list of all the available groups.

   b. Add your account to “docker” group:

   ```
   $ sudo usermod -aG docker <accountname>
   $ sudo gpasswd -a <accountname> docker
   ```

c. Restart service:

```
$ sudo service docker restart
```

3. Log out from current terminal session, and then log in again to ensure user can run `docker ps -a`.

   The user's network environment must have access to the remote Ubuntu official server.

4. Run `flex-builder` command for docker as given in the next section.

3.1.3 Download and assemble FRWY-LS1046A BSP images

In this BSP Quick Start Guide, the build framework Flexbuild is used. Flexbuild contains three scripts: `flex-builder`, `flex-installer`, and `flex-mkdistrorfs`. In this section ("Download and assemble FRWY-LS1046A BSP images"), `flex-builder` is used to generate a composite firmware image, boot partition, and Ubuntu userland.

In the next section, "Deploy FRWY-LS1046A BSP images on board", the script `flex-installer` is used to deploy complete BSP image that includes composite firmware image, boot partation, and Ubuntu userland to a storage device (micro-SD).

The composite firmware image contains RCW, U-boot, FMan, and other components. Boot partition contains a Linux kernel image and device tree blob. For a complete list of the boot partition components and the firmware components, see FRWY-LS1046A BSP memory layout on page 83.

See Build tools on page 84 for more information on Flexbuild.

To download and assemble FRWY-LS1046A BSP images, perform the steps below:

1. Download the Flexbuild source tarball and set up Flexbuild environment:

   a. Go to www.nxp.com and click the "Download" button to download Flexbuild source tarball, named `flexbuild_<version>.tgz`. It is required to log in and sign a license agreement before downloading the tarball.

   b. Set up Flexbuild environment:

```
$ tar xvzf flexbuild_<version>.tgz
$ cd flexbuild
$ source setup.env
```

   c. Run the following two commands in addition to the above three commands only when building BSP in Docker container:

```
$ flex-builder docker (this command emulates Ubuntu 18.04 environment on the host machine.)
$ source setup.env (again, set up the Flexbuild environment after entering Docker
container.)
```

2. Download prebuilt images for boot partition and NXP-specific components tarball:

   a. Download composite firmware image:

```
wget https://www.nxp.com/lgfiles/sdk/ls1046afrwy_bsp_01/
firmware_ls1046afrwy_uboot_sdboot.img
```

   b. Download prebuilt app components (for example, fmc, dpdk, openssl):

```
wget https://www.nxp.com/lgfiles/sdk/ls1046afrwy_bsp_01/app_components_LS_arm64.tgz
```

   c. Download prebuilt images for boot partition and Arm modules:

```
wget https://www.nxp.com/lgfiles/sdk/ls1046afrwy_bsp_01/bootpartition_LS_arm64_lts_4.14.tgz
wget https://www.nxp.com/lgfiles/sdk/ls1046afrwy_bsp_01/lib_modules_LS_arm64_4.14.83.tgz
```

3. Generate FRWY-LS1046A BSP Ubuntu userland, untar the prebuilt components tarball, and merge them into target userland.

   a. Generate Ubuntu arm64 userland:

   ```
   $ flex-builder -i mkrfs -a arm64
   ```

   b. Extract app components:

   ```
   $ tar xvzf app_components_LS_arm64.tgz -C build/apps
   ```

   c. Extract kernel modules (for example, cryptodev):

   ```
   $ sudo tar xvzf  lib_modules_LS_arm64_<kernel_version>.tgz -C build/rfs/
   rootfs_ubuntu_bionic_LS_arm64/lib/modules
   ```

   d. Merge all components packages and kernel modules into target userland and compress ubuntu arm64 rootfs as .tgz tarball:

   ```
   $ flex-builder -i merge-component -a arm64
   $ flex-builder -i compressrfs -a arm64
   ```

   e. Exit, if using docker:

   ```
   $ exit (optional, this command exits from docker when building in docker)
   ```

NOTE

If the Linux host machine is in a subnet that needs HTTP proxy to access external Internet, then set environment variables, http_proxy and https_proxy as follows:

```
Add the following proxy settings in ~/.bashrc
# No authentication:
export http_proxy=http://<domain>:<port>
export https_proxy=http://<domain>:<port>

# With authentication:
export http_proxy=http://<account>:<password>@<domain>:<port>
export https_proxy=http://<account>:<password>@<domain>:<port>

# Set no_proxy variable to bypass proxy for some local servers:
export no_proxy="localhost,<local-server1>,<local-server2>"
```

NOTE

Ubuntu userland is the only default file system that is system tested in the formal FRWY-LS1046A BSP release. Although other userland images (such as Debian, CentOS, Buildroot-based tiny distro, and so on) can be generated using the common FRWY-LS1046A BSP boot partition (containing the Linux kernel, DTBs, distro boot scripts, and so on) by Flexbuild, that exercise is left to the user, and is not supported by NXP.

3.1.4 Deploy FRWY-LS1046A BSP images on board

This section assumes that the FRWY-LS1046A BSP images have been assembled according to the “Download and assemble FRWY-LS1046A BSP images” section. You can choose one of the following two options to deploy FRWY-LS1046A BSP images on the FRWY-LS1046A board depending on whether the host (Linux host machine) and/or target (FRWY-LS1046A board) is present locally or is accessed remotely:

- Option 1 (when host and target are present locally): In this option, you can deploy the BSP images on the board using a removable storage device, such as a micro-SD card. To deploy images, first connect the micro-SD card to Linux host machine and deploy images on it. Then, remove the card from Linux machine and connect it to FRWY-LS1046A board.
- Option 2 (when host and/or target is accessed remotely): In this option, you need to deploy the BSP images directly to the storage device attached/plugged-in to the board. For this option, a network connection to the board is required.

The deployment covers how to program FRWY-LS1046A BSP composite firmware for "QSPI NOR flash boot" and "micro-SD boot". It also covers how to deploy boot partition images and Ubuntu userland on different storage media (micro-SD/USB).

NOTE

Micro-SD/USB storage capacity must be at least 8 GB.

Table 4. FRWY-LS1046A BSP storage location

Boot source	Composite firmware	Kernel image and DTB	Ubuntu rootfs
QSPI NOR flash	QSPI NOR flash	Micro-SD/USB Partition 2 "Boot Partition"	Micro-SD/USB Partition 3 "Rootfs Partition"
Micro-SD	Micro-SD "raw partition"		

3.1.4.1 FRWY-LS1046A reference information

This section provides general information about FRWY-LS1046A board. The information may come in handy as a reference while performing steps for deploying BSP images that are mentioned in sections that follow.

System memory map

Table 5. System memory map

Start address (Hex)	Module name	Size	Accessible with x-bit addressing		
			32	36	40
00_0000_0000	Secure Boot ROM	1 MB	Y	Y	Y
00_0010_0000	Extended Boot ROM	15 MB	Y	Y	Y
00_0100_0000	CCSR Register Space	240 MB	Y	Y	Y
00_1000_0000	OCRAM1	64 KB	Y	Y	Y
00_1001_0000	OCRAM2	64 KB	Y	Y	Y
00_1004_0000	Reserved	65408 KB	Y	Y	Y
00_1100_0000	Reserved	16 MB	Y	Y	Y
00_1200_0000	STM	16 MB	Y	Y	Y
00_1300_0000	Reserved	208 MB	Y	Y	Y
00_2000_0000	DCSR	64 MB	Y	Y	Y
00_2400_0000	Reserved	448 MB	Y	Y	Y
00_4000_0000	QuadSPI	512 MB	Y	Y	Y
00_6000_0000	IFC region 1(0 - 512 MB)	512 MB	Y	Y	Y

Table continues on the next page...

Table 5. System memory map (continued)

Start address (Hex)	Module name	Size	Accessible with x-bit addressing		
			32	36	40
00_8000_0000	DRAM1 (0 - 2 GB)	2 GB	Y	Y	Y
01_0000_0000	Reserved	0.0625 GB	N	Y	Y
01_0400_0000	Reserved	3.9375 GB	N	Y	Y
02_0000_0000	Reserved	1 GB	N	Y	Y
02_4000_0000	Reserved	7 GB	N	Y	Y
04_0000_0000	Reserved	0.25 GB	N	Y	Y
04_1000_0000	Reserved	0.25 GB	N	Y	Y
04_2000_0000	Reserved	0.25 GB	N	Y	Y
04_3000_0000	Reserved	1.25 GB	N	Y	Y
04_8000_0000	Reserved	2 GB	N	Y	Y
05_0000_0000	QMAN S/W Portal	128 MB	N	Y	Y
05_0800_0000	BMAN S/W Portal	128 MB	N	Y	Y
05_1000_0000	Reserved	4 GB - 256 MB	N	Y	Y
06_0000_0000	Reserved	0.5 GB	N	Y	Y
06_2000_0000	IFC region 2 (512 MB - 4 GB)	3.5 GB	N	Y	Y
07_0000_0000	Reserved	4 GB	N	Y	Y
08_0000_0000	Reserved	2 GB	N	Y	Y
08_8000_0000	DRAM2	30 GB	N	Y	Y
10_0000_0000	Reserved	64 GB	N	Y	Y
20_0000_0000	Reserved	128 GB	N	N	Y
40_0000_0000	PCI Express 1	32 GB	N	N	Y
48_0000_0000	PCI Express 2	32 GB	N	N	Y
50_0000_0000	PCI Express 3	32 GB	N	N	Y
58_0000_0000	Reserved	160 GB	N	N	Y
80_0000_0000	Reserved	32 GB	N	N	Y
88_0000_0000	DRAM3 (32 - 512 GB)	480 GB	N	N	Y

Supported boot options

The FRWY-LS1046A board supports the following boot options:

- QSPI NOR flash (referred to as "QSPI" or "QSPI flash" in the following sections). CS refers to chip select.
- Micro-SD card (SDHC1)

Onboard switch options

The FRWY-LS1046A board has user selectable switches for evaluating different boot options for the LS1046A device as given in the table below ('0' is OFF, '1' is ON).

Boot source	SW1[1:10]
QSPI NOR (default)	0_0100_0100_0
Micro-SD card (SDHC1)	0_0100_0000_0

In addition to the above switch settings, ensure that the following jumper settings are correct.

Table 6. FRWY-LS1046A jumper settings

Part identifier	Jumper type	Description	Jumper settings
J72	1x2 connector	UART selection header	• Open: UART1 port is accessed remotely through a 1x4 header (J73) • Shorted: A USB 2.0 micro AB connector (J58) is connected to UART1 port through a USB-to-UART bridge (default setting)
J8	1x2 connector	VDD voltage selection header	• Open: VDD = 0.9 V • Shorted: VDD = 1 V (default setting)
J14	1x2 connector	Reset mode selection header	• Open: RESET_REQ_B pin of the processor is disconnected • Shorted: RESET_REQ_B pin triggers system reset when asserted (default setting)
J11	1x2 connector	PROG_MTR voltage control header (NXP use only)	• Open: PROG_MTR pin of the processor is powered off (default setting) • Shorted: PROG_MTR pin is powered by OVDD (1.8 V)
J9	1x2 connector	TA_BB_VDD voltage control header	• Open: TA_BB_VDD pin of the processor is powered off • Shorted: TA_BB_VDD pin is powered by VDD (1/0.9 V) (default setting)

QSPI NOR flash

QSPI NOR flash is a simple and convenient destination for deploying images; therefore, it is most common medium for deploying images. When the board boots from QSPI, the U-Boot log looks as follows:

```
NOTICE:  Fixed DDR on board

NOTICE:  4 GB DDR4, 64-bit, CL=15, ECC on
NOTICE:  BL2: v1.5(release):bsp0.1_pre-4-g6a5cfdd3
NOTICE:  BL2: Built : 10:20:10, Mar 27 2019
NOTICE:  BL31: v1.5(release):bsp0.1_pre-4-g6a5cfdd3
NOTICE:  BL31: Built : 07:54:36, Apr  9 2019
```

```
NOTICE:  Welcome to LS1046 BL31 Phase


U-Boot 2018.09-00007-gf851eafae4-dirty (Apr 09 2019 - 08:32:35 +0530)

SoC:  LS1046AE Rev1.0 (0x87070010)
Clock Configuration:
       CPU0(A72):1600 MHz  CPU1(A72):1600 MHz  CPU2(A72):1600 MHz
       CPU3(A72):1600 MHz
       Bus:      700  MHz  DDR:      2100 MT/s  FMAN:     800  MHz
Reset Configuration Word (RCW):
       00000000: 0e150012 10000000 00000000 00000000
       00000010: 30400506 00805012 40025000 c1000000
       00000020: 00000000 00000000 00000000 00038800
       00000030: 20044100 24003101 00000096 00000001
Model: LS1046A FRWY Board
Board: LS1046AFRWY, Rev: A, boot from QSPI
SD1_CLK1 = 100.00MHZ, SD1_CLK2 = 100.00MHZ
I2C:   ready
DRAM:  3.9 GiB (DDR4, 64-bit, CL=15, ECC on)
SEC0: RNG instantiated
Using SERDES1 Protocol: 12352 (0x3040)
Using SERDES2 Protocol: 1286 (0x506)
NAND:  512 MiB
MMC:   FSL_SDHC: 0
Loading Environment from SPI Flash... SF: Detected mt25qu512a with page size 256 Bytes, erase size 64
KiB, total 64 MiB
OK
EEPROM: NXID v1
In:    serial
Out:   serial
Err:   serial
Net:   SF: Detected mt25qu512a with page size 256 Bytes, erase size 64 KiB, total 64 MiB
Fman1: Uploading microcode version 106.4.18
PCIe0: pcie@3400000 disabled
PCIe1: pcie@3500000 Root Complex: x1 gen1
PCIe2: pcie@3600000 Root Complex: x1 gen1
e1000: 68:05:ca:1c:02:c4
       FM1@DTSEC1, FM1@DTSEC5, FM1@DTSEC6, FM1@DTSEC10, e1000#0
Hit any key to stop autoboot:  0
=> print botargs
## Error: "botargs" not defined
=> print bootargs
bootargs=console=ttyS0,115200 root=/dev/sdb3 rw earlycon=uart8250,mmio,0x21c0500 ramdisk_size=20000000
rootdelay=3
=> edit bootargs
edit: console=ttyS0,115200 root=/dev/ram0 rw earlycon=uart8250,mmio,0x21c0500 ramdisk_size=20000000
rootdelay=3
=> ping 192.168.3.1
FM1@DTSEC1 Waiting for PHY auto negotiation to complete...user interrupt!
FM1@DTSEC1: Could not initialize
FM1@DTSEC5 Waiting for PHY auto negotiation to complete.user interrupt!
FM1@DTSEC5: Could not initialize
FM1@DTSEC6 Waiting for PHY auto negotiation to complete..user interrupt!
FM1@DTSEC6: Could not initialize
FM1@DTSEC10 Waiting for PHY auto negotiation to complete..user interrupt!
FM1@DTSEC10: Could not initialize
Using e1000#0 device
host 192.168.3.1 is alive
=> tftp 0xa0000000 nxf28358/bin/frwy/kernel.itb
```

```
Using e1000#0 device
TFTP from server 192.168.3.1; our IP address is 192.168.3.193
Filename 'nxf28358/bin/frwy/kernel.itb'.
Load address: 0xa0000000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
```

```
         ####################################
         1.6 MiB/s
done
Bytes transferred = 18818835 (11f2713 hex)
=> bootm 0xa0000000
## Loading kernel from FIT Image at a0000000 ...
   Using 'config@1' configuration
   Trying 'kernel@1' kernel subimage
     Description:  ARM64 Linux kernel
     Created:      2019-04-04  10:25:55 UTC
     Type:         Kernel Image
     Compression:  gzip compressed
     Data Start:   0xa00000dc
     Data Size:    9212643 Bytes = 8.8 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: 0x80080000
     Entry Point:  0x80080000
   Verifying Hash Integrity ... OK
## Loading ramdisk from FIT Image at a0000000 ...
   Using 'config@1' configuration
   Trying 'ramdisk@1' ramdisk subimage
     Description:  LS2 Ramdisk
     Created:      2019-04-04  10:25:55 UTC
     Type:         RAMDisk Image
     Compression:  gzip compressed
     Data Start:   0xa08d0ff8
     Data Size:    9572804 Bytes = 9.1 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
     Created:      2019-04-04  10:25:55 UTC
     Type:         Flat Device Tree
     Compression:  uncompressed
     Data Start:   0xa08c9474
     Data Size:    31485 Bytes = 30.7 KiB
     Architecture: AArch64
     Load Address: 0x90000000
   Verifying Hash Integrity ... OK
   Loading fdt from 0xa08c9474 to 0x90000000
   Booting using the fdt blob at 0x90000000
   Uncompressing Kernel Image ... OK
   Using Device Tree in place at 0000000090000000, end 000000009001aafc
WARNING failed to get smmu node: FDT_ERR_NOTFOUND
WARNING failed to get smmu node: FDT_ERR_NOTFOUND

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.14.83-00006-g49bfece-dirty (vsharma@lsv03032.swis.in-blr01.nxp.com) (gcc 
version 7.3.0 (GCC)) #12 SMP PREEMPT Thu Apr 4 15:53:32 IST 2019
[    0.000000] Boot CPU: AArch64 Processor [410fd082]
[    0.000000] Machine model: LS1046A FRWY Board
[    0.000000] earlycon: uart8250 at MMIO 0x00000000021c0500 (options '')
```

```
[    0.000000] bootconsole [uart8250] enabled
[    0.000000] efi: Getting EFI parameters from FDT:
[    0.000000] efi: UEFI not found.
[    0.000000] OF: reserved mem: initialized node qman-fqd, compatible id fsl,qman-fqd
[    0.000000] OF: reserved mem: initialized node qman-pfdr, compatible id fsl,qman-pfdr
[    0.000000] OF: reserved mem: initialized node bman-fbpr, compatible id fsl,bman-fbpr
[    0.000000] cma: Reserved 16 MiB at 0x00000000fac00000
[    0.000000] NUMA: No NUMA configuration found
[    0.000000] NUMA: Faking a node at [mem 0x0000000000000000-0x00000008ff7fffff]
[    0.000000] NUMA: NODE_DATA [mem 0x8ff7c2280-0x8ff7c3a3f]
[    0.000000] Zone ranges:
[    0.000000]   DMA      [mem 0x0000000080000000-0x00000000ffffffff]
[    0.000000]   Normal   [mem 0x0000000100000000-0x00000008ff7fffff]
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x0000000080000000-0x00000000fbdfffff]
[    0.000000]   node   0: [mem 0x0000000880000000-0x00000008fbffffff]
[    0.000000]   node   0: [mem 0x00000008ff000000-0x00000008ff7fffff]
[    0.000000] Initmem setup node 0 [mem 0x0000000080000000-0x00000008ff7fffff]
[    0.000000] psci: probing for conduit method from DT.
[    0.000000] psci: PSCIv1.1 detected in firmware.
[    0.000000] psci: Using standard PSCI v0.2 function IDs
[    0.000000] psci: MIGRATE_INFO_TYPE not supported.
[    0.000000] psci: SMC Calling Convention v1.1
[    0.000000] percpu: Embedded 24 pages/cpu @ffff80087f75d000 s61272 r8192 d28840 u98304
[    0.000000] Detected PIPT I-cache on CPU0
[    0.000000] Built 1 zonelists, mobility grouping on.  Total pages: 1001184
[    0.000000] Policy zone: Normal
[    0.000000] Kernel command line: console=ttyS0,115200 root=/dev/ram0 rw earlycon=uart8250,mmio,
0x21c0500 ramdisk_size=20000000 rootdelay=3
[    0.000000] PID hash table entries: 4096 (order: 3, 32768 bytes)
[    0.000000] software IO TLB [mem 0xf6c00000-0xfac00000] (64MB) mapped at [ffff800076c00000-
ffff80007abfffff]
[    0.000000] Memory: 3889844K/4069376K available (12860K kernel code, 1404K rwdata, 4956K rodata,
1344K init, 919K bss, 163148K reserved, 16384K cma-reserved)
[    0.000000] Virtual kernel memory layout:
[    0.000000]     modules : 0xffff000000000000 - 0xffff000008000000   (   128 MB)
[    0.000000]     vmalloc : 0xffff000008000000 - 0xffff7dffbfff0000   (129022 GB)
[    0.000000]       .text : 0xffff000008080000 - 0xffff000008d10000   ( 12864 KB)
[    0.000000]     .rodata : 0xffff000008d10000 - 0xffff0000091f0000   (  4992 KB)
[    0.000000]       .init : 0xffff0000091f0000 - 0xffff000009340000   (  1344 KB)
[    0.000000]       .data : 0xffff000009340000 - 0xffff00000949f200   (  1405 KB)
[    0.000000]        .bss : 0xffff00000949f200 - 0xffff000009584eb8   (   920 KB)
[    0.000000]     fixed   : 0xffff7dfffe7f9000 - 0xffff7dfffec00000   (  4124 KB)
[    0.000000]     PCI I/O : 0xffff7dfffee00000 - 0xffff7dffffe00000   (    16 MB)
[    0.000000]     vmemmap : 0xffff7e0000000000 - 0xffff800000000000   (  2048 GB maximum)
[    0.000000]               0xffff7e0000000000 - 0xffff7e0021fe0000   (   543 MB actual)
[    0.000000]     memory  : 0xffff800000000000 - 0xffff80087f800000   ( 34808 MB)
[    0.000000] Preemptible hierarchical RCU implementation.
[    0.000000]  RCU restricting CPUs from NR_CPUS=64 to nr_cpu_ids=4.
[    0.000000]  Tasks RCU enabled.
[    0.000000] RCU: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=4
[    0.000000] NR_IRQS: 64, nr_irqs: 64, preallocated irqs: 0
[    0.000000] GIC: Adjusting CPU interface base to 0x000000000142f000
[    0.000000] GIC: Using split EOI/Deactivate mode
[    0.000000] arch_timer: cp15 timer(s) running at 25.00MHz (phys).
[    0.000000] clocksource: arch_sys_counter: mask: 0xffffffffffffff max_cycles: 0x5c40939b5,
max_idle_ns: 440795202646 ns
[    0.000001] sched_clock: 56 bits at 25MHz, resolution 40ns, wraps every 4398046511100ns
[    0.008352] Console: colour dummy device 80x25
```

```
[    0.012839] Calibrating delay loop (skipped), value calculated using timer frequency.. 50.00 BogoMIPS
(lpj=100000)
[    0.023248] pid_max: default: 32768 minimum: 301
[    0.027933] Security Framework initialized
[    0.033066] Dentry cache hash table entries: 524288 (order: 10, 4194304 bytes)
[    0.040859] Inode-cache hash table entries: 262144 (order: 9, 2097152 bytes)
[    0.047973] Mount-cache hash table entries: 8192 (order: 4, 65536 bytes)
[    0.054726] Mountpoint-cache hash table entries: 8192 (order: 4, 65536 bytes)
[    0.077927] ASID allocator initialised with 32768 entries
[    0.091361] Hierarchical SRCU implementation.
[    0.104178] EFI services will not be available.
[    0.116737] smp: Bringing up secondary CPUs ...
[    0.149404] Detected PIPT I-cache on CPU1
[    0.149426] CPU1: Booted secondary processor [410fd082]
[    0.177412] Detected PIPT I-cache on CPU2
[    0.177425] CPU2: Booted secondary processor [410fd082]
[    0.205435] Detected PIPT I-cache on CPU3
[    0.205447] CPU3: Booted secondary processor [410fd082]
[    0.205475] smp: Brought up 1 node, 4 CPUs
[    0.237418] SMP: Total of 4 processors activated.
[    0.242149] CPU features: detected feature: 32-bit EL0 Support
[    0.248012] CPU features: detected feature: Kernel page table isolation (KPTI)
[    0.259367] CPU: All CPU(s) started at EL2
[    0.263487] alternatives: patching kernel code
[    0.268443] devtmpfs: initialized
[    0.273867] random: get_random_u32 called from bucket_table_alloc+0x108/0x260 with crng_init=0
[    0.282653] clocksource: jiffies: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns:
7645041785100000 ns
[    0.292459] futex hash table entries: 1024 (order: 4, 65536 bytes)
[    0.298878] xor: measuring software checksum speed
[    0.343711]    8regs     :  7673.000 MB/sec
[    0.387930]    8regs_prefetch:  6686.000 MB/sec
[    0.432498]    32regs    :  8644.000 MB/sec
[    0.476717]    32regs_prefetch:  7072.000 MB/sec
[    0.481354] xor: using function: 32regs (8644.000 MB/sec)
[    0.486781] pinctrl core: initialized pinctrl subsystem
[    0.492429] DMI not present or invalid.
[    0.496418] NET: Registered protocol family 16
[    0.504899] cpuidle: using governor menu
[    0.509034] Bman ver:0a02,02,01
[    0.514051] qman-fqd addr 0x00000008ff800000 size 0x800000
[    0.519597] qman-pfdr addr 0x00000008fc000000 size 0x2000000
[    0.525292] Qman ver:0a01,03,02,01
[    0.528780] vdso: 2 pages (1 code @ ffff000008d16000, 1 data @ ffff000009345000)
[    0.536220] hw-breakpoint: found 6 breakpoint and 4 watchpoint registers.
[    0.543406] DMA: preallocated 256 KiB pool for atomic allocations
[    0.549711] Serial: AMBA PL011 UART driver
[    0.556971] Machine: LS1046A FRWY Board
[    0.560827] SoC family: QorIQ LS1046A
[    0.564501] SoC ID: svr:0x87070010, Revision: 1.0
[    0.580361] HugeTLB registered 2.00 MiB page size, pre-allocated 0 pages
[    0.655202] raid6: int64x1  gen()  1112 MB/s
[    0.727199] raid6: int64x1  xor()  1039 MB/s
[    0.799236] raid6: int64x2  gen()  1534 MB/s
[    0.871236] raid6: int64x2  xor()  1471 MB/s
[    0.943299] raid6: int64x4  gen()  2366 MB/s
[    1.015298] raid6: int64x4  xor()  1574 MB/s
[    1.087354] raid6: int64x8  gen()  2318 MB/s
[    1.159372] raid6: int64x8  xor()  1576 MB/s
[    1.231398] raid6: neonx1   gen()  2284 MB/s
```

```
[    1.303421] raid6: neonx1   xor()  2462 MB/s
[    1.375458] raid6: neonx2   gen()  3025 MB/s
[    1.447485] raid6: neonx2   xor()  3142 MB/s
[    1.519514] raid6: neonx4   gen()  4152 MB/s
[    1.591546] raid6: neonx4   xor()  3522 MB/s
[    1.663583] raid6: neonx8   gen()  4404 MB/s
[    1.735603] raid6: neonx8   xor()  3692 MB/s
[    1.739891] raid6: using algorithm neonx8 gen() 4404 MB/s
[    1.745314] raid6: .... xor() 3692 MB/s, rmw enabled
[    1.750301] raid6: using neon recovery algorithm
[    1.755208] ACPI: Interpreter disabled.
[    1.760238] vgaarb: loaded
[    1.763074] SCSI subsystem initialized
[    1.767026] usbcore: registered new interface driver usbfs
[    1.772557] usbcore: registered new interface driver hub
[    1.777922] usbcore: registered new device driver usb
[    1.783275] imx-i2c 2180000.i2c: scl-gpios not found
[    1.788348] i2c i2c-0: IMX I2C adapter registered
[    1.793095] i2c i2c-0: using dma0chan16 (tx) and dma0chan17 (rx) for DMA transfers
[    1.801181] pps_core: LinuxPPS API ver. 1 registered
[    1.806170] pps_core: Software ver. 5.3.6 - Copyright 2005-2007 Rodolfo Giometti <giometti@linux.it>
[    1.815361] PTP clock support registered
[    1.819418] EDAC MC: Ver: 3.0.0
[    1.822675] dmi: Firmware registration failed.
[    1.827155] bman-fbpr addr 0x00000008fe000000 size 0x1000000
[    1.832863] Bman err interrupt handler present
[    1.837655] Bman portal initialised, cpu 0
[    1.841822] Bman portal initialised, cpu 1
[    1.845995] Bman portal initialised, cpu 2
[    1.850161] Bman portal initialised, cpu 3
[    1.854277] Bman portals initialised
[    1.858629] Qman err interrupt handler present
[    1.863444] QMan: Allocated lookup table at ffff00000b0cd000, entry count 131073
[    1.871194] Qman portal initialised, cpu 0
[    1.875354] Qman portal initialised, cpu 1
[    1.879512] Qman portal initialised, cpu 2
[    1.883670] Qman portal initialised, cpu 3
[    1.887785] Qman portals initialised
[    1.891415] Bman: BPID allocator includes range 32:32
[    1.896520] Qman: FQID allocator includes range 256:256
[    1.901773] Qman: FQID allocator includes range 32768:32768
[    1.907406] Qman: CGRID allocator includes range 0:256
[    1.912708] Qman: pool channel allocator includes range 1025:15
[    1.918718] No USDPAA memory, no 'fsl,usdpaa-mem' in device-tree
[    1.924875] fsl-ifc 1530000.ifc: Freescale Integrated Flash Controller
[    1.931445] fsl-ifc 1530000.ifc: IFC version 1.4, 8 banks
[    1.937034] Advanced Linux Sound Architecture Driver Initialized.
[    1.943431] clocksource: Switched to clocksource arch_sys_counter
[    1.949619] VFS: Disk quotas dquot_6.6.0
[    1.953583] VFS: Dquot-cache hash table entries: 512 (order 0, 4096 bytes)
[    1.960545] pnp: PnP ACPI: disabled
[    1.966724] NET: Registered protocol family 2
[    1.971321] TCP established hash table entries: 32768 (order: 6, 262144 bytes)
[    1.978686] TCP bind hash table entries: 32768 (order: 7, 524288 bytes)
[    1.985649] TCP: Hash tables configured (established 32768 bind 32768)
[    1.992293] UDP hash table entries: 2048 (order: 4, 65536 bytes)
[    1.998351] UDP-Lite hash table entries: 2048 (order: 4, 65536 bytes)
[    2.004903] NET: Registered protocol family 1
[    2.009391] RPC: Registered named UNIX socket transport module.
[    2.015348] RPC: Registered udp transport module.
```

```
[    2.020076] RPC: Registered tcp transport module.
[    2.024802] RPC: Registered tcp NFSv4.1 backchannel transport module.
[    2.031331] Trying to unpack rootfs image as initramfs...
[    2.036908] rootfs image is not initramfs (no cpio magic); looks like an initrd
[    2.052644] Freeing initrd memory: 9348K
[    2.056916] hw perfevents: enabled with armv8_cortex_a72 PMU driver, 7 counters available
[    2.065365] kvm [1]: 8-bit VMID
[    2.069077] kvm [1]: vgic interrupt IRQ1
[    2.073065] kvm [1]: Hyp mode initialized successfully
[    2.079417] audit: initializing netlink subsys (disabled)
[    2.084897] audit: type=2000 audit(1.984:1): state=initialized audit_enabled=0 res=1
[    2.085054] workingset: timestamp_bits=44 max_order=20 bucket_order=0
[    2.099371] squashfs: version 4.0 (2009/01/31) Phillip Lougher
[    2.105333] NFS: Registering the id_resolver key type
[    2.110421] Key type id_resolver registered
[    2.114624] Key type id_legacy registered
[    2.118655] nfs4filelayout_init: NFSv4 File Layout Driver Registering...
[    2.125443] fuse init (API version 7.26)
[    2.129438] 9p: Installing v9fs 9p2000 file system support
[    2.136116] Block layer SCSI generic (bsg) driver version 0.4 loaded (major 245)
[    2.143593] io scheduler noop registered
[    2.147549] io scheduler cfq registered (default)
[    2.152276] io scheduler mq-deadline registered
[    2.156828] io scheduler kyber registered
[    2.166137] OF: PCI: host bridge /soc/pcie@3500000 ranges:
[    2.171660] OF: PCI:    IO 0x4800010000..0x480001ffff -> 0x00000000
[    2.177965] OF: PCI:   MEM 0x4840000000..0x487fffffff -> 0x40000000
[    2.184355] layerscape-pcie 3500000.pcie: PCI host bridge to bus 0000:00
[    2.191095] pci_bus 0000:00: root bus resource [bus 00-ff]
[    2.196610] pci_bus 0000:00: root bus resource [io  0x0000-0xffff]
[    2.202826] pci_bus 0000:00: root bus resource [mem 0x4840000000-0x487fffffff] (bus address 
[0x40000000-0x7fffffff])
[    2.223522] pci 0000:00:00.0: BAR 14: assigned [mem 0x4840000000-0x48401fffff]
[    2.230788] pci 0000:00:00.0: BAR 6: assigned [mem 0x4840200000-0x48402007ff pref]
[    2.238402] pci 0000:01:00.0: BAR 0: assigned [mem 0x4840000000-0x48401fffff 64bit]
[    2.246140] pci 0000:00:00.0: PCI bridge to [bus 01-ff]
[    2.251392] pci 0000:00:00.0:   bridge window [mem 0x4840000000-0x48401fffff]
[    2.258667] pcieport 0000:00:00.0: Signaling PME with IRQ 68
[    2.264408] pcieport 0000:00:00.0: AER enabled with IRQ 69
[    2.270011] OF: PCI: host bridge /soc/pcie@3600000 ranges:
[    2.275532] OF: PCI:    IO 0x5000010000..0x500001ffff -> 0x00000000
[    2.281836] OF: PCI:   MEM 0x5040000000..0x507fffffff -> 0x40000000
[    2.288198] layerscape-pcie 3600000.pcie: PCI host bridge to bus 0001:00
[    2.294939] pci_bus 0001:00: root bus resource [bus 00-ff]
[    2.300454] pci_bus 0001:00: root bus resource [io  0x10000-0x1ffff] (bus address [0x0000-0xffff])
[    2.309466] pci_bus 0001:00: root bus resource [mem 0x5040000000-0x507fffffff] (bus address 
[0x40000000-0x7fffffff])
[    2.331486] pci 0001:00:00.0: BAR 14: assigned [mem 0x5040000000-0x50400fffff]
[    2.338751] pci 0001:00:00.0: BAR 13: assigned [io  0x10000-0x10fff]
[    2.345140] pci 0001:00:00.0: BAR 6: assigned [mem 0x5040100000-0x50401007ff pref]
[    2.352755] pci 0001:01:00.0: BAR 1: assigned [mem 0x5040000000-0x504007ffff]
[    2.359942] pci 0001:01:00.0: BAR 6: assigned [mem 0x5040080000-0x50400bffff pref]
[    2.367555] pci 0001:01:00.0: BAR 0: assigned [mem 0x50400c0000-0x50400dffff]
[    2.374741] pci 0001:01:00.0: BAR 3: assigned [mem 0x50400e0000-0x50400e3fff]
[    2.381929] pci 0001:01:00.0: BAR 2: assigned [io  0x10000-0x1001f]
[    2.388242] pci 0001:00:00.0: PCI bridge to [bus 01-ff]
[    2.393495] pci 0001:00:00.0:   bridge window [io  0x10000-0x10fff]
[    2.399796] pci 0001:00:00.0:   bridge window [mem 0x5040000000-0x50400fffff]
[    2.407055] pcieport 0001:00:00.0: Signaling PME with IRQ 70
[    2.412795] pcieport 0001:00:00.0: AER enabled with IRQ 71
```

```
[    2.421618] Freescale LS2 console driver
[    2.425609] fsl-ls2-console: device fsl_mc_console registered
[    2.431417] fsl-ls2-console: device fsl_aiop_console registered
[    2.439990] Serial: 8250/16550 driver, 4 ports, IRQ sharing enabled
[    2.447306] console [ttyS0] disabled
[    2.450920] 21c0500.serial: ttyS0 at MMIO 0x21c0500 (irq = 38, base_baud = 21875000) is a 16550A
[    2.459775] console [ttyS0] enabled
[    2.459775] console [ttyS0] enabled
[    2.466758] bootconsole [uart8250] disabled
[    2.466758] bootconsole [uart8250] disabled
[    2.475360] 21c0600.serial: ttyS1 at MMIO 0x21c0600 (irq = 38, base_baud = 21875000) is a 16550A
[    2.484348] 21d0500.serial: ttyS2 at MMIO 0x21d0500 (irq = 39, base_baud = 21875000) is a 16550A
[    2.493336] 21d0600.serial: ttyS3 at MMIO 0x21d0600 (irq = 39, base_baud = 21875000) is a 16550A
[    2.502506] SuperH (H)SCI(F) driver initialized
[    2.507370] msm_serial: driver initialized
[    2.514850] brd: module loaded
[    2.520285] loop: module loaded
[    2.524652] ahci-qoriq 3200000.sata: AHCI 0001.0301 32 slots 1 ports 6 Gbps 0x1 impl platform mode
[    2.533617] ahci-qoriq 3200000.sata: flags: 64bit ncq sntf pm clo only pmp fbs pio slum part ccc sds
apst
[    2.543654] scsi host0: ahci-qoriq
[    2.547115] ata1: SATA max UDMA/133 mmio [mem 0x03200000-0x0320ffff] port 0x100 irq 50
[    2.556899] nand: device found, Manufacturer ID: 0x01, Chip ID: 0xac
[    2.563262] nand: AMD/Spansion S34MS04G2
[    2.567182] nand: 512 MiB, SLC, erase size: 128 KiB, page size: 2048, OOB size: 128
[    2.575170] Bad block table found at page 262080, version 0x01
[    2.581570] Bad block table found at page 262016, version 0x01
[    2.588074] fsl,ifc-nand 7e800000.nand: IFC NAND device at 0x7e800000, bank 0
[    2.595511] fsl-quadspi 1550000.quadspi: n25q512a (65536 Kbytes)
[    2.603204] libphy: Fixed MDIO Bus: probed
[    2.607623] tun: Universal TUN/TAP device driver, 1.6
[    2.613352] libphy: Freescale XGMAC MDIO Bus: probed
[    2.618367] libphy: Freescale XGMAC MDIO Bus: probed
[    2.624145] libphy: Freescale XGMAC MDIO Bus: probed
[    2.629286] libphy: Freescale XGMAC MDIO Bus: probed
[    2.634399] libphy: Freescale XGMAC MDIO Bus: probed
[    2.639525] libphy: Freescale XGMAC MDIO Bus: probed
[    2.644660] libphy: Freescale XGMAC MDIO Bus: probed
[    2.649785] libphy: Freescale XGMAC MDIO Bus: probed
[    2.654913] libphy: Freescale XGMAC MDIO Bus: probed
[    2.660023] libphy: Freescale XGMAC MDIO Bus: probed
[    2.673670] Freescale FM module, FMD API version 21.1.0
[    2.680924] Freescale FM Ports module
[    2.684583] fsl_mac: fsl_mac: FSL FMan MAC API based driver
[    2.690487] fsl_mac 1ae0000.ethernet: FMan MEMAC
[    2.695103] fsl_mac 1ae0000.ethernet: FMan MAC address: 00:04:9f:04:f4:5b
[    2.702163] fsl_mac 1ae8000.ethernet: FMan MEMAC
[    2.706778] fsl_mac 1ae8000.ethernet: FMan MAC address: 00:04:9f:04:f4:5c
[    2.713831] fsl_mac 1aea000.ethernet: FMan MEMAC
[    2.718446] fsl_mac 1aea000.ethernet: FMan MAC address: 00:04:9f:04:f4:5d
[    2.725510] fsl_mac 1af2000.ethernet: FMan MEMAC
[    2.730124] fsl_mac 1af2000.ethernet: FMan MAC address: 00:04:9f:04:f4:5e
[    2.736938] fsl_dpa: FSL DPAA Ethernet driver
[    2.743946] fsl_dpa: fsl_dpa: Probed interface eth0
[    2.751522] fsl_dpa: fsl_dpa: Probed interface eth1
[    2.759722] fsl_dpa: fsl_dpa: Probed interface eth2
[    2.764723] fsl_dpa soc:fsl,dpaa:ethernet@8: of_find_device_by_node(/soc/fman@1a00000/
ethernet@f0000) failed
[    2.774576] fsl_dpa: probe of soc:fsl,dpaa:ethernet@8 failed with error -22
[    2.784736] fsl_dpa: fsl_dpa: Probed interface eth3
```

```
[    2.789673] fsl_advanced: FSL DPAA Advanced drivers:
[    2.794634] fsl_proxy: FSL DPAA Proxy initialization driver
[    2.800352] fsl_oh: FSL FMan Offline Parsing port driver
[    2.805980] e1000: Intel(R) PRO/1000 Network Driver - version 7.3.21-k8-NAPI
[    2.813026] e1000: Copyright (c) 1999-2006 Intel Corporation.
[    2.818794] e1000e: Intel(R) PRO/1000 Network Driver - 3.2.6-k
[    2.824624] e1000e: Copyright(c) 1999 - 2015 Intel Corporation.
[    2.830860] e1000e 0001:01:00.0: Interrupt Throttling Rate (ints/sec) set to dynamic conservative mode
[    2.868518] ata1: SATA link down (SStatus 0 SControl 300)
[    2.895892] e1000e 0001:01:00.0 0001:01:00.0 (uninitialized): registered PHC clock
[    2.965337] e1000e 0001:01:00.0 eth4: (PCI Express:2.5GT/s:Width x1) 68:05:ca:1c:02:c4
[    2.973257] e1000e 0001:01:00.0 eth4: Intel(R) PRO/1000 Network Connection
[    2.980141] e1000e 0001:01:00.0 eth4: MAC: 3, PHY: 8, PBA No: E46981-008
[    2.986864] igb: Intel(R) Gigabit Ethernet Network Driver - version 5.4.0-k
[    2.993823] igb: Copyright (c) 2007-2014 Intel Corporation.
[    2.999409] igbvf: Intel(R) Gigabit Virtual Function Network Driver - version 2.4.0-k
[    3.007236] igbvf: Copyright (c) 2009 - 2012 Intel Corporation.
[    3.013454] sky2: driver version 1.30
[    3.017646] VFIO - User Level meta-driver version: 0.3
[    3.024196] ehci_hcd: USB 2.0 'Enhanced' Host Controller (EHCI) Driver
[    3.030721] ehci-pci: EHCI PCI platform driver
[    3.035173] ehci-platform: EHCI generic platform driver
[    3.040493] ehci-orion: EHCI orion driver
[    3.044574] ehci-exynos: EHCI EXYNOS driver
[    3.048829] ehci-msm: Qualcomm On-Chip EHCI Host Controller
[    3.054463] ohci_hcd: USB 1.1 'Open' Host Controller (OHCI) Driver
[    3.060641] ohci-pci: OHCI PCI platform driver
[    3.065095] ohci-platform: OHCI generic platform driver
[    3.070403] ohci-exynos: OHCI EXYNOS driver
[    3.074858] xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
[    3.080347] xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 1
[    3.088171] xhci-hcd xhci-hcd.0.auto: hcc params 0x0220f66d hci version 0x100 quirks 0x22010810
[    3.096883] xhci-hcd xhci-hcd.0.auto: irq 47, io mem 0x02f00000
[    3.103012] hub 1-0:1.0: USB hub found
[    3.106767] hub 1-0:1.0: 1 port detected
[    3.110789] xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
[    3.116273] xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 2
[    3.123931] xhci-hcd xhci-hcd.0.auto: Host supports USB 3.0  SuperSpeed
[    3.130728] hub 2-0:1.0: USB hub found
[    3.134482] hub 2-0:1.0: 1 port detected
[    3.138528] xhci-hcd xhci-hcd.1.auto: xHCI Host Controller
[    3.144014] xhci-hcd xhci-hcd.1.auto: new USB bus registered, assigned bus number 3
[    3.151826] xhci-hcd xhci-hcd.1.auto: hcc params 0x0220f66d hci version 0x100 quirks 0x22010810
[    3.160535] xhci-hcd xhci-hcd.1.auto: irq 48, io mem 0x03000000
[    3.166622] hub 3-0:1.0: USB hub found
[    3.170376] hub 3-0:1.0: 1 port detected
[    3.174390] xhci-hcd xhci-hcd.1.auto: xHCI Host Controller
[    3.179875] xhci-hcd xhci-hcd.1.auto: new USB bus registered, assigned bus number 4
[    3.187531] xhci-hcd xhci-hcd.1.auto: Host supports USB 3.0  SuperSpeed
[    3.194322] hub 4-0:1.0: USB hub found
[    3.198075] hub 4-0:1.0: 1 port detected
[    3.202117] xhci-hcd xhci-hcd.2.auto: xHCI Host Controller
[    3.207605] xhci-hcd xhci-hcd.2.auto: new USB bus registered, assigned bus number 5
[    3.215421] xhci-hcd xhci-hcd.2.auto: hcc params 0x0220f66d hci version 0x100 quirks 0x22010810
[    3.224132] xhci-hcd xhci-hcd.2.auto: irq 49, io mem 0x03100000
[    3.230218] hub 5-0:1.0: USB hub found
[    3.233972] hub 5-0:1.0: 1 port detected
[    3.237983] xhci-hcd xhci-hcd.2.auto: xHCI Host Controller
[    3.243471] xhci-hcd xhci-hcd.2.auto: new USB bus registered, assigned bus number 6
[    3.251126] xhci-hcd xhci-hcd.2.auto: Host supports USB 3.0  SuperSpeed
```

```
[    3.257918] hub 6-0:1.0: USB hub found
[    3.261672] hub 6-0:1.0: 1 port detected
[    3.265887] usbcore: registered new interface driver usb-storage
[    3.273713] i2c /dev entries driver
[    3.285513] rtc-pcf2127-i2c 1-0051: rtc core: registered rtc-pcf2127-i2c as rtc0
[    3.294177] at24 1-0052: 65536 byte 24c512 EEPROM, writable, 1 bytes/write
[    3.302305] at24 1-0053: 65536 byte 24c512 EEPROM, writable, 1 bytes/write
[    3.309197] i2c i2c-0: Added multiplexed i2c bus 1
[    3.314057] i2c i2c-0: Added multiplexed i2c bus 2
[    3.318915] i2c i2c-0: Added multiplexed i2c bus 3
[    3.323770] i2c i2c-0: Added multiplexed i2c bus 4
[    3.328560] pca954x 0-0077: registered 4 multiplexed busses for I2C switch pca9546
[    3.336184] IR NEC protocol handler initialized
[    3.340714] IR RC5(x/sz) protocol handler initialized
[    3.345759] IR RC6 protocol handler initialized
[    3.350285] IR JVC protocol handler initialized
[    3.354809] IR Sony protocol handler initialized
[    3.359421] IR SANYO protocol handler initialized
[    3.364119] IR Sharp protocol handler initialized
[    3.368820] IR MCE Keyboard/mouse protocol handler initialized
[    3.374647] IR XMP protocol handler initialized
[    3.379355] ptp_qoriq: device tree node missing required elements, try automatic configuration
[    3.388095] pps pps0: new PPS source ptp1
[    3.394762] ina2xx 1-0040: power monitor ina220 (Rshunt = 1000 uOhm)
[    3.404835] imx2-wdt 2ad0000.watchdog: timeout 60 sec (nowayout=0)
[    3.411659] qoriq_cpufreq: Freescale QorIQ CPU frequency scaling driver
[    3.418534] sdhci: Secure Digital Host Controller Interface driver
[    3.424719] sdhci: Copyright(c) Pierre Ossman
[    3.429204] Synopsys Designware Multimedia Card Interface Driver
[    3.435872] sdhci-pltfm: SDHCI platform and OF driver helper
[    3.480450] mmc0: SDHCI controller on 1560000.esdhc [1560000.esdhc] using ADMA 64-bit
[    3.490349] platform caam_qi: Linux CAAM Queue I/F driver initialised
[    3.497044] caam 1700000.crypto: device ID = 0x0a11030100000000 (Era 8)
[    3.503658] caam 1700000.crypto: job rings = 3, qi = 1
[    3.511094] caam algorithms registered in /proc/crypto
[    3.517639] platform caam_qi: algorithms registered in /proc/crypto
[    3.524776] caam_jr 1710000.jr: registering rng-caam
[    3.529811] caam 1700000.crypto: caam pkc algorithms registered in /proc/crypto
[    3.537225] usbcore: registered new interface driver usbhid
[    3.542818] usbhid: USB HID core driver
[    3.546710] DPAA2-ETH: debugfs created
[    3.550991] Freescale USDPAA process driver
[    3.555170] fsl-usdpaa: no region found
[    3.559000] Freescale USDPAA process IRQ driver
[    3.564252] optee: probing for conduit method from DT.
[    3.569391] optee: api uid mismatch
[    3.573414] Netfilter messages via NETLINK v0.30.
[    3.578184] nf_conntrack version 0.5.0 (16384 buckets, 65536 max)
[    3.584356] nf_tables: (c) 2007-2009 Patrick McHardy <kaber@trash.net>
[    3.590882] nf_tables_compat: (c) 2012 Pablo Neira Ayuso <pablo@netfilter.org>
[    3.598187] ip_tables: (C) 2000-2006 Netfilter Core Team
[    3.603645] Initializing XFRM netlink socket
[    3.607940] NET: Registered protocol family 10
[    3.612742] Segment Routing with IPv6
[    3.616435] sit: IPv6, IPv4 and MPLS over IPv4 tunneling driver
[    3.622528] NET: Registered protocol family 17
[    3.626972] NET: Registered protocol family 15
[    3.627441] usb 1-1: new high-speed USB device number 2 using xhci-hcd
[    3.637943] Bridge firewalling registered
[    3.641951] Ebtables v2.0 registered
```

```
[    3.645589] 8021q: 802.1Q VLAN Support v1.8
[    3.649787] 9pnet: Installing 9P2000 support
[    3.654072] Key type dns_resolver registered
[    3.658559] registered taskstats version 1
[    3.662860] Btrfs loaded, crc32c=crc32c-generic
[    3.673746] rtc-pcf2127-i2c 1-0051: setting system clock to 2020-04-06 19:17:39 UTC (1586200659)
[    3.682621] ALSA device list:
[    3.685590]   No soundcards found.
[    3.689091] Waiting 3 sec before mounting root device...
[    3.772357] usb-storage 1-1:1.0: USB Mass Storage device detected
[    3.778617] scsi host1: usb-storage 1-1:1.0
[    3.827479] usb 4-1: new SuperSpeed USB device number 2 using xhci-hcd
[    3.852534] usb-storage 4-1:1.0: USB Mass Storage device detected
[    3.858750] scsi host2: usb-storage 4-1:1.0
[    4.800127] scsi 1:0:0:0: Direct-Access     SanDisk  Cruzer Blade     1.00 PQ: 0 ANSI: 6
[    4.808591] sd 1:0:0:0: [sda] 30464000 512-byte logical blocks: (15.6 GB/14.5 GiB)
[    4.817025] sd 1:0:0:0: [sda] Write Protect is off
[    4.822071] sd 1:0:0:0: [sda] Write cache: disabled, read cache: enabled, doesn't support DPO or FUA
[    4.836762] sd 1:0:0:0: [sda] Attached SCSI removable disk
[    4.864003] scsi 2:0:0:0: Direct-Access     SanDisk  Ultra            1.00 PQ: 0 ANSI: 6
[    4.872402] sd 2:0:0:0: [sdb] 60063744 512-byte logical blocks: (30.8 GB/28.6 GiB)
[    4.880694] sd 2:0:0:0: [sdb] Write Protect is off
[    4.885714] sd 2:0:0:0: [sdb] Write cache: disabled, read cache: enabled, doesn't support DPO or FUA
[    4.896181] random: fast init done
[    4.906915]  sdb: sdb1 sdb2 sdb3
[    4.911395] sd 2:0:0:0: [sdb] Attached SCSI removable disk
[    6.719479] RAMDISK: gzip image found at block 0
[    7.019379] EXT4-fs (ram0): mounted filesystem with ordered data mode. Opts: (null)
[    7.027048] VFS: Mounted root (ext4 filesystem) on device 1:0.
[    7.033019] devtmpfs: mounted
[    7.036482] Freeing unused kernel memory: 1344K
INIT: version 2.88 booting
Starting udev
[    7.125944] udevd[2374]: starting version 3.2.5
[    7.130682] random: udevd: uninitialized urandom read (16 bytes read)
[    7.137242] random: udevd: uninitialized urandom read (16 bytes read)
[    7.143737] random: udevd: uninitialized urandom read (16 bytes read)
[    7.152900] udevd[2375]: starting eudev-3.2.5
[    7.376937] EXT2-fs (sda): warning: mounting unchecked fs, running e2fsck is recommended
[    7.541898] EXT4-fs (sdb3): mounted filesystem with ordered data mode. Opts: (null)
[    7.554434] EXT4-fs (sdb2): mounted filesystem with ordered data mode. Opts: (null)
[    7.595406] EXT4-fs (ram0): re-mounted. Opts: (null)
Configuring packages on first boot....
(This may take several minutes. Please do not power off the machine.)
Running postinst /etc/rpm-postinsts/100-sysvinit-inittab...
INIT: Entering runlevel: 5un-postinsts exists during rc.d purge
Configuring network interfaces... done.
Starting Dropbear SSH server: Generating 2048 bit rsa key, this may take a while...
Public key portion is:
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCLvasBL3eOZGy212NfRvDYv6kGYftjZmIBJGhA+LVlAnDttOadAZYnvwPB/rb5/
ctRY4N5VJ6KeLW/FjgWjijgbJLEX+bkQpbvb0FJa+2sfWIWK7YRnla7beZxNIhtIxbyHMKohszusTNizkdo2bX/tJBRKTE7/Wv/
C9NvMBvSSJg2iUEDLcBWBpeX/jjRcIa4L8lObVihAZZqmZ6k5AoBfV/
jJh3oJ2PbZ93bmzG3fiKoKuCwW5N6iPhZiFEZx3tAn3Tj112nCgr7GUe3WAHQCoBXgUAaCngQXI0EyV7hS0Jh3nCLEJ75/
Z62kVg6Hs45gDcINq/7GR/90YZYo8RH root@TinyDistro
Fingerprint: sha1!! be:6b:43:de:9e:11:21:fa:ea:7d:4f:b1:73:b3:2a:63:6b:9b:0b:ba
dropbear.
Starting syslogd/klogd: done

NXP LSDK (NXP Reference Tiny Distro) 2.0 TinyDistro /dev/ttyS0
```

```
TinyDistro login: root
root@TinyDistro:~# cat /proc/cpuinfo
processor       : 0
BogoMIPS        : 50.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 2

processor       : 1
BogoMIPS        : 50.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 2

processor       : 2
BogoMIPS        : 50.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 2

processor       : 3
BogoMIPS        : 50.00
Features        : fp asimd evtstrm aes pmull sha1 sha2 crc32 cpuid
CPU implementer : 0x41
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0xd08
CPU revision    : 2

root@TinyDistro:~# uname -a
Linux TinyDistro 4.14.83-00006-g49bfece-dirty #12 SMP PREEMPT Thu Apr 4 15:53:32 IST 2019 aarch64 GNU/
Linux
root@TinyDistro:~# uname -r
4.14.83-00006-g49bfece-dirty
root@TinyDistro:~# cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  1:          0          0          0          0     GICv2  25 Level     vgic
  3:       1117       1104       1246       1116     GICv2  30 Level     arch_timer
  4:          0          0          0          0     GICv2  27 Level     kvm guest timer
  6:          0          0          0          0     GICv2 138 Level     arm-pmu
  7:          0          0          0          0     GICv2 139 Level     arm-pmu
  8:          0          0          0          0     GICv2 127 Level     arm-pmu
  9:          0          0          0          0     GICv2 129 Level     arm-pmu
11:         10          0          0          0     GICv2  75 Level     fsl-ifc
12:          4          0          0          0     GICv2 131 Level     1550000.quadspi
13:          0          0          0          0     GICv2  94 Level     mmc0
15:          0          0          0          0     GICv2  77 Level     bman-err, qman-err, fman-err
22:          0          0          0          0     GICv2 216 Level     QMan portal 3
23:          0          0          0          0     GICv2 218 Level     QMan portal 2
24:          0          0          0          0     GICv2 220 Level     QMan portal 1
25:          0          0          0          0     GICv2 222 Level     QMan portal 0
```

```
32:          0          0          0          0     GICv2 217 Level     BMan portal 3
33:          0          0          0          0     GICv2 219 Level     BMan portal 2
34:          0          0          0          0     GICv2 221 Level     BMan portal 1
35:          0          0          0          0     GICv2 223 Level     BMan portal 0
37:        121          0          0          0     GICv2  88 Level     2180000.i2c
38:        402          0          0          0     GICv2  86 Level     ttyS0
44:          0          0          0          0     GICv2 118 Level     29d0000.ftm0
45:          0          0          0          0     GICv2 115 Level     2ad0000.watchdog
46:          0          0          0          0     GICv2 135 Level     eDMA
47:        282          0          0          0     GICv2  92 Level     xhci-hcd:usb1
48:        838          0          0          0     GICv2  93 Level     xhci-hcd:usb3
49:          0          0          0          0     GICv2  95 Level     xhci-hcd:usb5
50:          0          0          0          0     GICv2 101 Level     ahci-qoriq[3200000.sata]
68:          0          0          0          0     GICv2 159 Level     PCIe PME
69:          0          0          0          0     GICv2 160 Level     aerdrv
70:          0          0          0          0     GICv2 193 Level     PCIe PME
71:          0          0          0          0     GICv2 194 Level     aerdrv
72:          0          0          0          0     GICv2  76 Level     fman, ptp_qoriq
78:          2          0          0          0     GICv2 103 Level     1710000.jr
79:          0          0          0          0     GICv2 104 Level     1720000.jr
80:          0          0          0          0     GICv2 105 Level     1730000.jr
IPI0:       1963       1485       2265       1375       Rescheduling interrupts
IPI1:         27        546        523        542       Function call interrupts
IPI2:          0          0          0          0       CPU stop interrupts
IPI3:          0          0          0          0       CPU stop (for crash dump) interrupts
IPI4:          0          0          0          0       Timer broadcast interrupts
IPI5:          0          0          0          0       IRQ work interrupts
IPI6:          0          0          0          0       CPU wake-up interrupts
Err:           0
root@TinyDistro:~#
```

3.1.4.2 Option 1: Deploy FRWY-LS1046A BSP images using removable storage device

Given below are the steps to deploy FRWY-LS1046A BSP images on the FRWY-LS1046A board using a removable storage device (micro-SD). Option 1 can be used when the user has access to a local Linux host machine and to a local FRWY-LS1046A board. For this option, the storage device (micro-SD using adapter if needed) is plugged into the host machine and programmed. Once the programming is complete, the storage device is removed from the host machine and then inserted into the FRWY-LS1046A board.

1. Connect the removable storage device to the local Linux host machine.
2. The composite firmware, boot partition, and rootfs images that were assembled during “Download and assemble FRWY-LS1046A BSP images” will be available under `flexbuild_<version>` directory on the host machine. The images are named as follows:
   - Composite firmware:

     ```
     firmware_ls1046afrwy_uboot_sdboot.img
     ```

   - Boot partition:

     ```
     bootpartition_LS_arm64_<version> or bootpartition_LS_arm64_<version>.tgz (64-bit)
     ```

   - rootfs:

     ```
     rootfs_ubuntu_bionic_LS_arm64 or rootfs_ubuntu_bionic_LS_arm64.tgz (64-bit)
     ```

3. Set up the environment for `flex-installer` to run:

```
$ cd flexbuild
$ source setup.env
```

4. Execute `flex-installer` with appropriate arguments to deploy FRWY-LS1046A BSP images to the storage device.

   For example:

```
flex-installer -b bootpartition_<arch>_<version>.tgz -r rootfs_ubuntu_<version>.tgz -f
firmware_ls1046afrwy_uboot_sdboot.img -d /dev/mmcblk0
```

WARNING

If the Linux host machine supports read/write SD card directly without an extra SD card reader device, then the device name of micro-SD card is typically `mmcblk0`.

5. After a successful installation, the message "installation finished successfully" appears. Execute the following command to unmount all mounted partitions of the target device.

   Example:

```
$ sudo umount /run/media/mmcblk0
```

6. Unplug removable storage device (micro-SD card) from the Linux host and plug it into the FRWY-LS1046A board.
7. Ensure that the DIP switch settings on the board enable booting from micro-SD card (see "Onboard switch options" in the preceding section for switch settings).
8. Power cycle the board. The system will automatically boot the BSP Ubuntu distro available on the micro-SD card.

Update/flash image in QSPI NOR flash

As an alternative option, the user can boot Linux with tiny distro from QSPI NOR flash using the following steps:

1. Power on the board and stop autoboot to enter the U-Boot prompt.
2. Program/update the composite firmware to QSPI NOR flash as follows:

   a. Load firmware image from storage media:

Storage media	Commands in U-Boot

USB	`=> usb start`
	`=> load usb 0:2 a0000000`
	`firmware_ls1046afrwy_uboot_qspiboot.img`

Micro-SD	`=> load mmc 0:2 a0000000` `firmware_ls1046afrwy_uboot_qspiboot.img`

   b. Execute the following command to program the QSPI NOR flash:

```
=> sf probe 0:0; sf erase 0 +$filesize; sf write a0000000 0 $filesize
```

3. Ensure that the DIP switch settings on the board enable booting from QSPI NOR flash (see "Onboard switch options" in the preceding section for switch settings).
4. Power cycle the board. The system will automatically boot the BSP tiny distro available on the QSPI NOR flash.

3.1.4.3 Option 2: Deploy BSP images directly to storage device on a board

If the user does not have a local Linux host machine that can connect to a removable media, and/or the user does not have local access to a FRWY-LS1046A board, then the user can directly deploy BSP images to the storage device on a FRWY-LS1046A board by accessing the board remotely. For this option, a network connection to the FRWY-LS1046A board is required.

1. Reboot the FRWY-LS1046A board from QSPI NOR flash and let it boot automatically.
   - If the DIP switch settings are configured for QSPI NOR flash, then the board will boot from QSPI NOR flash using the composite firmware present on the board.
2. Log in to TinyDistro as “root” and bring up a network interface:

```
$ udhcpc -i <port name in TinyDistro>
```

3. Use `flex-installer` to create and format the partitions for storage device (micro-SD card).

Storage device	Commands in Linux
Micro-SD card	`$ flex-installer -i pf -d /dev/mmcblk0`

WARNING

If the Linux host machine supports read/write SD card directly without an extra SD card reader device, then the device name of micro-SD card is typically `mmcblk0`.

4. Download and deploy composite firmware image, `firmware_ls1046afrwy_uboot_sdboot.img`, and two tarballs (boot partition and Ubuntu userland) to micro-SD card. The composite image and tarballs were assembled while performing steps mentioned in the “Download and assemble FRWY-LS1046A BSP images” section.

Storage device	Commands in Linux
Micro-SD card	a. `$ cd /run/media/mmcblk0p3` b. Download `firmware_ls1046afrwy_uboot_sdboot.img`, `bootpartition_<arch>_<version>.tgz`, and `ubuntu_<codename>_<arch>_rootfs_<timestamp>.tgz` using the wget or scp command. c. `$ flex-installer -b bootpartition_<arch>_lts_<version>.tgz -r rootfs_ubuntu_<codename>_<arch>_<timestamp>.tgz -f firmware_ls1046afrwy_uboot_sdboot.img -d /dev/mmcblk0`

5. Ensure that the DIP switch settings on the board enable booting from micro-SD card (see “Onboard switch options” in the preceding section for switch settings).
6. Power cycle the board. The system will automatically boot Ubuntu userland.
7. Log in using root/root or user/user.

3.2 How to build FRWY-LS1046A BSP with Flexbuild

Flexbuild provides command line interface, `flex-builder`, for various build scenarios. The FRWY-LS1046A BSP Quick Start on page 9 section introduces how to build the FRWY-LS1046A BSP distro userland with prebuilt boot partition and component tarballs for quick deployment on the target board. The current section explains how to build FRWY-LS1046A BSP with Flexbuild.

Go to www.nxp.com to download Flexbuild source tarball in the name format, `flexbuild_<version>.tgz`:

```
$  tar xvzf flexbuild_<version>.tgz
$  cd flexbuild
$  source setup.env
$  flex-builder -h
```

Build Trusted Firmware-A (TF-A) with RCW and U-Boot in Flexbuild

Layerscape platforms support TF-A, which provides a reference implementation of secure world software for Armv7-A and Armv8-A.

`flex-builder` can automatically build the dependent RCW, U-Boot, OPTEE, and CST when building TF-A to generate BL2 and FIP images for Layerscape platforms.

Use the commands below to build U-Boot-based TF-A image in Flexbuild:

```
Usage:
$ flex-builder -c atf  -m <machine>  -b <boottype> [-s]

Examples:
$ flex-builder -c atf -m ls1046afrwy -b qspi           # build uboot-based TF-A image for qspi-NOR boot
on FRWY-LS1046A
$ flex-builder -c atf -m ls1046afrwy -b sd             # build uboot-based TF-A image for micro-SD boot
on FRWY-LS1046A
$ flex-builder -c atf -m ls1046afrwy -b qspi -s         # build uboot-based TF-A image for qspi-NOR
secure boot on FRWY-LS1046A
```

NOTE

If the user wants to modify RCW source code (in `packages/firmware/rcw`) or specify RCW binary that is different from default one, then the user needs to reconfigure the `rcw_<boottype>` variable in `configs/board/<machine>/manifest` and run the `flex-builder -c rcw` command to regenerate RCW binary.

NOTE

If RCW or U-Boot source code is updated since last build, ensure to clean the obsolete image by `rm -rf build/firmware/u-boot/<machine>` and/or `rm -rf build/firmware/rcw/<machine>` command, and then rebuild TF-A by `flex-builder -c atf -m <machine> -b <boottype>` command.

NOTE

The '-s' option is used for secure boot; OPTEE and FUSE_PROVISIONING are not enabled by default, change `CONFIG_BUILD_OPTEE=n` to `y` and/or change `CONFIG_FUSE_PROVISIONING=n` to `y` in `configs/build_lsdk.cfg` to enable it, if required.

Build Linux kernel with Flexbuild

Building FRWY-LS1046A BSP kernel in standalone mode is explained in Configuring and building Linux kernel on page 88. Alternatively, FRWY-LS1046A BSP kernel can be built easily using `flex-builder`. This section explains how to build Linux kernel using `flex-builder`.

To build kernel using the default tree/branch/tag configurations specified in `configs/build_lsdk.cfg`:

```
$ flex-builder -c linux -a arm64               # for 64-bit mode of all Armv8 Layerscape platforms by
default
```

To build kernel with specified tree/branch/tag and additional fragment config:

```
Usage:
$ flex-builder -c linux:<kernel-repo>:<branch|tag> -a arm64 -B fragment:<custom>.config

Examples:
$ flex-builder -c linux:linux:linux-4.14-ls1046afrwy-bsp0.1 -a arm64 -B fragment:lttng.config
$ flex-builder -c linux:linux:ls1046afrwy-bsp0.1 -a arm64
```

The user can put a custom kernel fragment config file (for example, named as `ls1046afrwy.config`) in `flexbuild/packages/linux/<kernel-repo>/arch/arm64/configs` directory, then run the command below to compile kernel as per the default defconfig, `lsdk.config`, and the additional `ls1046afrwy.config`:

```
$ flex-builder -c linux -a arm64 -B fragment:ls1046afrwy.config
```

Platform	Command for building Linux
FRWY-LS1046A, 64-bit	`$ flex-builder -c linux:custom -a arm64` (optional, customize kernel config in interactive menu) `$ flex-builder -c linux -a arm64`

Build FRWY-LS1046A BSP composite firmware and boot partition

The FRWY-LS1046A BSP composite firmware consists of TF-A BL2, TF-A BL3 FIP firmware, bootloader environment, secure headers, Ethernet MAC/PHY firmware, dtb, kernel and tiny ramdisk rfs, and so on. This composite firmware can be programmed at offset 0x0 in flash device or at offset block #8 in micro-SD card.

To generate FRWY-LS1046A BSP composite firmware for Layerscape platform, run the following command:

```
Usage:
$ flex-builder -i mkfw -m <machine> -b <boottype> [-B <bootloader>] [-s]

Examples:
$ flex-builder -i mkfw -m ls1046afrwy -b qspi
  firmware_ls1046afrwy_uboot_qspiboot.img will be generated.

$ flex-builder -i mkfw -m ls1046afrwy -b qspi -s
  firmware_ls1046afrwy_uboot_qspiboot_secure.img will be generated.

$ flex-builder -i mkfw -m ls1046afrwy -b sd
  firmware_ls1046afrwy_uboot_sdboot.img will be generated.

$ flex-builder -i mkfw -m ls1046afrwy -b sd -s
  firmware_ls1046afrwy_uboot_sdboot_secure.img will be generated.
```

To generate FRWY-LS1046A BSP boot partition tarball, run the following command:

```
$ flex-builder -i mkbootpartition -a arm64
or
$ flex-builder -i mkbootpartition -a arm64 -s (for secure boot)
```

The command above will generate all required images, including kernel image, dtb, distro boot script, `flex_linux_<arch>.itb`, small ramdiskrfs, and so on. If any dependency exists on any other component, then `flex-builder` will automatically build that component.

How to build application components in Flexbuild

The following commands are some examples of building application components.

```
Usage:
$ flex-builder -c <component> -a <arch>  # build single application component for specified <arch>

Examples:
$ flex-builder -c apps          # build all apps components for arm64 arch
$ flex-builder -c fmc           # build fmc component for arm64 arch
$ flex-builder -c dpdk          # build dpdk component for arm64 arch
$ flex-builder -c openssl       # build openssl component for arm64 arch
$ flex-builder -c cst           # build cst component, needed for secure boot
(arm64 is the default arch if -a <arch> is not specified)
```

To add new application component in Flexbuild, follow the steps below:

1. Add new <component> to apps_repo_list and set `CONFIG_BUILD_<component>=y` in `configs/build_xx.cfg`.
2. Configure url/branch/tag/commit information for new <component_name> in `configs/build_xx.cfg`, default remote. Component git repository is specified by GIT_REPOSITORY_URL by default if <component>_url is not specified; the user can also directly create the new component git repository in `packages/apps` directory.
3. Add build support for new component in `packages/apps/Makefile`.
4. Run the `flex-builder -c <component-name> -a <arch>` command to build the new component.
5. Run the `flex-builder -i merge-component -a <arch>` command to merge the new component package into target distro userland.

How to update existing Linux kernel with new custom kernel for Ubuntu on target board in case of non-secure boot

The user can quickly install custom kernel and lib modules after Ubuntu had been deployed in micro-SD card on target board in case of non-secure boot, follow the steps below:

1. After modifying Linux kernel source code in `$FBDIR/packages/linux/<kernel-repo>` on demand, rebuild kernel as follows:

   ```
   $ flex-builder -c linux:custom (optional, to customize kernel config in interactive menu)
   $ flex-builder -c linux
   $ flex-builder -i mkbootpartition -a arm64
   ```

   The new kernel image tarball `$FBDIR/build/images/linux_4.14_LS_arm64_<timestamp>.tgz` will be generated.
2. Then, log in to Ubuntu system on target board, and update kernel image as follows:

   ```
   root@localhost:/# dhclient -i eth0
   root@localhost:~# cd /
   root@localhost:/# wget <path>/linux_4.14_LS_arm64_<timestamp>.tgz (or by scp command)
   root@localhost:/# tar xf linux_4.14_LS_arm64_<timestamp>.tgz
   root@localhost:/# reboot
   ```

   System will reboot and automatically boot to Ubuntu with new custom kernel.

Rebuild images after modifying source code of NXP user space components locally

Flexbuild supports building specific components after the source code is changed. Flexbuild then deploys the changes to the target board. Follow these steps:

1. Modify source code of user space components in the `packages/apps/<apps-component>` directory on demand.

2. Clean old build footprint under user space component:

```
$ make clean -C $FBDIR/packages/apps/<app-component>
$ rm -rf $FBDIR/build/apps/components_LS_arm64
```

3. Build the user space component and generate the compressed component tarball:

```
$ flex-builder -c <apps-component> -a arm64
$ flex-builder -i compressapps -a arm64
```

4. Upload and deploy the newly generated `app_components_LS_arm64.tgz` to target board on which Ubuntu distro was installed in micro-SD card:

```
=> run sd_bootcmd  (or run qspi_bootcmd to enter tiny distro Linux environment)
Download app_components_LS_arm64.tgz via wget or scp command
root@TinyDistro:~# tar xf app_components_LS_arm64.tgz
root@TinyDistro:~# cp -a components_LS_arm64/*  /run/media/mmcblk0p3
root@TinyDistro:~# reboot
```

How to generate a custom Ubuntu root filesystem with custom additional package list on x86 host machine

In Flexbulid, two default additional package lists are available for Ubuntu/Debian: `additional_packages_list_moderate` and `additional_packages_list_tiny`:

```
$ flex-builder -i mkrfs -a arm64   (as per additional_packages_list_moderate with more packages for
Ubuntu rootfs by default)
$ flex-builder -i mkrfs -r ubuntu:tiny -a <arch>   (as per additional_packages_list_tiny with less
packages for Ubuntu rootfs)
$ flex-builder -i mkrfs -r ubuntu -a <arch> -B <custom_packages_list>
```

To install a new package to `build/rfs/rootfs_ubuntu_bionic_LS_arm64` filesystem, run the following commands:

```
$ sudo chroot build/rfs/rootfs_ubunutu_bionic_LS_arm64
$ apt-get install <new_package_name>
# exit
```

How to install distro to micro-SD / USB storage device

Use the FRWY-LS1046A BSP `flex-installer` to install all the release binaries and distro userland on a storage media (for example, micro-SD card, USB disk) on the Linux host machine or on the target board. Follow the instructions below:

1. Plug micro-SD / USB storage device to Linux host machine or target board.
2. Install FRWY-LS1046A BSP distro:
    - If the prebuilt distro tarball generated by Flexbuild is available on Linux host machine, then run the following command:

    ```
    $ flex-installer -b bootpartition_LS_arm64_lts_4.14.tgz -r rootfs_ubuntu_bionic_LS_arm64.tgz -
    m ls1046afrwy -d /dev/sdx
    ```

    NOTE

    In the above command, replace `sdx` with actual device name on the host machine, for example `sdb`, `sdc`, or `mmcblk0`.

- If the user wants to modify source code and build a custom FRWY-LS1046A BSP distro with Flexbuild, then following commands need to be run on the Linux host machine:

```
$ flex-builder -c linux:custom -a <arch>  # customize kernel config in interactive menu
$ flex-builder -c linux -a <arch>
$ flex-builder -i mkfw -m <machine> -b <boottype>
$ flex-builder -i mkrfs -a <arch>
$ flex-builder -c apps -a <arch>
$ flex-builder -i merge-component -a <arch>
$ flex-builder -i mkbootpartition -a <arch>
$ flex-installer -b build/images/bootpartition_LS_arm64_lts_4.14 -r build/rfs/
rootfs_ubuntu_bionic_LS_arm64 -d /dev/sdx
```

- If the user wants to install distro rootfs directly on the micro-SD / USB disk on the FRWY-LS1046A board on which Linux is unavailable, then prebuilt image, `lsdk_linux_<arch>_LS_tiny.itb`, needs to be downloaded as follows:

```
$ wget https://www.nxp.com/lgfiles/sdk/ls1046afrwy_bsp_01/lsdk_linux_arm64_LS_tiny.itb
```

- Optionally, the prebuilt image can be built locally using the following command:

```
$ flex-builder -i mklinux -a <arch> to generate lsdk_linux_arm64_LS_tiny.itb
```

- Put `lsdk_linux_arm64_LS_tiny.itb` to a TFTP service directory, and then download it to the target board from U-Boot prompt as follows:

```
=> tftp a0000000 lsdk_linux_arm64_LS_tiny.itb
=> bootm a0000000#<board-name>
Note: Make sure to update bootargs as shown below
setenv bootargs console=ttyS0,115200 root=/dev/ram0 rw earlycon=uart8250,mmio,0x21c0500
ramdisk_size=20000000 rootdelay=3
```

  In the above command, <board-name> is ls1046afrwy.

- After booting and logging in to Linux on the target board, download the prebuilt distro tarballs generated by Flexbuild and install them using the following commands:

```
$ flex-installer -i pf -d /dev/sdx
$ cd /run/media/{mmcblk0p3 or sdx3}, then download distro images to micro-SD/USB storage
disk via wget or scp command
$ flex-installer -b bootpartition_LS_arm64_lts_<version>.tgz -r
rootfs_ubuntu_bionic_LS_arm64.tgz -d /dev/sdX
```

3. Power on or reboot the target board after finishing the distro installation. The system will enter boot loader (U-Boot), will automatically scan boot configuration script from the attached micro-SD / USB disk, and will boot the target FRWY-LS1046A BSP distro if found; otherwise, it will fall back to boot from QSPI NOR flash with tiny ramdisk distro.

How to program firmware to micro-SD / QSPI NOR flash media

- **Program firmware to micro-SD card:**
  1. Download the prebuilt firmware image / generate image:
     — **Option 1:** Download the prebuilt image using the `wget` command:

```
$ wget https://www.nxp.com/lgfiles/sdk/ls1046afrwy_bsp_01/
firmware_ls1046afrwy_uboot_sdboot.img
```

— **Option 2:** To generate `firmware_ls1046afrwy_uboot_sdboot.img` locally, run the following command:

```
$ flex-builder -i mkfw -m ls1046afrwy -b sd
```

2. Program `firmware_<machine>_uboot_sdboot.img` to micro-SD card:

— **Under U-Boot:**

a. Program image:

```
=> load mmc 0:2 a0000000 firmware_ls1046afrwy_uboot_sdboot.img
=> mmc write a0000000 8 1fff8
```

b. Power cycle the board.

NOTE

Ensure that switch settings on the board are for SD boot.

— **Under Linux:**

```
$ flex-installer -f firmware_ls1046afrwy_uboot_sdboot.img -d /dev/mmcblk0
```

- **Program firmware to QSPI NOR flash:**

1. Download the image using one of the following options:

— **Option 1:** Load prebuilt image from micro-SD card:

```
=> load mmc 0:2 a0000000 firmware_ls1046afrwy_uboot_qspiboot.img
```

— **Option 2:** Download the prebuilt image using the `wget` command:

```
$ wget https://www.nxp.com/lgfiles/sdk/ls1046afrwy_bsp_01/
firmware_ls1046afrwy_uboot_qspiboot.img
```

— **Option 3:** To generate `firmware_ls1046afrwy_uboot_qspiboot.img` locally, run the following command:

```
$ flex-builder -i mkfw -m ls1046afrwy -b qspi
```

2. Program `firmware_ls1046afrwy_uboot_qspiboot.img` to QSPI NOR flash:

```
=> sf probe 0:0
=> sf erase 0 +$filesize && sf write 0xa0000000 0 $filesize
```

3.3 Secure boot

3.3.1 Hardware Pre-Boot Loader (PBL) based platforms

3.3.1.1 Introduction

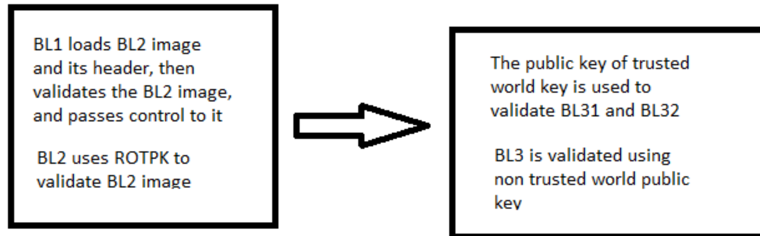
This section is intended for end-users to demonstrate the image validation process. The image validation can be split into stages, where each stage performs a specific function and validates the subsequent stage before passing control to that stage.

Chain Of Trust:

CoT starts from a set of implicitly trusted components.

The following images are included in the CoT:

- BL1
- BL2
- BL31
- BL32
- BL33
- Linux



For more details on the CoT refer trusted-board-boot.rst in the TF-A repository

3.3.1.2 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.

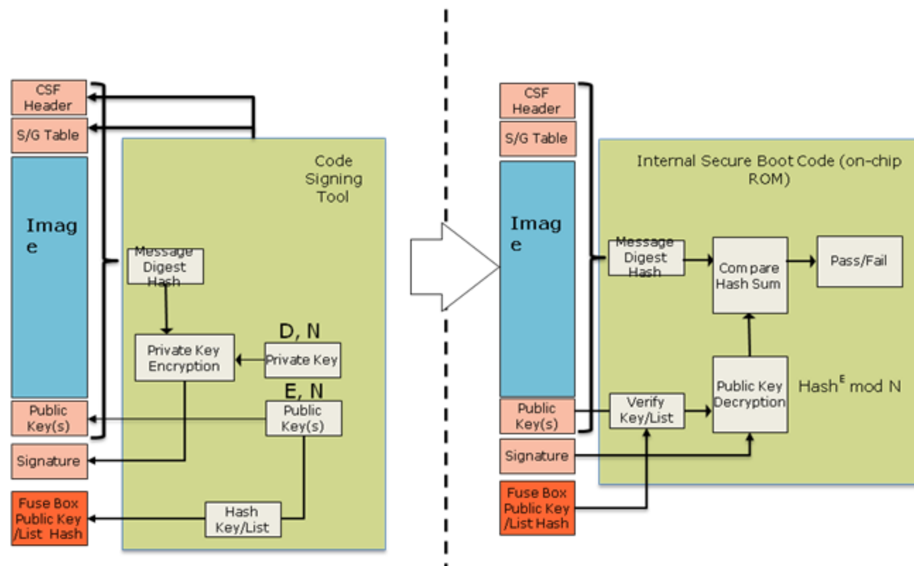


Figure 2. Secure boot process

3.3.1.3 Pre-boot phase

When the processor is powered on, reset control logic blocks all device activities (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.

Pre-Boot 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 does not 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.

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), NXP 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.

3.3.1.4 ISBC phase

3.3.1.4.1 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 BOOT ROM. The instructions inside the Internal BOOT ROM are NXP 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 (External Secure Boot Code) 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 do not 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 does not 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 NXP 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 do not 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 form 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.

3.3.1.4.2 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 cannot 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.

3.3.1.4.3 Key revocation

Trust Architecture 2.x 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 (SRK's).

OEM can use either a single key or a list of upto 4 SRK's in the Trust Arch v2.x devices.

In the NXP Code Signing Tool (CST), the OEM defines whether the device uses a single SRK, or offers a list of SRK's. If using a single SRK, 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 SRK's for trust arch v2.x onwards platforms and can calculate 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_OSPR). 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 an SRK 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 SRK does not have an associated revocation fuse.

3.3.1.4.4 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).

3.3.1.4.5 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 NXP Trust Architecture documentation.

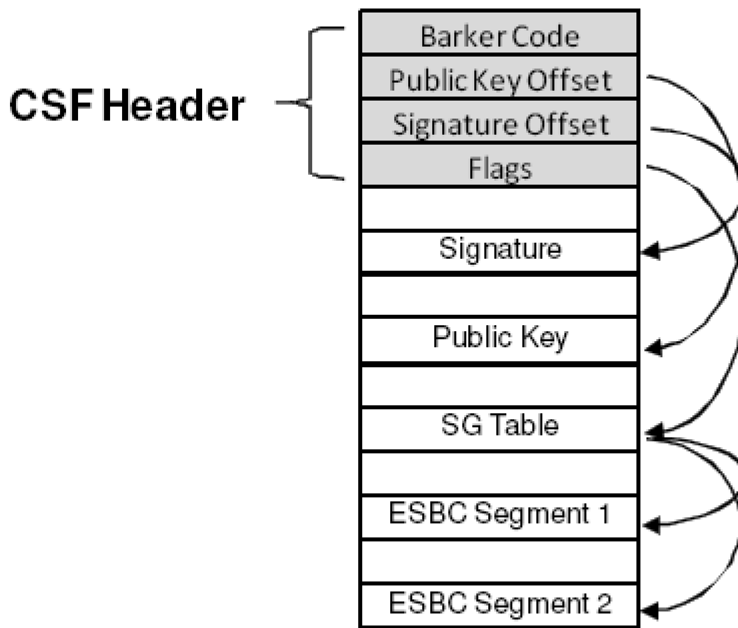


Figure 3. ESBC with CSF header

3.3.1.5 ESBC phase

Unlike ISBC, which is an internal ROM and unchangeable, ESBC is NXP – supplied reference code, and can be changed by OEMs. ESBC is the BL2 image, which is signed using private key. That image then loads a FIP image that includes, BL31 (EL3 runtime software) , BL32(optional image for platform storage) and BL33 (Uboot) to DDR and their headers to DDR, then validates these images.

BL33 (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 BL33 (U-Boot) image and is called default environment. This default environment cannot be stored on flash devices. User won't be able to edit this environment also as he cannot reach to U-Boot 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 NXP ESBC as it is provided or to use it as reference to modify their own secure boot system.

NOTE

On SoCs with Armv8 core (for example, LS1046A), 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 with respect to the 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).

3.3.1.5.1 Boot script

Boot script 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 are 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.

3.3.1.5.1.1 Where to place the boot script?

NXP's ESBC U-Boot expects the boot script to be loaded in flash as specified in address map. 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/arm/include/asm/fsl_secure_boot.h
```

Two new commands called esbc_validate and esbc_halt have been added in NXP ESBC U-Boot.

Two more commands are present, 'blob enc' and 'blob dec' for running Chain of Trust with confideniality.

3.3.1.5.1.2 Chain of Trust

Boot script contains information about the next level of images, For example, Linux, HV, and so on. 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 NXP ESBC as it is provided or to use it as reference to modify their own secure boot system.

The following figure shows the Chain of Trust established for validation with this ESBC U-Boot.

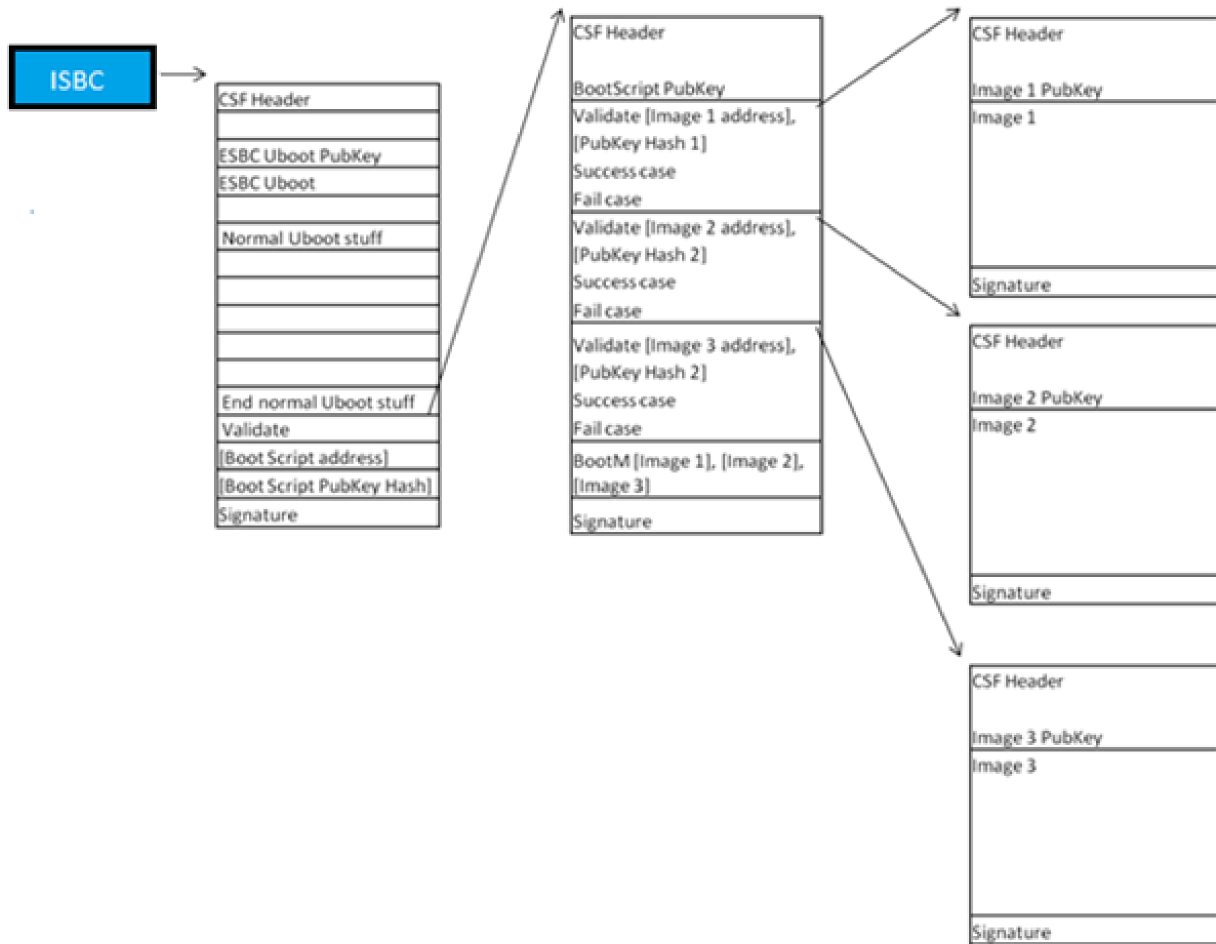


Figure 4. Secure boot flow (Chain of Trust)

3.3.1.5.1.2.1 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>
```

3.3.1.5.1.2.1.1 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 an 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

3.3.1.5.1.2.1.2 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 (for example, bootm not present in bootscript), core should just spin.

3.3.1.5.1.3 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 bootscript is used which creates a blob of the Linux images and saves them. After that, the system is booted after replacing the encap bootscript with decap bootscript which decapsulates the blobs and boot the Linux with the images.

The following figures show the Chain of Trust with confidentiality (Encapsulation and Decapsulation).

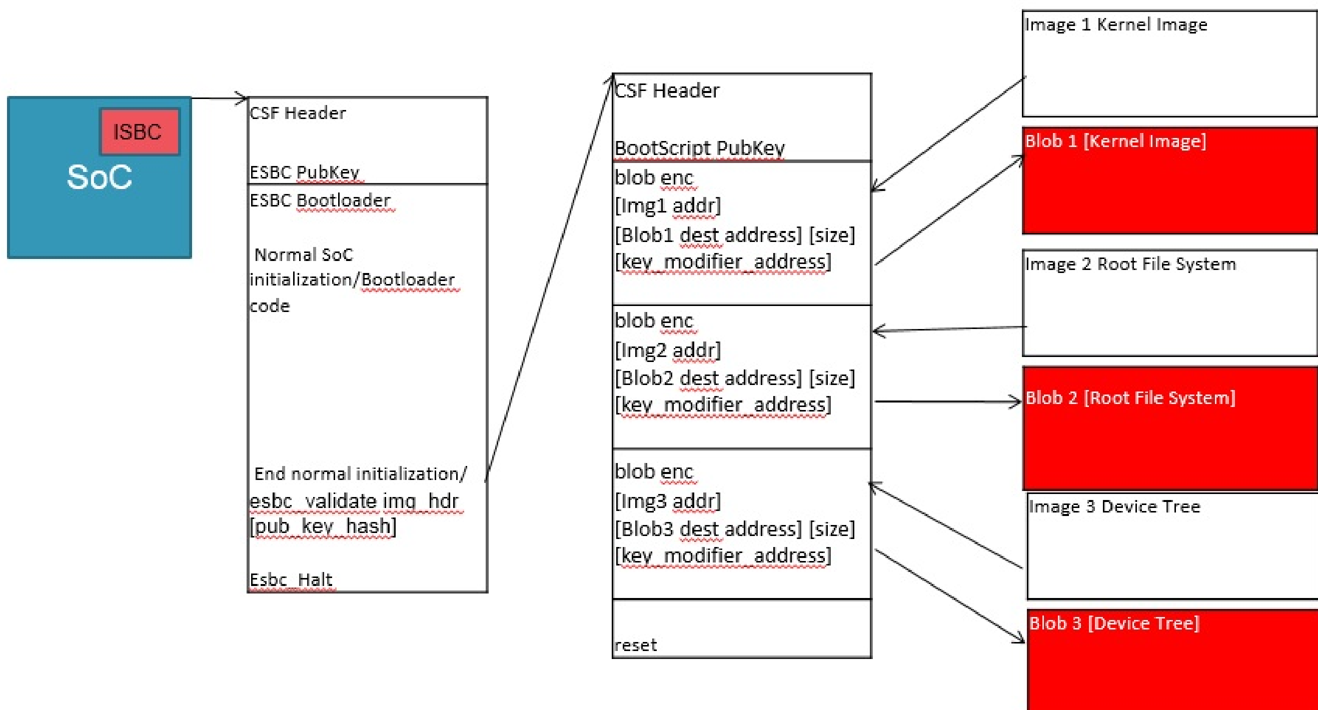


Figure 5. Chain of Trust with confidentiality (Encapsulation)

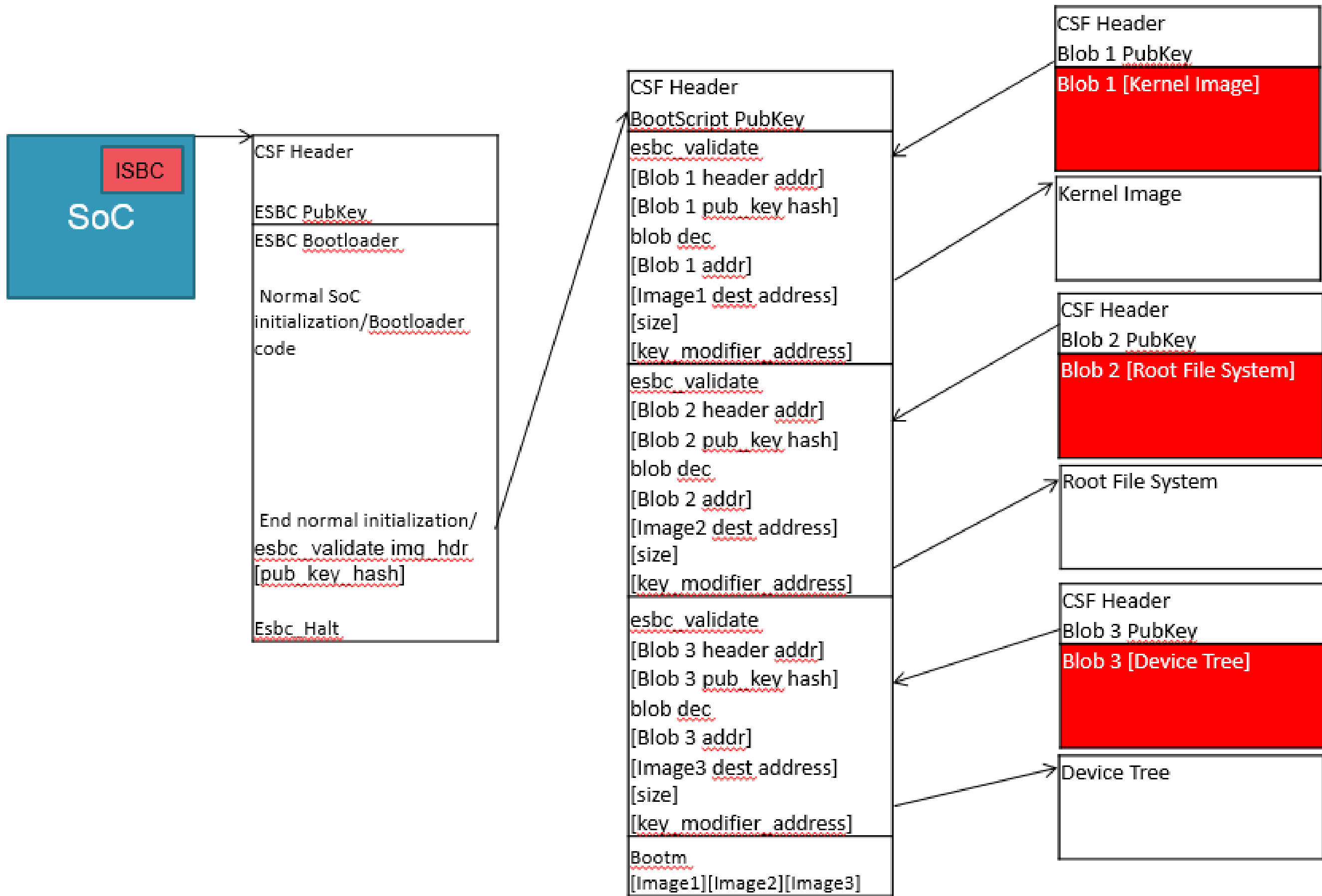


Figure 6. Chain of Trust with Confidentiality (Decapsulation)

3.3.1.5.1.3.1 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.

3.3.1.5.1.3.1.1 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 Imag1 size>
cp.b <Img1 dest addr> <encap Img1 addr> <encap Imag1 size>
```

```
blob enc <Img2 addr> <Img2 dest addr> <Img2 size> <key_modifier address>
erase <encap Img2 addr> +<encap Imag2 size>
cp.b <Img2 dest addr> <encap Img2 addr> <encap Imag2 size>

blob enc <Img3 addr> <Img3 dest addr> <Img3 size> <key_modifier address>
erase <encap Img3 addr> +<encap Imag3 size>
cp.b <Img3 dest addr> <encap Img3 addr> <encap Img3 size>

...
```

3.3.1.5.1.3.2 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.

3.3.1.5.1.3.2.1 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>
```

3.3.1.6 Next executable (Linux phase)

The bootloader finishes the platform initialization and passes 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, integrate RTIC to verify memory regions using Security Engine (SEC) during run time.

Chain of Trust

To execute Chain of Trust, follow the steps below:

Instruction on demo:

1. Make sure that the ISBC code validate the BL2 code.
2. BL2 loads FIP (BL31 (Secure Firmware) + BL32 (Optional) + BL33 (Uboot)) and validate them.
3. On successful validation, BL31 and BL32 passes for necessary configurations.
4. After configuration, U-Boot code runs and validates the boot script.
5. On successful validation of boot script, U-Boot code executes the commands.
6. The Boot script also contains commands to validate next level images, such as rootfs, Linux uImage, and device tree.
7. After boot script validating all the images, U-Boot executes the `bootm` command to pass control to Linux.

*Rest of the content in the topic remains the same.

Chain Of trust with confidentiality

Step 1: Creating blobs

1. Make sure that the ISBC code validate the BL2 code.
2. BL2 loads FIP (BL31 (Secure Firmware) + BL32 (Optional) + BL33 (Uboot)) and validate them.
3. On successful validation, BL31 and BL32 passes for necessary configurations.
4. After configuration, U-Boot code runs and validates the boot script.
5. On successful validation of boot script, U-Boot code executes the commands.
6. The boot script contains commands that encapsulates next level images, such as linux uImage and device tree.

Step 2: Decrypting blob and booting

1. Make sure that the ISBC code validate the BL2 code.
2. BL2 loads FIP (BL31 (Secure Firmware) + BL32 (Optional) + BL33 (Uboot)) and validate them.
3. On successful validation, BL31 and BL32 passes for necessary configurations.
4. After configuration, U-Boot code runs and validates the boot script.
5. On successful validation of boot script, U-Boot code executes the commands.
6. The boot script contains commands that decapsulate or decrypt next level images, such as rootfs, linux uImage and device tree.
7. After decryption, U-Boot code executes the `bootm` command in boot script to pass control to Linux.

*Rest of the content in the topic remains the same.

Running Secure Boot (Chain Of Trust)

Change wherever ESBC Uboot is used

3.3.1.7 Product execution

This section presents the steps need to be followed in order to properly run the software product according to its intended use and functionalities.

3.3.1.7.1 Introduction

Chain of Trust

This section presents the steps need to be followed in order to execute Chain of Trust.

Steps in the demo would be:

1. ISBC code would validate the BL2 image code.
2. On successful validation, BL2 code would run, which would then validate the BL31, BL32, BL33 images.
3. On successful validation of boot script by BL33 image, commands in boot script would be executed.
4. Boot script contains commands to validate next level images, that is, rootfs, Linux uImage, and device tree.
5. Once all the images are validated, bootm command in boot script would be executed which would pass control to Linux.

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 protyping phase, don't blow the ITS fuse, but use rcw with SB_EN = 1.

Blow other required fuses on the board. (OTPMK and SRK hash[1]) 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.

2. Flash all the generated images at locations as described in the address map.
   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.
3. 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.

Two additional features are provided in secure boot:

1. Chain of Trust with confidentiality
2. ISBC Key Extension

3.3.1.7.2 Chain of Trust with confidentiality

This section presents the steps need to be followed to execute Chain of Trust with confidentiality.

The demo is divided into two parts:

1. Creating or encrypting images in form of blobs.
2. Decrypting images, and booting from decrypted images.

Steps in the demo are:

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.

---

[1] 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. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to encapsulate next level images, that is rootfs, linux uImage 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 is stored.

$km - 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 or decrypt next level images, that is rootfs, linux uImage, 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 - 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.

3.3.1.7.2.1 Other images required for the demo

Apart from SDK images described above, the following images are also required:

1. Encap boot script

   Sample Boot script

```
load \$devtype \$devnum:2 \$kernelheader_addr_r /secboot_hdrs/ls1046ardb/hdr_linux.out;
esbc_validate \$kernelheader_addr_r;
load \$devtype \$devnum:2 \$fdtheader_addr_r /secboot_hdrs/ls1046ardb/hdr_dtb.out; esbc_validate
\$fdtheader_addr_r;
size \$devtype \$devnum:2 /vmlinuz; echo Encapsulating linux image;setenv key_addr 0x87000000; mw
\$key_addr $key_id_1;
setexpr \$key_addr \$key_addr + 0x4; mw \$key_addr $key_id_2;setexpr \$key_addr \$key_addr + 0x4;
mw \$key_addr $key_id_3;setexpr \$key_addr \$key_addr + 0x4; mw \$key_addr $key_id_4;
blob enc \$kernel_addr_r \$load_addr \$filesize \$key_addr; setexpr blobsize \$filesize
+ 0x30;echo Saving encrypted linux ;save \$devtype \$devnum:2 \$load_addr /vmlinuz \
$blobsize;size \$devtype \$devnum:2 /fsl-ls1046a-rdb-sdk.dtb;
echo Encapsulating dtb image; blob enc \$fdt_addr_r \$load_addr \$filesize \$key_addr; setexpr
blobsize \$filesize + 0x30;echo Saving encrypted dtb; save \$devtype \$devnum:2 \$load_addr /fsl-
ls1046a-rdb-sdk.dtb \$blobsize; size \$devtype \$devnum:2 /ls1046ardb_dec_boot.scr;
load \$devtype \$devnum:2 \$load_addr /ls1046ardb_dec_boot.scr;
echo replacing Bootscript; save \$devtype \$devnum:2 \$load_addr /ls1046ardb_boot.scr \
$filesize;size \$devtype \$devnum:2 /secboot_hdrs/ls1046ardb/hdr_ls1046ardb_bs_dec.out;
```

```
load \$devtype \$devnum:2 \$load_addr /secboot_hdrs/ls1046ardb/hdr_ls1046ardb_bs_dec.out ;echo
Replacing bootscript header; save \$devtype \$devnum:2 \$load_addr /hdr_ls1046ardb_bs.out \
$filesize;reset;'
```

2. Decap boot script

```
size \$devtype \$devnum:2 /vmlinuz;setexpr imgsize \$filesize - 0x30 ;
echo Decapsulating linux image; setenv key_addr 0x87000000; mw \$key_addr $key_id_1;setexpr \
$key_addr \$key_addr + 0x4; mw \$key_addr $key_id_2;setexpr \$key_addr \$key_addr + 0x4; mw \
$key_addr key_id_3;setexpr \$key_addr \$key_addr + 0x4; mw \$key_addr $key_id_4;
 blob dec \$kernel_addr_r \$load_addr \$imgsize \$key_addr; cp.b \$load_addr \$kernel_addr_r \
$filesize ;size \$devtype \$devnum:2 /fsl-ls1046a-rdb-sdk.dtb;setexpr imgsize \$filesize - 0x30 ;
echo Decapsulating dtb image; blob dec \$fdt_addr_r \$load_addr \$imgsize \$key_addr; cp.b \
$load_addr \$fdt_addr_r \$filesize ;
```

3.3.1.7.2.2 Running secure boot (Chain of Trust with confidentiality)

1. Setup the board for secure boot flow. You can choose any of the flows mentioned below.
   a. **Flow A**

      Program the ITS fuse. Use RCW with SB_EN=0

      Or
   b. **Flow B**

      For protyping phase, do not blow the ITS fuse, instead use rcw with SB_EN = 1.
2. Blow other required fuses on the board. (OTPMK and SRK hash)[2]) 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.
3. Flash all the generated images at locations as described in the address map.
   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. **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.
      - **First Boot: Encapsulaton Step (Should happen in OEM's premises)**
        i. By defult the enacap and decap bootscripts will be installed in the bootpartition.

[2] 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.

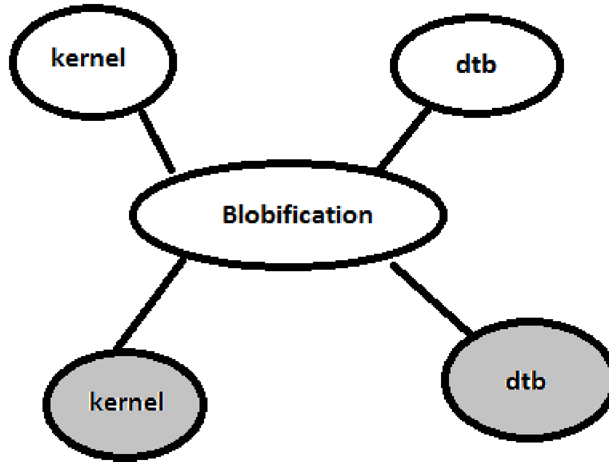
Write the SRK Hash value in SFP mirror registers and then release the core out of Boot Hold off by writing to Core Release Register in DCFG.

ii. When the board boots up for the first time after all images have been generated, Encap bootscript will execute. This bootscript:

   i. Authenticates and encapsulates linux and dtb images and replaces the unencrypted linux and dtb images with newly encapsulated linux and dtb.

   ii. Replaces the encap bootscript and header with the decap bootscript and it's header, already present in the bootpartition.

   iii. Issues reset

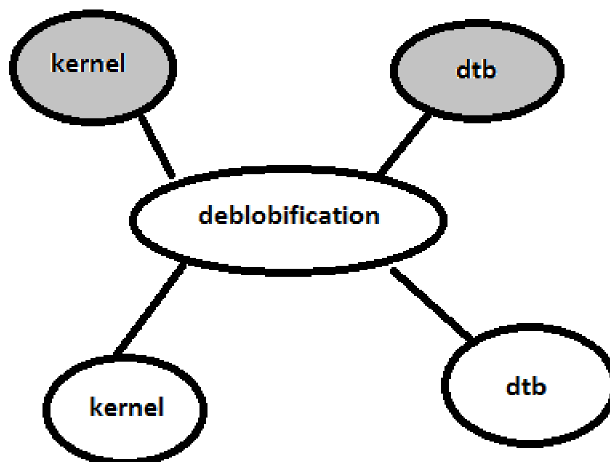


- **Subsequent Boot** .

i. Uboot would execute script with decap commands

   i. Un-blobify linux and dtb image in DDR

   ii. Pass control to these images



b. **Flow B** (*Secure boot images flashed on alternate bank*)

- On power on cycle, U-Boot prompt on bank0 would come up.
- On switching to alternate bank, the secure boot flow as mentioned above would execute.

3.3.1.8 Troubleshooting

Table 7. Troubleshooting

	Symptoms	**Reasons and/or Recommended actions**
1.	No print on UART console.	• Check the status register of sec mon block (location 0xfe314014). Refer to the details of the register from the Reference Manual. Bits OTPMK_ZERO, OTMPK_SYNDROME and PE should be 0 otherwise there is some error in the OTPMK fuse blown by you. • If OTMPK fuse is correct (see Step 1), check the SCRATCHRW2 register for errors. Refer to Section for error codes. • If **Error code = 0** then check the Security Monitor state in HPSR register of Sec Mon. **Sec Mon in Check State (0x9)** If ITS fuse = 1, then it means ISBC code has reset the board. This may be due to the following reasons: Hash of the public key used to sign the ESBC U-BOOT does not match with the value in SRK hash fuse Or Signature verification of the image failed **Sec Mon in Trusted State (0xd) or Non-Secure State (0xb)** Check the entry point field in the ESBC header. It should be 0xcffffffc for the demo described in Section 4. If entry point is correct, ensure that U-BOOT image has been compiled with the required secure boot configuration.
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. 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.

3.3.1.9 CSF Header Data Structure

The CSF Header provides the ISBC with most of the information needed to validate the image.

LS1 platform

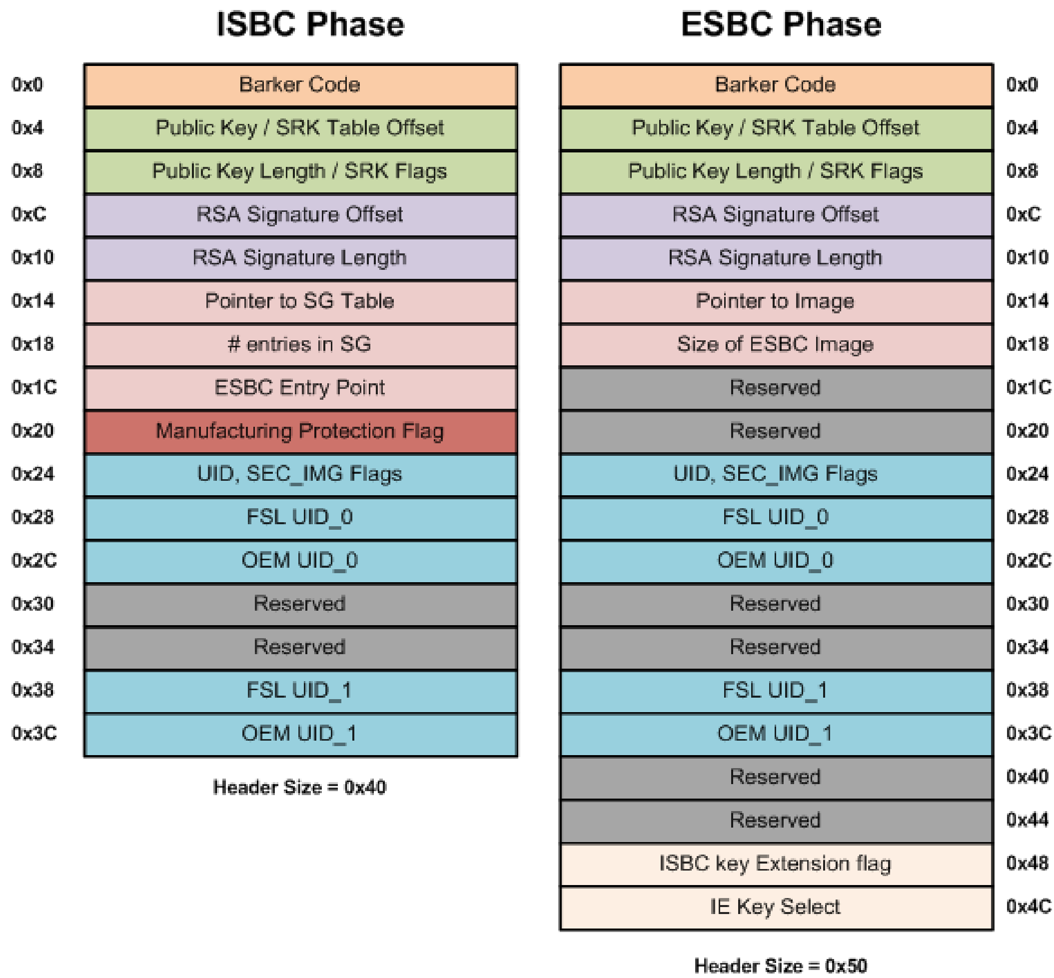


Figure 7. CSF header for LS1 (ISBC and ESBC phases)

Table 8. CSF Header Format (LS1 Platform)

Offset	**Data Bits [0:31]**
0x00-0x03	**Barker code**. 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 : • **Public key offset**: 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. If srk_table_flag is set: • **Srk table offset**: 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.

Table continues on the next page...

Table 8. CSF Header Format (LS1 Platform) (continued)

Offset	Data Bits [0:31]
0x08	**Srk table flag**. This flag indicates whether hash burnt in srk fuse is of a single key or of srk table.
0x0b-0x09	If the srk_table_flag is not set : • **0x0b-0x9** -- **Public key length**: This location contains the length of the public key in bytes. If srk_table_flag is set: • **0x09** – **Key Number from srk table** which is to be used for verification. • **0x0b-0x0a** – **Number of entries in srk table**. Minimum number of entries in table = 1, Maximum = 4.
0x0f-0x0c	**RSA Signature offset**. 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.
0x13-0x10	**RSA Signature length** in bytes.
0x17-0x14	**For ISBC Phase**: **SG Table offset** 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 is SG Table, the SG table is read. **For ESBC Phase**: Address of the image to be validated.
0x1b-0x18	**For ISBC Phase**: **Number of entries in SG Table** (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** **Size of image** to be validated
0x1f-0x1c	**For ISBC Phase**: **ESBC entry point**. ISBC transfers control to this location upon successful validation of ESBC image(s). **For ESBC Phase**: Reserved
0x21-0x20	**Manufacturing Protection Flag** Indicates if manufacturing protection has to be enabled or not in ISBC.
0x23-0x22	**Reserved** .(Earlier this field was SG Flag. SG flag is always assumed to be 1 in unified implementation.)

Table continues on the next page...

Table 8. CSF Header Format (LS1 Platform) (continued)

Offset	Data Bits [0:31]
0x24	For ISBC Phase: Reserved For ESBC Phase: Reserved
0x25	**For ISBC Phase** **Secondary Image flag** Indicates if user has a secondary image available in case of failures in validating primary iamge.Valid in case of primary Images's Header. **For ESBC Phase**:Reserved
0x27-0x26	**Unique ID Usage** This location contains a flag which specifies one of these possibilities • 0x00 - No UID's present • 0x01 - FSL UID and OEM UID are present • 0x02 - Only FSL UID is present • 0x04 - Only OEM UID is present
0x2b-0x28	**NXP unique ID 0** Upper 32 bits of a unique 64 bit value, which is specific to NXP. This value is compared with the FSL ID 1 in Secure Fuse Processor 's FSL-ID registers
0x2f-0x2c	**OEM unique ID 0** 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	**NXP unique ID 1** Lower 32 bits of a unique 64 bit value, which is specific to NXP. This value is compared with the FSL ID 1 in Secure Fuse Processor 's FSL-ID registers
0x3f-0x3c	**OEM unique ID 1** 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	**For ISBC Phase**: Not Applicable **For ESBC Phase**: **ISBC key Extension flag** If this flag is set, key to be used for validation needs to be picked up from IE Key table.

Table continues on the next page...

Table 8. CSF Header Format (LS1 Platform) (continued)

Offset	Data Bits [0:31]
0x4c-0x4f	**For ISBC Phase**: Not Applicable **For ESBC Phase**: **IE Key Select** Key Number to be used from the IE Key Table if IE flag is set.

Table 9. 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	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 10. 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 11. 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 12. SRK Table (LS1 Platform)

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)

LS1046A platform

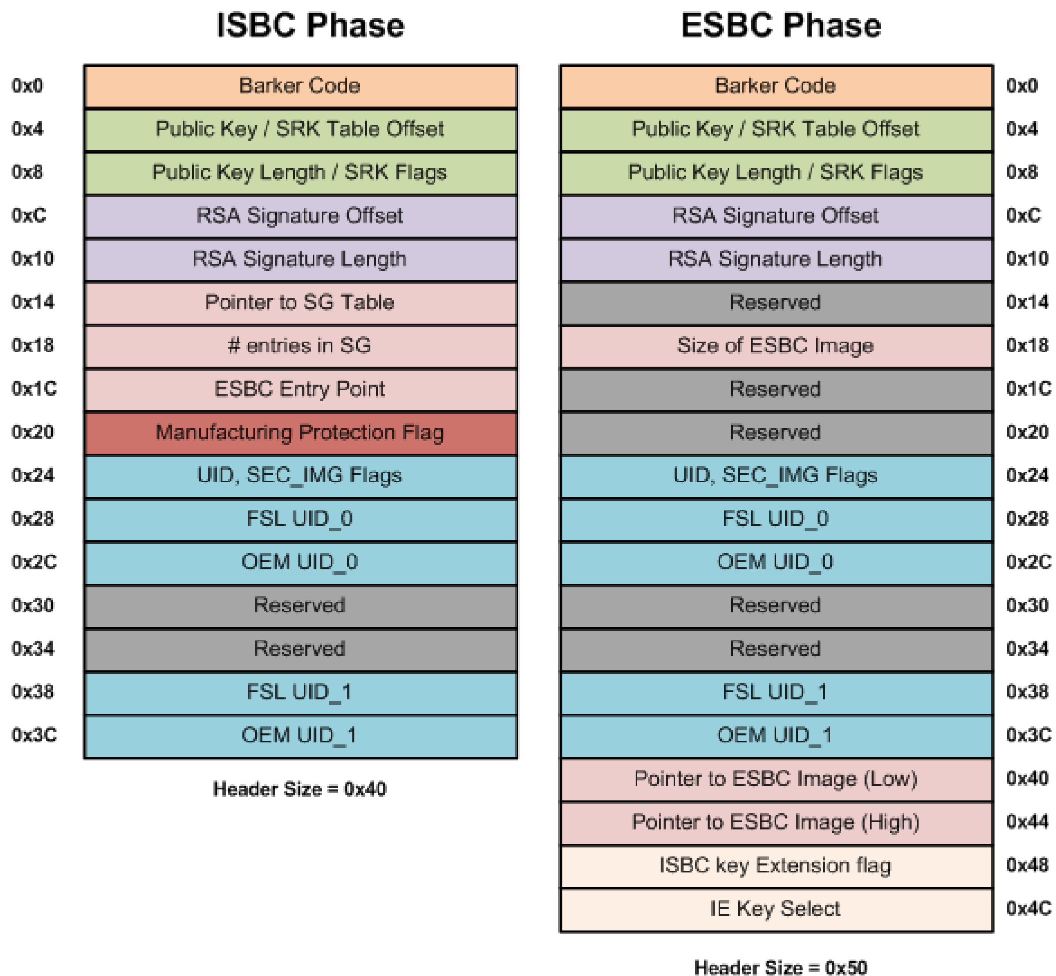


Figure 8. CSF header for LS1046A (ISBC and ESBC phases)

Table 13. CSF header format (LS1046A platform)

Offset	**Data Bits [0:31]**
0x00-0x03	**Barker code**. 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 : • **Public key offset**: 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. If srk_table_flag is set: • **Srk table offset**: 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.
0x08	**Srk table flag**. This flag indicates whether hash burnt in srk fuse is of a single key or of srk table.
0x0b-0x09	If the srk_table_flag is not set : • **0x0b-0x9** -- **Public key length**: This location contains the length of the public key in bytes. If srk_table_flag is set: • **0x09** – **Key Number from srk table** which is to be used for verification. • **0x0b-0x0a** – **Number of entries in srk table**. Minimum number of entries in table = 1, Maximum = 4.
0x0f-0x0c	**RSA Signature offset**. 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.
0x13-0x10	**RSA Signature length** in bytes.
0x17-0x14	**For ISBC Phase**: **SG Table offset** 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 is SG Table, the SG table is read. **For ESBC Phase**: Reserved
0x1b-0x18	**For ISBC Phase**: **Number of entries in SG Table** (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** **Size of image** to be validated

Table continues on the next page...

Table 13. CSF header format (LS1046A platform) (continued)

Offset	Data Bits [0:31]
0x1f-0x1c	**For ISBC Phase**: **ESBC entry point**. ISBC transfers control to this location upon successful validation of ESBC image(s). **For ESBC Phase**: Reserved
0x21-0x20	**Manufacturing Protection Flag** Indicates if manufacturing protection has to be enabled or not in ISBC.
0x23-0x22	**Reserved** .(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
0x25	**For ISBC Phase** **Secondary Image flag** Indicates if user has a secondary image available in case of failures in validating primary iamge.Valid in case of primary Images's Header. **For ESBC Phase**:Reserved
0x27-0x26	**Unique ID Usage** This location contains a flag which specifies one of these possibilities • 0x00 - No UID's present • 0x01 - FSL UID and OEM UID are present • 0x02 - Only FSL UID is present • 0x04 - Only OEM UID is present
0x2b-0x28	**NXP unique ID 0** Upper 32 bits of a unique 64 bit value, which is specific to NXP. This value is compared with the FSL ID 1 in Secure Fuse Processor 's FSL-ID registers
0x2f-0x2c	**OEM unique ID 0** 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	**NXP unique ID 1** Lower 32 bits of a unique 64 bit value, which is specific to NXP. This value is compared with the FSL ID 1 in Secure Fuse Processor 's FSL-ID registers

Table continues on the next page...

Table 13. CSF header format (LS1046A platform) (continued)

Offset	Data Bits [0:31]
0x3f-0x3c	**OEM unique ID 1** 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**: 64 bit pointer to ESBC image
0x48-0x4b	**For ISBC Phase**: Not Applicable **For ESBC Phase**: **ISBC key Extension flag** 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**: **IE Key Select** Key Number to be used from the IE Key Table if IE flag is set.

Table 14. Scatter gather table format (LS1046A 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	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 15. Signature (LS1046A platform)

Offset	Data Bits [0:31]
0x00-size	The RSA signature calculated over CSF Header, Scatter Gather table and ESBC image(s).

Table 16. Public key (LS1046A 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 17. SRK table (LS1046A platform)

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)

3.3.1.10 ISBC validation error codes

LS1/LS1046A platform

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 18. 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 19. Core exceptions (LS1046A platform)

Error Code	Value
Current EL with SP0	
ERROR_EXCEPTION_SYNC_SP0	0x01
ERROR_EXCEPTION_IRQ_SP0	0x02
ERROR_EXCEPTION_FIQ_SP0	0x03
ERROR_EXCEPTION_SERROR_SP0	0x04
Current EL with SPx	
ERROR_EXCEPTION_SYNC_SPX	0x05
ERROR_EXCEPTION_IRQ_SPX	0x06
ERROR_EXCEPTION_FIQ_SPX	0x07
ERROR_EXCEPTION_SERROR_SPX	0x08
Lower EL using AArch64	
ERROR_EXCEPTION_SYNC_L64	0x11
ERROR_EXCEPTION_IRQ_L64	0x12
ERROR_EXCEPTION_FIQ_L64	0x13
ERROR_EXCEPTION_SERROR_L64	0x14
Lower EL using AArch32	
ERROR_EXCEPTION_SYNC_L32	0x15
ERROR_EXCEPTION_IRQ_L32	0x16
ERROR_EXCEPTION_FIQ_L32	0x17
ERROR_EXCEPTION_SERROR_L32	0x18

Table 20. System state failures (LS1/LS1046A 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 21. General header checking failures (LS1/LS1046A 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_SUPPORT ED	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 22. Key/signature/UID related errors (LS1/LS1046A 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

Table continues on the next page...

Table 22. Key/signature/UID related errors (LS1/LS1046A platforms) (continued)

Value	Code	Definition
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_MOD	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.
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 23. Verification failures (LS1/LS1046A 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 24. SEC/PAMU failures (LS1/LS1046A platforms)

Value	Code	Definition
0x700	ERROR_SEC_ENQ	Error when enqueuing 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

Table continues on the next page...

Table 24. SEC/PAMU failures (LS1/LS1046A platforms) (continued)

Value	Code	Definition
0x800	ERROR_PAMU	Error while programming PAACT/SPAACT tables in PAMU (For PowerPC platforms only)

3.3.1.11 ESBC Validation Error Codes

For trust arch version 1.x and 2.x.

Table 25. ESBC Validation Failures

Value	Code	Definition
0x0	ERROR_ESBC_CLIENT_MAX	NULL
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
0x11	ERROR_ESBC_CLIENT_HEADER_KEY_REVOKED	Key used to sign the image revoked
0x12	ERROR_ESBC_CLIENT_HEADER_INVALID_SRK_NUM_ENTRY	Wrong key entry
0x13	ERROR_ESBC_CLIENT_HEADER_INVALID_KEY_NUM	Selected key no. not in SRK table
0x14	ERROR_ESBC_CLIENT_HEADER_INV_SRK_ENTRY_KEYLEN	Unsupported key length of key in SRK table
0x15	ERROR_ESBC_CLIENT_HEADER_IE_KEY_REVOKED	Selected key in IE key table revoked
0x16	ERROR_ESBC_CLIENT_HEADER_INVALID_IE_NUM_ENTRY	Wrong IE Key entry
0x17	ERROR_ESBC_CLIENT_HEADER_INVALID_IE_KEY_NUM	Selected key no. not in IE Key table
0x18	ERROR_ESBC_CLIENT_HEADER_INV_IE_ENTRY_KEYLEN	Unsupported key length of key in IE Key table
0x19	ERROR_IE_TABLE_NOT_FOUND	information about IE table missing
0x20	ERROR_ESBC_CLIENT_HEADER_KEY_LEN_NOT_TWICE_SIG_LEN	Public key length not twice of signature length
0x21	ERROR_KEY_TABLE_NOT_FOUND	SRK Key/key table not found

Table continues on the next page...

Table 25. ESBC Validation Failures (continued)

Value	Code	Definition
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
0x200	ERROR_ESBC_CLIENT_HEADER_SG_ESBC_EP	Entry Point error
0x400	ERROR_ESBC_CLIENT_HASH_COMPARE_KEY	Public key hash comparison failed
0x800	ERROR_ESBC_CLIENT_HASH_COMPARE_EM	RSA verification failed
0x1000	ERROR_ESBC_CLIENT_SSM_TRUSTSTS	SNVS not in TRUSTED state
0x2000	ERROR_ESBC_CLIENT_BAD_ADDRESS	Bad address error
0x4000	ERROR_ESBC_CLIENT_MISC	Miscallaneous error
0x8000	ERROR_ESBC_CLIENT_HEADER_SG_ENTIRES_BAD	Incorrect entries in SG table
0x10000	ERROR_ESBC_CLIENT_HEADER_SG	No SG support
0x20000	ERROR_ESBC_CLIENT_HEADER_IMG_SIZE	Invalid Image size
0x40000	ERROR_ESBC_WRONG_CMD	Failure in command/Unknown command/Wrong arguments of boot script.
0x80000	ERROR_ESBC_MISSING_BOOTM	Bootm command missing from boot script.

3.3.1.12 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	
LS1046A	2.1	3.3	1.89 V	A	SFP	2	SFP	

3.3.2 Code Signing Tool

To assist with signing of various images and creation of CSF header, NXP offers a Code Signing Tool (CST). It is generally expected that the CST signs images in an offline process

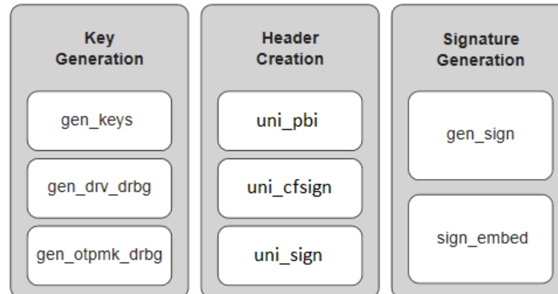


Figure 9. Tool in CST Package

3.3.2.1 Key generation

The CST begins by generating 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.

The application allows the user to feed 3 key sizes for generating keys. The key sizes are 1024 bits, 2048 bits, and 4096 bits.

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.

3.3.2.1.1 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 key sizes 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 file names through command line option.

Usage

./gen_keys [OPTION] SIZE

SIZE refers to size of public key in bits. (Modulus size).

Size 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.pri
```

```
$ ./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
```

3.3.2.1.2 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 -b <bit_order> --s [string]

<bit_order> : (1 or 2) OTPMK Bit Ordering Scheme in SFP

1 : TA1.x

2 : TA2.x, TA3.x

<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 -b 1

        #----------------------------------------------------#
        #-------         --------     --------        -------#
        #------- CST (Code Signing Tool) Version 2.0  -------#
        #-------         --------     --------        -------#
        #----------------------------------------------------#

Input string not provided
Generating a random string
-------------------------------------------
* Hash_DRBG library invoked
* Seed being taken from /dev/urandom
-------------------------------------------
OTPMK[255:0] is:
d2f63a662f69a1faa4c2406f83eedde7647fbd3c62ac442c67fad2d4cda8b3a0

 NAME    |     BITS     |    VALUE
_________|______________|____________
OTPMKR 0 |  31-  0      |   cda8b3a0
OTPMKR 1 |  63- 32      |   67fad2d4
OTPMKR 2 |  95- 64      |   62ac442c
OTPMKR 3 | 127- 96      |   647fbd3c
OTPMKR 4 | 159-128      |   83eedde7
OTPMKR 5 | 191-160      |   a4c2406f
```

```
OTPMKR 6 | 223-192      |   2f69a1fa
OTPMKR 7 | 255-224      |   d2f63a66
```

```
$ ./gen_otpmk_drbg -b 2 --s 1111111122222222333333334444444455555555666666667777777788888888


        #----------------------------------------------------#
        #-------         --------     --------        -------#
        #------- CST (Code Signing Tool) Version 2.0  -------#
        #-------         --------     --------        -------#
        #----------------------------------------------------#

OTPMK[255:0] is:
1111111122222222333333334444444455555555666666667777777788888888

 NAME     |      BITS      |     VALUE
_________|______________|____________
OTPMKR 0 | 255-224      |   11111111
OTPMKR 1 | 223-192      |   22222222
OTPMKR 2 | 191-160      |   33333333
OTPMKR 3 | 159-128      |   44444444
OTPMKR 4 | 127- 96      |   55555555
OTPMKR 5 |  95- 64      |   66666666
OTPMKR 6 |  63- 32      |   77777777
OTPMKR 7 |  31-  0      |   88888888
```

3.3.2.1.3 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 Genearted 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
```

3.3.2.2 Header creation

3.3.2.2.1 uni_pbi

Following options are available with the uni_pbi command.

```
$ ./uni_pbi
        --verbose    Display header Info after Creation. This option is invalid for TA2 platform
        --hash       Print the SRK(Public key) hash. This option is invalid for TA2 platform
        --img_hash   Header is generated without Signature.
                     Image Hash is stored in a separate file. This option is invalid for TA2 platform
        --help       Show the Help for Tool Usage.
```

The input to this tool will be an input file specifying the platform. Based on that, there are two separate behaviour of the tool.

uni_pbi for TA2.x platforms is used for the following:

- To add boot location pointer and set SB_EN and BOOT_HO value for secure boot
- (optional) To add PBI commands (ACS write commands to add U-Boot spl and its header to OCRAM from Non-XIP memory).
- (optional) To append images (U-Boot, Boot script, and their headers) to RCW file.

Refer Hardware Pre-Boot Loader (PBL) based platforms on page 35 for TA2.x based platforms.

uni_pbi for Service processor based platforms

- uni_pbi tool is used for creating signature and header over PBI commands.

Table 26. Description of fields in input files of both type of platforms (TA2.x and TA3.x)

Field name	Description	Platform supported
PLATFORM	The platform for which tool is being used	TA 2.x and TA 3.x
RCW_PBI_FILENAME	Input image file name. The rcw file which has to be modified.	TA 2.x and TA 3.x
BOOT1_PTR	Address of ISBC (Boot1) CSF Header	TA 2.x and TA 3.x
OUTPUT_RCW_PBI_FILENAME	To identify the platform for which the tool is being used. This field is optional. If not specified, it will take default name.	TA 2.x
BOOT_SRC	Only to be specified in case of **SD boot**	TA 2.x
SB_EN	Field to enable or disable secure boot, by setting SB_EN bit in rcw file to 1	TA 2.x
BOOT_HO	To put core in hold-off state to fuse key hash in case of secure boot, by setting BOOT_HO bit in rcw file to 1	TA 2.x
COPY_CMD	To add ACS write commands to write U-Boot spl and is header to OCRAM. This is an optional field. If not mentioned, won't add the command.	TA 2.x
APPEND_IMAGES	To append U-Boot, Boot script, and their headers to the new rcw generated. It is an optional field. This is an optional field, if not specified, no images will be appended.	TA 2.x
KEY_SELECT	Specify the key to be used in signature generation from the SRK table	TA 3.x
PRI_KEY	Private key file name in PEM format. The maximum keys supported are 8.	TA 3.x
FSL_UID_x	FSL UID(s) to be populated in the header	TA 3.x
OEM_UID_x	OEM UID(s) to be populated in the header	TA 3.x
OUTPUT_HDR_FILENAME	Output file name of the header. An output file name is generated with rcw commands appended with signed PBI commands.	TA 3.x
IMAGE_HASH_FILENAME	used with '--img_hash' option (Name of file in which Image Hash is stored)	TA 3.x
MP_FLAG	Manufacturing Protection Flag	TA 3.x
ISS_FLAG	Increment Security State Flag	TA 3.x
LW_FLAG	Leave Writeable Flag	TA 3.x
VERBOSE	Specify VERBOSE as 1, if you want to display header information. This can also be done with '--verbose' option	TA 3.x
IE_TABLE_ADDR	64-bit address of IE table(to be used in case of IE key extension feature usage)	TA 3.x

Sample input files are present in the CST tool at location: input_files/uni_pbi/<platform>/

For example, input_files/uni_pbi/ls1/input_pbi_sd_secure.

NOTE

In TA 3.x, SB_EN and BOOT_HO fields are by default set to 1 to enable secure boot.

NOTE

To know platforms under TA 2.x, see Trust Architecture and SFP information on page 65.

3.3.2.2.1.1 Sample Input File

Sample input file for TA2-based platforms:

```
/*
 * Copyright 2016 NXP
 */
-----------------------------------------------------------------------------
# For PBI Creation
# Name of RCW + PBI file [Mandatory]
RCW_PBI_FILENAME= u-boot-with-spl-pbl.bin
-----------------------------------------------------------------------------
# Specify the output file name [Optional].
# Default Values chosen in Tool
OUTPUT_RCW_PBI_FILENAME=u-boot-with-spl-pbl-sec.bin
-----------------------------------------------------------------------------
#specify the boot src
BOOT_SRC=SD_BOOT
# Specify the platform
PLATFORM=LS1020
# Specify the RCW Fields. (0 or 1) - [Optional]
SB_EN=1
BOOT_HO=1
BOOT1_PTR=10016000
-----------------------------------------------------------------------------
# Specify the PBI commands - [Optional]
# Argument: COPY_CMD = (src_offset, dest_offset, Image name)
# Split hdr_uboot_spl.out in PBI commads
COPY_CMD={ffffffff,10016000,hdr_uboot_spl.out;}
-----------------------------------------------------------------------------
# Specify the Images to be appended
# Arguments: APPEND_IMAGES=(Image name, Offset from start)
APPEND_IMAGES={u-boot-dtb.bin,00022000;}
APPEND_IMAGES={hdr_uboot.out,00122000;}
APPEND_IMAGES={hdr_bs.out, 00124000;}
APPEND_IMAGES={bootscript,00128000;}
-----------------------------------------------------------------------------
```

3.3.2.2.2 uni_sign

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

  uni_sign tool (with ESBC = 0 in input file) is used for creating signature and header over Boot1 image to be verified by ISBC

  uni_sign tool (with ESBC = 1 in input file) is used for creating signature and header over images to be verified by ESBC

Following options are available with the uni_sign command.

Usage:

To view usage of tool:

```
./uni_sign
    --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
    --out <file>  Header file name
    --in <file>   Input file for signature calculation. This option would override the filename in
IMAGE_1 in input_file if present
    --app <file>  File to be appended to the header
    --app_off <offset>    Offset at which file will be appended to the header
    --help       Show the Help for Tool Usage
```

For example:

```
./uni_sign --in <inp_file>  --out <op file> --app_off <offset> --app <file> <input_file>
```

NOTE

There are scenarios when a build script using the tool needs to modify the input file name or the output header file name. These command line options provide a way to override the values as specified in the input file.

Table 27. Description of fields

Field	Field description	Platform supported
PLATFORM	To identify the platform/SoC for which CF header needs to be created.	All
ESBC	Do not 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.	TA3.x
ENTRY_POINT	Entry point address or Image start address field in the header.	All
PRI_KEY	Private key file name 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.	All
PUB_KEY	Public key file name in PEM format. (default = srk.pub generated by gen_keys) FILE1 [,FILE2, FILE3, FILE4]. Multiple key support for Trust Arch v2.x devices only.	All
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)	All
IMAGE_1 - IMAGE_8	Create Entries for SG table in the format { IMAGE_NAME, SRC_ADDR, DST_ADDR }	All
OEM_UID_x	OEM UID to be populated in the header.	All
FSL_UID_x	FSL UID to be populated in the header.	All
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)	TA2.x
HKAREA_SIZE	House Keeping Area Size (Required for Trust Arch v2.x devices only when esbc option is not provided)	TA2.x

Table continues on the next page...

Table 27. Description of fields (continued)

Field	Field description	Platform supported
OUTPUT_HDR_FILENAME	Name of the combined header binary to be created by tool	All
SG_TABLE_ADDR	Specify SG_TABLE Address where Scatter Gather table is present for 2041/3041/4080/5020/5040 when ESBC=0.	TA1.x
OUTPUT_SG_BIN	Specify the output file name of sg table.	TA1.x
IMAGE_TARGET	Specify the target where image will be loaded. For example,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	All
SEC_IMG	Flag for Secondary Image. Required for Trust Arch v2.x devices only	TA2.x
MP_FLAG	Specify Manufacturing Protection Flag. Available for LS1 only.	All, only needed in ISBC phase
VERBOSE	Specify Verbose option. Contents of header generated will be printed.	All
IMAGE_HASH_FILENAME	used with '--img_hash' option (Name of file in which Image Hash is stored)	TA3.x
ISS_FLAG	Increment Security State Flag	TA3.x, only needed in ISBC phase
LW_FLAG	Leave Writeable Flag	TA3.x, only needed in ISBC phase
ESBC_HDRADDR	32-bit address where header generated will be placed. Used to calculate IE key table address	TA3.x, only to be used in case of IE key extension feature usage
IE_KEY	Comma separated list of files containing public keys(IE Keys)	TA3.x, only to be used in case of IE key extension feature usage
IE_REVOC	Comma separated list of numbers that are to be revoked from IE table	TA3.x, only to be used in case of IE key extension feature usage
IE_KEY_SEL	No. of keys in IE table that is to be used to validate image	TA3.x, only to be used in case of IE key extension feature usage

Sample input files can be referred to, from input_files/uni_sign/l<platform>.

For IE keys, refer to input_files/uni_sign/l<platform>/ie_ke.

To know platforms under TA 2.x, refer Trust Architecture and SFP information on page 65.

3.3.2.2.2.1 Sample Input File

The input files will not have ESBC field (ESBC=0).

```
---------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform -
# TRUST 3.1: LS2088, LS1088
# TRUST 1.0, 1.1, 2.0, 2.1: 1010/1040/2041/3041/4080/5020/5040/9131/9132/9164/4240/C290/LS1
PLATFORM=LS2088
---------------------------------------------------
# Entry Point/Image start address field in the header.[Mandatory]
# (default=ADDRESS of first file specified in images)
# Address can be 64 bit
ENTRY_POINT=30008000
---------------------------------------------------
# Specify the Key Information.
# PUB_KEY [Mandatory] Comma Seperated List
# Usage: <srk1.pub> <srk2.pub> .....
PUB_KEY=srk.pub
# KEY_SELECT [Mandatory]
# USAGE (for TRUST 3.1): (between 1 to 8)
KEY_SELECT=1
# PRI_KEY [Mandatory] Comma Seperated List for Signing
# USAGE: <srk.pri>, <srk2.pri>
PRI_KEY=srk.pri
---------------------------------------------------
# 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}
# Address can be 64 bit
IMAGE_1={u-boot.bin,30008000,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_0=11111111
FSL_UID_0=
FSL_UID_1=
OEM_UID_0=
OEM_UID_1=
OEM_UID_2=
OEM_UID_3=
OEM_UID_4=
---------------------------------------------------
# Specify the output file names [Optional].
# Default Values chosen in Tool
OUTPUT_HDR_FILENAME=hdr_uboot.out
IMAGE_HASH_FILENAME=
RSA_SIGN_FILENAME=

-------------------------------------------------
# Specify The Flags. (0 or 1) - [Optional]
MP_FLAG=0
```

```
ISS_FLAG=1
LW_FLAG=0

---------------------------------------------------
# Specify VERBOSE as 1, if you want to Display Header Information [Optional]
VERBOSE=0
---------------------------------------------------
# Following fields are Required for 4240/9164/1040/C290 only

# Specify House keeping Area
# Required for 42409164/1040/C290 only when ESBC flag is not set. [Mandatory]
HK_AREA_POINTER=
HK_AREA_SIZE=
---------------------------------------------------
# Following field Required for 4240/9164/1040/C290 only
# Specify Secondary Image Flag. (0 or 1) - [Optional]
# (Default is 0)
SEC_IMAGE=
---------------------------------------------------
# Specify SG table address, only for (2041/3041/4080/5020/5040) with ESBC=0 - [Optional]
SG_TABLE_ADDR=
```

3.3.2.3 Signature generation

The tools in this category are provided in case the user does not want to share the Private Key with the CST tool. The **--img_hash** option in Header creation on page 70 tools provides OEMs with the ability to perform code signing in a secure environment which does not run the NXP Code Signing Tool.

--img_hash option

- Generates hash file in binary format which contains SHA256 hash of the components required for signature.
- Generates output header binary file based on the fields specified in input file.
- Output header binary file does not contain signature.
- Provides flexibility to manually append signature at the end of output header file. Users 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.
- The SHA256 hash generated over CF header (in case of TA1.x platforms)) is then signed using RSA algorithm (OPENSSL APIs) with the private key. This encrypted hash is known as digital signature. This signature is placed at an offset from the CF header, which is later read by IBR.
- The SHA256 hash generated over CSF header, the public Key, the S/G table and the ESBC is also signed using RSA algorithm with the same private key. The signature generated is placed at an offset from the CSF header, which is again later read by IBR.

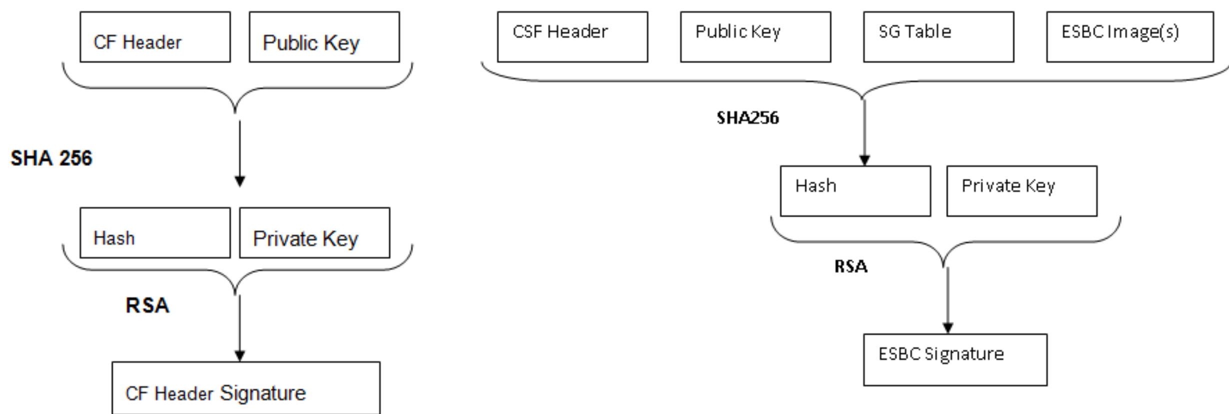


Figure 10. Dual signature generation

Usage example

```
$ ./uni_sign --img_hash --verbose input_files/uni_sign/<platform>/input_uboot_nor_secure


    #----------------------------------------------------#
    #-------         --------     --------        -------#
    #------- CST (Code Signing Tool) Version 2.0  -------#
    #-------         --------     --------        -------#
    #----------------------------------------------------#

==========================================================
This tool includes software developed by OpenSSL Project
for use in the OpenSSL Toolkit (http://www.openssl.org/)
This product includes cryptographic software written by
Eric Young (eay@cryptsoft.com)
==========================================================

Input File is input_files/uni_sign/<platform>/input_uboot_nor_secure

-----------------------------------------------
-    Dumping the Header Fields
-----------------------------------------------
- SRK Information
-     SRK Offset : 200
-     Number of Keys : 1
-     Key Select : 1
-     Key List :
-        Key1 srk.pub(100)
- UID Information
-     UID Flags = 00
-     FSL UID = 00000000_00000000
-     OEM UID0 = 00000000
-     OEM UID1 = 00000000
-     OEM UID2 = 00000000
-     OEM UID3 = 00000000
-     OEM UID4 = 00000000
- FLAGS Information
-     MISC Flags = 60
-         ISS = 1
-         MP = 0
```

```
-         LW = 0
-         B01 = 1
- Image Information
-     SG Table Offset : 800
-     Number of entries : 1
-     Entry Point : 30008000
-     Entry 1 : u-boot.bin (Size = 000c0000 SRC = 30008000 DST = ffffffff)
- RSA Signature Information
-     RSA Offset : a00
-     RSA Size : 80
-----------------------------------------------

Image Hash:
8588c174dd92f4a1b114b9029fc647e18cac4aaa46f03a6538ef20531e796e8f

************************************************
* Image Hash Stored in File: hash.out
* Header File is w/o Signature appended
************************************************

Header File Created: hdr_uboot.out

SRK (Public Key) Hash:
7df50d4256c4cbde4ef4ae9931042b1e44ff13aeb5107a7e0e9ee07e0fbfc236
     SFP SRKHR0 = 7df50d42
     SFP SRKHR1 = 56c4cbde
     SFP SRKHR2 = 4ef4ae99
     SFP SRKHR3 = 31042b1e
     SFP SRKHR4 = 44ff13ae
     SFP SRKHR5 = b5107a7e
     SFP SRKHR6 = 0e9ee07e
     SFP SRKHR7 = 0fbfc236
```

The tools are provided to create the signature file and embed the signature at the end of header file.

3.3.2.3.1 gen_sign

This tool is provided for the user to calculate signature for a given hash using CST tool. The tool requires only the hash file and private key file from the user as input. It would generate signature file as output.

It uses RSA_sign API of openssl to calculate signature over hash provided.

Usage

./gen_sign [option] <HASH_FILE> <PRIV_KEY_FILE>

--sign_file SIGN_FILE	Provides file name for signature to be generated as operand. SIGN_FILE is generated containing signature calculated over hash provided through HASH_FILE using private key provided through PRIV_KEY_FILE. With this option, HASH_FILE and PRIV_KEY_FILE are compulsory while SIGN_FILE is optional. The default value of SIGN_FILE is signout.
HASH_FILE	Name of hash file containing hash over signature needs to be calculated.
PRIV_KEY_FILE	Name of key file containing private key.

Usage example

After the hash file has been created as described in Signature generation on page 76, the tool can be used as described below.

```
$ ./uni_sign --img_hash input_files/uni_sign/<platform>/input_uboot_nor_secure
```

```
.
.
.

***********************************************
* Image Hash Stored in File: hash.out
* Header File is w/o Signature appended
***********************************************

Header File Created: hdr_uboot.out

$ ./gen_sign hash.out srk.pri

    #--------------------------------------------------#
    #-------         --------     --------        -------#
    #------- CST (Code Signing Tool) Version 2.0  -------#
    #-------         --------     --------        -------#
    #--------------------------------------------------#
Signature Length = 80
Hash in hash.out is signed with srk.pri
Signature is stored in file : sign.out
```

3.3.2.3.2 sign_embed

This tool embeds signature in the header file generated using img_hash option which generates header but does not embed signature in the header. This option opens header file and copies signature at the end of the file.

The header file generated with 'img_hash' option has padding added till signature offset, so that signature can be directly embedded to the end of the file.

Usage

./sign_embed <hdr_file> <sign_file>

hdr_file	Name of header file in which signature needs to be embedded
sign_file	Name of sign file containing signature which needs to be embedded

Usage example

```
$ ./sign_embed hdr_uboot.out sign.out

    #--------------------------------------------------#
    #-------         --------     --------        -------#
    #------- CST (Code Signing Tool) Version 2.0  -------#
    #-------         --------     --------        -------#
    #--------------------------------------------------#

hdr_uboot.out is appended with file sign.out (0x80)
```

NOTE

User can generate the complete header along with signature in a single step using uni_sign/uni_pbi tool without any option.

./uni_sign <input_file>

Or

User may wish to do it in three separate steps:

1. ./uni_sign **--img_hash** <input_file> (Create header file without signature and store the hash in a separate file)
2. ./gen_sign[3] [option] <HASH_FILE> <PRIV_KEY_FILE> (Sign the image hash using private key)
3. ./sign_embed <hdr_file> <sign_file> (Embed the signature at the end of header file)

3.3.3 Procedure to Run Secure Boot

This section describes the steps to be followed to run secure boot on a platform, after building the images.

3.3.3.1 Prepare board for secure boot

Blowing One Time Programmable Master Key (OTPMK) fuse

1. Check initial SNVS state:

```
md 1e90014
88000900
```

   The second nibble indicates that the OTPMK is not blown.

2. Enable POVDD for FRWY-LS1046A board:
   a. Boot the board to U-Boot prompt in non-secure mode.
   b. Enable POVDD using GPIO3[24] by using below commands:

```
mw 0x2320000 80000000
mw 0x2320008 f01c0000
```

3. Use the following command to generate OTPMK:

```
cd cst
./gen_otpmk_drbg 2
```

   NOTE

   For more information on gen_otpmk_drbg, see Code Signing Tool on page 65.

4. Write OTPMK fuse values on shadow registers:

```
mw.l 1e80234 <OTPMK1>
mw.l 1e80238 <OTPMK2>
mw.l 1e8023c <OTPMK3>
mw.l 1e80240 <OTPMK4>
mw.l 1e80244 <OTPMK5>
mw.l 1e80248 <OTPMK6>
mw.l 1e8024c <OTPMK7>
mw.l 1e80250 <OTPMK8>
```

[3] This may be done by user's own tool in case he does not want to share the private key with the CST tool.

5. Check SNVS state again. There should be no parity errors.

```
md 1e90014
80 000 900
```

Now, you will see ‘0’ in second nibble.

```
md 1e80024
00000000
```

No parity errors.

6. Use the below command (for LS1046A) to write to INGR register:

```
mw 1e80020 0x02000000
```

7. Reset and check that SNVS is in Check state:

```
md 1e90014
80 000 900
```

3.3.3.2 Running secure boot on target platforms

Platform LS1046A

1. After copying images to flash, select the boot source by changing the switch settings, then boot the board.
2. Flexbuild-generated RCW for secure boot has the boot core put in hold off by setting BOOT_HO = 1 and enabled secure boot by SB_EN=1.

   After booting the board, core would get stuck at its first instruction. This is done to allow the user to write SRKH in the register. When using pre-built images, use the SRK hash present in srk_hash.txt from github. If SRKH fuse is already blown, then set BOOT_HO = 0 in rcw file in flexbuild, else write the SRK hash value (displayed while signing images) in SFP mirror registers and then release the core out of Boot Hold off by writing to Boot Release Register in DCFG using the below commands:

```
ccs::config_server 0 10000
ccs::config_chain {<platform> dap sap2}
display ccs::get_config_chain
#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem <dap position> 0x1e90014 4 0 4
ccs::display_mem <dap position> 0x1ee0200 4 0 4
#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem <dap position> 0x1e80254 4 0 <SRKH1>
ccs::write_mem <dap position> 0x1e80258 4 0 <SRKH2>
ccs::write_mem <dap position> 0x1e8025c 4 0 <SRKH3>
ccs::write_mem <dap position> 0x1e80260 4 0 <SRKH4>
ccs::write_mem <dap position> 0x1e80264 4 0 <SRKH5>
ccs::write_mem <dap position> 0x1e80268 4 0 <SRKH6>
ccs::write_mem <dap position> 0x1e8026c 4 0 <SRKH7>
ccs::write_mem <dap position> 0x1e80270 4 0 <SRKH8>
#Get the Core Out of Boot Hold-Off
ccs::write_mem <dap position> 0x1ee00e4 4 0 0x00000001
```

NOTE

Platform to be used in the above commands is ls1043a for LS1046A.

After implementing all the steps, the board will boot and user will get the Linux prompt after **successful validation** of all the images.

NOTE

To blow SRKH in production environment, follow procedure similar to blowing OTPMK fuses. For more detail on production and development environments, see Flow A and Flow B under "Product execution" section.

3.3.3.3 Steps to run Chain of Trust with confidentiality

1. Generate all images:

```
$ flex-builder -c firmware
$ flex-builder -c linux -a <arch>
```

2. Generate autoboot script with `e` flag.

   a. With encapsulation flag enabled:

   ```
   $ flex-builder -i mkdistroscr -e
   ```

   (or)

   b. With encapsulation and key identifier (16 bytes):

   ```
   $ flex-builder -i mkdistroscr -e -k <key_id>
   ```

   For example, Key_id = 0x20000000.

   NOTE

   For more information on key identifier, see "Other images required for the demo" section.

3. Sign all images:

```
$ flex-builder -i signimg -m <platform> -b <boottype> -s -e
```

4. Generate firmware image:

```
$  flex-builder -i mkfw -m <platform> -b <boottype > -s
```

5. Generate boot partition:

```
$ flex-builder -i mkbootpartition -a <arch> -s
```

6. Write image to micro-SD card:

```
$ flex-installer -b build/images/bootpartition_LS_<arch>_lts_<version>.tgz -r build/rfs/
rootfs_lsdk_<version>_LS_<arch> -d /dev/sdx
```

BOOT FLOW

First Boot: Encapsulaton Step (Should happen in OEM's premises)

1. By defult, the enacap and decap boot scripts will be installed in the boot partition.
2. When the board boots up for the first time after all images have been generated, encap boot script will execute. This boot script:

   a. Authenticates and encapsulates Linux and DTB images and replaces the unencrypted Linux and DTB images with newly encapsulated Linux and DTB.

   b. Replaces the encap boot script and header with the decap boot script and its header, already present in the boot partition.

   c. Issues reset.

Subsequent Boot

1. U-Boot will execute script with decap commands.
    a. Un-blobify Linux and DTB image in DDR.
    b. Pass control to these images.

3.4 FRWY-LS1046A BSP memory layout

Flash layout

The following table shows the memory layout of various firmware stored in QSPI NOR flash device or micro-SD card on the FRWY-LS1046A board.

Table 28. Flash layout

Definition		Max. size	QSPI NOR / NAND Flash offset	Micro-SD card Start block no.
RCW+PBI + BL2 (bl2.pbl)		1MB	0x00000000	0x00008
TF-A FIP image (fip.bin) BL31 + BL32 + BL33		4MB	0x00100000	0x00800
Boot firmware environment		1MB	0x00500000	0x02800
DP firmware		256KB	0x00900000	0x04800
Kernel	lsdk_linux_<arch>.itb	16MB	0x01000000	0x08000
Ramdisk RFS		32MB	0x02000000	0x10000

Storage layout on micro-SD/USB for FRWY-LS1046A BSP image deployment

With FRWY-LS1046A BSP flex-installer, the FRWY-LS1046A BSP distro can be installed on a micro-SD/USB storage disk that has at least 8 GB of memory space.

Table 29. Storage layout on micro-SD/USB for FRWY-LS1046A BSP image deployment

Region 1 (4 KB)	Region 2 (RAW) 64 MB Firmware	Region 3 (Partition-1 FAT32) 20 MB EFI	Region 4 (Partition-2 EXT4) 1 GB Boot partition	Region 5 (Partition-3 EXT4) Remaining space RootFS partition
MBR/GPT	RCW U-Boot or UEFI TF-A firmware Secure boot headers FMan/DP firmware QE/uQE firmware Eth PHY firmware MC firmware DPC firmware DPL firmware DTB lsdk_linux_<arch>.itb	BOOTAA64.EFI grub.cfg	kernel image DTB lsdk_linux_<arch>.itb distro boot scripts secure headers other	Ubuntu or Ubuntu-Core or CentOS or Debian

3.5 Build tools

Flexbuild is a component-oriented build framework and integrated platform with capabilities of flexible, easy-to-use, scalable system build and distro installation. With flex-builder CLI tool, users can build various components (Linux, U-Boot, RCW, TF-A, and miscellaneous custom userspace applications) and distro userland to generate composite firmware, hybrid rootfs with customable userland. The following are Flexbuild's main features:

- Automatically builds Linux, U-Boot, TF-A, RCW, and miscellaneous user space applications
- Generates machine-specific composite firmware for various boot types: SD/QSPI NOR, both for non secure and secure boot.
- Supports integrated management with repo-fetch, repo-branch, repo-commit, repo-tag, repo-update for git repositories of all components
- Supports cross build on x86 Ubuntu 18.04 host machine for aarch64/armhf arch target
- Supports native build on aarch64/armhf machine for Arm arch target
- Supports creating an Ubuntu docker container and building BSP inside it when the host machine is using CentOS, RHEL, Fedora, SUSE, Debian, non-18.04 Ubuntu, and so on
- Scalability of integrating various components of both system firmware and user space applications
- Capability of generating custom aarch64/armhf Ubuntu userland integrated configurable packages and proprietary components

Flexbuild can separately build each component or automatically build all components, it generates the boot firmware (RCW, U-Boot, PHY firmware, kernel image, and ramdisk RFS), `lsdk_linux_arm64_LS_tiny.itb`, and the Ubuntu userland containing the specified packages and application components.

NOTE

Upgrading of toolchain is required for U-Boot, if your host machine is not a Ubutnu 18.04 system. Following are two ways to use Ubuntu 18.04 toolchain:

- Run `sudo do-release-upgrade` command to upgrade existing Ubuntu 16.04 to Ubuntu 18.04
- Run `flex-builder docker` command on the existing non-Ubuntu 18.04 host to create a ubuntu 18.04 docker container in which GCC 7.3.0 is available, and then build BSP in docker.

Chapter 4
Linux kernel

Introduction

The Linux kernel is a monolithic Unix-like computer operating system kernel. It is the central part of Linux operating systems that are extensively used on PCs, servers, handheld devices and various embedded devices such as routers, switches, wireless access points, set-top boxes, smart TVs, DVRs, and NAS appliances. It manages tasks/applications running on the system and manages system hardware. A typical Linux system looks like this:

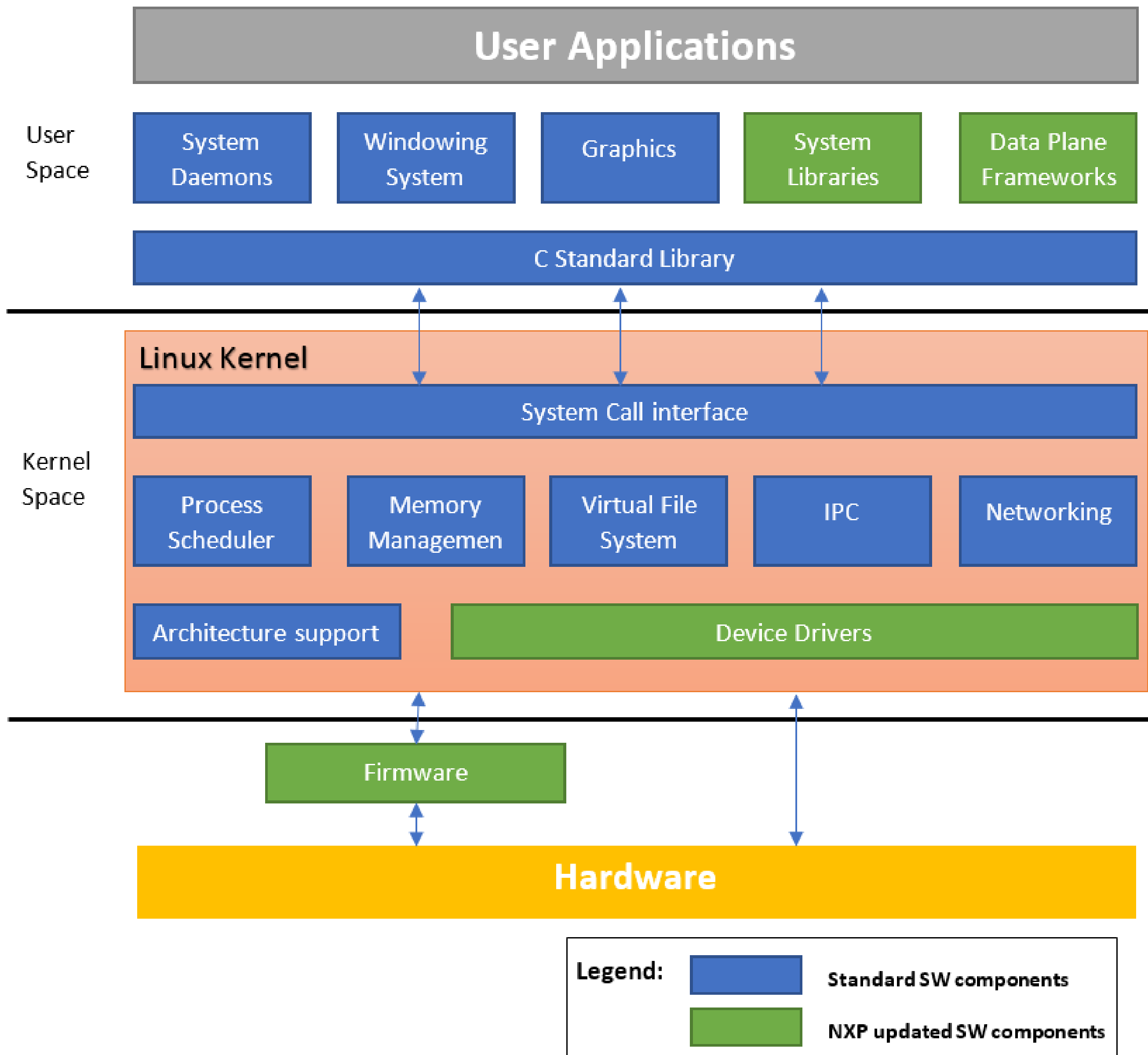


Figure 11. Typical Linux System

The Linux kernel was created in 1991 by Linus Torvalds and released as an open source project under GNU General Public License(GPL) version 2. It rapidly attracted developers around the world. In 2015 the Linux kernel has received contributions from nearly 12,000 programmers from more than 1,200 companies. The software is officially released on http://www.kernel.org website

through downloadable packages and GIT repositories. A general Linux kernel introduction from kernel.org can also be found at https://www.kernel.org/doc/html/latest/admin-guide/README.html.

Kernel Releases and relationship with Layerscape SDK

There are different Linux kernel releases coming from different sources. Below we listed the ones that are related to the LSDK kernel.

Kernel.org official kernel releases

- **Mainline**

  Mainline tree is maintained by Linus Torvalds. It's the tree where all new features are introduced and where all the exciting new development happens. New mainline kernels are released every 2-3 months.

- **Longterm (LTS)**

  There are usually several "longterm maintenance" kernel releases provided for the purposes of backporting bugfixes for older kernel trees. Only important bugfixes are applied to such kernels and they don't usually see very frequent releases, especially for older trees.

  Refer to https://www.kernel.org/category/releases.html for the current maintained Longterm releases.

Linaro LSK kernel release

Linaro is an open organization focused on improving Linux on ARM. They are also providing a Linux kernel release called Linaro Stable Kernel (LSK). It is based on kernel.org Longterm kernel releases and included ARM related features developed by Linaro. Normally these features are generic kernel features for the ARM architecture. Please refer to https://wiki.linaro.org/LSK for more information about the LSK releases.

NXP Layerscape SDK kernel

NXP's SDK kernel often contains patches that are not upstream yet so essentially the LSDK kernel is an enhanced Linaro LSK which is in turn an enhanced kernel.org LTS. In order to fully utilize the ARM open source eco-system. The kernel versions provided in NXP LSDK will be chosen from the kernel.org Longterm releases to include the important bugfixes backported. It will also include generic ARM kernel features provided by the Linaro LSK release which could be important for some users.

Getting the LSDK kernel source code

With Layerscape SDK, NXP owned/updated software components are published on github. You can use git commands to get the latest kernel source code.

- Install git command if not there already. For example, on Ubuntu:

  ```
  $ sudo apt-get install git
  ```

- Clone the Linux kernel source code with git.

  ```
  $ git clone https://source.codeaurora.org/external/qoriq/qoriq-components/linux
  ```

- Checkout the desired kernel version. As we provide support to the two latest LTS kernel versions in the SDK, it is possible that the default one is not your desired kernel version

  ```
  $ cd linux
  $ git branch
  ```

  Check the name of the current branch. If it is not the Kernel version you want, use the following command to checkout your desired kernel version: x.y

  ```
  $ git checkout -b linux-x.y origin/linux-x.y
  ```

4.1 Configuring and building Linux kernel

Configuring and building the Linux kernel is controlled by the Kbuild subsystem. You can find documents describing the internal of Kbuild subsystem under the Documentation/kbuild/ folder in the Linux source code tree, if you are adding new files or new config options to the kernel. Otherwise, as a Linux kernel user, you would probably require to only fine tune the kernel configuration based on your system requirements and build new kernel image with updated configuration. These tasks are performed using `make` commands. This section explains how to perform these tasks.

Setting environment for cross-compilation

Configuration changes in this subsection are only applicable when you are configuring and building kernel on an architecture that is different from the target. For example, compiling an Armv8 kernel on an x86 computer. If you are compiling the kernel on a machine natively of the same architecture as the target, then skip steps in this subsection.

1. Install the cross-compiler of your distribution.
2. Specify the target architecture in ARCH environment variable.
3. Specify the path and prefix of a cross-compiler in `CROSS_COMPILE` environment variable:

   ```
   $ export CROSS_COMPILE=/path/to/dir/tool-chain-prefix-
   ```

   Or, simply specify the prefix if the cross-compiler commands are already in the execution `PATH`:

   ```
   $ export CROSS_COMPILE=tool-chain-prefix-
   ```

For example, the commands needed on Ubuntu Linux will be as follows for 64-bit Arm platforms:

```
$ sudo apt-get install gcc-aarch64-linux-gnu
$ export CROSS_COMPILE=aarch64-linux-gnu-
$ export ARCH=arm64
```

For the shell environment variables exported above, you can also include them directly in each `make` command you use. For example, `$ ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- make {targets}`. Exporting them will save your time if you are frequently using `make` commands in kernel.

Configuring kernel

The current kernel configuration for a kernel source tree will be kept in a hidden .config file at the top level of the kernel source code, after you changed the configuration with a `make` config command variant. You can copy the .config file directly from one kernel source tree to another with the same kernel version to create a duplicate configuration. Also, you can edit it with a text editor, in which you can see a list of `CONFIG_*` symbols corresponding to each kernel config option.

The following targets from the Linux kernel Kbuild framework are used to load the default kernel configuration for FRWY-LS1046A BSP:

- `defconfig/${PLATFORM}_defconfig`

  Create the .config file by using the default config options of the architecture or platform defined in the `arch/$ARCH/configs/` directory. This normally includes all the device drivers needed for the architecture or platform.

- `${FRAGMENT}.config`

  Merge a configuration fragment that enables certain features into the .config file.

Specific command to load the default configuration of Layerscape Armv8 platforms (in 64-bit mode) for FRWY-LS1046A BSP is as follows:

```
$ ./scripts/kconfig/merge_config.sh arch/arm64/configs/defconfig arch/arm64/configs/lsdk.config
```

To further fine tune the configuration based on your system needs, you can use the following `make` commands:

- `$ make menuconfig`

  Choose config options in text-based color menus, radiolists, and dialogs. It is a good way to navigate through all the selectable kernel config options in a well-organized, human-readable hierarchy, and you can get a description of each option when it is highlighted by clicking the <Help> button. The "Device Drivers" section of this document also mentions the path to config options needed for a feature to work in the menuconfig.

- `$ make ${FRAGMENT}.config`

  You can also utilize this capability to enable options for a specific feature in your custom kernel configuration quickly, rather than selecting each option separately in the menuconfig. In the "Device Drivers" section of this document, the `CONFIG_*` symbols needed by a specific feature/driver are listed. Put these symbols with “=y” or “=m” depending on if you want these features/drivers to be built-in or built as loadable kernel module into a `${FEATURE}.config` file under `arch/$ARCH/configs/` directory. Run the `$ make ${FEATURE}.config` command, it will enable all these listed kernel config options together.

Building kernel

Building the kernel is simple.

- Use the following command to build kernel images and device tree images:

```
make
```

- Use the following command to build loadable kernel modules:

```
make modules
```

You can supply the `-j <NUM>` parameter to the above `make` commands to spin `NUM` concurrent threads to reduce build time on multicore systems.

After a successful build:

- Compiled kernel images are in `arch/${ARCH}/boot/` folder
- Compiled device trees (dtb files) are in `arch/${ARCH}/boot/dts` folder
- Compiled kernel modules are spread out in driver folders. You can extract them to a specific folder (for example, /folder/to/install) by using the following command:

```
$ make modules_install INSTALL_MOD_PATH=/folder/to/install
```

Install new kernel and modules

The path or naming convention of kernel images and modules are different for different Linux distributions. The following instructions are based on the convention of FRWY-LS1046A BSP.

Using Flexbuild scripts

1. Copy kernel image, DTB, and kernel modules from your kernel tree to the staging folder of the Flexbuild script (skip if you are using the *flex-builder -c linux* to build the kernel directly):

   For 64-bit Arm:

```
$ cp arch/arm64/boot/Image.gz ${path-to-flexbuild}/build/linux/kernel/arm64/
$ cp arch/arm64/boot/dts/freescale/*.dtb ${path-to-flexbuild}/build/linux/kernel/arm64/
$ make modules_install INSTALL_MOD_PATH=${path-to-flexbuild}/build/linux/kernel/arm64/
```

2. Regenerate the boot partition and rootfs (for commands below, `${ARCH} = arm32 | arm64`):

```
$ flex-builder -i mkbootpartition -a ${ARCH}
$ flex-builder -i merge-component -a ${ARCH}
$ flex-builder -i compressrfs -a ${ARCH}
```

3. Use flex-installer to deploy the updated boot partition and rootfs to the device by following instructions provided in FRWY-LS1046A BSP Quick Start on page 9.

Updating target filesystem directly

This option can be more convenient if you are compiling the kernel on the target device locally or you can easily update the filesystem of target device remotely (for example, using scp, tftp).

1. Copy your image file to `/boot` folder on the target device using `cp` command if compiled locally; use any remote update option available if compiled remotely.
2. Copy dtb files to `/boot` folder on the target device using `cp` if compiled locally; use any remote update option available if compiled remotely.
3. Update kernel modules:

NOTE

Kernel modules need to be updated after updating kernel image.

- If you compiled the kernel on the target device locally, then use the following command to update kernel modules:

```
$ make modules_install
```

- If you compiled the kernel remotely, then:
  a. Install the modules into a temporary folder (for example, `/tmp/ls1046afrwy_bsp_01/`):

  ```
  $ make modules_install INSTALL_MOD_PATH=/tmp/ls1046afrwy_bsp_01/
  ```

  b. Transfer the `lib/` directory from the above temporary location to the target device using any file transfer option and put it in the / path of the filesystem.

4.2 Device Drivers

4.2.1 Enhanced Secured Digital Host Controller (eSDHC)

Description

The enhanced secured host controller (eSDHC) provides an interface between the host system and the SD/SDIO/micro-SD cards.

The eSDHC device driver supports either kernel built-in or module.

Kernel Configure Options

Tree View

Kernel Configure Options Tree View	Description

`Device Drivers --->`
`<*>     MMC/SD/SDIO card support --->`
`<*>       MMC block device driver`
`(32)          Number of minors per block device`
	Enables SD/MMC block device driver support

`*** 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`
	Enables NXP eSDHC driver support

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_MMC	y/n	y	Enable SD/MMC bus protocol
CONFIG_MMC_BLOCK	y/n	y	Enable SD/MMC block device driver support
CONFIG_MMC_BLOCK_MINORS	integer	32	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

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

Device Tree Binding

Property	Type	Status	Description
compatible	String	Required	Should be 'fsl,esdhc'
reg	integer	Required	Register map

Example:

```
esdhc: esdhc@1560000 {
    compatible = "fsl,ls1046a-esdhc", "fsl,esdhc";
    reg = <0x0 0x1560000 0x0 0x10000>;
    interrupts = <GIC_SPI 62 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&clockgen 2 1>;
    voltage-ranges = <1800 1800 3300 3300>;
    sdhci,auto-cmd12;
    big-endian;
    bus-width = <4>;
};
```

Known Bugs, Limitations, or Technical Issues

1. Call trace of more than 120 seconds task blocking when running iozone to test card performance. This is not issue and use below command to disable the warning.

   ```
   echo 0 > /proc/sys/kernel/hung_task_timeout_secs
   ```

2. Layerscape boards could not provide a power cycle to SD card but according to SD specification, only a power cycle could reset the SD card working on UHS-I speed mode. When the card is on UHS-I speed mode, this hardware problem may cause unexpected result after board reset. The workaround is using power off/on instead of reset when using SD UHS-I card.
3. Transcend 8G class 10 SDHC card has some compatibility issue. It is observed it could not work on 50 MHz high-speed mode on LS2 boards, but other brand SD cards (Sandisk, Kingston, Sony ...) worked fine. Reducing SD clock frequency could also resolve the issue. The workaround is using other kind SD cards instead.
4. After sleep of the board, the card will get below interrupt timeout issue. This is hardware issue. CMD18 (multiple blocks read) has hardware interrupt timeout issue.

   ```
   mmc0: Timeout waiting for hardware interrupt.
   ```

5. Linux MMC stack does not have SD UHS-II support currently. It could not handle SD UHS-II card well. If UHS-I support is enabled in eSDHC dts node, the driver may make SD UHS-II card enter 1.8v mode. Only a power cycle could reset the card, so use power off/on instead of reset for SD UHS-II card if UHS-I support is enabled in eSDHC dts node.

4.2.2 Dual Universal Asynchronous Receiver/Transmitter (DUART)

A dual universal asynchronous receiver/transmitter (DUART) consists of two UARTs that act independently. The LS1046A processor implements two DUART modules.

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_SYS_NS16550_SERIAL	Enable UART support
CONFIG_SYS_NS16550_REG_SIZE	Enable UART IP register size
CONFIG_SYS_NS16550_CLK	Enable UART clock

Choosing predefined U-Boot board configs

Make the defconfig (such as `ls1046afrwy_tfa_defconfig`) include 'uart'. This will support UART.

Run-time options

Env variable	Env description	Sub option	Option description
bootargs	Kernel command line argument passed to kernel	console=ttyS0,115200	Select UART0 as the system console

Kernel configuration options

Tree view

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

Kernel configuration tree view options	Description

`Device Drivers --->`
`Character devices --->`
`Serial drivers --->`
`<*> 8250/16550 and compatible serial support`
`[*] Console on 8250/16550 and compatible serial port`
	UART driver and enable console support

Identifier

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

Option	Values	Default value	Description
CONFIG_SERIAL_8250	y/m/n	n	UART driver

Device tree binding

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

```
duart0: serial@21c0500 {
                        compatible = "fsl,ns16550", "ns16550a";
                        reg = <0x00 0x21c0500 0x0 0x100>;
                        interrupts = <GIC_SPI 54 IRQ_TYPE_LEVEL_HIGH>;
```

```
                         clocks = <&clockgen 4 1>;
                        };
```

Source files

The following source file is related to this feature in U-Boot.

Source File	Description
drivers/serial/serial_ns16550.c	The UART driver file

The following source file is related to this feature in Linux kernel.

Source File	Description
drivers/tty/serial/8250/8250_fsl.c	The UART driver file

Verification in U-Boot

1. Boot the board to bring the U-Boot to UART console:

```
U-Boot 2018.09-gde5a84f (Mar 28 2019 - 14:51:09 +0530)

SoC:  LS1046AE Rev1.0 (0x87070010)
Clock Configuration:
       CPU0(A72):1600 MHz  CPU1(A72):1600 MHz  CPU2(A72):1600 MHz
       CPU3(A72):1600 MHz
       Bus:      600  MHz  DDR:      1600 MT/s  FMAN:     700  MHz
Reset Configuration Word (RCW):
       00000000: 0c100010 0e000000 00000000 00000000
       00000010: 30400506 00805012 40025000 c1000000
       00000020: 00000000 00000000 00000000 00038800
       00000030: 20044100 24003101 00000096 00000001
Model: LS1046A FRWY Board
Board: LS1046AFRWY, boot from QSPI
SD1_CLK1 = 100.00MHZ, SD1_CLK2 = 100.00MHZ
I2C:   ready
DRAM:  1.9 GiB (DDR4, 32-bit, CL=11, ECC on)
SEC0: RNG instantiated
Using SERDES1 Protocol: 12352 (0x3040)
Using SERDES2 Protocol: 1286 (0x506)
NAND:  512 MiB
MMC:   FSL_SDHC: 0
Loading Environment from SPI Flash... SF: Detected mt25qu512a with page size 256 Bytes, erase size
64 KiB, total 64 MiB
OK
EEPROM: Read failed.
In:    serial
Out:   serial
Err:   serial
Net:   SF: Detected mt25qu512a with page size 256 Bytes, erase size 64 KiB, total 64 MiB
Fman1: Uploading microcode version 108.4.9
PCIe0: pcie@3400000 disabled
PCIe1: pcie@3500000 Root Complex: no link
PCIe2: pcie@3600000 Root Complex: no link
FM1@DTSEC1, FM1@DTSEC5, FM1@DTSEC6, FM1@DTSEC10
```

```
Hit any key to stop autoboot:  0
=>
```

Verification in Linux

1. After U-Boot startup, set the command line parameters (including `console=ttyS0,115200`) to pass to the Linux kernel in boootargs.

```
bootargs=console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500
mtdparts=1550000.quadspi:1m(rcw),15m(u-boot),48m(kernel.itb);7e800000.flash:16m(nand_uboot),
48m(nand_kernel),448m(nand_free)

=> fatload usb 0 0xa0000000 kernel-ls1046a-frwy.itb
20985175 bytes read in 541 ms (37 MiB/s)
=> bootm 0xa0000000
## Loading kernel from FIT Image at a0000000 ...
   Using 'config@1' configuration
   Trying 'kernel@1' kernel subimage
     Description:  ARM64 Linux kernel
     Type:         Kernel Image
     Compression:  gzip compressed
     Data Start:   0xa00000dc
     Data Size:    9160081 Bytes = 8.7 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: 0x80080000
     Entry Point:  0x80080000
   Verifying Hash Integrity ... OK
## Loading ramdisk from FIT Image at a0000000 ...
   Using 'config@1' configuration
   Trying 'ramdisk@1' ramdisk subimage
     Description:  LS2 Ramdisk
     Type:         RAMDisk Image
     Compression:  gzip compressed
     Data Start:   0xa08c43dc
     Data Size:    11791919 Bytes = 11.2 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:   0xa08bc724
     Data Size:    31794 Bytes = 31 KiB
     Architecture: AArch64
     Load Address: 0x90000000
   Verifying Hash Integrity ... OK
   Loading fdt from 0xa08bc724 to 0x90000000
   Booting using the fdt blob at 0x90000000
   Uncompressing Kernel Image ... OK
   Using Device Tree in place at 0000000090000000, end 000000009001ac31
WARNING failed to get smmu node: FDT_ERR_NOTFOUND
WARNING failed to get smmu node: FDT_ERR_NOTFOUND
```

```
Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.14.83-g76ab6899bd29 (root@pramod) (gcc version 7.3.0 (Ubuntu/
Linaro 7.3.0-16ubuntu3)) #2 SMP PREEMPT Mon Feb 18 12:53:04 IST 2019
[    0.000000] Boot CPU: AArch64 Processor [410fd082]
[    0.000000] Machine model: LS1046A FRWY Board
[    0.000000] earlycon: uart8250 at MMIO 0x00000000021d0500 (options '')
[    0.000000] bootconsole [uart8250] enabled
[    0.000000] efi: Getting EFI parameters from FDT:
[    0.000000] efi: UEFI not found.
[    0.000000] OF: reserved mem: initialized node qman-fqd, compatible id fsl,qman-fqd
[    0.000000] OF: reserved mem: initialized node qman-pfdr, compatible id fsl,qman-pfdr
[    0.000000] OF: reserved mem: initialized node bman-fbpr, compatible id fsl,bman-fbpr
[    0.000000] cma: Reserved 16 MiB at 0x00000000f7000000
[    0.000000] NUMA: No NUMA configuration found
[    0.000000] NUMA: Faking a node at [mem 0x0000000000000000-0x00000000fbdfffff]
[    0.000000] NUMA: NODE_DATA [mem 0xfbdc2b00-0xfbdc42bf]
[    0.000000] Zone ranges:
[    0.000000]   DMA      [mem 0x0000000080000000-0x00000000fbdfffff]
[    0.000000]   Normal   empty
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x0000000080000000-0x00000000f7ffffff]
[    0.000000]   node   0: [mem 0x00000000fb800000-0x00000000fbdfffff]
[    0.000000] Initmem setup node 0 [mem 0x0000000080000000-0x00000000fbdfffff]
[    0.000000] psci: probing for conduit method from DT.
[    0.000000] psci: PSCIv1.1 detected in firmware.
[    0.000000] psci: Using standard PSCI v0.2 function IDs
[    0.000000] psci: MIGRATE_INFO_TYPE not supported.
[    0.000000] psci: SMC Calling Convention v1.1
[    0.000000] percpu: Embedded 24 pages/cpu @ffff80007bd5d000 s61272 r8192 d28840 u98304
[    0.000000] Detected PIPT I-cache on CPU0
[    0.000000] Built 1 zonelists, mobility grouping on.  Total pages: 485128
[    0.000000] Policy zone: DMA
[    0.000000] Kernel command line: console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,mmio,
0x21d0500 mtdparts=1550000.quadspi:1m(rcw),15m(u-boot),48m(kernel.itb);7e800000.flash:
16m(nand_uboot),48m(nand_kernel),448m(nand_free)
[    0.000000] PID hash table entries: 4096 (order: 3, 32768 bytes)
[    0.000000] Memory: 1889440K/1972224K available (12668K kernel code, 1402K rwdata, 4944K
rodata, 1344K init, 917K bss, 66400K reserved, 16384K cma-reserved)
[    0.000000] Virtual kernel memory layout:
[    0.000000]     modules : 0xffff000000000000 - 0xffff000008000000   (   128 MB)
[    0.000000]     vmalloc : 0xffff000008000000 - 0xffff7dffbfff0000   (129022 GB)
[    0.000000]       .text : 0xffff000008080000 - 0xffff000008ce0000   ( 12672 KB)
[    0.000000]     .rodata : 0xffff000008ce0000 - 0xffff0000091c0000   (  4992 KB)
[    0.000000]       .init : 0xffff0000091c0000 - 0xffff000009310000   (  1344 KB)
[    0.000000]       .data : 0xffff000009310000 - 0xffff00000946ea00   (  1403 KB)
[    0.000000]        .bss : 0xffff00000946ea00 - 0xffff000009553eb8   (   918 KB)
[    0.000000]     fixed   : 0xffff7dfffe7f9000 - 0xffff7dfffec00000   (  4124 KB)
[    0.000000]     PCI I/O : 0xffff7dfffee00000 - 0xffff7dffffe00000   (    16 MB)
[    0.000000]     vmemmap : 0xffff7e0000000000 - 0xffff800000000000   (  2048 GB maximum)
[    0.000000]               0xffff7e0000000000 - 0xffff7e0001ef8000   (    30 MB actual)
[    0.000000]     memory  : 0xffff800000000000 - 0xffff80007be00000   (  1982 MB)
[    0.000000] Preemptible hierarchical RCU implementation.
[    0.000000]  RCU restricting CPUs from NR_CPUS=64 to nr_cpu_ids=4.
[    0.000000]  Tasks RCU enabled.
[    0.000000] RCU: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=4
[    0.000000] NR_IRQS: 64, nr_irqs: 64, preallocated irqs: 0
[    0.000000] GIC: Adjusting CPU interface base to 0x000000000142f000
```

```
[    0.000000] GIC: Using split EOI/Deactivate mode
[    0.000000] arch_timer: 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
[    0.008406] Console: colour dummy device 80x25
[    0.012905] Calibrating delay loop (skipped), value calculated using timer frequency.. 50.00
BogoMIPS (lpj=100000)
[    0.023325] pid_max: default: 32768 minimum: 301
[    0.028029] Security Framework initialized
[    0.032892] Dentry cache hash table entries: 262144 (order: 9, 2097152 bytes)
[    0.040470] Inode-cache hash table entries: 131072 (order: 8, 1048576 bytes)
[    0.047579] Mount-cache hash table entries: 4096 (order: 3, 32768 bytes)
[    0.054333] Mountpoint-cache hash table entries: 4096 (order: 3, 32768 bytes)
[    0.077545] ASID allocator initialised with 32768 entries
[    0.090987] Hierarchical SRCU implementation.
[    0.103821] EFI services will not be available.
[    0.116388] smp: Bringing up secondary CPUs ...
[    0.149081] Detected PIPT I-cache on CPU1
[    0.149108] CPU1: Booted secondary processor [410fd082]
[    0.177086] Detected PIPT I-cache on CPU2
```

2. After the kernel boots up to the console, you can type any shell command in the UART terminal.

4.2.3 Quad Serial Peripheral Interface (QSPI)

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 drviers --->
           < > 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  ----
     [*]   the framework for SPI-NOR support
     <*>     Freescale Quad SPI controller
```

```
Device Drivers --->
     [ ] Memory Controller drivers  ----
```

Compile-time Configuration Options

Config	Values	Defualt 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.
```

4.2.4 Universal Serial Bus interfaces

See table below for USB controllers present on the LS1046A SoC.

SoC	No. of USB 3.0 controllers present	No. of USB 2.0 controllers present
LS1046A	2	0

Typical USB node on device trees for USB 3.0 controller:

```
usb0: usb@2f00000 {
                        compatible = "snps,dwc3";
                        reg = <0x0 0x2f00000 0x0 0x10000>;
                        interrupts = <GIC_SPI 60 IRQ_TYPE_LEVEL_HIGH>;
                        dr_mode = "host";
                        snps,quirk-frame-length-adjustment = <0x20>;
                        snps,dis_rxdet_inp3_quirk;
                        usb3-lpm-capable;
                        snps,dis-u1u2-when-u3-quirk;
                        snps,incr-burst-type-adjustment = <1>, <4>, <8>, <16>;
                        snps,host-vbus-glitches;
                };
```

4.2.4.1 USB 3.0 Controller (DesignWare USB3)

Description

The U-Boot and Linux kernel driver support DWC3 USB 3.0 Dual-Role-Device (DRD) controller.

U-Boot

Host mode

With default configuration of LSDK, host mode should be ready to use, below are related CONFIG files to select.

Configure tree view options

<table>
<tr><th>Configure tree view options</th><th>Description</th></tr>
<tr><td><pre>U-Boot-->
  USB support -->
   [*] Enable driver model for USB
   [*] xHCI HCD (USB 3.0) support
   [*]   Designware USB3 DRD Core Support
   …
   [*]   Support for NXP Layerscape on-chip
xHCI USB controller
   …
   [*]   USB Mass Storage support</pre></td><td>Enables USB host controller support</td></tr>
</table>

Device tree (arch/arm/dts/fsl-ls1046a.dtsi)

```
usb0: usb@2f00000 {
                        compatible = "fsl,layerscape-dwc3";
                        reg = <0x0 0x2f00000 0x0 0x10000>;
                        interrupts = <0 60 4>;
                        dr_mode = "host";
                };
```

Source files

The driver source is maintained in the Linux kernel source tree.

Source file	Description
drivers/usb/host/xhci.c	USB HOST xHCI Controller stack
drivers/usb/host/xhci.c	USB HOST xHCI Controller stack
drivers/usb/host/xhci-fsl.c	FSL USB HOST xHCI Controller driver, basing on dwc3 driver
drivers/usb/host/xhci-dwc3.c	DWC3 controller driver
drivers/usb/host/usb-uclass.c	USB host driver
common/usb.c	USB generic driver
common/usb_hub.c	USB hub driver
cmd/usb.c	USB command-line support

Linux kernel
Host mode

With default configuration of LSDK, host mode should be ready to use, below are related CONFIGs that should have been selected.

Configure tree view options

Configure tree view options	Description

`USB support --->`
 	
`[*] xHCI HCD (USB3.0) support`
	USB host controller support.

`[*] USB Mass Storage support`	USB mass storage support.

`[*] DesignWare USB3 DRD Core support`
 	
`[*] DWc3 Mode Selection`
`[X] Dual Role mode`
	DesignWare USB3 DRD Core Support.

Table continues on the next page...

Table continued from the previous page...

Configure tree view options	Description

`Device Drivers`
`--> HID support`
`   --> USB HID support`
`      [*] USB HID transport layer    USB HID support`
	USB HID support

Device tree (arch/arm/boot/dts/freescale/fsl-ls1046a.dtsi)

```
usb0: usb@2f00000 {
                        compatible = "snps,dwc3";
                        reg = <0x0 0x2f00000 0x0 0x10000>;
                        interrupts = <GIC_SPI 60 IRQ_TYPE_LEVEL_HIGH>;
                        dr_mode = "host";
                        snps,quirk-frame-length-adjustment = <0x20>;
                        snps,dis_rxdet_inp3_quirk;
                        usb3-lpm-capable;
                        snps,dis-u1u2-when-u3-quirk;
                        snps,incr-burst-type-adjustment = <1>, <4>, <8>, <16>;
                        snps,host-vbus-glitches;
                };
```

Source files

The driver source is maintained in the Linux kernel source tree.

Source file	Description
drivers/usb/core/*	USB subsystem/framework
drivers/usb/host/xhci.c xhci-mem.c xhci-ring.c xhci-hub.c	USB xHCI (host) driver
drivers/usb/storage/scsiglue.c protocol.c transport.c usb.c	USB Mass Storage (device) driver

Known bugs, limitations, or technical issues

- Linux only allows one peripheral at one time. Make sure that when one of DWC3 controllers is set as peripheral, then the others should not be set to the same mode.
- For USB host mode, some pen drives such as Kingston / Transcend / SiliconPower / Samtec might have a compatibility issue.
- Some USB micro ports might have OTG 3.0 cable compatibility issue. An OTG 2.0 cable along with a USB standard port works fine.

4.2.5 Real Time Clock (RTC)

Description
Provides the RTC function.

Kernel configuration tree view options

Kernel configuration tree view options	Description

`Device Drivers->`
`      Real Time Clock-->`
`          [*] Set system time from RTC on startup and resume (new)`
`          (rtc0) RTC used to set the system time (new)`
`          <[*] /sys/class/rtc/rtcN (sysfs)`
`          <[*] /proc/driver/rtc (procfs for rtc0)`
`          <[*] /dev/rtcN (character devices)`
	Enable RTC driver

Compile-time configuration options

Option	Values	Default Value	Description
CONFIG_RTC_LIB	y/m/n	y	Enable RTC lib
CONFIG_RTC_CLASS	y/m/n	y	Enable generic RTC class support
CONFIG_RTC_HCTOSYS	y/n	y	Set the system time from RTC when startup and resume
CONFIG_RTC_HCTOSYS_DEVICE		"rtc0"	RTC used to set the system time
CONFIG_RTC_INTF_SYSFS	y/m/n	y	Enable RTC to use sysfs
CONFIG_RTC_INTF_PROC	y/m/n	y	Use RTC through the proc interface
CONFIG_RTC_INTF_DEV	y/m/n	y	Enable RTC to use /dev interface

Source files

The driver source is maintained in the Linux kernel source tree.

Source File	Description
drivers/rtc/	Linux RTC driver

Device tree binding

Preferred node name: rtc

Property	Type	Status	Description
compatible	string	Required	Should be "nxp,pcf2129"

Default node:

```
i2c@2180000 {
     compatible = "fsl,vf610-i2c", "fsl,ls1046a-vf610-i2c";
     #address-cells = <0x00000001>;
     #size-cells = <0x00000000>;
     reg = <0x00000000 0x02180000 0x00000000 0x00010000>;
     interrupts = <0x00000000 0x00000038 0x00000004>;
     clocks = <0x00000002 0x00000004 0x00000001>;
     dmas = <0x00000017 0x00000001 0x00000027 0x00000017 0x00000001 0x00000026>;
     dma-names = "tx", "rx";
     scl-gpios = <0x00000018 0x0000000c 0x00000000>;
     status = "okay";
     i2c-mux@77 {
         compatible = "nxp,pca9546";
         reg = <0x00000077>;
         #address-cells = <0x00000001>;
         #size-cells = <0x00000000>;
         i2c-mux-never-disable;
         i2c@0 {
             #address-cells = <0x00000001>;
             #size-cells = <0x00000000>;
             reg = <0x00000000>;
             rtc@51 {
                 compatible = "nxp,pcf2129";
                 reg = <0x00000051>;
             };
         };
     };
 };
```

Verification in Linux

Following is the RTC boot log:

```
...
rtc-ds3232 1-0068: rtc core: registered ds3232 as rtc0
MC object device driver dpaa2_rtc registered
rtc-ds3232 0-0068: setting system clock to 2000-01-01 00:00:51 UTC (946684851)
...
```

Change the RTC time in Linux kernel:

```
~ # ls /dev/rtc -l
lrwxrwxrwx    1 root     root            4 Jan 11 17:55 /dev/rtc -> rtc0
~ # date
Sat Jan  1 00:01:38 UTC 2000
~ # hwclock
Sat Jan  1 00:01:41 2000  0.000000 seconds
~ # date 011115502011
Tue Jan 11 15:50:00 UTC 2011
~ # hwclock -w
~ # hwclock
Tue Jan 11 15:50:36 2011  0.000000 seconds
~ # date 011115502010
Mon Jan 11 15:50:00 UTC 2010
~ # hwclock -s
~ # date
```

```
        Tue Jan 11 15:50:49 UTC 2011
        ~ #

        NOTE: Before using the rtc driver, make sure the /dev/rtc node in your file system is
correct. Otherwise, you need to make correct node for /dev/rtc
```

4.2.6 PCI Express Interface Controller

4.2.6.1 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

`Bus support --->`
`        [*] PCI support`
`        [*] Message Signaled Interrupts (MSI and MSI-X)`
	Enable PCI host bridge and message support

`Bus support --->`
`        PCI host controller drivers  --->`
`        [*] Freescale Layerscape PCIe controller`
	Enable NXP Layerscape PCIe controller

`Device Drivers --->`
`        [*]Network device support --->`
`            [*]Ethernet device support --->`
`                     [*] Intel devices --->`
`                        <*> Intel (R) PRO/1000 PCI-Express Gigabit Ethernet support`
	Intel PRO/1000 PCI-Express support

`Device Drivers --->`
 	
`    <*> Serial ATA and Parallel ATA drivers (libata) --->`
`                <*>   Silicon Image 3124/3132 SATA support`
	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

Table continues on the next page...

Table continued from the previous page...

Option	Values	Default Value	Description
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 conficted 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;
```

4.2.6.2 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

`Bus options --->`
`    [*] PCI Express support`
`    [*]   Root Port Advanced Error Reporting support`
`    <*>     PCIe AER error injector support`
	enable PCI-Express AER and AER-INJECTOR in kernel

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.

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@lsxxxx:~#

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 conroller(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.

4.2.6.3 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 PHC

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
None

4.2.7 CAAM Direct Memory Access (DMA)

Description

The CAAM DMA module implements a DMA driver that uses the CAAM DMA controller to provide both SG and MEMCPY DMA capability to be used by the platform. It is based on the CAAM JR interface that must be enabled in the *kernel config* as a prerequisite for the CAAM DMA driver.

The driver is based on the DMA engine framework and it is located under the DMA Engine support category in the kernel config menu.

Kernel Configure Options

Tree Overview

To enable the CAAM DMA module, set the following options for `make menuconfig`:

```
-*- Cryptographic API --->
    [*] Hardware crypto devices --->
            <*> Freescale CAAM-Multicore driver backend
            <*>     Freescale CAAM Job Ring driver backend
Device Drivers --->
            <*> DMA Engine support --->
            <*>     CAAM DMA engine support
```

NOTE

Be aware that the CAAM DMA driver depends on the CAAM and CAAM JR drivers, which also have to be enabled.

Identifier

The following configure identifier is used in kernel source code and default configuration files.

Option	Values	Default Value	Description
CONFIG_CRYPTO_DEV_FSL_CAAM_DMA	y/m/n	n	CAAM DMA engine support

Device Tree Node

Below is an example device tree node required by this feature.

```
caam_dma {
    compatible = "fsl,sec-v5.4-dma";
};
```

Source Files

The following source file is related to this feature in the Linux kernel.

Source File	Description
drivers/dma/caam_dma.c	The CAAM DMA driver

Verification in Linux

On a successful probing, the driver will print the following message in `dmesg`:

```
 [    1.443940] caam-dma 1700000.crypto:caam_dma: caam dma support with 4 job rings
```

Additionally, you can also run the following commands:

```
ls -l /sys/class/dma/
total 0
lrwxrwxrwx 1 root root 0 Jan  1  1970 dma0chan0 -> ../../devices/platform/soc/1700000.crypto/
1700000.crypto:caam_dma/dma/dma0chan0
lrwxrwxrwx 1 root root 0 Jan  1  1970 dma0chan1 -> ../../devices/platform/soc/1700000.crypto/
1700000.crypto:caam_dma/dma/dma0chan1
```

```
lrwxrwxrwx 1 root root 0 Jan  1  1970 dma0chan2 -> ../../devices/platform/soc/1700000.crypto/
1700000.crypto:caam_dma/dma/dma0chan2
lrwxrwxrwx 1 root root 0 Jan  1  1970 dma0chan3 -> ../../devices/platform/soc/1700000.crypto/
1700000.crypto:caam_dma/dma/dma0chan3
```

Component Testing

To test both the SG and memcpy capability of the CAAM DMA driver use the dmatest module provided by the kernel.

Build dmatest

Build the dmatest utility as a module by running the command:

```
$ make menuconfig
```

Then select from the kernel menuconfig to build the dmatest.ko as a module:

```
Device Drivers --->
    <*> DMA Engine support --->
    <M>    DMA Test client
```

Configure dmatest

Before testing insert the module:

```
$ insmod dmatest.ko
```

The configure the dmatest. There is a general configuration that applies for both the sg and memcpy functionality:

```
$ echo 1 > /sys/module/dmatest/parameters/max_channels
$ echo 2000 > /sys/module/dmatest/parameters/timeout
$ echo 0 > /sys/module/dmatest/parameters/noverify
$ echo 4 > /sys/module/dmatest/parameters/threads_per_chan
$ echo 0 > /sys/module/dmatest/parameters/dmatest
$ echo 1 > /sys/module/dmatest/parameters/iterations
$ echo 2000 > /sys/module/dmatest/parameters/test_buf_size
```

The above configuration is self explanatory except a few:

If you set the 'noverify' parameter to 0 it will not perform check of the copied buffer at the end of each testing round. This should be used for performance testing. Set the 'noverify' parameter to 1 for functional testing.

Set the 'dmatest' parameter to 0 to test the memcpy functionality and to 1 to test the sg functionality.

Perform the test

To perform the test simply run the command:

```
$ echo 1 > /sys/module/dmatest/parameters/run
```

Depending on the type of test performed (sg/memcpy) the output may vary. Here is an example of output obtained with the above parameters:

```
[   72.113769] dmatest: Started 4 threads using dma0chan0
[   72.105334] dmatest: dma0chan0-copy0: summary 1 tests, 0 failures 9009 iops 9009 KB/s (0)
[   72.113649] dmatest: dma0chan0-copy1: summary 1 tests, 0 failures 119 iops 119 KB/s (0)
[   72.114927] dmatest: dma0chan0-copy2: summary 1 tests, 0 failures 24390 iops 0 KB/s (0)
[   72.115098] dmatest: dma0chan0-copy3: summary 1 tests, 0 failures 37037 iops 0 KB/s (0)
```

4.2.8 Networking

4.2.8.1 DPAA1-specific Software

4.2.8.1.1 DPAA1 software architecture overview

4.2.8.1.1.1 Introduction

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 DPAA1 (Data Path Acceleration Architecture first generation) hardware and demonstrates how to best architect software to leverage the DPAA1 hardware.

NOTE

In most hardware and other past documentation, DPAA first generation is refered to as DPAA. To avoid confusion with DPAA2 (second generation), we will refer to the first generation as DPAA1 in this documentation set.

By exploring how the DPAA1 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 DPAA1 learning curve required to implement the application.

Our target audience is familiar with the material in **QorIQ Data Path Acceleration Architecture (DPAA1) Reference Manual**.

Benefits of DPAA1

The primary intent of DPAA1 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 DPAA1 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, DPAA1 determines and separates ingress flows then manages the temporary, permanent, or semi-permanent flow-to-core affinity. DPAA1 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 DPAA1 is leveraged in a particular design, the system architect can map out the application to meet the performance goals and to utilize the DPAA1 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.

4.2.8.1.1.1.1 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.

4.2.8.1.1.1.2 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.

4.2.8.1.1.1.3 Parse/classification software offload

DPAA1 provides intelligence within the IO subsection of the SoC to split traffic into user-defined queues. One advantage is that the intelligence used to divide the traffic can be leveraged at a system level.

In addition to sorting and separating the traffic, DPAA1 can append useful processing information into the stream; offloading the need for the software to perform these functions (see the following figure).

Note that DPAA1 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 DPAA1. The ability to identify and separate flow traffic is fundamental to how DPAA1 solves other multicore application issues.

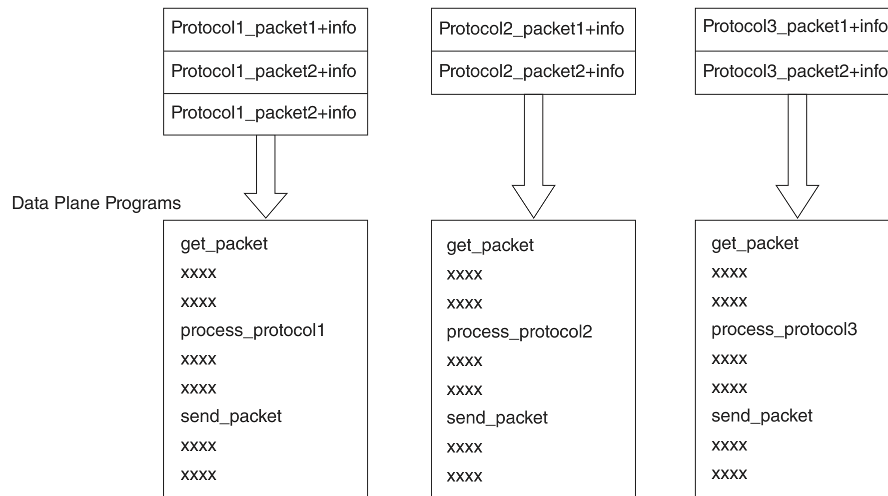


Figure 12. Hardware-sorted protocol flow

4.2.8.1.1.1.4 Flow order considerations

In most networking applications, individual traffic flows through the system require that the egress packets remain in the order 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 the below figure.

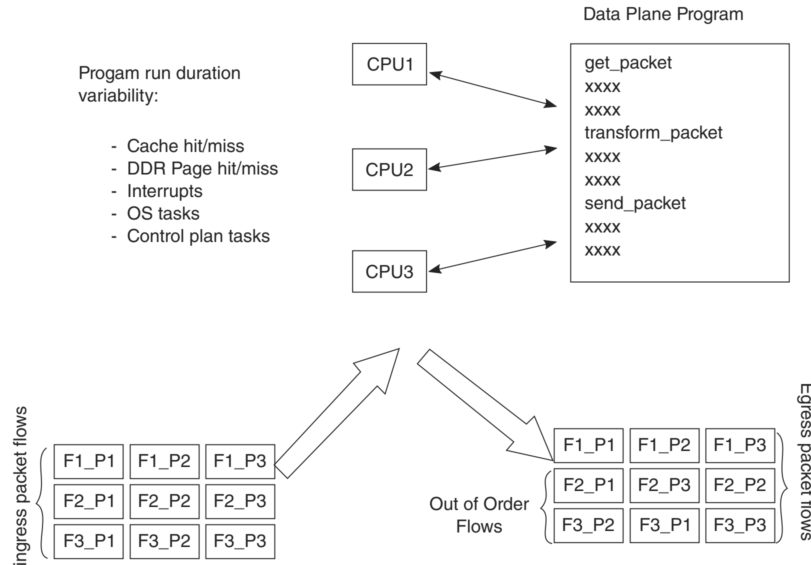


Figure 13. Multicore Flow Reordering

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. As 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 below figure). One of the goals of the DPAA1 architecture is to provide the system designer with hardware to assist with packet ordering issues.

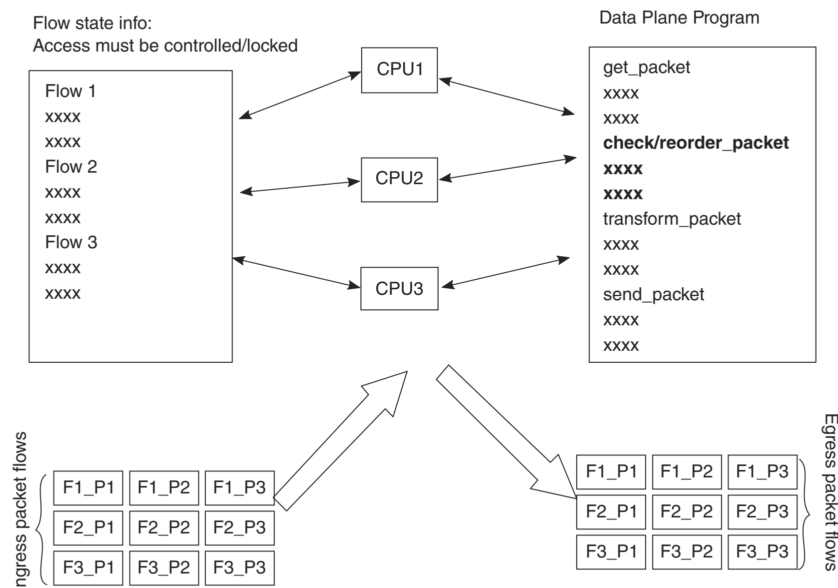


Figure 14. Implementing Order in Software

4.2.8.1.1.1.5 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 DPAA1 (Data Path Acceleration Architecture) hardware and demonstrates how to best architect software to leverage the DPAA1 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 advantage 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. DPAA1 offers several options to define and manage flow-to-core affinity.

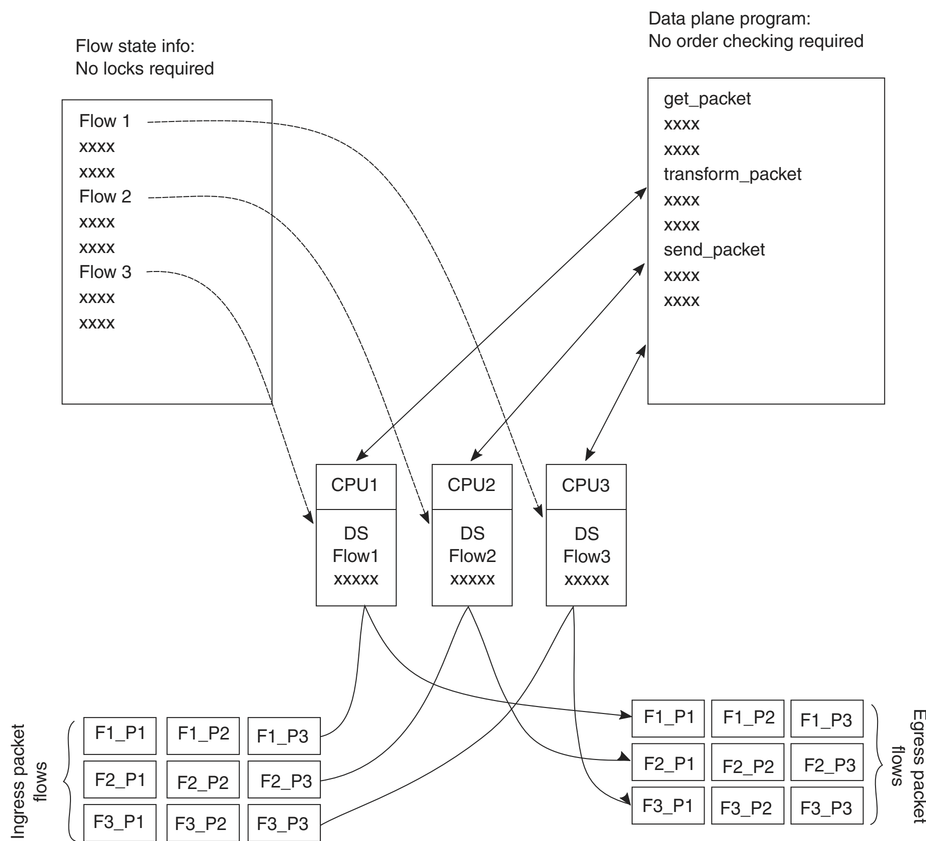


Figure 15. Managing flow-to-core affinity

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 DPAA1-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. DPAA1 can then be properly defined to implement the most efficient configuration or definition of the DPAA1 elements.

4.2.8.1.1.2 DPAA1 Goals

A brief overview of the DPAA1 elements in order to contextualize the application mapping activities. For more details on the DPAA1 architecture, see the **QorIQ Data Path Acceleration Architecture (DPAA1) Reference Manual**

The primary goals of DPAA1 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.

DPAA1 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.

4.2.8.1.1.3 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.

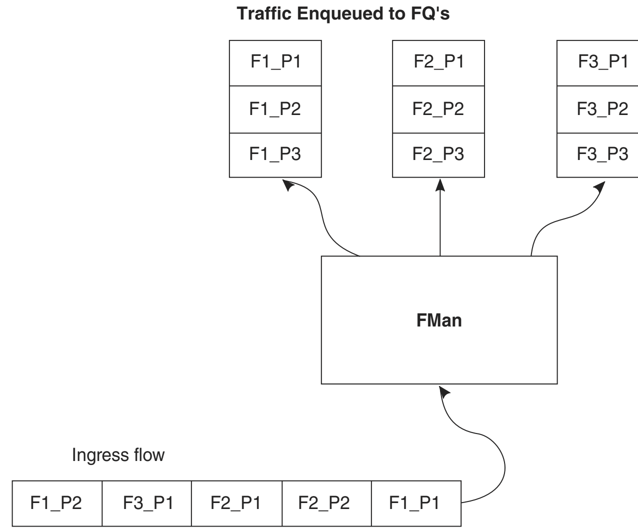


Figure 16. Ingress FMan Flow

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.

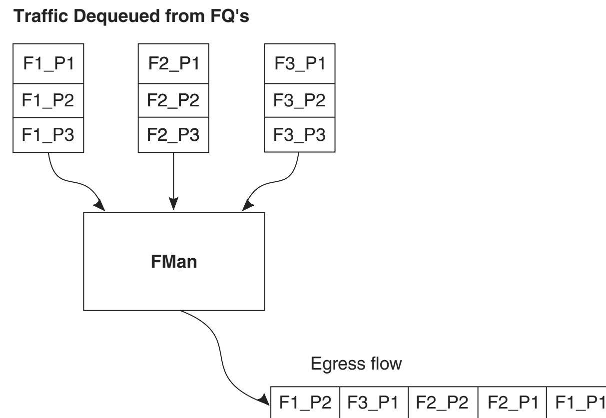


Figure 17. FMan Egress Flow

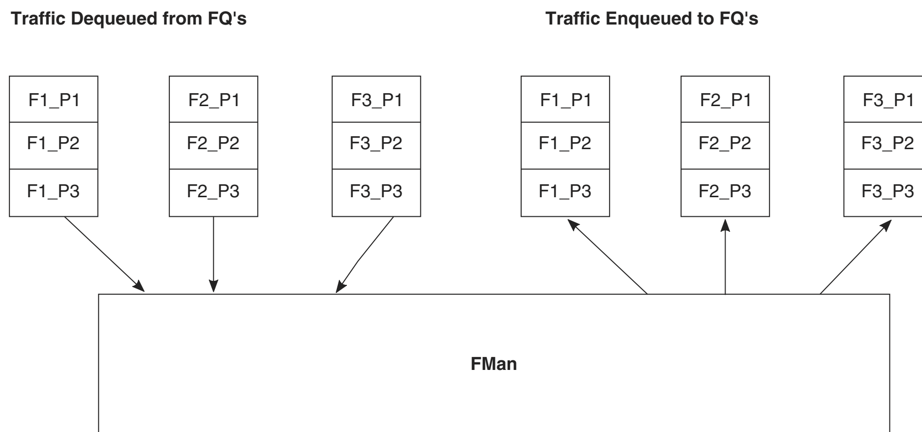


Figure 18. FMan Offline Flow

4.2.8.1.1.4 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.

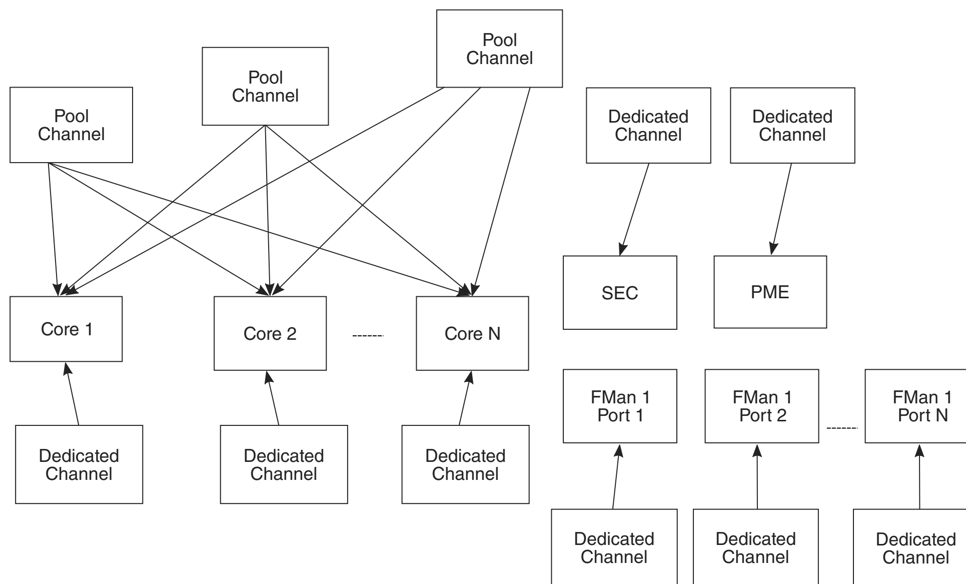


Figure 19. DPAA1 Channel Types

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 DPAA1 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.

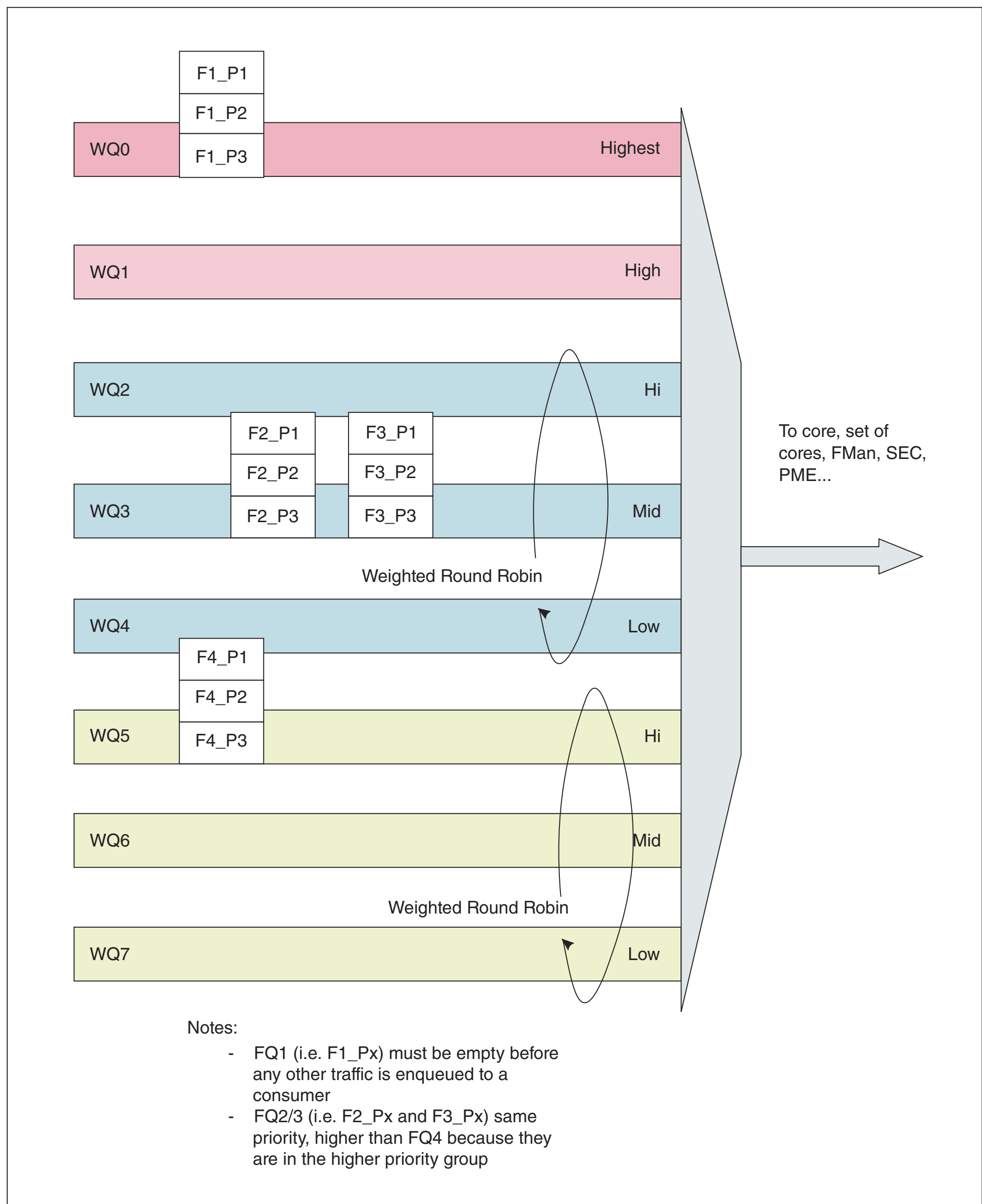


Figure 20. Prioritizing Work

QMan: Portals

A single portal exists for each non-core DPAA1 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 DPAA1 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.

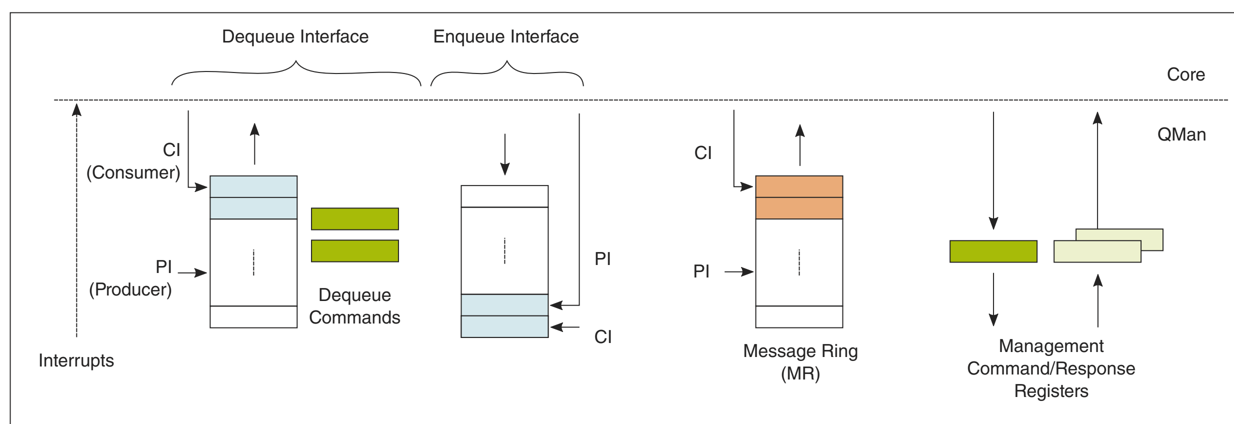


Figure 21. Processor Core Portal

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.

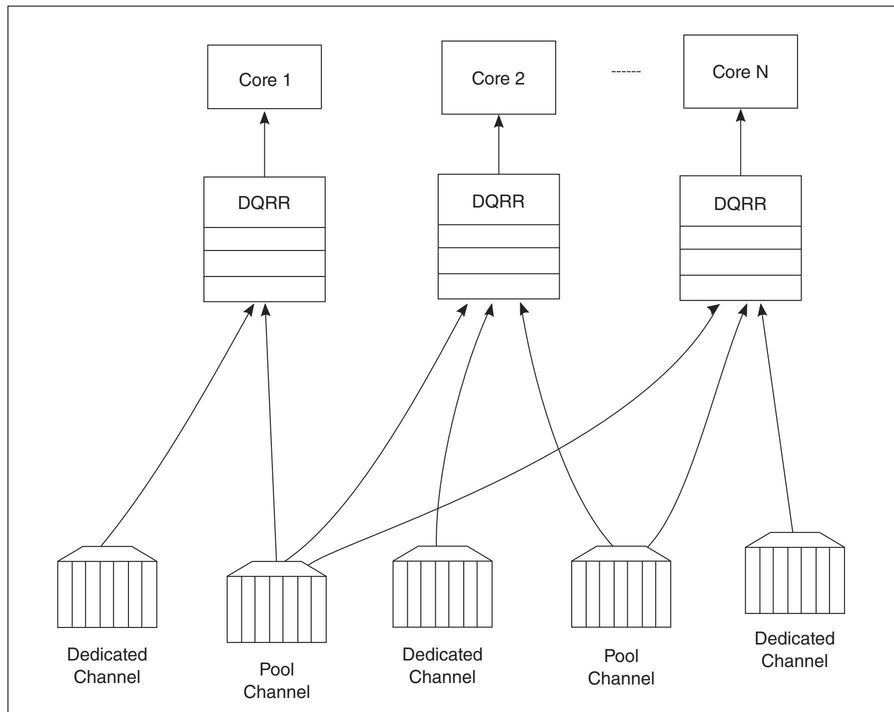


Figure 22. Ingress Channel to Portal Options

The DQRRs are tightly coupled to a processor core. DPAA1 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.

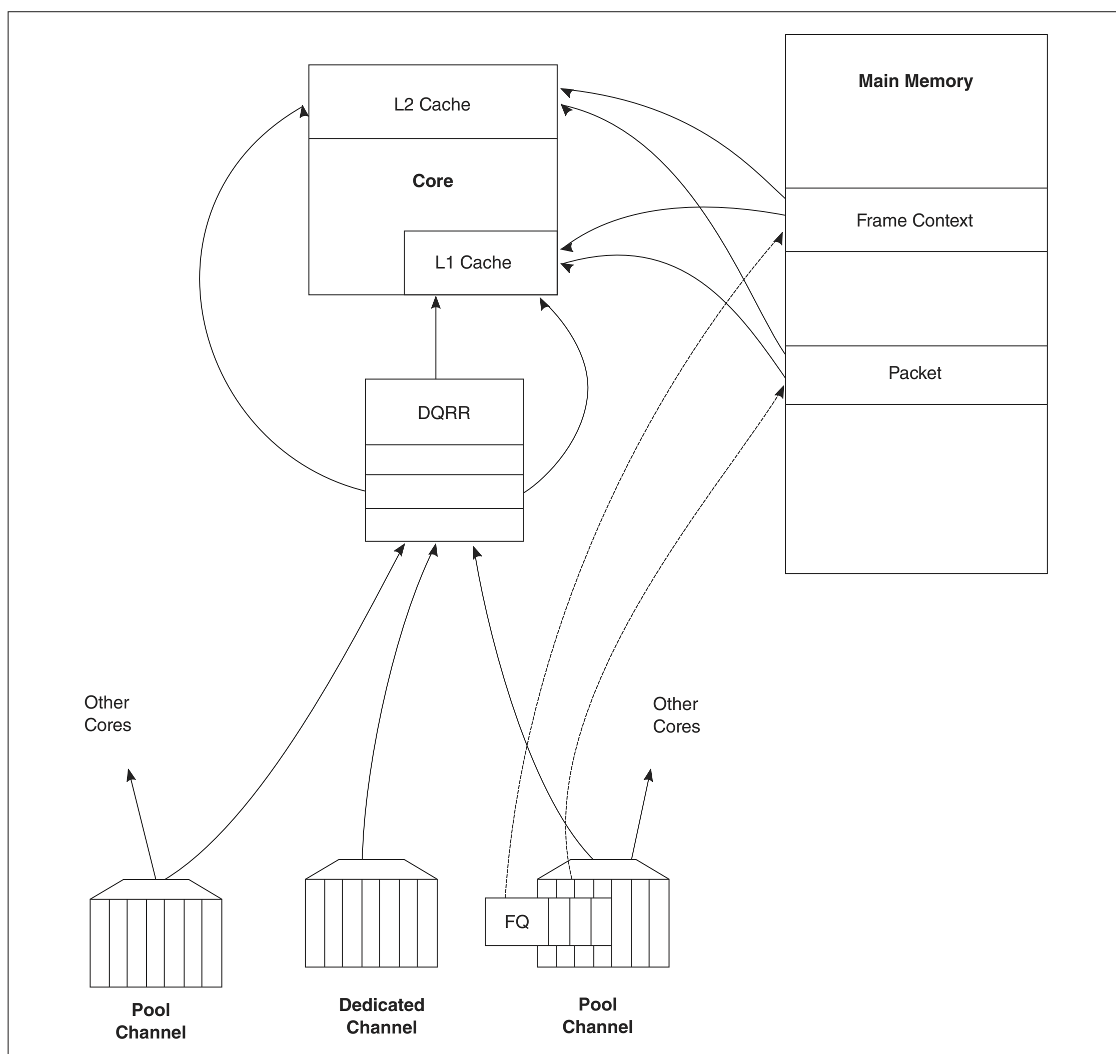


Figure 23. Cache Warming Options

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.

4.2.8.1.1.5 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 his makes the processing work granularity larger and may affect load balancing.

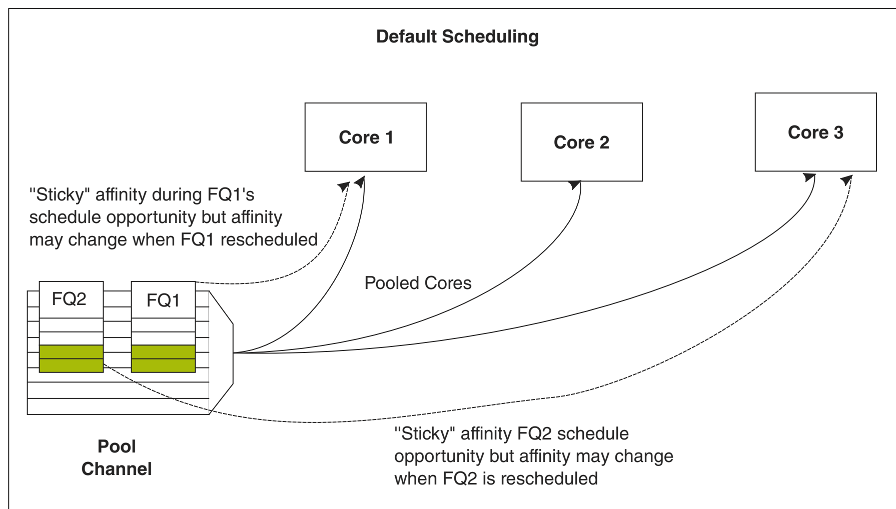


Figure 24. Default Scheduling

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, hen 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.

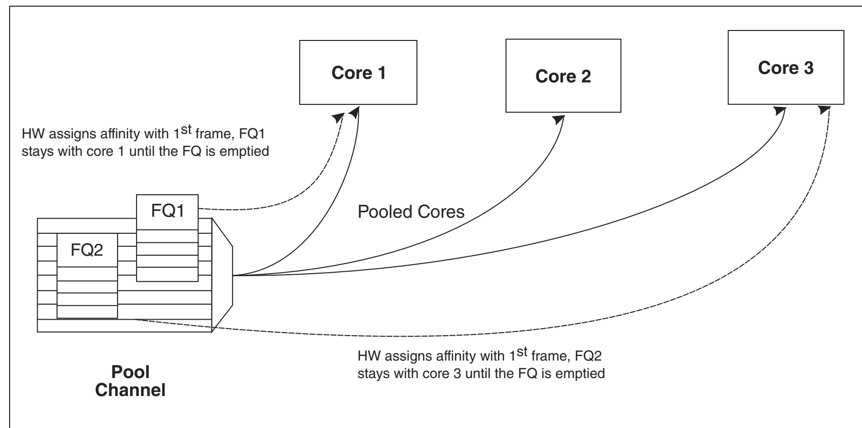


Figure 25. Hold Active Scheduling

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.

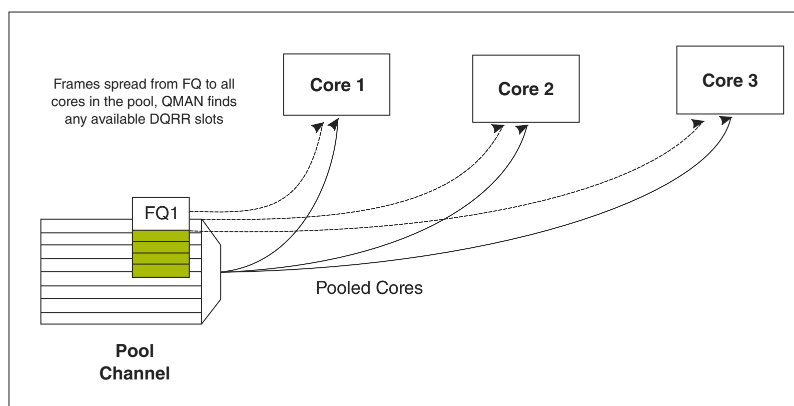


Figure 26. Avoid Blocking Scheduling

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.

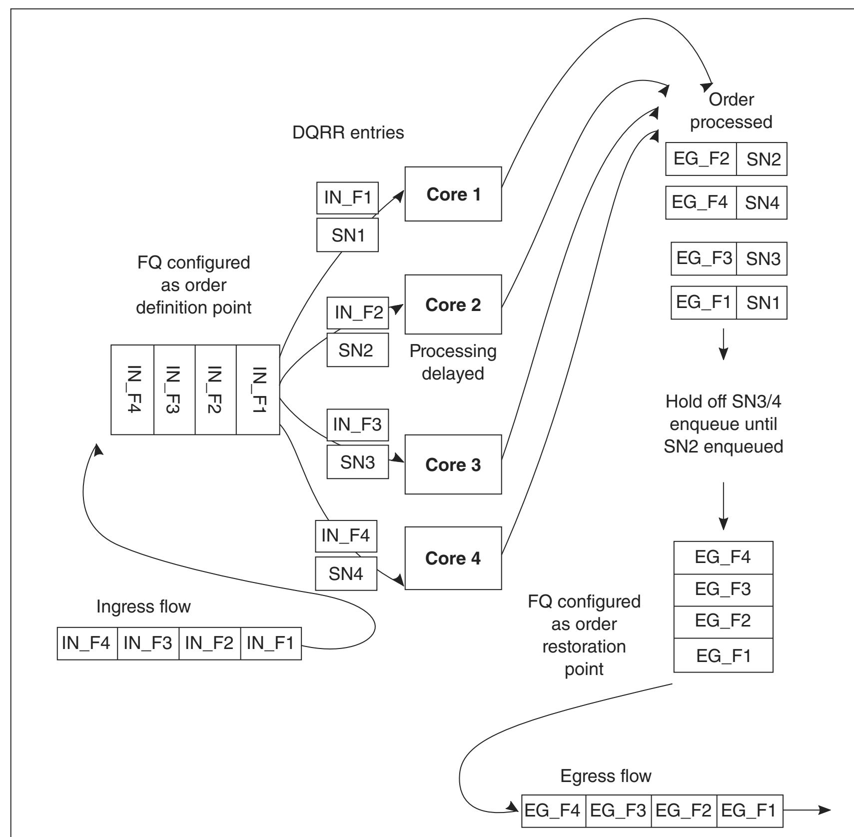


Figure 27. Order Definition/Restoration

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 enqueus 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 enqueu the late arriving frame.

4.2.8.1.1.6 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 oftware 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).

4.2.8.1.1.7 Order Handling

DPAA1 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 DPAA1 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 DPAA1 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.

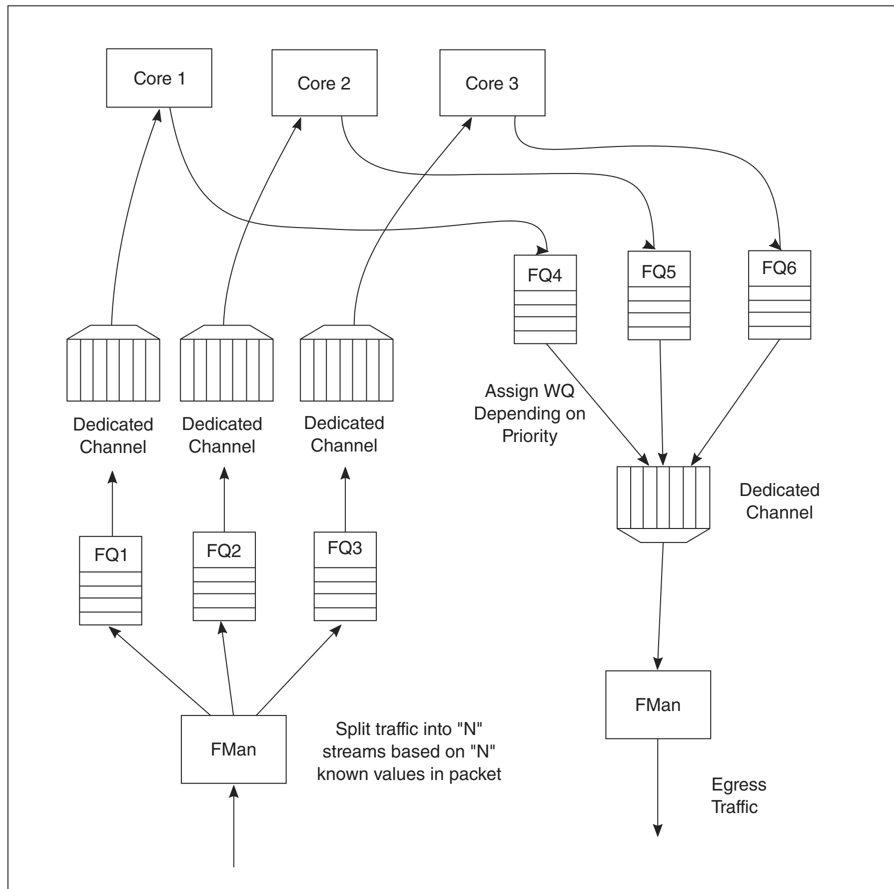


Figure 28. Direct Flow-to-Core Mapping (Order Preserved)

This technique is completely deterministic: the DPAA1 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.

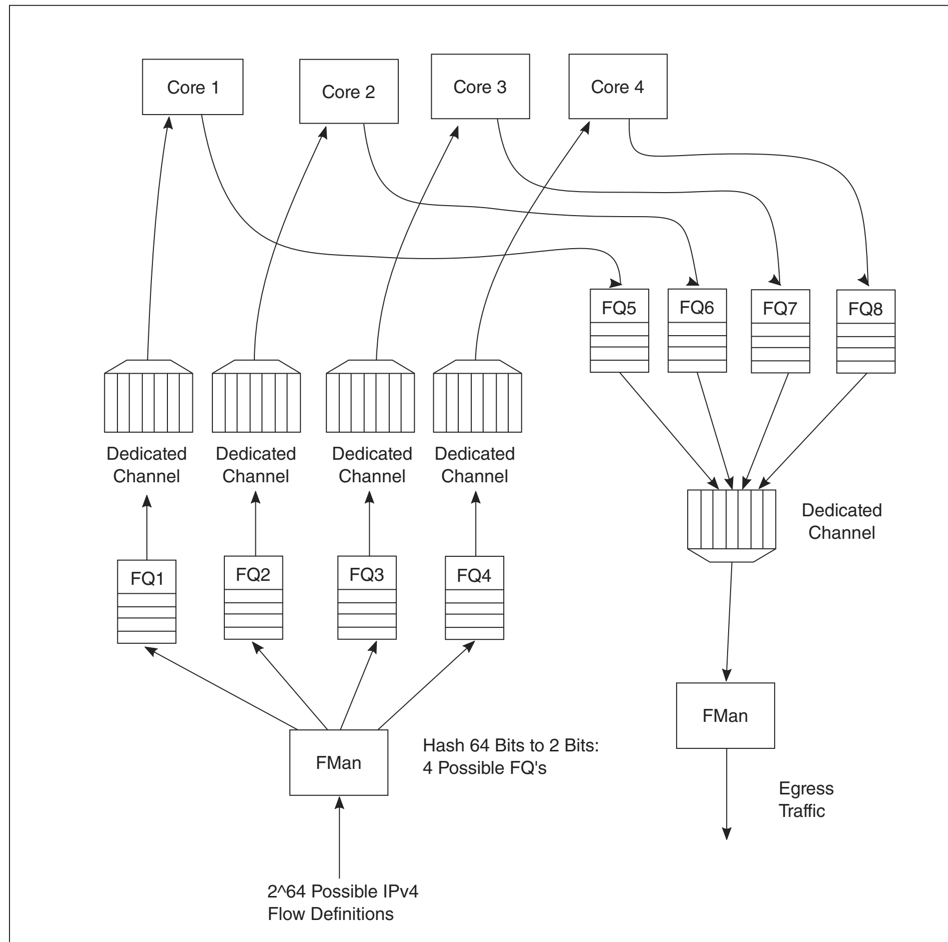


Figure 29. Simple flow distribution via hash (order preserved)

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.

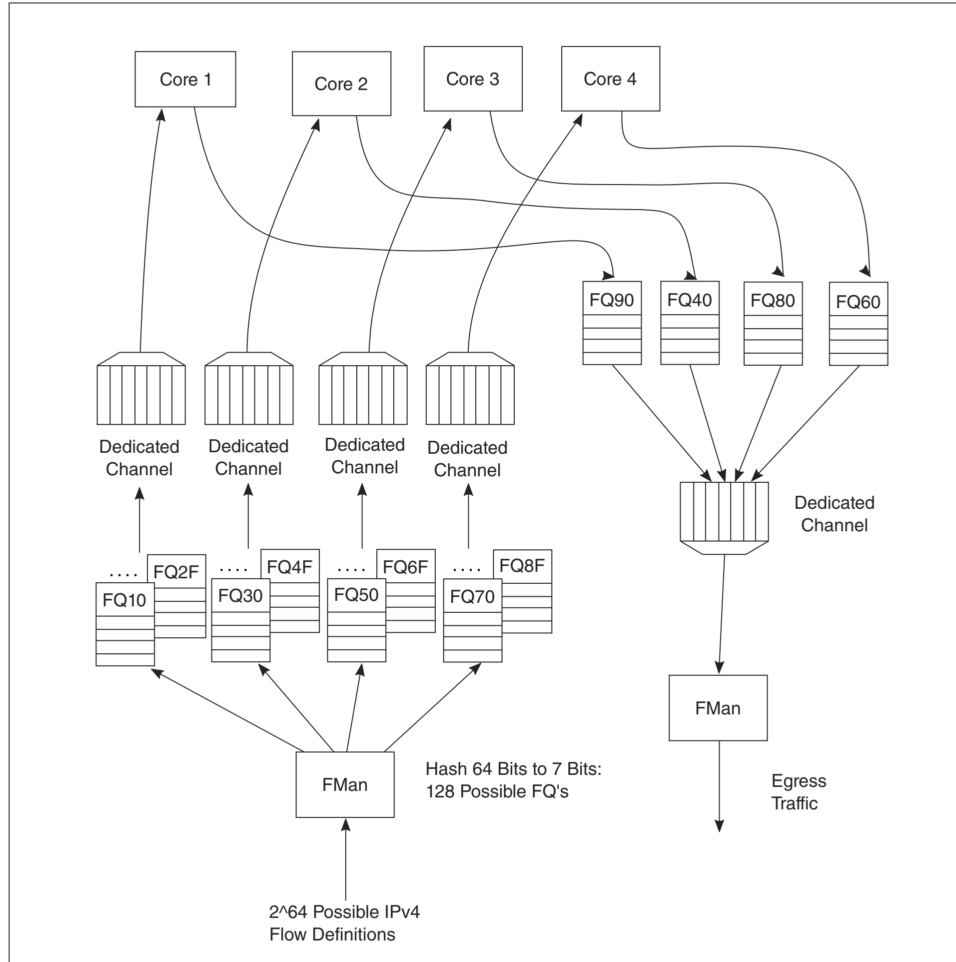


Figure 30. Using hash to assign one flow per FQ (order preserved and cache stashing effective)

4.2.8.1.1.8 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.

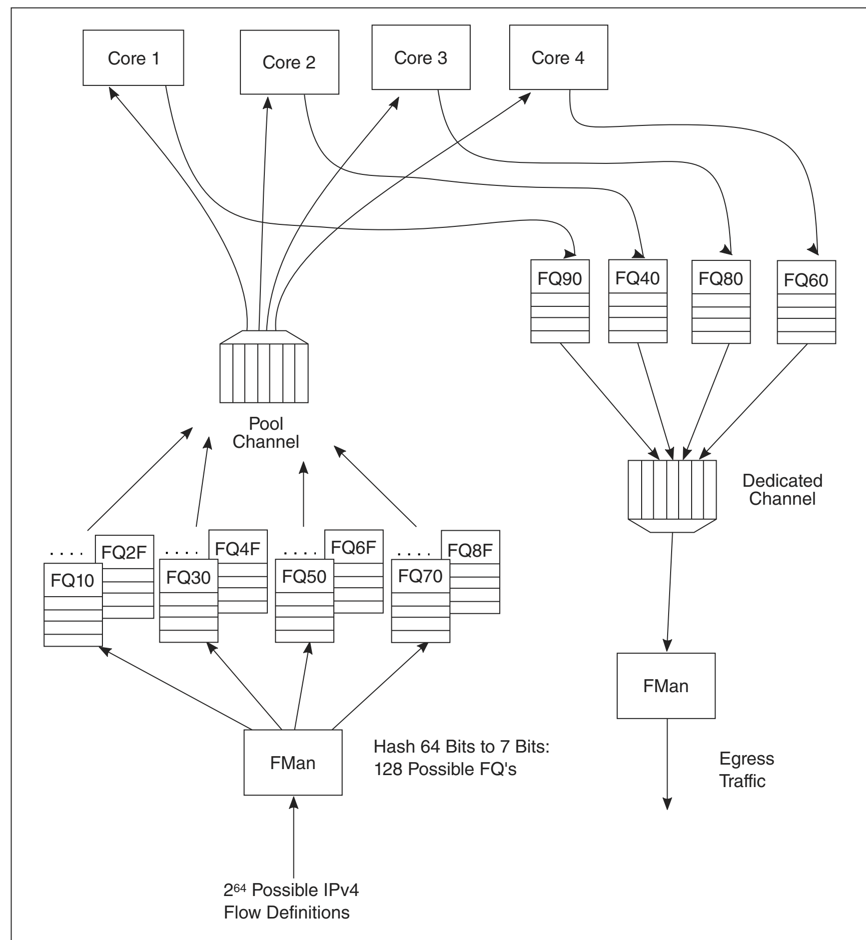


Figure 31. Using pool channel to balance processing

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 enqueuing 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.

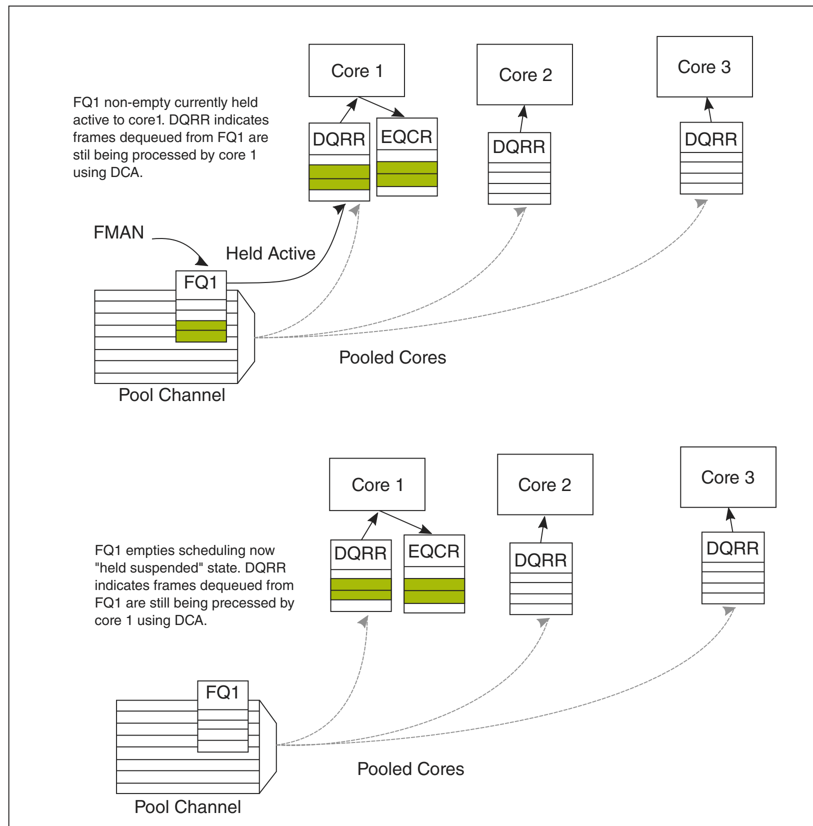


Figure 32. Hold active to held suspended mode

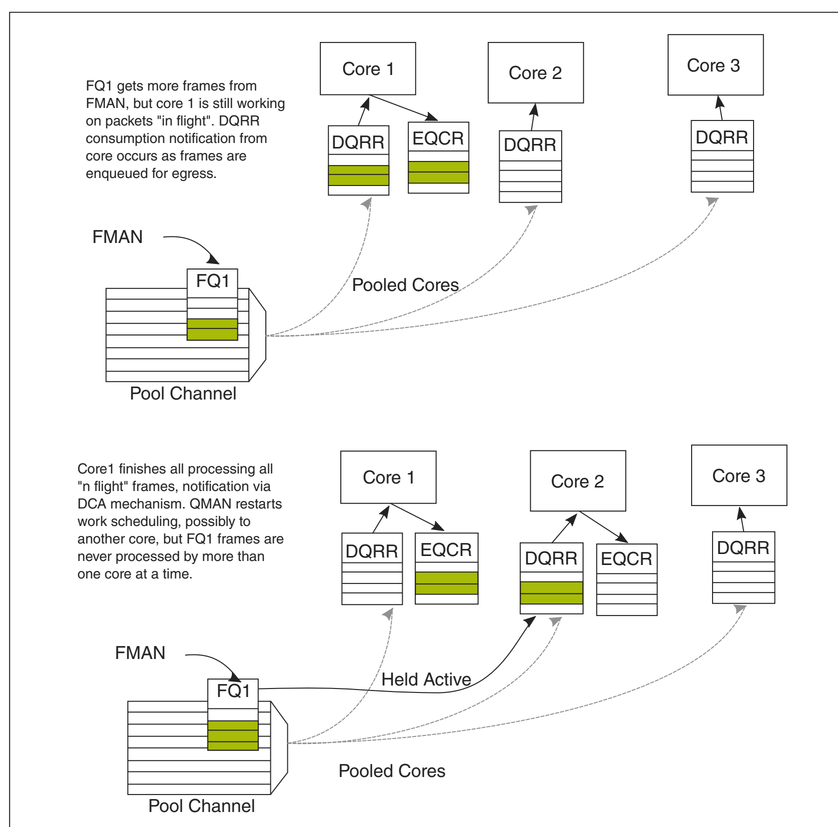


Figure 33. Held suspended to hold active mode

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 DPAA1 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.

4.2.8.1.1.9 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

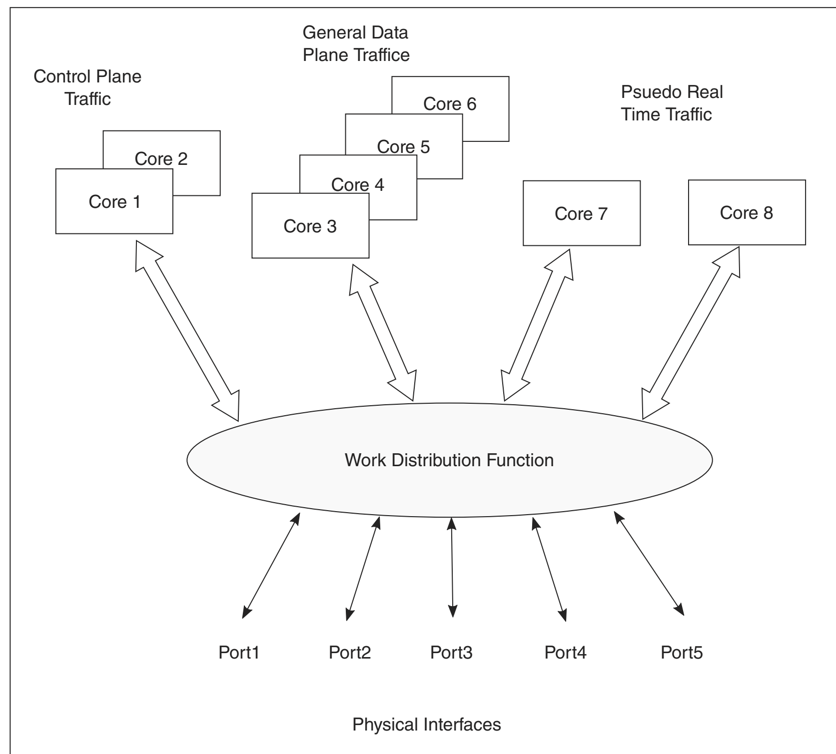


Figure 34. Example Application with Three Classes

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 30. Traffic characteristics

Characteristic	Definition
Control plane traffic	• 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 30. Traffic characteristics (continued)

Characteristic	Definition
General data plane traffic	• 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	• 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 DPAA1 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 **DPAA1 Reference Manual** and in the Software Developer Kit (SDK) used to develop the software.

4.2.8.1.1.10 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.

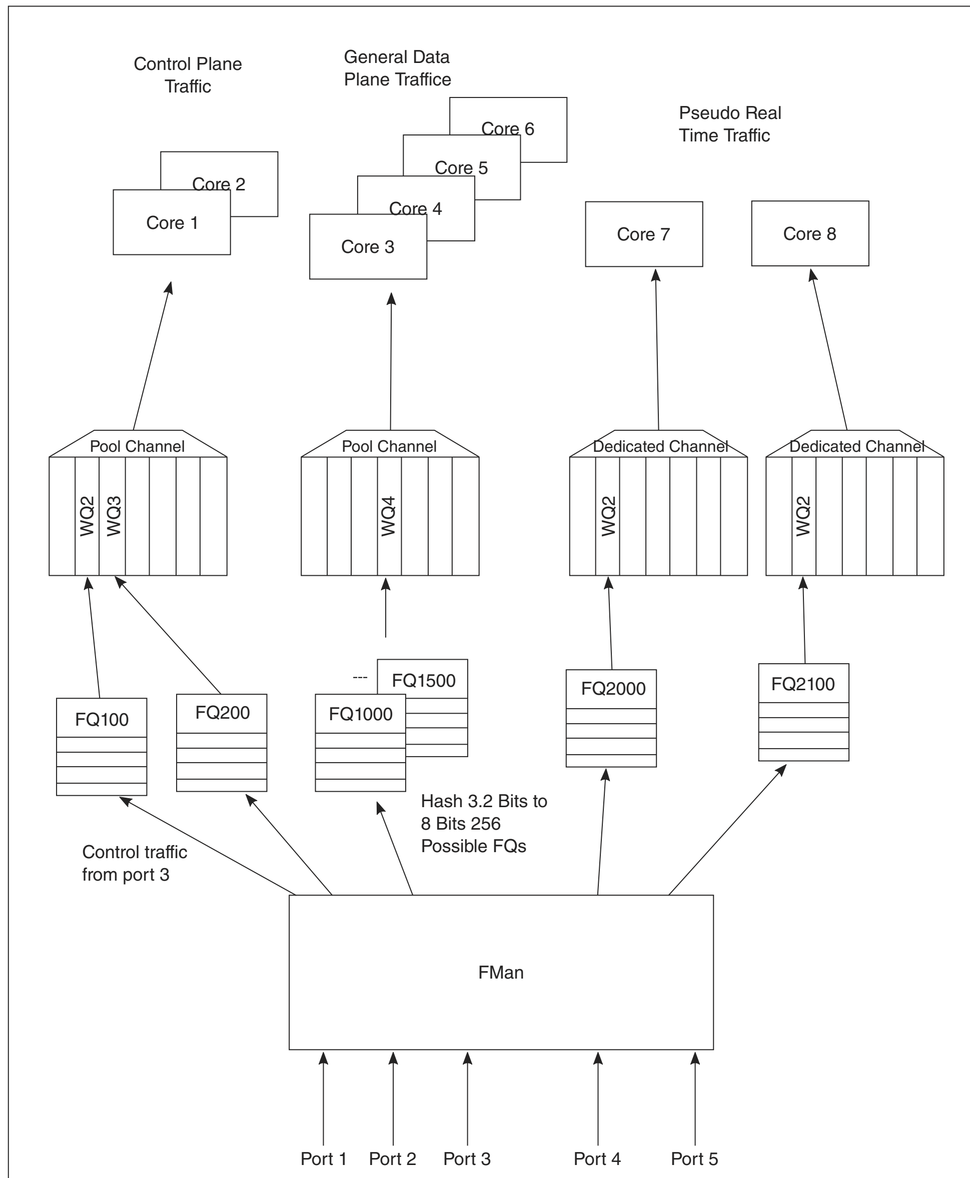


Figure 35. Ingress application map

Define egress FQ/WQ/channel configuration

For egress, the packets still flow through the system using DPAA1, 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 then 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 is 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.

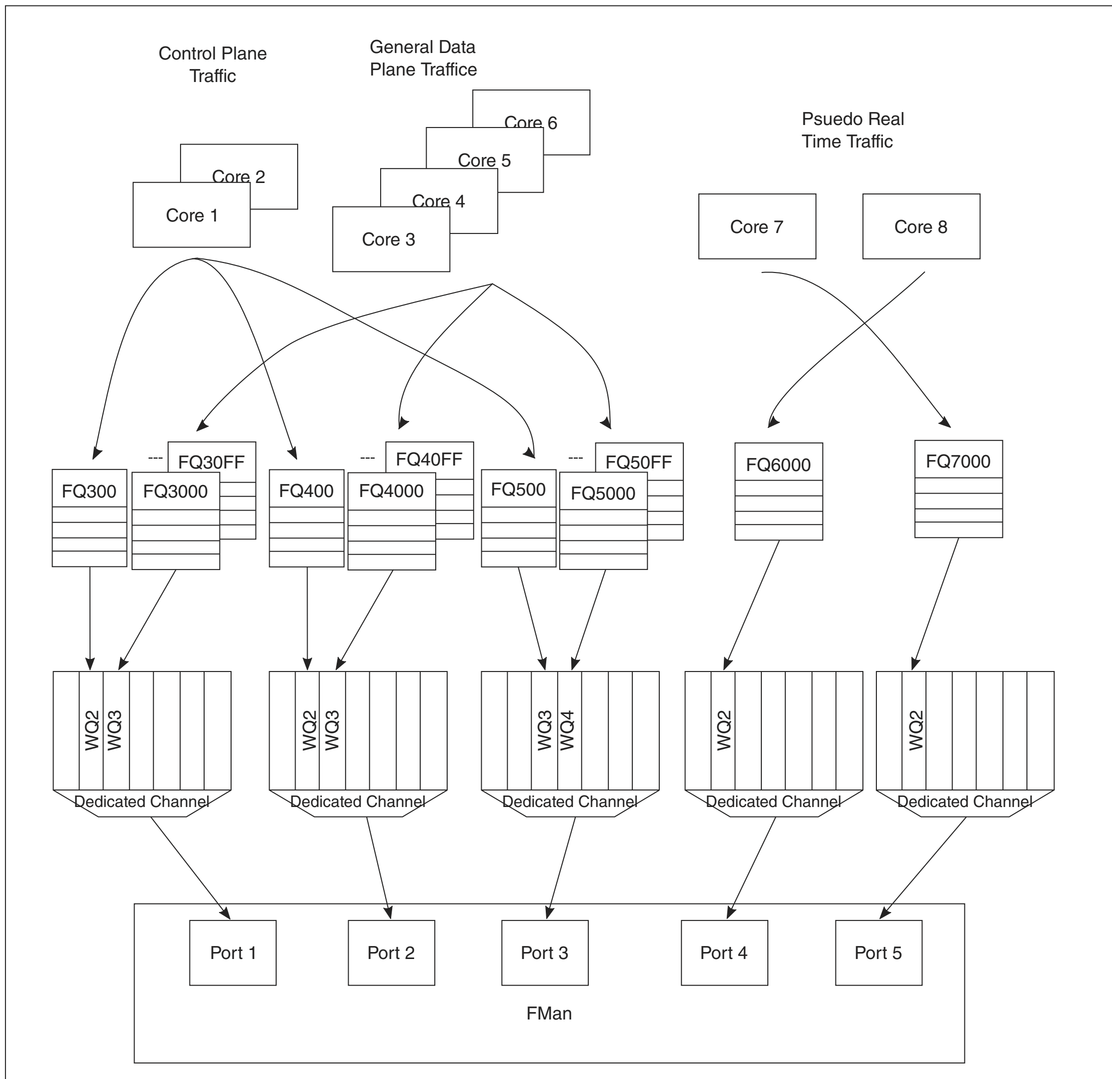


Figure 36. Egress application map

End of Document

4.2.8.1.2 Linux Ethernet

4.2.8.1.2.1 Introduction

An overview of the DPAA 1.x Ethernet network driver, in the more generic context of Linux device drivers.

The primary concepts of the DPAA 1.x Ethernet driver architecture are presented in the following sections without going into too much details as code structure. These pages are not a Linux Device Drivers tutorial, but a quick start guide which provides context for users.

The following sections describe the Linux Ethernet driver running on Datapath Acceleration Architecture (DPAA 1.x) processors. The driver is shipped with the standard QorIQ Layerscape SDK. The focus is on the theory and operation behind using Ethernet. It provides a limited discussion of the BMan, QMan, and FMan, describing the layer of software which allows all of these to interoperate. Enablement, configuration and debugging for the DPAA 1.x Ethernet Driver is also described.

Purpose

The DPAA 1.x Ethernet Driver is meant to configure the Datapath hardware for communication via the Ethernet protocol. This includes assisting in:

- 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 features are enabled on DPAA 1.x hardware by interconnecting the BMan, QMan, and FMan. The primary interactions are between the Linux Kernel and the QMan. Ethernet frames are exchanged between the Ethernet driver and the hardware Frame Queues via QMan Portals.

Usually, the Frame Queues are connected to an ingress or egress FMan port. Each FMan port has at least two queues assigned to it: a default queue and an error queue. This assignment can be specified in the device tree, or created dynamically by the driver on initialization.

Ethernet frames are often stored in buffers acquired from 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 use case, the buffers may be allocated and freed by the Kernel during network activity, or they may be allocated once and recycled by returning to the pool when not in use by the DPAA 1.x hardware.

DPAA 1.x Ethernet Driver types

The complexity of DPAA 1.x allows a variety of possible use cases. Although speed is the key factor for performance in most use cases, customization or community support are preferred in others. Building a single Ethernet driver to address all requests proved difficult without making compromises. Instead, we developed two Ethernet driver variants to approach both performance driver and community driven scenarios:

- The Private DPAA 1.x Ethernet Driver resembles the common Linux Ethernet driver. It is highly improved for performance and uses all the features that DPAA 1.x offers;
- The Upstream DPAA 1.x Ethernet Driver is integrated and maintained in the official Linux kernel tree. While younger, it benefits from streamline ease of use and community support.

Both drivers reside in the LSDK Linux kernel tree and can be built independently. The drivers can not be enabled or used at the same time. The Private Ethernet driver is enabled by default in the LSDK. Please refer to the Upstream Ethernet driver chapter for details on enabling it instead.

4.2.8.1.2.2 The DPAA1-Ethernet view of the world

This section presents the primary concepts behind the DPAA1-Ethernet driver design.

As a Linux driver, one of DPAA1-Ethernet driver's main goals is proper integration with the Linux kernel ecosystem. As a hardware device driver, the DPAA1-Ethernet driver integrates functions of several DPAA1 IP blocks, within the scope of the defined/supported use cases.

4.2.8.1.2.2.1 The Linux kernel APIs

The DPAA1-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 ARM platform device tree and discover the hardware modules that need to be configured. In particular, the DPAA1-Ethernet driver uses the platform device tree to discover:

- What net devices to probe and what type of hardware is underlying those devices;
- Which DPAA1 resources are involved; FQIDs, BPIDs, CGRIDs, FMan port IDs.

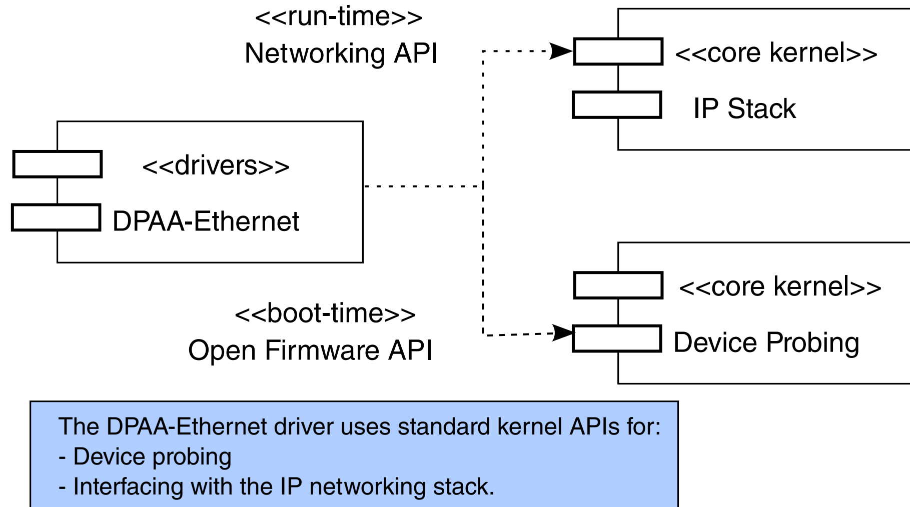


Figure 37. Platform device tree

Generally, we prefer driver 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 (For example, via booting with different RCWs)
- Seamless support of multiple partitions under the control of a hypervisor
- Cohabitation with other DPAA1 drivers (For example, a SEC driver) in the Layerscape SDK

4.2.8.1.2.2.2 The Driver's building blocks

This section presents the main structures and data entities with which the DPAA1-Ethernet driver operates.

The driver's building blocks are the relating components of the main entities with which it interacts, which are:

- The kernel's IP stack
- The DPAA1 hardware blocks and their drivers

4.2.8.1.2.2.2.1 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 DPAA1-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:

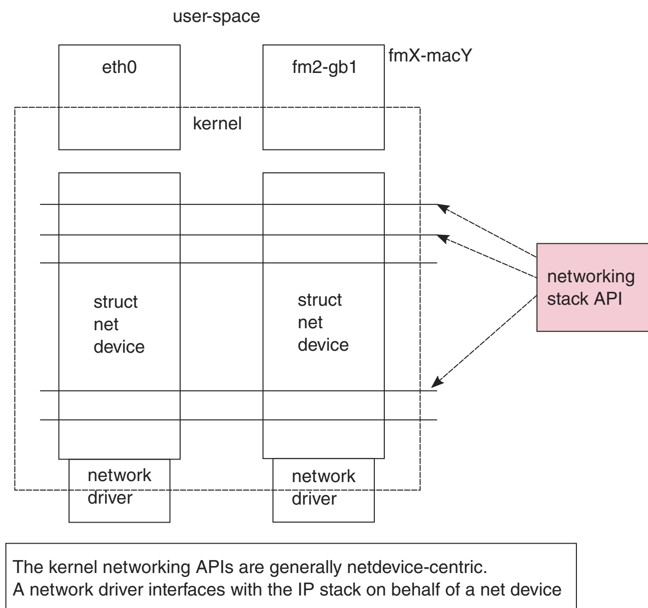


Figure 38. Every Linux Interface is Represented by a Net Device

4.2.8.1.2.2.2.2 Frame Queues

The Frame Queue is one of the fundamental concepts of DPAA1. In the case of DPAA1-Ethernet, it is the main interface between the network driver and the hardware blocks.

Ingress frames received by the DPAA1-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.

4.2.8.1.2.2.2.3 Buffer Pools

Buffer pool configuration is another fundamental part of the DPAA1-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 DPAA1-Ethernet driver.

  For instance, the driver 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 layout of incoming frames is also presumed by the driver. The actual buffer layout is outside the scope of this document and should not be assumed upon by driver users.

4.2.8.1.2.3 DPAA1 resources initialization

The rationale behind the “what”s, “why”s and “how”s of DPAA1 resource initializations made by the DPAA1-Ethernet driver are presented. This description does not go into the full detail of driver configuration.

4.2.8.1.2.3.1 What, Why and How resources are initialized

Following are the DPAA1 resources initialized by the various configurations of the DPAA1-Ethernet driver.

- FQs and FQIDs (where static config applies)
- BPs and BPIDs (where static config applies)
- Buffers (not quite “DPAA1” resources, rather “system” resources)

- CGRs (CGRIDs are always dynamic)
- FMan's online ports (Note that the offline ports are configured by a different driver than DPAA1-Ethernet)

Frame Queues and Buffer Pools have been covered at length in the previous sections. CGRs are of lesser interest from the initialization viewpoint.

FMan online ports are initially probed by the FMan Driver (FMD) and later in the boot process, they are configured by the DPAA1-Ethernet driver instances according to the specifications in the `.dts`.

4.2.8.1.2.3.2 Private Ethernet driver: Hashing/PCD frame queues

Among the frame queues initialized by the DPAA1-Ethernet driver, there is a predefined set of 128 core-affined Rx FQs, automatically initialized by the driver. They are there because most performance-enhanced setups must use a PCD configuration; to that end, the standard Layerscape 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 Core Affined Queues on page 157 section.

4.2.8.1.2.4 The (Simplified) Life of a packet

The following sections present a packet's lifecycle in the DPAA1-Ethernet driver.

4.2.8.1.2.4.1 Private net device: Tx

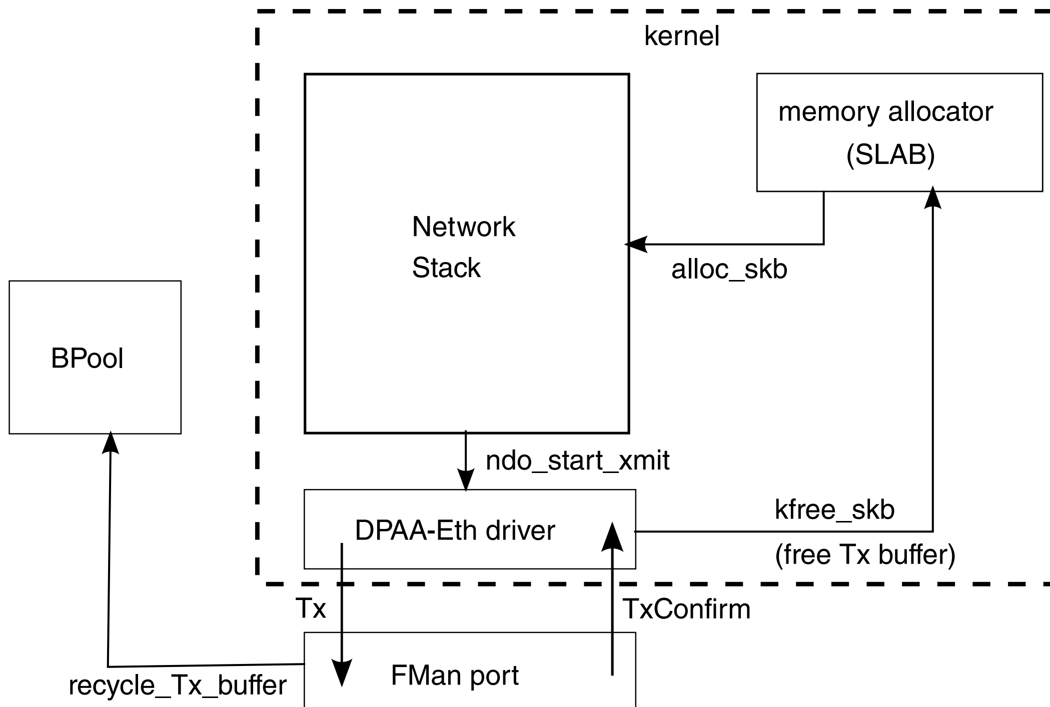


Figure 39. Buffers on the egress path

Arrows in the above diagram 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 DPAA1-Ethernet driver enqueues the packet to the FMan port with an indication to recycle the buffer if possible. If recycling is not possible, the DPAA1-Ethernet driver itself frees the buffer memory back to the kernel's allocator, when Tx delivery is confirmed by FMan.

4.2.8.1.2.4.2 Private net device: Rx

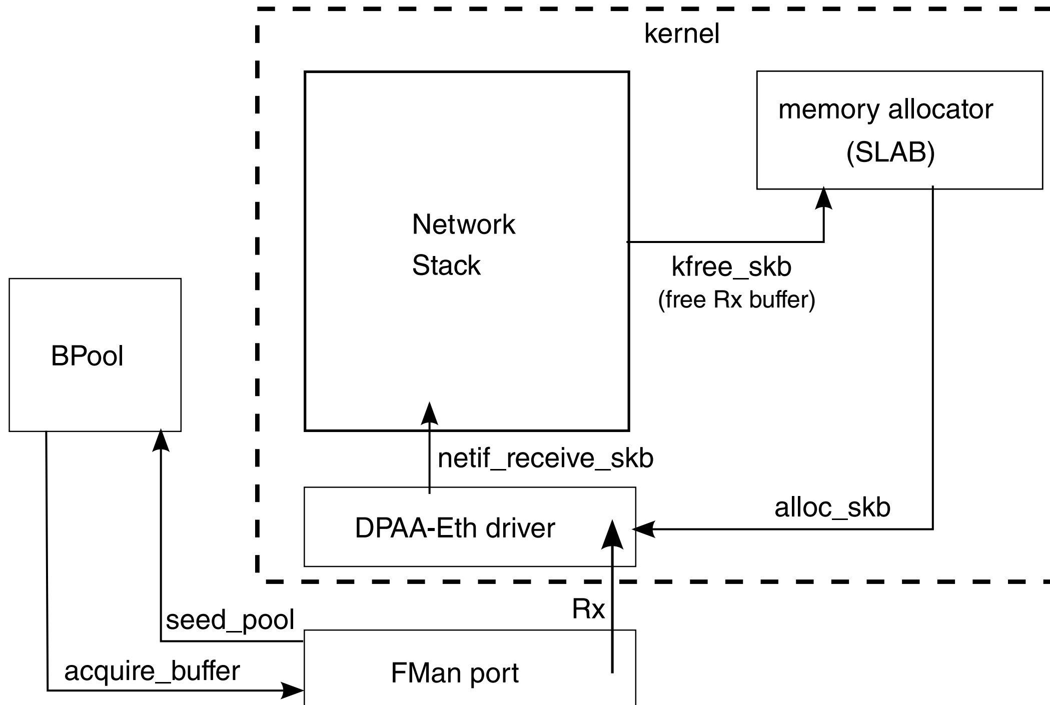


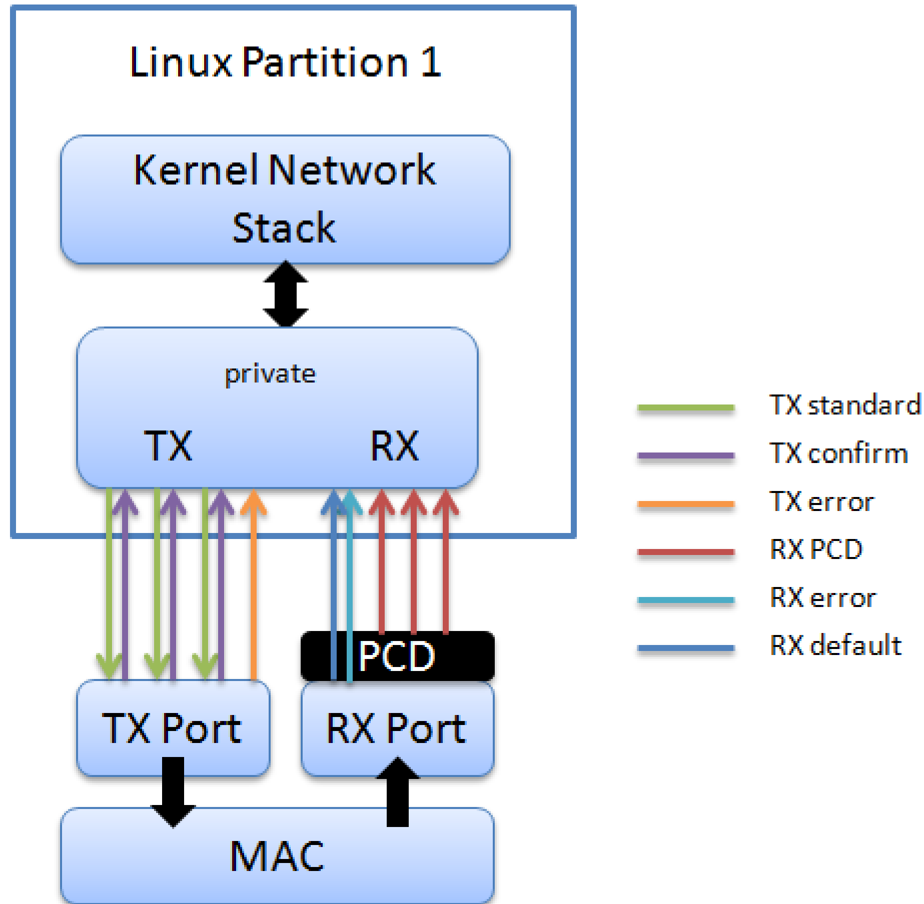
Figure 40. Buffers on the ingress path

Buffers on the ingress path are acquired by FMan directly from a Buffer Pool which was seeded by the DPAA1-Ethernet driver. Buffer layout is important to the driver, which assumes ownership on the BP. Arrows in the above diagram represent the direction of the buffer/packet flow.

4.2.8.1.2.5 Private Ethernet Driver

The Private DPAA 1.x 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 1.x facilities in order to increase the performance in both termination and forwarding scenarios.

The Private DPAA 1.x Ethernet driver will be further referenced as the Private driver.

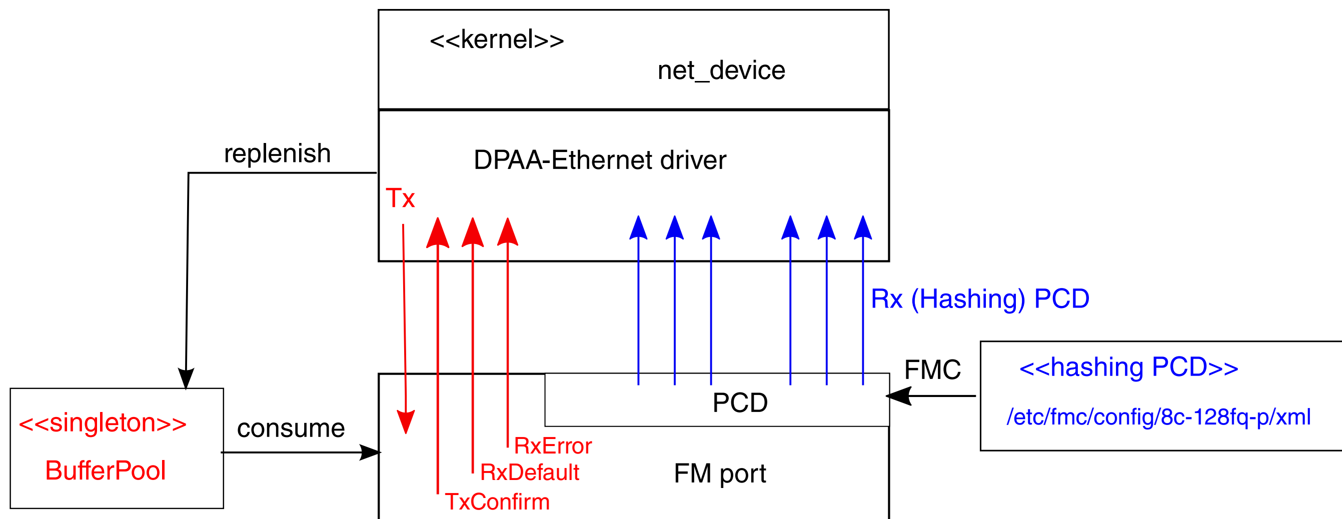


4.2.8.1.2.5.1 Network driver

The main characteristics of the private 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 frame queue configuration from the device tree at startup
- The above resources are allocated and visible only to the private driver

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 41. Network traffic between the Linux kernel and the physical FMan port

4.2.8.1.2.5.2 Configuration

This section presents the configuration options for the Private DPAA1 ethernet driver.

4.2.8.1.2.5.2.1 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 LS1043ARDB device tree node:

```
ethernet@0 {
    compatible = "fsl,dpa-ethernet";
    fsl,fman-mac = <&enet0>;
};
```

“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 are 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,dpa-ethernet";
        fsl,fman-mac = <&enet0>;
        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 raging 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.

4.2.8.1.2.5.2.2 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 DPAA1 ethernet driver throughput for large frames termination traffic (For example, 4K and above).

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.

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_CEETM - "DPAA1 CEETM QoS"

Enables QoS offloading support through the CEETM hardware block.

FSL_DPAA_CEETM_CCS_THRESHOLD_1G - "CEETM egress congestion threshold on 1G ports"

The size in bytes of the CEETM egress Class Congestion State threshold on 1G ports. The threshold needs to be configured keeping in mind the following factors:

- A threshold too large will buffer frames for a long time in the TX queues, when a small shaping rate is configured. This will cause buffer pool depletion or out of memory errors. This in turn will cause frame loss on RX.
- A threshold too small will cause unnecessary frame loss by entering congestion too often.

FSL_DPAA_CEETM_CCS_THRESHOLD_10G - "CEETM egress congestion threshold on 10G ports"

The size in bytes of the CEETM egress Class Congestion State threshold on 10G ports.

FSL_DPAA_ETH_USE_NDO_SELECT_QUEUE - "Use driver's Tx queue selection mechanism"

The DPAA1-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 you want to be in control of the egress FQ-to-CPU selection and mapping, or do not 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 DPAA1-Ethernet private port's buffer pool. One need not 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 DPAA1-Ethernet private port's buffer pool. One need not 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_DEBUG - "DPAA1 ethernet debug support"

This option compiles debug code for the DPAA1 Ethernet driver.

4.2.8.1.2.5.2.3 Bootargs

The following bootarg parameters are defined for the Frame Manager driver. However, 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 DPAA1 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 the following 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 the U-Boot environment, add "fsl_fm_max_frm=<your_MAXFRM>" directly to the "bootargs" variable.

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 communicate 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 (64 bytes) offers best performance for the case when forwarded frames are being encapsulated (For example, IPSec).

The RX extra headroom can be set in the following 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 the U-Boot environment, add "fsl_fm_rx_extra_headroom=< your_rx_extra_headroom>" directly to the "bootargs" variable.

4.2.8.1.2.5.2.4 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 weather 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.
```

4.2.8.1.2.5.3 Features

This section present the private DPAA1 ethernet driver features.

4.2.8.1.2.5.3.1 Congestion management

QMan offers the following three methods of managing congestion.

- WRED
- Congestion State Tail Drop (CSTD)
- FQ Tail Drop (FQTD)

The Private driver implements CSTD both on TX and 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. For more details, see section Kconfig options on page 148.

4.2.8.1.2.5.3.2 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 (For example, 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.

NOTE

On LS1043A, Scatter/Gather frames are not supported on Tx.

4.2.8.1.2.5.3.3 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 149.

4.2.8.1.2.5.3.4 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 DPAA1 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 GSO, 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 do not 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 10 Gbit/s traffic, and to a lesser extent (if any) on 1 Gbit/s 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 1 Gbit/s ports sees little benefit from GRO/GSO. Also, if the Private Driver detects that PCDs are not in place, GRO is automatically by-passed.

4.2.8.1.2.5.3.5 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 DPAA1 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.

4.2.8.1.2.5.3.6 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 31. 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.

4.2.8.1.2.5.3.7 Priority Flow Control

The DPAA1 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.

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 DPAA1 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.

skb_prio	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
```

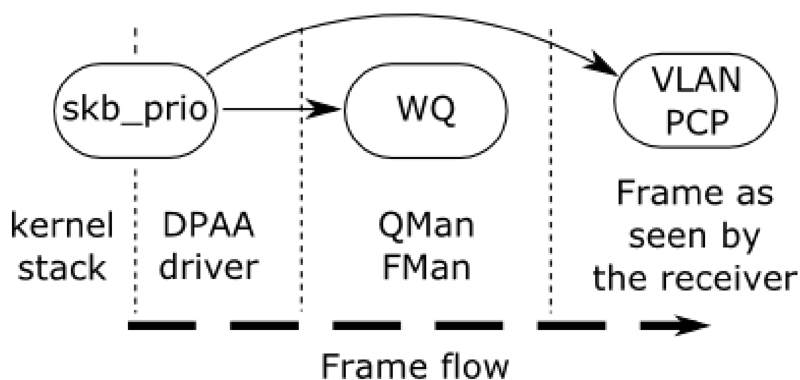
VLAN tagging

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 dequeing for 100 bit times, and dequeing is done from WQ1 and WQ2.

Generating PFC frames

All DPAA1 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							
...							

Enabling and disabling PFC using ethtool

Display PFC settings in use for an interface:

```
ethtool -a intf_name
```

Triggering PFC frames ON/OFF

PFC 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

When autonegotiation is enabled and the user enables/disables PFC frames on Rx/Tx, these will not automatically be triggered on/off. Instead, the local and the peer PFC 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 results of link negotiation.

PFC frame autonegotiation can also be enabled/disabled using ethtool -A:

```
ethtool -A intf_name autoneg on
ethtool -A intf_name autoneg off
```

4.2.8.1.2.5.3.8 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:

((MAC register address) & 0x1fffff) >> 6

Following are the values for various QorIQ DPAA1 platforms:

Table 32. FMan devices core affined queues

Interface	FQID base	LS1043A	LS1046A
fm1-mac1	0x3800	Y	
fm1-mac2	0x3880	Y	
fm1-mac3	0x3900	Y	Y
fm1-mac4	0x3980	Y	Y
fm1-mac5	0x3a00	Y	Y
fm1-mac6	0x3a80	Y	Y
fm1-mac9	0x3c00	Y	Y
fm1-mac10	0x3c80		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 + ((MAC register address) & 0x1fffff) >> 6

Following are the values for various QorIQ DPAA1 platforms:

Table 33. FMan devices high priority core affined queues

Interface	FQID base	LS1043A	LS1046A
fm1-mac1	0x13800	Y	
fm1-mac2	0x13880	Y	
fm1-mac3	0x13900	Y	Y
fm1-mac4	0x13980	Y	Y
fm1-mac5	0x13a00	Y	Y
fm1-mac6	0x13a80	Y	Y
fm1-mac9	0x13c00	Y	Y
fm1-mac10	0x13c80		Y

4.2.8.1.2.5.4 Quality of Service

DPAA1 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 Frame Manager Configuration (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.

4.2.8.1.2.5.4.1 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 Policer Section on page 315.

4.2.8.1.2.5.4.2 Scheduling and Shaping

4.2.8.1.2.5.4.2.1 Description

Specific DPAA1 platforms offer scheduling, shaping and prioritization capabilities through CEETM (Customer Edge Egress Traffic Management). The CEETM hardware block is a member of the QMan. Its purpose is to enhance the performances of DPAA1 platforms by moving the egress QoS logic from software to hardware.

This section 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.

4.2.8.1.2.5.4.2.1.1 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** section.

Refer the figure below for a CEETM block, which is available for each FMan and it is intended to be used by FMan sub-portals linked to Ethernet interfaces.

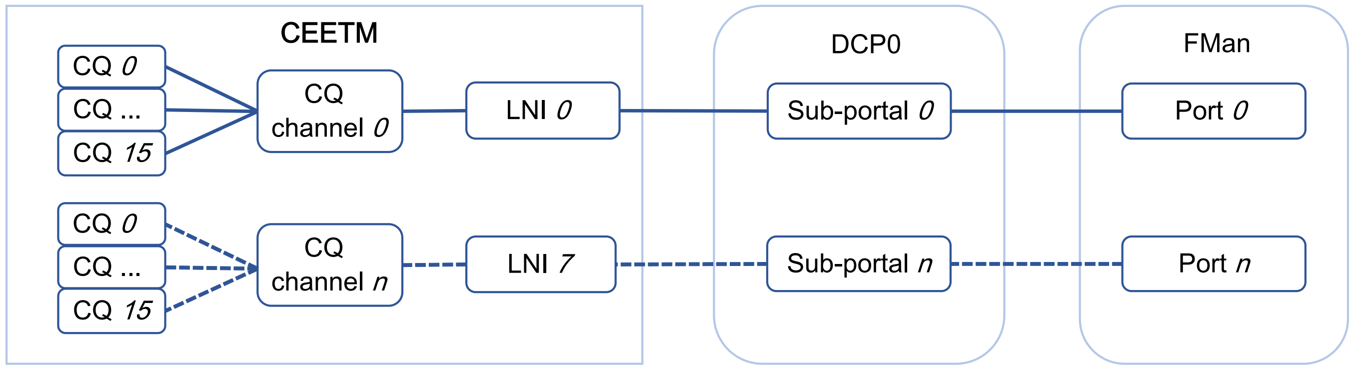


Figure 42. CEETM block

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*.

4.2.8.1.2.5.4.2.1.2 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 below:

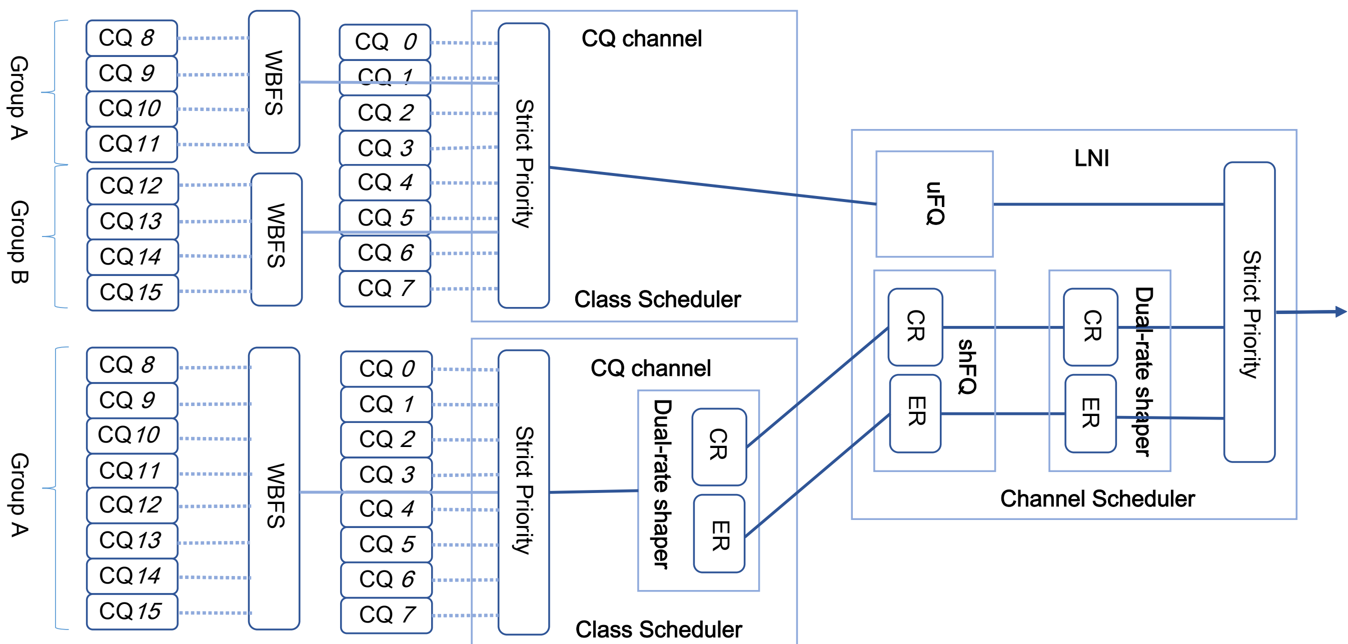


Figure 43. CEETM architecture

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.

4.2.8.1.2.5.4.2.1.3 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 below:

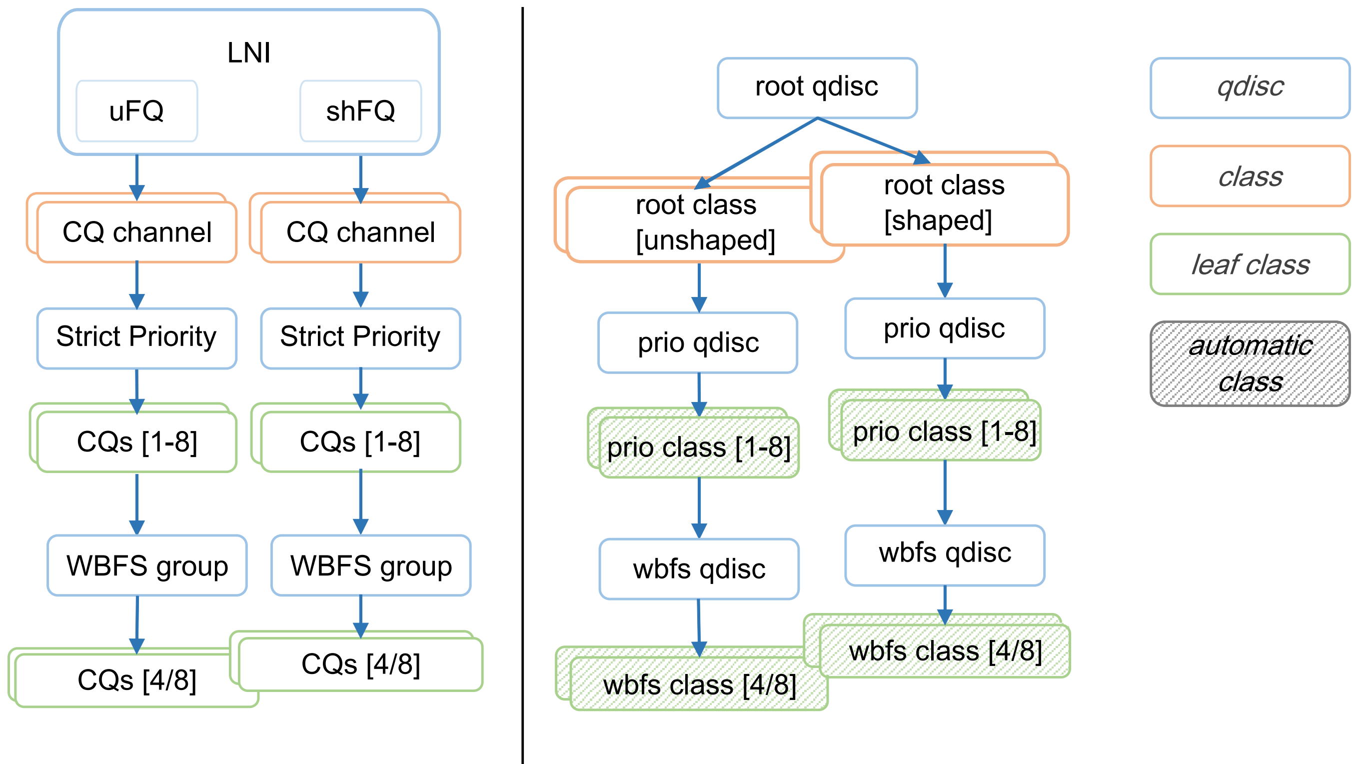


Figure 44. Comparison between CEETM and tc terminology

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 162 section.

4.2.8.1.2.5.4.2.2 User guide

4.2.8.1.2.5.4.2.2.1 Supported platforms

The CEETM block is present and configurable through the ceetm qdisc on the LS1043A/LS1046A platforms.

4.2.8.1.2.5.4.2.2.2 Getting started

1. Enable the networking QoS support in the kernel along with any classifiers or other features that might be needed, as well as the *ceetm* qdisc.

```
->  Networking support (NET [=y])
    -> Networking options
        -> QoS and/or fair queueing (NET_SCHED [=y])
            -> Universal 32bit comparisons w/ hashing (u32) (NET_CLS_U32 [=y])

-> 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])
```

2. Modify the Class Congestion State thresholds if necessary. The default values are chosen keeping in mind the following factors:

    - A threshold too large will buffer frames for a long time in the TX queues, when a small shaping rate is configured. This will cause buffer pool depletion or out of memory errors. This in turn will cause frame loss on RX.
    - A threshold too small will cause unnecessary frame loss by entering congestion too often.

```
-> 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])
                    -> CEETM egress congestion threshold on 1G ports
                       (FSL_DPAA_CEETM_CCS_THRESHOLD_1G [=0x000a0000])
                    -> CEETM egress congestion threshold on 10G ports
                       (FSL_DPAA_CEETM_CCS_THRESHOLD_10G [=0x00640000])
```

3. Build the ceetm app with the flexbuilder.

```
./flex-builder -c ceetm -a arm64
```

4.2.8.1.2.5.4.2.2.3 Limitations

- CEETM is supported on DPAA1 Private Ethernet interfaces only.
- CEETM isn't supported on top of Linux bonding interfaces.

4.2.8.1.2.5.4.2.2.4 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.

4.2.8.1.2.5.4.2.3 Examples
4.2.8.1.2.5.4.2.3.1 Rate limit two streams

Setup

In the following example a platform with CEETM support (LS1043ARDB - Client) is connected to another board (LS1046ARDB - Server) through a 1G link. The described setup is pictured in Figure 45. on page 163.

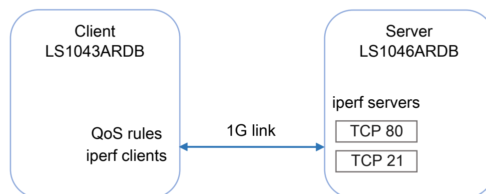


Figure 45. Rate example setup

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@ls1046ardb:~# iperf -s -p 21 &
root@ls1046ardb:~# iperf -s -p 80 &
```

PCDs are applied on both platforms in advance.

```
root@ls1046ardb:~# fmc -c /etc/fmc/config/private/ls1046ardb/RR_FFSSPPPH_1133_5559/config.xml -
p /etc/fmc/config/private/ls1046ardb/RR_FFSSPPPH_1133_5559/policy_ipv4.xml -a
root@ls1043ardb:~# fmc -c /etc/fmc/config/private/ls1043ardb/RR_FQPP_1455/config.xml -p /etc/fmc/
config/private/ls1043ardb/RR_FQPP_1455/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@ls1043ardb:~# arp -s <server IP address> <server HW address>
root@ls1046ardb:~# arp -s <client IP address> <client HW address>
```

After adding the qdiscs, the Client runs the iperf clients.

```
root@ls1043ardb:~# iperf -c <server IP address> -p 21 &
root@ls1043ardb:~# iperf -c <server IP address> -p 80 &
```

Execution

This example's corresponding qdisc and class hierarchy is pictured in Figure 46. on page 164.

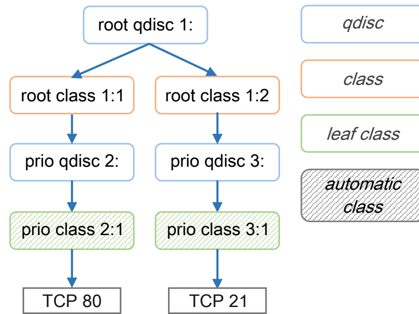


Figure 46. Rate example class hierarchy

Add a ceetm qdisc to the interface and configure the LNI's dual-rate shaper with a CR of 1Gbps.

```
root@ls1043ardb:~#  tc qdisc add dev fm1-mac3 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@ls1043ardb:~#  tc class add dev fm1-mac3 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@ls1043ardb:~#  tc class add dev fm1-mac3 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@ls1043ardb:~#  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 (marked by default as both CR and ER eligible).

```
root@ls1043ardb:~#  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@ls1043ardb:~#  tc filter add dev fm1-mac3 parent 1: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 1:1
root@ls1043ardb:~#  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 21 and lead them through the priority class of the second channel.

```
root@ls1043ardb:~#  tc filter add dev fm1-mac3 parent 1: prio 1 protocol ip u32 match ip dport 21
0xffff flowid 1:2
root@ls1043ardb:~#  tc filter add dev fm1-mac3 parent 3: prio 1 protocol ip u32 match ip dport 21
0xffff flowid 3:1
```

4.2.8.1.2.5.4.2.3.2 Prioritization of two streams

Setup

The same setup is used as for the rate limit example.

Execution

This example's corresponding qdisc and class hierarchy is pictured below:

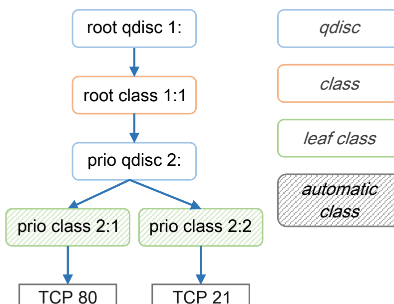


Figure 47. Prioritization example class hierarchy

Add a ceetm qdisc to the interface and configure the LNI's dual-rate shaper with a CR of 1Gbps.

```
root@ls1043ardb:~#  tc qdisc add dev fm1-mac3 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@ls1043ardb:~#  tc class add dev fm1-mac3 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@ls1043ardb:~#  tc qdisc add dev fm1-mac3 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@ls1043ardb:~#  tc filter add dev fm1-mac3 parent 1: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 1:1
root@ls1043ardb:~#  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 21 and lead them through the second (lowest) priority class of the channel.

```
root@ls1043ardb:~#  tc filter add dev fm1-mac3 parent 1: prio 1 protocol ip u32 match ip dport 8000
0xffff flowid 1:1
root@ls1043ardb:~#  tc filter add dev fm1-mac3 parent 2: prio 1 protocol ip u32 match ip dport 8000
0xffff flowid 2:2
```

4.2.8.1.2.5.4.2.3.3 Assigning weights to two streams

Setup

The same setup is used as for the rate limit example.

Execution

This example's corresponding qdisc and class hierarchy is pictured below:

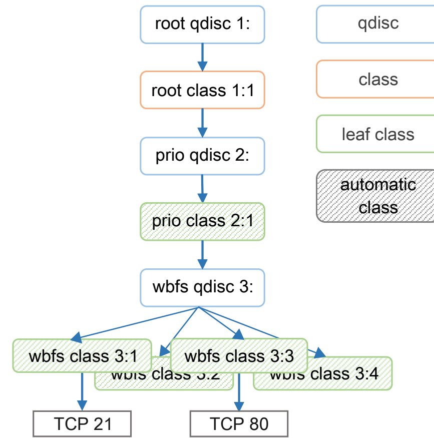


Figure 48. WBFS example class hierarchy

Add a ceetm qdisc to the interface and configure the LNI's dual-rate shaper with a CR of 1Gbps.

```
root@ls1043ardb:~#  tc qdisc add dev fm1-mac3 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@ls1043ardb:~#  tc class add dev fm1-mac3 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@ls1043ardb:~#  tc qdisc add dev fm1-mac3 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@ls1043ardb:~#  tc qdisc add dev fm1-mac3 parent 2:1 handle 3: ceetm type wbfs 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@ls1043ardb:~#  tc filter add dev fm1-mac3 parent 1: prio 1 protocol ip u32 match ip dport 21
0xffff flowid 1:1
root@ls1043ardb:~#  tc filter add dev fm1-mac3 parent 2: prio 1 protocol ip u32 match ip dport 21
0xffff flowid 2:1
root@ls1043ardb:~#  tc filter add dev fm1-mac3 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@ls1043ardb:~#  tc filter add dev fm1-mac3 parent 1: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 1:1
root@ls1043ardb:~#  tc filter add dev fm1-mac3 parent 2: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 2:1
root@ls1043ardb:~#  tc filter add dev fm1-mac3 parent 3: prio 1 protocol ip u32 match ip dport 80
0xffff flowid 3:3
```

4.2.8.1.2.5.4.2.3.4 Unshaped Fair Queuing of two streams

Setup

In the following example a platform with CEETM support (LS1043ARDB - Main) is connected to two other boards: a LS1043ARDB (Client) through a 10G link and a LS1046ARDB (Server) through a 1G link. The described setup is pictured below:

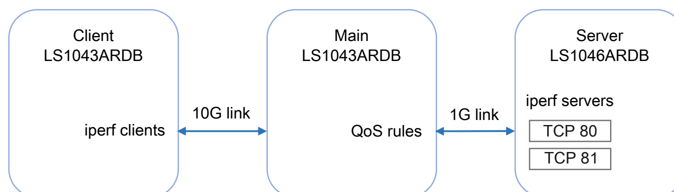


Figure 49. Unshaped Fair Queuing example setup

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@ls1046ardb:~# iperf -s -p 80 &
root@ls1046ardb:~# iperf -s -p 81 &
```

PCDs are applied on all platforms in advance.

```
root@ls1043ardb:~# fmc -c /etc/fmc/config/private/ls1043ardb/RR_FQPP_1455/config.xml -p /etc/fmc/
config/private/ls1043ardb/RR_FQPP_1455/policy_ipv4.xml -a
root@ls1046ardb:~# fmc -c /etc/fmc/config/private/ls1046ardb/RR_FFSSPPPH_1133_5559/config.xml -
p /etc/fmc/config/private/ls1046ardb/RR_FFSSPPPH_1133_5559/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@ls1046ardb:~# arp -s <main IP address> <main HW address>
# Main:
root@ls1043ardb:~# arp -s <client IP address> <client HW address>
root@ls1043ardb:~# arp -s <server IP address> <server HW address>
# Client:
root@ls1043ardb:~# 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@ls1043ardb:~# echo 1 > /proc/sys/net/ipv4/ip_forward
# Client:
root@ls1043ardb:~# route add -net <server network address> <server network mask> gw <main IP address>
# Server:
root@ls1046ardb:~# route add -net <client network address> <client network mask> gw <main IP address>
```

After adding the qdiscs, the Client runs the iperf clients.

```
root@ls1043ardb:~# iperf -c <server IP address> -p 80 &
root@ls1043ardb:~# iperf -c <server IP address> -p 81 &
```

Execution

This example's corresponding qdisc and class hierarchy is pictured in Figure 50. on page 168.

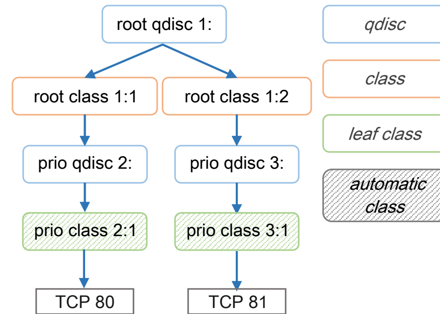


Figure 50. Unshaped Fair Queuing example class hierarchy

Add a ceetm qdisc to the interface and don't configure the LNI's dual-rate shaper.

```
root@ls1043ardb:~# 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@ls1043ardb:~# 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@ls1043ardb:~# 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@ls1043ardb:~# 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@ls1043ardb:~# 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@ls1043ardb:~#  tc filter add dev fm1-mac3 parent 1: prio 1 protocol ip u32 match ip dport 80 
0xffff flowid 1:1
root@ls1043ardb:~#  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@ls1043ardb:~#  tc filter add dev fm1-mac3 parent 1: prio 1 protocol ip u32 match ip dport 81 
0xffff flowid 1:2
root@ls1043ardb:~#  tc filter add dev fm1-mac3 parent 3: prio 1 protocol ip u32 match ip dport 81 
0xffff flowid 3:1
```

4.2.8.1.2.5.5 Debugging

This section describes the debugging capabilities of the DPAA1 Ethernet driver.

4.2.8.1.2.5.5.1 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 fm1-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
```

4.2.8.1.2.5.5.2 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:

```
0x1a00000 (FMan) +
  0xe8000 (MAC 5) +
    0x014 (Maximum frame length register) =
--------------------------------------
0x1ae8014
```

A memory print of the 0x1ae8014 address will display the maximum frame length configured for the fifth MAC device from the FMan on a LS1046A platform.

The entire memory map for all mapped registers of the DPAA1 hardware components ca be found in each platform's Reference Manual.

4.2.8.1.2.5.5.3 Sysfs support

To enable Sysfs in the Linux kernel one must set the **CONFIG_SYSFS** option in Kconfig. The DPAA1 Ethernet Driver exports a series of information in Sysfs such as the buffer pool IDs, the frame queue IDs used by the interface, and MAC registers and statistics, as shown in the following examples:

```

root@ls1046ardb:~# cat /sys/devices/platform/fsl,dpaa/fsl,dpaa:ethernet@2/net/fm1-mac3/bpids
32

root@ls1046ardb:~# cat /sys/devices/platform/fsl,dpaa/fsl,dpaa:ethernet@2/net/fm1-mac3/fqids
Rx error: 259
Rx default: 260
Rx PCD: 14592 - 14719
Rx PCD High Priority: 80128 - 80255
Tx confirmation (mq): 261 - 324
Tx error: 325
Tx default confirmation: 326
Tx: 327 - 390

root@ls1046ardb:~# cat /sys/devices/platform/fsl,dpaa/fsl,dpaa:ethernet@2/net/fm1-mac3/mac_regs
-------
FM MAC - MEMAC - 2 (0xFFFF8000801D6000)
----------------------------------------

0xFFFF8000801D6008: 0x00020840          command_config
0xFFFF8000801D600C: 0x38ca0568          mac_addr0.mac_addr_l
0xFFFF8000801D6010: 0x0000de30          mac_addr0.mac_addr_u
[...]
```

```
root@ls1046ardb:~# cat /sys/devices/platform/fsl,dpaa/fsl,dpaa:ethernet@2/net/fm1-mac3/mac_rx_stats
-------
FM MAC - MEMAC - 2 Rx stats (0xFFFF8000801D6000)
---------------------------------------

0xFFFF8000801D6100: 0x00000000          reoct_l
0xFFFF8000801D6104: 0x00000000          reoct_u
[...]

root@ls1046ardb:~# cat /sys/devices/platform/fsl,dpaa/fsl,dpaa:ethernet@2/net/fm1-mac3/mac_tx_stats
-------
FM MAC - MEMAC - 2 Tx stats (0xFFFF8000801D6000)
---------------------------------------

0xFFFF8000801D6200: 0x00000000          teoct_l
0xFFFF8000801D6204: 0x00000000          teoct_u
[...]
```

4.2.8.1.2.5.6 Frequently Asked Questions

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.

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 DPAA1-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 *very* brittle design, and hard to maintain, too.

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?

   It is not an intended use-case for private interfaces.

4. What format must an ingress frame have, from the standpoint of the DPAA1-Ethernet driver and the Linux kernel stack?

   The DPAA1-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. What channel are the FQs assigned to?

   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 DPAA1 Ethernet devices, and even with other DPAA1 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.

6. What work queue are the FQs assigned to?

   - Tx Confirmation FQs go to WQ1
   - Rx Error and Tx Error FQs go to WQ2
   - Rx Default, Tx and PCD FQs go to WQ3

7. 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/ls1043ardb/RR_FQPP_1455/config.xml -p /etc/fmc/config/private/
ls1043ardb/RR_FQPP_1455/policy_ipv4.xml -a
```

Note that `/etc/fmc/config/private/ls1043ardb/RR_FQPP_1455/policy_ipv4.xml` is a soft link to `/etc/fmc/config/private/common/policy_ipv4.xml`.

```
(2) fmc -c /etc/fmc/config/private/ls1043ardb/RR_FQPP_1455/config.xml -p /etc/fmc/config/private/
ls1043ardb/RR_FQPP_1455/policy_ipv6.xml -a
```

Note that `/etc/fmc/config/private/ls1043ardb/RR_FQPP_1455/policy_ipv6.xml` is a soft link to `/etc/fmc/config/private/common/policy_ipv6.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
```

4.2.8.1.2.5.7 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.
- Scatter Gather frames are not supported on LS1043A due to the FMan A010022 errata.

4.2.8.1.2.6 Upstream Ethernet Driver

The DPAA 1.x Upstream Ethernet driver variant has been actively maintained in the Linux kernel community since v4.10. Most features and fixes have been back ported to the kernel versions of this current LSDK release.

An overview of the driver, along with its main features and configuration options, is written in the Linux kernel's source tree in the documentation section at *Documentation/networking/dpaa.txt*.

Configuration

The Upstream and Private Ethernet driver variants are independent from one another and are built separately. The Private driver variant is enabled by default by the LSDK. If you wish to build the Upstream driver variant instead, enable the following build options:

```
CONFIG_FSL_DPAA=y
CONFIG_FSL_FMAN=y
CONFIG_FSL_DPAA_ETH=y
CONFIG_FSL_XGMAC_MDIO=y
```

Device Trees

The Upstream and Private Ethernet drivers use different Device Tree Source files. The LSDK enables the device trees associated with the Private driver by default. These end with the *-sdk* flag. The device trees that are used by the Upstream driver variant do not have a flag at the end. For example:

```
fsl-ls1046a-frwy.dts - used by the Upstream Ethernet driver
fsl-ls1046a-frwy-sdk.dts - used by the Private Ethernet driver
```

After building the kernel with the Upstream Ethernet driver enabled, also compile the correct Device Tree Blob for your platform. For example:

```
make freescale/fsl-ls1046a-frwy.dtb - build the DTB for the Upstream Ethernet driver
make freescale/fsl-ls1046a-frwy-sdk.dtb - build the DTB for the Private Ethernet driver
```

Limitations

A workaround for the LS1046A FMan A010022 errata is integrated into the Upstream Ethernet driver. Egress Scatter Gather frames cannot be used on this platform.

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

4.2.8.1.3.1 QMan/BMan Drivers Introduction

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.

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 pthreads 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 "QMan BMan API Reference on page 180").

DPAA1 allocator

The DPAA1 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 DPAA1 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. enqueuing 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 operatoins to complete. Eg. after starting a volatile dequeue, wait for all dequeues to complete. This is enabled unconditionally on linux.

Table continues on the next page...

Table continued from the previous page...

Common Kernel Configure Options	Description
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 noticable 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 enqueuing/ dequeuing 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

Table continues on the next page...

Table continued from the previous page...

BMan Kernel Configure Options	Description
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 `0x2000000-0x221fffff`, 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 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 discontiguous 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 discontiguous 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 DPAA1 QBMan drivers, all the files are located in `drivers/soc/fsl/qbman` directory

USDPAA

Source Files	Description
include/usdpaa/fsl_qman.h	The QMan driver APIs
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_*.*	The QMan driver
src/qbman/bman_*.*	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.

4.2.8.1.3.2 QMan BMan API Reference

4.2.8.1.3.2.1 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.

The software described in this document is targeted to the Linux kernel and Linux user-space (USDPAA) 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 228for more details.

4.2.8.1.3.2.2 Buffer Manager

4.2.8.1.3.2.2.1 Buffer Manager (BMan) Overview

Function

The QorIQ Buffer Manager (BMan) SoC block manages pools of buffers for use by software and hardware in the “Datapath” architecture.

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.

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.

4.2.8.1.3.2.2.2 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.

4.2.8.1.3.2.2.2.1 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.

4.2.8.1.3.2.2.2.1.1 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 FBPR entries or 2,097,152 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.

4.2.8.1.3.2.2.2.1.2 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 authorise 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.

4.2.8.1.3.2.2.2.2 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>;
                };
                [...]
};
```

4.2.8.1.3.2.2.2.2.1 Buffer Pool ID

The BMan device supports hardware managed buffer pools. Specifications and valid ID ranges vary between SoC's. Refer to the appropriate SoC Reference Manual for more information. The example above 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.

4.2.8.1.3.2.2.2.2.2 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].

4.2.8.1.3.2.2.2.2.3 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.

4.2.8.1.3.2.2.2.3 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.

4.2.8.1.3.2.2.2.3.1 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 USDPAA 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.

4.2.8.1.3.2.2.2.3.2 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.

4.2.8.1.3.2.3 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 184.

4.2.8.1.3.2.3.1 BMan High-Level Portal Interface

4.2.8.1.3.2.3.1.1 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 resouce is available. In any case where sleeping is an option, the caller can also specify whether the sleep should be interruptible.

4.2.8.1.3.2.3.1.2 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 184. 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);
```

4.2.8.1.3.2.3.1.2.1 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, refer to Portal sharing on page 185. These APIs will not 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);
```

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

If portal-sharing is in effect, refer to Portal sharing on page 185. These APIs will not 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
 * bmna_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 very 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);
```

4.2.8.1.3.2.3.1.2.3 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);
```

4.2.8.1.3.2.3.1.2.4 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);
```

4.2.8.1.3.2.3.1.3 Pool Management

To work with BMan buffer pools, a pool object must be created. As explained in Depletion State on page 190, 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 thresolds 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 occured.
 * 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);
```

4.2.8.1.3.2.3.1.4 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 issed 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);
```

4.2.8.1.3.2.3.1.5 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 187, 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) [...]
```

4.2.8.1.3.2.4 Queue Manager

4.2.8.1.3.2.4.1 QMan Overview

4.2.8.1.3.2.4.1.1 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.

4.2.8.1.3.2.4.1.2 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.

4.2.8.1.3.2.4.1.3 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.

4.2.8.1.3.2.4.1.4 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.

4.2.8.1.3.2.4.1.5 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 dequeueing 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.

4.2.8.1.3.2.4.1.6 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.

4.2.8.1.3.2.4.1.7 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).

4.2.8.1.3.2.4.1.8 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.

4.2.8.1.3.2.4.1.9 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".

4.2.8.1.3.2.4.1.10 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.

4.2.8.1.3.2.4.1.10.1 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 194).

4.2.8.1.3.2.4.1.10.2 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, dequeueing a FQ from that selected WQ, and then dequeuing a FD from that FQ.

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

4.2.8.1.3.2.4.1.10.3 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.

4.2.8.1.3.2.4.1.10.4 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 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.

4.2.8.1.3.2.4.1.10.5 Stashing to Processor Cache

When dequeueing frame queues and publishing entries in DQRR, QMan provides stashing features that involve prepositioning data in the 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 DLIODN 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.

4.2.8.1.3.2.4.1.11 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 dequeus 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 occured.
  - **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.

4.2.8.1.3.2.4.1.12 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 dequues 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.

4.2.8.1.3.2.4.1.12.1 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".

4.2.8.1.3.2.4.1.12.2 Parking Scheduled FQs

As noted above in Dequeue Atomicity on page 195, 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. 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.

4.2.8.1.3.2.4.1.12.3 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 196. The above "hold active" mechanisms are to prevent frames from getting out of order in the first place.

4.2.8.1.3.2.4.1.13 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.

4.2.8.1.3.2.4.1.14 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 desination 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. Ie. 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 enqueuing 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, ie. 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 207, 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 203, 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.

4.2.8.1.3.2.4.2 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.

4.2.8.1.3.2.4.2.1 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.

4.2.8.1.3.2.4.2.1.1 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 182.

4.2.8.1.3.2.4.2.1.2 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 182.

4.2.8.1.3.2.4.2.1.3 Logical I/O Device Number (QMan)

This property is the same as described in Logical I/O Device Number (BMan) on page 182, but for use by QMan when accessing FQD and PFDR memory (rather than BMan's FBPR memory).

4.2.8.1.3.2.4.2.2 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 191. 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>;
};
```

4.2.8.1.3.2.4.2.2.1 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>;
```

4.2.8.1.3.2.4.2.3 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 191 and Portal Sub-Interfaces on page 192. 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 197, the other QMan-specific portal properties are described below.

4.2.8.1.3.2.4.2.3.1 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>;
```

4.2.8.1.3.2.4.2.3.2 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 193. 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>;
```

4.2.8.1.3.2.4.2.3.3 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 185 for an explanation of this behaviour in the BMan documentation, the QMan behaviour is identical.

4.2.8.1.3.2.4.2.3.4 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 185 for an explanation of this behavior in the BMan documentation. The QMan behavior is identical.

4.2.8.1.3.2.5 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 197, and described in Portals on page 191 and Portal Sub-Interfaces on page 192.

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.

4.2.8.1.3.2.5.1 QMan High-Level Portal Interface

4.2.8.1.3.2.5.1.1 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 resouce 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.

4.2.8.1.3.2.5.1.2 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
};
```

4.2.8.1.3.2.5.1.3 Portal management (QMan)

The portal management API provides qman_affine_cpus(), which returns a mask that indicates which CPUs have auto-initialiazed portals associated with them. See QMan portal device-tree node on page 197. 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);
```

4.2.8.1.3.2.5.1.3.1 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 185), 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);
```

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

If portal-sharing is in effect (see Portal sharing on page 185), 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);
```

4.2.8.1.3.2.5.1.3.3 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);
```

4.2.8.1.3.2.5.1.3.4 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);
```

4.2.8.1.3.2.5.1.3.5 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);
```

4.2.8.1.3.2.5.1.3.6 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);
```

4.2.8.1.3.2.5.1.4 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 enqueuing 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);
```

4.2.8.1.3.2.5.1.4.1 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);
```

4.2.8.1.3.2.5.1.4.2 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);
```

4.2.8.1.3.2.5.1.4.3 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);
```

4.2.8.1.3.2.5.1.4.4 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);
```

4.2.8.1.3.2.5.1.4.5 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);
```

4.2.8.1.3.2.5.1.4.6 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);
```

4.2.8.1.3.2.5.1.4.7 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)
```

4.2.8.1.3.2.5.1.4.8 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);
```

4.2.8.1.3.2.5.1.5 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);
```

4.2.8.1.3.2.5.1.6 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 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 to 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);
```

4.2.8.1.3.2.5.1.7 DCA Mode

As described in Order Preservation & Discrete Consumption Acknowledgement on page 195, 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 207, 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);
```

4.2.8.1.3.2.5.1.8 Congestion Management Records

QMan supports a fixed number[4] 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 appropraite 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.

[4] 256 for P4080/P5020/P3041

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;
          };
} __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);
```

4.2.8.1.3.2.5.1.9 Zero-Configuration Messaging

As described in Overview (QMan) on page 199, 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 204).

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);
```

4.2.8.1.3.2.5.1.10 FQ allocation

4.2.8.1.3.2.5.1.10.1 Ad-hoc FQ allocator

As described in Seeding Buffer Pools on page 183>, BMan buffer pool ID zero is currently reserved for use as an ad-hoc FQ allocator. As seen in Frame queue management on page 203, 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).

4.2.8.1.3.2.5.1.10.2 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);
```

4.2.8.1.3.2.5.1.10.3 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.

4.2.8.1.3.2.5.1.11 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;
}
```

4.2.8.1.3.2.6 Sysfs and debugfs QMan/BMan interfaces

The following section describes the QMan and BMan interfaces available via sysfs and debugfs.

NOTE

Check the device-tree of each SoC to determine the interfaces available. For more information, see the Reference Manual for the SoC, and/or examine the sysfs filesystem at run-time.

4.2.8.1.3.2.6.1 QMan sysfs

4.2.8.1.3.2.6.1.1 /sys/devices/platform/soc/1880000.qman/

Description:

This directory contains a snapshot of the internal state of the qman device.

4.2.8.1.3.2.6.1.2 /sys/devices/ffe000000.soc/ffe318000.qman/error_capture

Description:

This directory contains a snapshot of error related qman attributes.

4.2.8.1.3.2.6.1.3 /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.

4.2.8.1.3.2.6.1.4 /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 read-only.

4.2.8.1.3.2.6.1.5 /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

4.2.8.1.3.2.6.1.6 /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.

4.2.8.1.3.2.6.1.7 /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.

4.2.8.1.3.2.6.1.8 /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.

4.2.8.1.3.2.6.1.9 /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

4.2.8.1.3.2.6.1.10 /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

4.2.8.1.3.2.6.2 BMan sysfs

4.2.8.1.3.2.6.2.1 /sys/devices/ffe000000.soc/ffe31a000.bman

Description:

This directory contains a snapshot of the internal state of the BMan device.

4.2.8.1.3.2.6.2.2 /sys/devices/ffe000000.soc/ffe31a000.bman/error_capture

Description:

This directory contains a snapshot of error related BMan attributes.

4.2.8.1.3.2.6.2.3 /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.

4.2.8.1.3.2.6.2.4 /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.

4.2.8.1.3.2.6.2.5 /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

4.2.8.1.3.2.6.2.6 /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.

4.2.8.1.3.2.6.3 QMan debugfs

4.2.8.1.3.2.6.3.1 /sys/kernel/debug/qman

Description:

This directory contains various QMan device debugging attributes.

4.2.8.1.3.2.6.3.2 /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_bcnt: 0

a_bcnt: 0

4.2.8.1.3.2.6.3.3 /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.

4.2.8.1.3.2.6.3.4 /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

4.2.8.1.3.2.6.3.5 /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

orp_nesn: 0

orp_ea_hseq: 0

orp_ea_tseq: 0

orp_ea_hptr: 0x0

orp_ea_tptr: 0x0

pfdr_hptr: 0x0

pfdr_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

4.2.8.1.3.2.6.3.6 /sys/kernel/debug/qman/query_cq_fields

Description:

Query all the fileds 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_fq_fields

Query CQ Fields Result cqid 0x4 on DCP 1

ccgid: 4

state: 0

pfdr_hptr: 0

pfdr_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

4.2.8.1.3.2.6.3.7 /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

4.2.8.1.3.2.6.3.8 /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

4.2.8.1.3.2.6.3.9 /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
```

4.2.8.1.3.2.6.3.10 /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
```

4.2.8.1.3.2.6.3.11 /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
```

4.2.8.1.3.2.6.3.12 /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
```

4.2.8.1.3.2.6.3.13 /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 seperated 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
```

4.2.8.1.3.2.6.3.14 /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 seperated 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
```

4.2.8.1.3.2.6.3.15 /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 seperated 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
```

4.2.8.1.3.2.6.3.16 /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 seperated 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
```

4.2.8.1.3.2.6.3.17 /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
```

4.2.8.1.3.2.6.3.18 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
```

4.2.8.1.3.2.6.3.19 /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
```

4.2.8.1.3.2.6.3.20 /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
```

4.2.8.1.3.2.6.3.21 /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
```

4.2.8.1.3.2.6.3.22 /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 cccsrmempeek
QMan register offset = 0xbf8
value = 0x0a010101
```

4.2.8.1.3.2.6.3.23 /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

4.2.8.1.3.2.6.4 BMan debugfs

4.2.8.1.3.2.6.4.1 /sys/kernel/debug/bman

Description:

This directory contains various BMan device debugging attributes.

4.2.8.1.3.2.6.4.2 /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

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

4.2.8.1.3.2.7 Error handling and reporting

This chapter describes the QMan and BMan error handling and reporting.

4.2.8.1.3.2.7.1 Handling and Reporting

The QMan and BMan error interrupt sevices 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 occurance 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).

4.2.8.1.3.2.8 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.

4.2.8.1.3.2.8.1 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 186 (for BMan) and Processing non-interrupt-driven portal duties (QMan) on page 201 (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.

4.2.8.1.3.2.8.2 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. 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().

4.2.8.1.3.2.8.3 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.

4.2.8.1.4 Configuring DPAA1 Frame Queues

4.2.8.1.4.1 Introduction

Describes configurations of Queue Manager (QMan) Frame Queues (FQs) associated with Frame Manager (FMan) network interfaces for the QorIQ Data Path Acceleration Architecture (DPAA1). 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 DPAA1 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.

DPAA1 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 network and I/O interfaces through which that data arrives and leaves
2. 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 netwok 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 DPAA1 technology
- The configuration information provided in this document assumes a top bin platform frequency.

4.2.8.1.4.2 FMan Network interface Frame Queue Configuration

Configuring the QMan Frame Queues (FQs) associated with the FMan network interfaces for QorIQ DPAA1.

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.

4.2.8.1.4.3 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 DPAA1.

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 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.

4.2.8.1.4.4 Ingress FQs common configuration guidelines

Guidelines and examples for configuring ingress Frame Queues (FQs) in the QorIQ DPAA1 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.

4.2.8.1.4.5 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.

4.2.8.1.4.6 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.

4.2.8.1.4.7 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 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.
  - 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.

4.2.8.1.4.8 FMan network interface egress FQs configuration

Configuration guidelines for egress Frame Queues (FQs) for QorIQ DPAA1

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.

.

4.2.8.1.4.9 Accelerator Frame Queue Configuration

Configurations for Frame Queues (FQs) used to communicate with accelerators for QorIQ DPAA1 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.

4.2.8.1.4.10 DPAA1 Frame Queue Configuration Guideline Summary

Summary of Configurations for Frame Queue (FQ) communication with accelerators for QorIQ DPAA1

Four tables comprise this summary:

- Global Configuration settings
- Network interface ingress FQ guidelines
- Network interface egress FQ guidelines
- Accelerator FQ guidelines

Table 34. Global Configuration Settings Summary

Parameter or subject	Guideline
FQD stashing	Recommend QMan explicitly stash FQDs: • 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: • 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: • 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 35. 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.	• 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.

Table continues on the next page...

Table 35. Network Interface Ingress FQs Guidelines Summary (continued)

Parameter or subject	Guideline
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
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.	• Prefer_in_Cache: 1 • CPC Stash Enable: 1 • ORP_Enable: 0 • Avoid_Blocking: — 0 if static distribution or dynamic load balancing with order preservation. — 1 if dynamic load balancing with order restoration. • Hold_Active — 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.	• 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 36. 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.	• 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 37. 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.	• 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 .

4.2.8.1.5 Frame Manager

4.2.8.1.5.1 Frame Manager Linux Driver User Guide

4.2.8.1.5.1.1 Introduction

This part is describing the Linux implementation of the driver for the Frame Manager, or FMD.

The Linux FMD implements a set of standard Linux character devices that rely on underlying OS-agnostic FMan drivers to do the actual communication with the hardware. The figure below describes this best:

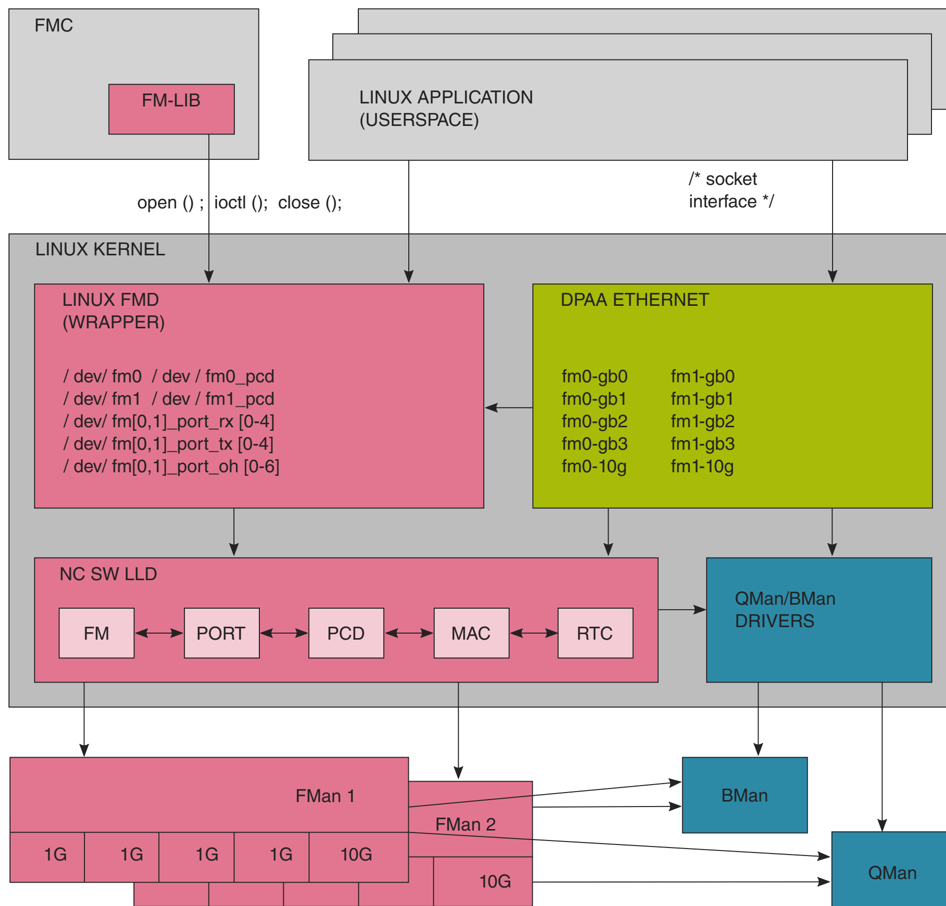


Figure 51. FMan-centric view of relationships between DPAA software and hardware blocks in the Linux environment.

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 underlaying low-level FMD and with each other, their feature sets do not overlap. The DPAA1 Ethernet driver is described in the Linux Ethernet on page 140 section.

4.2.8.1.5.1.2 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 devicesare created and initialized 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 (RCW). See the SoC's Reference Manual for more details.

NOTE

The assumption for the remainder of this section is that the device tree and the RCW are immutable.

Depending on the SoC and `RCW/.dts` configuration, only certain devices are available . The mapping of the devices to the physical ports is given by the following table:

Table 38. Mapping of Linux devices to low-level port IDs.

Linux Device	Low-Level ID	Identification
`/dev/fm0-port-rx0` `/dev/fm0-port-tx0`	0	1st FMan's 1st 1GbE Receive, Transmit
`/dev/fm0-port-rx1` `/dev/fm0-port-tx1`	1	1st FMan's 2nd GbE Receive, Transmit
`/dev/fm0-port-rx2` `/dev/fm0-port-tx2`	2	1st FMan's 3rd GbE Receive, Transmit
`/dev/fm0-port-rx3` `/dev/fm0-port-tx3`	3	1st FMan's 4th GbE Receive, Transmit
`/dev/fm0-port-rx4` `/dev/fm0-port-tx4`	4	1st FMan's 5th GbE Receive, Transmit
`/dev/fm0-port-rx5` `/dev/fm0-port-tx5`	5	1st FMan's 6th GbE Receive, Transmit
`/dev/fm0-port-rx6` `/dev/fm0-port-tx6`	6	1st FMan's 1st 10Gb Receive, Transmit
`/dev/fm0-port-rx7` `/dev/fm0-port-tx7`	7	1st FMan's 2nd 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

Table continues on the next page...

Table 38. Mapping of Linux devices to low-level port IDs. (continued)

Linux Device	Low-Level ID	Identification
`/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	2nd FMan's 5th 1GbE Receive, Transmit
`/dev/fm1-port-rx5` `/dev/fm1-port-tx5`	5	2nd FMan's 10Gb Receive, Transmit
`/dev/fm1-port-rx6` `/dev/fm1-port-tx6`	6	2nd FMan's 1st 10Gb Receive, Transmit
`/dev/fm1-port-rx7` `/dev/fm1-port-tx7`	7	2nd FMan's 2nd 10Gb Receive, Transmit
N/A	0	2nd 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
`/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 wiht the following information in order to succeesfully point to the desired port:

- FMan ID: 0 or 1 for FMan1 or FMan2, 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]-mac[1-10]` (e.g. `fm1-mac1`, `fm2-mac3`) scheme, which provides 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-mac1`, 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.

4.2.8.1.5.1.3 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 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.

4.2.8.1.5.1.4 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.

4.2.8.1.5.1.4.1 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 39. 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.
`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.

Table continues on the next page...

Table 39. IOCTLs for the FMan Device (continued)

IOCTL	LLD Mapping	Brief
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.

4.2.8.1.5.1.4.2 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.

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 ioctls for this device are mapped to the similarly-sounding FM_PCD_*() LLD APIs:

Table 40. 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_PrsLoadSw()	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.

Table continues on the next page...

Table 40. IOCTL List for the PCD Device (continued)

IOCTL	LLD Mapping	Brief
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".
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 `modify` should be TRUE. If the routine is called for a valid scheme and `modify` 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.

Table continues on the next page...

Table 40. IOCTL List for the PCD Device (continued)

IOCTL	LLD Mapping	Brief
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. `p_node_id` 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 FM_PCD_CcBuildTree()).
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 FM_PCD_MatchTableSet()).
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 FM_PCD_MatchTableSet()).
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.

Table continues on the next page...

Table 40. IOCTL List for the PCD Device (continued)

IOCTL	LLD Mapping	Brief
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.
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 41. Side-by-side comparison of IOCTL and LLD types

IOCTL Types	LLD Types

`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;`
	`typedef struct t_FmPcdCcNodeParams {`
	`    t_FmPcdExtractEntry             extractCcParams;`
	`    t_KeysParams                    keysParams;`
	
	`} t_FmPcdCcNodeParams;`

Table continues on the next page...

Table 41. Side-by-side comparison of IOCTL and LLD types (continued)

<table>
<tr><th>IOCTL Types</th><th>LLD Types</th></tr>
<tr><td>

```
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;
```

</td><td>

```
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;
```

</td></tr>
<tr><td>

```
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;
```

</td><td>

```
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;
```

</td></tr>
</table>

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. 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.

4.2.8.1.5.1.4.3 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 DPAA1 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=[0-7] represents the physical port ID (0 corresponding to the first 1 Gb port, and 6 to the first 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, LS1043A has one FMan with 6 x 1Gb ports and one 10Gb port, while LS1046A has one FMan with 6 x 1Gb and 2 x 10Gb ports. A side-by-side comparison of the corresponding port devices is given in the following table:

Table 42. Side-by-side comparison of port devices for LS1043 and LS1046

LS1043A	LS1046A

For the Receive side:
`/dev/fm0-port-rx0`
`/dev/fm0-port-rx1`
`/dev/fm0-port-rx2`
`/dev/fm0-port-rx4`
`/dev/fm0-port-rx5`
`/dev/fm0-port-rx6`
	For the Receive side:
	`/dev/fm0-port-rx0`
	`/dev/fm0-port-rx1`
	`/dev/fm0-port-rx2`
	`/dev/fm0-port-rx3`
	`/dev/fm0-port-rx4`
	`/dev/fm0-port-rx5`
	`/dev/fm0-port-rx6`
	`/dev/fm0-port-rx7`

For the Transmit side:
`/dev/fm0-port-tx0`
`/dev/fm0-port-tx1`
`/dev/fm0-port-tx2`
`/dev/fm0-port-tx3`
`/dev/fm0-port-tx4`
`/dev/fm0-port-tx5`
`/dev/fm0-port-tx6`
	For the Transmit side:
	`/dev/fm0-port-tx0`
	`/dev/fm0-port-tx1`
	`/dev/fm0-port-tx2`
	`/dev/fm0-port-tx3`
	`/dev/fm0-port-tx4`
	`/dev/fm0-port-tx5`
	`/dev/fm0-port-tx6`
	`/dev/fm0-port-tx7`

For Offline Parsing:
`/dev/fm0-port-oh0`
`/dev/fm0-port-oh1`
`/dev/fm0-port-oh2`
`/dev/fm0-port-oh3`
`/dev/fm0-port-oh4`
`/dev/fm0-port-oh5`
	For Offline Parsing:
	`/dev/fm0-port-oh0`
	`/dev/fm0-port-oh1`
	`/dev/fm0-port-oh2`
	`/dev/fm0-port-oh3`
	`/dev/fm0-port-oh4`
	`/dev/fm0-port-oh5`

The table below summarizes the IOCTLs available for the port device.

Table 43. 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.
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 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()).

Table continues on the next page...

Table 43. IOCTLs of the Port Device (continued)

IOCTLS	LLD Mapping	Brief
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()).
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_GRPS	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_GRPS	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 43. 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.

4.2.8.1.5.2 Frame Manager Driver User's Guide

4.2.8.1.5.2.1 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. 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 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.

4.2.8.1.5.2.2 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-Classifier (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.

4.2.8.1.5.2.3 Frame Manager Driver Components

The FMan driver contains following low-level modules, as shown in this figure.

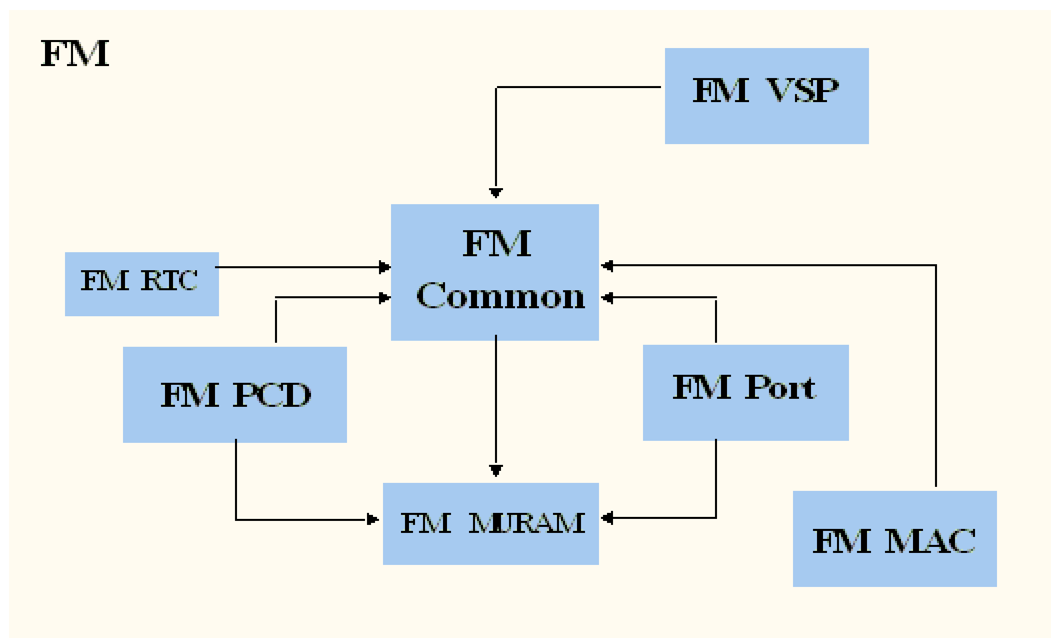


Figure 52. FMan Driver Modules (from a partition point of view)

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-Classifier-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 MAC controllers.

  This module can be run by each core or partition independently.

4.2.8.1.5.2.4 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.

NOTE

In this environment:

- All FMan driver modules are available and should be initialized by the user (unless if it is unnecessary for the user operation; for example, if PCD is not needed so it may not be called).
- The FMan driver modules have the full functionality of the hardware.
- Each module has full access to its hardware registers (i.e. each module will access its registers directly).

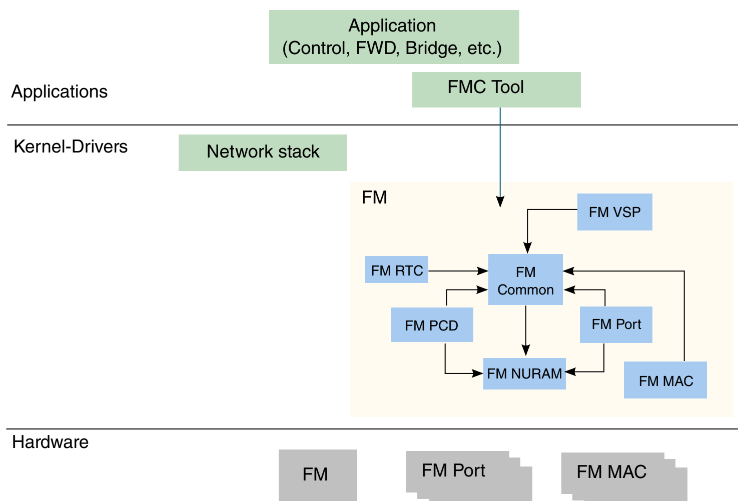


Figure 53. Single-Partition FM Building Blocks

4.2.8.1.5.2.4.1 Multicore Approach

The driver supports the Symmetric Multi-Processing (SMP) opperation method.

4.2.8.1.5.2.4.1.1 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.

4.2.8.1.5.2.5 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. in that stage, user should configure and intialize everything that is needed for the operation of a port outside the fman; e.g. buffer-pools, frame-queues, etc.
    c. Port Enablement
    d. MAC Enablement
    e. 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 "vitualized" into several software entities (using some classification to ditribute the traffic), user should configure and initialize the relevant buffer-pools and frame-queues.
8. 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)
9. 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).

10. FMan runtime routines
11. FMan Free sequence - in reverse order from initialization

4.2.8.1.5.2.6 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 spany of the specific modules such as PCD, RTC, MURAM, MAC etc.

4.2.8.1.5.2.6.1 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 enqueuing and dequeuing new frames to/from the multicore system.

4.2.8.1.5.2.6.1.1 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.

4.2.8.1.5.2.6.2 How to use the Global FMan Driver?

The following sections provide practical information for using the software drivers.

4.2.8.1.5.2.6.2.1 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

4.2.8.1.5.2.6.2.2 Global FMan Driver Sequence

- FMan config routine
- [Optional] FMan advance configuration routines
- FMan Init routine
- FMan runtime routines
- FMan free routine

4.2.8.1.5.2.6.2.3 Global FMan Driver Functional Description

The following sections describe main driver functionalities and their usage.

4.2.8.1.5.2.6.2.3.1 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.

4.2.8.1.5.2.6.2.3.2 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.

4.2.8.1.5.2.6.2.3.3 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.

4.2.8.1.5.2.6.2.3.4 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.

4.2.8.1.5.2.7 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.

4.2.8.1.5.2.7.1 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) 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.

- Policer-The Policer 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.

4.2.8.1.5.2.7.1.1 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.

4.2.8.1.5.2.7.1.1.1 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 Policer) 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 policer.
- If FMan-Controller is used, it will follow the Keygen. It may not be activated if Keygen is not used.
- Policer 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):

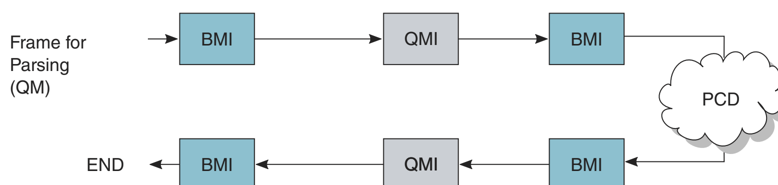


Figure 54. Default Rx Flow

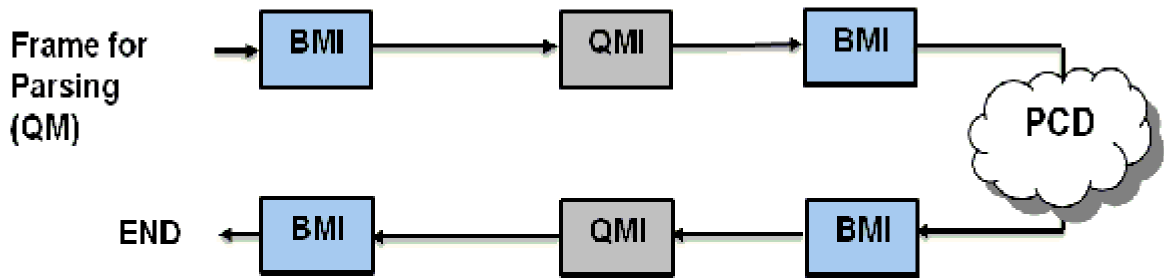


Figure 55. Default Offline Parsing Flow

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 54. on page 261, 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.

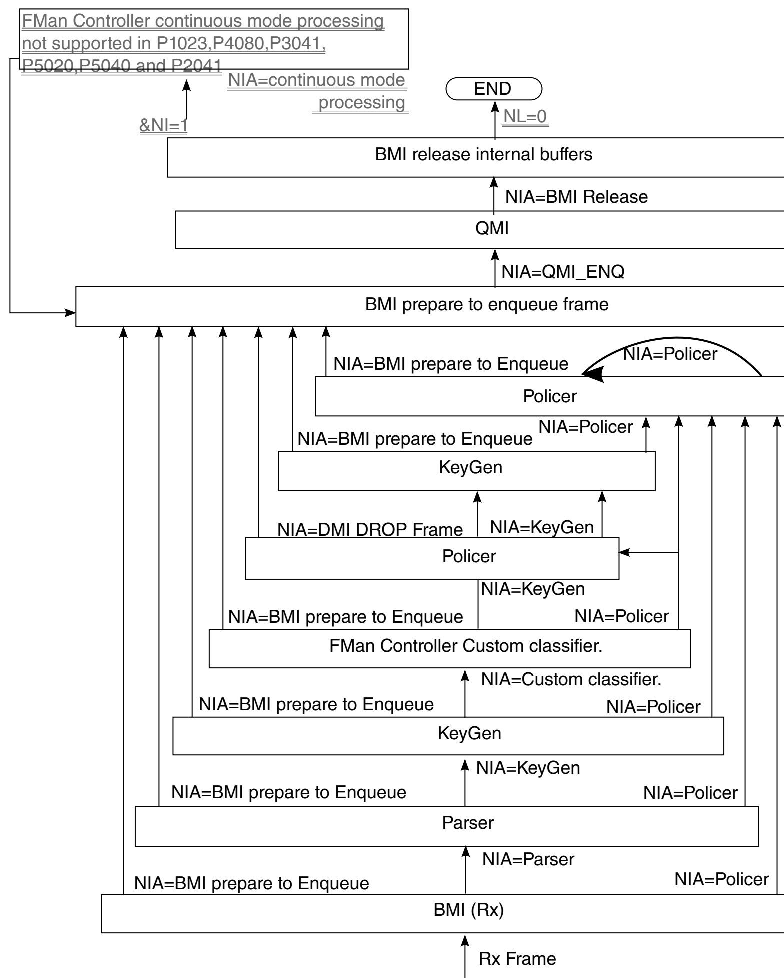


Figure 56. Available flows

4.2.8.1.5.2.7.1.1.2 Global FMan PCD Module

The FMan PCD driver deals with the configuration initialization and runtime setting of the PCD resources. The actual use of these resources is in fact activated only when an FMan-Port is enabled and is bound to the initialized PCD resources. In this chapter we will only deal with the initialization and organization of those resources.

The PCD driver is constructed by a global FMan-PCD module that must be initialized first, and a set of optional PCD resources that can be initialized at run-time. The FMan-PCD module is responsible for enabling the different engines, loading SW parser if required, registering PCD interrupts and other general configuration.

4.2.8.1.5.2.7.1.1.3 Global FMan-PCD Resources

PCD driver's resources are NOT identical to PCD hardware resources and provide an abstraction layer to the hardware resources. PCD is viewed as a graph of PCD resources where FMan RX & OP Ports may be bound to subsets of the PCD graph. Refer to Port-PCD Binding on page 287.

The following are the driver's PCD resources:

- Network Environment Characteristics
- Software Parser
- Keygen Schemes
- 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.

4.2.8.1.5.2.7.1.1.4 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		

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

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 classification plan
1	1	1	1	1	1	0	0	1...1	Ethernet 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

Table continues on the next page...

Table continued from the previous page...

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

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.

Ethernet 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

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 57. 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 271).

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 58. 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.

4.2.8.1.5.2.7.1.1.5 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.

4.2.8.1.5.2.7.1.1.6 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 271). 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.

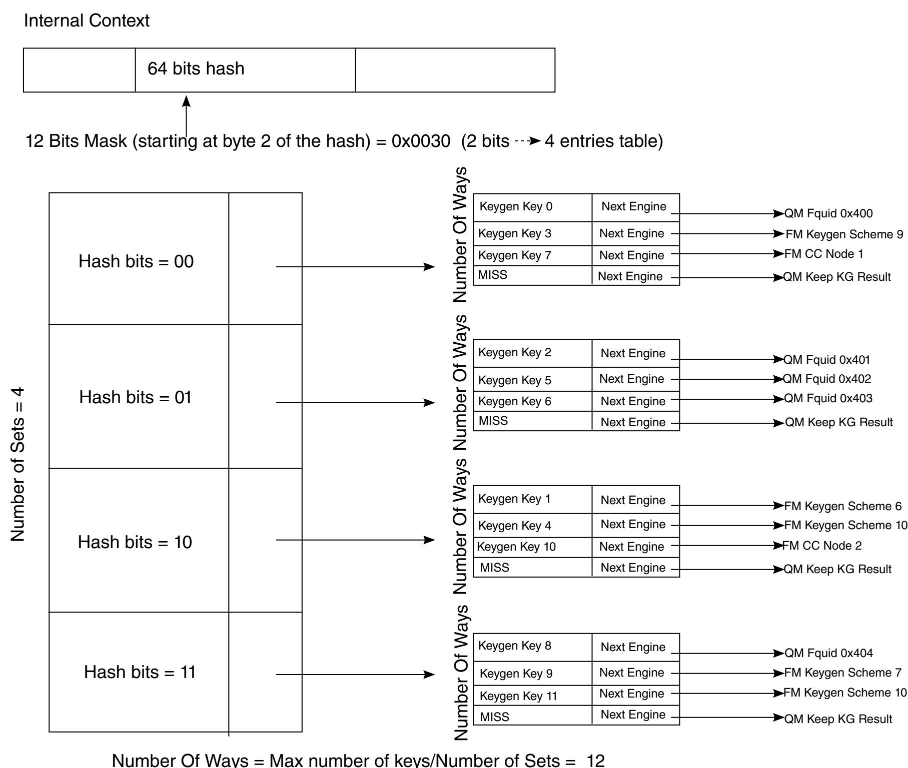


Figure 59. Hash_Table node example

4.2.8.1.5.2.7.2 How to use the FMan PCD Driver?

The following sections provide practical information for using the software drivers.

4.2.8.1.5.2.7.2.1 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

4.2.8.1.5.2.7.2.2 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

4.2.8.1.5.2.7.2.3 FMan PCD Driver Functional Description

The following sections describe main driver functionalities and their usage.

4.2.8.1.5.2.7.2.3.1 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 287 for more details on port-PCD binding.

4.2.8.1.5.2.7.2.3.2 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:

4.2.8.1.5.2.7.2.3.3 **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 294).

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 than 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 60. 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 retunes 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 263.

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`

4.2.8.1.5.2.7.2.3.4 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`

4.2.8.1.5.2.7.2.3.5 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 294.

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 268.

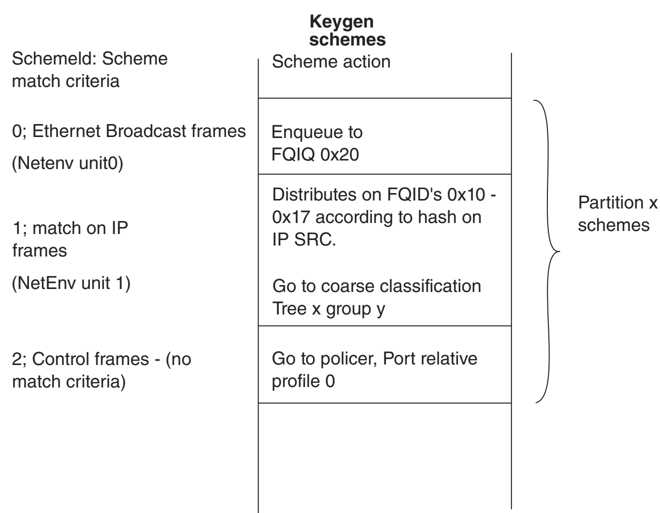


Figure 61. Schemes Example

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 retuned 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`

4.2.8.1.5.2.7.2.3.6 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 44. 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)

CC Tree

Group	Entry
Group 0	0 Go to CC Node 2
	1 Enqueue to FQID 0x10
	....
	7 Go to CC Node 3
Group 1	0 Go to CC Node 3
	1 Go to CC Node 4
	2 Go to PLCR Profile 0
	3 Go to KG Scheme 2
Group 2	0 Go to CC Node 1
	1 Go to CC Node 0
Group 3	0 Go to CC Node 5
Group 4	0 Go to CC Node 6

Figure 62. 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.

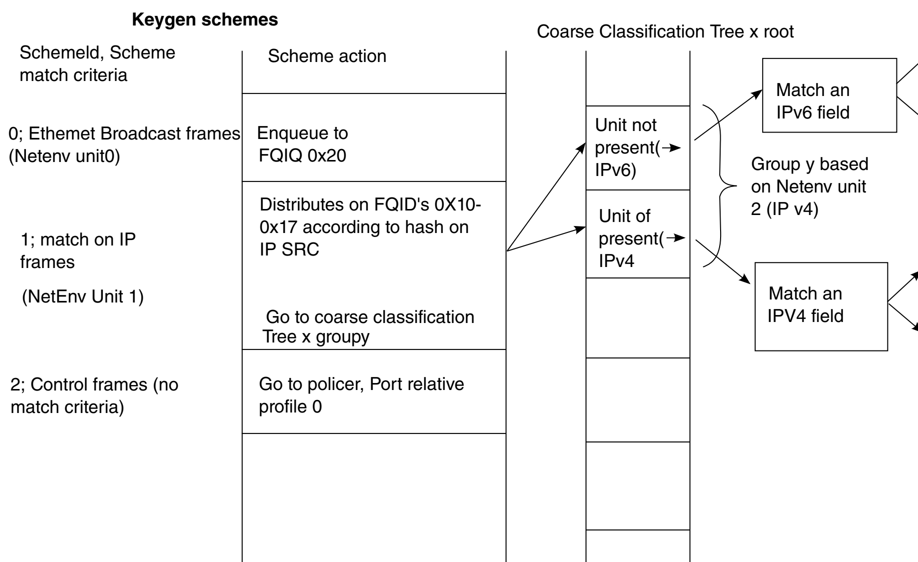


Figure 63. Keygen -> Custom Classifier Example

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`

4.2.8.1.5.2.7.2.3.7 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.

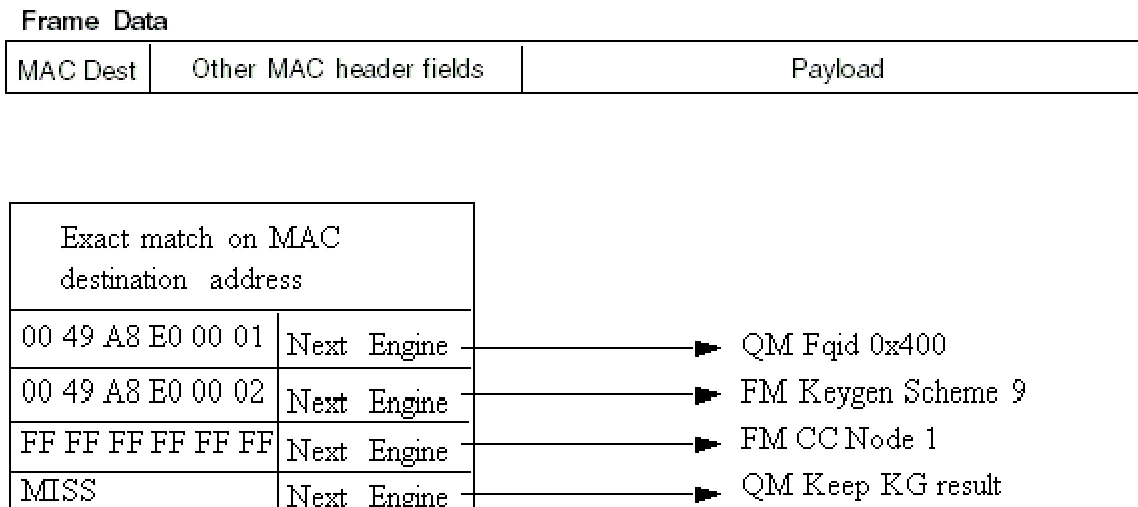


Figure 64. Exact-Match Node Example

On an Indexed-lookup node, up to 2^12 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.

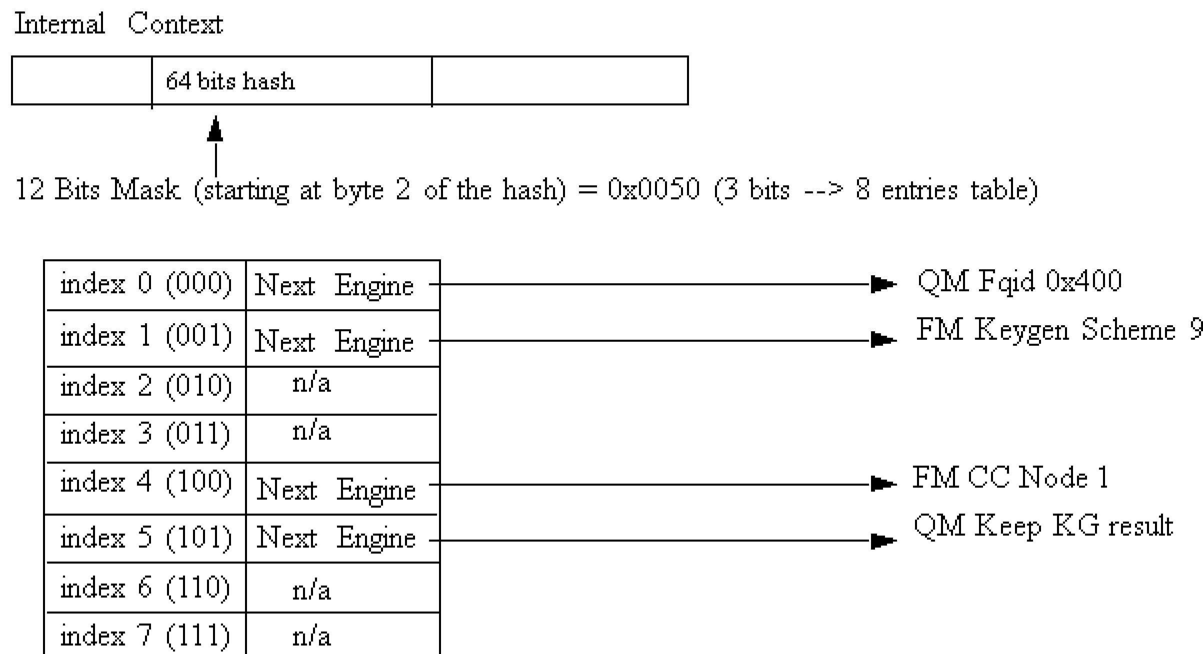


Figure 65. Indexed-Lookup node example

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.

4.2.8.1.5.2.7.2.3.8 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).

4.2.8.1.5.2.7.2.3.9 **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.

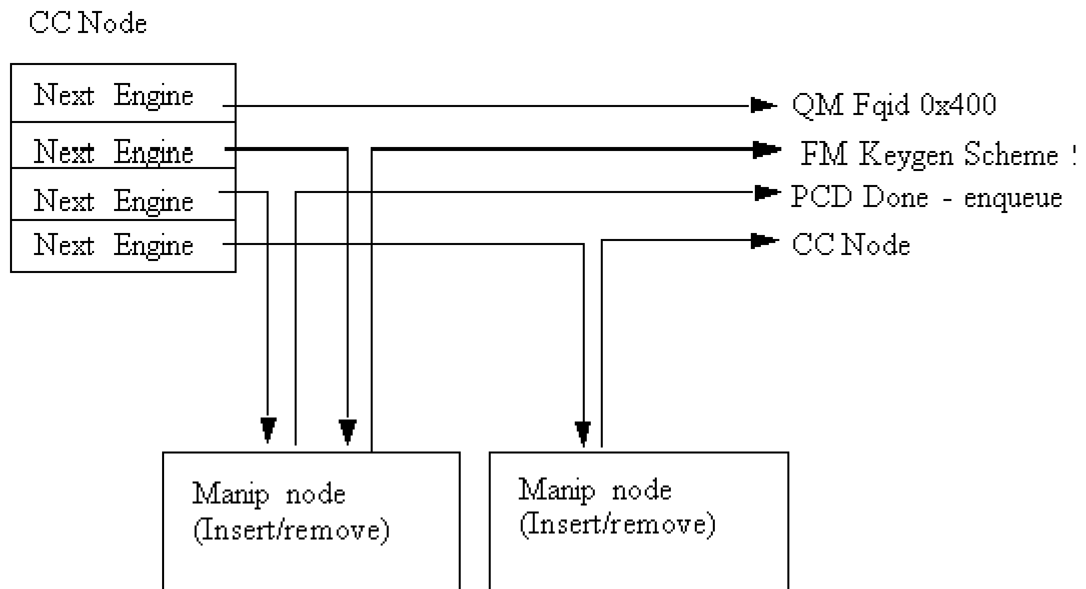


Figure 66. CC Node With Manipulation

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 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`'

4.2.8.1.5.2.7.2.3.9.1 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.

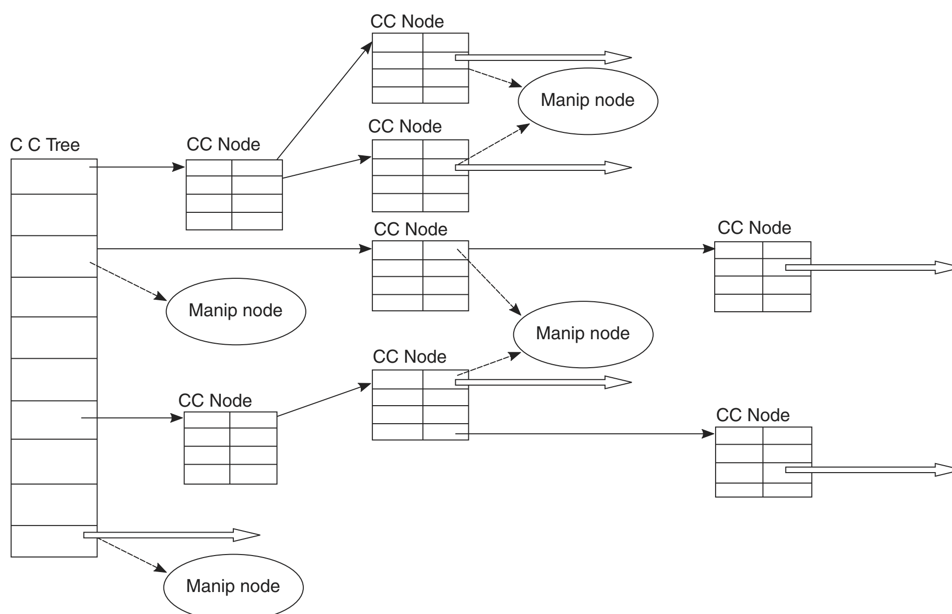


Figure 67. Header Manipulation CC Perspective

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 than 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.

4.2.8.1.5.2.7.2.3.9.2 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 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. 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.

4.2.8.1.5.2.7.2.3.9.3 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 271). 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.
- 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. fragmentation of IP-fragments is not supported
4. IPv4 packets containing header option field are fragments by copying all option fields to each fragment, regardless of the copy bit value.
5. Maximum number of fragments per frame is 16.

Suggestions of how to use IPF in a system:

In case one of the #1-# 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.

4.2.8.1.5.2.7.2.3.9.4 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

4.2.8.1.5.2.7.2.3.10 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 retuned 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`

4.2.8.1.5.2.7.2.3.11 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.

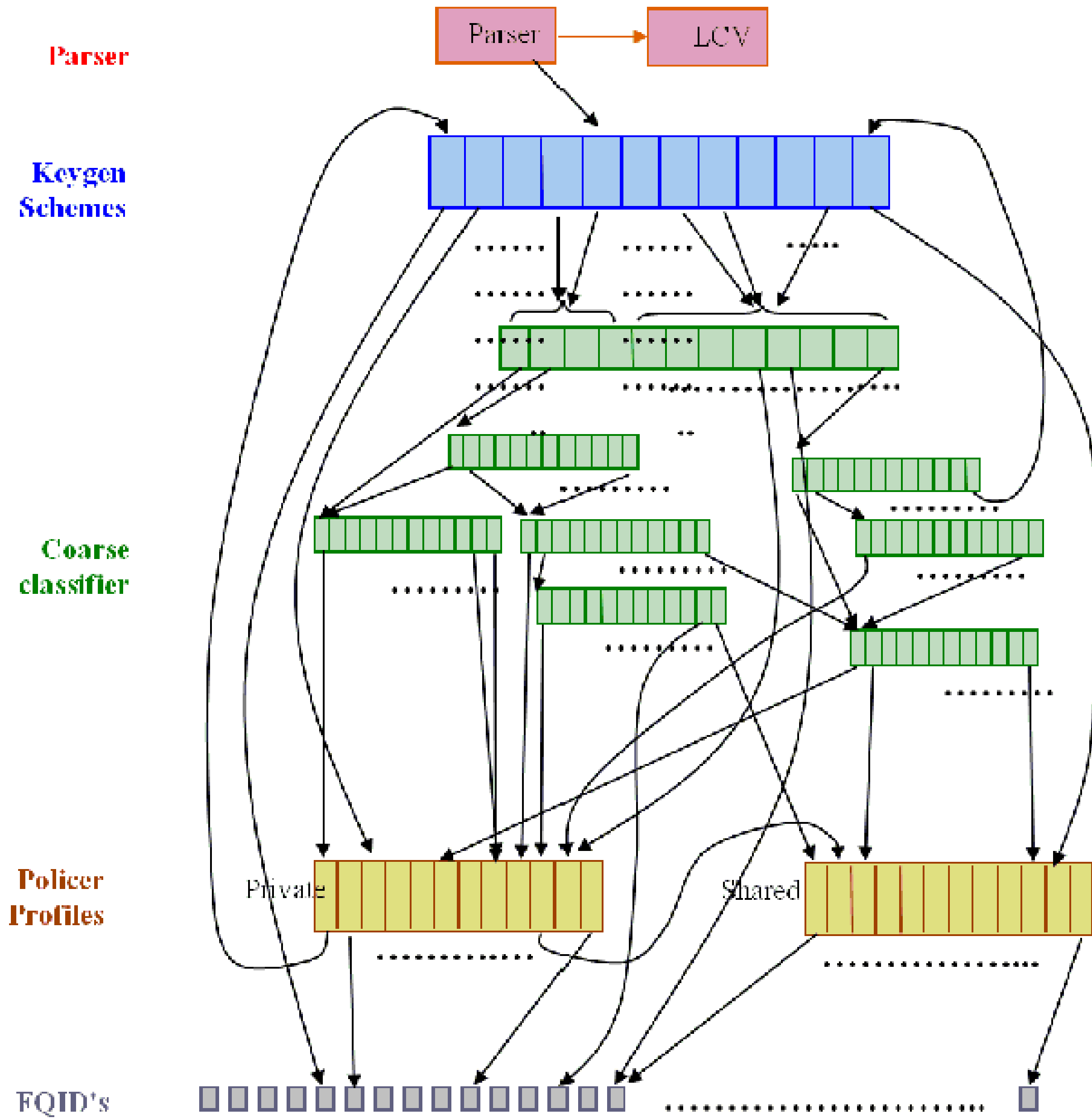


Figure 68. PCD Organization

4.2.8.1.5.2.7.2.3.12 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.

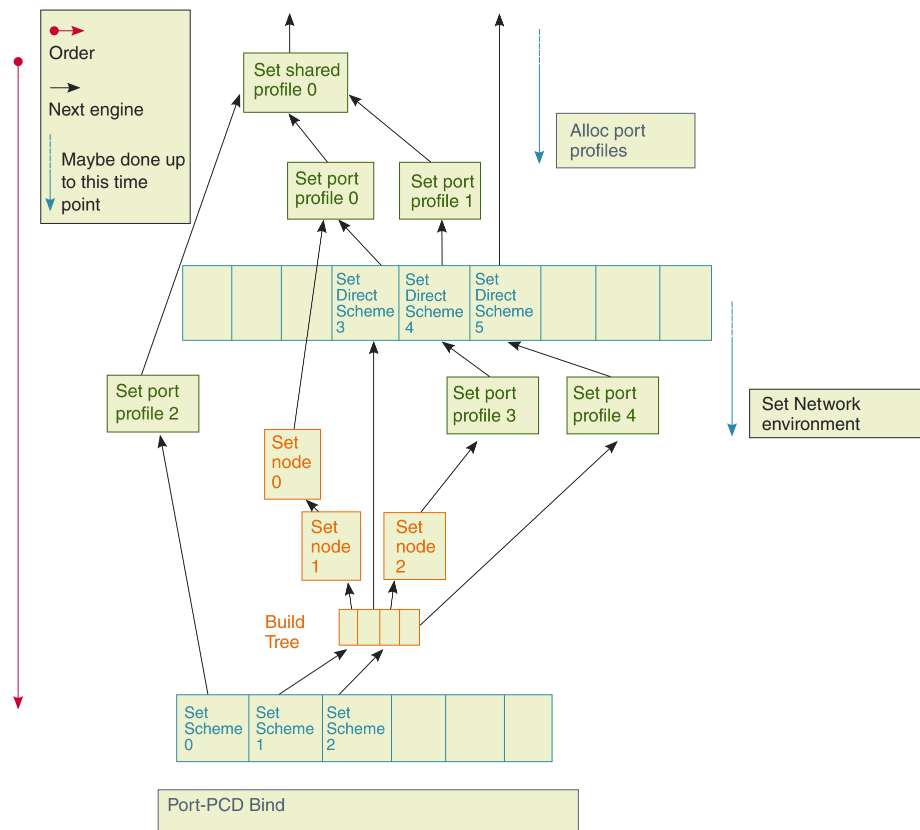


Figure 69. Definition Sequence

4.2.8.1.5.2.7.2.3.13 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.

4.2.8.1.5.2.7.2.3.14 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`

4.2.8.1.5.2.7.2.3.14.1 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`

4.2.8.1.5.2.8 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.

4.2.8.1.5.2.8.1 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.

4.2.8.1.5.2.8.1.1 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.

4.2.8.1.5.2.8.2 How to use the FMan Port Driver?

The following sections provide practical information for using the software drivers.

4.2.8.1.5.2.8.2.1 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

4.2.8.1.5.2.8.2.2 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

4.2.8.1.5.2.8.2.3 FMan Port Driver Functional Description

The following sections describe main driver functionalities and their usage.

4.2.8.1.5.2.8.2.3.1 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.

4.2.8.1.5.2.8.2.3.2 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.

4.2.8.1.5.2.8.2.3.3 Independent-Mode

Dpaa-im is an Ethernet driver using Dpaa to implement in independent mode.

Dependence:

1. All the DPAA drivers in kernel have conflict with dpaa-im, should be disabled in kernel configuration file, the list as below:

CONFIG_FSL_SDK_DPA

CONFIG_FSL_SDK_FMAN

CONFIG_FSL_SDK_DPAA_ETH

CONFIG_FSL_DPAA

CONFIG_FSL_FMAN

CONFIG_FSL_DPAA_ETH

2. linux should be built before building dpaa-im

3. dpaa-im is based on dash-lts 1812 release for linux-4.9 and linux-4.14

Building

To build dpaa-im as a module

cd dpaa-im

make build KERNEL_DIR=<path-to-linux> ARCH=arm64 CROSS_COMPILE=<arm64-toolchain>

e.g. make build KERNEL_DIR=~/linux ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-

after building, you will see module file "dpaa_eth_im.ko"

In addition, use "make clean KERNEL_DIR=<path-to-linux> ARCH=arm64 CROSS_COMPILE=<arm64-toolchain>" to clean

Using

1. Fman firmware should be loaded in uboot.

2. boot up linux

3. In linux, run command "insmod dpaa_eth_im.ko", kernel will print:

[0.535089] fman_im: QorIQ FMAN Independent Mode Ethernet Driver load ed

[0.541782] DEV: FM1@DTSEC3, DTS Node: fsl,dpaa:ethernet@6

4. run command "ifconfig -a", dpaa-im ethernet(FM1@DTSEC3) could be saw, then use it as normal ethernet.

FM1@DTSEC3 Link encap:Ethernet HWaddr 00:e0:0c:00:77:00

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)

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)

4.2.8.1.5.2.8.2.3.4 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-Classifier 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.

4.2.8.1.5.2.8.2.3.5 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.

4.2.8.1.5.2.8.2.3.6 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.

4.2.8.1.5.2.8.2.3.7 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.

4.2.8.1.5.2.8.2.3.8 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.

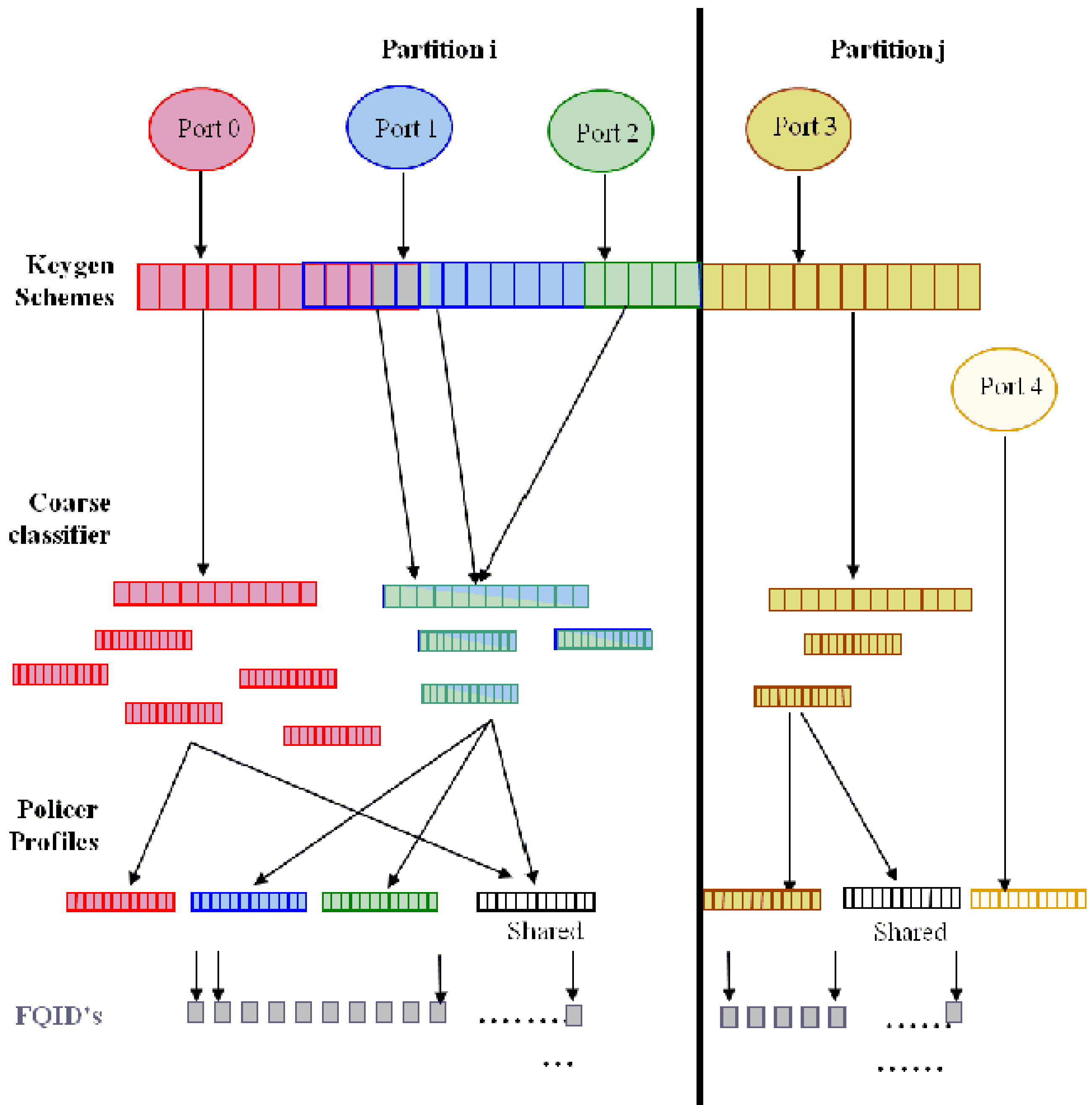


Figure 70. Port-To-PCD binding example

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 bounds 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_DetachPCD` 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 than 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`

4.2.8.1.5.2.8.2.3.9 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_PcdPrsModifyStartOfset
- **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

4.2.8.1.5.2.9 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.

4.2.8.1.5.2.9.1 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.

4.2.8.1.5.2.9.1.1 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.

4.2.8.1.5.2.9.2 How To Use The FMan MAC Driver?

The following sections provide practical information for using the software drivers.

4.2.8.1.5.2.9.2.1 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

4.2.8.1.5.2.9.2.2 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

4.2.8.1.5.2.9.2.3 FMan MAC Driver Functional Description

The following sections describe main driver functionalities and their usage.

4.2.8.1.5.2.9.2.3.1 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.

4.2.8.1.5.2.9.2.3.2 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.

4.2.8.1.5.2.9.2.3.3 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

4.2.8.1.5.2.9.2.3.4 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.

4.2.8.1.5.2.10 FMan RTC (IEEE 1588) Driver

The FMan RTC driver module refers to the software support provided for the IEEE 1588 hardware of the FMan.

4.2.8.1.5.2.10.1 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.

4.2.8.1.5.2.10.2 How To Use The RTC Driver?

The following sections provide practical information for using the software drivers.

4.2.8.1.5.2.10.2.1 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

4.2.8.1.5.2.10.2.2 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

4.2.8.1.5.2.10.2.3 RTC Driver Functional Description

The following sections describe main driver functionalities and their usage.

4.2.8.1.5.2.10.2.3.1 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.

4.2.8.1.5.2.10.2.3.2 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`

From this point and on all the Ethernet frames 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.

4.2.8.1.5.2.10.2.3.3 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.

4.2.8.1.5.2.11 FMan MURAM Driver

The FMan MURAM driver module refers to the memory management of the FMan Multi User RAM.

4.2.8.1.5.2.11.1 FMan MURAM Hardware Overview

The MURAM is the internal memory of the FMan.

4.2.8.1.5.2.11.1.1 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.

4.2.8.1.5.2.11.2 How To Use The FMan MURAM Driver?

The following sections provide practical information for using the software drivers.

4.2.8.1.5.2.11.2.1 FMan MURAM Driver Scope

This module manages the FMan MURAM. It includes MURAM allocation and de-allocation of different sizes of required memory blocks.

4.2.8.1.5.2.11.2.2 FMan MURAM Driver Sequence

- FMan MURAM config and init routine
- FMan MURAM allot and free runtime routines
- FMan MURAM free routine

4.2.8.1.5.2.11.2.3 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.

4.2.8.1.5.2.12 Supported Network Protocols

The following sections show the protocols that may be selected when defining NetEnv characteristics.

4.2.8.1.5.2.12.1 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
  - `MPLS_STACKED`
- `HEADER_TYPE_PPPoE`
- `HEADER_TYPE_LLC_SNAP`

4.2.8.1.5.2.12.2 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`

4.2.8.1.5.2.12.3 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`

4.2.8.1.5.2.12.4 Private Headers

- `HEADER_TYPE_USER_DEFINED_SHIM1`
- `HEADER_TYPE_USER_DEFINED_SHIM2`

4.2.8.1.5.2.12.5 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)
- HEADER_TYPE_IP
  - NET_HEADER_FIELD_IP_PROTO

    (index may apply:

    - 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`)

4.2.8.1.6 Frame Manager Configuration Tool User's Guide

4.2.8.1.6.1 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.

4.2.8.1.6.2 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 299.)
- Generate C source files containing this information that you can include in your application. (See FMC Tool - Host Mode on page 300.)

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 299.)

- 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 300.)

4.2.8.1.6.3 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://ftp.tuwien.ac.at/.vhost/analyzer.polito.it/30alpha/docs/dissectors/NetPDLCore.htm.

For documentation of NXP's customized version of NetPDL, see NXP NetPDL Reference on page 317.

4.2.8.1.6.4 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 71. on page 300shows, you pass these files to the FMC Tool as command-line arguments:

- Standard Protocol file - Optional; included in LSDK; see Standard Protocol File on page 303 for more information.
- Custom Protocol file - Optional; user written; see Custom Protocol File on page 304 for more information.
- Policy file - Required; user written; see Policy file on page 305 for more information.
- Configuration file - Required; user written; see Configuration File on page 317 for more information.

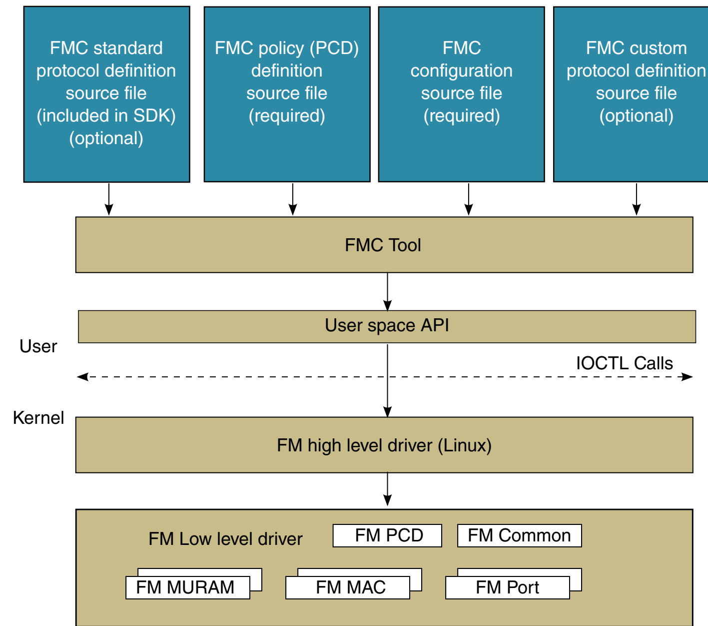


Figure 71. FMC Tool, Runtime Environment - Input XML Files / FMan Driver API Calls

See FMC Tool Command-Line Arguments on page 302 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.

4.2.8.1.6.5 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 72. on page 301 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 301 for more information.)

- Standard Protocol File - Optional; included in LSDK; see Standard Protocol File on page 303 for more information.
- Custom Protocol File - Optional; user written; see Custom Protocol File on page 304 for more information.
- Policy File - Required; user written; see Policy file on page 305 for more information.
- Configuration File - Required; user written; see Configuration File on page 317 for more information.

You pass these files to the FMC Tool as command-line arguments.

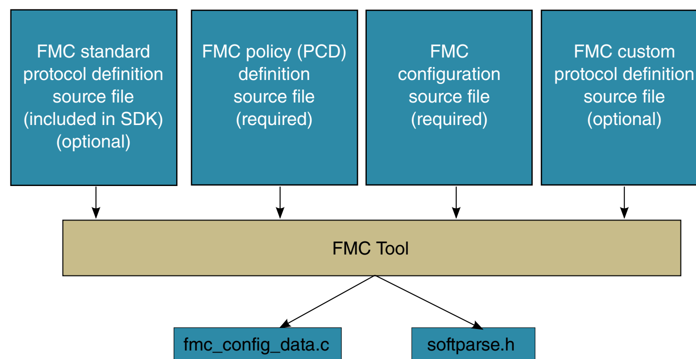


Figure 72. FMC Tool, Host Mode - Input XML Files / Generated C Source Code Files

See FMC Tool Command-Line Arguments on page 302 for documentation of each of the tool's command-line arguments.

4.2.8.1.6.5.1 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.

4.2.8.1.6.6 FMC Tool Command-Line Arguments

The table below lists and describes the FMC Tool's command-line arguments.

Table 45. FMC Tool Command-Line Arguments

Command-Line Argument Syntax ***(Both the verbose and abbreviated command forms are shown)***	**Description**
-d <pdl_file>, --pdl <pdl_file>	Path to and name of the Standard Protocol file. *(Optional)* You can use a full path or a relative path. See Standard Protocol File on page 303 for more information.
-p <pcd_file>, --pcd <pcd_file>	Path to and name of a Policy file. *(Required unless '--sp_only' is used)* You can use a full path or a relative path. See Policy file on page 305 for more information.
-c <data_file>, --config <data_file>	Path to and name of the Configuration file. *(Required unless '--sp_only' is used)* You can use a full path or a relative path. See Configuration File on page 317 for more information.
-s <custom_protocol_file>, --custom_protocol <custom_protocol_file>	Path to and name of the Custom Protocol file. *(Optional unless the '--sp_only' flag is used, in which case, this Custom Protocol file name is required.)* You can use a full path or a relative path. See Custom Protocol File on page 304 for more information.
-a, --apply	Apply the supplied configuration to the FMan rather than generating C source code. *(Optional; valid only when FMC Tool is executed in runtime environment)*
--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)*

Table continues on the next page...

Table 45. FMC Tool Command-Line Arguments (continued)

Command-Line Argument Syntax ***(Both the verbose and abbreviated command forms are shown)***	**Description**
--version	Display version information, then exit. *(Optional)*
-h, --help	Display usage information, then exit. *(Optional)*

4.2.8.1.6.7 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://ftp.tuwien.ac.at/.vhost/analyzer.polito.it/30alpha/docs/dissectors/NetPDLCore.htm).

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.

4.2.8.1.6.8 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.

4.2.8.1.6.8.1 Standard Protocol File

The LSDK 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.

The general structure of an FMC Standard Protocol XML file is shown below.

```
<netpdl>
    <protocol> <!-- one or more -->

        <format> <!-- only one -->
            <fields> <!-- only one -->
```

```
                <field/> <!-- one or more -->
            </fields>
        </format>

        <execute-code>
        </execute-code>

        <encapsulation>
        </encapsulation>

        <visualization>
        </visualization>

    </protocol>
</netpdl>
```

See the Standard Protocol File - Excerpt on page 369 topic to see a larger portion of the Standard Protocol file.

4.2.8.1.6.8.2 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 the you become familiar with the NXP NetPDL Reference on page 317 topic, which fully documents the custom version of NetPDL used in custom protocol definitions.

See Custom Protocol File - GTP Protocol Example on page 376, 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>
```

4.2.8.1.6.9 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 340 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 73. 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>
```

4.2.8.1.6.9.1 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 74. 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 342 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. Bitwise OR the result to the base FQID specified by the 'base' attribute of the 'queue' child element

Figure 75. on page 307 shows the algorithm the KeyGen sub block uses to calculate a FQID.

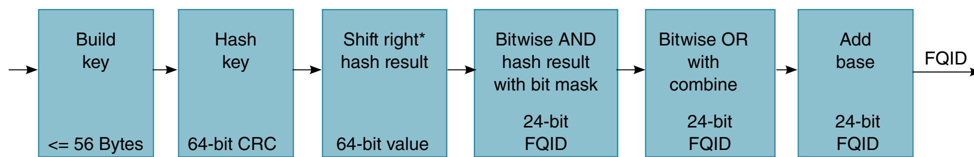


Figure 75. KeyGen Algorithm for FQID Calculation

* 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 76. on page 307 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:~/ltib-no-hv/ltib-e500mc-20091218/rpm/BUILD/lw
File  Edit  View  Terminal  Tabs  Help
<distribution name="eth_dist"
                description="Etherenet 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%
```

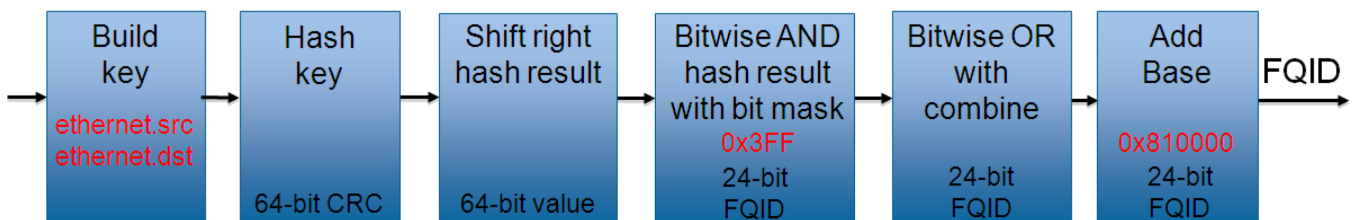


Figure 76. FQID Calculation - Elements/Attributes Used for Key, Bit Mask, and Base FQID

Figure 77. on page 308shows 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.

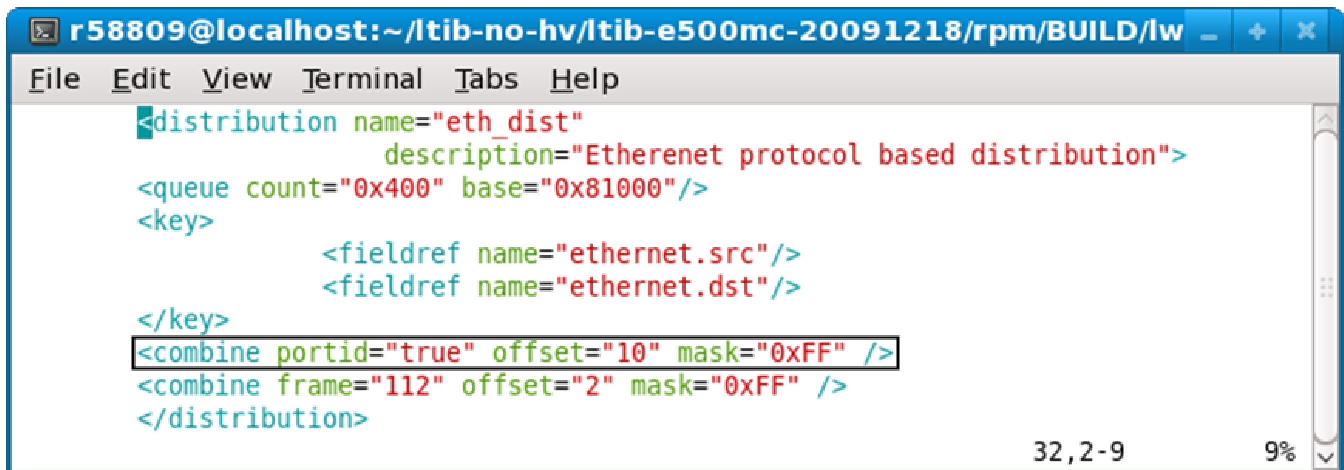


Figure 77. FQID Calculation - A 'combine' Element that Uses the 'portid' Attribute

Figure 78. on page 309 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="Etherenet 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 78. FQID Calculation - A 'combine' Element that Uses the 'frame' Attribute

Finally, Figure 79. on page 309 shows where the KeyGen sub block plugs the values from each of the combine elements into the bitwise OR part of the FQID calculation.

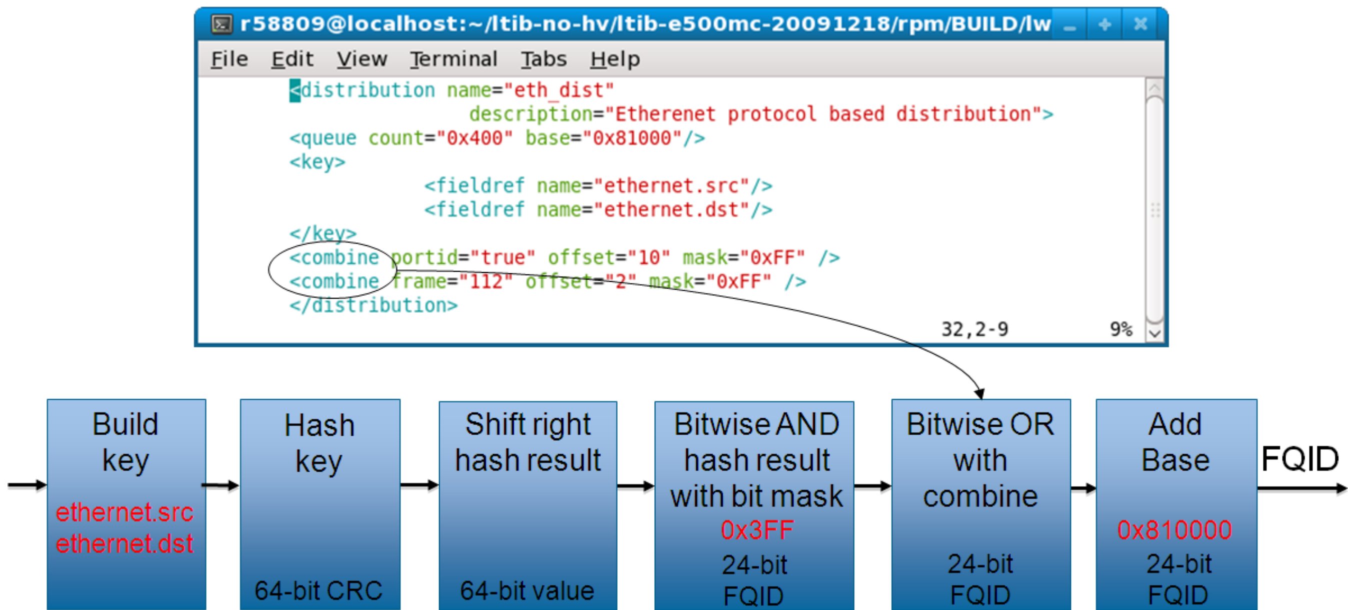


Figure 79. FQID Calculation - combine Elements Used in Bitwise OR

FQID Formula

```
FQID[0:23] = (Shifted Hash Key[0:23] & Hash Mask) |
              Data0[0:23] | Data1[0:23] | … | Data7[0:23] |
              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.

4.2.8.1.6.9.2 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 0 -->
<!-- Policy dl_policy is defined in the Policy file - see next code snippet -->
<cfgdata>
  <config>
    <engine name="fm0">
      <port type="MAC" number="9" 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 a LS1043A chip's first FMan (fm0). 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 340 for complete documentation of this element.

Example 2 - More Complex Use of the Policy Element

Figure 80. on page 311 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 an SoC'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.

```
<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 80. 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 81. on page 312.

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.dist fields as the hash key. The KeyGen sub block then uses the hash result to compute a FQID. (See the Distribution Section on page 306 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.

```
r58809@localhost:~/ltib-e500mc-20100630/rpm/BUILD/lwe_a
File  Edit  View  Terminal  Tabs  Help

<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 81. 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 82. on page 313.

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.

r58809@localhost:~/ltib-e500mc-20100630/rpm/BUILD/lwe_a

File Edit View Terminal Tabs Help

```
        <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 82. More Complex Policy File - 3

Let’s say that an FMan port is tied to policy 'pktwr_policy_1' - highlighted in Figure 83. on page 314.

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.

```
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>
```

Figure 83. More Complex Policy File - 4

4.2.8.1.6.9.3 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 350 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%--
```

4.2.8.1.6.9.4 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[7]
- PBS, Peak Burst Size[8]

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>
```

[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.
[7] If "unit" attribute is "packet" specify CIR and PIR in packets/second. If "unit" attribute is "byte" specify CIR and PIR in Kbits/second
[8] If "unit" attribute is "packet" specify CBS and PBS in packets. If "unit" attribute is "byte" specify CBS and PBS in bytes.

```
<policer name="vlan_congestion_control_red">
  <algorithm>pass_through</algorithm>

  <color_mode>color_blind</color_mode>

  <default_color>red</default_color>

  <action condition="on-red" type="drop"/>
</policer>
```

4.2.8.1.6.10 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 84. on page 317 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 84. on page 317 shows an example configuration file. It uses the optional 'portid' attribute for the 1 Gbps ports.

Figure 84. 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>
```

4.2.8.1.6.11 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.

4.2.8.1.6.11.1 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 alphabetics.

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"/>
```

4.2.8.1.6.11.2 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>
```

4.2.8.1.6.11.3 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 46. Valid values for the prevproto attribute on page 319 lists the values that you can assign to the 'prevproto' attribute.

Table 46. 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 **NOTE** 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 320.	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.

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>
```

4.2.8.1.6.11.3.1 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.

4.2.8.1.6.11.4 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

4.2.8.1.6.11.4.1 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

4.2.8.1.6.11.4.2 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

4.2.8.1.6.11.5 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>
```

4.2.8.1.6.11.5.1 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 the 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 the 'prevproto' attribute is 'otherl3' or 'otherl4'. See Effect of Setting prevproto Attribute to otherl3 or otherl4 on page 320 for more information.

4.2.8.1.6.11.5.2 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 338) 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>
```

4.2.8.1.6.11.5.3 Child Elements of the before and after Elements

4.2.8.1.6.11.5.3.1 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"/>
```

4.2.8.1.6.11.5.3.2 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 337 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>
```

4.2.8.1.6.11.5.3.2.1 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>
```

4.2.8.1.6.11.5.3.2.2 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>
```

4.2.8.1.6.11.5.3.3 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 swith 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>
```

4.2.8.1.6.11.5.3.3.1 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>
```

4.2.8.1.6.11.5.3.3.2 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>
```

4.2.8.1.6.11.5.3.4 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 349 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 from the table below:. The default value is 'return'.

Table 47. 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

Table continues on the next page...

Table 47. Parse Action for each Value of the nextproto Attribute (continued)

If nextproto is ...	The parse action is ...
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	Jump to the protocol that should follow the Ethernet header. The next protocol is determined from the value of the $nxtHdr variable. See Table 48. Next Protocol for each $nxtHdr Value if nextproto is 'after_ethernet' on page 328to find the next protocol for each possible value of $nxtHdr. **Note:**The 'advance' attribute must be set to 'yes' if 'nextproto' is set to 'after_ethernet'.
after_ip	Jump to the protocol that should follow the IP header. The next protocol is determined from the value of the $nxtHdr variable. See table: **Next Protocol for each $nxtHdr Value if nextproto is 'after_ethernet'** to find the next protocol for each possible value of $nxtHdr. **Note:**The 'advance' attribute must be set to 'yes' if 'nextproto' is set to 'after_ip'.
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 48. 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	otherl3

Table 49. 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	otherl4

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"/>
```

4.2.8.1.6.11.6 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').

4.2.8.1.6.11.6.1 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).

4.2.8.1.6.11.6.1.1 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 0xfffffffe).

4.2.8.1.6.11.6.1.2 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>
```

4.2.8.1.6.11.6.1.3 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.

4.2.8.1.6.11.6.1.3.1 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 50. Result Array Variables on page 330shows the available result array variables .

Table 50. 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
fragoffset	28-29
routtype	30-30
rhp	31-31
ipvalid	31-31
shimoffset_1	32-32
shimoffset_2	33-33

Table continues on the next page...

Table 50. Result Array Variables (continued)

Variable Name	Result Array Bytes Referenced
ip_pidoffset	34-34
ethoffset	35-35
llcs_napoffset	36-36
vlantcioffset_1	37-37
vlantcioffset_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
ccbase	72-75
ks	76-76
hpnia	77-79
sperc	80-80
ipver	85-85
iplength	86-87

Table continues on the next page...

Table 50. Result Array Variables (continued)

Variable Name	Result Array Bytes Referenced
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.

4.2.8.1.6.11.6.1.3.2 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.

4.2.8.1.6.11.6.1.3.3 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.

4.2.8.1.6.11.6.1.3.4 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>
```

4.2.8.1.6.11.6.1.3.5 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 51. Previous Protocol RA Return Values

If the previous protocol is ...	The value returned from result array is ...
ethernet	$ethoffset
gre	$greoffset
ipv4, ipv6	$Ipoffset_n
llc_snap	$llcsnapoffset
minencap	$minencapoffset
mpls	$mplsoffset_n
pppoe	$pppoeoffset
tcp, udp, sctp, dccp, ipsec_ah, ipsec_esp	$l4offset
vlan	$vlanoffset_n

Table continues on the next page...

Table 51. Previous Protocol RA Return Values (continued)

If the previous protocol is ...	The value returned from result array is ...
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 320.

4.2.8.1.6.11.6.2 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 338 and Logical Expressions on page 337 for more information.)

Table 52. Supported Operators and their Properties on page 334describes 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 52. 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	!=
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

Table continues on the next page...

Table 52. Supported Operators and their Properties (continued)

Name	Parameters	Description	Symbol
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 335 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 335 for full documentation	checksum

4.2.8.1.6.11.6.2.1 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">
```

4.2.8.1.6.11.6.2.2 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)
```

4.2.8.1.6.11.6.2.3 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.

4.2.8.1.6.11.6.2.4 Operator Precendence

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

4.2.8.1.6.11.6.2.5 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">
  ```

4.2.8.1.6.11.6.3 Expression Types

There are two main types of expressions: Logical expressions, which return "true" or "false", and arithmetic expressions, which return a numeric result.

4.2.8.1.6.11.6.3.1 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)
  ```

4.2.8.1.6.11.6.3.2 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
  ```

4.2.8.1.6.11.7 Tips and Recommendations

4.2.8.1.6.11.7.1 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 322, The after Element on page 322, and The action Element (for use in a Custom Protocol file) on page 326 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.

4.2.8.1.6.11.7.2 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.

4.2.8.1.6.11.7.3 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. In this case when "after_ethernet" is used as next protocol, $nxtHdr variable but be dynamically assigned accordingly from custom protocol header by using next protocol and field names as value.

```
<assign-variable name="$nxtHdr" value="protocol.field"/>
<action type="exit" advance="yes" next="after_ethernet">
```

4.2.8.1.6.11.8 Limitations

This section discusses limitations you should consider when working with the FMC Tool's Soft Parser functionality.

4.2.8.1.6.11.8.1 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.

4.2.8.1.6.12 NetPCD Reference

4.2.8.1.6.12.1 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.

4.2.8.1.6.12.1.1 netpcd Attribute Definitions

Table 53. 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

4.2.8.1.6.12.1.2 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>
```

4.2.8.1.6.12.2 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.

4.2.8.1.6.12.2.1 policy Attribute Definitions

Table 54. 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.

4.2.8.1.6.12.2.2 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>
```

4.2.8.1.6.12.3 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 processing sequentially, and the first conforming distribution is the distribution that is used. Thus, the order of distribution references is important.

4.2.8.1.6.12.3.1 dist_order Attribute Definitions

Table 55. dist_order Attribute Definitions

Attribute	Requirement	Description
none	n/a	n/a

4.2.8.1.6.12.3.2 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.

4.2.8.1.6.12.4 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.

4.2.8.1.6.12.4.1 distributionref Attribute Definitions

Table 56. distributionref Attribute Definitions

Attribute	Requirement	Description
name	required	Name of the referenced 'distribution' element

4.2.8.1.6.12.4.2 distributionref Example

```
<policy name="ipv4">
  <dist_order>
    <distributionref name="eth_dist"/>
    <distributionref name="default_dist"/>
  </dist_order>
</policy>
```

4.2.8.1.6.12.5 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.

4.2.8.1.6.12.5.1 distribution Attribute Definitions

Table 57. 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.

4.2.8.1.6.12.5.2 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_clsf"/>
</distribution>
```

4.2.8.1.6.12.5.3 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 58. '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 59. 'default' Element Attributes:

Attribute	Requirement	Description
type	required	Default type select. Possible values are: 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: 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>
```

4.2.8.1.6.12.6 The key element

The 'key' element contains a list of 'fieldref' elements. The 'filedref' 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.

4.2.8.1.6.12.6.1 key Attribute Definitions

Table 60. 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.

4.2.8.1.6.12.6.2 key Example

```
<key shift="16">
  <fieldref name="ethernet.src"/>
  <fieldref name="ethernet.dst"/>
</key>
```

4.2.8.1.6.12.7 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/.

4.2.8.1.6.12.7.1 fieldref Attribute Definitions

Table 61. 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".

4.2.8.1.6.12.7.2 fieldref Example

```
<key>
  <fieldref name="ethernet.src"/>
  <fieldref name="ethernet.dst"/>
</key>
```

4.2.8.1.6.12.8 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.

4.2.8.1.6.12.8.1 queue Attribute Definitions

Table 62. 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.

4.2.8.1.6.12.8.2 queue Example

```
<distribution name="eth_dist">
  <queue count="0x400" base="0x810000"/>
  <key>
    <fieldref name="ethernet.src"/>
    <fieldref name="ethernet.dst"/>
  </key>
</distribution>
```

4.2.8.1.6.12.9 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.

4.2.8.1.6.12.9.1 protocols and protocolref Attribute Definitions

Table 63. protocols and protocolref Attribute Definitions

Attribute	Requirement	Description
name	required	The name of the protocol.
opt	optional	Applicable only for protocolref attribute Use it in a scheme for detecting protocols with the chosen options (e.g. to detect ETHERNET with BROADCAST or MULTICAST option) 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

Table 64. Protocol options. Groups are separated by empty rows.

Value	Description
0x80000000	Ethernet Broadcast

Table continues on the next page...

Table 64. Protocol options. Groups are separated by empty rows. (continued)

Value	Description
0x40000000	Ethernet Multicast
0x20000000	Stacked VLAN
0x10000000	Stacked MPLS
0x08000000	IPv4 Broadcast
0x04000000	IPv4 Multicast
0x02000000	Tunneled IPv4 - Unicast
0x01000000	Tunneled IPv4 - Broadcast/Multicast
0x00000008	IPV4 reassembly option. When using this option, the IPV4 Reassembly manipulation requires network environment with IPV4 header
0x00800000	IPv6 Multicast
0x00400000	Tunneled IPv6 - Unicast
0x00200000	Tunneled IPv6 - Multicast
0x00000004	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).
0x00000008	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).

4.2.8.1.6.12.9.2 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>
```

4.2.8.1.6.12.10 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.

4.2.8.1.6.12.10.1 combine Attribute Definitions

Table 65. combine Attribute Definitions

Attribute	**Requirement**	**Description**
portid	required *(in absence of frame attribute)*	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. *Note that portid and frame are mutually exclusive attributes.*
frame	required *(in absence of portid attribute)*	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. *Note that portid and frame are mutually exclusive attributes.*
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.

4.2.8.1.6.12.10.2 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_clsf"/>
</distribution>
```

4.2.8.1.6.12.11 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.

4.2.8.1.6.12.11.1 action Attribute Definitions

Table 66. action Attribute Definitions

Attribute	Requirement	Description
type	required	The type of the 'action' element defines the next processing element. Valid values are: • "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 *(when used within a* '*policer*' *element)* optional *(when used within a* '*distribution*' *or* '*classification*' *element)*	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: • "on-green" • "on-yellow" • "on-red" • "on-miss"

4.2.8.1.6.12.11.2 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 67. 'statistics' Element Attributes:

Attribute	Requirement	Description
statistics	optional	Enable statistics for a particular action. Possible values are: • enable/yes/true – to enable it. • disable

4.2.8.1.6.12.11.3 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>1400000</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>
```

4.2.8.1.6.12.12 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

4.2.8.1.6.12.12.1 classification Attribute Definitions

Table 68. classification Attribute Definitions

Attribute	Requirement	Description
name	required	The name of the classification

4.2.8.1.6.12.12.2 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 69. 'statistics' Element Attributes:

Attribute	Requirement	Description
statistics	optional	Choose statistic mode for the particular entry. Possible values are: • none • frame • byteframe • rmon

4.2.8.1.6.12.12.3 classification Example

```
<classification name="eth_dest_clsf">
  <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_clsf"/>
  </entry>

  <action condition="on-miss" type="distribution" name="default_dist"/>
</classification>
```

4.2.8.1.6.12.12.4 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 70. '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>
```

4.2.8.1.6.12.12.5 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 71. '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).

4.2.8.1.6.12.12.6 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"/>
            <action statistics="enable"/>
        </entry>
    </classification>
```

4.2.8.1.6.12.12.7 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 attibutes **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 72. Resource Reservation Attributes:

Attribute	Requirement	Description
max	optional	If it exists, this parameter defines the maximum number of coarse classification entries allocated for this PCD element. **NOTE** The element **classification** may still contain pre-initialized entries, or, alternatively, be empty. **NOTE** For the case of empty or partially initialized element **classification**, usage of the element **may-use** might be required .
masks	optional	If provided, indicates that MURAM allocation should be done with the assumption that additional memory is required for an elements' masks. Possible values are: • no – don't allocate memory for masks (default) • yes – allocate memory for masks.

'may-use Element Description:

Table 73. 'may-use' Element Attributes:

Attribute	Requirement	Description
may-use	optional	Contains list of 'action' elements that may appear in the 'classification' entries or, be applied dynamically after partial initial configuration. **NOTE** Attention: the use of this element is required if initial 'classification' is empty and dynamic entries, added through FMD API, use those PCD entities

4.2.8.1.6.12.13 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

4.2.8.1.6.12.13.1 entry Attribute Definitions

Table 74. entry Attribute Definitions

Attribute	Requirement	Description
none	n/a	n/a

4.2.8.1.6.12.13.2 entry Example

```
<classification name="eth_dest_clsf">
  <key>
    <fieldref name="ethernet.dst"/>
  </key>

  <entry>
    <data>0x1234567890AB1234567890AB</data>
    <queue base="0x550000"/>
  </entry>
</classification>
```

4.2.8.1.6.12.14 The policer element

The 'policer' element is a container whose child elements define a policer profile that performs network bandwidth management.

4.2.8.1.6.12.14.1 policer Attribute Definitions

Table 75. 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[1]
PIR	required	Peak (or excess) information rate[1]
CBS	required	Committed burst size[2]
PBS	required	Peak (or excess) burst size[2]

1. If "unit" attribute is "packet" specify CIR and PIR in packets/second. If "unit" attribute is "byte" specify CIR and PIR in Kbits/second.
2. If "unit" attribute is "packet" specify CBS and PBS in packets. If "unit" attribute is "byte" specify CBS and PBS in bytes.

4.2.8.1.6.12.14.2 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>
```

4.2.8.1.6.12.15 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.

4.2.8.1.6.12.15.1 nonheader Attribute Definitions

Table 76. nonheader Attribute Definitions

Attribute	Requirement	Description
source	required	Non-header extraction source Valid values are: • "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.	The type of action for the extraction Valid values are: • "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

Table continues on the next page...

Table 76. nonheader Attribute Definitions (continued)

Attribute	Requirement	Description
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 nuber 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 ajdusts the maxim number of entries if it’s not configured as 0.

4.2.8.1.6.12.15.2 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>
```

4.2.8.1.6.12.16 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 77. '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^(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>
```

4.2.8.1.6.12.17 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.

4.2.8.1.6.12.17.1 vsp Attributes

Table 78. '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.
type	optional	The type of the VSP. Values: • 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.

4.2.8.1.6.12.17.2 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_clsf">
    <key>
         <fieldref name="ethernet.dst"/>
    </key>
...
    <vsp name=”storage03”>
    <action condition="on-miss" type="distribution" name="garbage"/>
</classification>
```

4.2.8.1.6.12.18 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.

4.2.8.1.6.12.18.1 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 79. 'fragmentation' Element Attributes:

Attribute	Requirement	Description
name	required	Name of the element. The name is used to refer the manipulation algorithm.

Table 80. '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: • 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>
```

4.2.8.1.6.12.18.2 IP Reassembly

XML element **reassembly** 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 **reassembly** participates in a configuration.

Table 81. 'reassembly' Element Attributes:

Attribute	Requirement	Description
Name	required	Name of the element. The name is used to refer the manipulation algorithm

Table 82. 'reassembly' 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: • 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.
numOfFramesPerHash Entry (numOfFramesPerHash Entry1)	required	Number of frames per hash entry needed for reassembly process – for ipv4. Possible values are: numeric values from 1 to 8.
numOfFramesPerHash Entry2	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>
    <reassembly name=”reasm1”>
        <sgBpid>2</sgBpid>
        <maxInProcess>1024</maxInProcess>
```

```
        <timeOutMode>fragment</timeOutMode>
        <fqidForTimeOutFrames>1024</fqidForTimeOutFrames>
        <numOfFramesPerHashEntry>8</numOfFramesPerHashEntry>
        <timeoutThreshold>1000000</timeoutThreshold>
        <ipv4minFragSize>0</ipv4minFragSize>
        <ipv6minFragSize>256</ipv6minFragSize>
    </reassembly>
</manipulations>

<policy name="udp_port">
   <dist_order>
       <distributionref name="custom_dist"/>
       <distributionref name="udp_port_dist"/>
       <distributionref name="default_dist"/>
   </dist_order>

   <reassembly name="reasm1"/>
</policy>
```

4.2.8.1.6.12.18.3 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 83. 'header' Element Attributes:

Attribute	Requirement	Description
name	required	Name of the element. The name is used to refer the manipulation algorithm

Table continues on the next page...

Table 83. 'header' Element Attributes: (continued)

Attribute	Requirement	Description
parse	optional	Activate the parser a second time after completing the manipulation of the frame (if 'yes')
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.

4.2.8.1.6.12.18.3.1 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 84. '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: • no - insert (default) • yes - replace
data	required	Data to insert

4.2.8.1.6.12.18.3.2 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 85. '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.

4.2.8.1.6.12.18.3.3 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 86. 'insert_header' Element Attributes

Element	Requirement	Description
type	required	The type of the header inserted. Only 'mpls' is valid at this time.

Table continues on the next page...

Table 86. 'insert_header' Element Attributes (continued)

Element	Requirement	Description
header_index	optional	The header index of the header has possible values "1" and "2". The restrictions on this attribute are: • 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 87. '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>
```

4.2.8.1.6.12.18.3.4 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 88. 'remove_header' Child Elements

Element	Requirement	Description
type	required	The type of the header remove. Possible values: • "qtags" • "mpls" • "ethmpls (or "ethernet_mpls") • "eth" (or "ethernet")

remove_header example:

```
<header name="remove_l2">
    <remove_header type="qtags/>
</header>
```

4.2.8.1.6.12.18.3.5 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 89. 'remove_header' Child Elements

Element	Requirement	Description
type	required	The type of the update. Possible values: • "vlan" • "ipv4" • "ipv6" • "tcpudp"

update Child Elements:

Table 90. '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 91. 'remove_header' Child Elements

Element	Requirement	Description
type	required	The type of the header remove. Possible values: • for 'vlan' — dscp - DSCP to VLAN priority bits translation. — vpri - Replace VPri of outer most VLAN tag . • for 'ipv4' — 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' — 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' — 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	The value used for the update. It is not valid for: • hl • ttl • checksum
fill	optional	Only valid for **dscp** - fills the entire array with the given value. The fill is performed before the other **dscp** operations.
index	optional	Only valid for **dscp**. Speciefies the index in the array where that value is set. The index starts from 0.

'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>
```

4.2.8.1.6.12.18.3.6 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 92. 'custom' Element Attributes:

Element	Requirement	Description
type	required	The type of the custom header manipulation. Possible values are: • “ipv4byipv6” (or just “ipv4”) – Replaces ipv4 by ipv6. • -“ipv6byipv4” (or just “ipv6”) – Replaces ipv6 by ipv4.

'custom' Child Elements

Table 93. 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: • "yes" • "no"
dechl	optional	Decrement Hop Limit by 1 (ipv6). Possible values: • "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>
```

4.2.8.1.6.12.18.3.7 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 94. Nextmanip element attributes

Element	Requirement	Description
name	required	The name of the next header manipulation

4.2.8.1.6.12.18.3.8 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>
```

4.2.8.1.6.13 Standard Protocol File - Excerpt

The 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.typelength"/>
    </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>
```

```
      <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="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="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(nexthdr)">
      <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.nexthdr" 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>

    <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 -->
```

4.2.8.1.6.14 Custom Protocol File - GTP Protocol Example

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>
```

4.2.8.1.7 Security Engine (SEC)

SEC Device Driver for DPAA1

Introduction

Current chapter is focused on DPAA1-specific SEC details - Queue Interface (QI) backend and frontend drivers. More information is provided in chapter Security Engine (SEC) on page 381, including:

- JRI - the common Job Ring Interface (on which QI is currently dependent)
- crypto algorithms supported by each backend (RI, JRI, QI, DPSECI)
- kernel configuration - how to build backend and frontend drivers
- how to make sure the algorithms registered successfully
- how to check that crypto requests are being offloaded on SEC engine

On SoCs with DPAA v1.x, QI backend can be used to submit crypto API service requests from the frontend drivers. The corresponding frontend compatible with QI backend is *caamalg_qi*, which supports symmetric encryption and 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.

Device Tree binding

There is no device tree node corresponding to SEC DPAA1. A platform device is created dynamically at runtime, as a child of the *crypto* node.

Module loading

Both QI backend and frontend drivers can be compiled either built-in or as modules. If compiled as modules, QI backend driver is (part of) the *caam* module, while the corresponding frontend driver is the *caamalg_qi* module.

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 message at initialization time:

```
platform caam_qi: algorithms registered in /proc/crypto
```

If the message is 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.

Another option is to examine the hardware statistics registers in debugfs.

Incrementing IRQs in /proc/interrupts

Given a time period when crypto requests are being made, the SEC hardware will fire completion notification interrupts on the corresponding QMan (Queue Manager) portal IRQ:

```
$ cat /proc/interrupts | grep QMan
            CPU0        CPU1        CPU2        CPU3
[...]
 21:           0           0           0          22     GICv2 214 Level     QMan portal 3
 22:           0           0          61           0     GICv2 216 Level     QMan portal 2
 23:           0          29           0           0     GICv2 218 Level     QMan portal 1
 24:         273           0           0           0     GICv2 220 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: 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 'self test' fields say 'passed' in /proc/crypto

An entry such as the one below means the driver has successfully registered support for the algorithm with the kernel crypto API:

```
name         : cbc(aes)
driver       : cbc-aes-caam-qi
module       : kernel
priority     : 2000
refcnt       : 1
selftest     : passed
internal     : no
type         : givcipher
async        : yes
blocksize    : 16
min keysize  : 16
max keysize  : 32
ivsize       : 16
geniv        : <built-in>
```

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(md5),cbc(aes)) (authenc-hmac-md5-cbc-aes-caam-qi)
alg: No test for echainiv(authenc(hmac(md5),cbc(aes))) (echainiv-authenc-hmac-md5-cbc-aes-caam-qi)
alg: No test for echainiv(authenc(hmac(sha1),cbc(aes))) (echainiv-authenc-hmac-sha1-cbc-aes-caam-qi)
alg: No test for authenc(hmac(sha224),cbc(aes)) (authenc-hmac-sha224-cbc-aes-caam-qi)
[...]
alg: No test for echainiv(authenc(hmac(sha384),cbc(des))) (echainiv-authenc-hmac-sha384-cbc-des-caam-
qi)
alg :No test for echainiv(authenc(hmac(sha512),cbc(des))) (echainiv-authenc-hmac-sha512-cbc-des-caam-
qi)
[...]
```

Supporting Documentation

General SEC information, Job Ring Interface (JRI): Security Engine (SEC) on page 381

4.2.8.1.8 Decompression Compression Engine (DCE)

Description

The following section describes the DCE software running on the DCE hardware block that is part of the QorIQ family of SoCs.

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 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 demontrates 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.

4.2.9 Security Engine (SEC)

SEC Device Drivers

Introduction and Terminology

The Linux kernel contains a Scatterlist Crypto API driver for the NXP SEC v4.x, v5.x security hardware blocks.

It integrates seamlessly with in-kernel crypto users, such as IPsec, in a way 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.

SEC v4.x hardware is known in Linux kernel as 'caam', after its internal block name: Cryptographic Accelerator and Assurance Module.

There are several HW interfaces ("backends") that can be used to communicate (i.e. submit requests) with the engine, their availability depends on the SoC:

- Register Interface (RI) - available on all SoCs (though access from kernel is restricted on DPAA2 SoCs)

  Its main purpose is debugging (for e.g. single-stepping through descriptor commands), though it is used also for RNG initialization.

- Job Ring Interface (JRI) - legacy interface, available on all SoCs; on most SoCs there are 4 rings

  Note: there are cases when fewer rings are accessible / visible in the kernel - for e.g. when firmware like Trusted Firmware-A (TF-A) reserves one of the rings.

- Queue Interface (QI) - available on SoCs implementing DPAA v1.x (Data Path Acceleration Architecture)

  Requests are submitted indirectly via Queue Manager (QMan) HW block that is part of DPAA1.

- Data Path SEC Interface (DPSECI) - available on SoCs implementing DPAA v2.x

  Similar to QI, requests are submitted via Queue Manager (QMan) HW block; however, the architecture is different - instead of using the platform bus, the Management Complex (MC) bus is used, MC firmware performing needed configuration to link DP* objects - see DPAA2 Linux Software chapter for more details.

NXP provides device drivers for all these interfaces. Current chapter is focused on JRI, though some general / common topics are also covered. For QI and DPSECI backends and compatible frontends, please refer to the dedicated chapters: for DPAA1, Security Engine for DPAA2.

On top of these backends, there are the "frontends" - drivers that sit between the Linux Crypto API and backend drivers. Their main tasks are to:

- register supported crypto algorithms

- process crypto requests coming from users (via the Linux Crypto API) and translate them into the proper format understood by the backend being used
- forward the CAAM engine responses from the backend being used to the users

Note: It is obvious that QI and DPSECI backends cannot co-exist (they can be compiled in the same "multi-platform" kernel image, however run-time detection will make sure only the proper one is active). However, JRI + QI and JRI + DPSECI are valid combinations, and both backends will be active if enabled; if a crypto algorithm is supported by both corresponding frontends (for e.g. both *caamalg* and *caamalg_qi* register cbc(aes)), a user requesting cbc(aes) will be bound to the implementation having the highest "crypto algorithm priority". If the user wants to use a specific implementation:

- it is possible to ask for it explicitly by using the specifc (unique) "driver name" instead of the generic "algorithm name" - please see official Linux kernel Crypto API documentation (section Crypto API Cipher References And Priority); currently default priorities are: 3000 for JRI frontend and 2000 for QI and DPSECI frontends
- crypto algorithm priority could be changed dynamically using the "Crypto use configuration API" (provided that CONFIG_CRYPTO_USER is enabled); one of the tools available that is capable to do this is "Linux crypto layer configuration tool" and an example of increasing the priority of QI frontend based implementation of echainiv(authenc(hmac(sha1),cbc(aes))) algorithm is:

```
$ ./crconf update driver "echainiv-authenc-hmac-sha1-cbc-aes-caam-qi" type 3 priority 5000
```

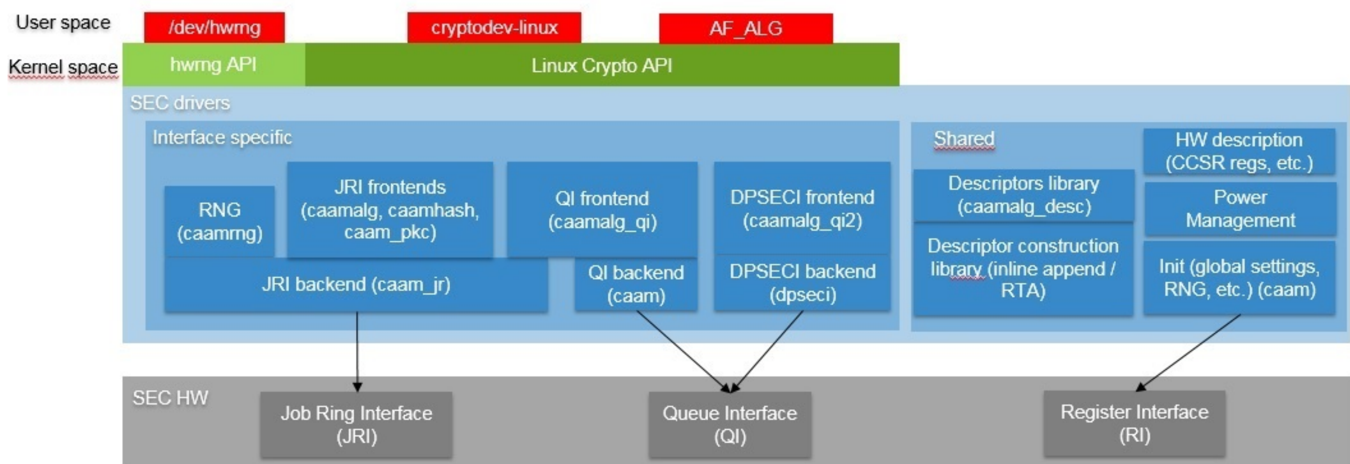


Figure 85. Linux kernel - SEC device drivers overview

Source Files

The drivers source code is maintained in the Linux kernel source tree, under *drivers/crypto/caam*. Below is a non-exhaustive list of files, mapping to Security Engine (SEC)(some files have been omitted since their existence is justified only by driver logic / design):

Source File(s)	Description	Module name
ctrl.[c,h]	Init (global settings, RNG, power management etc.)	caam
desc.h	HW description (CCSR registers etc.)	N/A
desc_constr.h	Inline append - descriptor construction library	N/A

Table continues on the next page...

Table continued from the previous page...

Source File(s)	Description	Module name
caamalg_desc.[c,h]	(Shared) Descriptors library (symmetric encryption, AEAD)	caamalg_desc
caamrng.c	RNG (runtime)	caamrng
jr.[c,h]	JRI backend	caam_jr
qi.[c,h]	QI backend	caam
dpseci.[c,h], dpseci_cmd.h	DPSECI backend	N/A (built-in)
caamalg.c	JRI frontend (symmetric encryption, AEAD)	caamalg
caamhash.c	JRI frontend (hashing)	caamhash
caampkc.c, pkc_desc.c	JRI frontend (public key cryptography)	caam_pkc
caamalg_qi.c	QI frontend (symmetric encryption, AEAD)	caamalg_qi
caamalg_qi2.[c,h]	DPSECI frontend (symmetric encryption, AEAD)	caamalg_qi2

Module loading

CAAM device drivers can be compiled either built-in or as modules (with the exception of DPSECI backend, which is always built-in). See section Source Files on page 382 for the list of module names and section Kernel Configuration on page 383 for how kernel configuration looks like and a mapping between menu entries and modules and / or functionalities enabled.

Kernel Configuration

CAAM device drivers are located in the "Cryptographic API" -> "Hardware crypto devices" sub-menu in the kernel configuration. Depending on the target platform and / or configuration file(s) used, the output will be different; below is an example taken from NXP Layerscape SDK for ARMv8 platforms with default options:

<table>
<tr><th>Kernel Configure Tree View Options</th><th>Description</th></tr>
<tr><td><pre>
Cryptographic API --->
     [*]   Hardware crypto devices   --->
     <*>   Freescale CAAM-Multicore platform
driver backend (SEC)
     [ ]   Enable debug output in CAAM
driver
     <*>   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
     <*>   Register hash algorithm
implementations with Crypto API
</pre></td><td>Enable CAAM device drivers, options:

basic platform driver: Freescale CAAM-Multicore platform driver backend (SEC); all non-DPAA2 sub-options depend on it
backends / interfaces:

Freescale CAAM Job Ring driver backend (SEC) - JRI; this also enables QI (QI depends on JRI)
QorIQ DPAA2 CAAM (DPSECI) driver - DPSECI

frontends / crypto algorithms:

symmetric encryption, AEAD, "stitched" AEAD, TLS; Register algorithm implementations with the Crypto API

</td></tr>
</table>

Table continues on the next page...

Table continued from the previous page...

<table>
<tr><th>Kernel Configure Tree View Options</th><th>Description</th></tr>
<tr><td><pre>   <*>   Register public key
cryptography implementations with Crypto API
     <*>   Register caam device for
hwrng API
     <M>   QorIQ DPAA2 CAAM (DPSECI) driver</pre></td><td>- via JRI (caamalg driver) or Queue Interface as Crypto API backend - via QI (caamalg_qi drive)
— Register hash algorithm implementations with Crypto API - hashing (only via JRI - caamhash driver)
— Register public key cryptography implementations with Crypto API - asymmetric / public key (only via JRI - caam_pkc driver)
— Register caam device for hwrng API - HW RNG (only via JRI - caamrng driver)
— QorIQ DPAA2 CAAM (DPSECI) driver - DPSECI
• options: debugging, JRI ring size, JRI interrupt coalescing</td></tr>
<tr><td><pre>Networking support --->
  Network option --->
    <*> TCP/IP networking
    <*>   IP: AH transformation
    <*>   IP: ESP transformation
    <*>   IP: IPsec transport mode
    <*>   IP: IPsec tunnel mode</pre></td><td>For IPsec support the TCP/IP networking option and corresponding sub-options should be enabled.</td></tr>
</table>

Device Tree binding

Property	Type	Status	Description
compatible	String	Required	fsl,sec-vX.Y (preferred) OR fsl,secX.Y

Sample Device Tree crypto 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-sec4.txt file in the Linux kernel tree for more info.

How to test the drivers

To test the drivers, under the `"Cryptographic API -> Cryptographic algorithm manager"` kernel configuration sub-menu, ensure that run-time self tests are not disabled, i.e. the "Disable run-time self tests" entry is not set (*CONFIG_CRYPTO_MANAGER_DISABLE_TESTS=n*). This will run standard test vectors against the drivers after they register 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 of ARMv8-based LS1046A platform, with JRI and QI enabled:

```
[...]
platform caam_qi: Linux CAAM Queue I/F driver initialised
caam 1700000.crypto: Instantiated RNG4 SH1
caam 1700000.crypto: device ID = 0x0a11030100000000 (Era 8)
caam 1700000.crypto: job rings = 4, qi = 1, dpaa2 = no
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 echainiv(authenc(hmac(md5),cbc(aes))) (echainiv-authenc-hmac-md5-cbc-aes-caam)
alg: No test for echainiv(authenc(hmac(sha1),cbc(aes))) (echainiv-authenc-hmac-sha1-cbc-aes-caam)
alg: No test for authenc(hmac(sha224),cbc(aes)) (authenc-hmac-sha224-cbc-aes-caam)
alg: No test for echainiv(authenc(hmac(sha224),cbc(aes))) (echainiv-authenc-hmac-sha224-cbc-aes-caam)
alg: No test for echainiv(authenc(hmac(sha256),cbc(aes))) (echainiv-authenc-hmac-sha256-cbc-aes-caam)
alg: No test for authenc(hmac(sha384),cbc(aes)) (authenc-hmac-sha384-cbc-aes-caam)
alg: No test for echainiv(authenc(hmac(sha384),cbc(aes))) (echainiv-authenc-hmac-sha384-cbc-aes-caam)
alg: No test for echainiv(authenc(hmac(sha512),cbc(aes))) (echainiv-authenc-hmac-sha512-cbc-aes-caam)
alg: No test for authenc(hmac(md5),cbc(des3_ede)) (authenc-hmac-md5-cbc-des3_ede-caam)
alg: No test for echainiv(authenc(hmac(md5),cbc(des3_ede))) (echainiv-authenc-hmac-md5-cbc-des3_ede-
caam)
alg: No test for echainiv(authenc(hmac(sha1),cbc(des3_ede))) (echainiv-authenc-hmac-sha1-cbc-des3_ede-
caam)
alg: No test for echainiv(authenc(hmac(sha224),cbc(des3_ede))) (echainiv-authenc-hmac-sha224-cbc-
des3_ede-caam)
alg: No test for echainiv(authenc(hmac(sha256),cbc(des3_ede))) (echainiv-authenc-hmac-sha256-cbc-
des3_ede-caam)
alg: No test for echainiv(authenc(hmac(sha384),cbc(des3_ede))) (echainiv-authenc-hmac-sha384-cbc-
des3_ede-caam)
alg: No test for echainiv(authenc(hmac(sha512),cbc(des3_ede))) (echainiv-authenc-hmac-sha512-cbc-
des3_ede-caam)
alg: No test for authenc(hmac(md5),cbc(des)) (authenc-hmac-md5-cbc-des-caam)
alg: No test for echainiv(authenc(hmac(md5),cbc(des))) (echainiv-authenc-hmac-md5-cbc-des-caam)
alg: No test for echainiv(authenc(hmac(sha1),cbc(des))) (echainiv-authenc-hmac-sha1-cbc-des-caam)
alg: No test for echainiv(authenc(hmac(sha224),cbc(des))) (echainiv-authenc-hmac-sha224-cbc-des-caam)
alg: No test for echainiv(authenc(hmac(sha256),cbc(des))) (echainiv-authenc-hmac-sha256-cbc-des-caam)
alg: No test for echainiv(authenc(hmac(sha384),cbc(des))) (echainiv-authenc-hmac-sha384-cbc-des-caam)
alg: No test for echainiv(authenc(hmac(sha512),cbc(des))) (echainiv-authenc-hmac-sha512-cbc-des-caam)
alg: No test for authenc(hmac(md5),rfc3686(ctr(aes))) (authenc-hmac-md5-rfc3686-ctr-aes-caam)
alg: No test for seqiv(authenc(hmac(md5),rfc3686(ctr(aes)))) (seqiv-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 seqiv(authenc(hmac(sha1),rfc3686(ctr(aes)))) (seqiv-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 seqiv(authenc(hmac(sha224),rfc3686(ctr(aes)))) (seqiv-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 seqiv(authenc(hmac(sha256),rfc3686(ctr(aes)))) (seqiv-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 seqiv(authenc(hmac(sha384),rfc3686(ctr(aes)))) (seqiv-authenc-hmac-sha384-rfc3686-
```

```
ctr-aes-caam)
alg: No test for authenc(hmac(sha512),rfc3686(ctr(aes))) (authenc-hmac-sha512-rfc3686-ctr-aes-caam)
alg: No test for seqiv(authenc(hmac(sha512),rfc3686(ctr(aes)))) (seqiv-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 echainiv(authenc(hmac(md5),cbc(aes))) (echainiv-authenc-hmac-md5-cbc-aes-caam-qi)
alg: No test for echainiv(authenc(hmac(sha1),cbc(aes))) (echainiv-authenc-hmac-sha1-cbc-aes-caam-qi)
alg: No test for authenc(hmac(sha224),cbc(aes)) (authenc-hmac-sha224-cbc-aes-caam-qi)
alg: No test for echainiv(authenc(hmac(sha224),cbc(aes))) (echainiv-authenc-hmac-sha224-cbc-aes-caam-
qi)
alg: No test for echainiv(authenc(hmac(sha256),cbc(aes))) (echainiv-authenc-hmac-sha256-cbc-aes-caam-
qi)
alg: No test for authenc(hmac(sha384),cbc(aes)) (authenc-hmac-sha384-cbc-aes-caam-qi)
alg: No test for echainiv(authenc(hmac(sha384),cbc(aes))) (echainiv-authenc-hmac-sha384-cbc-aes-caam-
qi)
alg: No test for echainiv(authenc(hmac(sha512),cbc(aes))) (echainiv-authenc-hmac-sha512-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 echainiv(authenc(hmac(md5),cbc(des3_ede))) (echainiv-authenc-hmac-md5-cbc-des3_ede-
caam-qi)
alg: No test for echainiv(authenc(hmac(sha1),cbc(des3_ede))) (echainiv-authenc-hmac-sha1-cbc-des3_ede-
caam-qi)
alg: No test for echainiv(authenc(hmac(sha224),cbc(des3_ede))) (echainiv-authenc-hmac-sha224-cbc-
des3_ede-caam-qi)
alg: No test for echainiv(authenc(hmac(sha256),cbc(des3_ede))) (echainiv-authenc-hmac-sha256-cbc-
des3_ede-caam-qi)
alg: No test for echainiv(authenc(hmac(sha384),cbc(des3_ede))) (echainiv-authenc-hmac-sha384-cbc-
des3_ede-caam-qi)
alg: No test for echainiv(authenc(hmac(sha512),cbc(des3_ede))) (echainiv-authenc-hmac-sha512-cbc-
des3_ede-caam-qi)
alg: No test for authenc(hmac(md5),cbc(des)) (authenc-hmac-md5-cbc-des-caam-qi)
alg: No test for echainiv(authenc(hmac(md5),cbc(des))) (echainiv-authenc-hmac-md5-cbc-des-caam-qi)
alg: No test for echainiv(authenc(hmac(sha1),cbc(des))) (echainiv-authenc-hmac-sha1-cbc-des-caam-qi)
alg: No test for echainiv(authenc(hmac(sha224),cbc(des))) (echainiv-authenc-hmac-sha224-cbc-des-caam-
qi)
alg: No test for echainiv(authenc(hmac(sha256),cbc(des))) (echainiv-authenc-hmac-sha256-cbc-desi-caam-
qi)
alg: No test for echainiv(authenc(hmac(sha384),cbc(des))) (echainiv-authenc-hmac-sha384-cbc-des-caam-
qi)
alg: No test for echainiv(authenc(hmac(sha512),cbc(des))) (echainiv-authenc-hmac-sha512-cbc-des-caam-
qi)
platform caam_qi: algorithms registered in /proc/crypto
caam_jr 1710000.jr: registering rng-caam
caam 1700000.crypto: caam pkc algorithms registered in /proc/crypto
[...]
```

Crypto algorithms support

Algorithms Supported in the linux kernel scatterlist Crypto API

The Linux kernel contains various users of the Scatterlist Crypto API, including its IPsec implementation, sometimes referred to as the NETKEY stack. The driver, after registering supported algorithms with the Crypto API, is therefore used to process per-packet symmetric crypto requests and forward them to the SEC hardware.

Since SEC hardware processes requests asynchronously, 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 are supported.

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 IPsec and TLS protocols. 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).

CAAM drivers currently supports offloading the following AEAD algorithms:

- "stitched" AEAD: all combinations of { NULL, CBC-AES, CBC-DES, CBC-3DES-EDE, RFC3686-CTR-AES } x HMAC-{MD-5, SHA-1,-224,-256,-384,-512}
- "true" AEAD: generic GCM-AES, GCM-AES used in IPsec: RFC4543-GCM-AES and RFC4106-GCM-AES
- TLS 1.0 record layer encryption using the "stitched" AEAD cipher suite CBC-AES-HMAC-SHA1

Encryption algorithms

The CAAM driver currently supports offloading the following encryption algorithms.

Authentication algorithms

The CAAM driver's ahash support includes keyed (hmac) and unkeyed hashing algorithms.

Asymmetric (public key) algorithms

Currently, RSA is the only public key algorithm supported.

Random Number Generation

caamrng frontend 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.

Table 95. Algorithms supported by each interface / backend

Algorithm name / Backend	Job Ring Interface	Queue Interface	DPSEC Interface
rsa	Yes	No	No
tls10(hmac(sha1),cbc(aes))	No	Yes	Yes
authenc(hmac(md5),cbc(aes))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(sha1),cbc(aes))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(sha224),cbc(aes))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)

Table continues on the next page...

Table 95. Algorithms supported by each interface / backend (continued)

Algorithm name / Backend	Job Ring Interface	Queue Interface	DPSEC Interface
authenc(hmac(sha256),cbc(aes))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(sha384),cbc(aes))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(sha512),cbc(aes))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(md5),cbc(des3_ede))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(sha1),cbc(des3_ede))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(sha224),cbc(des3_ede))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(sha256),cbc(des3_ede))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(sha384),cbc(des3_ede))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(sha512),cbc(des3_ede))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(md5),cbc(des))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(sha1),cbc(des))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(sha224),cbc(des))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(sha256),cbc(des))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(sha384),cbc(des))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(sha512),cbc(des))	Yes (also echainiv)	Yes (also echainiv)	Yes (also echainiv)
authenc(hmac(md5),rfc3686(ctr(aes)))	Yes (also seqiv)	Yes (also seqiv)	Yes (also seqiv)
authenc(hmac(sha1),rfc3686(ctr(aes)))	Yes (also seqiv)	Yes (also seqiv)	Yes (also seqiv)
authenc(hmac(sha224),rfc3686(ctr(aes)))	Yes (also seqiv)	Yes (also seqiv)	Yes (also seqiv)
authenc(hmac(sha256),rfc3686(ctr(aes)))	Yes (also seqiv)	Yes (also seqiv)	Yes (also seqiv)

Table continues on the next page...

Table 95. Algorithms supported by each interface / backend (continued)

Algorithm name / Backend	Job Ring Interface	Queue Interface	DPSEC Interface
authenc(hmac(sha384),rfc3686(ctr(aes)))	Yes (also seqiv)	Yes (also seqiv)	Yes (also seqiv)
authenc(hmac(sha512),rfc3686(ctr(aes)))	Yes (also seqiv)	Yes (also seqiv)	Yes (also seqiv)
authenc(hmac(md5),ecb(cipher_null))	Yes	No	No
authenc(hmac(sha1),ecb(cipher_null))	Yes	No	No
authenc(hmac(sha224),ecb(cipher_null))	Yes	No	No
authenc(hmac(sha256),ecb(cipher_null))	Yes	No	No
authenc(hmac(sha384),ecb(cipher_null))	Yes	No	No
authenc(hmac(sha512),ecb(cipher_null))	Yes	No	No
gcm(aes)	Yes	Yes	Yes
rfc4543(gcm(aes))	Yes	Yes	Yes
rfc4106(gcm(aes))	Yes	Yes	Yes
cbc(aes)	Yes	Yes	Yes
cbc(des3_ede)	Yes	Yes	Yes
cbc(des)	Yes	Yes	Yes
ctr(aes)	Yes	Yes	Yes
rfc3686(ctr(aes))	Yes	Yes	Yes
xts(aes)	Yes	Yes	Yes
hmac(md5)	Yes	No	Yes
hmac(sha1)	Yes	No	Yes
hmac(sha224)	Yes	No	Yes
hmac(sha256)	Yes	No	Yes
hmac(sha384)	Yes	No	Yes
hmac(sha512)	Yes	No	Yes
md5	Yes	No	Yes
sha1	Yes	No	Yes
sha224	Yes	No	Yes
sha256	Yes	No	Yes

Table continues on the next page...

Table 95. Algorithms supported by each interface / backend (continued)

Algorithm name / Backend	Job Ring Interface	Queue Interface	DPSEC Interface
sha384	Yes	No	Yes
sha512	Yes	No	Yes

CAAM Job Ring backend driver specifics

CAAM Job Ring backend driver (*caam_jr*) implements and utilizes the job ring interface (JRI) for submitting crypto API service requests from the frontend drivers (*caamalg*, *caamhash*, *caam_pkc*, *caamrng*) to CAAM engine.

CAAM drivers have a few options, most notably hardware job ring size and interrupt coalescing. They can be used to fine-tune performance for a particular use case.

The option *Freescale CAAM-Multicore platform driver backend* enables the basic platform driver (*caam*). All (non-DPAA2) sub-options depend on this.

The option *Freescale CAAM Job Ring driver backend (SEC)* enables the Job Ring backend (*caam_jr*).

The sub-option *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 sub-option *Job Ring interrupt coalescing* allows the user to select the use of the hardware's interrupt coalescing feature. Note that the driver already performs IRQ coalescing in software, 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 frontend drivers to register their algorithm capabilities with the corresponding APIs. They should be deselected only when the purpose is to perform Crypto API requests in software (on the GPPs) instead of offloading them on SEC engine.

caamhash frontend (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.

caam_pkc frontend (public key / asymmetric algorithms) can be turned off too, if needed.

caamrng frontend (Random Number Generation) may be turned off in case there is an alternate source of entropy available to the kernel.

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:

```
caam algorithms registered in /proc/crypto
caam_jr 1710000.jr: registering rng-caam
caam 1700000.crypto: caam pkc 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 on the corresponding Job Ring:

```
$ cat /proc/interrupts | grep jr
            CPU0       CPU1       CPU2       CPU3
[...]
 78:       1007          0          0          0     GICv2 103 Level     1710000.jr
 79:          7          0          0          0     GICv2 104 Level     1720000.jr
 80:          0          0          0          0     GICv2 105 Level     1730000.jr
 81:          0          0          0          0     GICv2 106 Level     1740000.jr
```

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.

Verifying the 'self test' fields say 'passed' in /proc/crypto

An entry such as the one below means the driver has successfully registered support for the algorithm with the kernel crypto API:

```
name         : cbc(aes)
driver       : cbc-aes-caam
module       : kernel
priority     : 3000
refcnt       : 1
selftest     : passed
internal     : no
type         : givcipher
async        : yes
blocksize    : 16
min keysize  : 16
max keysize  : 32
ivsize       : 16
geniv        : <built-in>
```

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 echainiv(authenc(hmac(md5),cbc(aes))) (echainiv-authenc-hmac-md5-cbc-aes-caam)
alg: No test for echainiv(authenc(hmac(sha1),cbc(aes))) (echainiv-authenc-hmac-sha1-cbc-aes-caam)
[...]
alg: No test for authenc(hmac(sha512),rfc3686(ctr(aes))) (authenc-hmac-sha512-rfc3686-ctr-aes-caam)
alg: No test for seqiv(authenc(hmac(sha512),rfc3686(ctr(aes)))) (seqiv-authenc-hmac-sha512-rfc3686-ctr-
aes-caam)
[...]
```

Examining the hardware statistics registers in debugfs

When using the JRI or QI backend, performance monitor registers can be checked, provided CONFIG_DEBUG_FS is enabled in the kernel's configuration. If debugfs is not automatically mounted at boot time, then a manual mount must be performed in order to view these registers. This normally can be done with a superuser shell command:

```
$ mount -t debugfs none /sys/kernel/debug
```

Once done, the user can read controller registers in /sys/kernel/debug/1700000.crypto/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)

For more information see section "Performance Counter, Fault and Version ID Registers" in the Security (SEC) Reference Manual (SECRM) of each SoC (available on company's website).

Note: for QI backend there is also *qi_congested*: SW-based counter that shows how many times queues going to / from CAAM to QMan hit the congestion threshold.

Kernel configuration to support CAAM device drivers

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 crypto operations to the SEC hardware. Documentation regarding how to set up an IPsec tunnel can be found in corresponding open source IPsec suite packages, e.g. strongswan.org, openswan, setkey, etc. DPAA2-specific section contains a generic helper script to configure IPsec tunnels.

Running OpenSSL

Please see Hardware Offloading with OpenSSL for more details on how to offload OpenSSL cryptographic operations in the SEC crypto engine (via cryptodev).

Executing custom descriptors

SEC drivers have public descriptor submission interfaces corresponding to the following backends:

- JRI: drivers/crypto/caam/jr.c:caam_jr_enqueue()

- QI: drivers/crypto/caam/qi.c:caam_qi_enqueue()
- DPSECI: drivers/crypto/caam/caamalg_qi2.c:dpaa2_caam_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 etc.

dpaa2_caam_enqueue()

Name

dpaa2_caam_enqueue — Enqueue a frame descriptor (FD) into a QMan frame queue. Returns 0 if OK, -EBUSY if QMan driver fails to enqueue the FD for some reason or if congestion is detected.

Synopsis

```
int dpaa2_caam_enqueue(struct device *dev, struct caam_request *req);
```

Arguments

dev: DPSECI device.

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 etc.

Please refer to the source code for usage examples.

Supporting Documentation

DPAA1-specific SEC details - Queue Interface (QI)

DPAA2-specific SEC details - Data Path SEC Interface (DPSECI)

4.2.10 Watchdog

Module loading
Watchdog device driver supports kernel built-in mode.

U-Boot configuration

Run-time 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 configuration options

Kernel configuration tree view options

Kernel configuration tree view options	Description
Device Drivers ---> [*] Watchdog Timer Support ---> [*] Disable watchdog shutdown on close [*] IMX2+ Watchdog	IMX2 watchdog timer

Compile-time configuration options

Option	Values	Default Value	Description
CONFIG_IMX2_WDT	y/n	y	IMX2 watchdog timer

Source Files

The driver source is maintained in the Linux kernel source tree.

Source file	Description
drivers/watchdog/imx2_wdt.c	IMX2 watchdog timer

User space application

The following application will be used during functional or performance testing.

Command name	Description	Package name
watch	watchdog is a daemon for watchdog feeding	watchdog

Verification in Linux

1. Set NFS rootfs. Build a rootfs image that includes watchdog daemon.
2. Set boot 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 a watchdog timeout period. Set this parameter with the proper value depending on your board bus frequency.

   `wdt_period` is inversely proportional to watchdog expiry time, it means, the higher the `wdt_period`, the lower the watchdog expiry time. Therefore, if `wdt_period` is increased to high, watchdog will expiry early.

   NOTE

   When using watchdog as wake-up source with the default Ubuntu root filesystem, add `watchdog-device = /dev/watchdog` to `/etc/watchdog.conf`.

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP 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. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP 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 NXP 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. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer’s applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP, the NXP logo, Freescale, the Freescale logo, CodeWarrior, Layerscape, PowerQUICC, QorIQ, CoreNet, and QUICC Engine are trademarks of NXP B.V. All other product or service names are the property of their respective owners. Arm, Cortex, and TrustZone are registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved.

© NXP B.V. 2019. All rights reserved.

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 18 April 2019

Document identifier: FRWY-LS1046ABSPUG