NXP Semiconductors
Reference Manual

Document Number: LSDK
Rev. 18.09, 09/2018

Layerscape Software Development Kit 18.09 Documentation

Contents

Chapter 1 About this document

About QorIQ Layerscape Software Development Kit (LSDK)

LSDK is a complete Linux kit for NXP QorIQ ARM-based SoC's and the reference and evaluation boards that are available for them.

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

- NXP firmware components including:
  - PPA, a resident EL3 privilege secure firmware for ARMv8A.
  - NXP peripheral firmware components for DPAA1, DPAA2, and PPFE.
- NXP boot loaders. Two are offered:
  - U-Boot, based on denx.de plus patches.
  - UEFI, based on TianoCore.
- NXP Linux kernel, based on kernel.org upstream plus patches.
- NXP added user space components.
- Ubuntu standard user space file set (user land), including compilers and cross compiler.

The use of Ubuntu user land is what makes LSDK a hybrid. It is not entirely an Ubuntu distribution because it uses an NXP kernel, but it still uses Ubuntu user space files. This hybrid is possible because NXP ARM SoC's are standards-based so programs like bash and thousands of others run without being recompiled.

The benefit of using Ubuntu user land is the easy availability of thousands of standard Linux user space packages. The experience of using the LSDK is similar to using Ubuntu, but the kernel, firmware, and some special NXP packages are managed separately.

NOTE

For the most up-to-date version of this documentation set, see the Knowledge Center for Layerscape Software Development Kit

Accessing LSDK

LSDK is distributed via git. See https://lsdk.github.io/.

There are two ways to use the LSDK, as an integration and as a source of individual components.

LSDK as an integration

Using the link above, notice the `flexbuild` component. You can clone it and run a script to create and install LSDK onto a mass storage device as an integration, ready for use on an NXP reference or evaluation board. You can build NXP components from source using a script called flex-builder or install from binaries of NXP components using flex-installer. See Layerscape SDK user guide on page 46.

LSDK as components

The same link shows git repositories for individual components, for example the LSDK Linux kernel. If you clone and examine this git, you will see a conventional kernel source tree. You can compile the kernel using `make` in the normal way, like a kernel.org kernel. However, notice the configuration fragment in a`rch/arm64/configs`. See Linux kernel on page 318.

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

LSDK git tags

LSDK git repositories use git tags to indicate component revisions that have been release tested together. Use the `git tag` command to examine them and chose a tag to check out.

LSDK Relies on Mass Storage Devices

Ubuntu user land is very convenient for evaluation because it is possible to use the command `apt-get install` on the standard Ubuntu components you need. It also provides native development tools.

But this richness means that the user space file is large, too large for RAM disks.

Therefore, LSDK requires installation to and use of a mass storage device such as

- SD card
- USB flash drive
- USB hard drive
- SATA drive, spinning, or SSD (for boards with a SATA port)
- eMMC flash (when available on board)

LSDK provides scripts that populate a mass storage device with the needed files. These scripts can run on a Linux PC. It is especially simple to use an SD card or USB flash drive because they are the easiest to move between a Linux PC and the NXP board.

Chapter 2
Acronyms and abbreviations

Term	Definition
AH	Authentication Header (RFC 4302) – a network protocol designed to provide authentication services in IPv4 and IPv6.
AIOP	Advanced I/O Processor
ACL	Access Control List
AMP	Asynchronous Multi-Processing, running multiple operating system images on different processors of the same machine without virtualization.
API	Application Programming Interface
ARP	Address Resolution Protocol
ASF	Application Specific Fast Path – a software abstraction layer providing different network services (e.g. firewall, IPSec, IP reassembly, forwarding, etc.) to top level applications implemented in various environments (such as Linux kernel, Linux user space, etc.). These services may be hardware accelerated or not depending on the hardware the ASF is running on. The hardware offloading is provided transparently to the top level applications.
BMan	Buffer Manager – a DPAA hardware block performing buffer and buffer pool management.
BPID	(BMan) Buffer Pool ID
CAAM	Cryptographic Acceleration and Assurance Module
CCSR	Configuration and Control Status Register
CEETM	Customer Edge Egress Traffic Management
CPU	Central Processing Unit, also known more generally as "Processor"
DCD	Device Configuration Data
DCE	Data Compression/Decompression Engine
DMA	Direct Memory Access
DPAA	Data Path Acceleration Architecture (First Generation)
DPAA2	Data Path Acceleration Architecture (Second Generation)
DPC	Data Path Configuration File
DPDK	Data Plane Development Kit
DPDMUX	Data Path De-multiplexer
DPL	Data Path Layout

Table continues on the next page...

Table continued from the previous page...

Term	Definition
DPNI	DPAA2 object modeling a network interface
DPRC	Data Path Resource Container
DSK	Device Secret Key
DTB	Device Tree Blob—the binary representation of device trees
DTS	Device Tree Syntax—the textual representation of device trees
DUT	Device Under Test
ESBC	External Secure Boot Code
ESP	Encapsulating Security Payload (RFC 4303) – a network protocol designed to provide a mix of security services in IPv4 and IPv6.
EVB	Edge Virtual Bridge
FDB	Forwarding Data Base
FMan	Frame Manager – a DPAA hardware block performing frame processing management.
FMC	Frame Manager Configuration application
FRA	Freescale RMan Application
FUID	Freescale Unique ID
GPP	General Purpose Processor
Guest/VM	The term 'Guest' is used for Linux running inside the virtual machine(s) that are in turn running over Host Linux operating system.VM and Guest have been used interchangeably in this guide.
GUEST_CONSOLE_TELNET_PORT	Telnet port for accessing guest console of VM.
HAL	Hardware Abstraction Library
HIF	Host Interface
HSM	Hardware security modules
IBR	Internal Boot ROM
IP_ADDR_BRD	This term is used for LS1088ARDB and LS2088ARDB IP address.
IP_ADDR_IMAGE_SERVER	This term is used for IP address of the machine on which all the software images are kept.
IPC	Inter-Process Communication, can be interpreted as being communication between distinct application execution flows or between distinct hardware processing units.

Table continues on the next page...

Table continued from the previous page...

Term	Definition
inbound (traffic)	Encrypted traffic which is coming from an unprotected interface. This traffic will be terminated on the CPU.
IPFwd	IPv4 Forward
IPSec	IP Security – a communication standard defined and refined by several public RFCs (such as RFC-2401 and RFC-4301) where hosts exchange encrypted IP data packets.
IPSec Tunnel	A communication convention between two network gateways to IPSec process certain network traffic in a particular way. An IPSec tunnel has two endpoints (which are the gateways), a clearly delimited set of encryption and authentication methods, keys, encapsulation headers and security policies, which define the traffic that is sent through the tunnel.
ISBC	Internal Secure Boot Code
ISR	Interrupt Status Register
ITF	Intent to Fail
ITS KASLR	Intent to Secure Kernel Address Space Layout Randomization
KVM	Kernel-based Virtual Machine - A Linux kernel module that allows a user space program access to the hardware virtualization features of NXP processors.
LIODN	Logical I/O Device Number
MC	Management Complex
NAT	Network Address Translation
ODP	OpenDataPlane
OEM	Original Equipment Manufacturer
OS	Operating System
OUID	OEM Unique ID
outbound (traffic)	Clear traffic which is coming from a software application which generates traffic that must be encrypted and forwarded via an unprotected interface.
PAMU	Peripheral Access Management Unit
PBL	Pre-Boot Loader
PCD	Parse, Classify, Distribute – a software architecture concept in NXP DPAA drivers which allows the user to configure the DPAA hardware (FMan) to do frame parsing, classification or distribution on a series of frame queues.

Table continues on the next page...

Table continued from the previous page...

Term	Definition
PDCP	Packet Data Convergence Protocol – It is one of the layers of the Radio Traffic Stack in UMTS/LTE and performs IP header compression and decompression, transfer of user data and maintenance of sequence numbers for Radio Bearers which are configured for lossless serving radio network subsystem (SRNS) relocation.
PME	Pattern Matcher Engine
PKCS	Public-Key Cryptography Standards
QEMU	Quick EMUlator - A hosted hypervisor that performs hardware virtualization.
QBMan	Queue and Buffer Manager
QMan	Queue Manager – a DPAA hardware block performing frame queue management.
RC	Route Cache
RCW	Reset Configuration Word
RFC	Request for Comments – a public document which describes a software standard.
RDB	Reference Design Board
SA	Security Association – a data record, defined by RFC 4301, which stores the information related to the IPSec processing needed for a specific network traffic type (such as encryption/decryption keys and algorithms, traffic endpoints description, authentication algorithms, and so on).
SAD	Security Association Database – the database holding all the valid SAs in a system.
SDK	Software Development Kit
SEC	Security Engine Coprocessor – a DPAA hardware block performing cryptographic acceleration and offloading hardware.
SFP	Secure Fuse Processor
SIP DIP	Source Internet Protocol and Destination Internal Protocol
SKMM	Secure Key Management Module
SMP	Symmetric Multi-Processing, running an operating system image on multiple CPUs simultaneously.
SNVS	Secure Non-Volatile Storage
SoC	System on a Chip, a chip integrating one or more processors and on-chip peripherals.
SP	Security Policy – a set of rules that network traffic must comply with in order to be eligible for IPSec processing.
SPD	Security Policy Database – the database storing all the SPs in a system.
SRE	Stateful Rule Engine

Table continues on the next page...

Table continued from the previous page...

Term	Definition
SRK	Super Root Key
SRKH	Super Root Key Hash
STP	Spanning Tree Protocol
SUI	String Under Inspection
TFTP_BASE_DIR	Base directory of TFTP server where all the images are kept.
TLB	Translation Lookaside Buffer
TTL	Time To Live
UDP	User Datagram Protocol
UID	Unique Device ID
UIO	User space I/O
USDPAA	User Space Data Path Acceleration Architecture
VEB	Virtual Ethernet Bridge
VEPA	Virtual Ethernet Port Aggregator
VID	Voltage IDentifier
WRIOP	Wire Rate I/O Processor

Chapter 3 Release notes

3.1 What's New

What's New in LSDK 18.09

Highlights

- LTS 4.9.124 and LTS 4.14.67 update
- EdgeScale enabled on LS1088A and LS2088A, and more functional updates
- OpenSSL v1.1 validated
- OP-TEE and fuse provisioning enabled on LS1088A and LS2088A
- Support of IMA-EVM on LTS 4.14
- Support of LS1012AFRWY with 1GB DDR
- Support of LS1088ARDB-PB
- Support of PREEMPT_RT on LS1046A and LS1088A on LTS 4.14
- Support of VPP in DPDK
- Includes several software fixes. Refer to Fixed, Open, and Closed Issues on page 40 which has a list of all fixed issues
- Includes additional workarounds for Chip Errata: A-008708, A-011379

See list of changes in below

Ubuntu Userland

- OpenSSL v1.1 validated

Linux Kernel Core, Virtualization

- LTS kernel 4.9.124 update
- LTS kernel 4.14.67 update
- Support of PREEMPT_RT on LS1046A and LS1088A on LTS 4.14

Linux Kernel Drivers

- DPAA1 Ethernet: 10Gbase-KR on LS1046A, jumbo frames enabled on LS1043
- DPAA2 CAAM: update to the latest MC f/w dpseci API, skcipher API and IV in DMAable buffer
- Support of IMA-EVM on LTS 4.14

Data Plane Development Kit (DPDK)

- Performance tuning
- Support of 3DES algorithm in ipsecgw
- Support of PCI to DPAA interworking
- Support of VPP

Virtualization - OVS-DPDK

- No change since the last LSDK

U-Boot Boot Loader

- Environment variables of MAC address to set the DPMAC MAC address for use by Linux
- Support of LS1012AFRWY with 1GB DDR

Unified Extensible Firmware Interface (UEFI)

- No change since the last LSDK

EdgeScale – Edge Compute

- Support of LS1088A and LS2088A
- Secure Manufacturing
    - No change since the last LSDK
- Secure Provisioning
    - Solution signature verification before install on the device
    - Signature and device certificate using OEM's root CA
- EdgeScale Dashboard for Users
    - Device management
        - Secure device enrollment
        - EdgeScale client agents in a debian pkg
        - Support of call home to get service endpoints for device
    - Application management
        - Support app/container signature
        - Support integration of private container repo
        - Support enrollment of new app

Other Tools and Utilities

- Flexbuild
    - Docker installation using docker-ce
    - OP-TEE disabled by default
    - Support of IMA-EVM
    - Support of OpenSSL v1.1, removal of OpenSSL v1.0
- FMC: support for cleanup of last FMC settings
- Linux Fman-IM driver as module
- MC upgrade to 10.10.0
- Platform Security
    - Fuse provisioning on LS1088A, LS2088A
    - OP-TEE client: support of LS1088A, LS2088A
    - PKCS#11 Library: support for ECC 256/384 keys
    - Support of IMA-EVM
- PPA
    - No change since the last LSDK
- RCW

— Support of LS1088ARDB-PB

What's New in LSDK 18.06

Highlights

- Spectre and Meltdown security patches included on LTS-4.9, LTS-4.14 and PPA. Further information will be available on “Security Updates” https://freescale.sdlproducts.com/LiveContent/web/ui.xql?action=html&resource=publist_home.html
- Removal of ODP from LSDK 18.06 and future releases.
- U-Boot v2018.03 update
- Ubuntu userland 18.04 update
- Secure provisioning being integrated into EdgeScale
- Support of FRWY-LS1012A board
- Support of Data Compression Engine
- Support of Direct device assignment using VFIO for DPDK in VM on DPAA2
- Includes several software fixes. Refer to Fixed, Open, and Closed Issues on page 40 which has a list of all fixed issues
- Includes additional workarounds for Chip Errata: A-010843

See list of changes in below

Ubuntu Userland

- Ubuntu userland 18.04 update
- Toolchain: gcc: Ubuntu/Linaro 7.3.0-16ubuntu3~18.04, glibc-2.27, binutils-2.30-0, gdb-8.1
- Libvirt 4.0
- QEMU 2.11

Linux Kernel Core, Virtualization

- LTS kernel 4.9.105 update
- LTS kernel 4.14.47 update
- Spectre and Meltdown security patches
- Enabling VFIO no-IOMMU being documented

Linux Kernel Drivers

- DPAA2 Ethernet: CEETM, MQPRIO qdisc support, configurable RX hashing key
- FRWY-LS1012A: CAAM, DDR, DUART, I2C, PCIe, PFE ethernet, SAI/I2S, SD, USB, Watchdog

Data Plane Development Kit (DPDK)

- Support of Direct device assignment using VFIO for DPDK in VM [DPAA2 processors]
- PFE with DPDK [LS1012A]
- Running DPDK over docker
- Flow director (classification) on [DPAA2 processors]
- UEFI boot [DPAA1 & DPAA2 processors]
- QDMA driver support [DPAA2 processors]

Virtualization - OVS-DPDK

- No change since the last LSDK

Open Data Plane (ODP)

- ODP support being removed from LSDK 18.06 and future releases

U-Boot Boot Loader

- U-Boot v2018.03 update
- FRWY-LS1012A: Non-secure boot, Secure Boot, Clock, DDR, DUARt, I2C, PCIe, QSPI, SD, USB, Watchdog
- Support of LS2088A 0.9v part

Unified Extensible Firmware Interface (UEFI)

- Support of AIOP application

EdgeScale – Edge Compute

- Secure Manufacturing
  - Tools being available for Secure Fuse provisioning
- Secure Provisioning
  - Secure library integration
- EdgeScale Dashboard for Users
  - Device management
    - Agent to report more 'status' information of devices
    - Device Logs on cloud
    - OTA: firmware update [LS1012A]
    - Support of un-enrolling device
  - Application management
    - App arguments at deployment time
    - Support of public and private apps

Other Tools and Utilities

- Support of Data Compression Engine in user space
- Flexbuild
  - Disabling ODP
  - Ubuntu-18.04 arm32 and arm64 filesystem support
  - Programming separate images into QSPI and NOR flash
- MC upgrade to 10.8.0
- OP-TEE client: FRWY-LS1012A support
- PKCS#11 Library
  - Support for Multithreaded Applications
  - Integration with PKCS#11 OpenSSL Engine from OpenSC/libp11
- PPA
  - HW generated key for OP-Tee
  - Support of FRWY-LS1012A
- RCW
  - Support of FRWY-LS1012A

  - — Support of LS2088A ar core frequency 2.1GHz
- Restool: Removal of DPNI_OPT_SINGLE_SENDER and 16 TCs

What's New in LSDK 18.03

Highlights

- Switching the dual kernel version to LTS 4.9.79 and LTS 4.14.16. LTS 4.4 is not tested in this release and under maintenance mode
- U-Boot v2017.11 update
- LS2084A and LS2088A top bin and non-E part
- Direct device assignment in guest kernel on LTS 4.14 [DPAA2 processors]
- Support for edge compute on EdgeScale, including secure manufacturing, secure keys, EdgeScale dashboard for users and application management
- MC upgrade to 10.6.0
- Python scripts to generate RCW binaries for LS1012A, LS1088A and LS2088A
- Support for DPDK 17.11 as base and OVS 2.9
- Includes several software fixes. Refer to Fixed, Open, and Closed Issues on page 40 which has a list of all fixed issues
- Includes additional workarounds for Chip Errata: A-008851, A-009611, A-009668, A-010131, A-010477, A-010843, A-011026

Ubuntu Userland

- No change since the last LSDK

Linux Kernel Core, Virtualization

- LTS kernel 4.9.79 update
- LTS kernel 4.14.16
- Direct device assignment in guest kernel on LTS 4.14 [DPAA2 processors]
- QEMU: MC portal emulation

Linux Kernel Drivers

- DPAA2 CAAM: generic gcm(aes), IPsec GCM - rfc4106
- DPAA2 Ethernet: XDP, L2Switch driver update to switchdev version
- USB: U1/U2 mode in host

Data Plane Development Kit (DPDK)

- Support of DPDK 17.11 as base
- Support of LTS 4.14
- AIOP cmdif support
- Ethernet poll mode driver with push mode queues, Crypto - Scatter Gather support, Eventde driver, tail drop using WRED - CGR [DPAA1 processors]
- IPSEC protocol offload support
- KNI support
- PKTGEN 3.4.8

Virtualization - OVS-DPDK

- Support of OVS 2.9

U-Boot Boot Loader

- U-Boot v2017.11
- Support for IFC and EMMC switch support in qixis [LS1088A]

Unified Extensible Firmware Interface (UEFI)

- No change since the last LSDK

EdgeScale – Edge Compute

- Secure Manufacturing
  - Tools available for Secure Fuse provisioning [LS1012A, LS1043A, LS1046A]
- Secure Keys [LS1046A]
  - Support of API's to import/generate RSA keys securely
  - Support of PKCS#11 interface for Ssigning operations
  - Support of OPENSSL engine to access these keys
- EdgeScale Dashboard for Users
  - Device management
    - Secure device enrolment
    - Secure key/certificate provisioning
    - OTA: firmware update [LS1043, LS1046]
    - Device status monitoring
  - Application management
    - Dynamic deployment of container-based applications

Other Tools and Utilities

- AIOPSL
  - AIOP boot error handling, error frame handling, TX buffer layout
  - API for configuring timestamp passing behavior inside a recycle path flow
  - ASA opaque
  - User-defined exception handler
- MC upgrade to 10.6.0
- RCW
  - LS2084A and LS2088A top bin and non-E part
  - Python scripts to generate RCW binaries for LS1012A, LS1088A and LS2088A

What's New in LSDK 17.12

Highlights

- Support for 2.5G PFE MAC [LS1012A]
- MC upgrade to 10.4.0
- Support for DPDK 17.05.02 as base and OVS 2.8
- Support for sleep (LPM20) [LS1088A, LS2088A]
- Integration of Open Portable Trusted Execution Environment (OP-TEE) [LS1046A]

- Includes several software fixes. Refer to Fixed, Open, and Closed Issues on page 40 which has a list of all fixed issues
- Includes additional workarounds for Chip Errata: A-008708, A-008428, A-008822, A-009007, A-009668, A-009611, A-009810, A-010151, A-010554, A-010650

Linux Kernel Core, Virtualization

- LTS kernel 4.4.98 update
- LTS kernel 4.9.62 update

Linux Kernel Drivers

- DPAA2 CAAM: Hashing
- DPAA2 Ethernet: Priority Flow Control
- OP-TEE driver [LS1046A]
- PFE: 2.5G MAC [LS1012A]

U-Boot Boot Loader

- HW load/store prefetch being disabled
- PFE: 2.5G MAC [LS1012A]

Unified Extensible Firmware Interface (UEFI)

- MC High Mem Support

Data Plane Development Kit (DPDK)

- DPDK 17.05.02 as base
- DPDK on docker [DPAA2 processors]
- UEFI support on LS2088A

Virtualization - OVS-DPDK

- OVS 2.8

Other Tools and Utilities

- MC upgrade to 10.4.0
- OP-TEE client [LS1046A]
- PPA: sleep (LPM20) [LS1088A, LS2088A], OP-TEE OS binary [LS1046A]
- Restool Bourne shell (sh) compatible

What's New in LSDK 17.09-Update-103017

Highlights

- LS1012A r1.0 and r2.0, LS1012A RDB at core frequency 1GHz by default
- MC upgrade to 10.3.4 to support 1000base-X and SGMII phyless
- Integration of PPFE driver
- Integration of CAAM DMA driver
- Including software fixes for LS2088A secure boot, MC failure to clear memory after use on LS1088A, fixup of MAC address in DPC on LS1088A

Processor and Board Support

- LS1012A r1.0 and r2.0

- LS1012A RDB at core frequency 1.0GHz by default

Linux Kernel Drivers

- LS1012A: Crypto driver supporting SEC 5 (CAAM), CAAM DMA, DDR, DUART, DSPI, eSDHC, I2C, PCIe RC, PFE Ethernet (Packet Rx/Tx), PHY support: RGMII & SGMII, Power management, QSPI, SAI/I2S, SATA, UART, USB 2/3 mass storage, Watchdog

U-Boot Boot Loader

- LS1012A: Non-secure boot, Secure Boot (silicon r1.0), Clock, CPLD, DDR4, DSPI, eSDHC, I2C, Generic Timers, PCIe, Primary Protected Application (PPA) firmware integration, QSPI, SATA, UART

Other Tools and Utilities

- MC upgrade to 10.3.4 to support 1000base-X
- Supporting multiple versions of docker image in Flexbuild

What's New in LSDK 17.09

Highlights

- LTS kernel 4.9.35, including KASLR
- U-Boot 2017.07
- MC 10.3.2 update
- Support for DPDK 17.05 as base and OVS 2.7
- Flexbuild to support dual kernel build for 4.4 and 4.14
- Includes several software fixes. Refer to Fixed, Open, and Closed Issues on page 40 which has a list of all fixed issues
- Includes additional workarounds for Chip Errata: A-007815, A-007997, A-010053, A-010151, A-010477, A-010571

Processor and Board Support

- LS1043ARDB-PD

Linux Kernel Core, Virtualization

- LTS kernel 4.9.35, including KASLR
- LTS kernel 4.4.80 update
- LXD and LXD-Bridge

Linux Kernel Drivers

- DPAA2 Ethernet: 10G-base-KR
- RTC: adding PCF85263
- TMU on LS1088A

Data Plane Development Kit (DPDK)

- DPDK 17.05 as base
- Support for MC 10.3.x

Virtualization - OVS-DPDK

- OVS-DPDK 2.7

U-Boot Boot Loader

- U-Boot 2017.07
- Chain of trust with confidentiality as part of distro boot

- LS1088A: QSPI boot, SD secure boot

Unified Extensible Firmware Interface (UEFI)

- Adding kernel 4.14 support
- KASLR support
- L2 cache Prefetch enable/disable support on LS2088A
- Ubuntu Distro boot support.
- USB 3.0 on LS2088A

Other Tools and Utilities

- AIOPSL: IPv6 Reassembly Atomic Fragment, QoS
- FLIB: AES-CTR algorithm, AES-GCM algorithm
- MC 10.3.2 update
- Flexbuild
    - Dual Kernel build for 4.4 and 4.14
    - Enhanced component's repository management to support single repository in one single command
    - Encapsulation and decapsulation feature for secure boot
    - Removing dpdk-extras repository
    - Adding lttng-modules repository
    - Renaming `flex_installer_<arch>.itb` to `flex_linux_<arch>.itb`

What's New in LSDK 17.06

Highlights

- LTS kernel 4.4.65, including KASLR
- U-Boot 2017.03
- Unified memory map
- Unified Extensible Firmware Interface (UEFI) Spec 2.6 on LS1043A RDB, LS1046A RDB and LS2088A RDB
- Ubuntu host 16.04, root filesystem and toolchain 5.4 verified, not shipped in this release
- Flexbuild to build component and generate the boot firmware, flex_installer.itb and the Ubuntu userland containing the specified packages and applications
- Integrating LS1088A BSP v0.4
- LS2088A r1.1 silicon
- MC 10.2.2 update
- FLIB update
- Release on https://lsdk.github.io/
- Includes several software fixes. Refer to Fixed, Open, and Closed Issues on page 40 which has a list of all fixed issues
- Includes additional workarounds for Chip Errata: A-010160, A-010679, A-010840

Processor and Board Support

- LS1088A r1.0 and Rev. B RDB
- LS2088A r1.0, LS2088A r1.1 and Rev. F RDB

Ubuntu Userland

- Ubuntu host 16.04
- Toolchain: gcc: Ubuntu/Linaro 5.4.0-6ubuntu1~16.04.4, glibc-2.26.1, binutils-2.23-0, gdb-7.11.1
- Linux Containers (LXC)
- QEMU 2.5

Linux Kernel Core, Virtualization

- LTS kernel 4.4.65, including KASLR

Linux Kernel Drivers

- LS1088A: DUART, DDR4, I2C, PCIe, SATA, USB, SD, MMC, NAND, Networking support, SEC
- CAAM: RSA form 1/2/3, TLS 1.0

Data Plane Development Kit (DPDK)

- Integrating LS1088A

Virtualization - OVS-DPDK

- Integrating LS1088A

U-Boot Boot Loader

- U-Boot 2017.03
- Unified memory map
- LS1088A: DUART, DDR4, I2C, PCIe, SATA, USB, SD, MMC, NAND flash, Networking support, Boot from SD
- LS2088A: QSPI boot

Unified Extensible Firmware Interface (UEFI)

- LS1043A, LS1046A, LS2088A
- Spec 2.6
- DDR4, DUART, DSPI, GPIO, I2C, IFC, PCIe, RTC, SATA, SD, Networking support, Watchdog
- PPA integration
- SMP Linux boot via EFI_STUB on SD card
- PXE boot via PCIe and DPAA interfaces
- QSPI boot

Other Tools and Utilities

- LS1088A: Primary Protected Application (PPA) firmware
- MC 10.2.2 updates
- FLIB: CAPWAP DTLS, L2 header copy
- Flexbuild to build component and generate the boot firmware, flex_installer.itb and the Ubuntu userland containing the specified packages and applications

3.2 Components

Overall

- Ubuntu userland

- Linux Kernel and Virtualization
- Linux Kernel Drivers
- Data Plane Development Kit (DPDK)
- Virtualization - OVS-DPDK
- U-Boot Boot Loader
- Unified Extensible Firmware Interface (UEFI)
- EdgeScale – Edge Compute
- Other Tools and Utilities

Ubuntu Userland

- Ubuntu host 18.04
- Toolchain: gcc: Ubuntu/Linaro 7.3.0-16ubuntu3~18.04, glibc-2.27, binutils-2.30-0, gdb-8.1
- Libvirt 4.0
- Linux Containers (LXC)
- OpenSSL 1.1
- QEMU 2.11

Linux Kernel Core and Virtualization

- LTS kernel 4.9.124, including KASLR
- LTS kernel 4.14.67, including KASLR
- ARM Cortex-A7 (AARCH32), Cortex-A53 and Cortex-A72 (AARCH64), Little Endian (default)
- 32-bit effective kernel addressing [Cortex-A53, Cortex-A72]
- 64-bit effective addressing [Cortex-A53, Cortex-A72]
- Direct device assignment in guest kernel [DPAA2 processors]
- Huge Pages (hugetlbfs)
- KVM and Containers
- LXD and LXD-Bridge
- PREEMPT_RT on LTS 4.14 [LS1046A, LS1088A]

Linux Kernel Drivers

- 10Gbase-KR
- Customer Edge Egress Traffic Management (CEETM)
- Crypto driver via SEC 5 & 6 (CAAM)
- CAAM DMA
- Display Control Unit (DCU) and HDMI [LS1021A]
- DUART, DSPI, I2C
- Edge Virtual Bridge (EVB) [DPAA2 processors]
- Ethernet DPAA [DPAA1 processors]
- Ethernet DPAA2 [DPAA2 processors]
- Ethernet eTSEC (gianfar) [LS1021A]
- Frame Manager (FMan) [DPAA1 processors]

- GIC-400, GIC-500, GIC-ITS
- IEEE1588
- IMA-EVM on LTS 4.14
- Integrated Flash Controller (IFC) NOR and NAND flash
- LPUART [LS1021A, LS1043A]
- Management Complex Bus [DPAA2 processors]
- MDIO
- Multiprocessor Interrupt Controller (MPIC)
- Open Portable Trusted Execution Environment (OP-TEE) [Excpet for LS1021A]
- PCIe Root Complex and Endpoint, MSI
- PFE ethernet [LS1012A]
- Platform DMA
- PHY support: RGMII, SGMII, XFI and XAUI
- Power Management (PM) – CPU hotplug (PH20), CPU idle (PW15/20), Sleep (LPM20), Deep sleep (LPM35), Auto-Response, Dynamic Frequency Scaling (DFS), Thermal Monitor, Power Monitor (board specific)
- Queue Manager (QMan) and Buffer Manager (BMan) [DPAA1 processors]
- QUICC Engine UART, TDM, HDLC, PPPoHT
- SAI/I2S [LS1012A]
- SATA
- Secured Digital Host Controller (eSDHC) and SD/MMC support
- System Memory Management Unit (SMMU) [ARM processors]
- Universal Serial Bus (USB) 2.0 and 3.0
- User space IO
- Virtual Function I/O (VFIO) - mmap PCI sources [Except for LS1021A]
- Watchdog Timers

Data Plane Development Kit (DPDK) [LS1012A, LS1043A, LS1046A, LS1088A, LS2088A]

- Support of DPDK v17.11 as base
- Flow director (classification) [DPAA2 processors]
- Following DPDK Applications have been verified
    - l2fwd
    - l3fwd
    - l2fwd_crypto
    - ipsecgateway
- Direct device assignment using VFIO for DPDK in VM [DPAA2 processors]
- DPDK with UEFI boot
- DPDK on docker
- AIOP cmdif
- IPSEC protocol offload

- KNI support
- PFE with DPDK [LS1012A]
- PKTGEN 3.4.8
- QDMA driver [DPAA2 processors]
- Vector Packet Processing

Virtualization - OVS-DPDK

- OVS-DPDK 2.9
- OVS-DPDK working with vhost-virtio interfaces
- DPDK working in Virtual Machine

U-Boot Boot Loader

- U-Boot: 2018.03
- Unified memory map
- On ARM platforms, the U-Boot image includes the device tree
- Non-secure and Secure Boot (ESBC)
- Primary Protected Application (PPA) firmware integration. See PPA features in “Other Tools …” below
- Boot from NOR, NAND flash, QSPI, SDHC
- CodeWarrior debug patch for U-Boot
- Clock, CPLD, DUART, DDR4, DSPI, eSDHC, GIC-400, GIC-500, I2C, OCRAM, PCIe, USB 2 & 3, SATA, UART
- DCU, eMMC 4.5, I2C3, LPUART, QSPI [LS1021A, LS1043A, LS1046A]
- HW load/store prefetch being disabled
- IFC access to NOR and NAND flash
- Networking support using eTSEC, FMAN Independent Mode, DPAA2 networking or PFE
- Voltage ID (board specific)

Unified Extensible Firmware Interface (UEFI) [LS1043A, LS1046A, LS2088A]

- Spec 2.6
- AIOP application
- DDR4, DUART, DSPI, GPIO, I2C, IFC, PCIe, RTC, SATA, SD, Networking support, Watchdog, USB 3.0
- KASLR
- MC High Mem support
- PPA integration
- SMP Linux boot via EFI_STUB on SD card
- PXE boot via PCIe and DPAA interfaces
- QSPI boot
- Ubuntu Distro boot

EdgeScale – Edge Compute

- Secure Manufacturing
    - Tools available for Secure Fuse provisioning
- Secure Provisioning

    - Secure library integration
    - Secure enrollment client downloading, verifying and installing the latest firmware (Linux) on the device
- Secure Keys
    - Support of API's to import/generate RSA keys securely
    - Support of PKCS#11 interface for Ssigning operations
    - Support of OPENSSL engine to access these keys
- EdgeScale Dashboard for Users
    - Device management
        - Agent to report more 'status' information of devices
        - Cert-agent in insecure mode
        - Device Logs on cloud
        - Device status monitoring
        - OTA: firmware update [
        - Secure device enrolment
        - Secure key/certificate provisioning
        - Support of call home to get endpoint from device
        - Un-enrolling device
    - Application management
        - App arguments at deployment time
        - Dynamic deployment of container-based applications
        - Support of public and private apps

Other Tools and Utilities

- AIOPSL [DPAA2 processors]
- Convenience scripts to create and manage common objects like network interfaces. There scripts are packaged in ls2-scripts tarball
- Data Compression Engine in user space
- DPAA2 resource container and object management tool (RESTOOL)
- Flexbuild to build component and generate the boot firmware, flex_linux.itb and the Ubuntu userland containing the specified packages and applications
- FLIB/RTA - SEC descriptor creation library [all processors with SEC 5]
- Frame Manager Configuration Tool (FMC) [DPAA1 processors]
- Frame Manager Ucode [DPAA1 processors]
- Management Complex (MC) Firmware version 10.10.0 – binary only, supporting DPAA2 resource containers and network objects, Resource Manager and Link Manager, DPDMUX basic configurations
- OpenSSL offload - includes TLS Record Layer and Public Key offload
- Platfform Security
    - OP-TEE client [Except for LS1021A]
    - PKCS#11 Library
    -
        - Dynamic deployment of container-based applications

    - Support of public and private apps
    - Support for Multithreaded Applications
    - Integration with PKCS#11 OpenSSL Engine from OpenSC/libp11
- PME Tools [DPAA1 processors]
- Primary Protected Application (PPA) firmware [Except for LS1021A]: Power Management, OP-TEE OS binary [Except for LS1021A]
- Python scripts to generate RCW binaries

3.3 Feature Support Matrix

The following tables show the features that are supported in this release. Refer to the legend below to decipher the entries.

Legend:

- **Y** - Feature is supported by software
- **/** - Feature is not supported by software
- **na** - Hardware feature is not available

Table 1. Key Features

Feature	LS1012A	LS1021A	LS1043A	LS1046A	LS1088A	LS2088A
32-bit Userspace, BE	/	/	/	/	/	/
64-bit Userspace, BE	Y	na	Y	Y	/	/
32-bit Userspace, LE	Y	Y	Y	Y	/	/
64-bit Userspace, LE	Y	na	Y	Y	Y	Y
36b phys mem	na	Y	na	na	na	na
40b phys mem	Y	na	Y	Y	Y	Y
AIOPSL	na	na	na	na	Y	Y
ASF	/	/	/	/	/	/
Data Plane Development Kit (DPDK)	Y	/	Y	Y	Y	Y
EdgeScale - Edge Comupte	Y	/	Y	Y	/	/
Hugetlbfs	Y	Y	Y	Y	Y	Y
Open Data Plane (ODP)	/	/	/	/	/	/
Open Portable Trust Execution Environment (OP-TEE)	Y	/	/	Y	/	/
Secure Boot	Y	Y	Y	Y	Y	Y
Unified Extensible Firmware Interface (UEFI)	/	/	Y	Y	/	Y
USDPAA Applications	na	na	/	/	na	na

Table 2. Virtualization

Feature	LS1012 A	LS1021 A	LS1043 A	LS1046 A	LS1088 A	LS2088 A
KVM/QEMU	Y	Y	Y	Y	Y	Y
LXC	Y	Y	Y	Y	Y	Y
Libvirt	Y	Y	Y	Y	Y	Y
VFIO	na	na	na	na	Y	Y
Docker	Y	/	Y	Y	Y	Y

Table 3. Linux Applications

Feature	LS1012 A	LS1021 A	LS1043 A	LS1046 A	LS1088 A	LS2088 A
Linux IPFwd	Y	Y	Y	Y	Y	Y
Linux IPSec	/	/	/	/	/	/
Linux Termination	Y	Y	Y	Y	Y	Y
Linux NAS	/	/	/	/	/	/
Linux RAID	/	/	/	/	/	/
Linux SATA	/	/	/	/	/	/

Table 4. Linux Kernel Drivers

Feature	LS1012 A	LS1021 A	LS1043 A	LS1046 A	LS1088 A	LS2088 A
Audio - SAI	Y	Y	na	na	na	na
CAAM DMA	Y	/	/	/	/	/
DCE	na	na	na	na	/	/
Direct Device Assignment	na	na	na	na	Y	Y
DMA	Y	Y	Y	Y	Y	Y
DPAA1	na	na	Y	Y	/	/
DPAA2	na	na	na	na	Y	Y
eSDHC	Y	Y	Y	Y	Y	Y
FlexCAN	na	/	na	na	na	na
I2C	Y	Y	Y	Y	Y	Y
IEEE1588, PTPD	na	Y	Y	Y	Y	Y
IFC	na	Y	Y	Y	Y	Y
IMA-EVM	Y	/	Y	Y	Y	Y
LPUART	na	Y	Y	Y	/	/

Table continues on the next page...

Table 4. Linux Kernel Drivers (continued)

QSPI	Y	Y	Y	Y	Y	Y
PCIe RC	Y	Y	Y	Y	Y	Y
PCIe EP	/	/	Y	Y	Y	Y
PFE	Y	na	na	na	na	na
Power Management	Y	Y	Y	Y	Y	Y
SATA	Y	Y	Y	Y	Y	Y
SEC	Y	Y	Y	Y	Y	Y
dSPI	/	Y	Y	Y	Y	Y
TDM (QE)	na	na	Y	na	na	na
USB	Y	Y	Y	Y	Y	Y
VeTSEC	na	Y	na	na	na	na
Video - DCU	na	Y	na	na	na	na
Watchdog	Y	Y	Y	Y	Y	Y

3.4 Supported Targets

Processors, development boards, and cards supported all releases.

NOTE

In the following tables, in the rows corresponding to the processors, the silicon revision is indicated. In the rows corresponding to the development boards, the board is marked with "Y" if it is supported. "N" means that a processor or development board is not supported.

Table 5. QorIQ Layerscape Processors Supported

Processor	Board	LSDK 17.06	LSDK 17.09	LSDK 17.09-update-103017	LSDK 17.12	LSDK 18.03	LSDK 18.06	LSDK 18.09
LS1012A		N	rev 1.0	rev 1.0 rev 2.0	rev 1.0 rev 2.0	rev 1.0 rev 2.0	rev 1.0 rev 2.0	rev 1.0 rev 2.0
	LS1012ARDB	N	Y	Y	Y	Y	Y	Y
	FRWY-LS1012A	N	N	N	N	N	Y	Y
LS1021A/LS1020A		rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0	rev 2.0
	TWR-LS1021A	Y	Y	Y	Y	Y	Y	Y
LS1043A/LS1023A		rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1	rev 1.1
	LS1043ARDB-PC	Y	Y	Y	Y	Y	Y	Y

Table continues on the next page...

Table 5. QorIQ Layerscape Processors Supported (continued)

	LS1043ARDB-PD	N	Y	Y	Y	Y	Y	Y
LS1046A/LS1026A		rev 1.0	rev 1.0	rev 1.0	rev 1.0	rev 1.0	rev 1.0	rev 1.0
	LS1046ARDB-PB	Y	Y	Y	Y	Y	Y	Y
LS1088A		rev 1.0	rev 1.0	rev1.0	rev 1.0	rev 1.0	rev 1.0	rev 1.0
	LS1088A-RDB	Y	Y	Y	Y	Y	Y	Y
	LS1088ARDB-PB	N	N	N	N	N	N	Y
LS2088A/LS2084A/ LS2081A		rev 1.0 rev 1.1	rev 1.0 rev 1.1	rev 1.0 rev 1.1	rev 1.0 rev 1.1	rev 1.0 rev 1.1	rev 1.0 rev 1.1	rev 1.0 rev 1.1
	LS2088A-RDB	Y	Y	Y	Y	Y	Y	Y

3.5 Fixed, Open, and Closed Issues

This section contains 3 tables: Fixed, Open, and Closed issues. Fixed issues have a software fix that has been integrated into the 'Fixed In' Release. Open issues do not currently have a resolution. Workaround suggestions are provided where possible. Closed issues are issues where the root cause and fix are outside the scope of the Layerscape SDK.

Table 6. LSDK 18.09 Fixed Issues

ID	Description	Disposition	Opened In	Fixed In
QLINUX-9121	Deep sleep is not supported on LS1021A TWR.	Fixed	LSDK 18.03	LSDK 18.09
QLINUX-9314	Linux netperf performance degrades on LS1088A.	Fixed	LSDK 18.06	LSDK 18.09
QSDK-4846	/gen_otpmk_drgb does not generate OTPMK for certain compiler and there will be `Error: Incorrect usage of cmd line options, urand flag has no usage along with string !!`	Fixed	LSDK 18.06	LSDK 18.09
QUBOOT-4202	Due to the failure of CRC check, the environment cannot be loaded from QSPI.	Fixed	LSDK 18.06	LSDK 18.09

Table 7. LSDK 18.09 Open Issues

ID	Description	Disposition	Opened In	Workarounds
DISTROS-564	Firmware lsdk_linux_arm32_tiny.itb does not work on LS1012AFRWY 512MB board for 32-bit mode. It works on LS1012AFRWY 1GB board	Open	LSDK 18.09	Delete the line 89 `load = <0x9ffe0000>` in flexbuild/configs/linux/linux_arm32.its, then run `flex-builder -i mklinux -a arm32` to generate new lsdk_linux_arm32_tiny.itb

Table continues on the next page...

Table 7. LSDK 18.09 Open Issues (continued)

DPDK-1364	DPDK L2 forward crypto performance degrades at 82 and 408 frame size on DPAA1 and DPAA2 platforms.	Open	LSDK 18.06	
DPDK-1367	QEMU command cannot be set up in DPDK OVS-VM L2 forward and L3 forward scenario.	Open	LSDK 18.06	
DPDK-1368	DPDK performance in docker is not measured on LTS 4.9.	Open	LSDK 18.06	
QLINUX-6595	On LS1046A RDB, transfer complete interrupt should be generated by eSDHC controller after it sends CMD18 (multiple blocks read) to card. However, after sleep, this interrupt did not occur for CMD18 and this caused software to report hardware timeout issue.	Open	SDK v2.0-1701	
QLINUX-8164	PCIe Advance Error Reporting is not available on LS1088A.	Open	LSDK 17.09	
QLINUX-8700	The DPAA2 hardware does not configure PFC congestion notifications for some DPNI objects created with restool. General congestion is reported correctly, as expected, but PFC flow control frames are not sent.	Open	LSDK 17.12	PFC congestion notifications and overall PFC support works as expected when creating DPNI objects via DPL. The problem only occurs when DPNI objects are created using restool.
QLINUX-8735	On Layerscape platforms, e1000 NIC card will lose PCIe link throughout the sleep process, which makes the kernel hang when resume. Though PCIe-SATA card will keep PCIe link in L0 in the sleep process, it still cannot resume from sleep all the time.	Open	LSDK 17.12	
QLINUX-9037	Linux IPv4 forwarding tests have a 15% degradation on DPAA 1.x platforms for small frame sizes (64 and 128 bytes) compared to LSDK 17.12.	Open	LSDK 18.03	
QLINUX-9078	PCIe MSI interrupt balancing does not work on both LTS 4.9 and LTS 4.14.	Open	LSDK 18.03	

Table continues on the next page...

Table 7. LSDK 18.09 Open Issues (continued)

QLINUX-9524	Sometimes PFE port has packet loss in Linux when two LS1012AFRWY boards are connected back-to-back. There is no packet loss when one LS1012AFRWY is connected to PC.	Open	LSDK 18.06	
QLINUX-10150	CPU dynamic offline fails on ARMv7 and ARMv8 32-bit	Openn	LSDK 18.09	
QLINUX-10181	Linux IPSec performance has large degradation on LTS 4.9 and LTS 4.14, compared with LTS 4.4	Open	LSDK 18.09	
QPPA-28	There is performance degradation in memory bandwidth test on LS2088A RDB.	Open	LSDK 17.12	
QSDK-5103	While validating OpenSSL 1.1.0g, rsa, ecdsa, dsa cipher do not work in aes and 3des.	Open	LSDK 18.09	
QSDK-5118	With OpenSSL 1.1.0g, asymmetric rsa does not work in IPSec offload test.	Open	LSDK 18.09	

Table 8. LSDK 18.09 Closed Issues

ID	Description	Disposition	Found In	Workarounds
DPDK-879	If traffic is sent on disconnected network ports of DPAA2 running DPDK, the board hangs and needs to be restarted.	Will not Fix	LSDK 17.06	Network ports shall be connected while sending traffic.
QLINUX-3357	On TWR-LS1021A, some resolutions (e.g. 1920x1080) may not work well with some monitors. The software will not downgrade to another resolution automatically.	Hardware Issue	NA	Manually set another resolution such as: 1024x768@60 : fbset -fb /dev/fb0 -g 1024 768 1024 768 24 -t 15384 168 8 29 3 144 6
QLINUX-5325	AQR PHY LED remains off if link is at 1Gbps on LS2088ARDB.	Hardware Issue	NA	
QLINUX-5417	Cortina PHY LEDs are permanently off on LS2088ARDB.	Hardware Issue	NA	

Table continues on the next page...

Table 8. LSDK 18.09 Closed Issues (continued)

QLINUX-5616	On LS1043A, KVM support on host machines with 64KB pages is not functional. The limitation exists, because the memory range associated with the GIC CPU interface, in the GIC400 memory map, is not aligned to 64KB.	Hardware Issue	SDK 2.0	Use only host machines with 4KB pages in order to support KVM virtualization.
QLINUX-5637	On TWR-LS1021A, after resuming from deep sleep, the kernel cannot initialize SD card and call trace occurs because the card never leaves busy state.	Hardware Issue	NA	Upgrade the on-board CPLD to version 3.2.
QLINUX-5661/ QUBOOT-1320	HP 2.0 pen drive not enumerated in Standard A port on LS2088ARDB and LS1043ARDB board. However, it is properly enumerated in micro port.	Hardware Issue	NA	
QLINUX-5671	On TWR-LS1021A, when doing deep sleep with all three Ethernet ports on, there may be the error message "PM: Device mdio@2d24000:02 failed to suspend: error -16".	Hardware Issue	NA	Upgrade the onboard CPLD to version 3.2.
QLINUX-7096	Jumbo and Scatter/Gather frames are not supported on LS1043A. All outgoing Scatter/Gather frames are linearized on egress. The limitation is caused by the software workaround for errata A-010022.	Hardware Issue	LSDK 17.09	
QLINUX-7733	KVM 32-bit is not supported on LS1043A and LS1046A. This is a limitation in KVM open source.	Will not Fix	LSDK 17.06	
QLINUX-9637	Busy poll feature is not supported by the DPAA2 Ethernet driver. Starting with kernel 4.10, busy polling support is included in the network stack core, with no code needed on the driver side. For previous kernel versions, driver support is necessary, but the DPAA2 Ethernet driver does not include this support.	Will not Fix	LSDK 18.06	
QLINUX-9693	SPI to DUART bridge is not supported on LS1012AFRWY.	Will not Fix	LSDK 18.06	
QSDK-1677	When telneting to board console from Linux server connected to on TWR-LS1021A, the board will power reset due to wrong signal sent.	Hardware Issue	NA	Remove R214 from the board.
QSDK-1841	On TWR-LS1021A, copy from NOR flash to NOR flash fails. It is a known limitation with Micron flash.	Hardware Issue	NA	

Table continues on the next page...

Table 8. LSDK 18.09 Closed Issues (continued)

QSDK-2478	On TWR-LS1021A, boot dtb, kernel, and filesystem directly from QSPI flash could not be supported.	Will not Fix	SDK v1.9	1.Program the general dtb, Linux kernel, and ramdisk to the QSPI flash by 'sf write' command(under sdboot or qspiboot). 2.Boot the dtb, kernel, and ramdisk from QSPI flash. Avoid booting from QSPI flash directly. Read the dtb, kernel and ramdisk from the QSPI flash to RAM by 'sf read' u-boot command and then boot from RAM. By default, the QSPI flash on the TWR-LS1021A includes: rcw, uboot, kernel, dtb, and ramdisk. Note: This workaround only applies to QSPI flash.
QSDK-3954	Transcend 8G class 10 SDHC card does not work with 50MHz high speed mode on LS2088ARDB.	Hardware Issue	LSDK 17.06	Reducing SD clock frequency or using SD cards from other vendors like Sandisk, Kingston, Sony.
QSDK-3955	USB flash drive from some vendors like Kingston, Transcend, Samtec does not work reliably on LS2088ARDB.	Hardware Issue	LSDK 17.06	Use USB flash drives from other vendors like ADATA, Sandisk, Lexar, Deloitte
QUBOOT-2055	The parameter `fdt_high` set to `0xffffffff` causes the failure of booting images from NOR flash directly.	Will not Fix	SDK 2.0-1611	Set fdt_high in uboot environment variable to 0xa0000000 by using the command in U-Boot: =>setenv fdt_high 0xa0000000

Table continues on the next page...

Table 8. LSDK 18.09 Closed Issues (continued)

QUBOOT-2161	The LS1021ATWR board and CPLD could not provide clock frequency information to the software. So the U-Boot software can only support static frequency settings (100M DDR clock and 100M system clock). If different clock settings are selected through the switch configurations.	Hardware Issue	NA	Change the settings in software manually(include/ configs/ ls1021atwr.h). Change the following settings to the frequency selected by the switches. #define CONFIG_SYS_CLK_FREQ 100000000 #define CONFIG_DDR_CLK_FREQ 100000000
QUBOOT-3480	SGMII PHY less support is not accepted in upstream, so the support is dropped in LSDK.	Will not Fix	LSDK 18.03	

Chapter 4
Layerscape SDK user guide

4.1 LSDK Quick Start

Introduction

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, uefi, rcw, ppa 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 build Linux, U-Boot, PPA, RCW and miscellaneous user space applications.
- Generate machine-specific composite firmware for various boot types: SD/QSPI/NOR/NAND boot, secure boot, U-Boot, UEFI.
- Support integrated management with repo-fetch, repo-branch, repo-commit, repo-tag, repo-update for git repositories of all components.
- Support cross build on x86 Ubuntu 18.04 host machine for `aarch64/armhf arch` target.
- Support native build on `aarch64/armhf` machine for ARM arch target.
- Support creating an Ubuntu docker container and building LSDK inside it when the host machine is using CentOS, RHEL, Fedora, SUSE, Debian, non-18.04 Ubuntu, etc.
- 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 automaticaly build all components, it generates the boot firmware (contains RCW, U-Boot/UEFI, PHY firmware, kernel image, and ramdiskrfs), `lsdk_linux_<arch>_tiny.itb`, and the Ubuntu userland containing the specified packages and application components.

NOTE: Since LSDK-18.06, upgrading of toolchain is required for U-Boot v2018.03 or later, if your host machine is not a Ubutnu 18.04 system, there are two ways to use Ubuntu 18.04 toolchain as below:

- 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, then build LSDK in docker.

4.1.1 Download and assemble LSDK images

Complete the following prerequisites before proceeding with downloading and assembling LSDK images.

- If Ubuntu 18.04 is installed on the host machine, run flex-builder command directly.

  Prerequisites:

  — 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.

- If a Linux distro other than Ubuntu 18.04 is installed on the host machine, use the "flex-builder docker" command to create an Ubuntu 18.04 Docker container to emulate the environment prior to running the other flex-builder commands.

  Prerequisites:

  — Make sure to install Docker on the host machine. You need `sudo` permission to execute the `docker` command or you need to be added to a group of docker. Refer to https://docs.docker.com/engine/installation/ as a reference on how to install Docker on the host machine.

  — To build the Ubuntu userland, the user's network environment must have *sudo* permission for *docker* command or the user must be added to a group of *docker* as specified below:

  ```
  $ sudo newgrp - docker
  $ sudo usermod -aG docker <accountname>
  $ sudo gpasswd - <accountname> docker
  $ sudo service docker restart
  ```

  Logout from current terminal session, then login again to ensure user can run `docker ps -a`

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

After completing the prerequisites, follow the steps below to download and assemble LSDK images:

1. **Login to Linux host machine and download the flexbuild source tarball.**

   Login 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 docker (optional, only in case local host machine is not Ubuntu 18.04
   system, to build LSDK in docker container)
   $ source setup.env    (execute only when using Docker, run this command after entering
   the docker container)
   ```

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

Platform	Commands to download prebuilt images

LS1012ARDB
LS1012A-FRWY
LS1043ARDB
LS1046ARDB
LS1088ARDB
LS1088ARDB-PB
LS2088ARDB
	`wget http://www.nxp.com/lgfiles/sdk/lsdk1809/app_components_arm64.tgz`
	For Linux 4.14:
	`wget http://www.nxp.com/lgfiles/sdk/lsdk1809/bootpartition_arm64_lts_4.14.tgz`
	`wget http://www.nxp.com/lgfiles/sdk/lsdk1809/arm64-modules-4.14.67.tgz`
	For Linux 4.9:
	`wget http://www.nxp.com/lgfiles/sdk/lsdk1809/bootpartition_arm64_lts_4.9.tgz`
	`wget http://www.nxp.com/lgfiles/sdk/lsdk1809/arm64-modules-4.9.124.tgz`

LS1021ATWR
LS1012ARDB 32bit
LS1012A-FRWY 32bit
LS1043ARDB 32bit
LS1046ARDB 32bit
	`wget http://www.nxp.com/lgfiles/sdk/lsdk1809/app_components_arm32.tgz`
	For Linux 4.14:
	`wget http://www.nxp.com/lgfiles/sdk/lsdk1809/bootpartition_arm32_lts_4.14.tgz`
	`wget http://www.nxp.com/lgfiles/sdk/lsdk1809/arm32-modules-4.14.67.tgz`
	For Linux 4.9:

Table continued from the previous page...

Platform	Commands to download prebuilt images

	`wget http://www.nxp.com/lgfiles/sdk/lsdk1809/bootpartition_arm32_lts_4.9.tgz`
	`wget http://www.nxp.com/lgfiles/sdk/lsdk1809/arm32-modules-4.9.124.tgz`

3. **Generate LSDK Ubuntu userland, untar the prebuilt components tarball, and merge it into the userland.**

```
 $ flex-builder -i mkrfs -a arm64 (locally generate default Ubuntu rootfs with
configurable packages)
 $ tar xvzf app_components_arm64.tgz -C build/apps
 $ sudo tar xvzf arm64-modules-<kernel_version>.tgz -C build/rfs/
rootfs_ubuntu_bionic_arm64/lib/modules
 $ flex-builder -i merge-component -a arm64
 $ flex-builder -i compressrfs -a arm64
 $ exit (optional,this command exits from docker when building in docker)
```

NOTE

For 32-bit platforms, change the architecture type (`-a` parameter) to `arm32`.

NOTE

In case your build machine is in a subnet that needs HTTP proxy to access external Internet, you can set environment variable http_proxy, https_proxy and no_porxy as below:

```
Add the following proxy settings in /etc/profile.d/proxy.sh (for all
accounts) or ~/.bashrc (for current account)
export http_proxy="http://<account>:<password>@<domain>:<port>"
export https_proxy="http://<account>:<password>@<domain>:<port>"
export no_proxy="localhost"
```

NOTE

Only Ubuntu userland is the default file system with full system test (various firmwares and userspace components are verified based on Ubuntu userland) in formal LSDK release. Although other non-official userland (Debian, CentOS, buildroot-based tiny distro, etc) can be generated and composed with the common LSDK boot partition (containing Linux kernel, DTBs, distro boot scripts, etc) by flexbuild , that is optional part besides the formal Ubuntu userland and there is no guarantee for other userlands as of now.

4.1.2 Deploy LSDK Images on the target board

This section describes how to deploy LSDK images on the target board. There are two major scenarios on how to deploy LSDK images on the target board:

- **Scenario 1: "Removable Media"**- If the user can connect a removable storage device to a local Linux host machine, the user can deploy LSDK images onto the removable storage device. The user can then connect it to the board and update the firmware from the storage device. For more information, refer to Deploy LSDK images from Linux Host on page 49.
- **Scenario 2: "Non-removable Media"**- If the user wants to deploy LSDK images to a non-removable media on the board or if the user does not have a local Linux host machine that can connect to a removable media, the user can directly deploy LSDK images on board. For more information, refer to Deploy LSDK images on board on page 51.

The deployment covers how to program LSDK composite firmware (QSPI boot, NOR boot, or SD boot) on the target board. It also covers how to deploy boot partition images and Ubuntu userland on different storage media (SD/USB/SATA).

Prerequisites: SD/USB/SATA capacity must be at least 8 GB.

4.1.2.1 Deploy LSDK images from Linux Host

To deploy LSDK images to the target board, users can connect a removable storage (SD/ USB/ SATA) device to a local Linux host machine, given that LSDK images have been generated as per the instructions in Download and assemble LSDK images on page 46section.

U-Boot based booting

1. Download appropriate LSDK images to local Linux host machine.
2. Setup the environment for flex-installer to run.

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

3. Execute flex-installer with appropriate arguments to deploy LSDK images to a second storage device.

```
In case U-Boot as bootloader
$ flex-installer -b <bootpart> -r <rootfs> -d <device>

In case UEFI as bootloader, option '-m <machine>' must be specified
$ flex-installer -b <bootpart> -r <rootfs> -d <device> -m <machine>

In case only updating composite firmware for SD boot (option '-f <firmware>' is not
supported for non SD boot)
$ flex-installer -f firmware_<machine>_uboot_sdboot.img -d <device>

Example:
$ flex-installer -b bootpartition_arm64_lts_4.14.tgz -r rootfs_ubuntu_bionic_arm64.tgz -
f firmware_ls046ardb_uboot_sdboot.img -d /dev/sdx
```

4. After a successful installation, “Installation finished successfully” message appears, then execute the following command to unmount the target device.

```
$ sudo umount /run/media/sdX
```

The following table summarizes the parameters for flex-installer command for various boards:

Board Name	<bootpart>	<rootfs>	<firmware>	<machine>	<device>
LS1021ATWR	bootpartition_arm32_lts_<version> or bootpartition_arm32_lts_<version>.tgz	rootfs_ubuntu_bionic_arm32 or rootfs_ubuntu_bionic_arm32.tgz	firmware_ls1021atwr_uboot_sdboot.img	ls1021atwr	/dev/sdX Refer to **Note** below
LS1012ARDB			firmware_ls1012ardb_uboot_qspiboot.img	ls1012ardb	
LS1012AFRWY			firmware_ls1012afrwy_uboot_qspiboot.img	ls1012afrwy	
LS1043ARDB			firmware_ls1043ardb_uboot_sdboot.img	ls1043ardb	
LS1046ARDB			firmware_ls1046ardb_uboot_sdboot.img	ls1046ardb	
LS1012ARDB	bootpartition_arm64_lts_<version> or bootpartition_arm64_lts_<version>.tgz	rootfs_ubuntu_bionic_arm64 or rootfs_ubuntu_bionic_arm64.tgz	firmware_ls1012ardb_uboot_qspiboot.img	ls1012ardb	/dev/sdX Refer to **Note** below
LS1012AFRWY			firmware_ls1012afrwy_uboot_qspiboot.img	ls1012afrwy	
LS1043ARDB			firmware_ls1043ardb_uboot_sdboot.img	ls1043ardb	

Table continues on the next page...

Table continued from the previous page...

LS1046ARDB			firmware_ls1046ardb_uboot_sdboot.img	ls1046ardb	
LS1088ARDB			firmware_ls1088ardb_uboot_sdboot.img	ls1088ardb	
LS1088ARDB-PB			firmware_ls1088ardb_pb_uboot_sdboot.img	ls1088ardb_pb	
LS2088ARDB			firmware_ls2088ardb_uboot_norboot.img	Ls2088ardb	

NOTE

- The SD/USB/SATA storage drive in the Linux PC is detected as /dev/sdX, where X is a letter such as a, b, c. Make sure to choose the correct device name, because data on this device will be replaced.
- Use the command cat /proc/partitions to see a list of devices and their sizes to make sure that the correct device names have been chosen.
- If your Linux host machine supports read/write SD card directly without an extra SD card reader device, the device name of SD card generally is mmcblk0.

5. After unmounting, unplug the SD card from the Linux host and plugin it into the board
6. Make sure the DIP Switch settings on the board enable SD boot. (Refer to “Board-specific Information” section for switch settings)
7. Power-on the board. The system will automatically boot up LSDK Ubuntu distro available on the SD card (exception for LS2088ARDB).

Deploying on LS2088ARDB

As LS2088ARDB board doesn't support SD boot, users can deploy LSDK images to SD/USB/SATA device on LS2088ARDB based on NOR boot as below.

1. Execute the following commands to program boot partition and Ubuntu userland to SD/USB/SATA storage device.

   Assumption: Removable storage device is mounted as /dev/sdb on host machine.

   ```
   $ flex-installer -b bootpartition_arm64_lts_<version>.tgz -r build/rfs/
   rootfs_ubuntu_bionic_arm64 -d /dev/sdb
   ```
2. Unplug SD/USB/SATA device from host machine and plug it to the target board, firmware_ls2088ardb_uboot_norboot.img is already stored in the bootpartition of SD/USB/SATA device.
3. Program the composite firmware to IFC-NOR/QSPI flash as described below:

   - Booting U-Boot on LS2088ARDB

Storage media	Command in U-Boot

USB	`=> usb start`
	`=> load usb 0:2 a0000000 firmware_ls2088ardb_uboot_norboot.img`

SATA	`=> load scsi 0:2 a0000000 firmware_ls2088ardb_uboot_norboot.img`
SD	`=> load mmc 0:2 a0000000 firmware_ls2088ardb_uboot_norboot.img`

   - Execute the commands below to program the alternate bank:

```
=> protect off 584000000 +$filesize && erase 584000000 +$filesize && cp.b a0000000
584000000 $filesize
=> qixis_reset altbank
```

Systems automatically boots up the LSDK Ubuntu distro, after LSDK images are successfully deployed on the target boards.

Use the following default credentials to log onto to the LSDK distro:

- root/root
- user/user

UEFI based booting

- UEFI booting is supported on the following platforms:
  - LS1043ARDB
  - LS1046ARDB
  - LS2088ARDB
- Download the composite firmware to target board via U-Boot in bank0 of NOR/QSPI flash as below:

Platform	Command
LS1043ARDB	`=> tftp a0000000 firmware_ls1043ardb_uefi_norboot.img`
LS1046ARDB	`=> tftp a0000000 firmware_ls1046ardb_uefi_qspiboot.img`
LS2088ARDB	`=> tftp a0000000 firmware_ls2088ardb_uefi_norboot.img`

- Program the composite firmware into the alternate bank:

LS1043ARDB	`=> protect off 64000000 +$filesize && erase 64000000 +$filesize && cp.b a0000000 64000000 $filesize`
	`=> cpld reset altbank`

LS1046ARDB	`=> sf probe 0:1`
	`=> sf erase 0 +$filesize && sf write 0xa0000000 0 $filesize`
	`=> cpld reset altbank`

LS2088ARDB	`=> protect off 584000000 +$filesize && erase 584000000 +$filesize && cp.b a0000000 584000000 $filesize`
	`=> qixis_reset altbank`

Systems automatically boots up the LSDK Ubuntu distro, after LSDK images are successfully deployed on the required target boards.

Use the following default credentials to log onto to the LSDK distro:

- root/root or
- user/user

4.1.2.2 Deploy LSDK images on board

LSDK images have been generated in Download and assemble LSDK images on page 46, so the user can start to deploy LSDK images on the board if user is deploying to a non-removable media on the board or user does not have a local Linux server to connect removable media. In order to deploy LSDK images on the board, follow the instructions below:

1. Download LSDK composite firmware from the NXP website.

There are four types of composite firmware depending on the boot type: NOR/NAND/QSPI/SD boot. Download the LSDK composite firmware to the Linux host machine and put it in the TFTP server root directory.

- `firmware_<platform>_uboot_qspiboot.img` means the firmware boots from QSPI flash. Only LS1012ARDB, LS1012A-FRWY, LS1046ARDB, LS1088ARD, LS1088ARD-PB and LS2088ARDB support QSPI boot in this release. Use the following images and command below depending on your platform:
  - `firmware_ls1012ardb_uboot_qspiboot.img`
  - `firmware_ls1012afrwy_uboot_qspiboot`
  - `firmware_ls1046ardb_uboot_qspiboot.img`
  - `firmware_ls1088ardb_uboot_qspiboot.img`
  - `firmware_ls1088ardb_pb_uboot_qspiboot.img`
  - `firmware_ls2088ardb_uboot_qspiboot.img`

  ```
  $ wget http://www.nxp.com/lgfiles/sdk/lsdk1809/firmware_ls1046ardb_uboot_qspiboot.img
  ```
- `firmware_<platform>_uboot_sdboot.img` means the firmware boots from SD card. LS1043ARDB, LS1046ARDB, and LS1088ARDB, LS1088ARDB-PB support SD boot. Use the following images and command below depending on your platform:
  - `firmware_ls1021atwr_uboot_sdboot.img`
  - `firmware_ls1043ardb_uboot_sdboot.img`
  - `firmware_ls1046ardb_uboot_sdboot.img`
  - `firmware_ls1088ardb_uboot_sdboot.img`
  - `firmware_ls1088ardb_pb_uboot_sdboot.img`

  ```
  $ wget http://www.nxp.com/lgfiles/sdk/lsdk1809/firmware_ls1088ardb_uboot_sdboot.img
  ```
- `firmware_<platform>_uboot_norboot.img` means firmware boots from IFC NOR. LS1021ATWR, LS1043ARDB, and LS2088ARDB support IFC NOR boot. Use the following images and command below depending on your platform:
  - `firmware_ls1021atwr_uboot_norboot.img`
  - `firmware_ls1043ardb_uboot_norboot.img`
  - `firmware_ls2088ardb_uboot_norboot.img`

  ```
  $ wget http://www.nxp.com/lgfiles/sdk/lsdk1809/firmware_ls2088ardb_uboot_norboot.img
  ```
- `firmware_<platform>_uboot_nandboot.img` means firmware boots from NAND. LS1043ARDB supports NAND boot. Use the following images and command below depending on your platform:

  ```
  $ wget http://www.nxp.com/lgfiles/sdk/lsdk1809/firmware_ls1043ardb_uboot_nandboot.img
  ```
- Place the LSDK composite firmware into a TFTP server, then download the firmware via TFTP to the target board under the U-Boot prompt using the commands below:

Platform	Command in U-Boot

LS1012ARDB	`=> tftp a0000000 firmware_ls1012ardb_uboot_qspiboot.img`
	`=> i2c mw 0x24 0x7 0xfc;i2c mw 0x24 0x3 0xf5`
	`=> sf probe 0:0`
	`=> sf erase 0 +$filesize && sf write 0xa0000000 0 $filesize`
	`=> reset`

LS1012A-FRWY	`=> tftp 0x96000000 firmware_ls1012afrwy_uboot_qspiboot.img`
	`=> sf probe 0:0;sf erase 0 +$filesize; sf write 96000000 0 $filesize`
	`=> reset`

Table continues on the next page...

Table continued from the previous page...

Platform	Command in U-Boot

LS1021ATWR	`=> tftp a0000000 firmware_ls1021atwr_uboot_norboot.img`
	`=> protect off 64000000 +$filesize && erase 64000000 +$filesize && cp.b a0000000 64000000 $filesize`
	`=> boot_bank 1`

LS1043ARDB	`=> tftp a0000000 firmware_ls1043ardb_uboot_norboot.img`
	`=> protect off 64000000 +$filesize && erase 64000000 +$filesize && cp.b a0000000 64000000 $filesize`
	`=> cpld reset altbank`

LS1046ARDB	`=> tftp a0000000 firmware_ls1046ardb_uboot_qspiboot.img`
	`=> sf probe 0:1`
	`=> sf erase 0 +$filesize && sf write 0xa0000000 0 $filesize`
	`=> cpld reset altbank`

LS1088ARDB / LS1088ARDB_PB	For SD boot:
	`=> tftp a0000000 firmware_ls1088ardb_uboot_sdboot.img`
	`or`
	`=> tftp a0000000 firmware_ls1088ardb_pb_uboot_sdboot.img`
	`=> mmc write a0000000 8 1fff8`
	For QSPI boot (programming from altbank):
	`=> sf probe 0:1`
	`=> tftp 0xa0000000 firmware_ls1088ardb_uboot_qspiboot.img`
	`or`
	`=> tftp 0xa0000000 firmware_ls1088ardb_pb_uboot_qspiboot.img`
	`=> sf erase 0 +$filesize`
	`=> sf write 0xa0000000 0 $filesize`
	`=> i2c mw 0x66 0x50 20;i2c mw 66 10 20;i2c mw 66 10 21`

LS2088ARDB	`=> tftp a0000000 firmware_ls2088ardb_uboot_norboot.img`
	`=> protect off 584000000 +$filesize && erase 584000000 +$filesize && cp.b a0000000 584000000 $filesize`
	`=> qixis_reset altbank`

LSDK firmware (for example: `firmware_ls1088ardb_uboot_qspiboot.img`) is a composite image for target board which includes RCW+PBI, U-Boot/UEFI, PPA, boot loader environment variables, DPAA1 FMan ucode/PFE firmware, QE/uQE firmware, Ethernet PHY firmware, DPAA2 MC firmware, DPAA2 DPL, DPAA2 DPC, dtb, and`lsdk_linux_<arch>_tiny.itb` images.

After the steps above are completed and the DIP switches are set properly (refer to Board-specific Information on page 75), power on the board. The board will automatically boot up and enter the Linux system.

2. Deploy boot partition and Ubuntu 18.04 userland to SD/USB/SATA

   Follow the instructions below to deploy boot partition and Ubuntu userland to SD/USB/SATA storage device.

   - Enable network connection to download LSDK images

Command	Target Board	Assumption
`$ udhcpc -i eth0`	LS1012ARDB	LS1012ARDB uses ETH0 port.
`$ udhcpc -i eth0`	LS1012A-FRWY	LS1012A-FRWY uses ETH0 port.
`$ udhcpc -i ni0`	LS2088ARDB	LS2088ARDB uses ETH0 port.

Table continues on the next page...

Table continued from the previous page...

Command	Target Board	Assumption
`$ udhcpc -i eth1`	LS1088ARDB	LS1088ARDB uses ETH8 port.
`$ udhcpc -i eth2`	LS1043ARDB	LS1043ARDB uses RGMII1 port
`$ udhcpc -i eth0`	LS1046ARDB	LS1046ARDB uses RGMII1 port
`$ udhcpc -i eth2`	LS1021ATWR	LS1021ATWR uses RGMII port.

NOTE

Ethernet port name can be found in frontend control panel of target board.

- Use flex-installer to create and format the partitions (USB/SATA/SD)

Storage Media	Command in Linux
USB	`$ flex-installer -i pf -d /dev/sdX`
SATA	`$ flex-installer -i pf -d /dev/sdX`
SD	`$ flex-installer -i pf -d /dev/mmcblk0`

- Download and deploy two tarballs (boot partition and Ubuntu userland) to USB/SATA/SD storage device. Take LS2088ARDB as example.

Storage Media	Command in Linux

USB	`$ cd /run/media/sdX3`
	Download `bootpartition_<arch>_<version>.tgz` and `rootfs_ubuntu_<codename>_<arch>_xx.tgz` using the `wget` or `scp` command.
	`$ flex-installer -b bootpartition_<arch>_lts_<version>.tgz -r rootfs_ubuntu_<codename>_<arch>_<timestamp>.tgz -d /dev/sdX`

SATA	`$ cd /run/media/sdX3`
	Download `bootpartition_<arch>_<version>.tgz` and `ubuntu_<codename>_<arch>_rootfs_xx.tgz` using the `wget` or `scp` command.
	`$ flex-installer -b bootpartition_<arch>_lts_<version>.tgz -r rootfs_ubuntu_<codename>_<arch>_<timestamp>.tgz -d /dev/sdX`

SD	`$ cd /run/media/mmcblk0p3`
	Download `bootpartition_<arch>_<version>.tgz` and `ubuntu_<codename>_<arch>_rootfs_xx.tgz` using the `wget` or `scp` command.
	`$ flex-installer -b bootpartition_<arch>_lts_<version>.tgz -r rootfs_ubuntu_<codename>_<arch>_<timestamp>.tgz -d /dev/mmcblk0`

- After completing the steps above, reboot the board with the LSDK images. The system will automatically boot Ubuntu userland.

4.1.3 Deploy LSDK Images for secure boot on the target board

This section describes how to deploy LSDK images on the target board, for secure boot. There are two major scenarios on how to deploy LSDK images on the target board:

- **Scenario 1: "Removable Media"**- If the user can connect a removable storage device to a local Linux host machine, the user can deploy the images onto the removable storage device. The user can then connect it to the board and update the firmware from the storage device. For more information, refer to Deploy LSDK images from Linux Host on page 49.
- **Scenario 2: "Non-removable Media"**- If the user wants to deploy LSDK images to a non-removable media on the board or if the user does not have a local Linux host machine that can connect to a removable media, the user can directly deploy LSDK images on board. For more information, refer to Deploy LSDK Secure images on board on page 57.

The deployment covers how to program LSDK composite firmware (QSPI boot, NOR boot, or SD boot) on the target board. It also covers how to deploy boot partition images and Ubuntu userland on different storage media (SD/USB/SATA).

Prerequisites: SD/USB/SATA capacity must be at least 8 GB.

4.1.3.1 Deploy LSDK Secure images from Linux Host

LSDK images for secure boot have been generated in Download and assemble LSDK images on page 46, so the user can start to deploy these images on the board once the user connects a removable storage device to a local Linux host.

U-Boot based booting

1. Download appropriate LSDK images to local Linux host machine.
2. Setup the environment for flex-installer to run.

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

U-Boot is used as the boot loader by default. LS1021ATWR, LS1043ARDB, LS1046ARDB and LS1088ARDB support SD boot, so you can directly program the LSDK composite firmware, boot partition images and Ubuntu userland into an SD card.

1. Execute flex-installer with appropriate arguments to deploy LSDK images to a second storage device.

```
$ flex-installer -b <bootpart> -r <rootfs> -d <device>
$ flex-installer -f firmware_<machine>_uboot_sdboot_secure.img -d <device>  (optional, 
only for SD boot, no need for QSPI/NOR/NAND boot)
For example:
$ flex-installer -b bootpartition_arm64_lts_4.14.tgz -r rootfs_ubuntu_bionic_arm64.tgz -
d /dev/sdb  (please append option '-m <machine>' in case UEFI as bootloader)
$ flex-installer -f firmware_ls1088ardb_uboot_sdboot_secure.img -d /dev/sdb
```

2. After a successful installation, “Installation Finished Successfully” message appears, then execute the following command to unmount the target device.

```
$ sudo umount /run/media/sdX
```

The following table summarizes the parameters for flex-installer command for various boards:

Board Name	<bootpart>	<rootfs>	<firmware>	<machine>	<device>
LS1021ATWR	bootpartition_arm32_lts_<version> or bootpartition_arm32_lts_<version>.tgz	rootfs_ubuntu_bionic_arm32 or rootfs_ubuntu_bionic_arm32.tgz	firmware_ls1021atwr_uboot_sdboot_secure.img	ls1021atwr	/dev/sdX Refer to **Note** below
LS1012ARDB			firmware_ls1012ardb_uboot_qspiboot_secure.img	ls1012ardb	
LS1012A-FRWY			firmware_ls1012afrwy_uboot_qspiboot_secure.img	ls1012afrwy	

Table continues on the next page...

Table continued from the previous page...

LS1043ARDB			firmware_ls1043ardb_uboot_sdboot_secure.img	ls1043ardb	
LS1046ARDB			firmware_ls1046ardb_uboot_sdboot_secure.img	ls1046ardb	
LS1012ARDB	bootpartition_arm64_lts_<version> or bootpartition_arm64_lts_<version>.tgz	rootfs_ubuntu_bionic_arm64 or rootfs_ubuntu_bionic_arm64.tgz	firmware_ls1012ardb_uboot_qspiboot_secure.img	ls1012ardb	/dev/sdX Refer to **Note** below
LS1012A-FRWY			firmware_ls1012afrwy_uboot_qspiboot_secure.img	ls1012afrwy	
LS1043ARDB			firmware_ls1043ardb_uboot_sdboot_secure.img	ls1043ardb	
LS1046ARDB			firmware_ls1046ardb_uboot_sdboot_secure.img	ls1046ardb	
LS1088ARDB			firmware_ls1088ardb_uboot_sdboot_secure.img	ls1088ardb	
LS1088ARDB-PB			firmware_ls1088ardb_pb_uboot_sdboot_secure.img	ls1088ardb_pb	
LS2088ARDB			firmware_ls2088ardb_uboot_norboot_secure.img	Ls2088ardb	

NOTE

- The SD/USB/SATA storage drive in the Linux PC is detected as /dev/sdX, where X is a letter such as a, b, c. Make sure to choose the correct device name, because data on this device will be replaced.
- Use the command cat /proc/partitions to see a list of devices and their sizes to make sure that the correct device names have been chosen.
- If your Linux host machine supports read/write SD card directly without an extra SD card reader device, the device name of SD card generally is mmcblk0.

3. After unmounting, unplug the SD card from the Linux host and plugin it into the board
4. Make sure the DIP Switch settings on the board enable SD boot. (Refer to “Board-specific Information” section for switch settings)
5. Power-on the board.
6. After the commands above are completed, all LSDK images are successfully deployed on the target board.Refer section Running secure boot on target platforms on page 72 for running secure boot
7. The system will automatically boot up LSDK Ubuntu distro available on the SD card (exception for LS2088ARDB).

Deploying on LS2088ARDB

As LS2088ARDB board doesn't support SD boot, users can deploy LSDK images to SD/USB/SATA device on LS2088ARDB based on NOR boot as below.

1. Execute the following commands to program boot partition and Ubuntu userland to SD/USB/SATA storage device.

   Assumption: Removable storage device is mounted as /dev/sdb on host machine.

   ```
   $ flex-installer -b bootpartition_arm64_lts_<version>.tgz -r build/rfs/
   rootfs_ubuntu_bionic_arm64 -d /dev/sdb
   （please append '-m ls2088ardb' in case UEFI as bootloader）
   ```

2. Unplug SD/USB/SATA device from host machine and plug it to the target board, firmware_ls2088ardb_uboot_norboot.img is already stored in the bootpartition of SD/USB/SATA device.
3. Program the composite firmware to IFC-NOR/QSPI flash as described below:

   Before Deploying the images, prepare the board for secure boot. For this refer section:Prepare board for Secure Boot on page 70

   - Booting U-Boot on LS2088ARDB

Storage media	Command in U-Boot

USB	`=> usb start`
	`=> load usb 0:2 a0000000 firmware_ls2088ardb_uboot_norboot_secure.img`

SATA	`=> load scsi 0:2 a0000000 firmware_ls2088ardb_uboot_norboot_secure.img`
SD	`=> load mmc 0:2 a0000000 firmware_ls2088ardb_uboot_norboot_secure.img`

   - Execute the commands below to program the alternate bank:

```
=> protect off 584000000 +$filesize && erase 584000000 +$filesize && cp.b a0000000
584000000 $filesize
=> qixis_reset altbank
```

     After the commands above are completed, all LSDK images are successfully deployed on the target board.Refer sectionRunning secure boot on target platforms on page 72 for running secure boot. Before running secure boot download srk_hash.txt file from github. Linux will boot up successfully, upon successful validation of the images. To log in to LSDK distro, the two default usernames and passwords are: `root/root` or `user/user`. After the first login, users can change the default login password Use the following default credentials to log onto to the LSDK distro:

     — root/root

     — user/user

4.1.3.2 Deploy LSDK Secure images on board

LSDK secure boot images have been generated in Download and assemble LSDK images on page 46, so the user can start to deploy LSDK images on the board if user is deploying to a non-removable media on the board or user does not have a local Linux server to connect removable media. In order to deploy LSDK images on the board, follow the instructions below:

1. Download LSDK composite firmware from the NXP website.

   There are four types of composite firmware depending on the boot type: NOR/NAND/QSPI/SD boot. Download the LSDK composite firmware to the Linux host machine and put it in the TFTP server service directory.

   - `firmware_<platform>_uboot_qspiboot_secure.img` means the firmware boots from QSPI flash. Only LS1012ARDB, LS1046ARDB, LS1088ARDB, and LS2088ARDB support QSPI boot in this release. Use the following images and command below depending on your platform:

     — `firmware_ls1046ardb_uboot_qspiboot_secure.img`

     — `firmware_ls1088ardb_uboot_qspiboot_secure.img`

     — `firmware_ls1088ardb_pb_uboot_qspiboot_secure.img`

     — `firmware_ls2088ardb_uboot_qspiboot_secure.img`

     — `firmware_ls1012ardb_uboot_qspiboot_secure.img`

     — `firmware_ls1012afrwy_uboot_qspiboot_secure.img`

```
$ wget http://www.nxp.com/lgfiles/sdk/lsdk1809/
firmware_ls1046ardb_uboot_qspiboot_secure.img
```

- `firmware_<platform>_uboot_sdboot_secure.img` means the firmware boots from SD card. LS1021ATWR, LS1043ARDB, LS1046ARDB, and LS1088ARDB support SD boot. Use the following images and command below depending on your platform:
  - `firmware_ls1021atwr_uboot_sdboot_secure.img`
  - `firmware_ls1043ardb_uboot_sdboot_secure.img`
  - `firmware_ls1046ardb_uboot_sdboot_secure.img`
  - `firmware_ls1088ardb_uboot_sdboot_secure.img`
  - `firmware_ls1088ardb_pb_uboot_sdboot_secure.img`

  ```
  $ wget http://www.nxp.com/lgfiles/sdk/lsdk1809/
  firmware_ls1088ardb_uboot_sdboot_secure.img
  ```
- `firmware_<platform>_uboot_norboot_secure.img` means firmware boots from IFC NOR. LS1021ATWR, LS1043ARDB, and LS2088ARDB support IFC NOR boot. Use the following images and command below depending on your platform:
  - `firmware_ls1021atwr_uboot_norboot_secure.img`
  - `firmware_ls1043ardb_uboot_norboot_secure.img`
  - `firmware_ls2088ardb_uboot_norboot_secure.img`

  ```
  $ wget http://www.nxp.com/lgfiles/sdk/lsdk1809/firmware_ls2088ardb_uboot_norboot.img
  ```

2. Program LSDK composite firmware for target board

   LSDK composite firmware includes RCW+PBI, U-Boot/UEFI, PPA, boot loader environment variables, DPAA1 FMan ucode, QE/uQE firmware, Ethernet PHY firmware, DPAA2 MC firmware, DPAA2 DPL, DPAA2 DPC, device tree, `flex_linux_<arch>.itb` images.

   Before Deploying the images, prepare the board for secure boot. For this refer section: Prepare board for Secure Boot on page 70

   Place the LSDK composite firmware into a TFTP server, then download the firmware via TFTP to the target board under the U-Boot prompt using the commands below:

Platform	Command in U-Boot

LS1021ATWR	`=> tftp a0000000 firmware_ls1021atwr_uboot_norboot_secure.img`
	`=> protect off 64000000 +$filesize && erase 64000000 +$filesize && cp.b a0000000 64000000 $filesize`
	`=> boot_bank 1`

LS1043ARDB	`=> tftp a0000000 firmware_ls1043ardb_uboot_norboot_secure.img`
	`=> protect off 64000000 +$filesize && erase 64000000 +$filesize && cp.b a0000000 64000000 $filesize`
	`=> cpld reset altbank`

LS1046ARDB	`=> tftp a0000000 firmware_ls1046ardb_uboot_qspiboot_secure.img`
	`=> sf probe 0:1`
	`=> sf erase 0 +$filesize && sf write 0xa0000000 0 $filesize`
	`=> cpld reset altbank`

LS1012ARDB	`=> tftp a0000000 firmware_ls1012ardb_uboot_qspiboot_secure.img`
	`=> i2c mw 0x24 0x7 0xfc; i2c mw 0x24 0x3 0xf5;`
	`=> sf probe 0:0`
	`=> sf erase 0 +$filesize && sf write 0xa0000000 0 $filesize`
	`=> reset`

Table continues on the next page...

Table continued from the previous page...

Platform	Command in U-Boot

LS1012AFRWY	`=> tftp a0000000 firmware_ls1012afrwy_uboot_qspiboot.img`
	`=> i2c mw 0x24 0x7 0xfc; i2c mw 0x24 0x3 0xf5;`
	`=> sf probe 0:0`
	`=> sf erase 0 +$filesize && sf write 0xa0000000 0 $filesize`
	`=> reset`

LS1088ARDB / LS1088ARDB-PB	`=> tftp a0000000 firmware_ls1088ardb_uboot_qspiboot_secure.img`
	`or`
	`=> tftp a0000000 firmware_ls1088ardb_pb_uboot_qspiboot_secure.img`
	`=> sf probe 0:1`
	`=> sf erase 0 +$filesize && sf write 0xa0000000 0 $filesize`
	`=> i2c mw 66 0x60 0x20; i2c mw 66 10 10;i2c mw 66 10 21`

LS2088ARDB	`=> tftp a0000000 firmware_ls2088ardb_uboot_norboot_secure.img`
	`=> protect off 584000000 +$filesize && erase 584000000 +$filesize`
	`&& cp.b a0000000 584000000 $filesize`
	`=> qixis_reset altbank`

Deploying on LS1012AFRWY

For deploying LSDK images from Linux host on ls1012afrwy using secure boot, first write non-secure firmware image on the board as mentioned in the commands in above table, and boot.After getting the uboot prompt, copy Kernel image and its header to SD card with the commands below.

```
tftp 0x80000000 lsdk_linux_arm64_tiny.itb; mmc write 0x80000000 0x8000 0x16000;
tftp 0x80000000 hdr_kernel.out; mmc write 0x80000000 0x4000 0x10;
```

lsdk_linux_arm64_tiny.itb will be present in bootpartition_arm64_lts_<version>/ and hdr_kernel.out will be present in bootpartition_arm64_lts_<version>/secboot_hdrs/

After writing these images, write secure boot firmware image with commands as mentioned below

```
=> tftp 80000000 firmware_ls1012afrwy_uboot_qspiboot.img
=> i2c mw 0x24 0x7 0xfc; i2c mw 0x24 0x3 0xf5;
=> sf probe 0:0
=> sf erase 0 +$filesize && sf write 0x80000000 0 $filesize
=> reset
```

After the steps above are completed and the DIP switches are set properly (refer to Board-specific Information on page 75), Refer section Running secure boot on target platforms on page 72 for running secure boot. Before running secure boot download srk_hash.txt file from github. Once secure boot is run, the board comes up.

3. Deploy boot partition and Ubuntu 18.04 userland to SD/USB/SATA

   Follow the instructions below to deploy boot partition and Ubuntu userland to SD/USB/SATA storage device.

   - Enable network connection to download LSDK images

Command	Target Board	Assumption
`udhcpc -i eth0`	LS2088ARDB	LS2088ARDB uses ETH0 port.
`$ udhcpc -i eth0`	LS1012ARDB	LS1012ARDB uses ETH0 port.
`$ udhcpc -i eth1`	LS1088ARDB	LS1088ARDB uses ETH8 port.
`$ udhcpc -i eth2`	LS1043ARDB	LS1043ARDB uses RGMII1 port
`$ udhcpc -i eth0`	LS1046ARDB	LS1046ARDB uses RGMII1 port
`$ udhcpc -i eth2`	LS1021ATWR	LS1021ATWR uses RGMII port.

NOTE

Ethernet port name can be found in frontend control panel of target board.

- Use flex-installer to create and format partitions (USB/SATA/SD)

Storage Media	Command in Linux
USB	`$ flex-installer -i pf -d /dev/sdX`
SATA	`$ flex-installer -i pf -d /dev/sdX`
SD	`$ flex-installer -i pf '-d /dev/mmcblkX`

- Download and deploy two tarballs boot partition and Ubuntu userland to USB/SATA/SD storage device. Take LS2088ARDB as example.

Storage Media	Command in Linux

USB	`$ cd /run/media/sdX3`
	Download `bootpartition_arm64.tgz` and `ubuntu_bionic_arm64_rootfs.tgz` using the `wget` or `scp` command.
	`$ flex-installer -b bootpartition_arm64.tgz -r rootfs_ubuntu_bionic_arm64.tgz -m ls2088ardb -d /dev/sdX`

SATA	`$ cd /run/media/sdX3`
	Download `bootpartition_arm64.tgz` and `ubuntu_bionic_arm64_rootfs.tgz` using the `wget` or `scp` command.
	`$ flex-installer -b bootpartition_arm64.tgz -r rootfs_ubuntu_bionic_arm64.tgz -m ls2088ardb -d /dev/sdX`

SD	`$ cd /run/media/sdX3`
	Download `bootpartition_arm64.tgz` and `ubuntu_bionic_arm64_rootfs.tgz` using the `wget` or `scp` command.
	`$ flex-installer -b bootpartition_arm64.tgz -r rootfs_ubuntu_bionic_arm64.tgz -m ls2088ardb -d /dev/mmcblk0`

- After completing the steps above, reboot the board with the LSDK images. The system will automatically boot Ubuntu userland.

NOTE

Make sure to configure the accurate name of the network interface on the Linux, as the label on the board may not be consistent with the name on Linux.

4.2 How to build LSDK with Flexbuild

Flexbuild provides cmdline for various build scenarios. The LSDK Quick Start on page 46 section introduces how to build the LSDK distro userland with prebuilt boot partition and component tarballs for quick deployment on the target board. This section introduces detailed steps to build LSDK with Flexbuild.

Login www.nxp.com/webapp/swlicensing/sso/downloadSoftware.sp?catid=LAYERSCAPE-SDK 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 custom kernel and update boot partition images with Flexbuild

NOTE

You can use the standard kernel build system to build LSDK kernel. It is possible to build and install a kernel manually. See Configuring and building on page 319. The section below describes how to use flex-builder to automate the kernel build process. Run the commands in the following table to build the kernel for your platform. The commands must be run on your Ubuntu 16.04 build system or in a Docker container based on Ubuntu 16.04 docker image hosted on CentOS, Fedora, RHEL, SUSE, Debian, Ubuntu system.

The section below describes how to use flex-builder to automate the kernel build process.

1. To build the kernel for your platform, run the commands in the table below. The commands must be run on your Ubuntu 16.04 build system or in a Docker container on it.

Platform	Command for building Linux

LS1012ARDB 64bit (ARMv8)
LS1012A-FRWY 64bit (ARMv8)
LS1043ARDB 64bit
LS1046ARDB 64bit
LS1088ARDB 64bit
LS1088ARDB-PB 64bit
LS2088ARDB 64bit
	`$ flex-builder -c linux:custom (optional, customize kernel config in interactive menu)`
	`or`
	`$ flex-builder -c linux:<kernel-repo>:<branch/tag> -B fragment:<mycustom>.config (optional, add additional fragment config)`
	`e.g. $ flex-builder -c linux:linux:LSDK-18.06-V4.14 -B fragment:lttng.config`
	
	`$ flex-builder -c linux`
	`$ flex-builder -i mkbootpartition`
	`(if -a <arch> is not specified,  arm64 arch is built by default)`

LS1012ARDB 32bit (ARMv8)
LS1012A-FRWY 32bit (ARMv8)
LS1043ARDB 32bit
LS1046ARDB 32bit
	`$ flex-builder -c linux:custom -a arm32 (optional, customize kernel config in interactive menu)`
	`$ flex-builder -c linux -a arm32`
	`$ flex-builder -i mkbootpartition -a arm32`

LS1021ATWR 32bit (ARMv7)	`$ flex-builder -c linux:custom -a arm32 -m ls1021atwr (optional, customize kernel config in interactive menu)`
	`$ flex-builder -c linux -c arm32 -m ls1021atwr`
	`$ flex-builder -i mkbootpartition -a arm32`

The commands above will generate bootpartition_<arch>_lts_<version>.tgz and <arch>-modules-<version>.tgz tarball in the directory `$WORKPATH/flexbuild/build/images`.

2. In case of secure boot, additional signing is required for bootpartition using the following command:

```
$ flex-builder -i signimg -m <machine> -b <boottype>
$ flex-builder -i mkbootpartition -a <arch> -s
```

3. Deploy updated boot partition tarball on target board

After the kernel image is rebuilt, deploy the updated boot partition images onto the storage device (USB/SATA/SD) of the board. Prerequisites:

- Target board is running with LSDK distro.
- System date is set correctly.

To update the boot partition, run the following steps within the Linux environment of the board:

a. Download the boot partition tarball onto the target board by using either the `wget` or `scp` command.

b. Extract the boot partition tarball to the directory `/boot` using the command `tar zxvf bootpartition_arm64_<date>.tgz -C /boot`

c. Extract `<arch>-modules-<version>.tgz` to the `/lib/modules` directory of the Ubuntu rootfs on the board using the command cd /lib/modules && tar zxvf <arch>-modules-<version>.tgz.

In case of using non-default device tree file for specific use case, follow the example as below:

Example: To replace default fsl-ls1046a-rdb-sdk.dtb with fsl-ls1046a-rdb-usdpaa.dtb on the boot partition of Ubuntu system on Layerscape platform do the following:

1. `cd /boot`
2. `mv fsl-ls1046a-rdb-sdk.dtb fsl-ls1046a-rdb-ori.dtb`
3. `ln -s fsl-ls1046a-rdb-usdpaa.dtb fsl-ls1046a-rdb-sdk.dtb`

Then reboot the board, the `fsl-ls1046a-rdb-usdpaa.dtb` will be used by Linux.

Rebuild images after modifying the 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.

1. Modify source code of user space components in the directory `packages/apps/<apps-component>`.
2. Clean old build footprint under user space component.
3. If necessary, build the kernel which is a dependency for the current application component (e.g. for FMC component)

   ```
   $ flex_builder -c linux -a arm64
   ```
4. Build the user space component.

   ```
   $ flex_builder -c <apps-component>  -a arm64     # <apps-component>  can be restool,
   openssl, dpdk, fmc, ptpd, etc
   ```
5. Generate the boot partition tarball.

   ```
   $ flex-builder -i mkbootpartition -a arm64
   ```
6. Generate the compressed components tarball.

   ```
   $ flex-builder -i compressapps -a arm64
   ```
7. Download the files `components_arm64.tgz` and `bootpartition_arm64.tgz` to the target board using either the `wget` or `scp` command.
8. Extract the boot partition tarball to the directory `/boot`.

   ```
   $ tar zxvf bootpartition_arm64 -C /boot
   ```
9. Extract `components_arm64.tgz` to `$HOME` directory.

   ```
   $ tar zxvf components_arm64.tgz -C $HOME
   ```
10. Copy all files in `$HOME/components_arm64` to the root filesystem.

    ```
    $ cp -a $HOME/components_arm64/* /
    ```

Native build and deploying LSDK images on the target board

If LSDK Ubuntu 18.04 userland is already deployed on the target board, native build can be done using the following instructions. Correspondingly, the redeployment with different media (USB/SATA/SD) can also be done directly on the target board.

Prerequisites:

- Set the system date and time by using the `date -s` command.
- To build Ubuntu userland, the user's network environment must have access to the remote Ubuntu official server.

- Deploy Ubuntu 18.04 userland with full package list on target board. For example, `flex-builder -i mkrfs -a arm64.`
- SD/USB/SATA capacity must at least be 16 GB.

Follow the instructions below to perform native build and deploy LSDK images on the target board.

1. Rebuild kernel and deploy kernel images on target board. For more information on how to modify the kernel source code and how to rebuild the kernel, refer to section above: "*Build custom kernel and update boot partition images with Flexbuild*".

   After completing the kernel image build, use the command below to deploy kernel images and kernel modules on target board:

   ```
   $ cp -a $HOME/flexbuild/build/images/bootpartition_arm64/* /boot
   ```

2. Modify and build source of user space application and deploy changes on target board.

   To modify user space application source code, refer to step 1 through step 6 of the section above: "*Rebuild images after modifying the source code of NXP user space components locally*".

   After completing the user space application build, run the following steps to deploy changes on the target board.

   ```
   $ cp -a $HOME/flexbuild/build/apps/components_arm64/* /
   $ cp -a $HOME/flexbuild/build/images/bootpartition_arm64_lts_<version>/* /boot
   ```

3. Generate LSDK composite firmware and update firmware on target board.

   Take the LS1043RDB booting from IFC NOR as an example:

   ```
   $ flex-builder -i mkfw -m ls1043ardb -b nor
   ```

   NOTE

   firmware is generated in the directory `$HOME/flexbuild/build/images/firmware_ls1043ardb_uboot_norboot.img`

   Additional steps to be run, to generate secure firmware image

   ```
   $ flex-builder -i signimg -m ls1043ardb -b nor
   $ flex-builder -i mkfw -m ls1043ardb -b nor -s
   ```

   For more information on how to update the firmware, refer to section "How to program firmware to SD/NOR/QSPI flash media".

   NOTE

   secure firmware is generated in the directory $HOME/flexbuild/build/images/firmware_ls1043ardb_uboot_norboot_secure.img

How to build Linux and U-Boot with various repository and branch

To select various git repository and branch rather than the default repo and branch, run the following commands:

```
$ flex-builder -c linux:<repo-name>:<tag-name> -a <arch>
$ flex-builder -c linux:<repo-name>:<branch-name> -a <arch>
$ flex-builder -c uboot:<repo-name>:<tag-name> -m <machine> -b <boottype>
$ flex-builder -c uboot:<repo-name>:<branch-name> -m <machine> -b <boottype>
Example:
$ flex-builder -c linux:linux:LSDK-18.09-V4.14 -a arm64
$ flex-builder -c linux:dash-lts:linux-4.9 -a arm32
$ flex-builder -c uboot:u-boot:LSDK-18.09 -m ls1043ardb -b sd
$ flex-builder -c uboot:dash-uboot:devel -m ls2088ardb -b nor
```

To use a private Linux git repository instead of the official git repository, put a private Linux git repository `<custom-linux>` in `packages/linux directory`, then run flex-builder -c linux:<custom-linux>:<branch> -a <arch>

How to generate boot partition images for distro in Flexbuild

Use the following commands to generate boot partition images for distro in Flexbuild:

```
$ flex-builder -i mkbootpartition -a arm64
$ flex-builder -i mkbootpartition -a arm32
or
$ flex-builder -i mkbootpartition -a arm64 -s (for secure boot)
$ flex-builder -i mkbootpartition -a arm32 -s (for secure boot)
```

Generate all needed images including kernel image, dtb files, boot scripts for U-Boot, `grub.conf` for UEFI, `flex_linux_<arch>.itb`, `small ramdiskrfs`, etc. The flex-builder script will automatically build dependent images if not present.

How to build Linux kernel in Flexbuild

To build the kernel using the default configurations specified in configs/build_lsdk.cfg, run the following commands:

```
$ flex-builder -c linux -a arm64              #for 64-bit mode of all armv8 platforms by
default
$ flex-builder -c linux -a arm32              #for 32-bit mode of all armv8 platforms by
default
$ flex-builder -c linux -a arm32 -m ls1021atwr #for 32-bit armv7 ls1021atwr

To build kernel with specified linux repo, specific branch and addtional fragment config,
run as below:
$ flex-builder -c linux:<kernel-repo>:<branch> -a arm64 -B fragment:<custom-fragment>.config
Example:
flex-builder -c linux:dash-lts:linux-4.14 -a arm64 -B fragment:lttng.config

Optionally, users can build kernel for big-endian system (e.g. on LS1043ARDB, LS1046ARDB
platforms) as below:
$ flex-builder -c linux -a arm64:be
```

To select a different Linux git tree and a different branch instead of default configuration, refer to "*How to select various git trees and branches for Linux and U-Boot*".

To change the default kernel `config` to customize kernel, there are two ways to customize `configs` to build the kernel.

1. Run `flex-builder -c linux:custom -a arm64` to customize kernel config in interactive menu, then run `flex-builder -c linux -a arm64` to compile kernel with customized kernel config
2. Put user-specific `configs` (e.g. custom1.config, custom2.config) in `packages/linux/<kernel-repo>/arch/arm64/configs` and run flex-builder -c linux -B fragment:"custom1.config custom2.config"

How to build U-Boot in Flexbuild

Use the commands below to build U-Boot and ATF in Flexbuild

```
$ flex-builder -c uboot -m <machine> -b <boottype>  #build uboot for <machine> to generate
specified nor/sd/qspi/nand boot image
or
$ flex-builder -c uboot -m <machine> #build uboot for <machine> to generate all nor/sd/qspi/
nand boot images
or
$ flex-builder -c uboot -m all         #build uboot for all machines to generate all nor/sd/
qspi/nand boot images
```

How to build application components in Flexbuild

The following commands are some examples of building application components

```
$ flex-builder -c <component> -a <arch>  #build single application component for specified
<arch>
$ flex-builder -c apps                     #build all apps components for arm64 arch
$ flex-builder -c dpdk                    #build DPDK component for SoCs integrated DPAA1
or DPAA2
$ flex-builder -c ovs-dpdk                 #build OVS-DPDK component
$ flex-builder -c fmc -a arm32             #build FMC component for arm32 arch
$ flex-builder -c fmc -a arm64             #build FMC component for arm64 arch
$ flex-builder -c restool                 #build RESTOOL component for arm64 arch, default
arch is arm64
$ flex-builder -c ptpd                     #build ptpd component for IEEE 1588 on arm64
platform
```

```
$ flex-builder -c edgescale                 #build edgescale-related components 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)
```

How to add new application component in Flexbuild

Follow the instructions below to add an application component in Flexbuild:

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

How to generate a custom Ubuntu root filesystem with custom additional package list during the build stage

In Flexbulid, there are two default additional package lists for Ubuntu or Debian: `additional_packages_list_moderate`, and `additional_packages_list_tiny`.

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

Optionally, if you do not want to use default Ubuntu userland in some use cases, you can generate buildroot-based small userland by following instruction by Flexbuild, for examples:

```
$ flex-builder -i mkrfs -r buildroot:tiny -a arm64      (generate arm64 LE buildroot userland
with qoriq_arm64_tiny_defconfig)
$ flex-builder -i mkrfs -r buildroot:moderate -a arm64  (generate arm64 LE buildroot userland
with qoriq_arm64_moderate_defconfig)
$ flex-builder -i mkrfs -r buildroot:custom -a arm64    (generate arm64 LE buildroot userland
with custom qoriq_arm64_moderate_defconfig)
$ flex-builder -i mkrfs -r buildroot:custom -a arm64:be (generate arm64 big-endian buildroot
userland with custom qoriq_arm64_moderate_defconfig)
$ flex-builder -i mkrfs -r buildroot:imaevm -a arm64    (generate arm64 LE initramfs userland
with qoriq_arm64_imaevm_defconfig used for secure boot with IMA EVM)
```

To install a new package to `build/rfs/rootfs_ubuntu_bionic_arm64 filesystem`, run the following commands:

```
$ sudo chroot build/rfs/rootfs_ubunutu_bionic_arm64
$ apt-get install <new_package_name>
```

How to enable or disable various components in Flexbuild

Set `CONIFG_BUILD_<component>` to `y` or `n` in `configs/build_xx.cfg` to include/exclude the specified <component>.

How to generate composite firmware and boot partition

- To generate the following firmware in `build/images` directory for all machines (`<boottype>` can be `nor`, `qspi`, `sd`), run the following commands for examples:

  ```
  $ flex-builder -i mkfw -m ls1043ardb -b sd -B uboot
    firmware_ls1043ardb_uboot_sdboot.img will be generated.

  $ flex-builder -i mkfw -m ls1046ardb -b qspi -B uefi
    firmware_ls1046ardb_uefi_qspi.img will be generated.

  $ flex-builder -i mkfw -m ls2088ardb -b nor -B uboot
    firmware_ls2088ardb_uboot_norboot.img will be generated.
  ```

```
$ flex-builder -i signimg -m ls2088ardb -b nor
$ flex-builder -i mkfw -m ls2088ardb -b nor -B uboot -s
  firmware_ls2088ardb_uboot_norboot_secure.img will be generated for secure boot.

Similarly, the following all composite firmware can be generated by command 'flex-builder
-i mkfw -m <machine> -b <boottype> [-s]'.
firmware_ls1012ardb_uboot_qspiboot.img
firmware_ls1012ardb_uboot_qspiboot_secure.img
firmware_ls1012aafrwy_uboot_qspiboot.img
firmware_ls1012aafrwy_uboot_qspiboot_secure.img
firmware_ls1021atwr_uboot_norboot.img
firmware_ls1021atwr_uboot_norboot_secure.img
firmware_ls1021atwr_uboot_sdboot.img
firmware_ls1043ardb_uboot_norboot.img
firmware_ls1043ardb_uboot_norboot_secure.img
firmware_ls1043ardb_uboot_nandboot.img
firmware_ls1043ardb_uboot_nandboot_secure.img
firmware_ls1043ardb_uboot_sdboot.img
firmware_ls1043ardb_uboot_sdboot_secure.img
firmware_ls1043ardb_uefi_norboot.img
firmware_ls1046ardb_uboot_qspiboot.img
firmware_ls1046ardb_uboot_qspiboot_secure.img
firmware_ls1046ardb_uboot_sdboot.img
firmware_ls1046ardb_uboot_sdboot_secure.img
firmware_ls1046ardb_uefi_qspiboot.img
firmware_ls1088ardb_uboot_qspiboot.img
firmware_ls1088ardb_pb_uboot_qspiboot.img
firmware_ls1088ardb_uboot_qspiboot_secure.img
firmware_ls1088ardb_pb_uboot_qspiboot_secure.img
firmware_ls1088ardb_uboot_sdboot.img
firmware_ls2088ardb_uboot_norboot.img
firmware_ls2088ardb_uboot_norboot_secure.img
firmware_ls2088ardb_uboot_qspiboot.img
firmware_ls2088ardb_uboot_qspiboot_secure.img
firmware_ls2088ardb_uefi_norboot.img
```

- To generate `bootpartition_arm64_<version>.tgz` or `bootpartition_arm32_<version>.tgz` run following commands:

  ```
  $ flex-builder -i mkbootpartition -a <arch>
  or
  $ flex-builder -i mkbootpartition -a <arch> -s  (option '-s' is for secureboot)
  ```

How to add support for a new custom machine in flexbuild based on LSDK release

Assumption: Adding a new custom machine named LS1043AXX based on LS1043A SoC

1. Run `flex-builder -i repo-fetch` to fetch all git repositories of LSDK components for the first time
2. Add new machine support in U-Boot. For example: LSDK-18.09:

   ```
   $ cd packages/firmware/u-boot
   $ git checkout LSDK-18.09 -b LSDK-18.09-LS1043AXX
   ```

   Add U-Boot patches and commit the patches in this branch for new machine.

   Assume that you have added `packages/firmware/u-boot/configs/ls1043axx_sdcard_defconfig` and added CONFIG_MACHINE_LS1043AXX=y

   in configs/build_lsdk.cfg, then run command below to build U-Boot image for SD boot on LS1043AXX.

   ```
   flex-builder -c uboot -m ls1043axx -b sd
   ```
3. Add new machine support in Linux kernel. For example: LSDK-18.09:

   ```
   $ cd packages/linux/linux
   $ git checkout LSDK-18.09-V4.14 -b LSDK-18.09-V4.14-LS1043AXX
   ```

   now you can add kernel patches and submit the patches in this branch for the new machine, then make a tag as below

   ```
   $ git tag LSDK-18.09-V4.14-LS1043AXX
   ```

Assume that you have added a new ls1043axx.dts in `packages/linux/linux/arch/arm64/boot/dts directory`

Add `ls1043axx.dtb to ARM64_DTB_LIST` in packages/linux/Makefile, then Run command below to build kernel image for new LS1043AXX.

```
$ flex-builder -c linux:linux:LSDK-18.06-V4.14-LS1043AXX
```

4. Add configs in flexbuild for new machine.

   Add ls1043axx node in configs/linux/linux_arm64.its

   Add CONFIG_MACHINE_LS1043AXX=y in configs/build_lsdk.cfg

   Set `linux_repo_tag to LSDK-18.09-V4.14-LS1043AXX` and `set u_boot_repo_tag to LSDK-18.09-LS1043AXX` in `configs/build_lsdk.cfg`.

   Set BUILD_DUAL_KERNEL to n in configs/build_lsdk.cfg if user doesn't need the second version of linux.

   optionally, user can use different memory layout from default settings by modifying them in `configs/board/common/memorylayout.cfg`.

   Add manifest for new machine `$ mkdir configs/board/ls1043axx $ cp configs/board/ls1043ardb/manifest configs/board/ls1043axx`.

   Update the settings in `configs/board/ls1043axx/manifest` on demand.

   NOTE

   Generally, user can reuse the settings of `rcw/ppa/fman/qe/eth-phy` used for existing ls1043ardb if those components are same for new ls1043axx, otherwise user needs to add new support in `packages/firmware/rcw`.

   Run `flex-builder -i mklinux -a arm64` to generate `lsdk_linux_arm64_tiny.itb` image.

   Run `flex-builder -i mkfw -m ls1043ardb -b sd` to generate the shared `ppa/rcw/fman/qe/eth-phy` images for new ls1043axx.

   Run `flex-builder -i mkfw -m ls1043axx -b sd` to generate `firmware_ls1043axx_uboot_sdboot.img` user can boot the new `lsdk_linux_arm64_tiny.itb` from U-Boot prompt duringdebugging stage on LS1043AXX machine

   ```
   => tftp a0000000 lsdk_linux_arm64_tiny.itb
   ```

   ```
   => bootm a0000000#ls1043axx
   ```

5. Build all other images for new custom machine with Ubuntu userland if required as below:

   ```
   $ flex-builder -i mkrfs -a arm64
   $ flex-builder -c apps -a arm64
   $ flex-builder -i mkbootpartition -a arm64
   $ flex-builder -i merge-component -a arm64
   $ flex-builder -i compressrfs -a arm64
   ```

   Finally, install `bootpartition_arm64_lts_<version>.tgz` and `rootfs_ubuntu_<codename>_<arch>.tgz` to SD/USB/SATA storage drive on LS1043AXX machine by `flex-installer`.

4.3 Advanced Use Case Instructions

Section “LSDK Quick Start” states the quick way of deploying LSDK distro onto the target board, this section elaborates the instructions.

How to install distro to SD/USB/SATA storage drive

Use the LSDK flex-installer to install all the release binaries and distro userland onto a storage media (e.g. SD/eMMC card, USB/SATA disk) on the Linux host machine or on the target board.

Follow the instructions below:

Step 1: Plug SD/USB/SATA storage device to Linux Host machine or target board

Step 2: Install LSDK distro

- If the prebuilt distro tarball generated by Flexbuild is available on Linux host machine, run the following command:

  ```
  $ flex-installer -b bootpartition_xx.tgz -r ubuntu_bionic_arm64_rootfs_xx.tgz -m <machine>
  -d /dev/sdx
  ```

  NOTE

  `sdx` should be the actual device name on the host machine, for example: `sdb, sdc, mmcblk0`, etc.

- If the user wants to modify source code and build a custom LSDK distro with flexbuild, use the commands below 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 -c firmware -m <machine>
  $ flex-builder -i mkfw -m <machine> -b <boottype> -B uboot
  $ 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_arm64 -r build/rfs/
  rootfs_ubuntu_bionic_arm64 -m <machine> -d /dev/sdx
  ```

- If the user wants to install disrto rootfs directly onto SD/USB/SATA disk on QorIQ board on which Linux is unavailable, download the prebuilt `flex_linux_<arch>.itb` image using the command:

  ```
  $ wget http://www.nxp.com/lgfiles/sdk/lsdk1806/lsdk_linux_arm64_tiny.itb
  ```

  Optionally, locally build it using the command below:

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

  Put the `lsdk_linux_<arch>_tiny.itb` to a TFTP service directory, then download it to the target board under U-Boot prompt as below:

  — For arm64 platforms:

  ```
  => tftp a0000000 lsdk_linux_arm64_tiny.itb
  => bootm a0000000#<board-name>
  ```

  The <board-name> can be: ls1012ardb, ls1012afrdm, ls1043ardb, ls1046ardb, ls1088ardb, ls2088ardb.

  — For arm32 platforms:

  ```
  => tftp a0000000 lsdk_linux_arm32_tiny.itb
  => bootm a0000000#<board-name>
  ```

  The <board-name> can be: ls1012ardb, ls1012afrdm, ls1021atwr, ls1043ardb, ls1046ardb

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

  ```
  $ flex-installer -i pf -d /dev/sdx
  $ cd /run/media/{mmcblk0p3 or sdx3}, then download distro images to sd/usb/sata storage
  disk via wget or scp command
  $ flex-installer -i install -b bootpartition_arm64_lts_<version>.tgz -r
  rootfs_ubuntu_bionic_arm64.tgz -m <machine> -d /dev/sdX
  ```

Step 3: Power on or reboot the target board after finishing the distro installation, the system will enter boot loader (U-Boot or UEFI) and automatically scan boot configuration script from the attached SD/USB/SATA disk and boot the target LSDK distro if found, otherwise it falls back to boot from NOR/QSPI flash with `flex_linux_<arch>.itb`.

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

- **For SD/eMMC card (on all platforms):**

1. Download the prebuilt image (take LS1043ARDB for example):
   - **Option 1:** Load the prebuilt image from SD card in U-Boot:

     ```
     => load mmc 0:2 a0000000 firmware_ls1043ardb_uboot_sdboot.img
     ```

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

     `http://www.nxp.com/lgfiles/sdk/lsdk1806/lsdk_linux_arm64_tiny.itb`

   - **Option 3:** To generate `firmware_ls1043ardb_uboot_sdboot.img` locally, run `flex-builder -i mkfw -m ls1043ardb -b sd -B uboot`.
2. Program `firmware_<machine>_uboot_sdboot.img` to SD card:
   - **Under U-Boot:**

     ```
     => load mmc 0:2 a0000000 firmware_ls1043ardb_uboot_sdboot.img
     => mmc write a0000000 8 1fff8 (same on all platforms)
     => cpld reset sd
     ```

   - **Under Linux:**

     ```
     $ flex-installer -f firmware_ls1043ardb_uboot_sdboot.img -s 8 -d /dev/mmcblk0
     ```

- **For IFC-NOR flash**
  - On LS1043ARDB & LS1021ATWR:
    1. Download the prebuilt image (take LS1043ARDB for example) using the following options:
       - **Option 1:** Load prebuilt image from SD/USB/SATA disk:

         ```
         => load mmc 0:2 a0000000 firmware_ls1043ardb_uboot_norboot.img
         ```

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

         `http://www.nxp.com/lgfiles/sdk/lsdk1806/firmware_ls1043ardb_uboot_norboot.img`

       - **Option 3:** To generate `firmware_ls1043ardb_uboot_norboot.img` locally, run `flex-builder -i mkfw -m ls1043ardb -b nor -B uboot`.
    2. Program `firmware_<machine>_uboot_norboot.img` into IFC-NOR flash:
       - To program current bank:

         ```
         => protect off 60000000 +$filesize && erase 60000000 +$filesize && cp.b
         a0000000 60000000 $filesize
         ```

       - To program alternate bank:

         ```
         => protect off 64000000 +$filesize && erase 64000000 +$filesize && cp.b
         a0000000 64000000 $filesize
         ```

  - On LS2088ARDB:
    1. Download the image using the following options:
       - **Option 1:** Load prebuilt image from SD card

         ```
         => load mmc 0:2 a0000000 firmware_ls2088ardb_uboot_norboot.img
         ```

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

         `http://www.nxp.com/lgfiles/sdk/lsdk1806/firmware_ls2088ardb_uboot_norboot.img`

       - **Option 3:** To `firmware_ls2088ardb_uboot_norboot.img` generate locally, run `flex-builder -i mkfw -m ls2088ardb -b nor -B uboot`.
    2. Program `firmware_ls2088ardb_uboot_norboot.img` into IFC-NOR flash:
       - To program current bank:

         ```
         => protect off 580000000 +$filesize && erase 580000000 +$filesize && cp.b
         a0000000 580000000 $filesize
         ```

       - To program alternate bank:

```
=> protect off 584000000 +$filesize && erase 584000000 +$filesize && cp.b
a0000000 584000000 $filesize
```

- **For QSPI flash:**
  - — On LS1046ARDB:
    1. Download the image using the following options:
       - ◦ **Option 1:** Load prebuilt image from SD card.

         ```
         => load mmc 0:2 a0000000 firmware_ls1046ardb_uboot_qspiboot.img
         ```
       - ◦ **Option 2:** Download the prebuilt image using the `wget` command.

         ```
         http://www.nxp.com/lgfiles/sdk/lsdk1806/firmware_ls1046ardb_uboot_qspiboot.img
         ```
       - ◦ **Option 3:** To generate `firmware_ls1046ardb_uboot_qspiboot.img` locally, run `flex-builder -i mkfw -m ls1046ardb -b qspi -B uboot`.
    2. Program `firmware_ls1046ardb_uboot_qspiboot.img` to QSPI flash:

       ```
       => sf probe 0:1
       => sf erase 0 +$filesize && sf write 0xa0000000 0 $filesize
       ```
  - — On LS1088ARDB:
    1. Download the image using the following options:
       - ◦ **Option 1**: Load prebuilt image from SD card.

         ```
         => load mmc 0:2 a0000000 firmware_ls1088ardb_uboot_qspiboot.img
         ```
       - ◦ **Option 2**: Download the prebuilt image using the wget command.

         ```
         http://www.nxp.com/lgfiles/sdk/lsdk1806/firmware_ls1088ardb_uboot_qspiboot.img
         ```
       - ◦ **Option 3**: To generate `firmware_ls1088ardb_uboot_qspiboot.img` locally, run `flex-builder -i mkfw -m ls1088ardb -b qspi -B uboot`.
    2. Program `firmware_ls1088ardb_uboot_qspiboot.img` to QSPI flash:

       ```
       => sf probe 0:1
       => sf erase 0 +$filesize && sf write 0xa0000000 0 $filesize
       ```

4.4 Procedure to Run Secure Boot

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

4.4.1 Prepare board for Secure Boot

Steps to blow fuses

1. Enable POVDD
   a. LS1021A TWR Board
      - To enable SNVS in check state – J 11
      - POVDD (J8 and J9)

   b. LS1043A RDB Board
      - Put J13 to enable PWR_PROG_SFP

   c. LS1012A RDB
      - Through i2c transactions you need to write to LDO1CT register to change LDO1EN bit in vr5100
      - i2c mw 0x08 0x6c 0x10

d. LS1012 AFRWY

- **J37** to enable PROG_SFP
- Through i2c transactions you need to write to LDO1CT register to change LDO1EN bit in vr5100
- i2c mw 0x08 0x6c 0x10

e. LS1046A RDB Board

- Put J21 to enable PWR_PROG_SFP

f. LS2088A RDB Board

- Put J12 to enable PWR_PROG_SFP

g. LS1088A RDB Board

- Put J10 to enable PWR_PROG_SFP.

2. Write the required values to be fused in the corresponding SFP Registers. Check SFP Block Guide in the SoC RM for details..
3. Blow the fuses by writing PROGFB (0x2) in the INST field in INGR register of SFP.

Blowing One Time Programmable Master Key (OTPMK) fuse using the above procedure

- Check initial SNVS state

```
md 1e90014
88000900
```

The second nibble indicates that the OTPMK is not blown

- Enable POVDD
- Command to generate OTPMK

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

NOTE

For more information on gen_otpmk_drbg, refer Code Signing Tool section in Secure Boot User Guide

- 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>
```

- 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 .

Use the below command write to INGR register :

For LS1, LS1043 and LS1046 use :

```
mw 1e80020 0x02000000
```

For LS1088 and LS2088 use the below command:

```
mw 1e80020 0x2
```

- Reset and check that SNVS is in Check state

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

4.4.2 Running secure boot on target platforms

1. **Platforms LS1021, LS1043, LS1046**

   a. After copying images to flash, select the boot source by changing the switch settings, then boot the board.

   b. In platforms LS1021, LS1043, LS1046 flexbuild generated rcw for secure boot has the boot core put in holdoff 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 in the above commands to be used are ls1043a for ls1043, ls1046 and ls1012.
   - LS1021ATWR board use config command as ccs::config_chain {ls1020a dap {8 1} }

2. **Platforms LS1088, LS2088**

   In these platforms key hash is written into registers by putting the core into RSP, after this, connect to the board and blow SRKH using CCS. When using pre-built images, use the SRK hash present in srk_hash.txt from github.

   If running in production environment (See the note below for more information), i.e if the SRKH fuses are already blown, then no need to put the SoC into RSP, just change the bank/boot-source and boot, else follow the steps below:

   a. **Steps to put SOC in RSP (Reset Sequence Pause)**

      i. LS2088:

      - Rev1 RDB Board Switch (Rev B): SW3.8 – 0. Switch (Rev C to Rev F): SW4.8 – 0. To boot from vbank4, change SW9[3:5] to 100.

      ii. LS1088:

      - U-Boot Command to put SOC in RSP

```
sd secure boot:   i2c mw 66 60 20;i2c mw 66 66 7f;i2c mw 66 10 10;i2c mw
66 10 21
qspi secureboot : i2c mw 66 50 20 ;i2c mw 66 66 7f;i2c mw 66 10 20;i2c mw
66 10 21
```

b. After putting the SoC into RSP, reset the board. Then use the below commands to write SRKHR in the register.

```
ccs::config_chain {<platform> sap2}
display ccs::get_config_chain
puts "Entry RSP: "
ccs::write_mem 2 0x7 0x001000D0 0x4 0x0 0
set ::littleendian(2) 1
ccs::write_mem <sap position> 0x1e80254 4 0 <SRKH1>
ccs::write_mem <sap position> 0x1e80258 4 0 <SRKH2>
ccs::write_mem <sap position> 0x1e8025c 4 0 <SRKH3>
ccs::write_mem <sap position> 0x1e80260 4 0 <SRKH4>
ccs::write_mem <sap position> 0x1e80264 4 0 <SRKH5>
ccs::write_mem <sap position> 0x1e80268 4 0 <SRKH6>
ccs::write_mem <sap position> 0x1e8026c 4 0 <SRKH7>
ccs::write_mem <sap position> 0x1e80270 4 0 <SRKH8>

set ::littleendian(2) 0
puts "Exiting RSP: "
ccs::write_mem 2 0x7 0x001000D0 0x4 0x0 0x400;
```

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

NOTE

- Platform in the above commands to be used are: ls2085a for ls2088 and ls1080a for ls1088.
- To blow SRKH in production environment follow procedure similar to blowing OTPMK fuses. For more detail on production and development environment refer Flow A and Flow B under Product Execution section in Secure Boot User Guide

4.4.3 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>
 Eg. Key_id = 0x20000000
```

NOTE

For more information on key identifier, refer Section 1.7.2 of Secure Boot Guide.

3. Signing all images

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

4. Generating firmware image

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

5. Generating bootpartition

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

6. Writing image to sd card

```
$ flex-installer -b build/images/bootpartition_arm64_lts_<version>.tgz -r build/rfs/
rootfs_ubuntu_bionic_arm64 -m <platform> -d /dev/sdx
```

BOOT FLOW

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

1. By defult the enacap and decap bootscripts will be installed in the bootpartition.
2. When the board boots up for the first time after all images have been generated, Encap bootscript will execute. This bootscript:
    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 bootscript and header with the decap bootscript and it's header, already present in the bootpartition.
    c. Issues reset

Subsequent Boot

1. Uboot would execute script with decap commands
    a. Un-blobify linux and dtb image in DDR
    b. Pass control to these images

NOTE

Chain of trust with confidentiality is currently not supported for LS1012 in flexbuild

4.5 LSDK Memory Layout

Flash layout

The following table shows the memory layout of various firmware stored in NOR/NAND/QSPI flash device or SD card on all QorIQ Reference Design Boards.

Table 9. Flash layout

Definition	Max Size	NOR/QSPI/NAND Flash Offset	SD Card Start Block No.
RCW+PBI	1MB	0x00000000	0x00008
Boot firmware (U-Boot or UEFI)	2MB	0x00100000	0x00800
Boot firmware Environment	1MB	0x00300000	0x01800
PPA firmware	2MB	0x00400000	0x02000
Secure boot headers	3MB	0x00600000	0x03000
DPAA1 FMAN ucode	256KB	0x00900000	0x04800
QE/uQE firmware	256KB	0x00940000	0x04A00
Ethernet PHY firmware	256KB	0x00980000	0x04C00

Table continues on the next page...

Table 9. Flash layout (continued)

Reserved		256KB	0x009C0000	0x04E00
DPAA2 MC or PFE firmware		3MB	0x00A00000	0x05000
DPAA2 DPL		1MB	0x00D00000	0x06800
DPAA2 DPC		1MB	0x00E00000	0x07000
Device tree (needed by UEFI)		1MB	0x00F00000	0x07800
Kernel	lsdk_linux_<arch>.itb	16MB	0x01000000	0x08000
Ramdisk RFS		32MB	0x02000000	0x10000

Storage layout on SD/USB/SATA for LSDK images deployment

With LSDK flex-installer, the LSDK distro can be installed into an SD/USB/SATA storage disk which should have at least 8GB of memory space.

Table 10. Storage Layout on SD/USB/SATA for LSDK Image Deployment

Region 1 (4KB)	Region 2 (RAW) 64MB Firmware	Region 3 (Partition-1 FAT32) 20MB EFI	Region 4 (Partition-2 EXT4) 1GB Boot partition	Region 5 (Partition-3 EXT4) Remaining space RootFS partition
MBR/GPT	RCW U-Boot or UEFI PPA firmware Secure boot headers FMan 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

4.6 Board-specific Information

4.6.1 TWR-LS1021A

This section provides TWR-LS1021A-specific information on switch setting configurations, U-Boot environment variable settings as well as supported binaries. It also provides a description of the virtual banks and flash memory layouts. For more information about the TWR-LS1021A refer to QorIQ TWR-LS1021A Reference Manual and the QorIQ TWR-LS1021A Getting Started Guide

4.6.1.1 On-board Switch Settings

Default Settings

Table 11. Default TWR-LS1021A PB Switch Settings

Switch	1	2	3	4	5	6	7	8
SW2	ON [1]	OFF [0]	OFF [0]	OFF [0]	ON [1]	ON [1]	ON [1]	ON [1]
SW3	OFF	ON	ON	OFF	OFF	ON	OFF	ON

Table 12. Default Clock Frequency

ARM CPU Core	Platform	DDR rate
1200 MHZ	300 MHZ	1600 MT/S

1. **On-board Switch Settings**

   Use the following SW settings :

   Boot options:

   ```
   SW3: 01100101 (No change default value)
   SW2: 10001111 (Nor boot/default value),
   SW2: 0010_0000 (SD-QSPI boot), boot from SD card, and enables QSPI
   SW2: 0010_1000 (SD-IFC boot), boot from SD card, and enables IFC
   SW2: 00010111 (QSPI boot)
   ```

2. **Supported Boot Options**

   SD boot: (on default U-Boot):

   ```
   tftp a0000000 uboot_ls1021atwr_sdcard_ifc.bin or uboot_ls1021atwr_sdcard_qspi.bin
   mmc erase 8 0x800
   mmc write a0000000 8 0x800
   Boot the board from SD card, by changing DIP Switcher to SW2: 0010_0000
   ```

   Qspi boot:

   ```
   On SD uboot:
   RCW:
   tftp a0000000 rcw_1200_qspiboot.bin.swapped sf probe 0:0; sf erase 0 40000; sf write
   a000000 0 40000
   u-boot:
   tftp 81000000 u-boot sf probe 0:0;
   sf erase 100000 100000; sf write 81000000 100000 100000 Switch to QSPI boot
   Boot the board from Qspi, by changing DIP Switcher to SW2: 00010111
   ```

4.6.1.2 Clock Frequency

The Configuration of the Clock Frequency on TWR-LS1021A consist of two parts:

1. From the Hardware side, set the switches on the TWR-LS1021A board for the desired frequency - SW3[3:4]. Details about the frequency options available in the TWR-LS1021AGS.pdf doc.
2. From the SW side since the U-Boot software cannot read out SW3[3:4] settings, change the clock configuration manually in U-Boot software code, so that it matches with hardware switches (`include/configs/ls1021atwr.h`).

Change the following settings to the desired frequency to match the HW switch configuration:

```
#define CONFIG_SYS_CLK_FREQ 100000000
#define CONFIG_DDR_CLK_FREQ 100000000
```

4.6.1.3 U-Boot Environment Variables

`hwconfig`

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

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

  ```
  =>setenv hwconfig
  ```

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

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

The default setting for DDR, which will disable interleaving, is as follows:

```
hwconfig = fsl_ddr:ctlr_intlv=null,bank_intlv=null
```

4.6.1.4 Supported Boot Options

TWR-LS1021A supports the following boot options:

- SD
- QSPI
- NOR

4.6.1.5 System Memory Map

Start Physical Address	End Physical Address	Memory Type	Size
0x0100_0000	0x0FFF_FFFF	CCSR	240MB
0x1000_0000	0x1000_FFFF	OCRAM0	64KB
0x1001_0000	0x1001_FFFF	OCRAM1	64 KB
0x2000_0000	0x20FF_FFFF	DCSR	16MB
0x4000_0000	0x5FFF_FFFF	QSPI	512MB
0x6000_0000	0x67FF_FFFF	NOR Flash	128MB
0x7FB0_0000	0x7FB0_0FFF	Board CPLD	4KB
0x8000_0000	0xFFFF_FFFF	DDR	2GB

4.6.1.6 NOR Flash (Virtual) Banks

NOR flash is a simple and convenient destination for deploying images so it is frequently used. Many NXP development and reference boards provide a special feature that allows a single NOR flash to be divided into multiple parts called “banks”. This is done by board-level logic that modifies address signals. Because there is only one NOR flash physically, the banks are sometimes called "virtual" banks.

The benefit of this feature is that it allows more than one set of images to be independently deployed to the one NOR flash. This is very helpful during development, because you can use the U-Boot image in one bank to program an image set into a different bank. If the new images are flawed, the old images are still functional to let you deploy corrected images.

The logic on the board usually allows the NOR flash to be divided into up to 8 banks, but almost always the feature is used to divide the flash into two halves only. For the TWR-LS1021A, the 2 halves are called bank0 and bank1.

Switch Settings and NOR Banks

The NOR flash on the board can be seen as two flash banks. The board DIP switch configuration (for LS1021ATWR, SW3[5]) preselects bank 0 as the hardware default bank. The NOR flash on the TWR-LS1021A board is divided in two banks. There are different images in each banks that supports the different functionality.

The board DIP switch configuration:

Nor boot:

```
SW2[1:8]=1000 1111
SW3[1:8]=0010 0101
Bank 0:
SW[5]=0
Bank 1:
SW[5]=1
```

Bank0 is programmed with the RCW support for the QE, and the bank1 is programmed with the RCW support for the 2D-ACE.

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

```
CPU:   Freescale LayerScape LS1021E, Version: 2.0, (0x87081120)
Clock Configuration:
       CPU0(ARMV7):1200 MHz,
       Bus:300  MHz, DDR:800  MHz (1600 MT/s data rate),
Reset Configuration Word (RCW):
       00000000: 0608000c 00000000 00000000 00000000
       00000010: 70000000 00007900 e0025a00 21046000
       00000020: 00000000 00000000 00000000 20000000
       00000030: 00080000 881b7340 00000000 00000000
Board: LS1021ATWR
CPLD:  V2.0
PCBA:  V1.0
VBank: 0
```

Bank Switching

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

- Switch to bank 0:

  ```
  =>reset
  ```

- Switch to alternate bank:

  ```
  =>boot_bank 1
  ```

NOR Flash Memory Map

NOR flash memory map for LS1021A TWR can be found on the link below.

Related information

LSDK Memory Layout on page 74

4.6.1.7 Supported Reset Configuration Word (RCW) Binaries

The release contains the following RCW binary for use on the TWR-LS1021A:

- RSR_PPS_70/rcw_1200.bin (default for bank0)
- SSR_PNS_30/rcw_1200.bin
- SSR_PNS_30/rcw_1200_usb2.bin
- SSR_PNS_30/rcw_1200_lpuart.bin (default for bank1)

The RCW directories' names for the TWR-LS1021A conform to the following naming convention:

```
abc_def_g
```

Table 13. TWR-LS1021A Directories Naming Convention Legend

Slot	Convention
a[What is available for eTSEC1]	'R' indicates RGMII@eTSEC1 is supported 'S' indicates SGMII@eTSEC1 is supported 'N' if not available/not used
b[What is available for eTSEC2]	'R' indicates RGMII@eTSEC2 is supported 'S' indicates SGMII@eTSEC3 is supported 'N' if not available/not used
c[What is available for eTSEC3]	'R' indicates RGMII@eTSEC3 is supported 'S' indicates SGMII@eTSEC3 is supported 'N' if not available/not used
d	'P' indicates PCIe@slot1 is supported 'N' if not available/not used
e	'P' indicates PCIe@slot2 is supported 'N' if not available/not used
f	'S' indicates sata is supported 'N' if not available/not used
g	Hex value of serdes protocol value

For example,

```
SSR_PNS_30
```

means:

- SGMII@eTSEC1
- SGMII@eTSEC2
- RGMII@eTSEC3
- PCIE [Slot 1]
- SATA [Support]
- SERDES Protocol is 0x30

The RCW file names for the LS1021ATWR conform to the following naming convention:

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

Table 14. LS1021TWR Files Naming Convention Legend

Code		Convention
frequency		Core frequency(MHZ)
specialsetting	bootmode	sdboot (default is nor boot)
	special support	lpuart:used for lpuart sben:Secure boot support usb2:USB 2.0 support

For example,

```
rcw_1200_sd.rcw means rcw for core frequency of 1200MHz with sd boot.
rcw_1200_lpuart.rcw means rcw for core frequency of 1200MHz with nor boot special for enable
lpuart.
```

Default rcw:

```
SSR_PNS_30/rcw_1200_lpuart.bin[1 SGMII, 1 RGMII, 2D-ACE, lpuart1, 2PCIE, SATA, CAN, SAI]
RSR_PPS_70/rcw_1200.bin[2 SGMII, 1 RGMII, 1SATA, 1PCIE, CAN]
```

4.6.1.8 FlexCAN User Manual

Description

The FlexCAN module is a communication controller implementing the CAN protocol according to the CAN 2.0B protocol specification. NXP’s LS1021A can support 4 Flexcan IP module instances if there are 2 CAN ports brought out on DB9 male connectors on the board.

Dependencies

Hardware:

TWR-IND-IO board is needed to use flexcan with LS1021A-TWR board.

Software:

LSDK Images (refer Deploy LSDK Images on the target board) and can-utils and iproute2 (both packages included in LSDK ubuntu userland)

U-Boot Configuration

Execute the following command at the u-boot prompt:

=> setenv hwconfig “can3”

=> saveenv

Compile time options

N/A

Runtime options

Env Variable	Env Description	Option Description
hwconfig	setenv hwconfig "can3"	CAN3 and USB2 are muxed on LS1021A-TWR. This option is used to select CAN3 feature on the LS1021A-TWR board.

Kernel Configure Options

Tree View

Following is an example of how to enable FlexCAN driver support.

Kernel Configure Tree View Options	Description

`[*] Networking support -->`
`  <*> CAN bus subsystem support -->`
`    -- CAN bus subsystem support`
`    <*> Raw CAN Protocol (raw access with CAN-ID filtering)`
`    < > Broadcast Manager CAN Protocol (with content filtering)`
`       CAN Device Drivers -->`
`         <*> Platform CAN drivers with Netlink support`
`         [*] CAN bit-timing calculation`
`         <*> Support for NXP FlexCAN based chips`
	Enable FlexCAN driver and CAN protocol stack

NOTE

The FlexCAN driver can be statically/dynamically linked with the kernel Image.

Identifier

Option	Values	Default Value	Description
CONFIG_NET	y/n	y	Enable networking support
CONFIG_CAN	y/m/n	n	Enable CAN bus support
CONFIG_CAN_RAW	y/m/n	n	Enable CAN RAW Protocol support
CONFIG_CAN_DEV	y/m/n	n	Enable platform CAN driver support
CONFIG_CAN_CALC_BITTIMING	y/n	n	Enable baud rate setting using sysfs
CONFIG_CAN_FLEXCAN	y/m/n	n	Enable NXP FlexCAN protocol

Device Tree Binding

Below is the definition of the device tree node required by this feature

Property	Description
compatible = " fsl,ls1021ar2-flexcan"	Should be "fsl,ls1021ar2-flexcan”
reg = <0x0 0x2a70000 0x0 0x1000>	Offset and length of the register set for the device
interrupts = < GIC_SPI 126 IRQ_TYPE_LEVEL_HIGH>	IRQ line for FlexCAN controller CAN0

Table continues on the next page...

Table continued from the previous page...

Property	Description
clocks = <&clockgen 4 1>	CAN module Clock Source.
clock-names = <per>	CAN Engine Clock Source. This property selects the peripheral clock. Valid values are **ipg** and **per** **ipg**: CAN engine clock source is oscillator clock. Current release doesn’t support this option. **per:** The CAN engine clock source is the peripheral clock (platform clock).
big-endian	This means the registers of FlexCAN controller are big endian.

Below is an example device tree node required by FlexCAN.

```
can0: can@2a70000 {
    compatible = "fsl,ls1021ar2-flexcan";
    reg = <0x0 0x2a70000 0x0 0x1000>;
    interrupts = <GIC_SPI 126 IRQ_TYPE_LEVEL_HIGH>;
    clocks = <&clockgen 4 1>, <&clockgen 4 1>;
    clock-names = "ipg", "per";
    big-endian;
};
```

Source Files

The following source file is related to Flexcan feature in u-boot.

Source File	Description
arch/arm/cpu/armv7/ls102xa/fdt.c	Providing the fix-up for clock_freq in the dts file.
board/freescale/ls1021atwr/ls1021atwr.c	Provides the support for board level FlexCAN muxing when u-boot environment variable **hwconfig** is set to can3.

The following source files are related to Flexcan feature in Linux kernel.

Source File	Description
drivers/net/can/flexcan.c	Flexcan driver module

Board Connections

For hardware connections refer to LS1021A-TWR_FLEXCAN_A manual.

User Space Application

can-utils and iproute2 (included in LSDK ubuntu userland) are needed for using flexcan.

Command Name	Description	Package Name
cansend	Userspace SocketCAN test program for sending CAN frames on a CAN interface	can-utils
candump	Userspace SocketCAN test program for sniffing CAN frames received on a CAN interface	can-utils

Table continues on the next page...

Table continued from the previous page...

Command Name	Description	Package Name
ip	The iproute package contains networking utilities (ip and rtmon, for example) which are designed to use the advanced networking capabilities of the Linux kernel.	iproute(2)

Verification in U-Boot

N/A

Verification in Linux

To cross check whether FlexCAN has been configured in the kernel or not, run the following command at Linux prompt:

```
root@ls1021aqds:~# cat /proc/interrupts
           CPU0       CPU1
118:       3889          0       GIC 118  serial
120:        126          0       GIC 120  2180000.i2c
125:          0          0       GIC 125  xhci-hcd:usb1
126:       1521          0       GIC 126  mmc0
129:          0          0       GIC 129  2110000.dspi
135:          2          0       GIC 135  1710000.jr
136:          0          0       GIC 136  1720000.jr
137:          0          0       GIC 137  1730000.jr
138:          0          0       GIC 138  1740000.jr
150:       5371          0       GIC 150  Freescale ftm timer
158:          1          0       GIC 158  can0
167:          0          0       GIC 167  eDMA
189:         21          0       GIC 189  eth2_g0_tx
190:          4          0       GIC 190  eth2_g0_rx
191:          0          0       GIC 191  eth2_g0_er
201:          0          0       GIC 201  ds3232
IPI0:          0          0  CPU wakeup interrupts
IPI1:          0       5359  Timer broadcast interrupts
IPI2:       3158       3718  Rescheduling interrupts
IPI3:          0          0  Function call interrupts
IPI4:          3          3  Single function call interrupts
IPI5:          0          0  CPU stop interrupts
Err:          0
```

Test Procedure

Guide lines for testing FlexCAN on the LS1021A board.

Internal LoopBack test (SoC Level loopback)

- Enable CAN interface **can0** with root permissions on the Linux prompt

  ```
  # ip link set can0 up type can bitrate 125000 loopback on
  ```

- Set candump to snif packets on can0 interface:

  ```
  # candump can0 -n 2 &
  ```

- Set cansend to send a packet with standard identifier on can0 interface:

  ```
  # cansend can0 5A1#123412341234
  ```

Expected behavior after internal loopback testing:

- CAN0 interface which is receiving the can frame should show, that it has read the frame ID and data correctly:

  ```
  5A1 [6] 12 34 12 34 12 34
  ```

  — After the successful Tx or RX from the CAN controller, the Tx and Rx Interrupt should increment for CAN0 interface.

  ```
  root@ls1021aqds:~# cat /proc/interrupts
  ```

```
           CPU0       CPU1
118:       3889          0       GIC 118  serial
120:        126          0       GIC 120  2180000.i2c
125:          0          0       GIC 125  xhci-hcd:usb1
126:       1521          0       GIC 126  mmc0
129:          0          0       GIC 129  2110000.dspi
135:          2          0       GIC 135  1710000.jr
136:          0          0       GIC 136  1720000.jr
137:          0          0       GIC 137  1730000.jr
138:          0          0       GIC 138  1740000.jr
150:       5371          0       GIC 150  Freescale ftm timer
158:          2          0       GIC 158  can0
167:          0          0       GIC 167  eDMA
189:         21          0       GIC 189  eth2_g0_tx
190:          4          0       GIC 190  eth2_g0_rx
191:          0          0       GIC 191  eth2_g0_er
201:          0          0       GIC 201  ds3232
IPI0:          0          0  CPU wakeup interrupts
IPI1:          0       5359  Timer broadcast interrupts
IPI2:       3158       3718  Rescheduling interrupts
IPI3:          0          0  Function call interrupts
IPI4:          3          3  Single function call interrupts
IPI5:          0          0  CPU stop interrupts
Err:          0
```

NOTE

The underlined number represents the interrupts on successful Tx or RX. So this number should increase on successful Tx and Rx.

External LoopBack test (Single Board).

- Attach the CAN0 and CAN1 interface on the LS1021A board with serial cable (prepared as mentioned in **Dependencies** section).
- Enable the CAN0 and CAN1 interface on the board:

```
# ip link set can0 up type can bitrate 125000
# ip link set can1 up type can bitrate 125000
```

- Set candump to snif packets on can0 interface:

```
# candump can0 -n 1 &
```

- Set cansend to send a packet with standard identifier on can1 interface:

```
# cansend can1 5A1#123412341234
```

Expected behavior after external loopback testing:

- CAN0 interface which is receiving the can frame should show, that it has read the frame ID and data correctly:

```
5A1 [6] 12 34 12 34 12 34
```

- After the successful Tx or RX from the CAN controllers, the Tx Interrupt should increase for CAN1 interface and Rx Interrupt should increment for CAN0 interface.

Benchmarking

TBD

Known Bugs, Limitations, or Technical Issues

N/A

Supporting Documentation

- SoC Reference Manual (for eg. LS1021A RM)
- QorIQ TWR-LS1021A Reference Manual

- LS1021A-TWR_FLEXCAN_A Manual

4.6.2 LS1012ARDB

This section provides LS1012ARDB-specific information on switch setting configurations, U-Boot environment variable settings as well as supported binaries. It also provides a description of the virtual banks and flash memory layouts. For more information about the LS1012A refer to QorIQ LS1012A Reference Manual and the QorIQ LS1012A Reference Design Board Getting Started Guide

4.6.2.1 On-board Switch Settings

Table 15. LS1012ARDB Switch Settings for Booting from QSPI Bank 1

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
SW1	1	0	1	0	0	1	1	0
SW2	0	0	0	0	0	0	0	0

Table 16. LS1012ARDB Switch Settings for Booting from QSPI Bank 2

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
SW1	1	0	1	0	0	1	1	0
SW2	0	0	0	0	0	0	1	0

4.6.2.2 U-Boot Environment Variables

`hwconfig`

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

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

  ```
  =>setenv hwconfig
  ```
- It can be modified in the U-Boot command prompt using the `editenv` command.
- It can be saved in the U-Boot environment via the `saveenv` command.

Variable `hwconfig` is set to a sequence of `option:value` entries separated by semicolons.The default setting for for `hwconfig` on LS1012ARDB is as follows:

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

4.6.2.3 Supported Boot Options

LS1012ARDB supports the following boot options:

- QSPI

4.6.2.4 System Memory Map

Start Physical Address	End Physical Address	Memory Type	Size

Table continues on the next page...

Table continued from the previous page...

`0x00_0000_0000`	`0x00_000F_FFFF`	Secure Boot ROM	1MB
`0x00_0100_0000`	`0x00_0FFF_FFFF`	CCSR	240MB
`0x00_1000_0000`	`0x00_1000_FFFF`	OCRAM1	64KB
`0x00_1001_0000`	`0x00_1001_FFFF`	OCRAM2	64 KB
`0x00_4000_0000`	`0x00_5FFF_FFFF`	QSPI	512MB
`0x00_8000_0000`	`0x00_FFFF_FFFF`	DRAM	2GB
`0x08_8000_0000`	`0x0F_FFFF_FFFF`	DRAM2	30G
`0x40_0000_0000`	`0x47_FFFF_FFFF`	PCI Express1	32G

4.6.2.5 Supported Reset Configuration Word (RCW) Binaries

There are four RCW binaries, however, the default configuration is - `RR_SPNH_3508/PBL_0x35_0x08_1000_250_1000_default.bin:.`

```
RR_SPNH_3508/PBL_0x35_0x08_1000_250_1000_default.bin
RR_SPNH_3508/PBL_0x35_0x08_1000_250_1000_sben.bin
RR_SPNH_3508/PBL_0x35_0x08_800_250_1000_default.bin
RR_SPNH_3508/PBL_0x35_0x08_800_250_1000_sben.bin
```

`RR_SPNH_3508/PBL_0x35_0x08_1000_250_1000_default.bin:` RCW enables:

- Boot from QSPI
- 1000MHz Core, 250MHz Platfrom, 1000MT/s DDR
- SDHC1, SDHC2, I2C1,
- SerDes Protocol 0x3508
- PCIe, SATA,
- RGMII, SGMII
- USB 3.0

4.6.2.6 Flash Bank Usage

LS1012ARDB has 2 QSPI flash connected over QSPI contoller.

Only one QSPI flash is available at a time depending upon the board switch settings. These switch settings can also be overriden by I2C commands.

To protect the default U-Boot in flash1, it is a convention employed by NXP to deploy work images into flash2, and then switch to flash2 for testing. Switching to flash2 can be done in software using I2C commands and effectively swaps flash1 with flash2. This protects flash1 and keeps the board bootable under all circumstances.

```
U-Boot 2017.07-02268-g109fd35 (Dec 15 2017 - 10:31:00 +0530)

SoC:  LS1012AE Rev2.0 (0x87040020)
Clock Configuration:
       CPU0(A53):1000 MHz
       Bus:      250  MHz  DDR:      1000 MT/s
Reset Configuration Word (RCW):
       00000000: 0800000a 00000000 00000000 00000000
       00000010: 35080000 c000000c 40000000 00001800
       00000020: 00000000 00000000 00000000 00014571
       00000030: 00000000 18c2a120 00000096 00000000
I2C:   ready
DRAM:  958 MiB
Using SERDES1 Protocol: 13576 (0x3508)
PPA Firmware: Version LSDK-17.09-update-103017
WARNING: Calling __hwconfig without a buffer and before environment is ready
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
SF: Detected s25fs512s with page size 512 Bytes, erase size 256 KiB, total 64 MiB
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Model: LS1012A RDB Board
Board: LS1012ARDB Version: unknown, boot from QSPI: bank1
SATA link 0 timeout.
AHCI 0001.0301 32 slots 1 ports 6 Gbps 0x1 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc apst
Found 0 device(s).
SCSI:  Net:   PFE class pe firmware
PFE tmu pe firmware
ls1012a_configure_serdes 0
PCIe0: pcie@3400000 Root Complex: x1 gen1
e1000: 68:05:ca:36:8d:8e
       pfe_eth0
Warning: pfe_eth0 (eth0) using random MAC address - 02:05:3c:25:2e:c3
, pfe_eth1
Warning: pfe_eth1 (eth1) using random MAC address - 2a:0c:d0:10:73:1f
, e1000#0
Hit any key to stop autoboot:  0
```

How to boot from flash 2

1. To check which bank booted, refer to “**Board: LS1012ARDB Version: unknown, boot from QSPI: bank1"** in the U-Boot logs.
2. i2C command to switch from flash1 to flash2 “ **i2c mw 0x24 0x7 0xfc; i2c mw 0x24 0x3 0xf5 “**
3. Program QSPI flash as per flash layout
4. To boot from flash2 give “**reset”** command.
5. To move back to flash1 from flash2, power on/off the board or use **“i2c mw 0x24 0x3 0xf4 “** and then give “**reset**” command.

QSPI flash Layout

Image	Size	Start Address
RCW + PBI	1MB	0x4000_0000
U-boot boot loader	1MB	0x4010_0000
U-boot Env	1MB	0x4030_0000
PPA FIT image	2MB	0x4040_0000
PFE Firmware	20K	0x40A0_0000
Kernel ITB	59MB	0x4100_0000

4.6.2.7 Basic Networking Ping Test

U-BOOT

The LS1012ARDB has one SGMII and one RGMII. The log below shows how to ping from those 2 interfaces.

```
U-Boot 2017.07-02268-g109fd35 (Dec 15 2017 - 10:31:00 +0530)

SoC:  LS1012AE Rev2.0 (0x87040020)
Clock Configuration:
       CPU0(A53):1000 MHz
       Bus:      250  MHz  DDR:      1000 MT/s
Reset Configuration Word (RCW):
       00000000: 0800000a 00000000 00000000 00000000
       00000010: 35080000 c000000c 40000000 00001800
       00000020: 00000000 00000000 00000000 00014571
       00000030: 00000000 18c2a120 00000096 00000000
I2C:   ready
DRAM:  958 MiB
Using SERDES1 Protocol: 13576 (0x3508)
PPA Firmware: Version LSDK-17.09-update-103017
WARNING: Calling __hwconfig without a buffer and before environment is ready
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
SF: Detected s25fs512s with page size 512 Bytes, erase size 256 KiB, total 64 MiB
*** Warning - bad CRC, using default environment

In:    serial
Out:   serial
Err:   serial
Model: LS1012A RDB Board
Board: LS1012ARDB Version: unknown, boot from QSPI: bank1
SATA link 0 timeout.
AHCI 0001.0301 32 slots 1 ports 6 Gbps 0x1 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc apst
Found 0 device(s).
SCSI:  Net:   PFE class pe firmware
PFE tmu pe firmware
ls1012a_configure_serdes 0
PCIe0: pcie@3400000 Root Complex: x1 gen1
e1000: 68:05:ca:36:8d:8e
       pfe_eth0
Warning: pfe_eth0 (eth0) using random MAC address - 22:04:f9:6a:ff:13
, pfe_eth1
Warning: pfe_eth1 (eth1) using random MAC address - 4a:0f:97:59:b8:ec
, e1000#0
Hit any key to stop autoboot:  0
=> edit ipaddr
edit: 192.168.1.136
=> ping 192.168.1.1
Speed detected 3e8
Using pfe_eth0 device
host 192.168.1.1 is alive
=>
```

LINUX

To enable PFE in Linux, first stop PFE in U-Boot. In order to do this, first bring the kernel-ls1012a-rdb.itb via PFE interface then type `pfe stop` command on the U-Boot prompt:

```
=> tftp 0xa0000000  kernel-ls1012a-rdb.itb
Speed detected 3e8
Using pfe_eth1 device
TFTP from server 192.168.1.1; our IP address is 192.168.1.136
Filename 'kernel-ls1012a-rdb.itb'.
Load address: 0xa0000000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         ############
         5.6 MiB/s
done
Bytes transferred = 38342491 (2490f5b hex)
=> pfe stop
Stopping PFE...
=> 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:  uncompressed
     Data Start:   0xa00000dc
     Data Size:    12482048 Bytes = 11.9 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: 0x80080000
     Entry Point:  0x80080000
## Loading ramdisk from FIT Image at a0000000 ...
   Using 'config@1' configuration
   Trying 'ramdisk@1' ramdisk subimage
     Description:  LS2 Ramdisk
     Type:         RAMDisk Image
     Compression:  uncompressed
     Data Start:   0xa0be9ba4
     Data Size:    25849963 Bytes = 24.7 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: unavailable
     Entry Point:  unavailable
```

```
## 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:   0xa0be7790
     Data Size:    9101 Bytes = 8.9 KiB
     Architecture: AArch64
   Loading fdt from 0xa0be7790 to 0x90000000
   Booting using the fdt blob at 0x90000000
   Loading Kernel Image ... OK
   Using Device Tree in place at 0000000090000000, end 000000009000538c
Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 4.1.8-rt8+g2511ec0 (jenkins@neptune) (gcc version 4.9.4 20150629
(prerelease) (Linaro GCC 4.9-2015.06) ) #1 SMP Sat Aug 27 04:44:19 CST 2016
[    0.000000] CPU: AArch64 Processor [410fd034] revision 4
[    0.000000] Detected VIPT I-cache on CPU0
[    0.000000] alternatives: enabling workaround for ARM erratum 845719
[    0.000000] earlycon: Early serial console at MMIO 0x21c0500 (options '')
[    0.000000] bootconsole [uart0] enabled
[    0.046613] No BMan portals available!
[    0.052448] No QMan portals available!
[    0.186759] Freescale FM module, FMD API version 21.1.0
[    0.192162] Freescale FM Ports module
[    0.200605] vfio_fsl_mc_driver_init: Driver registration fails as no fsl_mc_bus found
[    0.658570] fsl-mc bus not found, restool driver registration failed
[    0.927488] usb usb1-port1: over-current condition
[    0.932320] usb usb2-port1: over-current condition
INIT: version 2.88 booting
Starting udev
[    3.038179] pe_load_ddr_section: load address(3fb0000) and elf file
address(ffff0000003fb000) rcvd
Populating dev cache
hwclock: can't open '/dev/misc/rtc': No such file or directory
Fri Aug 26 20:58:57 UTC 2016
hwclock: can't open '/dev/misc/rtc': No such file or directory
Running postinst /etc/rpm-postinsts/100-sysvinit-inittab...
Running postinst /etc/rpm-postinsts/101-inetutils-inetd...
Running postinst /etc/rpm-postinsts/102-inetutils-ftpd...
update-rc.d: /etc/init.d/run-postinsts exists during rc.d purge (continuing)
Removing any system startup links for run-postinsts ...
INIT: Entering runlevel: 5
Configuring network interfaces... done.
Starting system log daemon...0
Starting kernel log daemon...0
Starting internet superserver: xinetd.

QorIQ SDK (FSL Reference Distro) 2.0 ls1012ardb /dev/ttyS0

ls1012ardb login: root
root@ls1012ardb:~# find / -name pfe.ko | xargs insmod
[   30.958325] pe_load_ddr_section: load
address(3fb0000) and elf file address(ffff00000c795000) rcvd

root@ls1012ardb:~# ifconfig -a
eth0      Link encap:Ethernet  HWaddr 68:05:ca:36:8d:8e
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:29 Memory:40400c0000-40400e0000

eth1      Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          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)

eth2      Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          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:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

sit0      Link encap:UNSPEC  HWaddr 00-00-00-00-3A-30-30-30-00-00-00-00-00-00-00-00
          NOARP  MTU:1480  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

root@ls1012ardb:~# ifconfig eth1 hw ether 00:80:48:BA:d1:30
root@ls1012ardb:~# ifconfig eth1 192.168.1.23 up
root@ls1012ardb:~# ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=3 ttl=128 time=1.92 ms
64 bytes from 192.168.1.1: icmp_seq=4 ttl=128 time=4.29 ms
64 bytes from 192.168.1.1: icmp_seq=5 ttl=128 time=0.974 ms
64 bytes from 192.168.1.1: icmp_seq=6 ttl=128 time=1.19 ms
^C
--- 192.168.1.1 ping statistics ---
6 packets transmitted, 4 received, 33% packet loss, time 5036ms
rtt min/avg/max/mdev = 0.974/2.096/4.295/1.318 ms
root@ls1012ardb:~#
```

4.6.2.8 Check 'Link Up' for Serial Ethernet Interfaces

This section provides some basic checks that can be performed in U-Boot to help diagnose the cause of the networking errors when experiencing problems with Ethernet interfaces.

Check Communication to External PHY

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

Example:

```
=> mdio list
PFE_MDIO:
1 - RealTek RTL8211F <--> pfe_eth1
2 - RealTek RTL8211F <--> pfe_eth0
```

The results from the `mdio list` command above show that U-Boot was able to see PHYs on each of the RGMII/SGMII interfaces.

Check Link Status for External PHY

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

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

Example:

```
=> mdio read pfe_eth0 1
Reading from bus PFE_MDIO
PHY at address 2:
1 - 0x79ad
=> mdio read pfe_eth1 1
Reading from bus PFE_MDIO
PHY at address 1:
1 - 0x79ad
```

The link partner (“copper side”) link status bit is in Register #1 on the PHY. The 'Link Status' bit is bit #2 (from the left) of the last nibble. In the example above, the nibble of interest is "d" (d = b'1101'), and therefore the 'Link Status' = 1, which means 'link up'. If the link were down this bit would be a "0," and we would see 0x7989.

4.6.3 LS1012A-FRWY

The Layerscape LS1012A Freeway (LS1012A-FRWY) board is an ultra-low-cost development platform for LS1012A.This board refines the FRDM-LS1012A with more features for a better hands-on experience for IoT, edge computing, and various advanced embedded applications. Features include easy access to processor I/O, low-power operation, micro SDHC card storage, an M2 connector, a small form factor, and expansion board options via mikroBUS™ Click module. The MicroBUS module provides easy expansion via hundreds of powerful modules supporting sensors, actuators, memories and displays. For more details see Layerscape LS1012A Freeway Board Getting Started Guide .

4.6.3.1 Switch Settings

The LS1012A-FRWY does not have On-Board switches.

4.6.3.2 U-Boot Environment Variables

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

4.6.3.2.1 U-Boot environment variables

Some of the important U-Boot environment variables, which need to be set are listed below:

1. Set MAC addresses for the Ethernet ports in the U-Boot.

```
For ETH1:
=> setenv ethaddr <MAC address from the port label>
For Eth2:
=> setenv eth1addr <MAC address from the port label>
```

2. Set server IP to the IP of the Linux host machine on which you have configured the TFTP server.

```
=> setenv serverip <ipaddress1>
For example:
Updating board software and deploying Ubuntu 16.04 userland on the board
=> setenv serverip 10.232.133.88
```

3. Set IP of the board. To check IP of the board connected to network, you can use the dhcp command.

```
=> setenv ipaddr <ipaddress2>
For example:
=> setenv ipaddr 10.232.199.0
```

NOTE

The server IP and board IP address should be in the same domain.

4. Set Ethact as the Ethernet port ETH2 (J7) that is connected to the TFTP server and save the settings.

```
=> setenv ethact pfe_eth1
=> saveenv
```

NOTE

For more information on U-Boot environment variables and default settings refer: **Layerscape LS1012A Freeway Board Getting Started Guide**

4.6.3.3 RCW (Reset Configuration Word) and Ethernet interfaces

The following RCW binary is used on the LS1012A-FRWY board:

`N_SSNP_3305/rcw_1000_default.bin.swapped`

This RCW enables:

- Boot from QSPI
- 1000MHz Core, 250MHz Platfrom, 1000MT/s DDR
- I2C1,
- SerDes Protocol 0x3305
- SGMII, SGMII
- USB 3.0

4.6.3.4 System Memory Map

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

Start Physical Address	End Physical Address	Memory Type	Size
`0x00_0000_0000`	`0x00_00FF_FFFF`	Secure Boot ROM	1MB
`0x00_0100_0000`	`0x00_0FFF_FFFF`	CCSR	240MB
`0x00_1000_0000`	`0x00_1000_FFFF`	OCRAM1	64KB
`0x00_1001_0000`	`0x00_1001_FFFF`	OCRAM2	64 KB
`0x00_4000_0000`	`0x00_5FFF_FFFF`	QSPI	512MB
`0x00_8000_0000`	`0x00_FFFF_FFFF`	DRAM	2GB
`0x08_8000_0000`	`0x0F_FFFF_FFFF`	DRAM2	30G
`0x40_0000_0000`	`0x47_FFFF_FFFF`	PCI Express1	32G

4.6.3.5 Flash Bank Usage

The LS1012A-FRWY only has one QSPI flash connected over QSPI contoller.

```
U-Boot 2017.07-gcfcf893 (Mar 22 2018 - 20:05:29 +0800)
SoC: LS1012AE Rev2.0 (0x87040020)
Clock Configuration:
CPU0(A53):1000 MHz
Bus: 250 MHz DDR: 1000 MT/s
Reset Configuration Word (RCW):
00000000: 0800000a 00000000 00000000 00000000
00000010: 33050000 c000000c 40000000 00001800
00000020: 00000000 00000000 00000000 000c47f2
00000030: 00000000 1082a120 00000096 00000000
I2C: ready
DRAM: 446 MiB
```

```
PPA Firmware: Version ls1012a_board_ckout_rc1
WARNING: Calling __hwconfig without a buffer and before environment is ready
Using SERDES1 Protocol: 13061 (0x3305)
MMC: FSL_SDHC: 0, FSL_SDHC: 1
SF: Detected w25q16dw with page size 256 Bytes, erase size 4 KiB, total 2 MiB
In: serial
Out: serial
Err: serial
Model: LS1012A FRWY Board
Board: LS1012AFRWY Net: PFE class pe firmware
PFE tmu pe firmware
ls1012a_configure_serdes 0
ls1012a_configure_serdes 1
PCIe0: pcie@3400000 Root Complex: no link
pfe_eth0
Warning: pfe_eth0 (eth0) using random MAC address - ea:78:16:1d:f1:96
, pfe_eth1
Warning: pfe_eth1 (eth1) using random MAC address - 82:73:78:2e:b6:69
Hit any key to stop autoboot: 0
```

QSPI flash Layout

Start address	End address	Image	Maximum size
0x0000_0000	0x0000_FFFF	RCW+ PBI	64 KB
0x0001_0000	0x0001_FFFF	Hole	64 KB
0x0002_0000	0x0005_FFFF	PFE	256 KB
0x0006_0000	0x000D_FFFF	PPA	512 KB
0x000E_0000	0x000F_FFFF	Hole	128 KB
0x0010_0000	0x001C_FFFF	U-Boot	832 KB
0x001D_0000	0x001D_FFFF	Env Space	64 KB
0x001E_0000	0x001E_FFFF	Hole	64 KB
0x001F_0000	0x001F_FFFF	Secure header	64 KB

4.6.3.6 Check 'Link Up' for Serial Ethernet Interfaces

This section provides some basic checks that can be performed in U-Boot to help diagnose the cause of the networking errors when experiencing problems with Ethernet interfaces.

Check Communication to External PHY

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

Example:

```
=> mdio list
PFE_MDIO:
1 - RealTek RTL8211F <--> pfe_eth1
2 - RealTek RTL8211F <--> pfe_eth0
```

The results from the `mdio list` command above show that U-Boot was able to see PHYs on each of the SGMII interfaces.

Check Link Status for External PHY

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

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

Example:

```
=> mdio read pfe_eth0 1
Reading from bus PFE_MDIO
PHY at address 2:
1 - 0x79ad
=> mdio read pfe_eth1 1
Reading from bus PFE_MDIO
PHY at address 1:
1 - 0x79ad
```

The link partner (“copper side”) link status bit is in Register #1 on the PHY. The 'Link Status' bit is bit #2 (from the left) of the last nibble. In the above example the nibble of interest is "d" (d = b'1101'), and therefore the 'Link Status' = 1, which means 'link up'. If the link were down this bit would be a "0," and we would see 0x7989.

4.6.3.7 Basic Networking Ping Test

U-BOOT

The FRWY-LS1012A has 2 SGMII interfaces. The log below shows how to ping from those 2 interfaces.

```
U-Boot 2018.03-g7d5b814d89 (Jun 19 2018 - 02:24:33 +0800)

SoC:  LS1012AE Rev2.0 (0x87040020)
Clock Configuration:
       CPU0(A53):1000 MHz
       Bus:      250  MHz  DDR:      1000 MT/s
Reset Configuration Word (RCW):
       00000000: 0800000a 00000000 00000000 00000000
       00000010: 33050000 c000000c 40000000 00001800
       00000020: 00000000 00000000 00000000 000c47f2
       00000030: 00000000 1082a120 00000096 00000000
I2C:   ready
DRAM:  446 MiB
PPA Firmware: Version LSDK-18.06-TC3
SEC Firmware: 'loadables' present in config
loadables: 'trustedOS@1'
WARNING: Calling __hwconfig without a buffer and before environment is ready
Using SERDES1 Protocol: 13061 (0x3305)
MMC:   FSL_SDHC: 0, FSL_SDHC: 1
Loading Environment from SPI Flash... SF: Detected w25q16dw with page size 256 Bytes, erase
size 4 KiB, total 2 MiB
OK
In:    serial
Out:   serial
Err:   serial
Model: FRWY-LS1012A Board
Board: FRWY-LS1012A Version: RevA SATA link 0 timeout.
AHCI 0001.0301 32 slots 1 ports 6 Gbps 0x1 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc apst
Found 0 device(s).
SCSI:  Net:   PFE class pe firmware
PFE tmu pe firmware
pfe_configure_serdes 0

Warning: pfe_eth0 (eth0) using random MAC address - c6:cf:08:58:d3:3c
eth0: pfe_eth0pfe_configure_serdes 1

Warning: pfe_eth1 (eth1) using random MAC address - e2:ff:bf:06:ef:4c
, eth1: pfe_eth1
Hit any key to stop autoboot:  0
```

LINUX

To enable PFE in Linux, stop pfe in u-boot, however, first bring

the kernel-ls1012a-frwy.itb via pfe interface, after that perform pfe stop

On U-Boot prompt:

```
=> tftp 96000000 b35734/lsdk1806/ls1012afrwy/flex_linux_arm64.itb
Speed detected 3e8
Using pfe_eth0 device
TFTP from server 10.193.21.125; our IP address is 10.193.21.152
Filename 'b35734/lsdk1806/ls1012afrwy/flex_linux_arm64.itb'.
Load address: 0x96000000
Loading: #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #################################################################
         #
         9.9 MiB/s
done
Bytes transferred = 20051816 (131f768 hex)
=> pfe stop
Stopping PFE...
=> bootm 96000000#ls1012afrwy
## Loading kernel from FIT Image at 96000000 ...
   Using 'ls1012afrwy' configuration
   Trying 'kernel' kernel subimage
     Description:  ARM64 Kernel
     Type:         Kernel Image
     Compression:  gzip compressed
     Data Start:   0x960000d0
     Data Size:    9003624 Bytes = 8.6 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: 0x80080000
     Entry Point:  0x80080000
     Hash algo:    crc32
     Hash value:   7c1e3306
## Loading ramdisk from FIT Image at 96000000 ...
   Using 'ls1012afrwy' configuration
   Trying 'initrd' ramdisk subimage
     Description:  initrd for arm64
     Type:         RAMDisk Image
     Compression:  uncompressed
     Data Start:   0x96896418
     Data Size:    10906934 Bytes = 10.4 MiB
     Architecture: AArch64
     OS:           Linux
     Load Address: 0x00000000
     Entry Point:  0x00000000
     Hash algo:    crc32
     Hash value:   f7a71279
```

```
## Loading fdt from FIT Image at 96000000 ...
   Using 'ls1012afrwy' configuration
   Trying 'ls1012afrwy-dtb' fdt subimage
     Description:  ls1012afrwy-dtb
     Type:         Flat Device Tree
     Compression:  uncompressed
     Data Start:   0x9731bf60
     Data Size:    12853 Bytes = 12.6 KiB
     Architecture: AArch64
     Load Address: 0x90000000
     Hash algo:    crc32
     Hash value:   1c734ebb
   Loading fdt from 0x9731bf60 to 0x90000000
   Booting using the fdt blob at 0x90000000
   Uncompressing Kernel Image ... OK
   Using Device Tree in place at 0000000090000000, end 0000000090006234
WARNING: fdt_fixup_stdout: could not read serial0 alias: FDT_ERR_NOTFOUND
PCIe0: pcie@3400000 Root Complex: no link

Starting kernel ...

[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.9.105-00019-ga4b7c9d771f7 (jenkins@saturn) (gcc version 7.3.0
(Ubuntu/Linaro 7.3.0-16ubuntu3) ) #1 SMP PREEMPT Tue Jun 19 03:13:44 CS8
[    0.000000] Boot CPU: AArch64 Processor [410fd034]
[    0.000000] earlycon: uart8250 at MMIO 0x00000000021c0500 (options '')
[    0.000000] bootconsole [uart8250] enabled
[    1.405095] dmi: Firmware registration failed.
[    1.411240] No BMan portals available!
[    1.415700] No QMan portals available!
[    1.677033] Freescale FM module, FMD API version 21.1.0
[    1.683951] Freescale FM Ports module
[    2.619800] EXT4-fs (ram0): couldn't mount as ext3 due to feature incompatibilities
INIT: version 2.88 booting
Starting udev
[    6.291663] pe_load_ddr_section: load address(3fb0000) and elf file
address(ffff000009a0b000) rcvd
Populating dev cache
hwclock: can't open '/dev/misc/rtc': No such file or directory
Wed Jul 12 02:59:57 UTC 2017
hwclock: can't open '/dev/misc/rtc': No such file or directory
Running postinst /etc/rpm-postinsts/100-sysvinit-inittab...
update-rc.d: /etc/init.d/run-postinsts exists during rc.d purge (continuing)
Removing any system startup links for run-postinsts ...
INIT: Entering runlevel: 5
Configuring network interfaces... done.
Starting OpenBSD Secure Shell server: sshd
  generating ssh RSA key...
  generating ssh ECDSA key...
  generating ssh DSA key...
  generating ssh ED25519 key...
done.
hwclock: can't open '/dev/misc/rtc': No such file or directory
Starting syslogd/klogd: done

QorIQ LSDK (NXP Reference Distro) 2.0 TinyDistro /dev/ttyS0

TinyDistro login: root

root@TinyDistro:~# lsmod
    Tainted: G
pfe 106496 0 - Live 0xffff000000cd0000 (C)
root@TinyDistro:~# ifconfig -a
eth0      Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          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)
```

```
eth1      Link encap:Ethernet  HWaddr 00:00:00:00:00:00
          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:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

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

root@TinyDistro:~# ifconfig eth0 hw ether 00:11:22:33:44:55
root@TinyDistro:~# ifconfig eth1 hw ether 00:11:22:33:44:66
root@TinyDistro:~# udhcpc -i eth0
udhcpc (v1.23.2) started
Sending discover...
Sending select for 10.193.21.42...
Lease of 10.193.21.42 obtained, lease time 1800
/etc/udhcpc.d/50default: Adding DNS 10.192.130.201
/etc/udhcpc.d/50default: Adding DNS 10.228.49.200
/etc/udhcpc.d/50default: Adding DNS 10.201.141.100
root@TinyDistro:~# ping 10.193.21.125 -c 4
PING 10.193.21.125 (10.193.21.125): 56 data bytes
64 bytes from 10.193.21.125: seq=0 ttl=64 time=0.527 ms
64 bytes from 10.193.21.125: seq=1 ttl=64 time=0.270 ms
64 bytes from 10.193.21.125: seq=2 ttl=64 time=0.225 ms
64 bytes from 10.193.21.125: seq=3 ttl=64 time=0.237 ms

--- 10.193.21.125 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.225/0.314/0.527 ms
root@TinyDistro:~# ifconfig eth0 down
root@TinyDistro:~# udhcpc -i eth1
udhcpc (v1.23.2) started
Sending discover...
Sending select for 10.193.21.81...
Lease of 10.193.21.81 obtained, lease time 1800
/etc/udhcpc.d/50default: Adding DNS 10.192.130.201
/etc/udhcpc.d/50default: Adding DNS 10.228.49.200
/etc/udhcpc.d/50default: Adding DNS 10.201.141.100
root@TinyDistro:~# ping 10.193.21.125 -c 4
PING 10.193.21.125 (10.193.21.125): 56 data bytes
64 bytes from 10.193.21.125: seq=0 ttl=64 time=0.453 ms
64 bytes from 10.193.21.125: seq=1 ttl=64 time=0.236 ms
64 bytes from 10.193.21.125: seq=2 ttl=64 time=0.297 ms
64 bytes from 10.193.21.125: seq=3 ttl=64 time=0.243 ms

--- 10.193.21.125 ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.236/0.307/0.453 ms
```

4.6.4 LS1043ARDB

This section provides LS1043ARDB-specific information on switch setting configurations, U-Boot environment variable settings as well as supported binaries. It also provides a description of the virtual banks and flash and memory map layouts.

For more information on the LS1043ARDB refer to the QorIQ LS1043A Reference Design Board Reference Manual and the QorIQ LS1043A Reference Design Board Getting Started Guide.

4.6.4.1 On-board Switch Settings

Default Settings

The RDB has user selectable switches for evaluating different boot options and different configurations for the LS1043A device. The table below lists the default switch settings.

Table 17. Default Switch Settings

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
SW3	ON [1]	OFF [0]	ON [1]	ON [1]	OFF [0]	OFF [0]	ON [1]	ON [1]
SW4	OFF	OFF	OFF	ON	OFF	OFF	ON	OFF
SW5	ON	OFF	ON	OFF	OFF	OFF	ON	OFF

4.6.4.2 Supported boot options

LS1043ARDB supports the following boot options:

- NAND
- NOR
- SD

4.6.4.3 U-Boot Environment Variables

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

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

  ```
  =>setenv hwconfig
  ```

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

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

The default setting for for "hwconfig" on LS1043ARDB is as follows:

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

4.6.4.4 System Memory Map

Start Physical Address	End Physical Address	Memory Type	Size
0x00_0000_0000	0x00_000F_FFFF	Secure Boot ROM	1MB
0x00_0100_0000	0x00_0FFF_FFFF	CCSRBAR	240MB
0x00_1000_0000	0x00_1000_FFFF	OCRAM0	64KB
0x00_1001_0000	0x00_1001_FFFF	OCRAM1	64KB
0x00_2000_0000	0x00_20FF_FFFF	DCSR	16MB
0x00_6000_0000	0x00_67FF_FFFF	IFC - NOR Flash	128MB
0x00_7E80_0000	0x00_7E80_FFFF	IFC - NAND Flash	64KB
0x00_7FB0_0000	0x00_7FB0_0FFF	IFC - FPGA	4KB
0x00_8000_0000	0x00_FFFF_FFFF	DRAM1	2GB

4.6.4.5 NOR Flash (Virtual) Banks

NOR flash is a simple and convenient destination for deploying images so it is frequently used. Many NXP development and reference boards provide a special feature that allows a single NOR flash to be divided into multiple parts called “banks”. This is done by board-level logic that modifies address signals. Because there is only one NOR flash physically, the banks are sometimes called "virtual" banks.

The benefit of this feature is that it allows more than one set of images to be independently deployed to the one NOR flash. This is very helpful during development, because you can use the U-Boot image in one bank to program an image set into a different bank. If the new images are flawed, the old images are still functional to let you deploy corrected images.

The logic on the board usually allows the NOR flash to be divided into up to 8 banks, but almost always the feature is used to divide the flash into two halves only. For the LS1043ARDB, the 2 halves are called bank0 and bank4.

Switch Settings and NOR Banks

The NOR flash on the board can be seen as two flash banks. The board DIP switch configuration preselects bank 0 as the hardware default bank.

Switching to the alternate bank can be done in software and effectively swaps the first bank with the second bank, thereby putting the alternate bank in the bank 0 address range until further configuration or until a reset occurs. This protects banks 0 and keeps the board bootable under all circumstances.

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

```
U-Boot 2016.012.0+g2ea81ad

Clock Configuration:
       CPU0(A53):1600 MHz  CPU1(A53):1600 MHz  CPU2(A53):1600 MHz
       CPU3(A53):1600 MHz
       Bus:      400  MHz  DDR:      1600 MT/s  FMAN:     500  MHz
Reset Configuration Word (RCW):
       00000000: 08100010 0a000000 00000000 00000000
       00000010: 14550002 80004012 e0025000 c1002000
       00000020: 00000000 00000000 00000000 00038800
       00000030: 00000000 00001101 00000096 00000001
I2C:   ready
Model: LS1043A RDB Board
Board: LS1043ARDB, boot from vBank 4
CPLD:  V1.4
PCBA:  V3.0
SERDES Reference Clocks:
```

```
SD1_CLK1 = 156.25MHZ, SD1_CLK2 = 100.00MHZ
DRAM:  Initializing DDR....
Detected UDIMM Fixed DDR on board
2 GiB (DDR4, 32-bit, CL=11, ECC off)
SEC: RNG instantiated
Firmware 'Microcode version 0.0.1 for LS1021a r1.0' for 1021 V1.0
QE: uploading microcode 'Microcode for LS1021a r1.0' version 0.0.1
Waking secondary cores to start from ffd0e000
All (4) cores are up.
Using SERDES1 Protocol: 5205 (0x1455)
Flash: 128 MiB
NAND:  512 MiB
MMC:   FSL_SDHC: 0
EEPROM: NXID v1
PCIe1: disabled
PCIe2: Root Complex no link, regs @ 0x3500000
PCIe3: Root Complex x1 gen1, regs @ 0x3600000
PCI:
     01:00.0    - 8086:10d3 - Network controller
PCIe3: Bus 00 - 01
In:    serial
Out:   serial
Err:   serial
Net:   Fman1: Uploading microcode version 106.4.18
e1000: 00:1b:21:46:61:df
       FM1@DTSEC1, FM1@DTSEC2, FM1@DTSEC3, FM1@DTSEC4, FM1@DTSEC5, FM1@DTSEC6, FM1@TGEC1, 
e1000#0 [PRIME]
Warning: e1000#0 MAC addresses don't match:
Address in SROM is         00:1b:21:46:61:df
Address in environment is  00:e0:0c:00:85:07

=>
```

Bank Switching

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

- Switch to bank 0:

  ```
  =>cpld reset
  ```

- Switch to bank4:

  ```
  =>cpld reset altbank
  ```

NOR Flash Memory Map

NOR flash memory map for LS1043A RDB can be found on the link below.

Related information

LSDK Memory Layout on page 74

4.6.4.6 Supported Reset Configuration Word (RCW) Binaries

The following RCW binaries are used on the ls1043ardb

`RR_FQPP_1455/rcw_1500.bin`

`RR_FQPP_1455/rcw_1500_qetdm.bin`

`RR_FQPP_1455/rcw_1500_sben.bin`

`RR_FQPP_1455/rcw_1600.bin`

`RR_FQPP_1455/rcw_1600_qetdm.bin`

`RR_FQPP_1455/rcw_1600_sben.bin`

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

`ab_cdef_g`

Table 18. RCW directories Naming Convention Legend

Slot	Convention
a	'R' indicates RGMII1@DTSEC3 is supported 'N' if not available/not used
b	'R' indicates RGMII2@DTSEC4 is supported 'N' if not available/not used
c	What is available in lane A
d	What is available in lane B
e	What is available in lane C
f	What is available in lane D
g	Hex value of serdes1 protocol value

Table 19. For Lanes (C through F)

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

For example,

```
RR_FQPP_1455
```

means:

- RGMII1@DTSEC3 on board
- RGMII2@DTSEC4 on board
- XFI
- QSGMII
- PCIe2 on Mini-PCIe slot
- PCIe3 on PCIe Slot
- SERDES1 Protocol is 0x1455

The RCW file names for the ls1043ardb conform to the following naming convention:

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

Table 20. RCW Files Naming Convention Legend

Code		Convention
frequency		Core frequency(MHZ)
specialsetting	bootmode	SD/NAND/NOR and so on
	special support	nand: Nand boot sben: Secure boot lpuart: lpuart1 qspiboot: qspi boot

For example,

```
rcw_1500_sd.rcw means rcw for core frequency of 1500MHz with sd boot.
ls1043ardb/RR_FQPP_1455/rcw_1500.rcw means rcw for core frequency 0f 1500MHz with NOR boot.
```

4.6.4.7 Frame Manager Microcode (FMan ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for LS1043A version information and also for the version of FMan microcode currently flashed on the LS1043A (e.g., microcode version 106.4.18). For Layerscape SDK, the following FMan microcode binaries should be used:

```
fsl_fman_ucode_ls1043_r1.1_106_4_18.bin (*)
fsl_fman_ucode_ls1043_r1.1_108_4_9.bin
```

NOTE

(i) (*) Denotes the default FMan Microcode.

(ii) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(iii) Using a microcode binary from an older SDK with a Linux kernel from the current SDK is not supported.

4.6.4.8 Frame Manager configuration tool usage example

Introduction

The Frame Manager Configuration Tool (FMC) is a command line tool used to configure Frame Manager to perform the desired parse-classify-distribute (PCD) function for a given application. Parse-classify-distribute (PCD) is the Frame Manager's (FMan) ability to parse a received frame's protocol headers, perform a classification based on protocol header fields, and distribute or enqueue the received frame to a specific frame queue identifier (FQID) based on the results of the classification.

With the FMC tool, FMan PCD can be configured to forward ingress packets per flow to a dedicated frame queue. All packets in a dedicated frame queue are delivered to a single core, thus ensuring flow-to-core affinity. This guarantees preservation of ingress packet ordering per flow.

In absence of using the FMC tool to configure FMan for PCD, FMan will enqueue all received frames to a default frame queue. All cores are eligible to receive frames from this frame queue. In this case, the preservation of ingress packet ordering cannot be guaranteed.

The FMC tool uses following two input files:

- Configuration file

- Policy file

Configuration and policy files created with the objective of preserving packet ordering per flow are provided in NXP's Linux LSDK. The following section describes the steps to run FMC tool using these files.

Running the FMC tool

After the board boots to Linux, use the following commands to run the FMC tool to configure FMan to enqueue received frames to a dedicated frame queue:

1. Change directory to the parent directory of config.xml file
    - Example – For LS1043ARDB as the target board:
    - $ cd /etc/fmc/config/private/ls1043ardb/RR_FQPP_1455
2. Run the FMC tool command:
    - $ **fmc –c config.xml –p policy_ipv4.xml –a**

The FMC tool will only configure the interfaces that are enabled on the target board and will ignore the rest of the interfaces listed in the config.xml file. All policy files in the different target board directories point to the respective common policy file under /etc/fmc/config/private/common directory.

Note: After running the above FMC tool command, the Frame Manager is configured to perform the desired PCD function and there is no further action required from the user. The following sections are only informational and aim at providing insight into configuration and policy files.

Walk-through configuration file config.xml

Following is an example of the contents of the config.xml file for LS1043ARDB target board:

```
<cfgdata>
    <config>
        <engine name="fm0">
            <port type="MAC" number="1" policy="fm_policy_1"/>
            <port type="MAC" number="2" policy="fm_policy_2"/>
            <port type="MAC" number="3" policy="fm_policy_3"/>
            <port type="MAC" number="4" policy="fm_policy_4"/>
            <port type="MAC" number="5" policy="fm_policy_5"/>
            <port type="MAC" number="6" policy="fm_policy_6"/>
            <port type="MAC" number="9" policy="fm_policy_7"/>
        </engine>
    </config>
</cfgdata>
```

The value of the element `engine` - "fm0" corresponds to the hardware block FMan1. Its child element "port" has attributes – "type", "number" and "policy". The port type in this example is "MAC". The attribute "number" specifies the number of the port of a "type". See *Frame Manager Configuration Tool User Guide* in LSDK documentation to understand element and attribute requirements and their allowed values in a configuration file.

Frame Manager performs PCD on received frames and computes a FQID for each received frame. When Frame Manager enqueues a frame to Queue Manager it provides the frame descriptor (FD) and FQID. The policy "fm_policy_*n*" describes to Frame Manager how it should compute the FQID for frames received on the MAC port #*n*. The policies listed in the above configuration file are defined in the policy file policy_ipv4.xml.

Walk through policy file policy_ipv4.xml

The policy file policy_ipv4.xml defines 16 policies "fm_policy_1" through "fm_policy_16" that are intended to be used to configure specific ports.

The policy and distribution elements applicable to "fm_policy_3" are shown in the table below. Note that it is not required to use all the policies that have been defined in a policy file.

Element	Code segment

policy:
fm_policy_3
	`<policy name="fm_policy_3">`
	`    <dist_order>`
	`        <distributionref name="udpeth3"/>`
	`        <distributionref name="tcpeth3"/>`
	`        <distributionref name="ipv4eth3"/>`
	`        <distributionref`
	`name="garbage_dist_3"/>`
	`    </dist_order>`
	`</policy>`

• distribution `udpeth3`
• applied FIRST
	`<distribution name="udpeth3">`
	`    <queue count="128" base="0x3900"/>`
	`    <key>`
	`        <fieldref name="ipv4.src"/>`
	`        <fieldref name="ipv4.dst"/>`
	`        <fieldref name="ipv4.nextp"/>`
	`        <fieldref name="udp.sport"/>`
	`        <fieldref name="udp.dport"/>`
	`    </key>`
	`</distribution>`

• distribution `tcpeth3`
• applied SECOND
	`<distribution name="tcpeth3">`
	`    <queue count="128" base="0x3900"/>`
	`    <key>`
	`        <fieldref name="ipv4.src"/>`
	`        <fieldref name="ipv4.dst"/>`
	`        <fieldref name="ipv4.nextp"/>`
	`        <fieldref name="tcp.sport"/>`
	`        <fieldref name="tcp.dport"/>`
	`    </key>`
	`</distribution>`

• distribution `ipv4eth3`
• applied THIRD
	`<distribution name="ipv4eth3">`
	`    <queue count="128" base="0x3900"/>`
	`    <key>`
	`        <fieldref name="ipv4.src"/>`
	`        <fieldref name="ipv4.dst"/>`
	`        <fieldref name="ipv4.tos"/>`
	`    </key>`
	`</distribution>`

• distribution `garbage_dist_3`
• applied FOURTH
	`<distribution name="garbage_dist_3">`
	`    <queue count="1" base="0x3900"/>`
	`</distribution>`

A policy element lists the distributions that Frame Manager will use for ports configured to use the policy. In this example, four distributions are listed: “udpeth3”, “tcpeth3”, “ipv4eth3” and “garbage_dist_3”. Frame Manager will try the distributions in the order listed in the policy element.

Distribution `udpeth3`

The distribution “udpeth3” given in the table above has five keys:

Keys	Protocol	Field
`ipv4.src`	`IPv4`	Source IP Address
`ipv4.dst`	`IPv4`	Destination IP Address

Table continues on the next page...

Table continued from the previous page...

Keys	Protocol	Field
`ipv4.nextp`	`IPv4`	Protocol
`udp.sport`	`UDP`	Source Port
`udp.dport`	`UDP`	Destination Port

The presence of a IPv4 header and a UPD header are required for Frame Manager to extract these 5 keys from a frame. So, if a Rx frame has an IPv4 header and a UDP header then Frame Manager will use the “udpeth3” distribution to compute the frame’s FQID. In addition to the child element key the “udpeth3” distribution also has a child element queue. The element queue attribute count specifies the number of FQIDs and attribute base is the base FQID. For “udpeth3”, the number of FQIDs is 128 and the base FQID is 0x3900.

Distribution `tcpeth3`

If an Rx frame does not have both an IPv4 header and a UDP header, then the next distribution “tcpeth3” listed in the policy “fm_policy_3” will be tried.

The distribution “udpeth3” given in the table above has five keys:

Keys	Protocol	Field
`ipv4.src`	`IPv4`	Source IP Address
`ipv4.dst`	`IPv4`	Destination IP Address
`ipv4.nextp`	`IPv4`	Protocol
`udp.sport`	`TCP`	Source Port
`udp.dport`	`TCP`	Destination Port

If a Rx frame has an IPv4 header and a TCP header then Frame Manager will use distribution “tcpeth3” to compute the frame’s FQID. For distribution “tcpeth3” the queue count is 128 and the base FQID is 0x3900 (same base as used for distribution “udpeth3”).

.

Distribution `IPv4eth3`

If an Rx frame does not have both an IPv4 header and a TCP header, then the next distribution “ipv4eth3” listed in the policy “fm_policy_3” will be tried.

The distribution “ipv4eth3” has three keys:

Keys	Protocol	Field
`ipv4.src`	`IPv4`	Source IP Address
`ipv4.dst`	`IPv4`	Destination IP Address
`ipv4.toc`	`IPv4`	Protocol

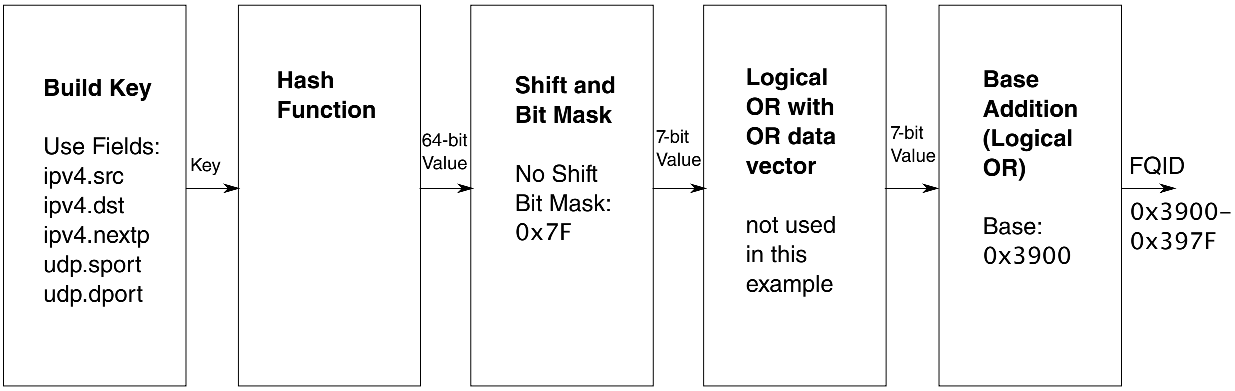
If a Rx frame has an IPv4 header then Frame Manager will use distribution “tcpeth3” to compute the frame’s FQID. For distribution “tcpeth3” the queue count is 128 and the base FQID is 0x3900 (same base as used for “udpeth3” and “tcpeth3”).

Distribution `garbage_dist_3`

If an Rx frame does not have an IPv4 header, then the next distribution listed in the policy “fm_policy_3” will be tried. The next distribution to try is the distribution “garbage_dist_3” listed in the table above. Distribution “garbage_dist_3” does not have a child element key. In this case, there are no requirements for specific protocol headers to be present. For distribution “garbage_dist_3” the queue count is 1 (a single frame queue) and the base FQID is 0x3900. When Frame Manager uses distribution “garbage_dist_3” for a Rx frame, Fame Manager will enqueue the frame to FQID 0x3900. Note: If distribution “garbage_dist_3” is listed first instead of last in the element dist_order, then all frames received on FMan1 MAC #3 interface will be enqueued to FQID 0x3900.

5. Walk through Frame Manager FQID Generation

Given below is an example of Frame Manager FQID generation for “udpeth3” distribution based on the FMan’s KeyGen algorithm for FQID calculation. (See Frame Manager Configuration Tool User Guide> Policy File>Distribution Section in LSDK documentation.)



Frame Manager KeyGen will concatenate the frame’s “ipv4.src”, “ipv4.dst”, “ipv4.nextp”, “udp.sport”, and “udp.dport” fields and will use the concatenated fields as input to its hash function. The result will be a 64-bit hash value. The 64-bit hash value will be ANDed with 0x7F (yielding 128 possible FQIDs) and then ORed with 0x3900. All Rx frames that have IPv4 and UDP protocol headers will be enqueued to a FQID in the range 0x3900 – 0x397f. Rx frames belonging to the same flow (i.e., having the same “ipv4.src”, “ipv4.dst”, “ipv4.nextp”, “udp.sport”, and “udp.dport” fields) will be enqueued to the same FQID because their 64-bit hash value will be identical. Note that it is possible for multiple flows to be enqueued to the same frame queue.

Further reading and references

For more information on FMC tool, see Frame Manager Configuration Tool User Guide in LSDK documentation. If you want to examine config.xml and policy_ipv4.xml on your host, these files are included in the LSDK package “eth-config”.

4.6.5 LS1046ARDB

This section provides LS1046ARDB-specific information on switch setting configurations, U-Boot environment variable settings as well as supported binaries. It also provides a description of the virtual banks and flash and memory map layouts.

For more information on the LS1046ARDB refer to the QorIQ LS1046A Reference Design Board Reference Manual and the QorIQ LS1046A Reference Design Board Getting Started Guide.

4.6.5.1 On-board Switch Settings

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

Table 21. Default Switch Settings

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

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

Table 22. LS1046ARDB Switch Settings

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

4.6.5.2 Supported Boot Options

LS1046ARDB supports the following boot options:

- SD
- QSPI

4.6.5.3 U-Boot Environment Variables

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

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

```
=>setenv hwconfig
```

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

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

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

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

4.6.5.4 System Memory Map

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

Start Physical Address	End Physical Address	Memory Type	Size
0x00_0000_0000	0x00_000F_FFFF	Secure Boot ROM	1MB

Table continues on the next page...

Table continued from the previous page...

Start Physical Address	End Physical Address	Memory Type	Size
0x00_0100_0000	0x00_0FFF_FFFF	CCSRBAR	240MB
0x00_1000_0000	0x00_1000_FFFF	OCRAM0	64KB
0x00_1001_0000	0x00_1001_FFFF	OCRAM1	64KB
0x00_2000_0000	0x00_20FF_FFFF	DCSR	16MB
0x00_7E80_0000	0x00_7E80_FFFF	IFC - NAND Flash	64KB
0x00_7FB0_0000	0x00_7FB0_0FFF	IFC - CPLD	4KB
0x00_8000_0000	0x00_FFFF_FFFF	DRAM1	2GB
0x05_0000_0000	0x05_07FF_FFFF	QMAN S/W Portal	128M
0x05_0800_0000	0x05_0FFF_FFFF	BMAN S/W Portal	128M
0x08_8000_0000	0x09_FFFF_FFFF	DRAM2	6GB
0x40_0000_0000	0x47_FFFF_FFFF	PCI Express1	32G
0x48_0000_0000	0x4F_FFFF_FFFF	PCI Express2	32G
0x50_0000_0000	0x57_FFFF_FFFF	PCI Express3	32G

4.6.5.5 QSPI Flash (Virtual) Banks

LS1046ARDB has 2 QSPI flash connected over QSPI contoller. Only one QSPI flash is available at a time depending upon the board switch settings. These switch settings can also be overriden by CPLD commands.

Switch Setting and QSPI Flash

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

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

```
U-Boot 2017.11-g00cde47 (Mar 22 2018 - 00:32:36 +0800)

SoC:  LS1046AE Rev1.0 (0x87070010)
Clock Configuration:
       CPU0(A72):1800 MHz  CPU1(A72):1800 MHz  CPU2(A72):1800 MHz
       CPU3(A72):1800 MHz
       Bus:      700  MHz  DDR:      2100 MT/s  FMAN:     800  MHz
Reset Configuration Word (RCW):
       00000000: 0e150012 10000000 00000000 00000000
       00000010: 11335559 40005012 40025000 c1000000
       00000020: 00000000 00000000 00000000 00238800
       00000030: 20124000 00003101 00000096 00000001
Model: LS1046A RDB Board
Board: LS1046ARDB, boot from QSPI vBank 4
CPLD:  V2.2
PCBA:  V2.0
SERDES Reference Clocks:
SD1_CLK1 = 156.25MHZ, SD1_CLK2 = 100.00MHZ
I2C:   ready
DRAM:  Initializing DDR....using SPD
Detected UDIMM 18ASF1G72AZ-2G3B1
7.9 GiB (DDR4, 64-bit, CL=15, ECC on)
```

```
       DDR Chip-Select Interleaving Mode: CS0+CS1
SEC0: RNG instantiated
PPA Firmware: Version LSDK-18.03
SEC Firmware: 'loadables' present in config
loadables: 'trustedOS@1'
Using SERDES1 Protocol: 4403 (0x1133)
Using SERDES2 Protocol: 21849 (0x5559)
NAND:  512 MiB
MMC:   FSL_SDHC: 0
SF: Detected s25fs512s with page size 512 Bytes, erase size 256 KiB, total 64 MiB
*** Warning - bad CRC, using default environment

EEPROM: NXID v1
In:    serial
Out:   serial
Err:   serial
Target spinup took 0 ms.
AHCI 0001.0301 32 slots 1 ports 6 Gbps 0x1 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc apst
Found 1 device(s).
SCSI:  Net:   SF: Detected s25fs512s with page size 512 Bytes, erase size 256 KiB, total 64 MiB
Fman1: Uploading microcode version 106.4.18
PCIe0: pcie@3400000 Root Complex: no link
PCIe1: pcie@3500000 Root Complex: no link
PCIe2: pcie@3600000 Root Complex: x1 gen1
e1000: 00:15:17:8e:7f:1c
       FM1@DTSEC3 [PRIME], FM1@DTSEC4, FM1@DTSEC5, FM1@DTSEC6, FM1@TGEC1, FM1@TGEC2, e1000#0
```

Bank Switching

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

- Switch to QSPI bank 0 (default):

  ```
  =>cpld reset
  ```

- Switch to QSPI bank 4:

  ```
  =>cpld reset altbank
  ```

QSPI Flash Memory Map

QSPI flash memory map for LS1046ARDB can be found on the link below.

Related information

LSDK Memory Layout on page 74

4.6.5.6 Supported Reset Configuration Word (RCW) Binaries

The following RCW binary is used on the LS1046ARDB:

```
RR_FFSSPPPH_1133_5559/rcw_1800_qspiboot.bin.swapped
```

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

```
ab_cdefghij_k_l
```

Table 23. RCW directories Naming Convention Legend

Slot	Convention
a	'R' indicates RGMII1@DTSEC3 is supported

Table continues on the next page...

Table 23. RCW directories Naming Convention Legend (continued)

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

Table 24. Serdes Protocol

Slot	Convention
k	hex value of serdes1 protocol value
l	hex value of serdes2 protocol value

Table 25. For Lanes (c through j)

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

For example,

```
RR_FFSSPPPH_1133_5559
```

means:

- RGMII1@DTSEC3 on board
- RGMII2@DTSEC4 on board
- XFI9@TGEC1 on board

- XFI10@TGEC2 on SFP cage
- SGMII5@DTSEC5 on board
- SGMII6@DTSEC6 on board
- PCIe1 on Slot 1, mPCIe
- PCIe2 on Slot 2
- PCIe3 on Slot 3
- SATA
- SERDES1 Protocol is 0x1133
- SERDES2 Protocol is 0x5559

The RCW file names for the LS1046ARDB conform to the following naming convention:

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

Table 26. RCW Files Naming Convention Legend

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

For example,

```
RR_FFSSPPPH_1133_5559/rcw_1800_qspiboot.bin.swapped means rcw for core frequency is 1800MHz
with QSPI boot.
```

Note: The default frequency for LS1046ARDB is 1800MHz.

4.6.5.7 Frame Manager Microcode (FMan ucode)

There are microcode binaries for the Frame Manager hardware block that is in QorIQ products. Specific platforms require specific binaries, and those also have to match specific software versions (i.e., match Frame Manager Driver version). See the U-Boot log for LS1046A version information and also for the version of FMan microcode currently flashed on the LS1046A (e.g., microcode version 106.4.18). For Layerscape SDK, one of the following FMan microcode binaries should be used:

```
fsl_fman_ucode_ls1046_r1.0_106_4_18.bin(*)
fsl_fman_ucode_ls1046_r1.0_108_4_9.bin
```

NOTE

(i) (*) Denotes the default FMan Microcode.

(i) Refer to the "readme" and release notes in the microcode git repository for a description of the various microcode releases.

(iii) Using a microcode binary from an older SDK with a Linux kernel from the current SDK is not supported.

4.6.5.8 Frame Manager configuration tool usage example

Introduction

The Frame Manager Configuration Tool (FMC) is a command line tool used to configure Frame Manager to perform the desired parse-classify-distribute (PCD) function for a given application. Parse-classify-distribute (PCD) is the Frame Manager's (FMan) ability to parse a received frame's protocol headers, perform a classification based on protocol header fields, and distribute or enqueue the received frame to a specific frame queue identifier (FQID) based on the results of the classification.

With the FMC tool, FMan PCD can be configured to forward ingress packets per flow to a dedicated frame queue. All packets in a dedicated frame queue are delivered to a single core, thus ensuring flow-to-core affinity. This guarantees preservation of ingress packet ordering per flow.

In absence of using the FMC tool to configure FMan for PCD, FMan will enqueue all received frames to a default frame queue. All cores are eligible to receive frames from this frame queue. In this case, the preservation of ingress packet ordering cannot be guaranteed.

The FMC tool uses following two input files:

- Configuration file
- Policy file

Configuration and policy files created with the objective of preserving packet ordering per flow are provided in NXP's Linux LSDK. The following section describes the steps to run FMC tool using these files.

Running the FMC tool

After the board boots to Linux, use the following commands to run the FMC tool to configure FMan to enqueue received frames to a dedicated frame queue:

1. Change directory to the parent directory of config.xml file
   - Example – For LS1046ARDB as the target board:
   - $ cd /etc/fmc/config/private/ls1046ardb/RR_FFSSPPPH_1133_5559
2. Run the FMC tool command:
   - $ **fmc –c config.xml –p policy_ipv4.xml –a**

The FMC tool will only configure the interfaces that are enabled on the target board and will ignore the rest of the interfaces listed in the config.xml file. All policy files in the different target board directories point to the respective common policy file under /etc/fmc/config/private/common directory.

Note: After running the above FMC tool command, the Frame Manager is configured to perform the desired PCD function and there is no further action required from the user. The following sections are only informational and aim at providing insight into configuration and policy files.

Walk-through configuration file config.xml

Following is an example of the contents of the config.xml file for LS1046ARDB target board:

```
<cfgdata>
    <config>
        <engine name="fm0">
            <port type="MAC" number="3" policy="fm_policy_3"/>
            <port type="MAC" number="4" policy="fm_policy_4"/>
               <port type="MAC" number="5" policy="fm_policy_5"/>
            <port type="MAC" number="6" policy="fm_policy_6"/>
            <port type="MAC" number="9" policy="fm_policy_7"/>
               <port type="MAC" number="10" policy="fm_policy_8"/>
        </engine>
    </config>
</cfgdata>
```

The value of the element `engine` - “fm0” corresponds to the hardware block FMan1. Its child element “port” has attributes – “type”, “number” and “policy”. The port type in this example is “MAC”. The attribute “number” specifies the number of the port of a “type”. See *Frame Manager Configuration Tool User Guide* in LSDK documentation to understand element and attribute requirements and their allowed values in a configuration file.

Frame Manager performs PCD on received frames and computes a FQID for each received frame. When Frame Manager enqueues a frame to Queue Manager it provides the frame descriptor (FD) and FQID. The policy “fm_policy_*n*” describes to Frame Manager how it should compute the FQID for frames received on the MAC port #*n*. The policies listed in the above configuration file are defined in the policy file policy_ipv4.xml.

Walk through policy file policy_ipv4.xml

The policy file policy_ipv4.xml defines 16 policies “fm_policy_1” through “fm_policy_16” that are intended to be used to configure specific ports.

The policy and distribution elements applicable to “fm_policy_3” are shown in the table below. Note that it is not required to use all the policies that have been defined in a policy file.

Element	Code segment

policy:
fm_policy_3
	`<policy name="fm_policy_3">`
	`    <dist_order>`
	`        <distributionref name="udpeth3"/>`
	`        <distributionref name="tcpeth3"/>`
	`        <distributionref name="ipv4eth3"/>`
	`        <distributionref`
	`name="garbage_dist_3"/>`
	`    </dist_order>`
	`</policy>`

• distribution `udpeth3`
• applied FIRST
	`<distribution name="udpeth3">`
	`    <queue count="128" base="0x3900"/>`
	`    <key>`
	`        <fieldref name="ipv4.src"/>`
	`        <fieldref name="ipv4.dst"/>`
	`        <fieldref name="ipv4.nextp"/>`
	`        <fieldref name="udp.sport"/>`
	`        <fieldref name="udp.dport"/>`
	`    </key>`
	`</distribution>`

• distribution `tcpeth3`
• applied SECOND
	`<distribution name="tcpeth3">`
	`    <queue count="128" base="0x3900"/>`
	`    <key>`
	`        <fieldref name="ipv4.src"/>`
	`        <fieldref name="ipv4.dst"/>`
	`        <fieldref name="ipv4.nextp"/>`
	`        <fieldref name="tcp.sport"/>`
	`        <fieldref name="tcp.dport"/>`
	`    </key>`
	`</distribution>`

• distribution `ipv4eth3`
• applied THIRD
	`<distribution name="ipv4eth3">`
	`    <queue count="128" base="0x3900"/>`
	`    <key>`
	`        <fieldref name="ipv4.src"/>`
	`        <fieldref name="ipv4.dst"/>`
	`        <fieldref name="ipv4.tos"/>`
	`    </key>`
	`</distribution>`

Table continues on the next page...

Table continued from the previous page...

Element	Code segment

- distribution `garbage_dist_3`
- applied FOURTH
	`<distribution name="garbage_dist_3">`
	`    <queue count="1" base="0x3900"/>`
	`</distribution>`

A policy element lists the distributions that Frame Manager will use for ports configured to use the policy. In this example, four distributions are listed: “udpeth3”, “tcpeth3”, “ipv4eth3” and “garbage_dist_3”. Frame Manager will try the distributions in the order listed in the policy element.

Distribution `udpeth3`

The distribution “udpeth3” given in the table above has five keys:

Keys	Protocol	Field
`ipv4.src`	`IPv4`	Source IP Address
`ipv4.dst`	`IPv4`	Destination IP Address
`ipv4.nextp`	`IPv4`	Protocol
`udp.sport`	`UDP`	Source Port
`udp.dport`	`UDP`	Destination Port

The presence of a IPv4 header and a UPD header are required for Frame Manager to extract these 5 keys from a frame. So, if a Rx frame has an IPv4 header and a UDP header then Frame Manager will use the “udpeth3” distribution to compute the frame’s FQID. In addition to the child element key the “udpeth3” distribution also has a child element queue. The element queue attribute count specifies the number of FQIDs and attribute base is the base FQID. For “udpeth3”, the number of FQIDs is 128 and the base FQID is 0x3900.

Distribution `tcpeth3`

If an Rx frame does not have both an IPv4 header and a UDP header, then the next distribution “tcpeth3” listed in the policy “fm_policy_3” will be tried.

The distribution “udpeth3” given in the table above has five keys:

Keys	Protocol	Field
`ipv4.src`	`IPv4`	Source IP Address
`ipv4.dst`	`IPv4`	Destination IP Address
`ipv4.nextp`	`IPv4`	Protocol
`udp.sport`	`TCP`	Source Port
`udp.dport`	`TCP`	Destination Port

If a Rx frame has an IPv4 header and a TCP header then Frame Manager will use distribution “tcpeth3” to compute the frame’s FQID. For distribution “tcpeth3” the queue count is 128 and the base FQID is 0x3900 (same base as used for distribution “udpeth3”).

Distribution `IPv4eth3`

If an Rx frame does not have both an IPv4 header and a TCP header, then the next distribution “ipv4eth3” listed in the policy “fm_policy_3” will be tried.

The distribution “ipv4eth3” has three keys:

Keys	Protocol	Field
`ipv4.src`	`IPv4`	Source IP Address
`ipv4.dst`	`IPv4`	Destination IP Address
`ipv4.toc`	`IPv4`	Protocol

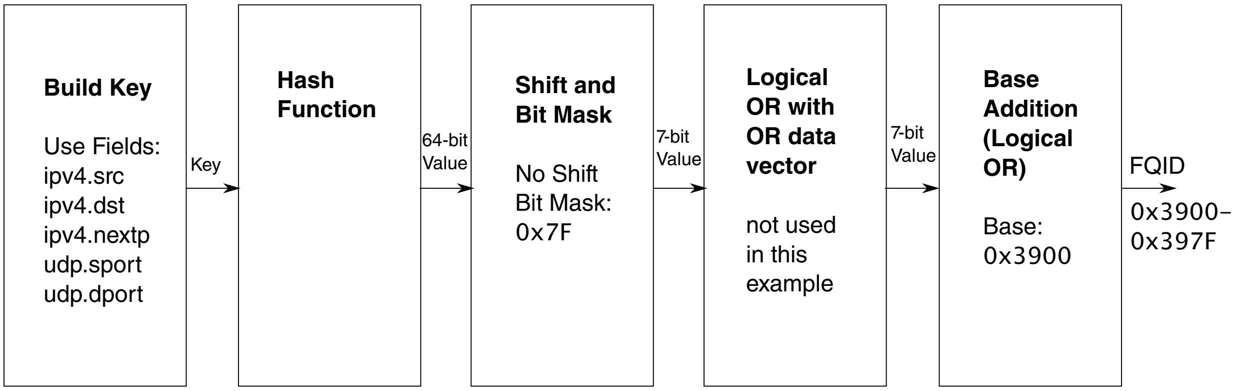
If a Rx frame has an IPv4 header then Frame Manager will use distribution “tcpeth3” to compute the frame’s FQID. For distribution “tcpeth3” the queue count is 128 and the base FQID is 0x3900 (same base as used for “udpeth3” and “tcpeth3”).

Distribution `garbage_dist_3`

If an Rx frame does not have an IPv4 header, then the next distribution listed in the policy “fm_policy_3” will be tried. The next distribution to try is the distribution “garbage_dist_3” listed in the table above. Distribution “garbage_dist_3” does not have a child element key. In this case, there are no requirements for specific protocol headers to be present. For distribution “garbage_dist_3” the queue count is 1 (a single frame queue) and the base FQID is 0x3900. When Frame Manager uses distribution “garbage_dist_3” for a Rx frame, Fame Manager will enqueue the frame to FQID 0x3900. Note: If distribution “garbage_dist_3” is listed first instead of last in the element dist_order, then all frames received on FMan1 MAC #3 interface will be enqueued to FQID 0x3900.

5. Walk through Frame Manager FQID Generation

Given below is an example of Frame Manager FQID generation for “udpeth3” distribution based on the FMan’s KeyGen algorithm for FQID calculation. (See Frame Manager Configuration Tool User Guide > Policy File > Distribution Section in LSDK documentation.)



Frame Manager KeyGen will concatenate the frame’s “ipv4.src”, “ipv4.dst”, “ipv4.nextp”, “udp.sport”, and “udp.dport” fields and will use the concatenated fields as input to its hash function. The result will be a 64-bit hash value. The 64-bit hash value will be ANDed with 0x7F (yielding 128 possible FQIDs) and then ORed with 0x3900. All Rx frames that have IPv4 and UDP protocol headers will be enqueued to a FQID in the range 0x3900 – 0x397f. Rx frames belonging to the same flow (i.e., having the same “ipv4.src”, “ipv4.dst”, “ipv4.nextp”, “udp.sport”, and “udp.dport” fields) will be enqueued to the same FQID because their 64-bit hash value will be identical. Note that it is possible for multiple flows to be enqueued to the same frame queue.

Further reading and references

For more information on FMC tool, see Frame Manager Configuration Tool User Guide in LSDK documentation. If you want to examine config.xml and policy_ipv4.xml on your host, these files are included in the LSDK package “eth-config”.

4.6.6 LS1088ARDB

This section provides LS1088ARDB-specific information on switch setting configurations, U-Boot environment variable settings as well as supported binaries. It also provides a description of the virtual banks and flash and memory map layouts.

For more information on the LS1088ARDB refer to the QorIQ LS1088A Reference Design Board Reference Manual and the QorIQ LS1088A Reference Design Board Getting Started Guide.

4.6.6.1 On-board Switch Settings

Default Settings

The table below lists the default switch settings for the LS1088ARDB.

Table 27. Default Switch Settings

Switch	Settings (1: ON; 0: OFF)
SW1	0011 0001 (for QSPI Boot) 0010 0000 (for SD Boot)
SW2	0100 0000
SW3	1111 0010
SW4	1001 0011
SW5	1111 0000

To switch from QSPI to SD you may also use the following command:

```
=> i2c mw 66 0x60 0x20; i2c mw 66 10 10;i2c mw 66 10 21
```

4.6.6.2 Supported Boot Types

LS1088ARDB supports the following boot types for this release:

- SD
- QSPI

4.6.6.3 Booting

Booting U-Boot

By default (as per board switch settings), the boot loader (U-Boot) image located in QSPI flash0 or SD card (Link to top where switch settings are listed) runs when the power is turned on. Press any key while U-Boot is counting down to stop U-Boot from automatically running the bootcmd variable and booting Linux.

As the U-Boot boots to its prompt, users can use the commands listed below to deploy new images onto the RDB.

For more details refer to On-board Switch Settings on page 117.

Booting Linux

Booting Linux is controlled by the contents of U-Boot enviroment variable `bootcmd`.

U-Boot passes the contents of U-Boot environment variable `bootargs` to Linux as boot-time kernel parameters. The commands in this variable are automatically run and Linux boots after U-Boot counts down a number of seconds given by U-Boot environment variable `bootdelay`. Variable bootdelay can be set to -1 to avoid automatically booting Linux.

One can view the values of U-Boot enviroment variables using the U-Boot "printenv" command. The critical variables for booting Linux and their default values are shown below.

1. bootargs: contains parameters that are passed to the Linux kernel before it starts.

   ```
   => printenv bootargs
   bootargs=console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500,
   ramdisk_size=0x2000000 default_hugepagesz=2m hugepagesz=2m hugepages=256
   ```

   Notes on these kernel parameters:

   - Select ttyS0 as the console serial port.
   - Select file system to be a RAM disk populated from an image in the kernel itb file.

2. Environment variable `mcinitcmd` contains commands used during U-Boot for Management Complex load. It is not mandatory but very useful. Refer to U-Boot Environment Variables on page 119 for more information.

   For SD/eMMC boot:

   ```
   pri mcinitcmd
   mcinitcmd=mmcinfo;mmc read 0x80000000 0x5000 0x800;mmc read 0x80100000 0x7000 0x800;env
   exists secureboot && mmc read 0x80700000 0x3800 0x10 && mmc read 0x80740000 0x3A00 0x10
   && esbc_validate 0x80700000 && esbc_validate 0x80740000 ;fsl_mc start mc 0x80000000
   0x80100000
   ```

   For QSPI boot:

   ```
   pri mcinitcmd
   mcinitcmd=sf probe 0:0;sf read 0x80000000 0xA00000 0x100000;sf read 0x80100000 0xE00000
   0x100000;env exists secureboot && sf read 0x80700000 0x700000 0x40000 && sf read
   0x80740000 0x740000 0x40000 && esbc_validate 0x80700000 && esbc_validate
   0x80740000 ;fsl_mc start mc 0x80000000 0x80100000
   ```

3. Environment variable `bootcmd` contains commands used to boot Linux. The image needs to be copied to DDR first.

   For SD/eMMC boot:

   ```
   => pri bootcmd
   env exists mcinitcmd && mmcinfo; mmc read 0x80001000 0x6800 0x800; env exists mcinitcmd
   && env exists secureboot  && mmc read 0x80780000 0x3C00 0x10 && esbc_validate
   0x80780000;env exists mcinitcmd && fsl_mc lazyapply dpl 0x80001000;run
   distro_bootcmd;run sd_bootcmd;env exists secureboot && esbc_halt;
   => pri distro_bootcmd
   distro_bootcmd=setenv scsi_need_init; for target in ${boot_targets}; do run bootcmd_$
   {target}; done
   => pri sd_bootcmd
   sd_bootcmd=echo Trying load from sd card..;mmcinfo; mmc read $load_addr $kernel_addr_sd
   $kernel_size_sd ;env exists secureboot && mmc read $kernelheader_addr_r
   $kernelhdr_addr_sd $kernelhdr_size_sd  && esbc_validate ${kernelheader_addr_r};bootm
   $load_addr#$BOARD
   ```

   For QSPI boot:

   ```
   => pri bootcmd
   bootcmd=sf read 0x80001000 0xd00000 0x100000;env exists mcinitcmd && env exists
   secureboot  && sf read 0x80780000 0x780000 0x100000 && esbc_validate 0x80780000;env
   exists mcinitcmd && fsl_mc lazyapply dpl 0x80001000;run distro_bootcmd;run
   qspi_bootcmd;env exists secureboot && esbc_halt;
   => pri distro_bootcmd
   distro_bootcmd=setenv scsi_need_init; for target in ${boot_targets}; do run bootcmd_$
   {target}; done
   => pri qspi_bootcmd
   qspi_bootcmd=echo Trying load from qspi..;sf probe && sf read $load_addr $kernel_addr
   ```

```
    $kernel_size ; env exists secureboot && sf read $kernelheader_addr_r $kernelheader_addr
    $kernelheader_size && esbc_validate ${kernelheader_addr_r}; bootm $load_addr#$BOARD
```

U-Boot enviroment variables such as `kernel_load` contain addresses used in the boot process. You can inspect them using the U-Boot `printenv` command. The `bootcmd` command may change depending upon the `mcinitcmd env` variable.

4.6.6.4 U-Boot Environment Variables

DPAA2-specific Environment Vairables

- ***mcboottimeout***: Defines Management Complex boot timeout in milliseconds. If this variable is not defined, the complile-time value `CONFIG_SYS_LS_MC_BOOT_TIMEOUT_MS` will be the default. Normally, users do not need to set this variable because the default is acceptable.
- ***mcmemsize***: Defines amount of system DDR to be used by the Management Complex. If this variable is not defined, the compile-time value `0x70000000` or 1.75GB will be the default. Normally, users do not need to set this variable because the default is acceptable.
- ***mcinitcmd***: Contains commands to load and start the Management Complex automatically before the U-Boot count down to boot starts. If this variable is defined, its contents are run. The default value assumes that the Management Complex (MC) firmware and Data Path Control file are stored in QSPI/SD flash at fixed addresses.

Environment variables that are not specific to DPAA2

- ***bootcmd***: Contains commands that are automatically executed when the U-Boot `boot` command is run. This happens automatically when the user does not interrupt U-Boot's initial count down.

```
=> pri
BOARD=ls1088ardb
arch=arm
baudrate=115200
board=ls1088a
board_name=ls1088a
boot_a_script=load ${devtype} ${devnum}:${distro_bootpart} ${scriptaddr} ${prefix}${script};
env exists secureboot && load ${devtype} ${devnum}:${distro_bootpart} ${scripthdraddr} $
{prefix}${boot_script_hdr} && esbc_validate ${scripthdraddr};source ${scriptaddr}
boot_efi_binary=load ${devtype} ${devnum}:${distro_bootpart} ${kernel_addr_r} efi/boot/
bootaa64.efi; if fdt addr ${fdt_addr_r}; then bootefi ${kernel_addr_r} ${fdt_addr_r};else
bootefi ${kernel_addr_r} ${fdtcontroladdr};fi
boot_extlinux=sysboot ${devtype} ${devnum}:${distro_bootpart} any ${scriptaddr} $
{prefix}extlinux/extlinux.conf
boot_net_pci_enum=pci enum
boot_net_usb_start=usb start
boot_prefixes=/ /boot/
boot_script_dhcp=boot.scr.uimg
boot_script_hdr=hdr_ls1088ardb_bs.out
boot_scripts=ls1088ardb_boot.scr
boot_targets=usb0 mmc0 scsi0 dhcp
bootargs=console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500
ramdisk_size=0x3000000 default_hugepagesz=2m hugepagesz=2m hugepages=256
bootcmd=env exists mcinitcmd && run mcinitcmd ;&& env exists mcinitcmd && mmcinfo; mmc read
0x88000000 0x6800 0x800; && fsl_mc apply dpl 0x88000000;run distro_bootcmd;run sd_bootcmd
bootcmd_dhcp=run boot_net_usb_start; run boot_net_pci_enum; if dhcp ${scriptaddr} $
{boot_script_dhcp}; then source ${scriptaddr}; fi;setenv efi_fdtfile ${fdtfile}; setenv
efi_old_vci ${bootp_vci};setenv efi_old_arch ${bootp_arch};setenv bootp_vci PXEClient:Arch:
00011:UNDI:003000;setenv bootp_arch 0xb;if dhcp ${kernel_addr_r}; then tftpboot $
{fdt_addr_r} dtb/${efi_fdtfile};if fdt addr ${fdt_addr_r}; then bootefi ${kernel_addr_r} $
{fdt_addr_r}; else bootefi ${kernel_addr_r} ${fdtcontroladdr};fi;fi;setenv bootp_vci $
{efi_old_vci};setenv bootp_arch ${efi_old_arch};setenv efi_fdtfile;setenv
efi_old_arch;setenv efi_old_vci;
bootcmd_mmc0=setenv devnum 0; run mmc_boot
bootcmd_scsi0=setenv devnum 0; run scsi_boot
bootcmd_usb0=setenv devnum 0; run usb_boot
bootdelay=2
cpu=armv8
```

```
distro_bootcmd=setenv scsi_need_init; for target in ${boot_targets}; do run bootcmd_$
{target}; done
efi_dtb_prefixes=/ /dtb/ /dtb/current/
eth10addr=68:05:ca:26:57:43
ethact=DPMAC1@xgmii
ethprime=DPMAC1@xgmii
fdt_addr=0x64f00000
fdt_addr_r=0x90000000
fdt_high=0xa0000000
fdtcontroladdr=ffd18160
fdtheader_addr_r=0x80100000
hwconfig=fsl_ddr:bank_intlv=auto
initrd_high=0xffffffffffffffff
installer=load mmc 0:2 $load_addr /flex_linux_arm64.itb; env exists mcinitcmd && run
mcinitcmd && mmc read 0x80200000 0x6800 0x800; fsl_mc apply dpl 0x80200000;bootm
$load_addr#ls1088ardb
kernel_addr=0x1000000
kernel_addr_r=0x81000000
kernel_addr_sd=0x8000
kernel_size=0x2800000
kernel_size_sd=0x14000
kernel_start=0x580100000
kernelheader_addr=0x800000
kernelheader_addr_r=0x80200000
kernelheader_size=0x40000
kernelheader_start=0x580800000
load_addr=0xa0000000
load_efi_dtb=load ${devtype} ${devnum}:${distro_bootpart} ${fdt_addr_r} ${prefix}$
{efi_fdtfile}
mcinitcmd=mmcinfo;mmc read 0x80000000 0x5000 0x800;mmc read 0x80100000 0x7000 0x800;env
exists secureboot && mmc read 0x80700000 0x3800 0x800 && mmc read 0x80740000 0x3A00 0x800 &&
esbc_validate 0x80700000 && esbc_validate 0x80740000 ;fsl_mc start mc 0x80000000 0x80100000
mcmemsize=0x70000000
mmc_boot=if mmc dev ${devnum}; then setenv devtype mmc; run scan_dev_for_boot_part; fi
nor_bootcmd=echo Trying load from nor..;cp.b $kernel_start $load_addr $kernel_size ; env
exists secureboot && cp.b $kernelheader_start $kernelheader_addr_r $kernelheader_size &&
esbc_validate ${kernelheader_addr_r}; bootm $load_addr#$BOARD
qspi_bootcmd=echo Trying load from qspi..;sf probe && sf read $load_addr $kernel_addr
$kernel_size ; env exists secureboot && sf read $kernelheader_addr_r $kernelheader_addr
$kernelheader_size && esbc_validate ${kernelheader_addr_r};  bootm $load_addr#$BOARD
ramdisk_addr=0x800000
ramdisk_size=0x2000000
scan_dev_for_boot=echo Scanning ${devtype} ${devnum}:${distro_bootpart}...; for prefix in $
{boot_prefixes}; do run scan_dev_for_scripts; done;
scan_dev_for_boot_part=part list ${devtype} ${devnum} devplist; env exists devplist ||
setenv devplist 1; for distro_bootpart in ${devplist}; do if fstype ${devtype} ${devnum}:$
{distro_bootpart} bootfstype; then run scan_dev_for_boot; fi; done
scan_dev_for_efi=setenv efi_fdtfile ${fdtfile}; for prefix in ${efi_dtb_prefixes}; do if
test -e ${devtype} ${devnum}:${distro_bootpart} ${prefix}${efi_fdtfile}; then run
load_efi_dtb; fi;done;if test -e ${devtype} ${devnum}:${distro_bootpart} efi/boot/
bootaa64.efi; then echo Found EFI removable media binary efi/boot/bootaa64.efi; run
boot_efi_binary; echo EFI LOAD FAILED: continuing...; fi; setenv efi_fdtfile
scan_dev_for_extlinux=if test -e ${devtype} ${devnum}:${distro_bootpart} ${prefix}extlinux/
extlinux.conf; then echo Found ${prefix}extlinux/extlinux.conf; run boot_extlinux; echo
SCRIPT FAILED: continuing...; fi
scan_dev_for_scripts=for script in ${boot_scripts}; do if test -e ${devtype} ${devnum}:$
{distro_bootpart} ${prefix}${script}; then echo Found U-Boot script ${prefix}${script}; run
boot_a_script; echo SCRIPT FAILED: continuing...; fi; done
scriptaddr=0x80000000
scripthdraddr=0x80080000
scsi_boot=run scsi_init; if scsi dev ${devnum}; then setenv devtype scsi; run
scan_dev_for_boot_part; fi
scsi_init=if ${scsi_need_init}; then setenv scsi_need_init false; scsi scan; fi
scsidevs=1
sd_bootcmd=echo Trying load from sd card..;mmcinfo; mmc read $load_addr $kernel_addr_sd
$kernel_size_sd ; bootm $load_addr#$BOARD
soc=fsl-layerscape
stderr=serial
stdin=serial
```

```
stdout=serial
usb_boot=usb start; if usb dev ${devnum}; then setenv devtype usb; run
scan_dev_for_boot_part; fi
vendor=freescale

Environment size: 5821/8188 bytes
=>
```

4.6.6.5 Supported Reset Configuration Word (RCW) Binaries

LS1088A supports following RCW binaries:

Table 28. RCW binary

SoC	Binary Name	Descripton	Required Switch setting

LS1088A	`rcw_x.bin`
	`rcw_x_bootmode.bin`
	`rcw_x_specialsetting.bin`
		Core: 1600 MHz, Platform: 700 MHz, DDR: 2100 MHz
		x = Core frequency
		bootmode = SD/NAND/NOR and so on.
		specialsetting = special setting: sben:Secure boot
		For example, rcw_1600.rcw means rcw for core frequency of 1600MHz with nor boot. rcw_1600_sd.rcw means rcw for core frequency of 1600MHz with sd boot.
		SerDes protocol used here is 0x1d, 0xd
			SYSCLK 100 MHz, SW3[6:8] = 010 SYSCLK Differential, SW 5[1] = 1

NOTE

The RCW's provided with the release enable the following features:

1. The figure below shows the SERDES configuration supported for LS1088A.

Table 29. SerDes1

Protocol	D	C	B	A
0x1D	XFI2	XFI1	QSGb	QSGa

Table 30. SerDes2

Protocol	A	B	C	D
0x0D	PCIe1 x1	PCIe2 x1	PCIe3 x1	SATA

2. Enables 4 UART without flow control
3. Enables I2C, SDHC, PCIe, SATA, USB

4.6.6.6 Ethernet MAC Connectivity

For DPAA2 SoCs, it is helpful to know which SoC Ethernet MACs are connected to which physical Ethernet ports on the board. The figure below shows this.

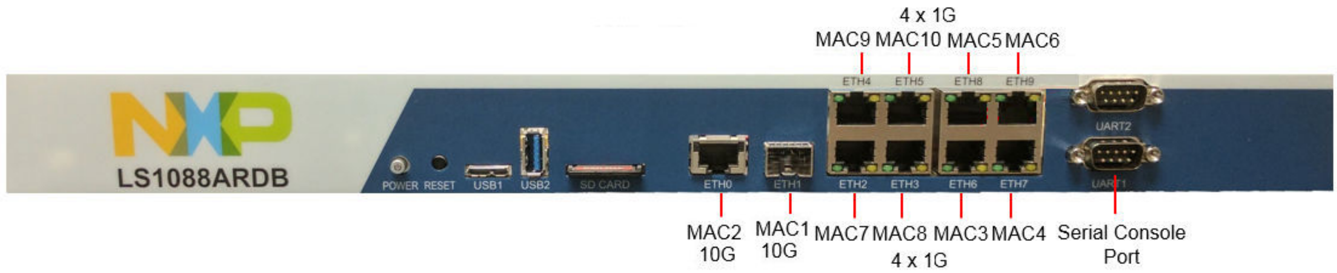


Figure 1. Ethernet MAC Connectivity

In U-Boot, Ethernet interfaces are named according to MACs. For example, the interface connected to MAC1 is called `DPMAC1@xgmii`.

On Linux, by default, only one MAC is enabled as a standard Kernel Ethernet Interface. This interface is named as ETH8 on Chassis front panel. By default, this interface is enabled on DPAA2, as the ethernet ports need not be associated with Linux kernel ethernet driver instances. These ports can be assigned as ports on switches or allocated to user space. These associations can be made dynamically. For example, if you want the Ethernet port to be connected to MAC1 to act as standard Linux Ethernet interface enter the following (as root):

```
ls-addni dpmac.1
```

A new Linux Ethernet interface will be created on the fly and hot-plugged into the kernel.

Interface `eth1` is created automatically because the default data path layout (DPL) creates it. DPL's are a mechanism to create DPAA2 entities prior to Linux boot. One can customize the DPL to get a different set of DPAA2 entities to be present automatically.

See the DPAA2-specific Software on page 673 for more information.

4.6.7 LS1088ARDB-PB

This section provides LS1088ARDB-PB specific information on switch setting configurations, U-Boot environment variable settings as well as supported binaries. It also provides a description of the virtual banks and flash and memory map layouts.

NOTE

- LS1088ARDB-PB is a variant of LS1088ARDB, hence other than Switch Settings, RCW, and U-Boot all the components are same as LS1088ARDB.
- Configruing LS1088ARDB-PB is as similar as LS1088ARDB

4.6.7.1 On-board Switch Settings

Default Settings

The table below lists the default switch settings for the LS1088ARDB-PB.

Table 31. Default Switch Settings

Switch	Settings (1: ON; 0: OFF)
SW1	0011 0001
SW2	X10X 0000
SW3	1111 0010
SW4	1001 0011
SW5	0111 0000

You can use either of the following are the two commands to switch from QSPI to SD:

```
=> i2c mw 66 0x60 0x20; i2c mw 66 10 10;i2c mw 66 10 21
=> qixis_reset sd
```

4.6.7.2 Supported Boot Types

LS1088ARDB-PB supports the following boot types for this release:

- SD
- QSPI

4.6.7.3 Booting

Booting U-Boot

By default (as per board switch settings), the boot loader (U-Boot) image located in QSPI flash0 or SD card (Link to top where switch settings are listed) runs when the power is turned on. Press any key while U-Boot is counting down to stop U-Boot from automatically running the bootcmd variable and booting Linux.

As the U-Boot boots to its prompt, users can use the commands listed below to deploy new images onto the RDB.

For more details refer to On-board Switch Settings on page 117.

Booting Linux

Booting Linux is controlled by the contents of U-Boot enviroment variable `bootcmd`.

U-Boot passes the contents of U-Boot environment variable `bootargs` to Linux as boot-time kernel parameters. The commands in this variable are automatically run and Linux boots after U-Boot counts down a number of seconds given by U-Boot environment variable `bootdelay`. Variable bootdelay can be set to -1 to avoid automatically booting Linux.

One can view the values of U-Boot enviroment variables using the U-Boot "printenv" command. The critical variables for booting Linux and their default values are shown below.

1. bootargs: contains parameters that are passed to the Linux kernel before it starts.

   ```
   => printenv bootargs
   bootargs=console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,mmio,0x21c0500,
   ramdisk_size=0x2000000 default_hugepagesz=2m hugepagesz=2m hugepages=256
   ```

   Notes on these kernel parameters:

   - Select ttyS0 as the console serial port.
   - Select file system to be a RAM disk populated from an image in the kernel itb file.

2. Environment variable `mcinitcmd` contains commands used during U-Boot for Management Complex load. It is not mandatory but very useful. Refer to U-Boot Environment Variables on page 119 for more information.

   For SD/eMMC boot:

   ```
   pri mcinitcmd
   mcinitcmd=mmcinfo;mmc read 0x80000000 0x5000 0x800;mmc read 0x80100000 0x7000 0x800;env
   exists secureboot && mmc read 0x80700000 0x3800 0x10 && mmc read 0x80740000 0x3A00 0x10
   && esbc_validate 0x80700000 && esbc_validate 0x80740000 ;fsl_mc start mc 0x80000000
   0x80100000
   ```

   For QSPI boot:

   ```
   pri mcinitcmd
   mcinitcmd=sf probe 0:0;sf read 0x80000000 0xA00000 0x100000;sf read 0x80100000 0xE00000
   0x100000;env exists secureboot && sf read 0x80700000 0x700000 0x40000 && sf read
   0x80740000 0x740000 0x40000 && esbc_validate 0x80700000 && esbc_validate
   0x80740000 ;fsl_mc start mc 0x80000000 0x80100000
   ```

3. Environment variable `bootcmd` contains commands used to boot Linux. The image needs to be copied to DDR first.

   For SD/eMMC boot:

   ```
   => pri bootcmd
   env exists mcinitcmd && mmcinfo; mmc read 0x80001000 0x6800 0x800; env exists mcinitcmd
   && env exists secureboot  && mmc read 0x80780000 0x3C00 0x10 && esbc_validate
   0x80780000;env exists mcinitcmd && fsl_mc lazyapply dpl 0x80001000;run
   distro_bootcmd;run sd_bootcmd;env exists secureboot && esbc_halt;
   => pri distro_bootcmd
   distro_bootcmd=setenv scsi_need_init; for target in ${boot_targets}; do run bootcmd_$
   {target}; done
   => pri sd_bootcmd
   sd_bootcmd=echo Trying load from sd card..;mmcinfo; mmc read $load_addr $kernel_addr_sd
   $kernel_size_sd ;env exists secureboot && mmc read $kernelheader_addr_r
   $kernelhdr_addr_sd $kernelhdr_size_sd  && esbc_validate ${kernelheader_addr_r};bootm
   $load_addr#ls1088ardb
   ```

   For QSPI boot:

   ```
   => pri bootcmd
   bootcmd=sf read 0x80001000 0xd00000 0x100000;env exists mcinitcmd && env exists
   secureboot  && sf read 0x80780000 0x780000 0x100000 && esbc_validate 0x80780000;env
   exists mcinitcmd && fsl_mc lazyapply dpl 0x80001000;run distro_bootcmd;run
   qspi_bootcmd;env exists secureboot && esbc_halt;
   => pri distro_bootcmd
   distro_bootcmd=setenv scsi_need_init; for target in ${boot_targets}; do run bootcmd_$
   {target}; done
   => pri qspi_bootcmd
   qspi_bootcmd=echo Trying load from qspi..;sf probe && sf read $load_addr $kernel_addr
   $kernel_size ; env exists secureboot && sf read $kernelheader_addr_r $kernelheader_addr
   $kernelheader_size && esbc_validate ${kernelheader_addr_r}; bootm $load_addr#ls1088ardb
   ```

U-Boot enviroment variables such as `kernel_load` contain addresses used in the boot process. You can inspect them using the U-Boot `printenv` command. The `bootcmd` command may change depending upon the `mcinitcmd env` variable.

4.6.7.4 U-Boot Environment Variables

For more details on U-Boot environment variables, see U-Boot Environment Variables on page 119.

4.6.7.5 Supported Reset Configuration Word (RCW) Binaries

LS1088ARDB-PB supports following RCW binaries:

Table 32. RCW binary

SoC	Binary Name	Descripton	Required Switch setting

LS1088ARDB-PB	`rcw_1600_qspi.bin`
	`rcw_1600_sd.bin`
		Core: 1600 MHz, Platform: 700 MHz, DDR: 2100 MHz
		x = Core frequency
		bootmode = SD/NAND/NOR and so on.
		specialsetting = special setting: sben:Secure boot
		For example, rcw_1600.rcw means rcw for core frequency of 1600MHz with nor boot. rcw_1600_sd.rcw means rcw for core frequency of 1600MHz with sd boot.
		SerDes protocol used here is 0x1d, 0x13
			SYSCLK 100 MHz, SW3[6:8] = 010 SYSCLK Differential, SW 5[1] = 1

NOTE

The RCW's provided with the release enable the following features:

1. The figure below shows the SERDES configuration supported for LS1088ARDB-PB.

Table 33. SerDes interface options

SerDes	Lane	Protocol	Interface	Connectors	Device and Protocol
1	3 (“A”)	QSGMII	F104 PHY	J75	LS1088A: srds1_prtcl = 0x1D
	2 (“B”)	QSGMII	F104 PHY	J77	LS1088A: srds1_prtcl = 0x1D
	1 (“C”)	XFI Optical	SFP Cage	J40	LS1088A: srds1_prtcl = 0x1D
	0 (“D”)	XFI Copper	Aquantia	J50	LS1088A: srds1_prtcl = 0x1D
2	0 (“A”)	PCIe (Gen3) x2	M.2 Slot 1 (Key E, 2230 card)	J45	LS1088A: srds2_prtcl = 0x13
	1 (“B”)				
	2 (“C”)	PCIe (Gen3) x2	M.2 Slot 2 (Key E, 2230 card)	J42	LS1088A: srds2_prtcl = 0x13
	3 (“D”)				

2. Enables 4 UART without flow control
3. Enables I2C, SDHC, PCIe, USB

4.6.7.6 Ethernet MAC Connectivity

For DPAA2 SoCs, it is helpful to know which SoC Ethernet MACs are connected to which physical Ethernet ports on the board. The figure below shows this.

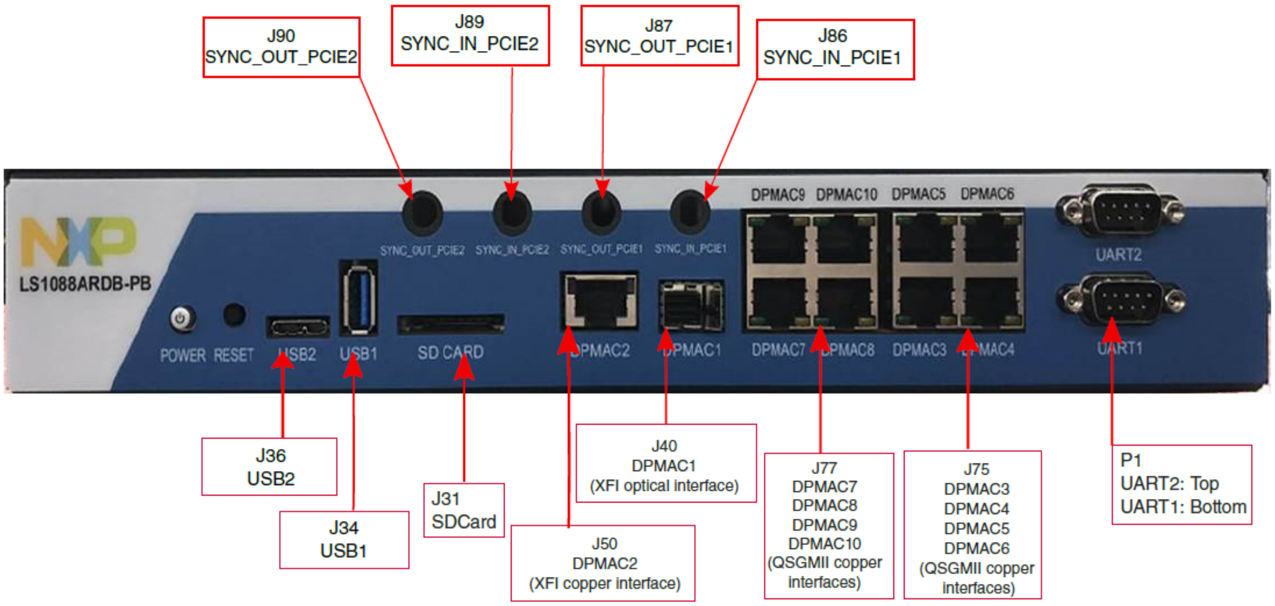


Figure 2. Ethernet MAC Connectivity

In U-Boot, Ethernet interfaces are named according to MACs. For example, the interface connected to MAC1 is called `DPMAC1@xgmii`.

On Linux, by default, only one MAC is enabled as a standard Kernel Ethernet Interface. This interface is named as DPMAC5 on Chassis front panel. By default, this interface is enabled on DPAA2, as the ethernet ports need not be associated with Linux kernel ethernet driver instances. These ports can be assigned as ports on switches or allocated to user space. These associations can be made dynamically. For example, if you want the Ethernet port to be connected to MAC1 to act as standard Linux Ethernet interface enter the following (as root):

```
ls-addni dpmac.1
```

A new Linux Ethernet interface will be created on the fly and hot-plugged into the kernel.

Interface `eth1` is created automatically because the default data path layout (DPL) creates it. DPL's are a mechanism to create DPAA2 entities prior to Linux boot. One can customize the DPL to get a different set of DPAA2 entities to be present automatically.

See the DPAA2-specific Software on page 673 for more information.

4.6.8 LS2088ARDB

This section provides LS2088ARDB-specific information on switch setting configurations, U-Boot environment variable settings as well as supported binaries. It also provides a description of the virtual banks and flash memory layouts.

For more information on the LS2088ARDB refer to the QorIQ LS2085/LS2088A Reference Design Board Reference Manual and the QorIQ LS2085/LS2088A Reference Design Board Getting Started Guide.

4.6.8.1 On-board Switch Settings

Default Settings

The figure below shows default configuration switch settings for rev D - F boards. Earlier boards are not supported.

The bits on the switches are numbered from 1 to 8. In the figure, bit 1 is on the left.

Table 34. Default Switch Settings for NOR Boot

Switch	Settings (1: ON; 0: OFF)
SW5	1111 1111
SW3	0001 0010
SW4	1111 1111
SW6	1111 1111
SW7	0100 0010 (100 MHz SYSCLOCK)
SW9	0100 0000
SW8	0111 1111

Table 35. Default Switch Settings for QSPI Boot

Switch	Settings (1: ON; 0: OFF)
SW5	1111 1111
SW3	0011 0001
SW4	0111 1111
SW6	1111 1111
SW7	0100 1010 (100 MHz SYSCLOCK)
SW9	01000100
SW8	0111 1111

Jumper Settings

Make sure the following jumper settings are correct based on the preferred type of boot.

Jumper	Settings
J8	For QSPI-boot, via on-board qspi flash: 1-2 For QSPI-boot, via qspi emulator: 2-3
J14	For NOR-boot: 1-2 For QSPI-boot: 2-3

SYSCLK Frequency

The SYSCLK frequency is controlled by switch settings on the RDB. For revisions D – E of the LS2088A RDB, SYSCLK is controlled by bits 1:3 of SW7. The available SYSCLK frequencies are:

Table 36. Available SYSCLK frequencies

Switch Setting	Frequency
off off off	66.666 MHz
off off on	83.333 MHz

Table continues on the next page...

Table 36. Available SYSCLK frequencies (continued)

Switch Setting	Frequency
off on off	100.000 MHz
off on on	125.000 MHz
on off off	133.333 MHz

You must set the switches to match what your PBL file expects. There is some complexity regarding when changes to switches take effect. The easy approach is to change the switches when the board is powered off. That will always work.

4.6.8.2 Supported Boot Options

LS2088ARDB supports the following boot options:

- QSPI
- NOR

4.6.8.3 Booting

Booting U-Boot

By default (per board switch settings), the boot loader (U-Boot) image located in NOR flash bank 0 runs on power on. Press any key while U-Boot is counting down to stop U-Boot from automatically running the "bootcmd" variable and booting Linux.

As the U-Boot boots to its prompt, users can use the commands listed below to deploy new images onto the RDB.

Booting Linux

Booting Linux is controlled by the contents of U-Boot enviroment variable "bootcmd".

U-Boot passes the contents of U-Boot environment variable "bootargs" to Linux as boot-time kernel parameters. The commands in this variable are automatically run and Linux boots after U-Boot counts down a number of seconds given by U-Boot environment variable "bootdelay". Variable bootdelay can be set to -1 to avoid automatically booting Linux.

One can view the values of U-Boot enviroment variables using the U-Boot "printenv" command. The critical variables for booting Linux and their default values are shown below.

1. bootargs: contains parameters that are passed to the Linux kernel before it starts.

   ```
   printenv bootargs
   bootrgs=console=ttyS1,115200 root=/dev/ram0 earlycon=uart8250,mmio,0x21c0600
   ramdisk_size=0x2000000 default_hugepagesz=2m hugepagesz=2m hugepages=256
   ```

2. Notes on these kernel parameters:
   - Select ttyS1 (UART2) as the console serial port (ttyS0 will not work on early LS2085A RDB/ LS2088A RDB boards, prior to Rev D).
   - Select file system to be a RAM disk populated from an image in the kernel itb file.
   - Allocate huge pages for user space (ODP) use. This is not needed unless you plan to use ODP. Please see ODP documentation for more information.
3. Environment variable "mcinitcmd" contains commands used during U-Boot for Management Complex load. It is not mandatory but very useful. Refer to U-Boot Environment Variables on page 129 for more information.

   For NOR boot

```
printenv mcinitcmd
mcinitcmd=fsl_mc start mc 0x580a00000 0x580e00000
```

For QPSI boot

```
printenv mcinitcmd
mcinitcmd==sf probe 0:0;sf read 0x20a00000 0xA00000 0x100000;sf read 0x20e00000
0xE00000 0x100000;fsl_mc start mc 0x20a00000 0x20e00000
```

4. Environment variable "bootcmd" contains commands used to boot Linux. The image needs to be copied to DDR first.

   e.g. For NOR boot

   ```
   printenv bootcmd
   bootcmd=run mcinitcmd && fsl_mc lazy apply dpl 0x580d00000 && cp.b $kernel_start
   $kernel_load $kernel_size && bootm $kernel_load || run distro_bootcmd
   ```

   U-Boot enviroment variables such as "kernel_load" contain addresses used in the boot process. You can inspect them using the U-Boot "printenv" command. "bootcmd" command may change depending upon mcinitcmd env variable.

4.6.8.4 U-Boot Environment Variables

DPAA2-specific Environment Variables

- ***mcboottimeout***: Defines Management Complex boot timeout in milliseconds. If this variable is not defined the complile-time value, `CONFIG_SYS_LS_MC_BOOT_TIMEOUT_MS` will be the default. Normally, users do not need to set this variable because the default is acceptable.
- ***mcmemsize***: Defines amount of system DDR to be use by the Management Complex. If this variable is not defined, the compile-time value `CONFIG_SYS_LS_MC_DRAM_BLOCK_MIN_SIZE` will be the default. Normally, users do not need to set this variable because the default is acceptable.
- ***mcinitcmd***: Contains commands to load and start the Management Complex automatically before the U-Boot count down to boot starts. If this variable is defined, its contents are `run`. The default value assumes that the Management Complex (MC) firmware and Data Path Control file are stored in NOR flash at fixed addresses. The default value for NOR boot is `fsl_mc start mc 0x580a00000 0x580e00000` and for QSPI boot is `fsl_mc start mc 0x20a00000 0x20e00000`. Users may change this variable as needed to load the MC files from sources other than NOR into DDR and then start the MC using the fsl_mc command. For example, the files may be on a disk drive.

Environment variables that are not specific to DPAA2

bootcmd: Contains commands that are automatically executed when the U-Boot "boot" command is run. This happens automatically when the user does not interrupt U-Boot's initial count down. In normal usage, bootcmd should contain the command to apply the Management Complex Data Path Layout (DPL) file because this must be done before booting Linux. The default value of bootcmd assumes that the DPL file is stored in NOR flash at a fixed address. The default is `mcinitcmd && fsl_mc lazyapply dpl 0x580d00000 && cp.b $kernel_start $kernel_load $kernel_size && bootm $kernel_load || run distro_bootcmd for NOR-Boot`

4.6.8.5 NOR Flash (Virtual) Banks

NOR flash is a simple and convenient destination for deploying images so it is frequently used. Many NXP development and reference boards provide a special feature that allows a single NOR flash to be divided into multiple parts called "banks". This is done by board-level logic that modifies address signals. Because there is only one NOR flash physically, the banks are sometimes called "virtual" banks.

The benefit of this feature is that it allows more than one set of images to be independently deployed to the one NOR flash. This is very helpful during development because you can use the U-Boot image in one bank to program an image set into a different bank. If the new images are flawed, the old images are still functional to let you deploy corrected images.

The logic on the board usually allows the NOR flash to be divided into up to 8 banks, but almost always the feature is used to divide the flash into two halves only. The halves are called bank 0 and bank 4.

This is relevant to SYSCLK and PBLs because PBL files are one of the image types that must be deployed. It is possible to have different PBL files in the two banks, and they may require different SYSCLK frequencies. One must operate the board correctly in order to avoid using the wrong SYSCLK with a PBL.

Switch Settings and NOR Banks

On the LS2085ARDB/LS2088ARDB revisions D – F, switch SW9 bits 3:5 control which bank the SoC loads from when it powers up. The relevant values are:

Table 37. Switch Settings and NOR Banks

Switch Settings	NOR Banks
off off off	Load from NOR Flash Bank 0
on off off	Load from NOR Flash Bank 4

U-Boot prints which bank is loaded from. The output looks like following.

- For bank0

```
U-Boot 2016.01LS2088A-SDK+ge54c320 (Jul 17 2016 - 22:56:33 +0800)

SoC:  LS2088E Version:1.0 (0x87090010)
Clock Configuration:
       CPU0(A72):1800 MHz  CPU1(A72):1800 MHz  CPU2(A72):1800 MHz
       CPU3(A72):1800 MHz  CPU4(A72):1800 MHz  CPU5(A72):1800 MHz
       CPU6(A72):1800 MHz  CPU7(A72):1800 MHz
       Bus:      700  MHz  DDR:      1866.667 MT/s     DP-DDR:   1600 MT/s
Reset Configuration Word (RCW):
       00000000: 48303838 48480048 00000000 00000000
       00000010: 00000000 00000000 00a00000 00000000
       00000020: 00801180 00002581 00000000 00000000
       00000030: 00000c0b 00000000 00000000 00000000
       00000040: 00000000 00000000 00000000 00000000
       00000050: 00000000 00000000 00000000 00000000
       00000060: 00000000 00000000 00027000 00000000
       00000070: 412a0000 00000000 00000000 00000000
I2C:   ready
Model: Freescale Layerscape 2088A RDB Board
Board: LS2085A/LS2088A-RDB, Board Arch: V1, Board version: D, boot from vBank: 0
```

- For bank4

```
U-Boot 2016.01LS2088A-SDK+ge54c320 (Jul 17 2016 - 22:56:33 +0800)

SoC:  LS2088E Version:1.0 (0x87090010)
Clock Configuration:
       CPU0(A72):1800 MHz  CPU1(A72):1800 MHz  CPU2(A72):1800 MHz
       CPU3(A72):1800 MHz  CPU4(A72):1800 MHz  CPU5(A72):1800 MHz
       CPU6(A72):1800 MHz  CPU7(A72):1800 MHz
       Bus:      700  MHz  DDR:      1866.667 MT/s     DP-DDR:   1600 MT/s
Reset Configuration Word (RCW):
       00000000: 48303838 48480048 00000000 00000000
       00000010: 00000000 00000000 00a00000 00000000
       00000020: 00801180 00002581 00000000 00000000
       00000030: 00000c0b 00000000 00000000 00000000
       00000040: 00000000 00000000 00000000 00000000
       00000050: 00000000 00000000 00000000 00000000
       00000060: 00000000 00000000 00027000 00000000
       00000070: 412a0000 00000000 00000000 00000000
I2C:   ready
Model: Freescale Layerscape 2088A RDB Board
Board: LS2085A/LS2088A-RDB, Board Arch: V1, Board version: D, boot from vBank: 4
```

Bank Switching

U-Boot on the LS2085ARDB/LS2088ARDB has a useful command called `qixis_reset`. The command `qixis_reset` does a hard reset, loading from the bank specified by the switches.

If the switches are set to load from bank 0, then the command `qixis_reset altbank` will cause a reset into bank 4 (without a need to change any switches). This is useful if banks 0 and 4 contain PBL files that require the same SYSCLK.

NOR Flash Banks and Addresses

The “current” bank is the bank that was loaded and is currently running. The “alternate” bank is the other bank. For example, if you are running from bank 0 then bank 4 is the alternate bank. If you are running from bank 4, bank 0 is the alternate bank. The alternate bank if often called the "other" bank because it is the bank from which you are NOT booted.

Addressing is based on current and alternate/other banks, not banks 0 and 4. This is best explained by an example. Suppose the address of something in the current bank is `0x5_8000_0000`. The corresponding address in the alternate bank is changed in one bit, `0x5_8400_0000`. For example, suppose the byte at `0x5_8000_0000` have value `0xaa` and, while running U-Boot in the current bank, and you read the byte at `0x5_8400_0000` and see `0xbb`. Now, reboot after changing SW9 bits 3:5 so you are running U-Boot from what was the alternate bank and is now the current bank. You will see `0xbb` at `0x5_8000_0000` and `0xaa` at `0x5_8400_0000`.

This scheme is very useful, because it means you can always deploy images to the alternate bank using the same addresses, regardless of which bank happens to be “current”. In other words, the steps to write an image into bank 4 using U-Boot running from bank 0 are the same as the steps to write an image to bank 0 using U-Boot running from bank 4.

NOR Flash Memory Map
NOR flash memory map for LS2088A can be found on the link below.
Related information
LSDK Memory Layout on page 74

4.6.8.6 Supported RCW (Reset Configuration Word) Binaries

The release contains the RCW binaries for the LS2088ARDB with SoC revisions r1.0 and r1.1.

- For LS2088ARDB containing a LS2088A r1.0 SoC, use the RCW directory `ls2088ardb`.
- For LS2088ARDB containing a LS2088A r1.1 SoC, use the RCW directory `ls2088ardb_rev1.1`.

These directories contains following folders:

- FFFFFFFF_PP_HH_0x2a_0x41
- FFFFFFFF_PP_NN_0x2a_0x3f

The RCW sub-directories names for the LS2088ARDB boards conform to the following naming convention:

```
abcdefgh_ij_kl_mn
```

Table 38. LS2088ARDB RCW Directory Naming Convention

Slot	Convention
a	What is available in SFP cage 1
b	What is available in SFP cage 2
c	What is available in SFP cage 3
d	What is available in SFP cage 4
e	What is available in SFP cage 5
f	What is available in SFP cage 6

Table continues on the next page...

Table 38. LS2088ARDB RCW Directory Naming Convention (continued)

g	What is available in SFP cage 7
h	What is available in SFP cage 8
i	What is available in Slot 1
j	What is available in Slot 2
k	SATA1
l	SATA2

NOTE

For the SFP cage and Slots (a..j) the following symbols are used:

- 'N' is NULL, not available/not used
- 'P' is PCIe
- 'S' is SGMII
- 'Q' is QSGMII
- 'F' is XFI
- 'H' is SATA

Serdes1 protocol (m):

m = 'hex value of serdes1 protocol value'

Serdes2 protocol (n):

n = 'hex value of serdes2 protocol value'

=============================

RCW binaries present in above sub-directories follow below naming convention

Files naming convention

=============================

```
rcw_x.rcw
rcw_x_y.rcw
rcw_x_bootmode.rcw

x = Core frequency
y = Platform frequency
bootmode = nor(default)/sd/qspi

For example,
  rcw_2000.rcw means rcw for core frequency of 2000MHz.
  rcw_2000_700.rcw means rcw for core frequency 2000MHz and Platform frequecny 700MHz.
  rcw_2000_qspi.rcw means rcw for core frequency of 2000MHz with QSPI boot.
```

Default recommended settings

================================

Default flash image in the Release is created with for rcw_2000.bin from ls2088ardb directory. One needs to update rcw as per the below recommendation

directory

For LS2088ARDB with LS2088A r1.0, it is recommended to use RCWs from directory ls2088ardb

For LS2088ARDB with LS2088A r1.1, it is recommended to use RCWs from directory ls2088ardb_rev1.1

NOTE

To be compatible with both LS2088A 1.0 and 1.1, RCW image rcw_1800.bin is used by **default** for SoC running at 1800MHz in this release, if users want to use a different RCW image for LS2088A 1.1 running at 2000MHz, follow the steps as below:

step 1: `$ cd flexbuild`

step 2: `$ flex-builder -c rcw`

step 3: download RCW image from flexbuild/build/firmware/rcw/ls2088ardb_rev1.1/FFFFFFFF_PP_HH_0x2a_0x41/rcw_2000.bin to ls2088ardb board under U-Boot prompte, replace rcw_1800.bin with rcw_2000.bin

```
=> tftp a0000000 rcw_2000.bin
```

To program current bank:

```
=> protect off 580000000 +$filesize && erase 580000000 +$filesize && cp.b a0000000
580000000 $filesize
```

To program alternate bank:

```
=> protect off 584000000 +$filesize && erase 584000000 +$filesize && cp.b a0000000
584000000 $filesize
```

sub-directories

----

default serdes protocol for LS2088ARDB: 0x2a_0x41, so it is recommended to use RCWs from FFFFFFFF_PP_HH_0x2a_0x41

For PCI gen3 use case, it is recommeded to use RCWs from FFFFFFFF_PP_NN_0x2a_0x3f

RCW binary

For LS2088ARDB-RevD , it is recommended to use bin2 or bin3 (details mentioed above) . Due to board errata, bin1 is instable on this board.

For LS2088ARDB-RevF (NOR-boot), either of bin1. bin2. bin3 can be used.

For LS2088ARDB-RevF (QSPI-boot), it is recommended to use rcw_2000_qspi.bin

4.6.8.7 Ethernet MAC Connectivity

For DPAA2 SoCs, it is helpful to know which SoC Ethernet MACs are connected to which physical Ethernet ports on the board. The figure below shows this.

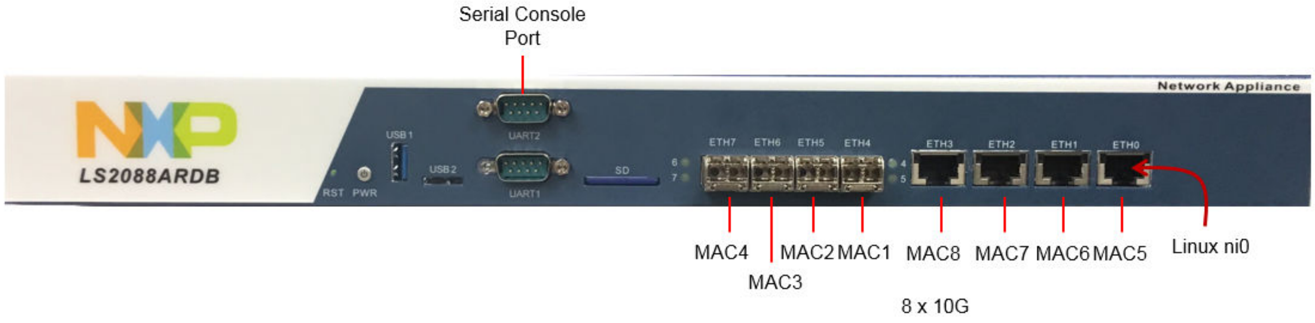


Figure 3. Ethernet MAC Connectivity

In U-Boot, Ethernet interfaces are named according to MACs. For example, the interface connected to MAC1 is called `DPMAC1@xgmii`.

In Linux, only one MAC is enabled by default as a standard kernel Ethernet Interface. This interface is named `ni0` by default. The Ethernet port it corresponds to is shown in the figure. The reason that only one interface is enabled by default is because, in DPAA2, Ethernet ports need not be associated with Linux kernel Ethernet driver instances. For example, they can be assigned as ports on switches or allocated to user space. Furthermore, these associations can be made dynamically. As an example, suppose that you want the Ethernet port connected to MAC1 to act as standard Linux Ethernet interface. To accomplish this, enter the following (as root):

```
ls-addni dpmac.1
```

A new Linux Ethernet interface will be created on the fly and hot-plugged into the kernel.

Interface `ni0` is created automatically because the default data path layout (DPL) creates it. DPL's are a mechanism to create DPAA2 entities prior to Linux boot. One can customize the DPL to get a different set of DPAA2 entities to be present automatically.

See the DPAA2-specific Software on page 673 for more information.

Chapter 5 Bootloaders

5.1 General boot flow

NXP SoC Booting Principles

The high-level (some details omitted) boot flow of an ARMv8a SoC is:

1. SoC comes out of reset and reads and RCW/PBL image from a boot source such as NOR flash or an SD card or eMMC flash. The RCW/PBL image contains configuration bits that control things such as:
   - Pin muxing and the protocol selected for SerDes pins.
   - Clock parameters and PLL multipliers.
   - Device containing the first software (not in an internal SoC ROM) to run.
2. Code in an internal boot ROM starts running and configures low-level aspects of the SoC.
3. The boot ROM must then load the first external software to run from a boot device such as NOR flash or SD/eMMC. In today's LSDK, this first software is a boot loader, either U-Boot or UEFI.
4. The boot ROM transfers control to the boot loader.
5. The boot loader does additional system configuration and then loads and starts the PPA image from NOR flash or SD/eMMC.
6. The PPA is a special resident firmware that runs at the highest ARMv8A privilege level EL3. It provides services to both the boot loader and, later, the operating system. These services include controlling core power state and bringing additional cores out of reset.
7. Usually, the boot loader must also load peripheral firmware, firmware required to make peripherals such as Ethernet controllers work. The details of this differ from SoC to SoC.
8. When the boot loader finishes initialization, its job is to locate a Linux kernel image and a Linux device tree image. The device tree is a description of the board and SoC hardware that Linux uses, for example, to know which peripherals are available for use and to associate drivers with them. Often, boot loaders do some on-the-fly "fixups" to the device tree to pass information to Linux.
9. In summary, the boot loader read kernel and device tree images from memory or mass storage device. Because boot loaders have many drivers, there are many possible choices for the location of the images.
   - NOR flash (serial QSPI or parallel)
   - NAND
   - SD/eMMC
   - USB mass storage devices of all types.
   - SATA drives of all types.
   - Ethernet, normally via TFTP.
10. After the boot loader loads the kernel and device tree and does fixups, it puts kernel boot parameters and the device tree into DDR where the kernel can find them and passes control to the kernel. One of the key kernel parameters is "root=". It tells the Linux kernel what device contains the user space file set (user land). U-Boot stores kernel parameters in environment variable `bootargs`.
11. Because the Linux kernel supports even more device drivers than boot loaders support, the array of choices for the user land device is even larger.

- NOR flash (serial QSPI or parallel)
- NAND (using special file systems)
- SD/eMMC
- USB mass storage devices of all types.
- SATA drives of all types.
- Ethernet, normally via NFS.
- RAM disks (which the boot loader populates)
- Third party PCIe-based mass storage devices and controllers
    a. SATA controllers
    b. SAS controllers
    c. Fibre Channel Host Bus Adaptors
    d. NVMe cards
    e. And more.

Once the kernel is up, it starts user land, starting with `systemd`. The startup process is part of the Ubuntu file set and conforms to normal Ubuntu procedures.

Notes on General Boot Principles

- In the future, the boot flow will be altered to have the boot ROM load the PPA before the boot loader. The PPA will then load and start the boot loader. This mechanism allows consolidation of more configuration into one place, the PPA. This configuration is then removed from the boot loaders.
- Secure boot does not change the overall sequence. The significant difference is that secure boot involves each component (starting with the boot ROM) validating the images it loads and starts. This sequence of image validations is called the “chain of trust”.

Linux often resets peripherals and reloads their firmware. This process is specific to SoC's.

5.2 U-Boot

5.2.1 LSDK U-Boot uses distro boot feature

As in previous versions of the NXP SDK, the U-Boot variable `bootcmd` contains commands that represent the default boot process. LSDK is different in that it uses a standard U-Boot feature called `distro boot` to make automatic booting more flexible. In distro boot, `bootcmd` runs additional commands in the variable `distro_bootcmd`. These commands are the heart of the distro boot process.

Distro boot sequential examines partitions on mass storage devices looking for a script file. When U-Boot finds one, it loads and executes it to initiate the boot process.

The mass storage devices to be searched are defined in the U-Boot environment variable `boot_targets`. Set it to control which mass storage devices are searched and the order in which they are searched. For example,

```
=> printenv boot_targets
boot_targets=usb0 mmc0 scsi0 dhcp
```

The command above shows the search order USB device 0, MMC (or SD) device 0, SCSI (SATA) device 0, followed by DHCP.

The process of searching involves a number of U-Boot variables. It ends with the variables shown below in an example from an LS2088ARDB.

```
=> printenv scan_dev_for_scripts
scan_dev_for_scripts=for script in ${boot_scripts}; do if test -e ${devtype} ${devnum}:$
{distro_bootpart} ${prefix}${script}; then echo Found U-Boot script ${prefix}${script}; run
boot_a_script; echo SCRIPT FAILED: continuing...; fi; done => printenv boot_scripts
boot_scripts=ls2088ardb_boot.scr => printenv boot_a_script boot_a_script=load ${devtype} $
{devnum}:${distro_bootpart} ${scriptaddr} ${prefix}${script}; env exists secureboot && load $
{devtype} ${devnum}:${distro_bootpart} ${scripthdraddr} ${prefix}${boot_script_hdr} &&
esbc_validate ${scripthdraddr};source ${scriptaddr}
```

The process searches for a script named by the variable `boot_scripts`. In this example, the search is for a script named `ls2088ardb_boot.scr`. When this script is located, it is loaded into RAM using the `load` command and run using the `source` command. This causes Linux to boot.

LSDK puts boot scripts into a file system on the second partition of a mass storage device. U-Boot can display files in a file system. Continuing the example, the following U-Boot commands list the files in the second partition of USB device 0 (do a `usb start` first):

```
=> ls usb 0:2
<DIR>       4096 .
<DIR>       4096 ..
<DIR>      16384 lost+found
        33301280 firmware_ls1043ardb_uboot_norboot.img
        33301280 firmware_ls1046ardb_uboot_qspiboot.img
        33301280 firmware_ls1088ardb_uboot_qspiboot.img
        33301280 firmware_ls2088ardb_uboot_norboot.img
        16524064 flex_linux_arm64.itb
           10005 fsl-ls1012a-frdm.dtb
           10145 fsl-ls1012a-qds.dtb
            8974 fsl-ls1012a-rdb.dtb
           30832 fsl-ls1043a-qds.dtb
           28619 fsl-ls1043a-rdb-sdk.dtb
           29342 fsl-ls1043a-rdb-sdk.dtb
           30134 fsl-ls1043a-rdb-usdpaa.dtb
           30010 fsl-ls1046a-qds.dtb
           27125 fsl-ls1046a-rdb-sdk.dtb
           27909 fsl-ls1046a-rdb-sdk.dtb
           28556 fsl-ls1046a-rdb-usdpaa.dtb
           14692 fsl-ls1088a-qds.dtb
           15451 fsl-ls1088a-rdb.dtb
           19713 fsl-ls2080a-qds.dtb
           19243 fsl-ls2080a-rdb.dtb
           20651 fsl-ls2088a-qds.dtb
           19545 fsl-ls2088a-rdb.dtb
<DIR>       4096 grub
            1152 hdr_ls1043ardb_bs.out
            1152 hdr_ls1046ardb_bs.out
        16654848 Image
         7443102 Image.gz
             703 ls1043ardb_boot.scr
             703 ls1046ardb_boot.scr
             862 ls1088ardb_boot.scr
             853 ls2088ardb_boot.scr
         9035568 perf
         8948941 ramdisk_rootfs_arm64.ext4.gz
         8949005 ramdisk_rootfs_arm64.ext4.gz.uboot
<DIR>       4096 secboot_hdrs
         7443166 uImage-4.4.65-dirty
<SYM>         19 vmlinuz
               0 c80546ca-02
```

It shows that this USB drive contains scripts (and necessary images) to boot any of the boards LS1043ARDB, LS1046ARDB, LS1088ARDB, and LS2088ARDB. For example, the LS2088ARDB boot script is `ls2088ardb_boot.scr`. The script files are binary. But one can boot Linux and look at them. LSDK mounts the boot partition containing the scripts at mount point `/boot`.

```
user@localhost:~$ ls /boot
c80546ca-02                              fsl-ls2088a-qds.dtb
firmware_ls1043ardb_uboot_norboot.img    fsl-ls2088a-rdb.dtb
firmware_ls1046ardb_uboot_qspiboot.img   grub firmware_ls1088ardb_uboot_qspiboot.img
hdr_ls1043ardb_bs.out
```

```
firmware_ls2088ardb_uboot_norboot.img   hdr_ls1046ardb_bs.out
flex_linux_arm64.itb                Image
fsl-ls1012a-frdm.dtb                    Image.gz
fsl-ls1012a-qds.dtb                     lost+found
flash_images.scr                        ls1012afrwy_boot.scr
fsl-ls1012a-rdb.dtb                     ls1043ardb_boot.scr
fsl-ls1043a-qds.dtb                     ls1046ardb_boot.scr
fsl-ls1043a-rdb-sdk.dtb                     ls1088ardb_boot.scr
fsl-ls1043a-rdb-sdk.dtb                 ls1088ardb_boot.scr
fsl-ls1043a-rdb-usdpaa.dtb              ls2088ardb_boot.scr
fsl-ls1046a-qds.dtb                     perf
fsl-ls1046a-rdb-sdk.dtb                     ramdisk_rootfs_arm64.ext4.gz
fsl-ls1046a-rdb-usdpaa.dtb              ramdisk_rootfs_arm64.ext4.gz.uboot
fsl-ls1088a-qds.dtb                     secboot_hdrs
fsl-ls1088a-rdb.dtb                     uImage-4.4.65-dirty
fsl-ls2080a-qds.dtb                     vmlinuz
fsl-ls2080a-rdb.dtb
```

The boot scripts are sophisticated due to secure boot. Ignore secure boot, and the key steps in a boot script are:

```
part uuid $devtype $devnum:3 partuuid3
setenv bootargs console=ttyS1,115200 earlycon=uart8250,mmio,0x21c0600 root=PARTUUID=
$partuuid3 rw rootwait $othbootargs default_hugepagesz=2m hugepagesz=2m hugepages=256 load
$devtype $devnum:2 $kernel_addr_r /vmlinuz; load $devtype $devnum:2 $fdt_addr_r /fsl-ls2088a-
rdb.dtb; bootm $kernel_addr_r - $fdt_addr_r
```

The distro boot search process sets the variables `devtype` and `devnum`. In this example, they would be "`usb`" and "`0`".

The U-Boot `part` command sets variable `partuuid3` to the partition universal unique identifier of partition 3 of USB device 0. This value is used in `bootargs` to identify the root partition to the Linux kernel. This method is better than using a device name (like `/dev/sda3`) because it is not dependent on probe order.

The next steps are to load the kernel image (`vmlinuz`) and device tree (`fsl-ls2088a-rdb.dtb`) into RAM and then boot Linux using `bootm`.

In summary (and ignoring secure boot), the distro boot processes searches for a partition with a file system containing a boot script. It loads and runs the boot script. The boot script does the five steps above to boot Linux.

To boot your own kernel, replace the kernel and device tree images in `/boot` and reboot your system. But also install any needed kernel modules first.

There are two types of userland in LSDK: 1) Large standard distro (Ubuntu rootfs) deployed on external SD/USB/SATA media storage. 2) Prebuilt tiny ramdisk rootfs(currently non-customizable) deployed in flash media onboard for arm32/arm64 target.

If U-Boot is used as boot loader, after LSDK is installed by flex-installer and reboots the target board, U-Boot will first automatically search for boot script `<platform>_boot.scr` from SD/eMMC/USB/SATA storage media, if a valid `<platform>_boot.scr` is found, U-Boot will boot the external distro (Ubuntu as default) deployed on SD/USB/SATA media storage, otherwise U-Boot will fall back to boot the tiny distro deployed on flash media onboard.

In case of booting LSDK tiny rootfs from flash media:

The default U-Boot environment `bootargs` is used and user can directly modify bootargs for custom kernel ondemand.

In case of booting LSDK distro from external SD/USB/SATA storage disk:

The default U-Boot environment '`bootargs`' is NOT used by external distro, `bootargs` is preset in `<platform>_boot.scr`, users can indirectly modify othbootargs ondemand, for example, `setenv othbootargs fsl_fm_max_frm=9600` in U-Boot prompt.

5.2.2 LSDK U-Boot flash image feature

In case user needs to flash different image (e.g. rcw/pbi, u-boot, ppa, dtb, kernel, etc) to current or other bank to evaluate a feature on Layerscape board,

for example, to evaluate TDM feature with the non-default rcw_1600_qetdm.bin on LS1043ARDB, execute the commands below:

```
=> setenv bd_type mmc
=> setenv board ls1043ardb
=> setenv bank other
=> ls $bd_type 0:2 flash_images/ls1043ardb/rcw
=> setenv img rcw
=> setenv rcw_img flash_images/ls1043ardb/rcw/RR_FQPP_1455/rcw_1600_qetdm.bin
=> load $bd_type 0:2 $load_addr flash_images.scr
=> source $load_addr
   Starting to Program the other bank of nor flash according to LSDK standard flash map:
   Using bd_type = mmc
   Using bd_part = 0:2
   Using img = rcw
   Using addr_rcw 0x64000000 for rcw_img flash_images/ls1043ardb/rcw/RR_FQPP_1455/
rcw_1600_qetdm.bin
   192 bytes read in 211 ms (0 Bytes/s)
   Un-Protected 1 sectors
   Erased 1 sectors
   Copy to Flash... 9done
   Success: flashed flash_images/ls1043ardb/rcw/RR_FQPP_1455/rcw_1600_qetdm.bin
```

Similarly, to evaluate LS2088A rev1.1 running at 2.0GHz with a different RCW image from the default one, follow example as below::

```
=> setenv bd_type usb
=> setenv board ls2088ardb
=> setenv img rcw
=> setenv rcw_img flash_images/ls2088ardb/rcw/rev1.1/FFFFFFFF_PP_HH_0x2a_0x41/rcw_2000.bin
=> load $bd_type 0:2 $load_addr flash_images.scr
=> source $load_addr
```

- To flash single image to current or other bank, set u-boot environment variable **img** to one of following variables : rcw, uboot, ppa, mcfw, mcdpc, mcdpl, fman, qe, pfe, phy, dtb, or kernel , For example:

  ```
  # for rcw
  => setenv img rcw
  => source $load_addr
  # for uboot
  => setenv img uboot
  => source $load_addr
  # for ppa
  => setenv img ppa
  => source $load_addr
  # for dtb
  => setenv img dtb
  => setenv dtb_img fsl-ls1043a-rdb-usdpaa.dtb
  => source $load_addr
  ```

- To flash all images to current or other bank, set u-boot environment variable img to all by command 'setenv img all'.
- If necessary, you can override the default setting of variable bd_part, flash_type, rcw_img, uboot_img, ppa_img, dtb_img, kernel_itb, qe_img, fman_img, phy_img, mcfw_img, mcdpl_img, mcdpc_img before running 'source $load_addr'.
- Note: for LS1088ARDB, please make sure first execute 'setenv board ls1088ardb' to set correct board name.

5.3 UEFI

Release Description

This section provides information about the LSDK UEFI release on QorIQ LS boards. The following features are supported in this release:

- DDR4
- DUART
- I2C
- DSPI

- SATA
- SD, FAT32 filesystem
- GPIO Support
- RTC, Watchdog
- PPA Integration
- PCIe – e1000 NIC card support
- DPAA 1.x support – XFI, RGMII, and SGMI
- DPAA2.x support - XFI
- SMP Linux boot via EFI_STUB on SD card
- PXE boot via PCIe and DPAA interfaces
- DDR BIST Test
- Ubuntu Distro boot support
- KASLR Support in UEFI
- QSPI boot
- USB 3.0
- Prefetch configuration support
- MC High Mem Support
- RTC Support for LS1043A x2 Board

Feature Summary

Features\Board	LS1043ARDB	LS2088ARDB	LS1046ARDB
DDR4	YES	YES	YES
UART	YES	YES	YES
I2C	Yes	YES	YES
DSPI	Yes	Yes	No
SATA	N/A	YES	YES
SD,FAT32 filesystem	YES	YES	YES
GPIO	YES	NO	YES
IFC-NAND	YES	YES	NO
IFC-NOR	YES	YES	N/A
RTC	YES	YES	YES
Watchdog	YES	YES	YES
PPA Integration	YES	YES	YES
PCIe – e1000 NIC	YES	YES	YES (PCIe Using legacy interrupt)
DPAA 1.x	YES	N/A	YES

Table continues on the next page...

Table continued from the previous page...

DPAA 2.x	N/A	YES	N/A
SMP Linux boot via EFI_STUB on SD card	YES	YES	YES
PXE boot via PCIe and DPAA interfaces	YES	YES	YES
DDR BIST Test	YES	YES	YES
Ubuntu Distro boot support	YES	YES	YES
KASLR Support in UEFI	YES	YES	YES
QSPI boot	No	NO	YES
USB 3.0	No	YES	NO
Prefetch Config support	No	YES	NO
MC High Mem Support	N/A	YES	N/A
Silicon Rev	1.0/1.1	1.0	1.0

Tool Chain

- `gcc-linaro-4.9-2016.02-x86_64_aarch64-linux-gnu-` Used to compile UEFI firmware (can be downloaded from `https://releases.linaro.org/components/toolchain/binaries/4.9-2016.02/aarch64-linux-gnu/gcc-linaro-4.9-2016.02-x86_64_aarch64-linux-gnu.tar.xz`)

Known issues

QUEFI-780: ls1043ardb_uefi reconnect hang

Limitation

On LS1046ARDB, QSPI flash is disabled during device tree fix-up and Linux will not be able to use QSPI flash as UEFI run time services will be using it for variables storage.

5.3.1 Introduction

Purpose

This section describes how to use the accompanying LSDK release on the QorIQ Layerscape platforms and how to boot LSDK distro with UEFI. The section covers UEFI enablement on QorIQ Layerscape platforms and does not describe UEFI specifications in detail.

References

- Unified Extensible Firmware Interface Specification
- QorIQ LS1043A Reference Manual
- QorIQ LS1046A Reference Manual

5.3.2 UEFI overview

UEFI (Unified Extensible Firmware Interface) describes an interface between the operating system (OS) and the platform firmware. The interface consists of data tables that contain platform-related information, plus boot and runtime service calls that are available to the operating system and its loader. Together, these provide a standard, modern environment for booting an operating system and running pre-boot applications.

UEFI implementations are governed by the UEFI specifications, which are designed as a pure interface specification. As such, the specification defines the set of interfaces and structures that platform firmware must implement.

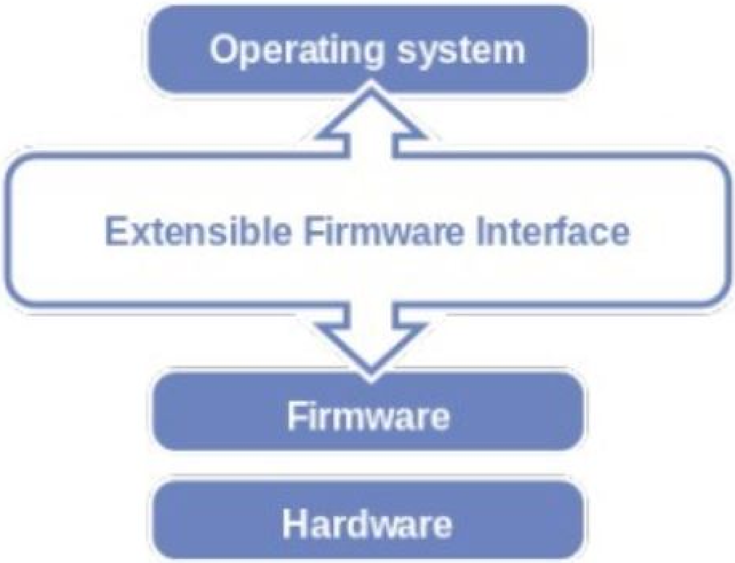


Figure 4. UEFI

For the latest version of UEFI, refer to References

UEFI Bootflow

The following is the Boot Execution Order on QorIQ Layerscape boards:

- Execution begins in the PBI state machine when the SoC comes out of reset
- After PBI, execution starts with SP bootcore which gives control to GPP Bootcore
- GPP bootcore gives control to UEFI
- Bootcore in UEFI branches to EL3 init code in PPA
- When PPA completes EL3 initialization, it branches back to UEFI in EL2
- Bootcore in UEFI branches to Linux kernel in EL1
- Linux Kernel invokes PSCI (`cpu_on`) to release secondary core
- Execution starts in the GPP bootrom when secondary core released from reset
- If core is marked to be disabled, core enters power-down sequence in bootrom
- Cores not disabled branch to EL3 init code in PPA
- Upon completion of EL3 initialization, cores branch to start address at EL1 in kernel

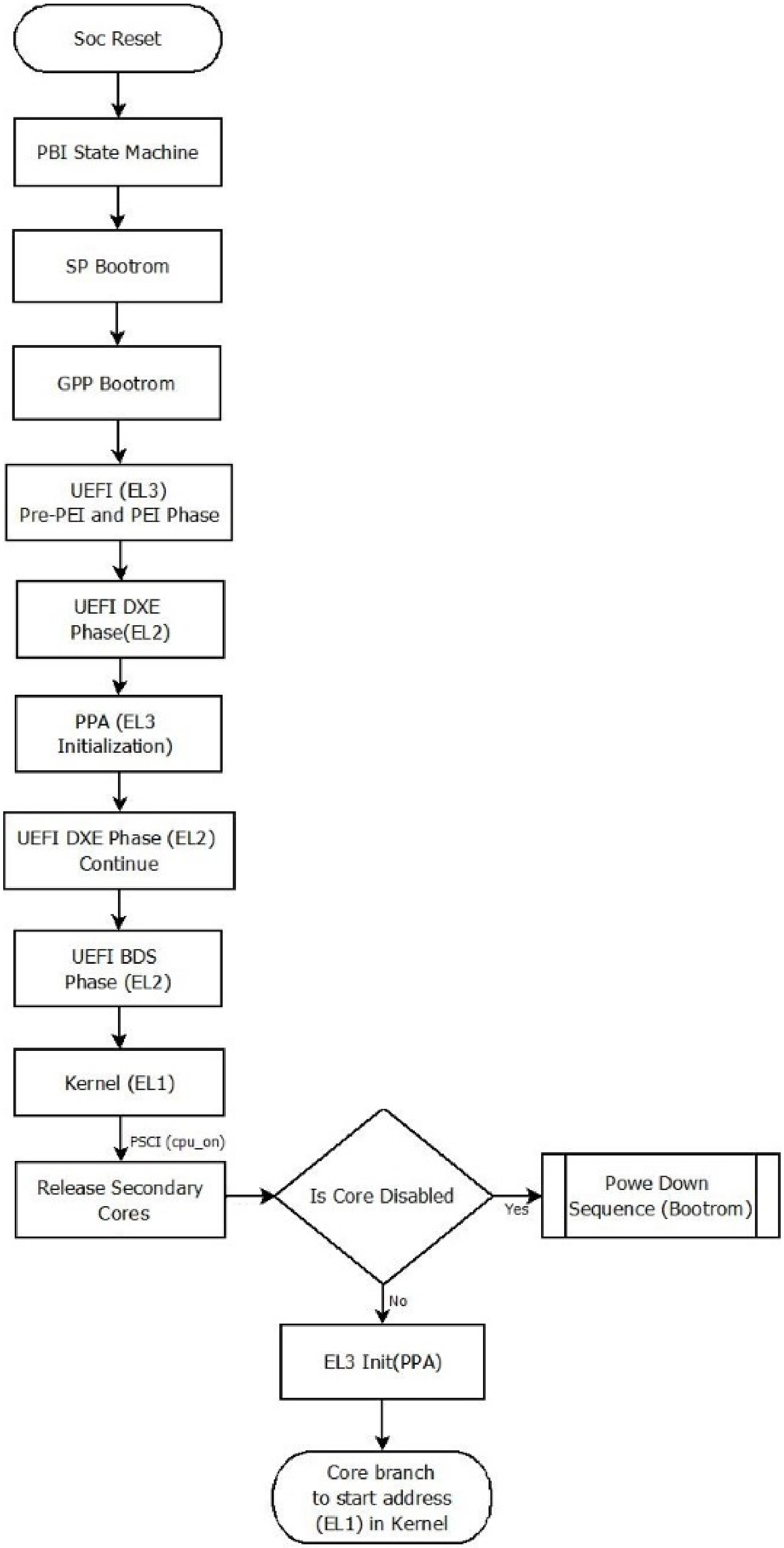


Figure 5. UEFI bootflow

Here, the PPA (Primary Protected Application) firmware is the platform and runtime security firmware which allows configuring and enforcing platform specific security policies.

Environment requirements

Hardware requirements

- Host PC: Ubuntu (64-bit variant with at least 2 GB RAM) host is preferred to compile/build the UEFI firmware.
- Board: QorIQ Layerscape, with UART cable.
- SD Card: Preferably from well-known vendors like SanDisk.

Software requirements

- To build the UEFI firmware on the Ubuntu host, install uuid-dev.

  ```
  $ sudo apt-get install uuid-dev
  ```

- To build the UEFI firmware on the Ubuntu host, install Linaro GCC-4.9 toolchain on your host machine using the following commands.

  ```
  $ wget https://releases.linaro.org/components/toolchain/binaries/4.9-2016.02/aarch64-
  linux-gnu/gcc-linaro-4.9-2016.02-x86_64_aarch64-linux-gnu.tar.xz
  $ tar -xvf gcc-linaro-4.9-2016.02-x86_64_aarch64-linux-gnu.tar.xz
  ```

  Add the toolchain paths to the `PATH` environment variable.

  ```
  $ export PATH=/home/user/linaro_4.9_2016/gcc-linaro-4.9-2016.02-x86_64_aarch64-linux-gnu/
  bin:$PATH
  ```

- Ensure that Python (2.7 or higher) is installed on the build machine, for successful compilation.

  ```
  $ python –version
   Python 2.7.12
  ```

5.3.3 LSDK distro boot with UEFI

Boot up Image Requirements

The following images are required for UEFI boot.

- UEFI images - UEFI boot loader images
- RCW (Reset Configuration Word) firmware
- PPA (Primary Protected Application) firmware - For secure monitor and associated functions that comprise the base EL3 sw foundation
- Device Tree image - Data structure for describing hardware
- FMAN (Frame Manager) Micro Code - Required for DPAA1 (Data Path Acceleration Architecture) interfaces (applicable for LS1043ARDB and LS1046ARDB)
- MC Firmware, DPL and DPC binaries for DPAA2 interfaces (applicable for LS2088ARDB)
- Phy Firmware for Cortina Phy (applicable for LS2088ARDB)

UEFI Boot Order

The UEFI boot manager will try to boot from all entries as they appear in the UEFI boot menu. Boot entries can be divided into the following three categories.

- Boot entries for Block devices
- Boot entries for Network Boot (PXE boot)
- Boot entry for UEFI Shell

For block devices, the UEFI boot manager will look for an EFI application with a predefined name (`BOOT{machine type short name}.EFI`) in `/EFI/BOOT`. If found, the boot manager will automatically run the EFI application. In our case (AArch64 ARM platforms), the application should be named as `BOOTAA64.EFI`.

Linux Boot Storage Media Layout

Table 39. Media Layout

Region 1 (4 KB)	Region 2 (64 MB)	Region 3 (20 MB)	Region 4 (1GB)	Region 5 (left space of disk)
MBR/GPT	Loaders & FW UEFI PPA firmware FMan or MC firmware lsdk_linux_arm64_tiny.itb	Partition 1 (FAT 16/32) LABEL: EFI BOOTAA64.EFI grub.cfg	Partition 2 (EXT4) LABEL: boot Kernel DTB lsdk_linux_arm64_tiny.itb distro boot scripts other	Partition 3 (EXT4) LABEL: rootfs Ubuntu or Ubuntu-Core or CenOS or Debian

In the Linux environment, fdisk utility can be used to partition and format the target as per the table above and then copy the required images (Kernel image, Rootfs) to the target.

For LSDK, `flex-builder` & `flex-installer` utility can be used to partition and install required images to target as per above layout. For more information on how to build and install LSDK using flex-builder and flex-installer, refer to Layerscape SDK user guide on page 46.

Image name	Partition
BOOTAA64.EFI, grub.cfg	EFI partition
Kernel image	boot partition
Root file system	rootfs partition

NOTE

- The `Kernel Image` should be the standard kernel images build as `arch/arm64/boot/Image` (for arm64).
- The device tree has to be stored in flash at a fixed offset (board specific) as per the LSDK flash layout. For NOR flash, default value is (`0x60F00000` to `0x60FFFFFF`). Update '`PcdFdtDevicePaths`' PCD in platform description file (.dsc) for a different flash layout.

Sample files

- **BOOTAA64.EFI**

  Represents grub boot loader. It will load the `grub.cfg` kept in the same directory and provides grub menu to the user. The user can select the required menu entry. Follow '**Generate BOOTAA64.EFI**' for compilation steps.

- **grub.cfg**

  grub.cfg provides boot options to user. For example, the grub.cfg can be used for booting the LSDK distro.

```
set default="1"
set timeout=10

menuentry 'LSDK on QorIQ ARM64 ls1043ardb' {
    search --no-floppy --file /42013eab-02 --set root
    linux /Image console=ttyS0,115200 root=PARTUUID=42013eab-03 rootwait rw
earlycon=uart8250,mmio,0x21c0600
}
```

NOTE

The example above uses a LSDK distribution. Specific kernel boot arguments could vary per distribution.

The table below represents the configurable parameters in grub.cfg.

• Option	Explanation	Comments
• *search --no-floppy --file /<FileName> --set root*	Search all partitions for the given file name. First Partition containing the specified file is set at root so that Grub will look for required image (kernel image) in this partition.	For example, for LSDK, flex-installer will update the FileName as PARTUUID of the partition containing Kernel image. This removes the ambiguity of finding a specific partition based on UUID/LABEL when multiple devices are connected.
• *root=PARTUUID=XXXXXXXX-YY*	It represents the PARTUUID of the partition containing rootfs This is passed as boot argument to kernel.	For example, for LSDK, flex-installer will update it with PARTUUID of the partition containing rootfs. This make sure that correct rootfs is passed to kernel and removes the ambiguity of finding the correct rootfs based on UUID/LABEL when multiple devices are connected.
• *set timeout=N*	If defined, GRUB will wait 'N' Seconds, before booting the default menu entry.If not defined, user always has to select the required menu entry.	Adjust the timeout value as per requirement.

LSDK Distro Bootflow with UEFI

- All required files (RCW, PPA, Device Tree, FMan, and UEFI images) are stored in NOR flash.
- UEFI boot starts from NOR
- When prompted, press ESCAPE to select a Boot Option (UEFI SHELL/PXE Boot) OR else UEFI boot manager tries to boot from all boot entries starting from 'Removable Media' followed by 'Network Boot' and 'UEFI SHELL'
- If a Removable Media (e.g. SD card) has a FAT formatted partition with */EFI/BOOT/BOOT{machine type short name}.EFI* (For example, BOOTAA64.EFI for arm64), it will be executed by UEFI boot manager.
- BOOTAA64.EFI will load grub.cfg.
- grub.cfg contains menu entry for distro along with required parameter to identify kernel image and pass 'rootfs' path to kernel as boot argument.

Flow diagram

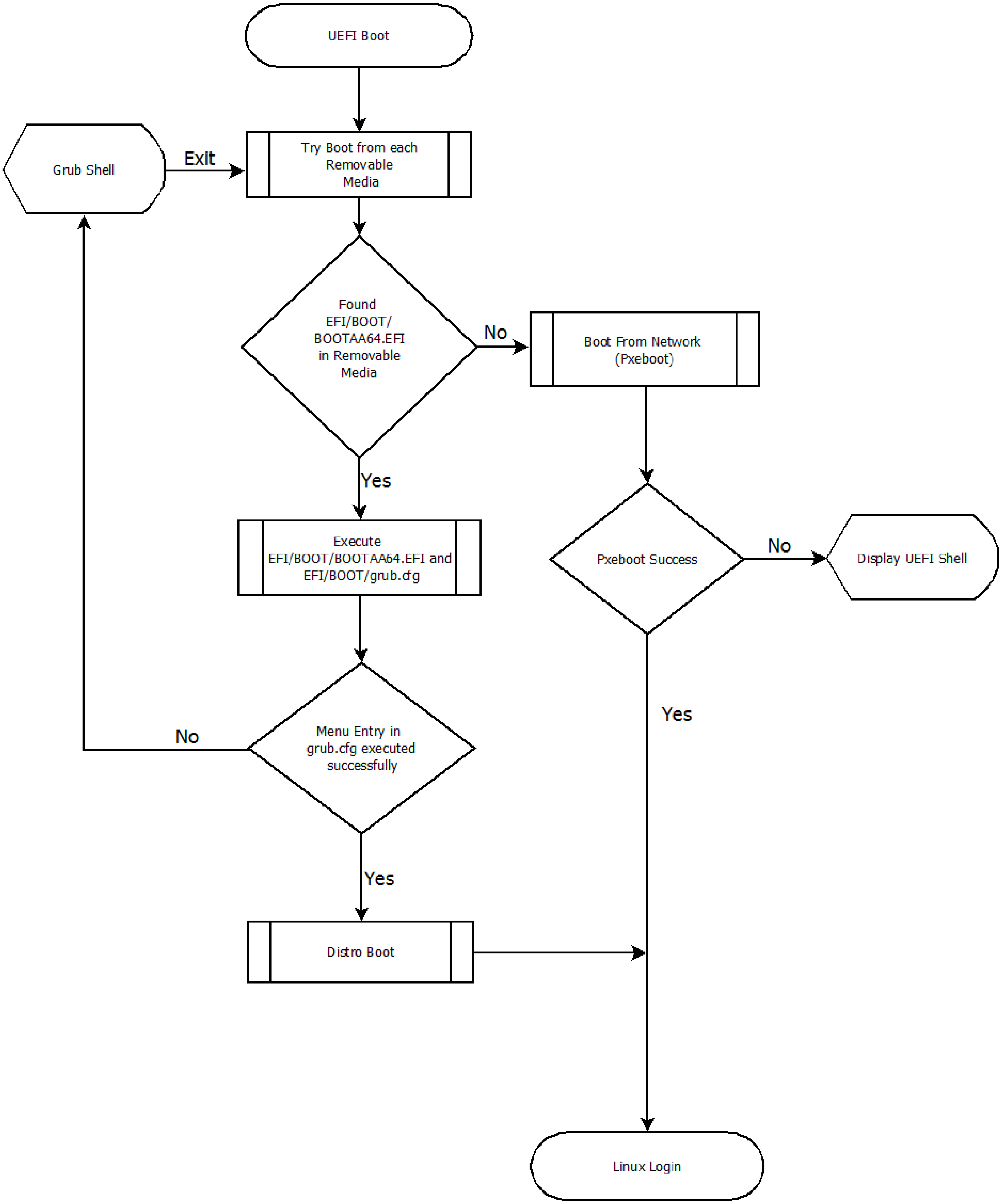

UEFI Boot
Try Boot from each Removable Media
Grub Shell
Exit
Found EFI/BOOT/ BOOTAA64.EFI in Removable Media
No
Boot From Network (Pxeboot)
Yes
Execute EFI/BOOT/BOOTAA64.EFI and EFI/BOOT/grub.cfg
Pxeboot Success
No
Display UEFI Shell
Menu Entry in grub.cfg executed successfully
No
Yes
Yes
Distro Boot
Linux Login

Figure 6. UEFI Bootflow Diagram

Generate BOOTAA64.EFI

Follow the steps below to compile the grub source code to generate BOOTAA64.EFI. In general, BOOTAA64.EFI is provided by the distribution like LSDK provides prebuild BOOTAA64.EFI.

- Get the grub source code and install the prerequisites as mentioned in INSTALL file.

```
$ git clone git://git.savannah.gnu.org/grub.git;
$ cd grub
```

- Use grub-2.02 tag.

```
$ git checkout tags/grub-2.02
```

- Set the toolchain path and set CROSS_COMPILE environment variable. See the Software Requirements section to download and install the toolchain.

```
$ export PATH=/home/user/gcc-linaro-4.9-2016.02-x86_64_aarch64-linux-gnu/bin:$PATH
$ export CROSS_COMPILE=aarch64-linux-gnu-
```

- Configure and compile source code for target (arm64)

```
$ ./autogen.sh
$ ./configure --target=aarch64-linux-gnu
$ make
```

- Create standalone GRUB image.

```
$ echo 'configfile ${cmdpath}/grub.cfg' > grub.cfg
$ grub-mkstandalone --directory=./grub-core -O arm64-efi -o BOOTAA64.EFI --modules
"part_gpt part_msdos" /boot/grub/grub.cfg=./grub.cfg
```

NOTE

- GRUB standalone application has all the modules embedded in application itself and capability to recognize different file system (ext2, ext4, and so on), thus removing the need for having a separate directory populated with all of the GRUB UEFI modules and other related files.
- 'configfile ${cmdpath}/grub.cfg' instruct GRUB EFI (BOOTAA64.EFI) to use grub.cfg placed in same directory. Thus, making them portable.
- Option *–modules=”part_gpt part_msdos”* (with the quotes) is necessary for *${cmdpath}* feature to work properly and to recognize MBR and GPT partitioning.

Conventions for UEFI and U-Boot compatibility

UEFI requires that the device tree binary (dtb) be stored along with other firmware. This means that for the use case where

- Firmware (RCW, PPA, Fman/MC ucode, boot loader, and so on) is in NOR flash
- Kernel image is on /boot on a mass storage device
- Root file system is in a mass storage device (/)

—

U-Boot must boot Linux with the dtb in NOR (not in a kernel itb) but the kernel image (not itb form) is stored in /boot. This means that U-Boot boot.scr must extload a kernel image but not the dtb. It is consistent with what the LSDK specification says about using booti.

5.3.4 Product Execution

5.3.4.1 Flash Layout

Some prebuilt images will be stored into flash on board as per the table below:

Definition	Max Size	NOR/QSPI/NAND Flash Offset	SD Card Start Block No.
RCW+PBI	1MB	0x00000000	0x00008
Boot firmware (U-Boot, UEFI)	2MB	0x00100000	0x00800
Boot firmware Environment	1MB	0x00300000	0x01800
PPA firmware	2MB	0x00400000	0x02000
Secure boot headers	3MB	0x00600000	0x03000
DPAA1 FMAN ucode	256KB	0x00900000	0x04800
QE / uQE firmware	256KB	0x00940000	0x04A00
Ethernet PHY firmware	256KB	0x00980000	0x04C00
Scripts	256KB	0x009C0000	0x04E00
DPAA2 MC	3MB	0x00A00000	0x05000
DPAA2 DPL	1MB	0x00D00000	0x06800
DPAA2 DPC	1MB	0x00E00000	0x07000
Device tree	1MB	0x00F00000	0x07800
kernel	16MB	0x01000000	0x08000
ramdisk rootfs	32MB	0x02000000	0x10000

5.3.4.2 LS1043ARDB

Obtain the UEFI source code

```
$ git clone https://source.codeaurora.org/external/qoriq/qoriq-components/uefi; cd uefi
$ git checkout tags/LS1043ARDB-QUEFI_DASH_V5-0_1806
```

Compile the UEFI image

1. Set the toolchain path. Follow the 'Software Requirements' section to install and set the toolchain path.
2. Compile the source code to boot via NOR flash
   a. To compile UEFI source code to boot via NOR flash, use the following commands.

   ```
   $ cd uefi
   $ source edksetup.sh
   $ cd LS1043aRdbPkg
   $ source ls1043a_env.cshrc
   $ cd ../BaseTools
   $ make clean
   $ make
   ```

   If you face `g++ command not found error` (most likely on Ubuntu), while making BaseTools then run the following commands:

   ```
   $sudo apt-get install g++
   $ make clean
   $ make
   ```

   For booting via NOR flash, use the command below for compilation.

```
$ cd ../LS1043aRdbPkg
$./build.sh RELEASE XIP clean
$./build.sh RELEASE XIP
```

b. After a successful compilation, the UEFI firmware image is built at this path:

```
UEFI Firmware Image:
uefi/Build/LS1043aRdb/RELEASE_GCC49/FV/LS1043ARDB_EFI.fd

UEFI Non Volatile Variables Image:
uefi/Build/LS1043aRdb/RELEASE_GCC49/FV/LS1043ARDBNV_EFI.fd
```

NOTE

In case the compilation fails, ensure the following:

ARCH environment variable is unset:

```
$ unset ARCH
```

GCC49 AARCH64_PREFIX variable is set properly:

```
$ export GCC49_AARCH64_PREFIX=' aarch64-linux-gnu-'
```

Booting LS1043ARDB to UEFI prompt (via NOR Boot)

- **Prerequisites for NOR boot**

  TFTP server is required so relevant binaries can be flashed via NOR flash within the `/tftpboot` folder. Make sure that the correct permissions are set for this folder.

- **LS1043A NOR flash map**

  Flash Layout on page 148 represents NOR flash available on LS1043ARDB. The base address for the primary bank (VBank0/Bank 0 64MB) is `0x60000000` and the base address for secondary bank (VBank4/Bank 4 64MB) is `0x64000000`.

NOTE

For a different flash layout, offsets/address as shown in Flash Layout on page 148 should be changed in `dec` (platform declaration file), `dsc` (platform description file), `fdf` (firmware device file) and `VarStore.fdf.inc` (firmware device file for non-volatile variables).

Flashing UEFI images on NOR flash bank 4 (alternate NOR flash bank)

- Boot to U-Boot prompt from NOR flash primary bank (Bank 0).

  1. Setup serial port connection on host machine to capture logs from the LS1043ARDB.
  2. Reset the board in order to boot U-Boot on bank 0. Make sure that a valid U-Boot image is flashed.

- Copy Images to NOR flash alternate bank using following commands:

```
=> setenv uefi 'tftp 0xa0000000 LS1043ARDB_EFI.fd && erase 0x64100000 +$filesize && cp.b
0xa0000000 0x64100000 $filesize'
=> run uefi

=> setenv uefi_env 'tftp 0xa0000000 LS1043ARDBNV_EFI.fd && erase 0x64300000 +$filesize &&
cp.b 0xa0000000 0x64300000 $filesize'
=> run uefi_env

=> setenv rcw 'tftp 0xa0000000 rcw_1500.bin && erase 0x64000000 0x640FFFFF && cp.b
0xa0000000 0x64000000 $filesize'
=> run rcw

=> setenv dtb 'tftp 0xa0000000 fsl-ls1043a-rdb-sdk.dtb && erase 64F00000 +$filesize && cp.b
0xa0000000 64F00000 $filesize'
=> run dtb

=> setenv ppa 'tftp 0xa0000000 ppa.itb && erase 0x64400000 +$filesize && cp.b 0xa0000000
0x64400000 $filesize'
```

```
=> run ppa
Note: PPA should be built without enabling OPTEE.
=> setenv fman 'tftp 0xa0000000 fsl_fman_ucode_ls1043_r1.1_108_4_9.bin && protect off
0x64900000 +$filesize && erase 0x64900000 +$filesize && cp.b 0xa0000000
0x64900000 $filesize && protect on 0x64900000 +$filesize'
=> run fman
```

The precompiled binary containing the images above can be downloaded from http://www.nxp.com (`wget http://www.nxp.com/lgfiles/sdk/lsdk1806/firmware_ls1043ardb_uefi_norboot.img`). Follow the steps below to use the precompiled image:

```
Boot to U-Boot prompt from NOR flash primary bank (Bank 0)
=> setenv raddr_temp 0xa0000000
=> setenv faddr_firmware 0x64000000
=> tftp $raddr_temp firmware_ls1043ardb_uefi_norboot.img
=> if cmp.b $raddr_temp $faddr_firmware $filesize; then echo "No update"; else protect off
$faddr_firmware +$filesize && erase $faddr_firmware +$filesize && cp.b $raddr_temp
$faddr_firmware $filesize && protect on $faddr_firmware +$filesize; fi
=> cpld reset altbank
```

NOTE

DTB present in composite firmware should match the Linux Kernel (which needs to be booted). To boot different Linux kernel, update the DTB.

- Reset RDB to boot from NOR flash alternate bank (Bank 4).

  Use the command below to switch to bank 4:

  ```
  => cpld reset altbank
  ```

5.3.4.3 LS1046ARDB

Obtain the UEFI source code

```
$ git clone https://source.codeaurora.org/external/qoriq/qoriq-components/uefi; cd uefi
$ git checkout tags/LS1046ARDB-QUEFI_DASH_V5-0_1806
```

Compile the UEFI image

1. Set the toolchain path. Follow the 'Software Requirements' section to install and set the toolchain path.
2. Compile the source code to boot via QSPI flash
   a. To compile UEFI source code for QSPI flash boot, use the following commands:

      ```
      $ cd uefi
      $ source edksetup.sh
      $ cd LS1046aRdbPkg
      $ source ls1046a_env.cshrc
      $ cd ../BaseTools
      $ make clean
      $ make
      ```

      If you face `g++ command not found error` (most likely on Ubuntu) while making BaseTools, then run the following commands:

      ```
      $sudo apt-get install g++
      $ make clean
      $ make
      ```

      For booting via QSPI flash, use the command below for compilation.

      ```
      $ cd ../LS1046aRdbPkg
      $./build.sh RELEASE QSPI clean
      $./build.sh RELEASE QSPI
      ```

   b. After a successful compilation, the UEFI firmware image is built at this path:

```
UEFI Firmware Image:
uefi/Build/LS1046aRdb/RELEASE_GCC49/FV/LS1046ARDB_EFI.fd

UEFI Non Volatile Variables Image:
uefi/Build/LS1046aRdb/RELEASE_GCC49/FV/LS1046ARDBNV_EFI.fd
```

NOTE

In case the compilation fails, ensure the following:

ARCH environment variable is unset:

```
$ unset ARCH
```

GCC49 AARCH64_PREFIX variable is set properly:

```
$ export GCC49_AARCH64_PREFIX=' aarch64-linux-gnu-'
```

Booting LS1046ARDB to UEFI prompt (via QSPI Boot)

- **Prerequisites for QSPI boot**

  TFTP server is required so relevant binaries can be flashed via QSPI flash within the `/tftpboot` folder. Make sure that the correct permissions are set for this folder.

- **LS1046A QSPI flash map**

  Flash Layout on page 148 represents QSPI flash available on LS1046ARDB. The base address for the primary bank (VBank0/Bank 0 64MB) is `0x40000000` and the base address for secondary bank (VBank4/Bank 4 64MB) is `0x44000000`..

NOTE

For a different flash layout, offsets/address as shown in Flash Layout on page 148 should be changed in `dec` (platform declaration file), `dsc` (platform description file), `fdf` (firmware device file) and `VarStore.fdf.inc` (firmware device file for non-volatile variables).

Flashing UEFI images on QSPI flash bank 4 (alternate QSPI flash bank)

- Boot to U-Boot prompt from QSPI flash primary bank (Bank 0).
  1. Setup serial port connection on host machine to capture logs from the LS1046ARDB.
  2. Reset the board in order to boot U-Boot on bank 0. Make sure that a valid U-Boot image is flashed.
- Copy Images to QSPI flash alternate bank using following commands:

```
=> sf probe 0:1
=>setenv uefi 'tftpboot 0x82000000 LS1046ARDB_EFI.fd; sf erase 100000 +$filesize && sf
write 0x82000000 100000 $filesize'
=> run uefi

=> setenv uefi_env ' tftpboot 0x82000000 LS1046ARDBNV_EFI.fd;sf erase 300000 +$filesize &&
sf write 0x82000000 300000 $filesize'
=> run uefi_env

=> setenv rcw 'tftpboot 0x82000000 rcw_1800_qspiboot.bin.swapped; sf erase 0 +$filesize &&
sf write 0x82000000 0 $filesize'
=> run rcw

=> setenv dtb 'tftpboot 0x82000000 fsl-ls1046a-rdb-sdk.dtb; sf erase F00000 +$filesize &&
sf write 0x82000000 F00000 $filesize '
=> run dtb

=> setenv ppa 'tftpboot 0x82000000 ppa.itb;sf erase 400000 +$filesize && sf write
0x82000000 400000 $filesize'
=> run ppa
Note: PPA should be built without enabling OPTEE.
=> setenv fman 'tftpboot 0x82000000 fsl_fman_ucode_ls1046_r1.0_108_4_9.bin; sf erase
```

```
900000 +$filesize && sf write 0x82000000 900000 $filesize'
=> run fman
```

The precompiled binary containing the images above can be downloaded from http://www.nxp.com (`wget http://www.nxp.com/lgfiles/sdk/lsdk1806/firmware_ls1046ardb_uefi_qspiboot.img`). Follow the steps below to use the precompiled image:

```
Boot to U-Boot prompt from QSPI flash primary bank (Bank 0)
=> setenv raddr_temp 0xa0000000
=> setenv raddr_temp_2 0x90000000
=> setenv faddr_firmware 0x0
=> tftp $raddr_temp firmware_ls1046ardb_uefi_qspiboot.img
=> sf probe 0:1
=> sf read $raddr_temp_2 $faddr_firmware $filesize
=> if cmp.b $raddr_temp $raddr_temp_2 $filesize; then echo "No update"; else sf erase
$faddr_firmware +$filesize  && sf write $raddr_temp $faddr_firmware $filesize; fi
=> cpld reset altbank
```

NOTE

DTB present in composite firmware should match the Linux Kernel (which needs to be booted). To boot different Linux kernel, update the DTB..

- Reset RDB to boot from QSPI flash alternate bank (Bank 4).

  Use the command below to switch to bank 4:

  ```
  => cpld reset altbank
  ```

5.3.4.4 LS2088ARDB

Obtain the UEFI source code

```
$ git clone https://source.codeaurora.org/external/qoriq/qoriq-components/uefi; cd uefi
$ git checkout tags/LS2088ARDB-QUEFI_DASH_V5-0_1806
```

Compile the UEFI image

1. Set the toolchain path. Follow the 'Software Requirements' section to install and set the toolchain path.
2. Compile the source code to boot via NOR flash
   a. To compile UEFI source code to boot via NOR flash, use the following commands.

      ```
      $ cd uefi
      $ source edksetup.sh
      $ cd LS2088aRdbPkg
      $ source ls2088a_env.cshrc
      $ cd ../BaseTools
      $ make clean
      $ make
      ```

      If you face `g++ command not found error` (most likely on Ubuntu), while making BaseTools then run the following commands:

      ```
      $sudo apt-get install g++
      $ make clean
      $ make
      ```

      For booting via NOR flash, use the command below for compilation.

      ```
      $ cd ../LS2088aRdbPkg
      $./build.sh RELEASE XIP clean
      $./build.sh RELEASE XIP
      ```

   b. After a successful compilation, the UEFI firmware image is built at this path:

      ```
      UEFI Firmware Image:
      uefi/Build/LS2088aRdb/RELEASE_GCC49/FV/LS2088ARDB_EFI.fd
      ```

```
UEFI Non Volatile Variables Image:
uefi/Build/LS2088aRdb/RELEASE_GCC49/FV/LS2088ARDBNV_EFI.fd
```

NOTE

In case the compilation fails, ensure the following:

ARCH environment variable is unset:

```
$ unset ARCH
```

GCC49 AARCH64_PREFIX variable is set properly:

```
$ export GCC49_AARCH64_PREFIX=' aarch64-linux-gnu-'
```

Booting LS2088ARDB to UEFI prompt (via NOR Boot)

- **Prerequisites for NOR boot**

  TFTP server is required so relevant binaries can be flashed via NOR flash within the `/tftpboot` folder. Make sure that the correct permissions are set for this folder.

- **LS2088A NOR flash map**

  Flash Layout on page 148 represents NOR flash available on LS2088ARDB. The base address for the primary bank (VBank0/Bank 0 64MB) is `0x580000000` and the base address for secondary bank (VBank4/Bank 4 64MB) is `0x584000000`.

NOTE

For a different flash layout, offsets/address as shown in Flash Layout on page 148 should be changed in `dec` (platform declaration file), `dsc` (platform description file), `fdf` (firmware device file) and `VarStore.fdf.inc` (firmware device file for non-volatile variables).

Flashing UEFI images on NOR flash bank 4 (alternate NOR flash bank)

- Boot to U-Boot prompt from NOR flash primary bank (Bank 0).
  1. Setup serial port connection on host machine to capture logs from the LS2088A RDB.
  2. Reset the board in order to boot U-Boot on bank 0. Make sure that a valid U-Boot image is flashed.
- Copy Images to NOR flash alternate bank using following commands:

```
=> setenv rcw 'tftp 0xa0000000 rcw_1800.bin; erase 0x584000000 0x5840FFFFF; cp.b 0xa0000000
0x584000000 $filesize;'
=> run rcw

=> setenv uefi 'tftp 0xa0000000 LS2088ARDB_EFI.fd; erase 0x584100000 0x5842FFFFF; cp.b
0xa0000000 0x584100000 $filesize'
=> run uefi

=> setenv uefi_nv 'tftp 0xa0000000 LS2088ARDBNV_EFI.fd; erase 0x584300000 0x5843FFFFF;
cp.b 0xa0000000 0x584300000 $filesize'
=> run uefi_nv

=> setenv mc 'tftp 0xa0000000 mc_10.4.0_ls2088a_20171101.itb; erase 0x584A00000
0x584EFFFFF; cp.b 0xa0000000 0x584A00000 $filesize ; tftp 0xa0000000 dpc-bman-4M.
0x2A_0x41.dtb ; cp.b 0xa0000000 0x584E00000 $filesize ; tftp 0xa0000000 dpl-eth.
0x2A_0x41.dtb ;cp.b 0xa0000000 0x584D00000 $filesize'
=> run mc

=> setenv ppa 'tftp 0xa0000000 ppa.itb; erase 0x584400000 0x5844FFFFF ; cp.b 0xa0000000
0x584400000 $filesize'
=> run ppa

=> setenv dtb 'tftp 0xa0000000 fsl-ls2088a-rdb.dtb; erase 0x584F00000 0x584FFFFFF; cp.b
0xa0000000 0x584F00000 $filesize'
=> run dtb
=> setenv cortina 'tftp 0x80000000 cs4315-cs4340-PHY-ucode.txt; erase 0x584980000
0x5849BFFFF ; cp.b 0x80000000 0x584980000 $filesize'
=> run cortina
```

The precompiled binary containing the images above can be downloaded from http://www.nxp.com (`wget http://www.nxp.com/lgfiles/sdk/lsdk1806/firmware_ls2088ardb_uefi_norboot.img`). Follow the steps below to use the precompiled image:

```
Boot to U-Boot prompt from NOR flash primary bank (Bank 0)
=> setenv raddr_temp 0xa0000000
=> setenv faddr_firmware 0x584000000
=> tftp $raddr_temp firmware_ls2088ardb_uefi_norboot.img
=> if cmp.b $raddr_temp $faddr_firmware $filesize; then echo "No update"; else protect off
$faddr_firmware +$filesize && erase $faddr_firmware +$filesize && cp.b $raddr_temp
$faddr_firmware $filesize && protect on $faddr_firmware +$filesize; fi
=> qixis_reset altbank
```

NOTE

DTB present in composite firmware should match the Linux Kernel (which needs to be booted). To boot different Linux kernel, update the DTB.

- Reset RDB to boot from NOR flash alternate bank (Bank 4).

  Use the command below to switch to bank 4:

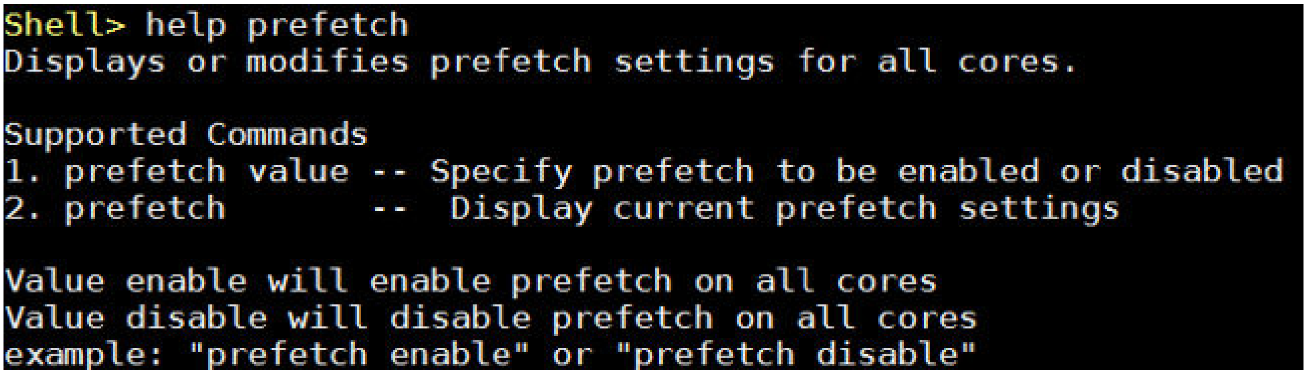
  ```
  => qixis_reset altbank
  ```

Prefetch Enable/Disable support

LS2088ARDB can enable/disable hardware LOAD and STORE prefetching. Some applications require hardware prefetching to be disabled in order to achieve better performance.

- Run the command below from the UEFI shell to get help for prefetch command.

  ```
  => help prefetch
  ```



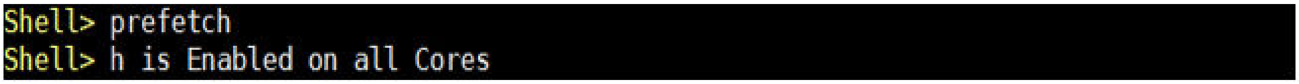
```
Shell> help prefetch
Displays or modifies prefetch settings for all cores.

Supported Commands
1. prefetch value -- Specify prefetch to be enabled or disabled
2. prefetch       --  Display current prefetch settings

Value enable will enable prefetch on all cores
Value disable will disable prefetch on all cores
example: "prefetch enable" or "prefetch disable"
```

- Run the command below from the UEFI shell to get the current prefetch settings.

NOTE

By default, prefetch is enabled on all the cores.

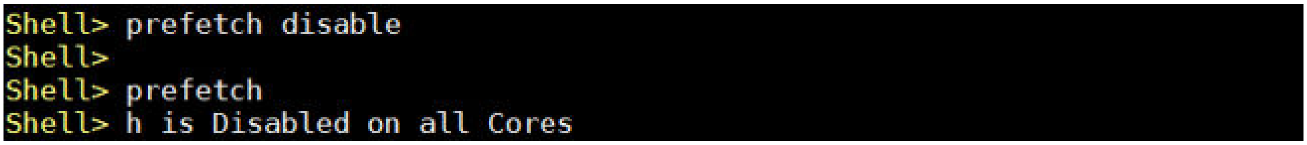
```
=> prefetch
```



```
Shell> prefetch
Shell> h is Enabled on all Cores
```

- Run the command below from the UEFI shell to disable prefetch on all the cores.

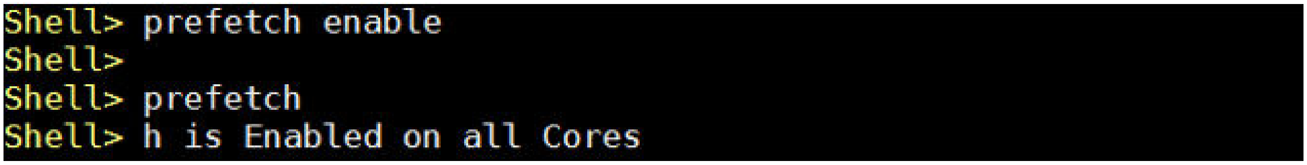
  ```
  => prefetch disable
  ```



```
Shell> prefetch disable
Shell>
Shell> prefetch
Shell> h is Disabled on all Cores
```

- Run the command below from the UEFI shell to enable prefetch on all the cores.

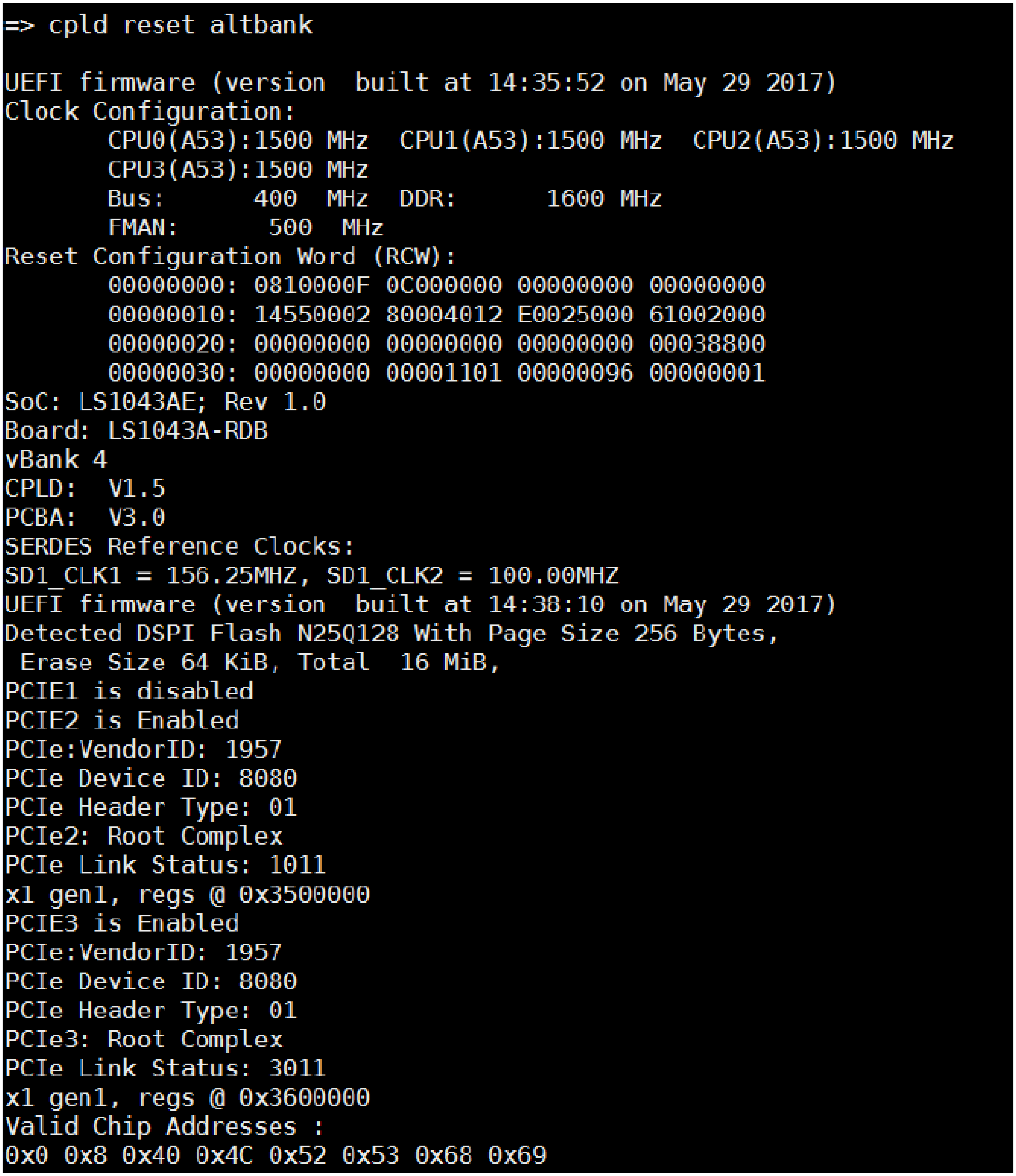
  ```
  => prefetch enable
  ```



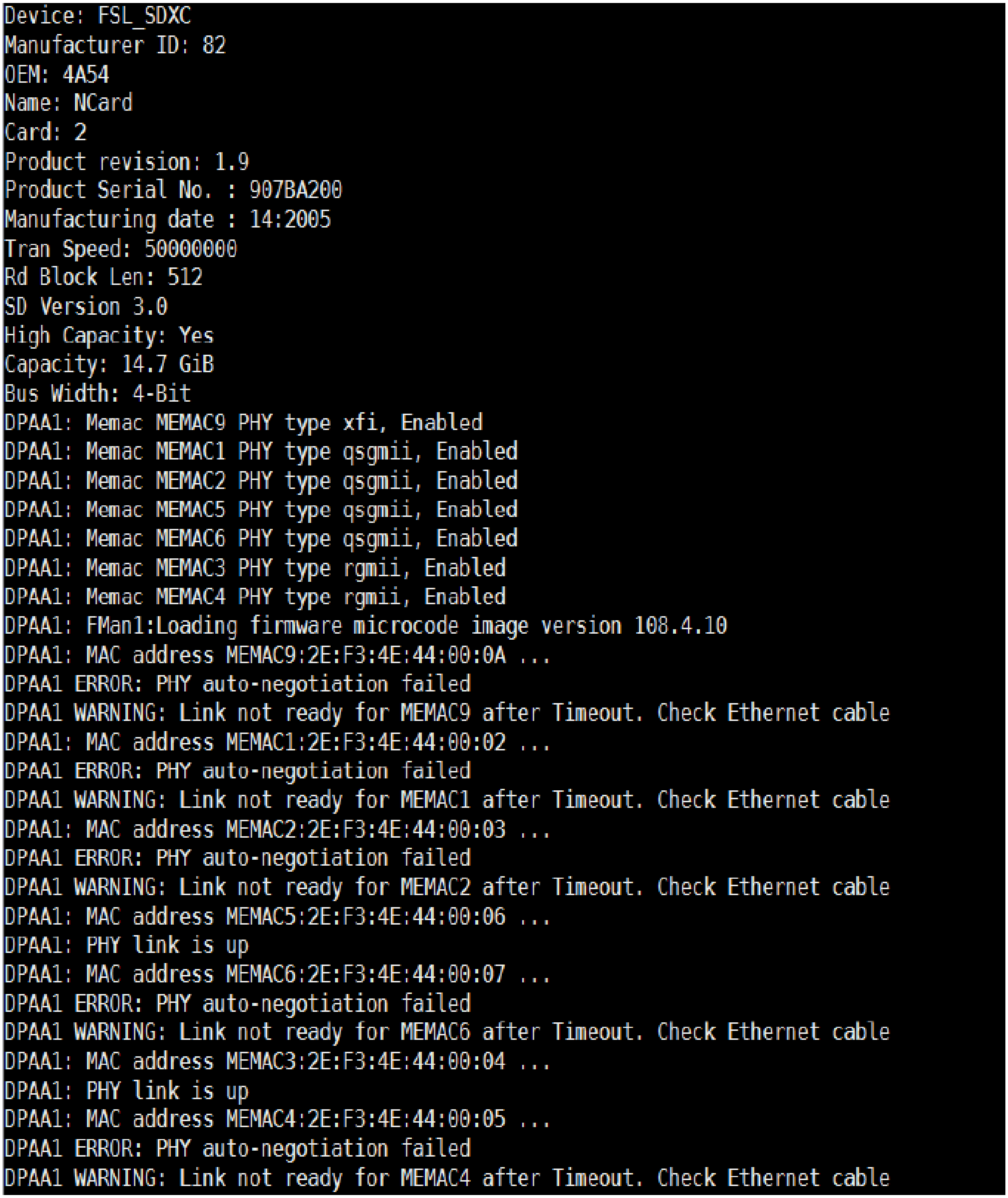
```
Shell> prefetch enable
Shell>
Shell> prefetch
Shell> h is Enabled on all Cores
```

5.3.5 LSDK Distro Boot Logs

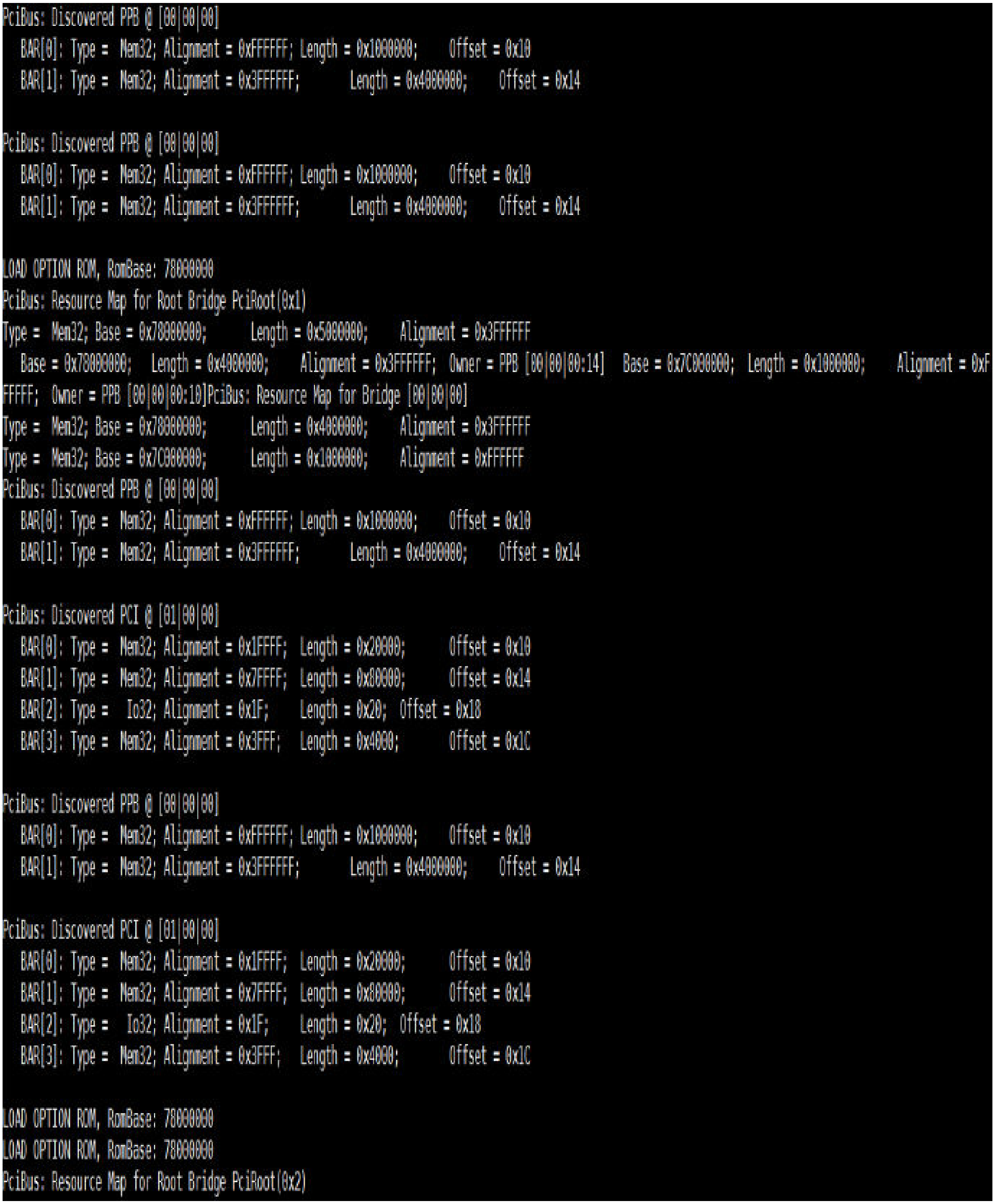
Below are the reference boot logs of LSDK distro boot on QorIQ LS1043ARDB .



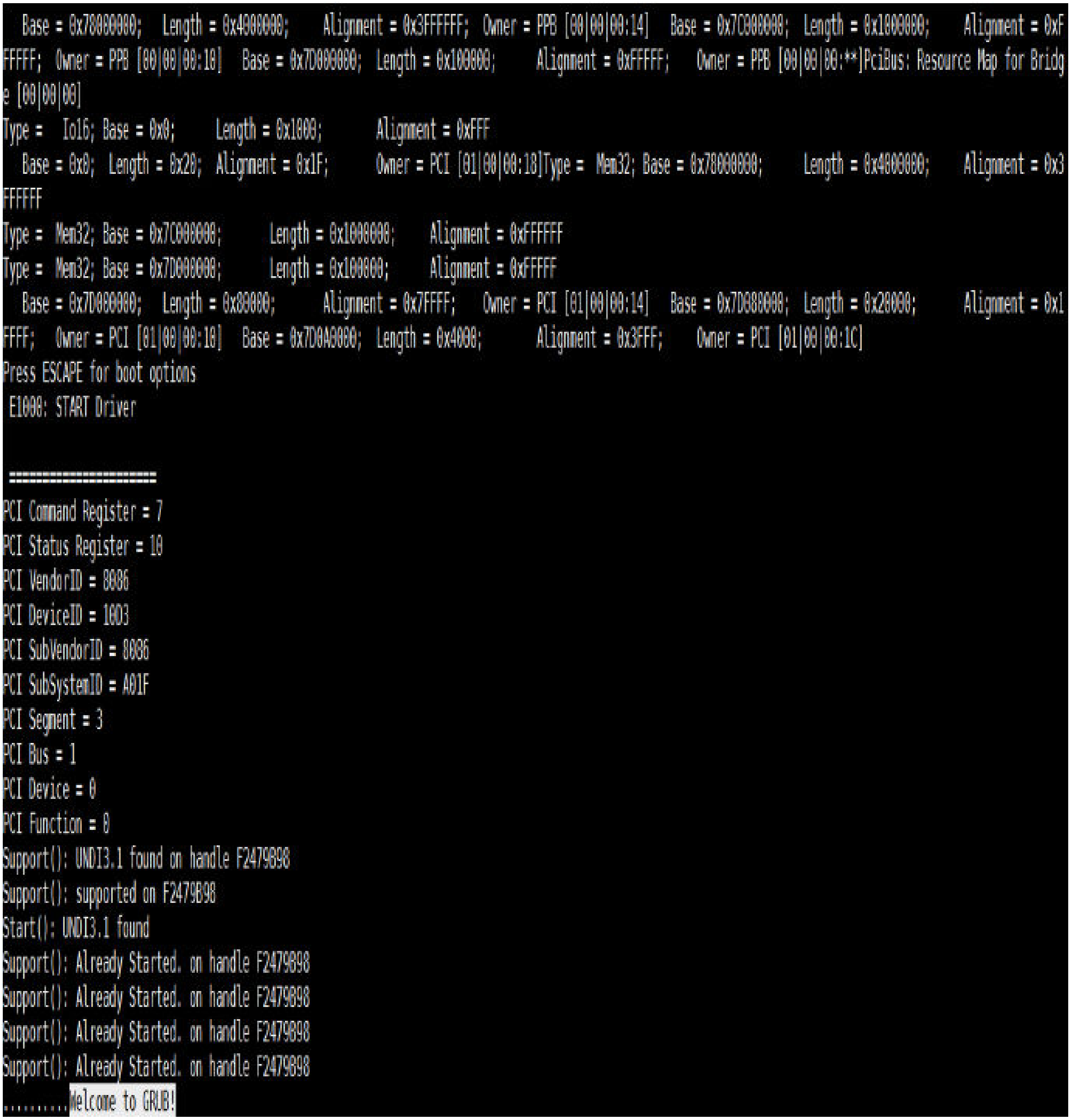
```
=> cpld reset altbank

UEFI firmware (version  built at 14:35:52 on May 29 2017)
Clock Configuration:
       CPU0(A53):1500 MHz  CPU1(A53):1500 MHz  CPU2(A53):1500 MHz
       CPU3(A53):1500 MHz
       Bus:      400  MHz  DDR:      1600 MHz
       FMAN:      500  MHz
Reset Configuration Word (RCW):
       00000000: 0810000F 0C000000 00000000 00000000
       00000010: 14550002 80004012 E0025000 61002000
       00000020: 00000000 00000000 00000000 00038800
       00000030: 00000000 00001101 00000096 00000001
SoC: LS1043AE; Rev 1.0
Board: LS1043A-RDB
vBank 4
CPLD:  V1.5
PCBA:  V3.0
SERDES Reference Clocks:
SD1_CLK1 = 156.25MHZ, SD1_CLK2 = 100.00MHZ
UEFI firmware (version  built at 14:38:10 on May 29 2017)
Detected DSPI Flash N25Q128 With Page Size 256 Bytes,
 Erase Size 64 KiB, Total  16 MiB,
PCIE1 is disabled
PCIE2 is Enabled
PCIe:VendorID: 1957
PCIe Device ID: 8080
PCIe Header Type: 01
PCIe2: Root Complex
PCIe Link Status: 1011
x1 gen1, regs @ 0x3500000
PCIE3 is Enabled
PCIe:VendorID: 1957
PCIe Device ID: 8080
PCIe Header Type: 01
PCIe3: Root Complex
PCIe Link Status: 3011
x1 gen1, regs @ 0x3600000
Valid Chip Addresses :
0x0 0x8 0x40 0x4C 0x52 0x53 0x68 0x69
```



```
Device: FSL_SDXC
Manufacturer ID: 82
OEM: 4A54
Name: NCard
Card: 2
Product revision: 1.9
Product Serial No. : 907BA200
Manufacturing date : 14:2005
Tran Speed: 50000000
Rd Block Len: 512
SD Version 3.0
High Capacity: Yes
Capacity: 14.7 GiB
Bus Width: 4-Bit
DPAA1: Memac MEMAC9 PHY type xfi, Enabled
DPAA1: Memac MEMAC1 PHY type qsgmii, Enabled
DPAA1: Memac MEMAC2 PHY type qsgmii, Enabled
DPAA1: Memac MEMAC5 PHY type qsgmii, Enabled
DPAA1: Memac MEMAC6 PHY type qsgmii, Enabled
DPAA1: Memac MEMAC3 PHY type rgmii, Enabled
DPAA1: Memac MEMAC4 PHY type rgmii, Enabled
DPAA1: FMan1:Loading firmware microcode image version 108.4.10
DPAA1: MAC address MEMAC9:2E:F3:4E:44:00:0A ...
DPAA1 ERROR: PHY auto-negotiation failed
DPAA1 WARNING: Link not ready for MEMAC9 after Timeout. Check Ethernet cable
DPAA1: MAC address MEMAC1:2E:F3:4E:44:00:02 ...
DPAA1 ERROR: PHY auto-negotiation failed
DPAA1 WARNING: Link not ready for MEMAC1 after Timeout. Check Ethernet cable
DPAA1: MAC address MEMAC2:2E:F3:4E:44:00:03 ...
DPAA1 ERROR: PHY auto-negotiation failed
DPAA1 WARNING: Link not ready for MEMAC2 after Timeout. Check Ethernet cable
DPAA1: MAC address MEMAC5:2E:F3:4E:44:00:06 ...
DPAA1: PHY link is up
DPAA1: MAC address MEMAC6:2E:F3:4E:44:00:07 ...
DPAA1 ERROR: PHY auto-negotiation failed
DPAA1 WARNING: Link not ready for MEMAC6 after Timeout. Check Ethernet cable
DPAA1: MAC address MEMAC3:2E:F3:4E:44:00:04 ...
DPAA1: PHY link is up
DPAA1: MAC address MEMAC4:2E:F3:4E:44:00:05 ...
DPAA1 ERROR: PHY auto-negotiation failed
DPAA1 WARNING: Link not ready for MEMAC4 after Timeout. Check Ethernet cable
```



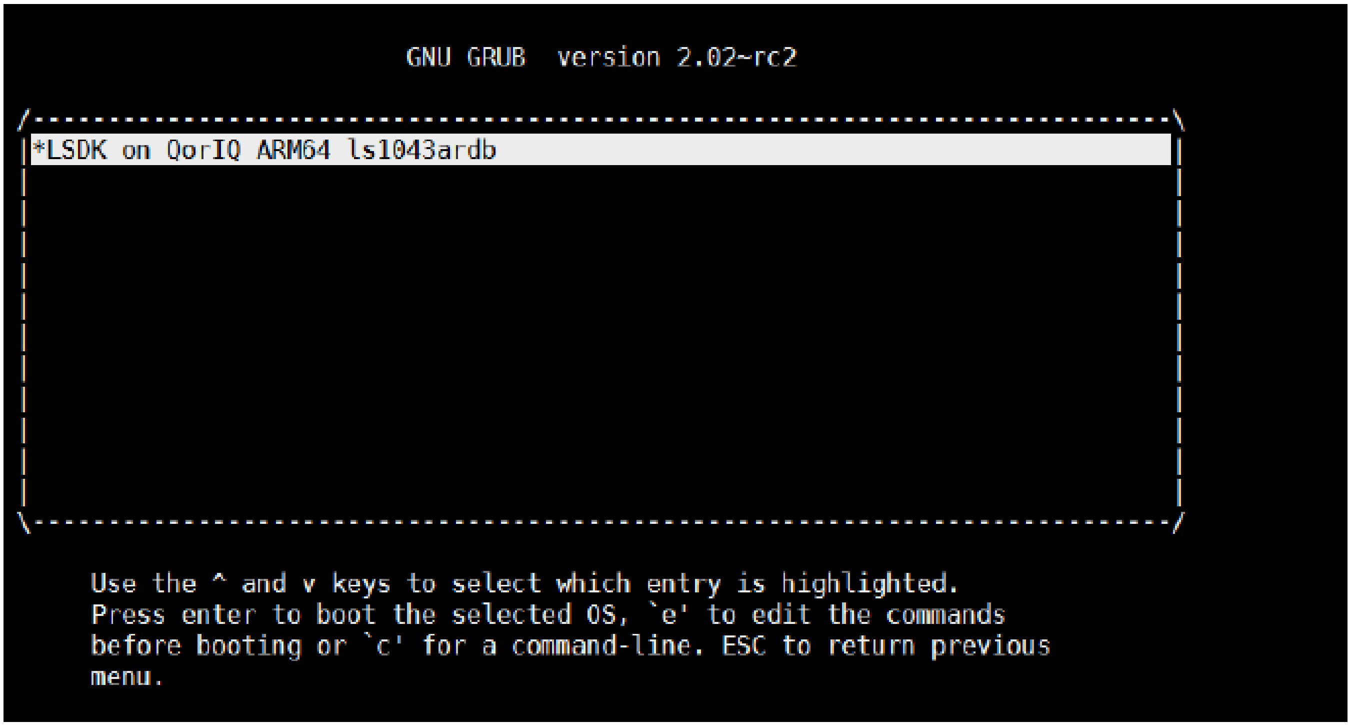
```
PciBus: Discovered PPB @ [00|00|00]
   BAR[0]: Type =  Mem32; Alignment = 0xFFFFFF; Length = 0x1000000;     Offset = 0x10
   BAR[1]: Type =  Mem32; Alignment = 0x3FFFFFF;       Length = 0x4000000;     Offset = 0x14

PciBus: Discovered PPB @ [00|00|00]
   BAR[0]: Type =  Mem32; Alignment = 0xFFFFFF; Length = 0x1000000;     Offset = 0x10
   BAR[1]: Type =  Mem32; Alignment = 0x3FFFFFF;       Length = 0x4000000;     Offset = 0x14

LOAD OPTION ROM, RomBase: 78000000
PciBus: Resource Map for Root Bridge PciRoot(0x1)
Type =  Mem32; Base = 0x78000000;       Length = 0x5000000;     Alignment = 0x3FFFFFF
   Base = 0x78000000;   Length = 0x4000000;     Alignment = 0x3FFFFFF;  Owner = PPB [00|00|00:14]   Base = 0x7C000000;  Length = 0x1000000;     Alignment = 0xF
FFFFF;  Owner = PPB [00|00|00:10]PciBus: Resource Map for Bridge [00|00|00]
Type =  Mem32; Base = 0x78000000;       Length = 0x4000000;     Alignment = 0x3FFFFFF
Type =  Mem32; Base = 0x7C000000;       Length = 0x1000000;     Alignment = 0xFFFFFF
PciBus: Discovered PPB @ [00|00|00]
   BAR[0]: Type =  Mem32; Alignment = 0xFFFFFF; Length = 0x1000000;     Offset = 0x10
   BAR[1]: Type =  Mem32; Alignment = 0x3FFFFFF;       Length = 0x4000000;     Offset = 0x14

PciBus: Discovered PCI @ [01|00|00]
   BAR[0]: Type =  Mem32; Alignment = 0x1FFFF;  Length = 0x20000;      Offset = 0x10
   BAR[1]: Type =  Mem32; Alignment = 0x7FFFF;  Length = 0x80000;      Offset = 0x14
   BAR[2]: Type =   Io32; Alignment = 0x1F;     Length = 0x20;  Offset = 0x18
   BAR[3]: Type =  Mem32; Alignment = 0x3FFF;   Length = 0x4000;       Offset = 0x1C

PciBus: Discovered PPB @ [00|00|00]
   BAR[0]: Type =  Mem32; Alignment = 0xFFFFFF; Length = 0x1000000;     Offset = 0x10
   BAR[1]: Type =  Mem32; Alignment = 0x3FFFFFF;       Length = 0x4000000;     Offset = 0x14

PciBus: Discovered PCI @ [01|00|00]
   BAR[0]: Type =  Mem32; Alignment = 0x1FFFF;  Length = 0x20000;      Offset = 0x10
   BAR[1]: Type =  Mem32; Alignment = 0x7FFFF;  Length = 0x80000;      Offset = 0x14
   BAR[2]: Type =   Io32; Alignment = 0x1F;     Length = 0x20;  Offset = 0x18
   BAR[3]: Type =  Mem32; Alignment = 0x3FFF;   Length = 0x4000;       Offset = 0x1C

LOAD OPTION ROM, RomBase: 78000000
LOAD OPTION ROM, RomBase: 78000000
PciBus: Resource Map for Root Bridge PciRoot(0x2)
```



```
  Base = 0x78000000;  Length = 0x4000000;    Alignment = 0x3FFFFFF;  Owner = PPB [00|00|00:14]   Base = 0x7C000000;  Length = 0x1000000;    Alignment = 0xF
FFFFF;  Owner = PPB [00|00|00:10]   Base = 0x7D000000;  Length = 0x100000;     Alignment = 0xFFFFF;    Owner = PPB [00|00|00:**]PciBus: Resource Map for Bridg
e [00|00|00]
Type =   Io16; Base = 0x0;     Length = 0x1000;      Alignment = 0xFFF
  Base = 0x0;  Length = 0x20;  Alignment = 0x1F;      Owner = PCI [01|00|00:18]Type =  Mem32; Base = 0x78000000;     Length = 0x4000000;    Alignment = 0x3
FFFFFF
Type =  Mem32; Base = 0x7C000000;      Length = 0x1000000;    Alignment = 0xFFFFFF
Type =  Mem32; Base = 0x7D000000;      Length = 0x100000;     Alignment = 0xFFFFF
  Base = 0x7D000000;  Length = 0x80000;      Alignment = 0x7FFFF;    Owner = PCI [01|00|00:14]   Base = 0x7D080000;  Length = 0x20000;      Alignment = 0x1
FFFF;   Owner = PCI [01|00|00:10]   Base = 0x7D0A0000;  Length = 0x4000;       Alignment = 0x3FFF;     Owner = PCI [01|00|00:1C]
Press ESCAPE for boot options
 E1000: START Driver


 ====================
PCI Command Register = 7
PCI Status Register = 10
PCI VendorID = 8086
PCI DeviceID = 10D3
PCI SubVendorID = 8086
PCI SubSystemID = A01F
PCI Segment = 3
PCI Bus = 1
PCI Device = 0
PCI Function = 0
Support(): UNDI3.1 found on handle F2479B98
Support(): supported on F2479B98
Start(): UNDI3.1 found
Support(): Already Started. on handle F2479B98
Support(): Already Started. on handle F2479B98
Support(): Already Started. on handle F2479B98
Support(): Already Started. on handle F2479B98
..........Welcome to GRUB!
```

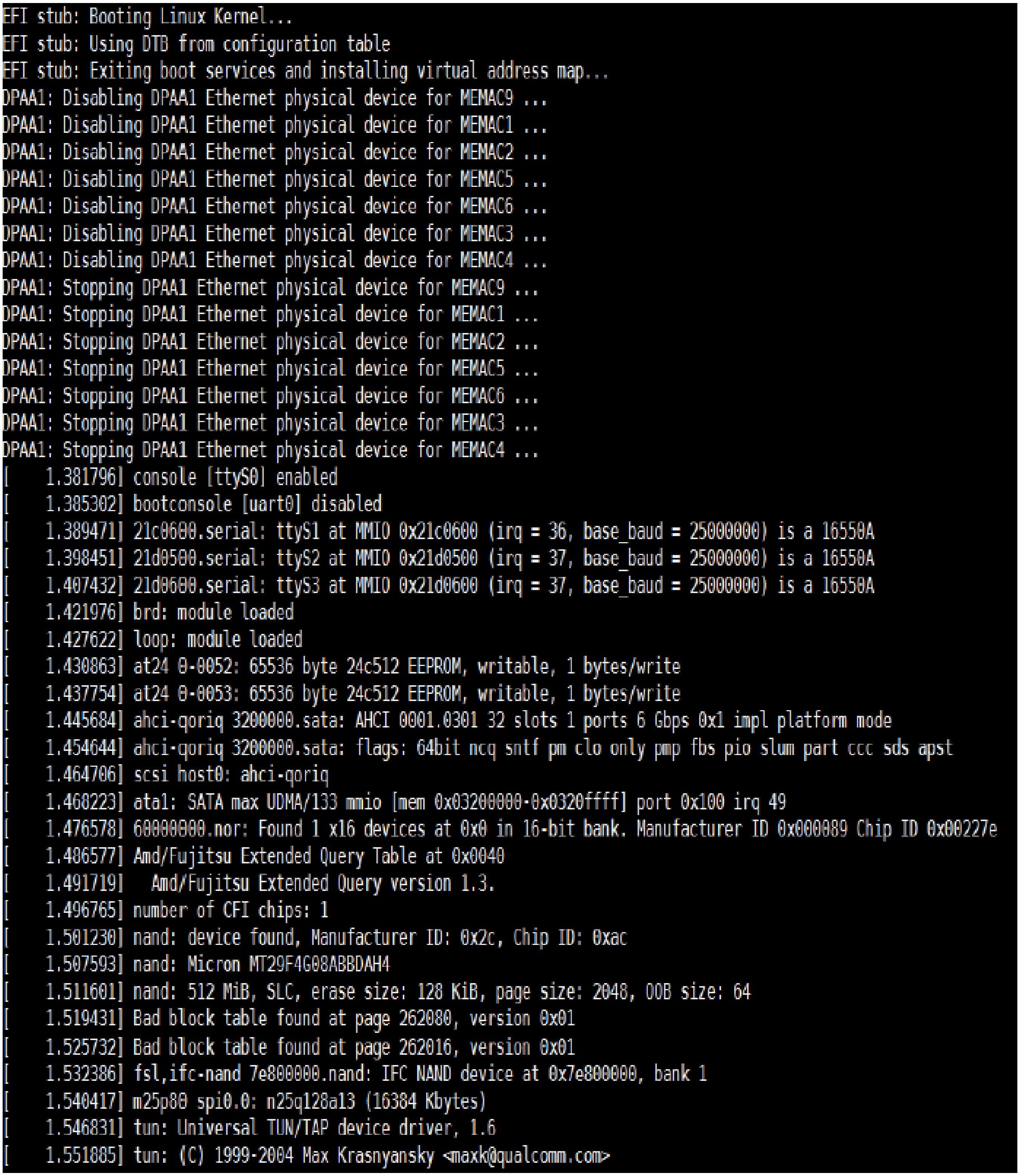
NOTE

By default UEFI Boot Manager loads GRUB bootloader placed in /EFI/BOOT/ directory of EFI partitions. Press ESC for launching splash bios screen:

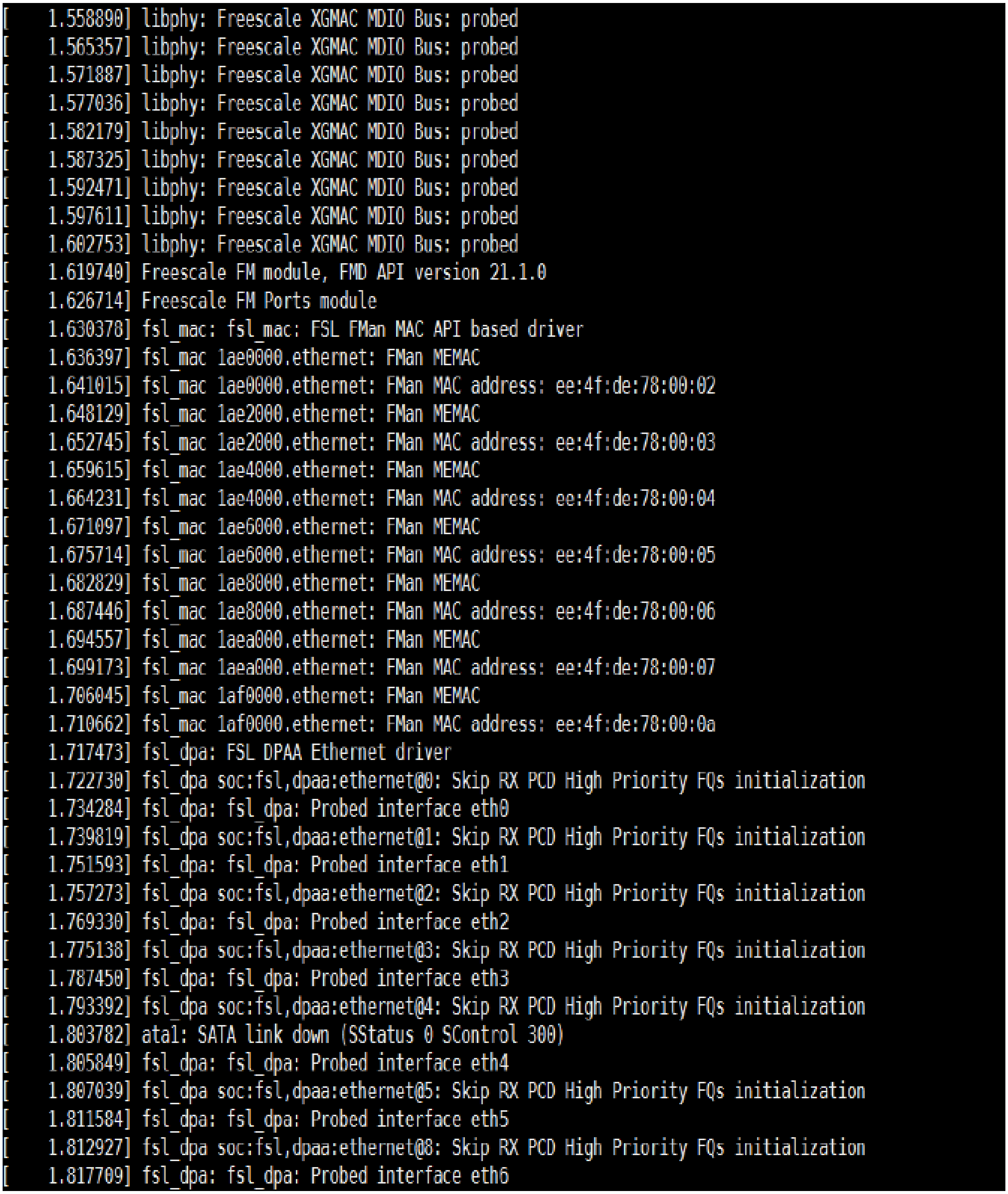


```
                        GNU GRUB  version 2.02~rc2

/----------------------------------------------------------------------------\
|*LSDK on QorIQ ARM64 ls1043ardb                                             |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
\----------------------------------------------------------------------------/

     Use the ^ and v keys to select which entry is highlighted.
     Press enter to boot the selected OS, `e' to edit the commands
     before booting or `c' for a command-line. ESC to return previous
     menu.
```

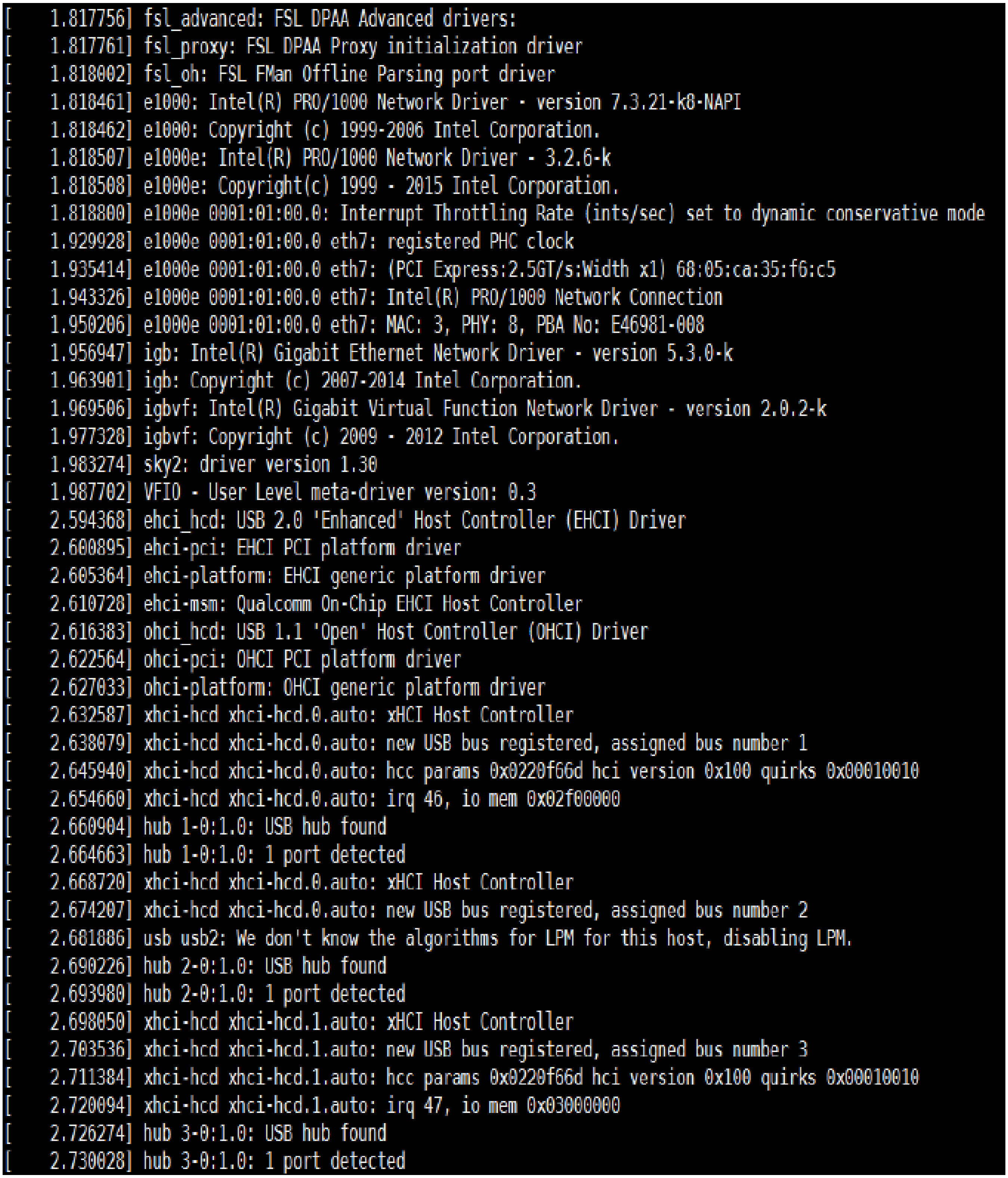
Press Enter to boot LSDK distro.



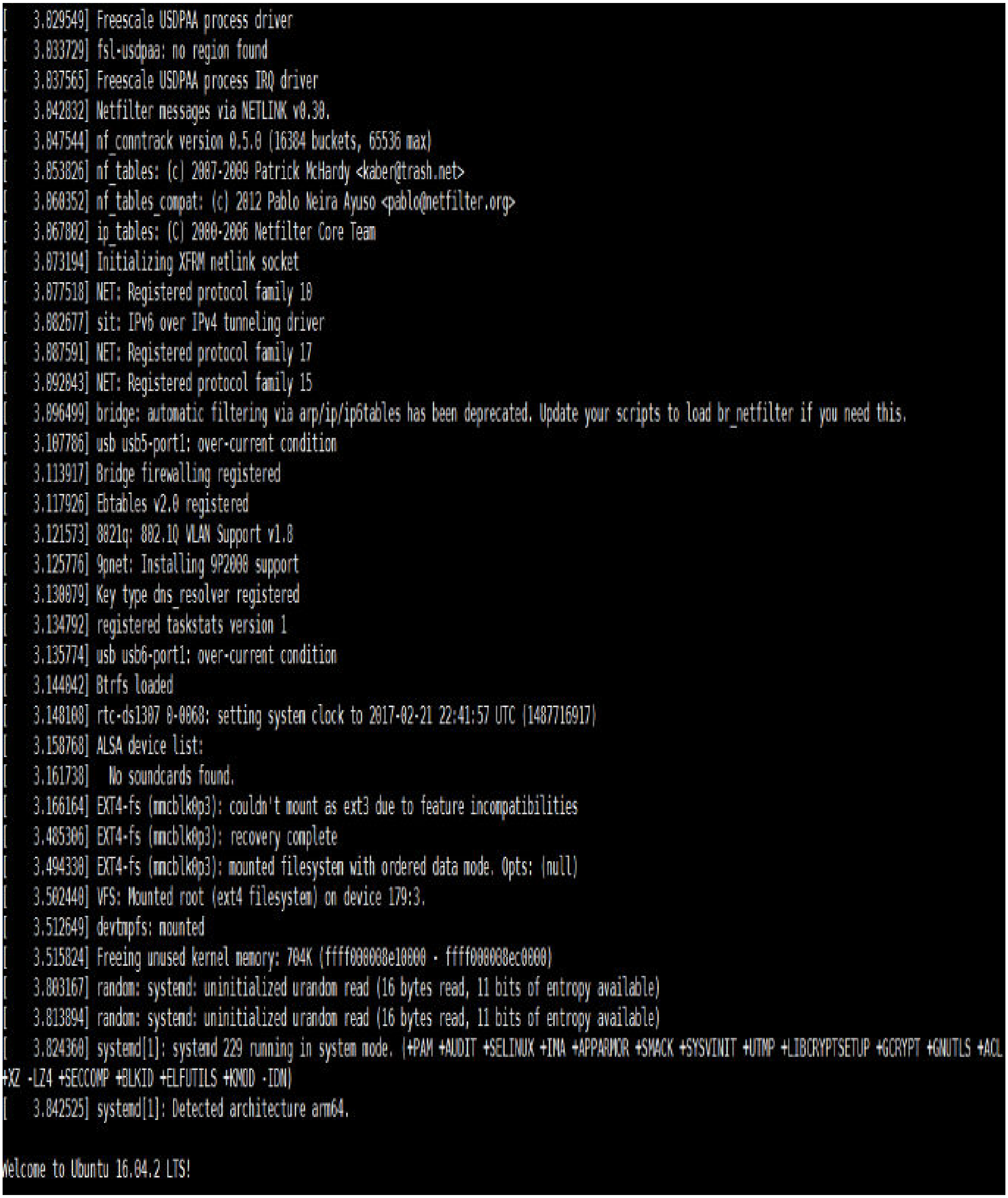
```
EFI stub: Booting Linux Kernel...
EFI stub: Using DTB from configuration table
EFI stub: Exiting boot services and installing virtual address map...
DPAA1: Disabling DPAA1 Ethernet physical device for MEMAC9 ...
DPAA1: Disabling DPAA1 Ethernet physical device for MEMAC1 ...
DPAA1: Disabling DPAA1 Ethernet physical device for MEMAC2 ...
DPAA1: Disabling DPAA1 Ethernet physical device for MEMAC5 ...
DPAA1: Disabling DPAA1 Ethernet physical device for MEMAC6 ...
DPAA1: Disabling DPAA1 Ethernet physical device for MEMAC3 ...
DPAA1: Disabling DPAA1 Ethernet physical device for MEMAC4 ...
DPAA1: Stopping DPAA1 Ethernet physical device for MEMAC9 ...
DPAA1: Stopping DPAA1 Ethernet physical device for MEMAC1 ...
DPAA1: Stopping DPAA1 Ethernet physical device for MEMAC2 ...
DPAA1: Stopping DPAA1 Ethernet physical device for MEMAC5 ...
DPAA1: Stopping DPAA1 Ethernet physical device for MEMAC6 ...
DPAA1: Stopping DPAA1 Ethernet physical device for MEMAC3 ...
DPAA1: Stopping DPAA1 Ethernet physical device for MEMAC4 ...
[    1.381796] console [ttyS0] enabled
[    1.385302] bootconsole [uart0] disabled
[    1.389471] 21c0600.serial: ttyS1 at MMIO 0x21c0600 (irq = 36, base_baud = 25000000) is a 16550A
[    1.398451] 21d0500.serial: ttyS2 at MMIO 0x21d0500 (irq = 37, base_baud = 25000000) is a 16550A
[    1.407432] 21d0600.serial: ttyS3 at MMIO 0x21d0600 (irq = 37, base_baud = 25000000) is a 16550A
[    1.421976] brd: module loaded
[    1.427622] loop: module loaded
[    1.430863] at24 0-0052: 65536 byte 24c512 EEPROM, writable, 1 bytes/write
[    1.437754] at24 0-0053: 65536 byte 24c512 EEPROM, writable, 1 bytes/write
[    1.445684] ahci-qoriq 3200000.sata: AHCI 0001.0301 32 slots 1 ports 6 Gbps 0x1 impl platform mode
[    1.454644] ahci-qoriq 3200000.sata: flags: 64bit ncq sntf pm clo only pmp fbs pio slum part ccc sds apst
[    1.464706] scsi host0: ahci-qoriq
[    1.468223] ata1: SATA max UDMA/133 mmio [mem 0x03200000-0x0320ffff] port 0x100 irq 49
[    1.476578] 60000000.nor: Found 1 x16 devices at 0x0 in 16-bit bank. Manufacturer ID 0x000089 Chip ID 0x00227e
[    1.486577] Amd/Fujitsu Extended Query Table at 0x0040
[    1.491719]   Amd/Fujitsu Extended Query version 1.3.
[    1.496765] number of CFI chips: 1
[    1.501230] nand: device found, Manufacturer ID: 0x2c, Chip ID: 0xac
[    1.507593] nand: Micron MT29F4G08ABBDAH4
[    1.511601] nand: 512 MiB, SLC, erase size: 128 KiB, page size: 2048, OOB size: 64
[    1.519431] Bad block table found at page 262080, version 0x01
[    1.525732] Bad block table found at page 262016, version 0x01
[    1.532386] fsl,ifc-nand 7e800000.nand: IFC NAND device at 0x7e800000, bank 1
[    1.540417] m25p80 spi0.0: n25q128a13 (16384 Kbytes)
[    1.546831] tun: Universal TUN/TAP device driver, 1.6
[    1.551885] tun: (C) 1999-2004 Max Krasnyansky <maxk@qualcomm.com>
```



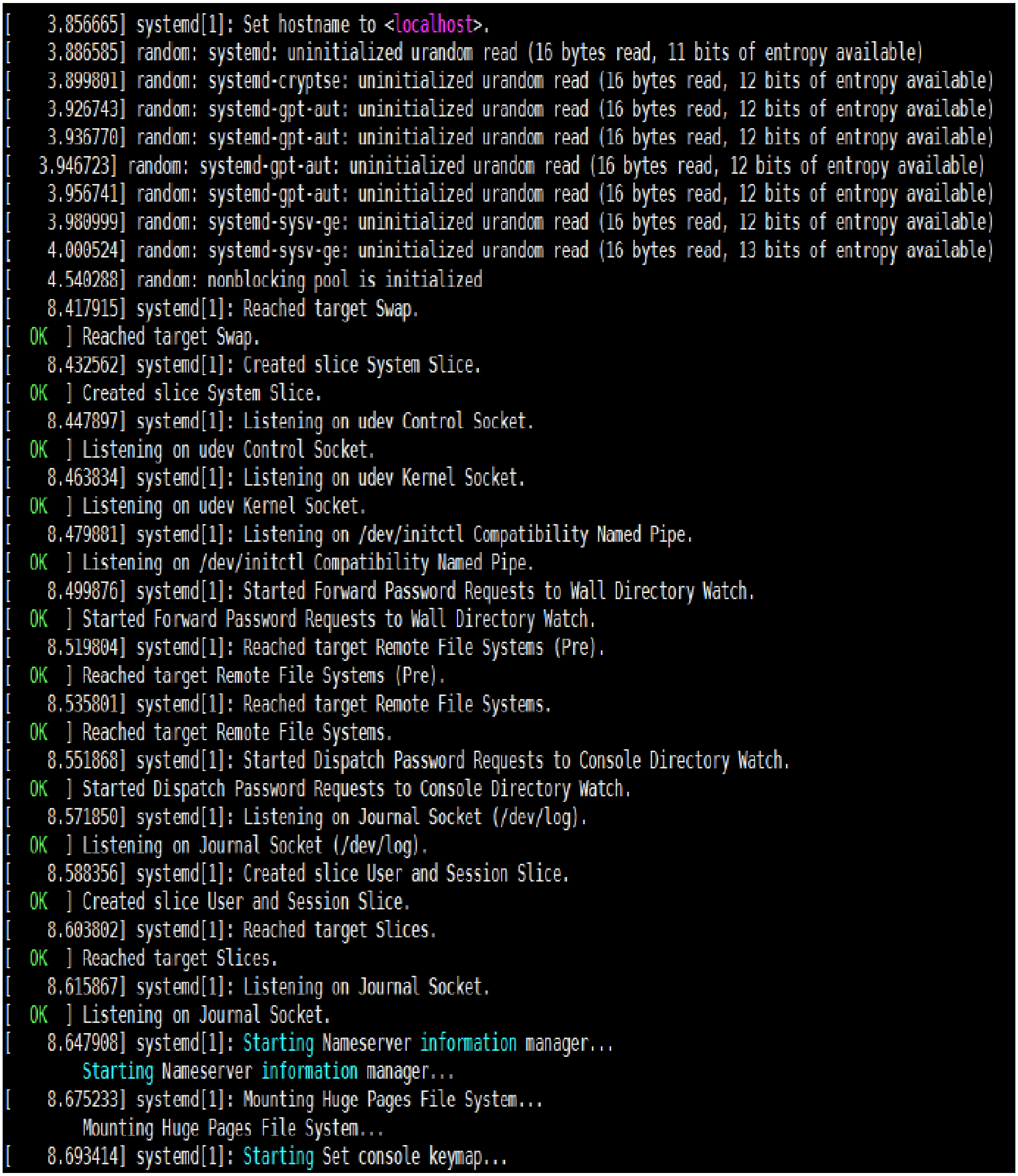
```
[    1.558890] libphy: Freescale XGMAC MDIO Bus: probed
[    1.565357] libphy: Freescale XGMAC MDIO Bus: probed
[    1.571887] libphy: Freescale XGMAC MDIO Bus: probed
[    1.577036] libphy: Freescale XGMAC MDIO Bus: probed
[    1.582179] libphy: Freescale XGMAC MDIO Bus: probed
[    1.587325] libphy: Freescale XGMAC MDIO Bus: probed
[    1.592471] libphy: Freescale XGMAC MDIO Bus: probed
[    1.597611] libphy: Freescale XGMAC MDIO Bus: probed
[    1.602753] libphy: Freescale XGMAC MDIO Bus: probed
[    1.619740] Freescale FM module, FMD API version 21.1.0
[    1.626714] Freescale FM Ports module
[    1.630378] fsl_mac: fsl_mac: FSL FMan MAC API based driver
[    1.636397] fsl_mac 1ae0000.ethernet: FMan MEMAC
[    1.641015] fsl_mac 1ae0000.ethernet: FMan MAC address: ee:4f:de:78:00:02
[    1.648129] fsl_mac 1ae2000.ethernet: FMan MEMAC
[    1.652745] fsl_mac 1ae2000.ethernet: FMan MAC address: ee:4f:de:78:00:03
[    1.659615] fsl_mac 1ae4000.ethernet: FMan MEMAC
[    1.664231] fsl_mac 1ae4000.ethernet: FMan MAC address: ee:4f:de:78:00:04
[    1.671097] fsl_mac 1ae6000.ethernet: FMan MEMAC
[    1.675714] fsl_mac 1ae6000.ethernet: FMan MAC address: ee:4f:de:78:00:05
[    1.682829] fsl_mac 1ae8000.ethernet: FMan MEMAC
[    1.687446] fsl_mac 1ae8000.ethernet: FMan MAC address: ee:4f:de:78:00:06
[    1.694557] fsl_mac 1aea000.ethernet: FMan MEMAC
[    1.699173] fsl_mac 1aea000.ethernet: FMan MAC address: ee:4f:de:78:00:07
[    1.706045] fsl_mac 1af0000.ethernet: FMan MEMAC
[    1.710662] fsl_mac 1af0000.ethernet: FMan MAC address: ee:4f:de:78:00:0a
[    1.717473] fsl_dpa: FSL DPAA Ethernet driver
[    1.722730] fsl_dpa soc:fsl,dpaa:ethernet@0: Skip RX PCD High Priority FQs initialization
[    1.734284] fsl_dpa: fsl_dpa: Probed interface eth0
[    1.739819] fsl_dpa soc:fsl,dpaa:ethernet@1: Skip RX PCD High Priority FQs initialization
[    1.751593] fsl_dpa: fsl_dpa: Probed interface eth1
[    1.757273] fsl_dpa soc:fsl,dpaa:ethernet@2: Skip RX PCD High Priority FQs initialization
[    1.769330] fsl_dpa: fsl_dpa: Probed interface eth2
[    1.775138] fsl_dpa soc:fsl,dpaa:ethernet@3: Skip RX PCD High Priority FQs initialization
[    1.787450] fsl_dpa: fsl_dpa: Probed interface eth3
[    1.793392] fsl_dpa soc:fsl,dpaa:ethernet@4: Skip RX PCD High Priority FQs initialization
[    1.803782] ata1: SATA link down (SStatus 0 SControl 300)
[    1.805849] fsl_dpa: fsl_dpa: Probed interface eth4
[    1.807039] fsl_dpa soc:fsl,dpaa:ethernet@5: Skip RX PCD High Priority FQs initialization
[    1.811584] fsl_dpa: fsl_dpa: Probed interface eth5
[    1.812927] fsl_dpa soc:fsl,dpaa:ethernet@8: Skip RX PCD High Priority FQs initialization
[    1.817709] fsl_dpa: fsl_dpa: Probed interface eth6
```



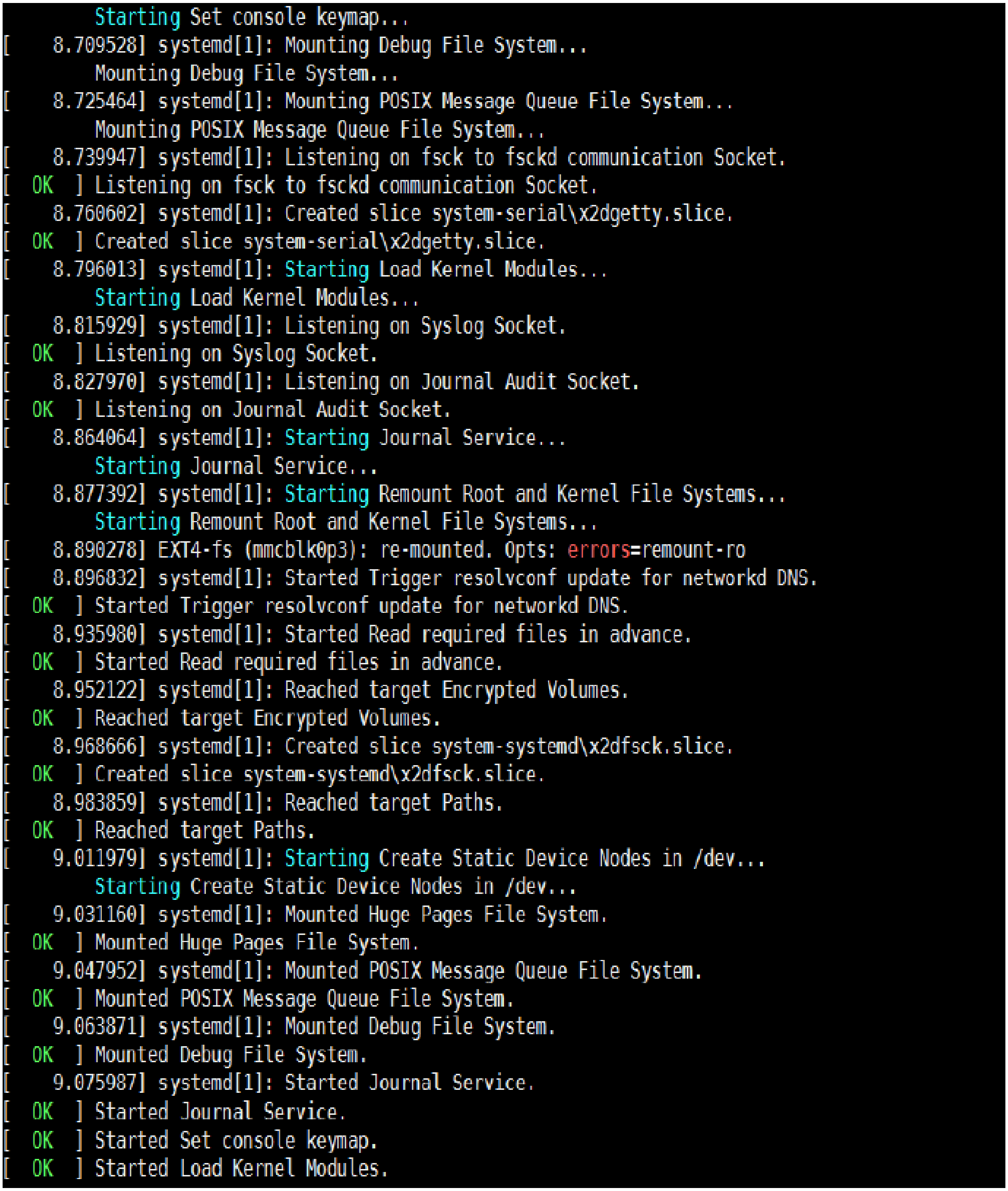
```
[    1.817756] fsl_advanced: FSL DPAA Advanced drivers:
[    1.817761] fsl_proxy: FSL DPAA Proxy initialization driver
[    1.818002] fsl_oh: FSL FMan Offline Parsing port driver
[    1.818461] e1000: Intel(R) PRO/1000 Network Driver - version 7.3.21-k8-NAPI
[    1.818462] e1000: Copyright (c) 1999-2006 Intel Corporation.
[    1.818507] e1000e: Intel(R) PRO/1000 Network Driver - 3.2.6-k
[    1.818508] e1000e: Copyright(c) 1999 - 2015 Intel Corporation.
[    1.818800] e1000e 0001:01:00.0: Interrupt Throttling Rate (ints/sec) set to dynamic conservative mode
[    1.929928] e1000e 0001:01:00.0 eth7: registered PHC clock
[    1.935414] e1000e 0001:01:00.0 eth7: (PCI Express:2.5GT/s:Width x1) 68:05:ca:35:f6:c5
[    1.943326] e1000e 0001:01:00.0 eth7: Intel(R) PRO/1000 Network Connection
[    1.950206] e1000e 0001:01:00.0 eth7: MAC: 3, PHY: 8, PBA No: E46981-008
[    1.956947] igb: Intel(R) Gigabit Ethernet Network Driver - version 5.3.0-k
[    1.963901] igb: Copyright (c) 2007-2014 Intel Corporation.
[    1.969506] igbvf: Intel(R) Gigabit Virtual Function Network Driver - version 2.0.2-k
[    1.977328] igbvf: Copyright (c) 2009 - 2012 Intel Corporation.
[    1.983274] sky2: driver version 1.30
[    1.987702] VFIO - User Level meta-driver version: 0.3
[    2.594368] ehci_hcd: USB 2.0 'Enhanced' Host Controller (EHCI) Driver
[    2.600895] ehci-pci: EHCI PCI platform driver
[    2.605364] ehci-platform: EHCI generic platform driver
[    2.610728] ehci-msm: Qualcomm On-Chip EHCI Host Controller
[    2.616383] ohci_hcd: USB 1.1 'Open' Host Controller (OHCI) Driver
[    2.622564] ohci-pci: OHCI PCI platform driver
[    2.627033] ohci-platform: OHCI generic platform driver
[    2.632587] xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
[    2.638079] xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 1
[    2.645940] xhci-hcd xhci-hcd.0.auto: hcc params 0x0220f66d hci version 0x100 quirks 0x00010010
[    2.654660] xhci-hcd xhci-hcd.0.auto: irq 46, io mem 0x02f00000
[    2.660904] hub 1-0:1.0: USB hub found
[    2.664663] hub 1-0:1.0: 1 port detected
[    2.668720] xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
[    2.674207] xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 2
[    2.681886] usb usb2: We don't know the algorithms for LPM for this host, disabling LPM.
[    2.690226] hub 2-0:1.0: USB hub found
[    2.693980] hub 2-0:1.0: 1 port detected
[    2.698050] xhci-hcd xhci-hcd.1.auto: xHCI Host Controller
[    2.703536] xhci-hcd xhci-hcd.1.auto: new USB bus registered, assigned bus number 3
[    2.711384] xhci-hcd xhci-hcd.1.auto: hcc params 0x0220f66d hci version 0x100 quirks 0x00010010
[    2.720094] xhci-hcd xhci-hcd.1.auto: irq 47, io mem 0x03000000
[    2.726274] hub 3-0:1.0: USB hub found
[    2.730028] hub 3-0:1.0: 1 port detected
```



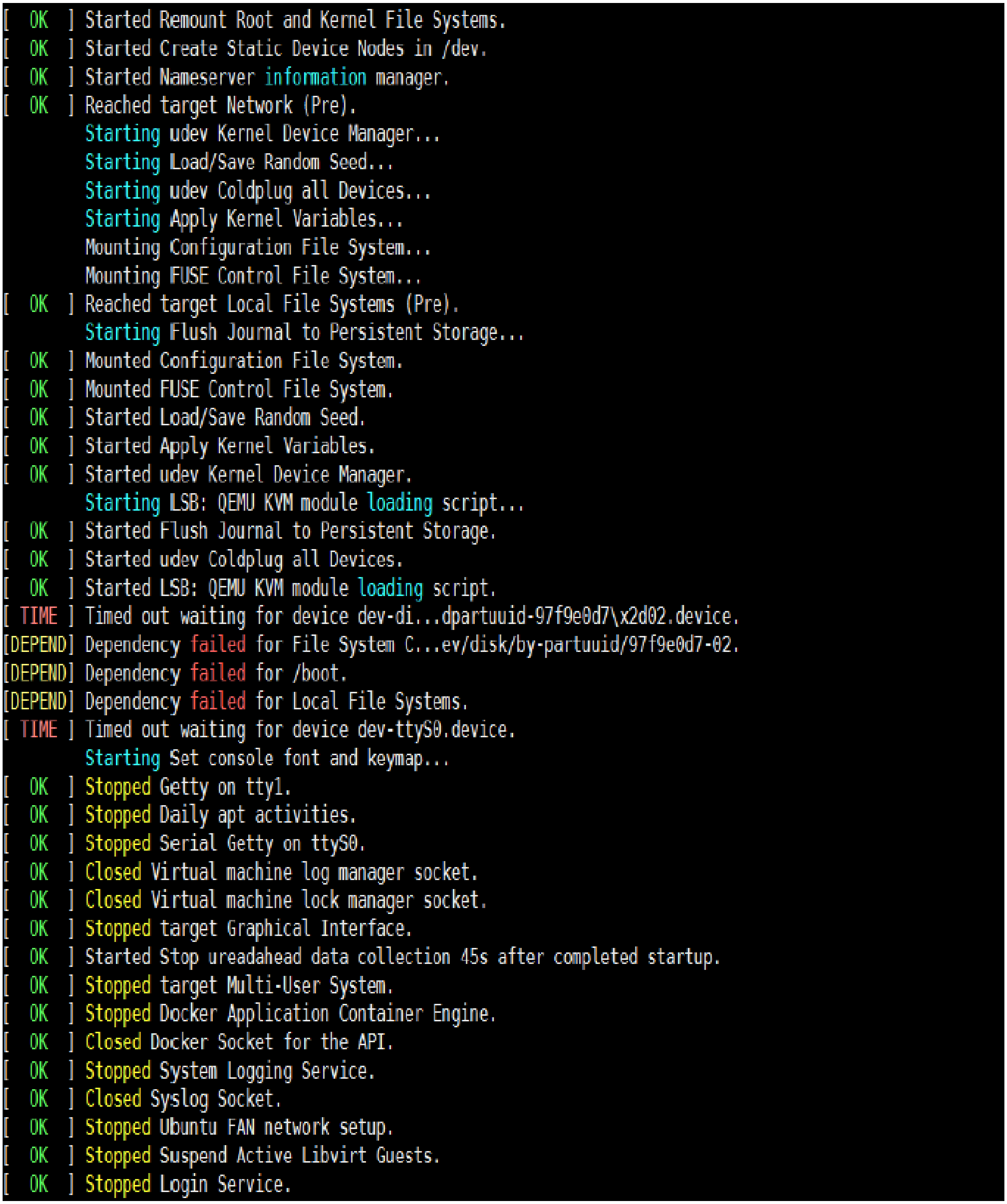
```
[    3.829549] Freescale USDPAA process driver
[    3.833729] fsl-usdpaa: no region found
[    3.837565] Freescale USDPAA process IRQ driver
[    3.842832] Netfilter messages via NETLINK v0.30.
[    3.847544] nf_conntrack version 0.5.0 (16384 buckets, 65536 max)
[    3.853826] nf_tables: (c) 2007-2009 Patrick McHardy <kaber@trash.net>
[    3.860352] nf_tables_compat: (c) 2012 Pablo Neira Ayuso <pablo@netfilter.org>
[    3.867802] ip_tables: (C) 2000-2006 Netfilter Core Team
[    3.873194] Initializing XFRM netlink socket
[    3.877518] NET: Registered protocol family 10
[    3.882677] sit: IPv6 over IPv4 tunneling driver
[    3.887591] NET: Registered protocol family 17
[    3.892843] NET: Registered protocol family 15
[    3.896499] bridge: automatic filtering via arp/ip/ip6tables has been deprecated. Update your scripts to load br_netfilter if you need this.
[    3.107786] usb usb5-port1: over-current condition
[    3.113917] Bridge firewalling registered
[    3.117926] Ebtables v2.0 registered
[    3.121573] 8021q: 802.1Q VLAN Support v1.8
[    3.125776] 9pnet: Installing 9P2000 support
[    3.130079] Key type dns_resolver registered
[    3.134792] registered taskstats version 1
[    3.135774] usb usb6-port1: over-current condition
[    3.144042] Btrfs loaded
[    3.148108] rtc-ds1307 0-0068: setting system clock to 2017-02-21 22:41:57 UTC (1487716917)
[    3.158768] ALSA device list:
[    3.161738]   No soundcards found.
[    3.166164] EXT4-fs (mmcblk0p3): couldn't mount as ext3 due to feature incompatibilities
[    3.485306] EXT4-fs (mmcblk0p3): recovery complete
[    3.494338] EXT4-fs (mmcblk0p3): mounted filesystem with ordered data mode. Opts: (null)
[    3.502440] VFS: Mounted root (ext4 filesystem) on device 179:3.
[    3.512649] devtmpfs: mounted
[    3.515824] Freeing unused kernel memory: 704K (ffff000008e10000 - ffff000008ec0000)
[    3.803167] random: systemd: uninitialized urandom read (16 bytes read, 11 bits of entropy available)
[    3.813894] random: systemd: uninitialized urandom read (16 bytes read, 11 bits of entropy available)
[    3.824360] systemd[1]: systemd 229 running in system mode. (+PAM +AUDIT +SELINUX +IMA +APPARMOR +SMACK +SYSVINIT +UTMP +LIBCRYPTSETUP +GCRYPT +GNUTLS +ACL
+XZ -LZ4 +SECCOMP +BLKID +ELFUTILS +KMOD -IDN)
[    3.842525] systemd[1]: Detected architecture arm64.


Welcome to Ubuntu 16.04.2 LTS!
```



```
[    3.856665] systemd[1]: Set hostname to <localhost>.
[    3.886585] random: systemd: uninitialized urandom read (16 bytes read, 11 bits of entropy available)
[    3.899801] random: systemd-cryptse: uninitialized urandom read (16 bytes read, 12 bits of entropy available)
[    3.926743] random: systemd-gpt-aut: uninitialized urandom read (16 bytes read, 12 bits of entropy available)
[    3.936770] random: systemd-gpt-aut: uninitialized urandom read (16 bytes read, 12 bits of entropy available)
[    3.946723] random: systemd-gpt-aut: uninitialized urandom read (16 bytes read, 12 bits of entropy available)
[    3.956741] random: systemd-gpt-aut: uninitialized urandom read (16 bytes read, 12 bits of entropy available)
[    3.980999] random: systemd-sysv-ge: uninitialized urandom read (16 bytes read, 12 bits of entropy available)
[    4.000524] random: systemd-sysv-ge: uninitialized urandom read (16 bytes read, 13 bits of entropy available)
[    4.540288] random: nonblocking pool is initialized
[    8.417915] systemd[1]: Reached target Swap.
[  OK  ] Reached target Swap.
[    8.432562] systemd[1]: Created slice System Slice.
[  OK  ] Created slice System Slice.
[    8.447897] systemd[1]: Listening on udev Control Socket.
[  OK  ] Listening on udev Control Socket.
[    8.463834] systemd[1]: Listening on udev Kernel Socket.
[  OK  ] Listening on udev Kernel Socket.
[    8.479881] systemd[1]: Listening on /dev/initctl Compatibility Named Pipe.
[  OK  ] Listening on /dev/initctl Compatibility Named Pipe.
[    8.499876] systemd[1]: Started Forward Password Requests to Wall Directory Watch.
[  OK  ] Started Forward Password Requests to Wall Directory Watch.
[    8.519804] systemd[1]: Reached target Remote File Systems (Pre).
[  OK  ] Reached target Remote File Systems (Pre).
[    8.535801] systemd[1]: Reached target Remote File Systems.
[  OK  ] Reached target Remote File Systems.
[    8.551868] systemd[1]: Started Dispatch Password Requests to Console Directory Watch.
[  OK  ] Started Dispatch Password Requests to Console Directory Watch.
[    8.571850] systemd[1]: Listening on Journal Socket (/dev/log).
[  OK  ] Listening on Journal Socket (/dev/log).
[    8.588356] systemd[1]: Created slice User and Session Slice.
[  OK  ] Created slice User and Session Slice.
[    8.603802] systemd[1]: Reached target Slices.
[  OK  ] Reached target Slices.
[    8.615867] systemd[1]: Listening on Journal Socket.
[  OK  ] Listening on Journal Socket.
[    8.647908] systemd[1]: Starting Nameserver information manager...
         Starting Nameserver information manager...
[    8.675233] systemd[1]: Mounting Huge Pages File System...
         Mounting Huge Pages File System...
[    8.693414] systemd[1]: Starting Set console keymap...
```

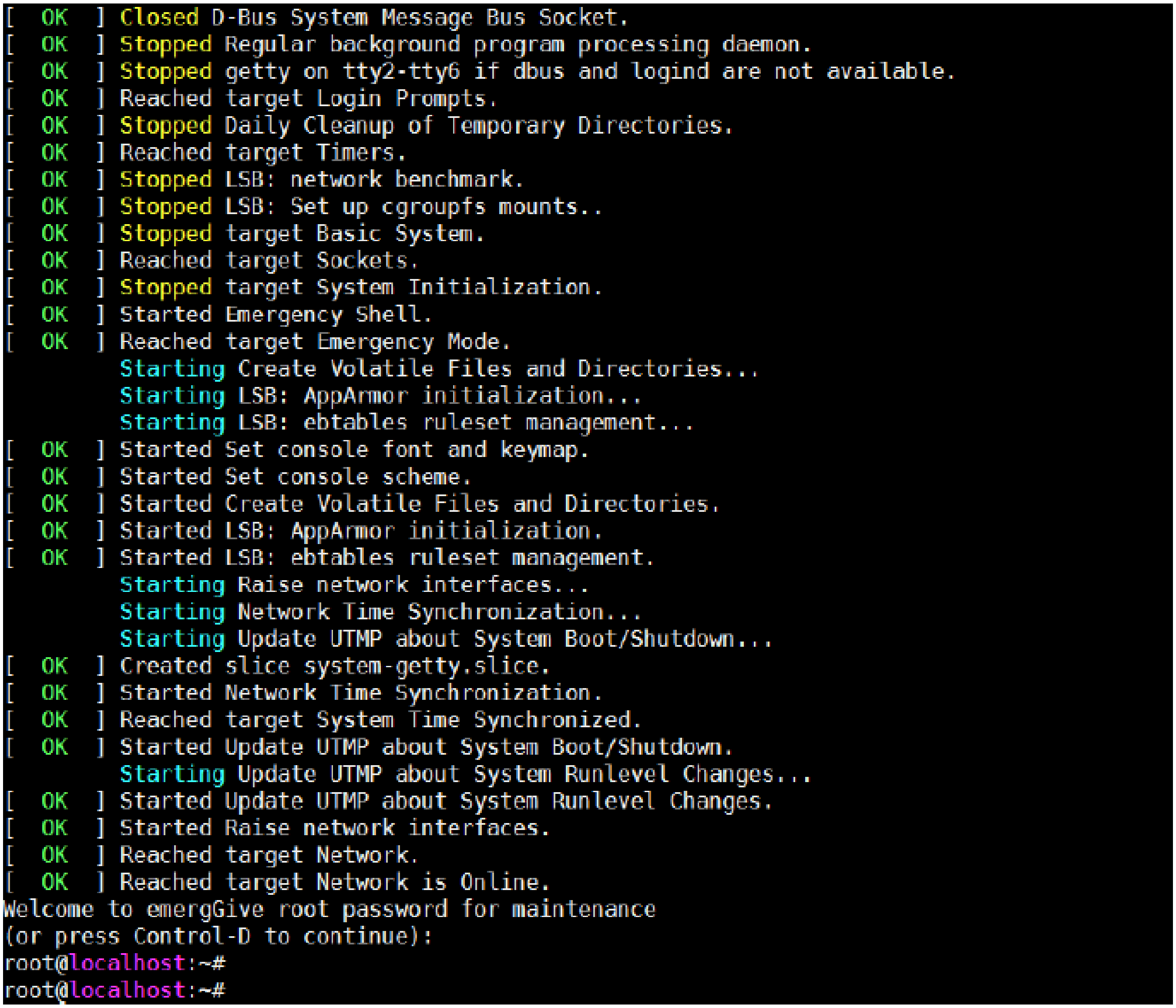


```
         Starting Set console keymap...
[    8.709528] systemd[1]: Mounting Debug File System...
         Mounting Debug File System...
[    8.725464] systemd[1]: Mounting POSIX Message Queue File System...
         Mounting POSIX Message Queue File System...
[    8.739947] systemd[1]: Listening on fsck to fsckd communication Socket.
[  OK  ] Listening on fsck to fsckd communication Socket.
[    8.760602] systemd[1]: Created slice system-serial\x2dgetty.slice.
[  OK  ] Created slice system-serial\x2dgetty.slice.
[    8.796013] systemd[1]: Starting Load Kernel Modules...
         Starting Load Kernel Modules...
[    8.815929] systemd[1]: Listening on Syslog Socket.
[  OK  ] Listening on Syslog Socket.
[    8.827970] systemd[1]: Listening on Journal Audit Socket.
[  OK  ] Listening on Journal Audit Socket.
[    8.864064] systemd[1]: Starting Journal Service...
         Starting Journal Service...
[    8.877392] systemd[1]: Starting Remount Root and Kernel File Systems...
         Starting Remount Root and Kernel File Systems...
[    8.890278] EXT4-fs (mmcblk0p3): re-mounted. Opts: errors=remount-ro
[    8.896832] systemd[1]: Started Trigger resolvconf update for networkd DNS.
[  OK  ] Started Trigger resolvconf update for networkd DNS.
[    8.935980] systemd[1]: Started Read required files in advance.
[  OK  ] Started Read required files in advance.
[    8.952122] systemd[1]: Reached target Encrypted Volumes.
[  OK  ] Reached target Encrypted Volumes.
[    8.968666] systemd[1]: Created slice system-systemd\x2dfsck.slice.
[  OK  ] Created slice system-systemd\x2dfsck.slice.
[    8.983859] systemd[1]: Reached target Paths.
[  OK  ] Reached target Paths.
[    9.011979] systemd[1]: Starting Create Static Device Nodes in /dev...
         Starting Create Static Device Nodes in /dev...
[    9.031160] systemd[1]: Mounted Huge Pages File System.
[  OK  ] Mounted Huge Pages File System.
[    9.047952] systemd[1]: Mounted POSIX Message Queue File System.
[  OK  ] Mounted POSIX Message Queue File System.
[    9.063871] systemd[1]: Mounted Debug File System.
[  OK  ] Mounted Debug File System.
[    9.075987] systemd[1]: Started Journal Service.
[  OK  ] Started Journal Service.
[  OK  ] Started Set console keymap.
[  OK  ] Started Load Kernel Modules.
```

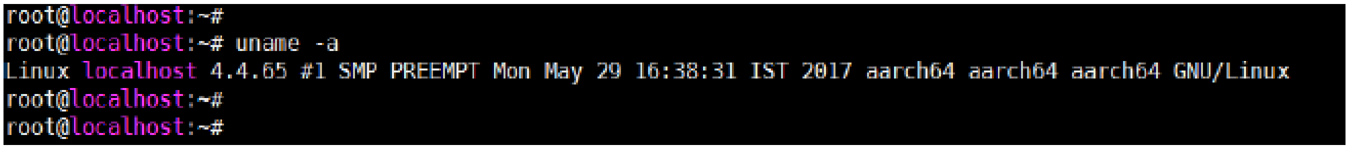


```
[  OK  ] Started Remount Root and Kernel File Systems.
[  OK  ] Started Create Static Device Nodes in /dev.
[  OK  ] Started Nameserver information manager.
[  OK  ] Reached target Network (Pre).
         Starting udev Kernel Device Manager...
         Starting Load/Save Random Seed...
         Starting udev Coldplug all Devices...
         Starting Apply Kernel Variables...
         Mounting Configuration File System...
         Mounting FUSE Control File System...
[  OK  ] Reached target Local File Systems (Pre).
         Starting Flush Journal to Persistent Storage...
[  OK  ] Mounted Configuration File System.
[  OK  ] Mounted FUSE Control File System.
[  OK  ] Started Load/Save Random Seed.
[  OK  ] Started Apply Kernel Variables.
[  OK  ] Started udev Kernel Device Manager.
         Starting LSB: QEMU KVM module loading script...
[  OK  ] Started Flush Journal to Persistent Storage.
[  OK  ] Started udev Coldplug all Devices.
[  OK  ] Started LSB: QEMU KVM module loading script.
[ TIME ] Timed out waiting for device dev-di...dpartuuid-97f9e0d7\x2d02.device.
[DEPEND] Dependency failed for File System C...ev/disk/by-partuuid/97f9e0d7-02.
[DEPEND] Dependency failed for /boot.
[DEPEND] Dependency failed for Local File Systems.
[ TIME ] Timed out waiting for device dev-ttyS0.device.
         Starting Set console font and keymap...
[  OK  ] Stopped Getty on tty1.
[  OK  ] Stopped Daily apt activities.
[  OK  ] Stopped Serial Getty on ttyS0.
[  OK  ] Closed Virtual machine log manager socket.
[  OK  ] Closed Virtual machine lock manager socket.
[  OK  ] Stopped target Graphical Interface.
[  OK  ] Started Stop ureadahead data collection 45s after completed startup.
[  OK  ] Stopped target Multi-User System.
[  OK  ] Stopped Docker Application Container Engine.
[  OK  ] Closed Docker Socket for the API.
[  OK  ] Stopped System Logging Service.
[  OK  ] Closed Syslog Socket.
[  OK  ] Stopped Ubuntu FAN network setup.
[  OK  ] Stopped Suspend Active Libvirt Guests.
[  OK  ] Stopped Login Service.
```

Use ‘root’ as Username and Password to login.



```
[  OK  ] Closed D-Bus System Message Bus Socket.
[  OK  ] Stopped Regular background program processing daemon.
[  OK  ] Stopped getty on tty2-tty6 if dbus and logind are not available.
[  OK  ] Reached target Login Prompts.
[  OK  ] Stopped Daily Cleanup of Temporary Directories.
[  OK  ] Reached target Timers.
[  OK  ] Stopped LSB: network benchmark.
[  OK  ] Stopped LSB: Set up cgroupfs mounts..
[  OK  ] Stopped target Basic System.
[  OK  ] Reached target Sockets.
[  OK  ] Stopped target System Initialization.
[  OK  ] Started Emergency Shell.
[  OK  ] Reached target Emergency Mode.
         Starting Create Volatile Files and Directories...
         Starting LSB: AppArmor initialization...
         Starting LSB: ebtables ruleset management...
[  OK  ] Started Set console font and keymap.
[  OK  ] Started Set console scheme.
[  OK  ] Started Create Volatile Files and Directories.
[  OK  ] Started LSB: AppArmor initialization.
[  OK  ] Started LSB: ebtables ruleset management.
         Starting Raise network interfaces...
         Starting Network Time Synchronization...
         Starting Update UTMP about System Boot/Shutdown...
[  OK  ] Created slice system-getty.slice.
[  OK  ] Started Network Time Synchronization.
[  OK  ] Reached target System Time Synchronized.
[  OK  ] Started Update UTMP about System Boot/Shutdown.
         Starting Update UTMP about System Runlevel Changes...
[  OK  ] Started Update UTMP about System Runlevel Changes.
[  OK  ] Started Raise network interfaces.
[  OK  ] Reached target Network.
[  OK  ] Reached target Network is Online.
Welcome to emergGive root password for maintenance
(or press Control-D to continue):
root@localhost:~#
root@localhost:~#
```



```
root@localhost:~#
root@localhost:~# uname -a
Linux localhost 4.4.65 #1 SMP PREEMPT Mon May 29 16:38:31 IST 2017 aarch64 aarch64 aarch64 GNU/Linux
root@localhost:~#
root@localhost:~#
```

5.3.6 PXE Boot

This section describes the steps required to boot the Linux kernel using PXE boot. UEFI is the primary bootloader. It loads the `GRUB2` bootloader image. PXE boot is used to load the kernel and root file system images .

Hardware Requirements

- **Host PC**: Ubuntu (64-bit variant with at least 2GB RAM) host is preferred to compile/build the UEFI firmware.
- **Board**: QorIQ Layerscape reference development board (RDB) with a UART cable.

Software Requirements

- To build the UEFI firmware on the Ubuntu Host, install `uuid-dev`. Run the following command:

  ```
  $ sudo apt-get install uuid-dev
  ```

- Ensure that Python (2.7 or higher) is installed on the build machine.
- DHCP server: isc-dhcp-server Version 4.2.4 or higher.
- Tftp Server: Any of below tftp server should be installed on host machine.
  - — tftpd-hpa
  - — atftpd
  - — dnsmasq
- To build the grub bootloader on the ubuntu Host, Install Linaro GCC-4.9 toolchain on your Host machine using the following commands:

  ```
  $ wget https://releases.linaro.org/components/toolchain/binaries/4.9-2016.02/aarch64-
  linux-gnu/gcc-linaro-4.9-2016.02-x86_64_aarch64-linux-gnu.tar.xz
  $ tar -xvf gcc-linaro-4.9-2016.02-x86_64_aarch64-linux-gnu.tar.xz
  $ export PATH=/home/user/linaro_4.9_2016/gcc-linaro-4.9-2016.02-x86_64_aarch64-linux-gnu/
  bin:$PATH
  ```

5.3.6.1 Creating the PXE Boot Setup

Setting up DHCP server for PXE boot

1. Open `/etc/dhcp/dhcpd.conf` with write permissions on the server machine.
2. Add a configuration block for PXE boot to the file.

   ```
   host ls1043rdbboard15 {
       hardware ethernet 26:5E:3D:21:00:02;
       fixed-address 192.168.3.41;
       next-server 192.168.3.161;
       filename "grub.efi"
   }
   ```

   NOTE

   Use the MAC address for which PXE boot entry is created.

3. Restart the dhcp server (The command below is for Ubuntu. It may change for a different Host)

   ```
   sudo service isc-dhcp-server restart
   ```

4. Place the following files in the tftp server root directory with execute permission.
   - `ramdisk_rootfs_arm64.ext4.gz` (Root file system) : It can be fetched using flex-builder.

     Run `flex-builder -i repo-fetch` to download the Root File System.

     Path: `packages/installer/ramdiskrfs/ramdisk_rootfs_arm64.ext4.gz`.

     Refer to #unique_113 for more info flex-builder usage.
   - Image (Kernel Image) : It can be generated using flex-builder.

     Run `flex-builder -c linux -a arm64` to generate kernel image (Image).

     Refer to #unique_113 for more info flex-builder usage.

     NOTE

     Image is the standard kernel image generated at `arch/arm64/boot/Image`.
   - `grub.cfg` : Below is the sample `grub.cfg` for LS1043ARDB.

```
set default="0"
function load_video {
if [ x$feature_all_video_module = xy ]; then
insmod all_video
else
insmod efi_gop
insmod efi_uga
insmod ieee1275_fb
insmod vbe
insmod vga
insmod video_bochs
insmod video_cirrus
fi
}
load_video
set gfxpayload=keep
set timeout=10
menuentry 'LSDK 1706 on QorIQ ARM64' --class red --class gnu-linux --class gnu --class
os {
linux /Image console=ttyS0,115200 root=/dev/ram0 rw earlycon=uart8250,mmio,0x21c0500
ramdisk_size=500000 default_hugepagesz=2m hugepagesz=2m hugepages=256
initrd /ramdisk_rootfs_arm64.ext4.gz
}
```

NOTE

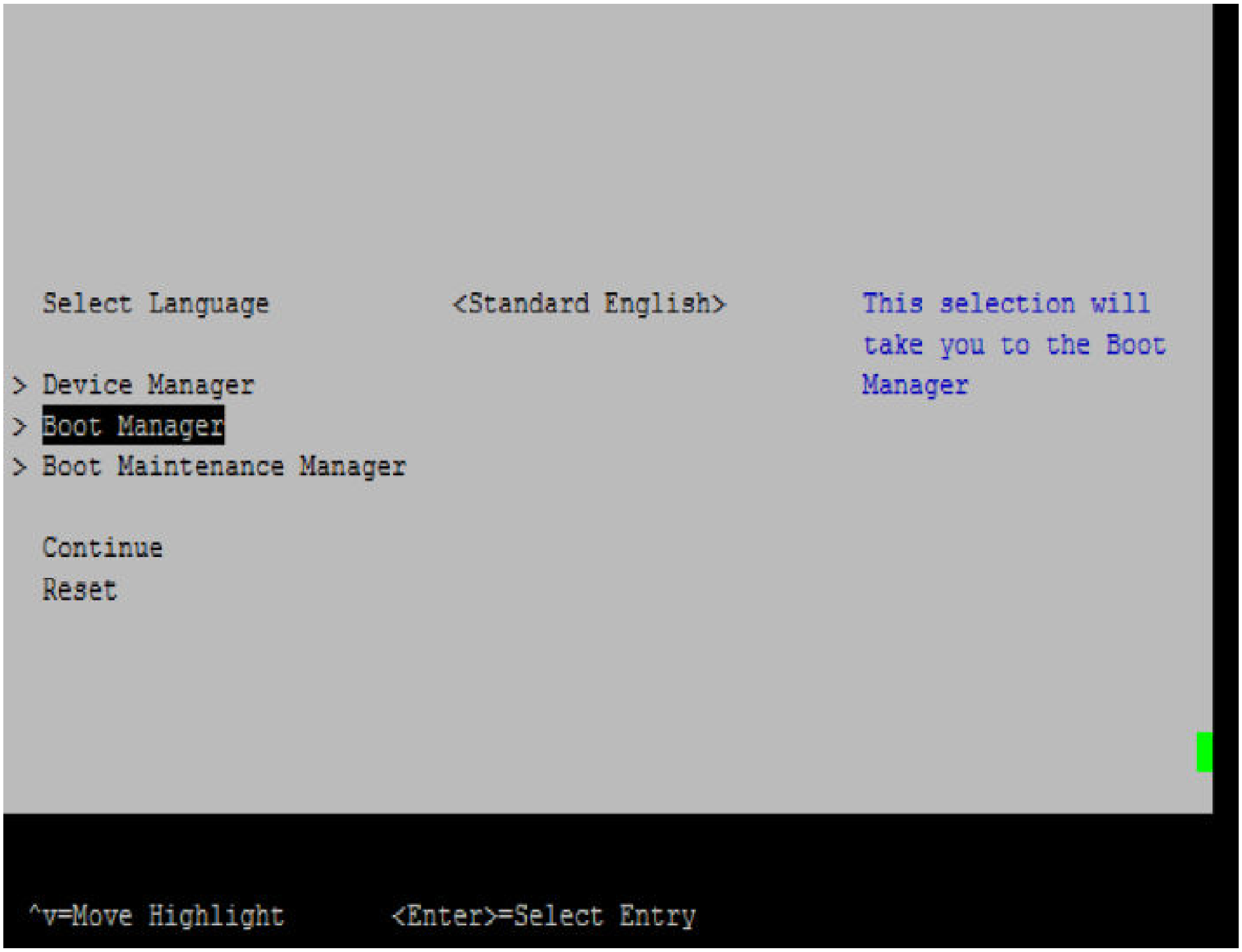
This is a sample grub configuration file for LS1043ARDB. Kernel boot arguments may change for different QorIQ LS board.

5. `grub.efi`: Grub bootloader image. The steps below can be followed to generate `grub.efi`:

```
git clone git://git.savannah.gnu.org/grub.git;cd grub
git checkout tags/grub-2.02
export PATH=/home/user/gcc-linaro-4.9-2016.02-x86_64_aarch64-linux-gnu/bin:$PATH
export CROSS_COMPILE=aarch64-linux-gnu-
./autogen.sh
./configure --target=aarch64-linux-gnu
Make
echo ' set root=(tftp)' > grub.cfg
echo ‘configfile /grub.cfg’ >> grub.cfg
grub-mkstandalone --directory=./grub-core -O arm64-efi -o grub.efi --modules "tftp net
efinet gzio linux efifwsetup part_gpt part_msdos font gfxterm all_video" /boot/grub/
grub.cfg=./grub.cfg
```

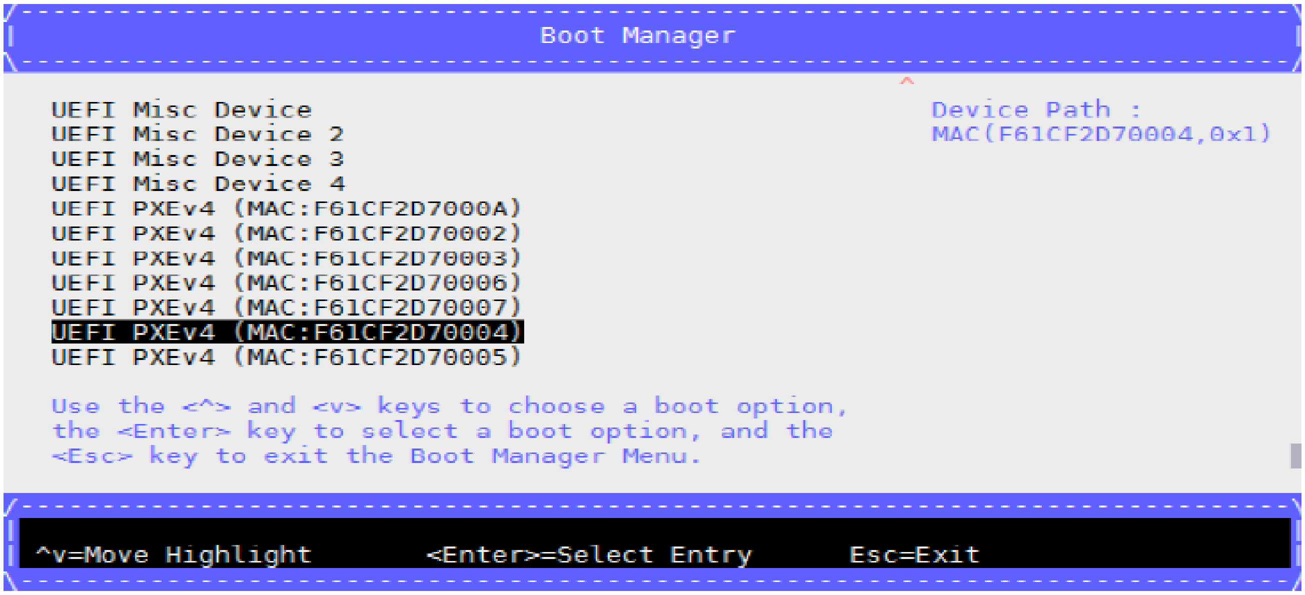
5.3.6.2 Installing the Kernel

- Boot UEFI to prompt using NOR Flash.
- Press Esc when prompted to enter the Boot Menu.
- Enter the Boot Manager



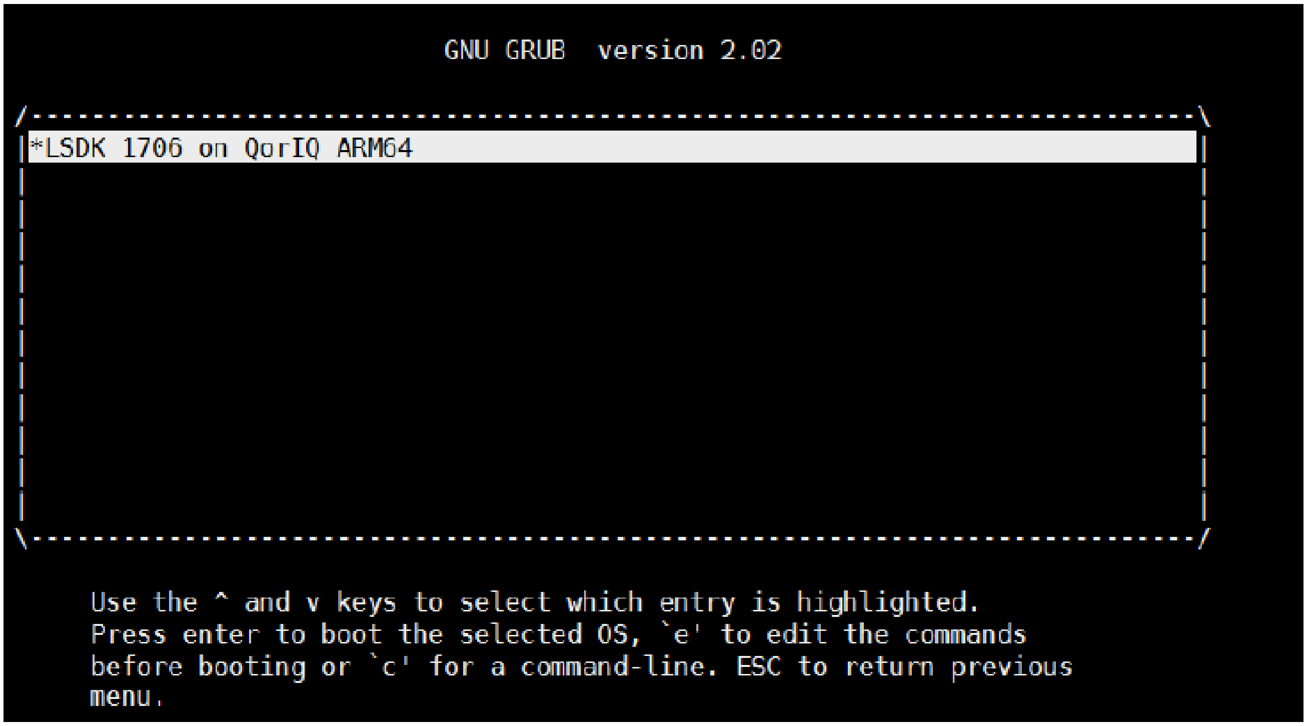
```
  Select Language          <Standard English>        This selection will
                                                     take you to the Boot
> Device Manager                                     Manager
> Boot Manager
> Boot Maintenance Manager

  Continue
  Reset



 ^v=Move Highlight       <Enter>=Select Entry
```

- Choose PXE boot option. Make sure that connectivity to dhcp server is fine and dhcp server is set up for PXE boot.

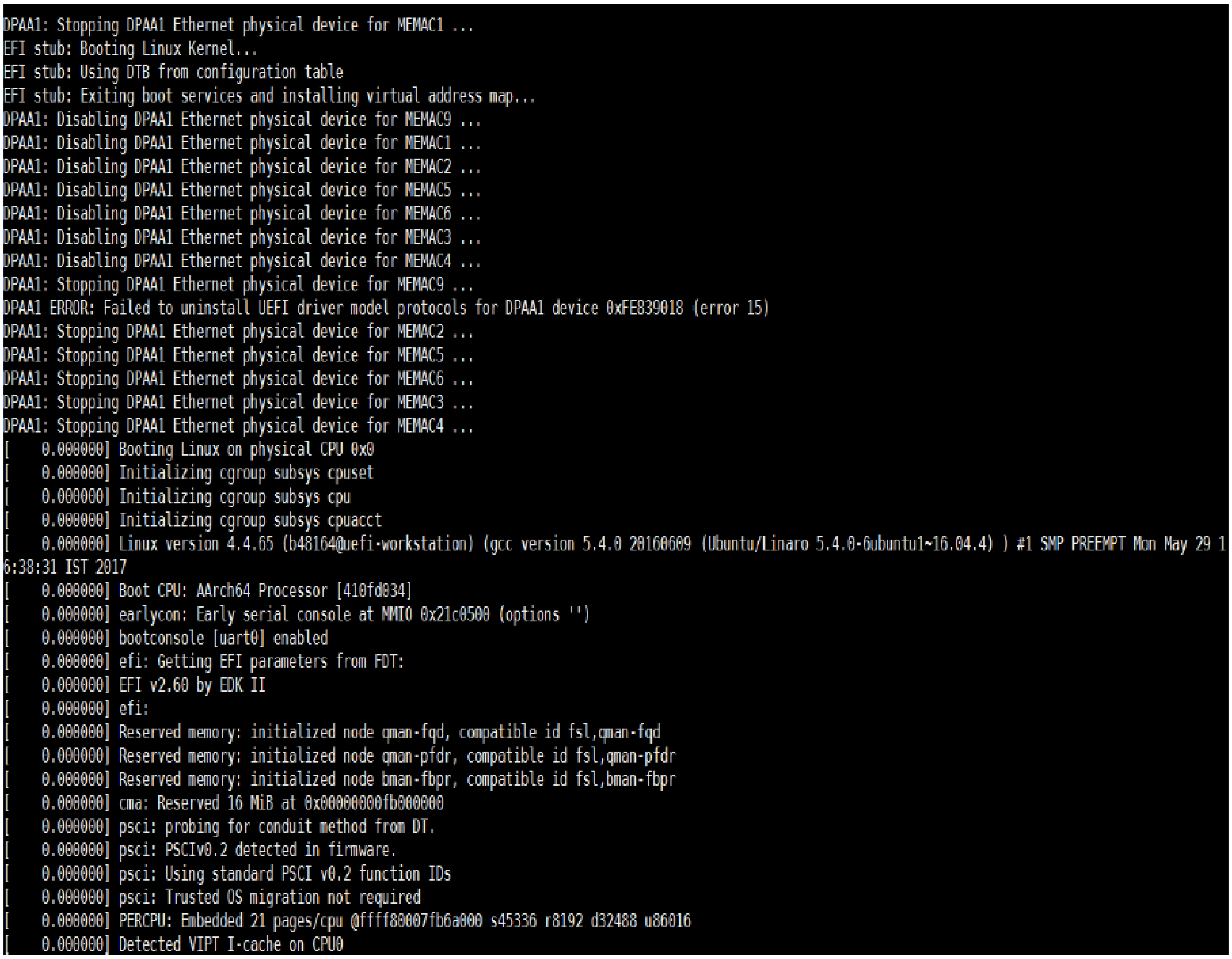


```
                                 Boot Manager

   UEFI Misc Device                                      Device Path :
   UEFI Misc Device 2                                    MAC(F61CF2D70004,0x1)
   UEFI Misc Device 3
   UEFI Misc Device 4
   UEFI PXEv4 (MAC:F61CF2D7000A)
   UEFI PXEv4 (MAC:F61CF2D70002)
   UEFI PXEv4 (MAC:F61CF2D70003)
   UEFI PXEv4 (MAC:F61CF2D70006)
   UEFI PXEv4 (MAC:F61CF2D70007)
   UEFI PXEv4 (MAC:F61CF2D70004)
   UEFI PXEv4 (MAC:F61CF2D70005)

   Use the <^> and <v> keys to choose a boot option,
   the <Enter> key to select a boot option, and the
   <Esc> key to exit the Boot Manager Menu.

  ^v=Move Highlight       <Enter>=Select Entry      Esc=Exit
```

- From grub menu select the option to install linux kernel.

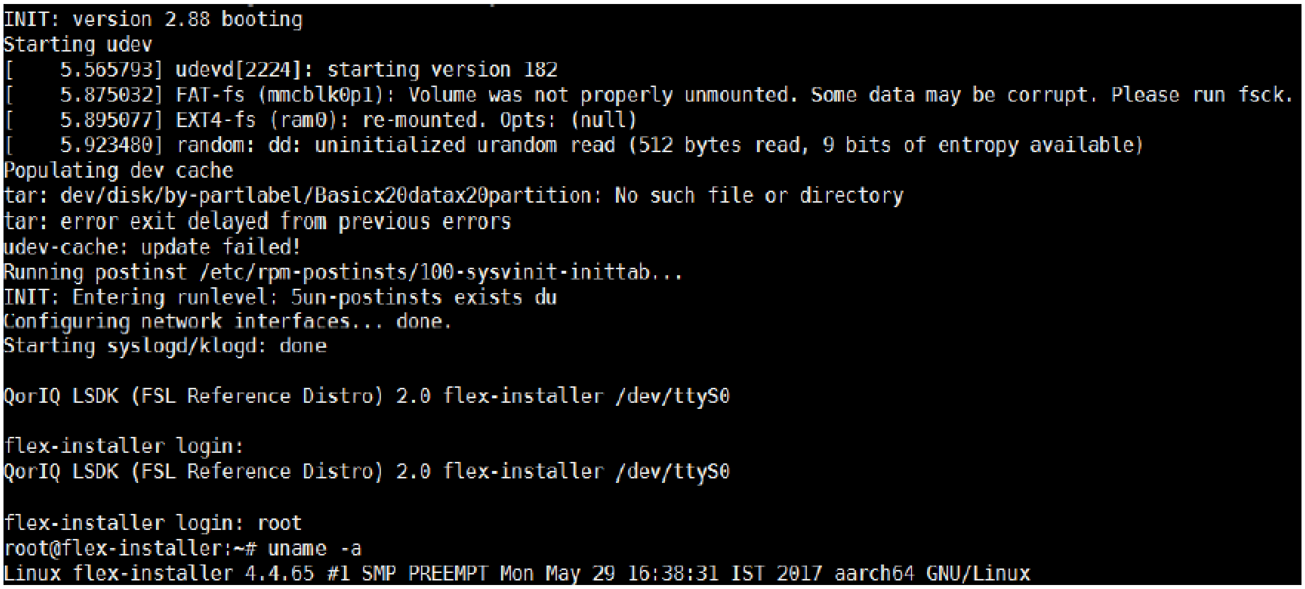


```
                    GNU GRUB  version 2.02

/----------------------------------------------------------------------------\
|*LSDK 1706 on QorIQ ARM64                                                   |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
\----------------------------------------------------------------------------/

     Use the ^ and v keys to select which entry is highlighted.
     Press enter to boot the selected OS, `e' to edit the commands
     before booting or `c' for a command-line. ESC to return previous
     menu.
```



```
DPAA1: Stopping DPAA1 Ethernet physical device for MEMAC1 ...
EFI stub: Booting Linux Kernel...
EFI stub: Using DTB from configuration table
EFI stub: Exiting boot services and installing virtual address map...
DPAA1: Disabling DPAA1 Ethernet physical device for MEMAC9 ...
DPAA1: Disabling DPAA1 Ethernet physical device for MEMAC1 ...
DPAA1: Disabling DPAA1 Ethernet physical device for MEMAC2 ...
DPAA1: Disabling DPAA1 Ethernet physical device for MEMAC5 ...
DPAA1: Disabling DPAA1 Ethernet physical device for MEMAC6 ...
DPAA1: Disabling DPAA1 Ethernet physical device for MEMAC3 ...
DPAA1: Disabling DPAA1 Ethernet physical device for MEMAC4 ...
DPAA1: Stopping DPAA1 Ethernet physical device for MEMAC9 ...
DPAA1 ERROR: Failed to uninstall UEFI driver model protocols for DPAA1 device 0xFE839018 (error 15)
DPAA1: Stopping DPAA1 Ethernet physical device for MEMAC2 ...
DPAA1: Stopping DPAA1 Ethernet physical device for MEMAC5 ...
DPAA1: Stopping DPAA1 Ethernet physical device for MEMAC6 ...
DPAA1: Stopping DPAA1 Ethernet physical device for MEMAC3 ...
DPAA1: Stopping DPAA1 Ethernet physical device for MEMAC4 ...
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 4.4.65 (b48164@uefi-workstation) (gcc version 5.4.0 20160609 (Ubuntu/Linaro 5.4.0-6ubuntu1~16.04.4) ) #1 SMP PREEMPT Mon May 29 1
6:38:31 IST 2017
[    0.000000] Boot CPU: AArch64 Processor [410fd034]
[    0.000000] earlycon: Early serial console at MMIO 0x21c0500 (options '')
[    0.000000] bootconsole [uart0] enabled
[    0.000000] efi: Getting EFI parameters from FDT:
[    0.000000] EFI v2.60 by EDK II
[    0.000000] efi:
[    0.000000] Reserved memory: initialized node qman-fqd, compatible id fsl,qman-fqd
[    0.000000] Reserved memory: initialized node qman-pfdr, compatible id fsl,qman-pfdr
[    0.000000] Reserved memory: initialized node bman-fbpr, compatible id fsl,bman-fbpr
[    0.000000] cma: Reserved 16 MiB at 0x00000000fb000000
[    0.000000] psci: probing for conduit method from DT.
[    0.000000] psci: PSCIv0.2 detected in firmware.
[    0.000000] psci: Using standard PSCI v0.2 function IDs
[    0.000000] psci: Trusted OS migration not required
[    0.000000] PERCPU: Embedded 21 pages/cpu @ffff80007fb6a000 s45336 r8192 d32488 u86016
[    0.000000] Detected VIPT I-cache on CPU0
```

- Login after successful installation.



```
INIT: version 2.88 booting
Starting udev
[    5.565793] udevd[2224]: starting version 182
[    5.875032] FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
[    5.895077] EXT4-fs (ram0): re-mounted. Opts: (null)
[    5.923480] random: dd: uninitialized urandom read (512 bytes read, 9 bits of entropy available)
Populating dev cache
tar: dev/disk/by-partlabel/Basicx20datax20partition: No such file or directory
tar: error exit delayed from previous errors
udev-cache: update failed!
Running postinst /etc/rpm-postinsts/100-sysvinit-inittab...
INIT: Entering runlevel: 5un-postinsts exists du
Configuring network interfaces... done.
Starting syslogd/klogd: done

QorIQ LSDK (FSL Reference Distro) 2.0 flex-installer /dev/ttyS0

flex-installer login:
QorIQ LSDK (FSL Reference Distro) 2.0 flex-installer /dev/ttyS0

flex-installer login: root
root@flex-installer:~# uname -a
Linux flex-installer 4.4.65 #1 SMP PREEMPT Mon May 29 16:38:31 IST 2017 aarch64 GNU/Linux
```

Chapter 6
Security

6.1 Secure boot

6.1.1 Hardware Pre-Boot Loader (PBL) based platforms

6.1.1.1 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. In the example, the ESBC is NXP* provided reference code referred to as ESBC U-Boot.

Chain of Trust ESBC U-Boot performs minimal SoC configuration before validating the next executable using the same CSF header format as the ISBC used to validate ESBC U-Boot. The CSF Header and signature are added to the next executable using the NXP code signing tool.

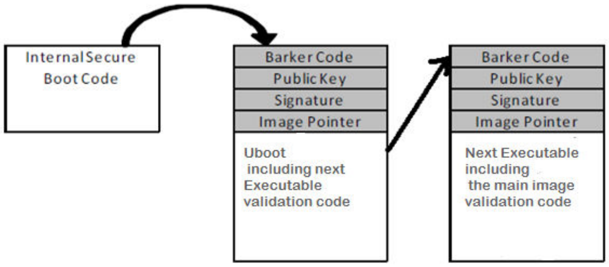

Internal Secure Boot Code
Barker Code
Public Key
Signature
Image Pointer
Uboot including next Executable validation code
Barker Code
Public Key
Signature
Image Pointer
Next Executable including the main image validation code

Figure 16. Chain of Trust

Chain of Trust with confidentiality The validated ESBC U-Boot image is allowed to use the One Time Programmable Master Key to decrypt system secrets. Cryptographic blob mechanism is used to establish Chain of Trust with confidentiality.

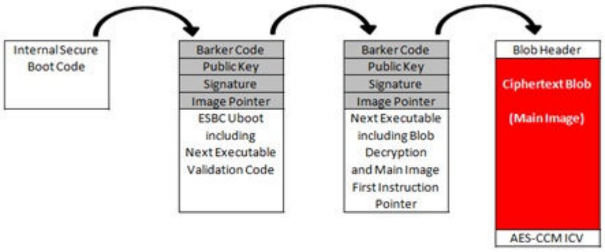

Internal Secure Boot Code
Barker Code
Public Key
Signature
Image Pointer
ESBC Uboot including Next Executable Validation Code
Barker Code
Public Key
Signature
Image Pointer
Next Executable including Blob Decryption and Main Image First Instruction Pointer
Blob Header
Ciphertext Blob
(Main Image)
AES-CCM ICV

Figure 17. Chain of Trust with confidentiality

This document provides more details on the secure boot flow, ISBC, ESBC, and NXP Code signing tool.

NOTE

NXP and Freescale may be used synonymously in this document. Registers, register file names, and Trust Architecture software may not be updated to NXP for an extended period of time.

6.1.1.2 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.

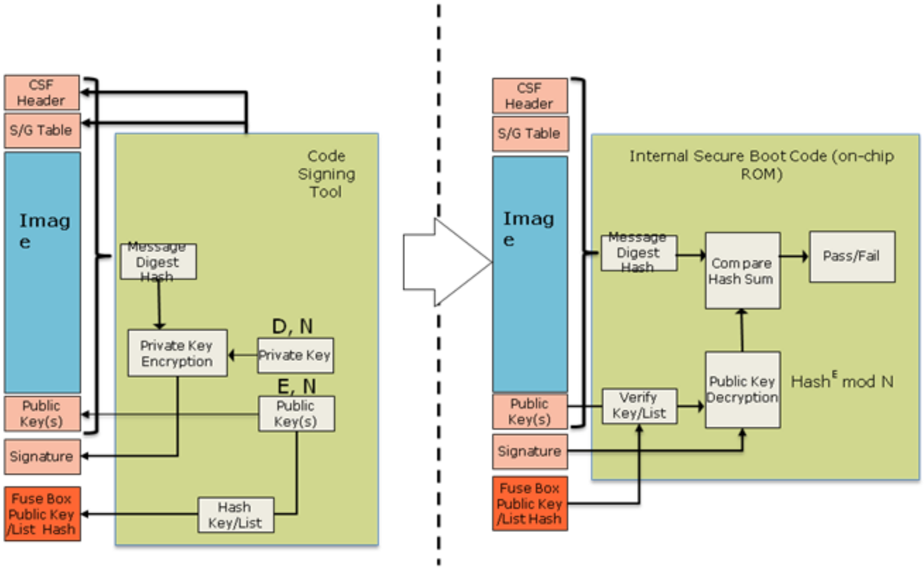


Figure 18. Secure boot process

6.1.1.3 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.

6.1.1.4 ISBC phase

6.1.1.4.1 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.

6.1.1.4.2 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.

6.1.1.4.3 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.

6.1.1.4.4 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).

6.1.1.4.5 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.

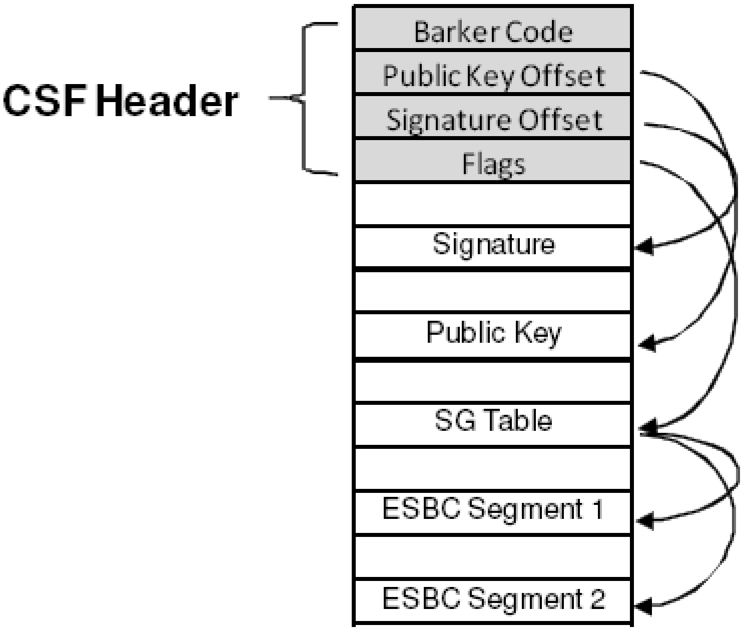


Figure 19. ESBC with CSF header

6.1.1.5 ESBC phase

Unlike the ISBC, which is in an internal ROM and therefore unchangeable, the ESBC is NXP-supplied reference code, and can be changed by OEMs. The remainder of this section is the description of a reasonable secure boot chain of trust based on NXP's reference software for secure boot. Depending on the requirement, ESBC can be a monolithic image, including U-Boot, device trees, boot firmware, drivers along with the OS and applications or can be mini U-Boot.

NXP provided ESBC consists of standard U-Boot which has been signed using a private key. U-Boot reserves a small space for storing environment variables. This space is typically one sector above or below the U-Boot and is stored on persistent storage devices like NOR flash if macro CONFIG_ENV_IS_IN_FLASH is used. In case of secure boot, macro CONFIG_ENV_IS_NOWHERE is used and so, environment is compiled in 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 Soc's with ARMv8 core (For example, LS1043, LS1046, LS1012), 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).

6.1.1.5.1 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.

6.1.1.5.1.1 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.

6.1.1.5.1.2 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.

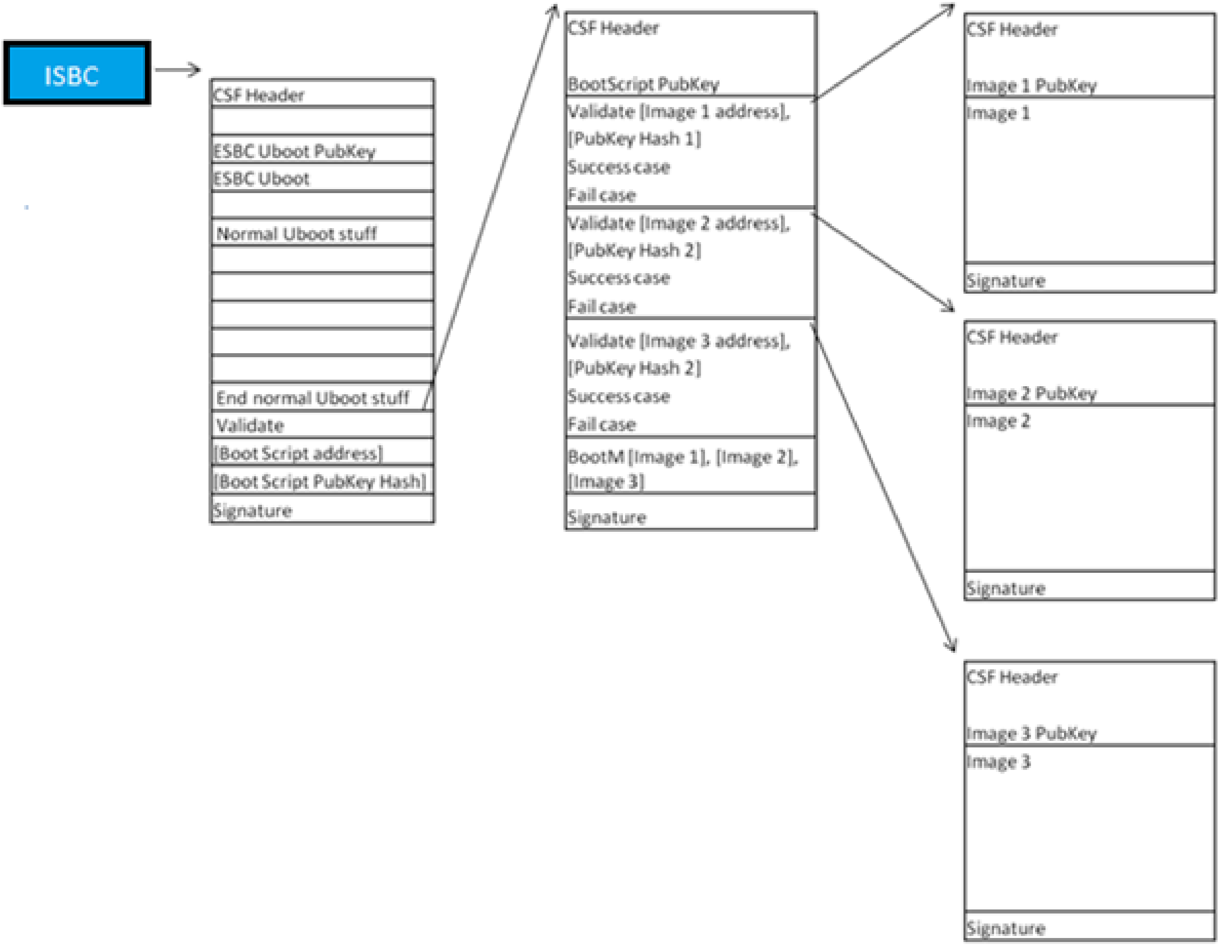


Figure 20. Secure boot flow (Chain of Trust)

6.1.1.5.1.2.1 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>
```

6.1.1.5.1.2.1.1 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

6.1.1.5.1.2.1.2 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.

6.1.1.5.1.3 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).

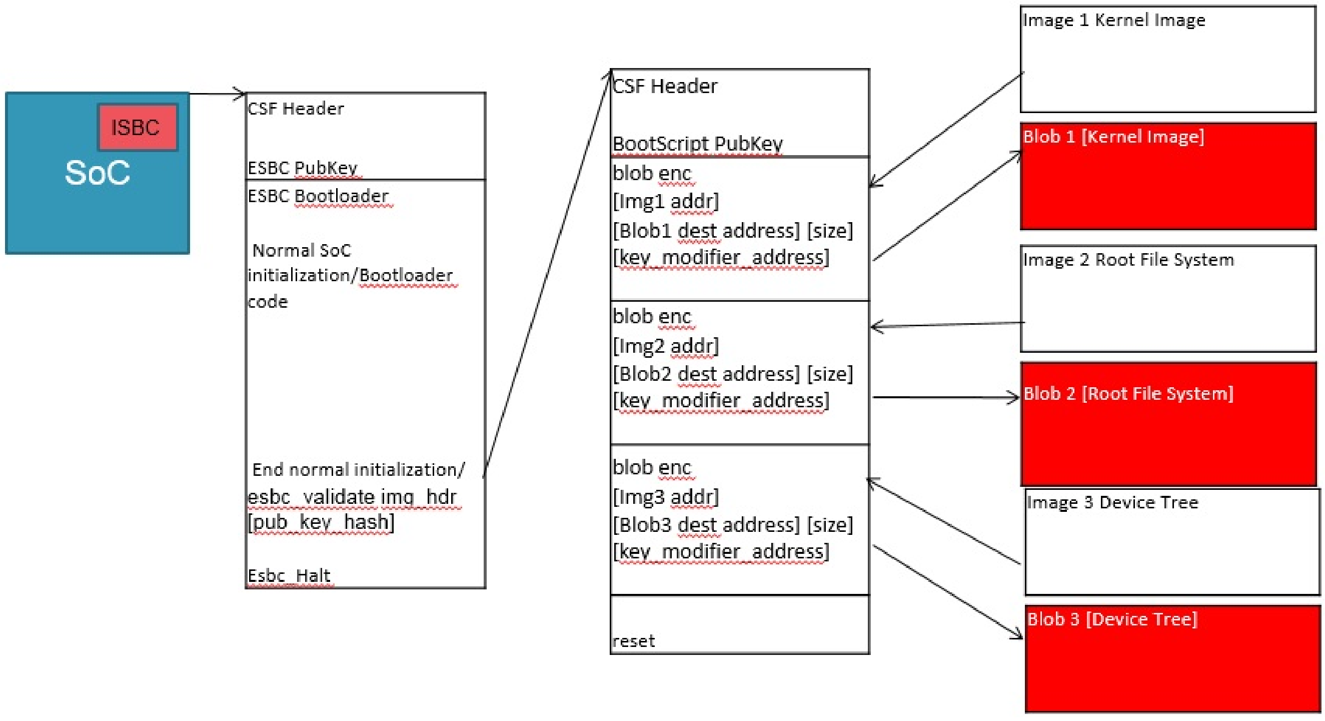


Figure 21. Chain of Trust with confidentiality (Encapsulation)

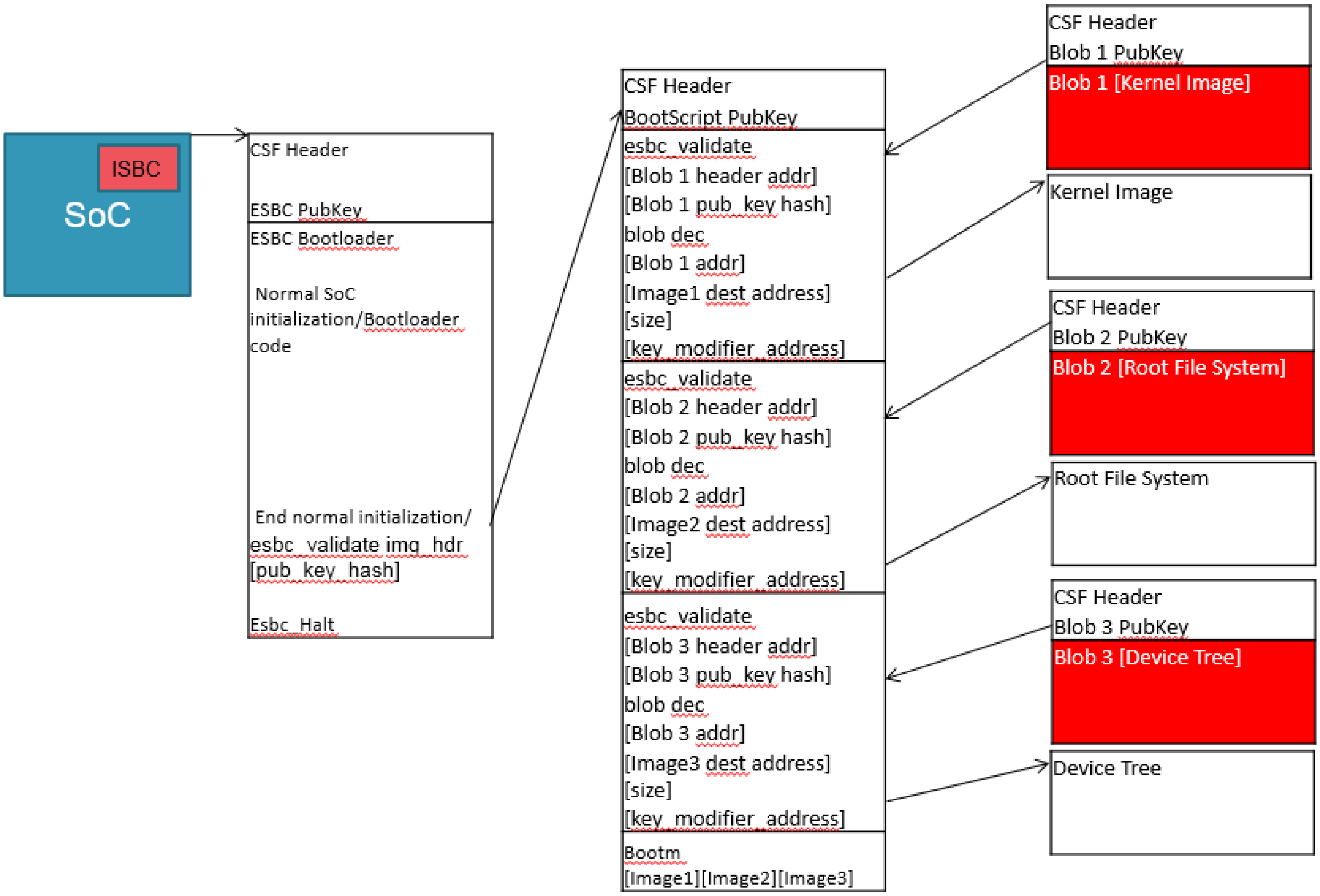


Figure 22. Chain of Trust with Confidentiality (Decapsulation)

6.1.1.5.1.3.1 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.

6.1.1.5.1.3.1.1 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>

...
```

6.1.1.5.1.3.2 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.

6.1.1.5.1.3.2.1 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>
```

6.1.1.6 Next executable (Linux phase)

The bootloader (ESBC) finishes the platform initialization and passed control to the Linux image. The boot-chain can be further extended to be able to sign application which would be running on Linux prompt. Further RTIC can be integrated to verify memory regions using Security Engine (SEC) during run time.

6.1.1.7 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.

6.1.1.7.1 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 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. 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

6.1.1.7.2 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.

[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.

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.
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.

6.1.1.7.2.1 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 ;
```

6.1.1.7.2.2 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.

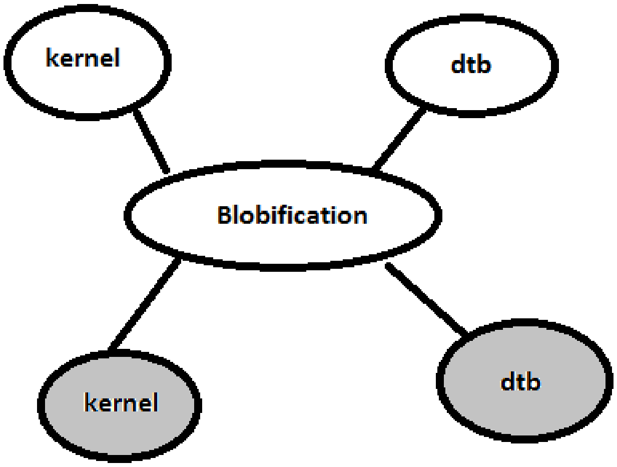
Write the SRK Hash value in SFP mirror registers and then release the core out of Boot Hold off by writing to Core Release Register in DCFG.

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

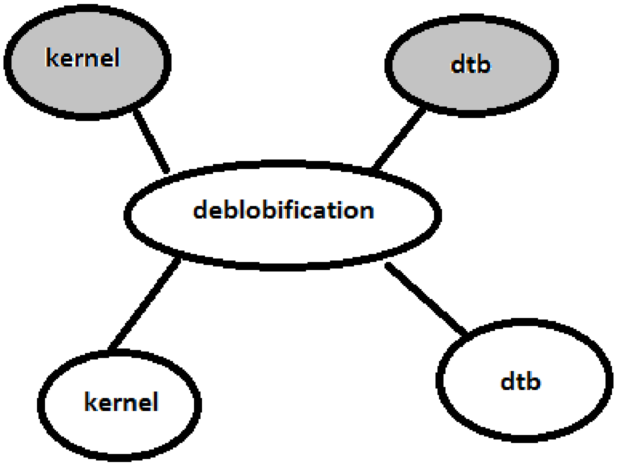


- **Subsequent Boot** .

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.

6.1.1.7.3 ISBC Key Extension (IE)

6.1.1.7.3.1 Introduction

The ISBC Key Extension feature allows the user to extend the ISBC and the number of keys available for signature validation. The ISBC uses a key directly bound to the silicon via the SRKH, the ISBC extension code (added to downstream images in a chain of trust) use IE_Keys, which are validated by the ISBC.

6.1.1.7.3.2 How it works

If IE feature is enabled in input file, the CST signs the image along with a number of public keys. Logically, it will be used when signing Boot 1 (bootloader), so that the bootloader and downstream images in the chain of trust can use keys which are not directly bound to the silicon via the SRKH. Decoupling the chain of trust from the hardware super root keys minimizes the need to perform hardware key revocation.

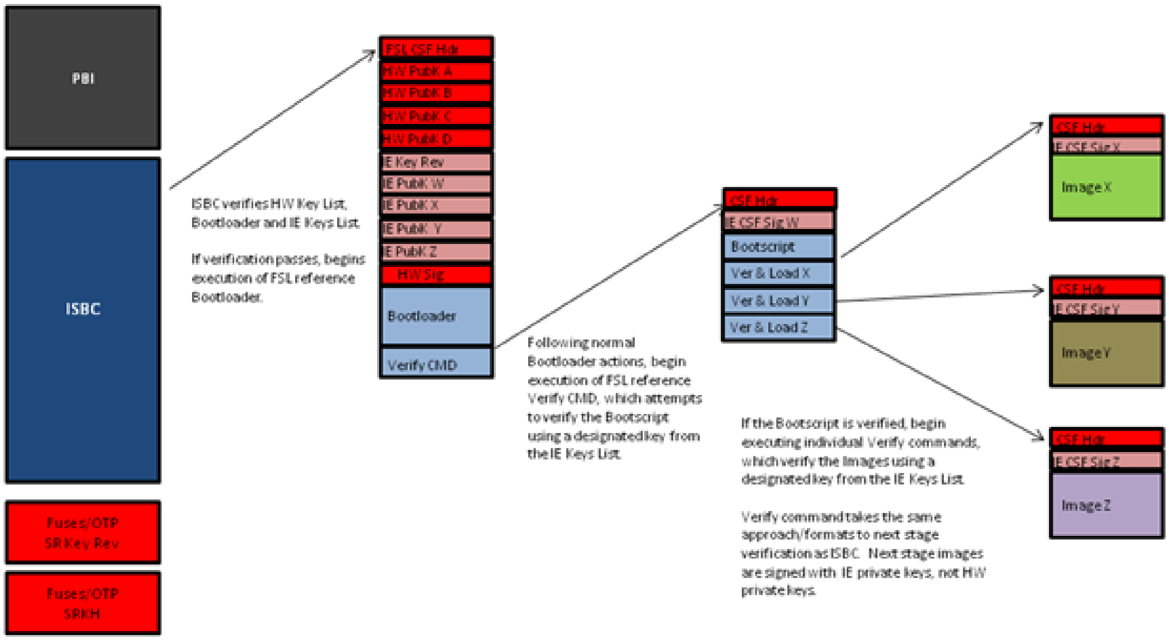

PBI
ISBC
Fuses/OTP SR Key Rev
Fuses/OTP SRKH
ISBC verifies HW Key List, Bootloader and IE Keys List.
If verification passes, begins execution of FSL reference Bootloader.
FSL CSF Hdr
HW PubK A
HW PubK B
HW PubK C
HW PubK D
IE Key Rev
IE PubK W
IE PubK X
IE PubK Y
IE PubK Z
HW Sig
Bootloader
Verify CMD
Following normal Bootloader actions, begin execution of FSL reference Verify CMD, which attempts to verify the Bootscript using a designated key from the IE Keys List.
CSF Hdr
IE CSF Sig W
Bootscript
Ver & Load X
Ver & Load Y
Ver & Load Z
If the Bootscript is verified, begin executing individual Verify commands, which verify the Images using a designated key from the IE Keys List.
Verify command takes the same approach/formats to next stage verification as ISBC. Next stage images are signed with IE private keys, not HW private keys.
CSF Hdr
IE CSF Sig X
Image X
CSF Hdr
IE CSF Sig Y
Image Y
CSF Hdr
IE CSF Sig Z
Image Z

Figure 23. Execution and verification of images using Key_Ext feature.

NOTE

Next stage images are signed with corresponding pair of Extension private keys list, not HW private keys.

Key Extension feature is applicable only for NOR secure Boot. It is not applicable for RAMBOOT (where data has to be copied onto RAM, for example, NAND, SD, and SPI)

6.1.1.7.3.3 IE Key Structure

Table 40. IE Key Structure which is embedded in header and placed in memory.

Offset	Data Bits [0:31]
0x00-0x03	This 32 bit word can be used to represent which keys from the table below have been revoked and are no longer available for use. Each bit represents 1 Key, Bit 0 represents Key 1 in the table ….Bit 31 is the 32nd key in the table
0x04-0x07	Total number of keys (Max N = 32 as 32 bit key revocation field is provided)
0x08-0x0b	Key 1 length.
0x0c-0x40b	Key 1 value.
0x40c-0x40f	Key 2 length.
0x410-0x80f	Key 2 value.
-	-
-	Key N value

6.1.1.7.3.4 Sample Input File and Output

This file is same as file described above in <link to 4.1.2> except fields required for IE Key extension highlighted in red.

```
---------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform - 1040/2080/2041/3041/4080/5020/5040/4860/4240/LS1
PLATFORM=1040
# ESBC Flag. Specify ESBC=0 to sign u-boot and ESBC=1 to sign ESBC images.(default is 0)
ESBC=0
# ESBC Header address. It contains address where ESBC header is loaded in memory.
ESBC_HDRADDR=c0b00000
---------------------------------------------------
# Entry Point/Image start address field in the header.[Mandatory]
# (default=ADDRESS of first file specified in images)
ENTRY_POINT=cffffffc
---------------------------------------------------
# Specify the file name of the keys seperated by comma.
# The number of files and key select should lie between 1 and 4 for 1040/2080 and C290.
# For rest of the platforms only one key is required and key select should not be provided.

# USAGE (for 4080/5020/5040/3041/2041/1010/913x): PRI_KEY = <key1.pri>
# USAGE (for 1040/2080/C290/4860/4240): PRI_KEY = <key1.pri>,<key2.pri>,<key3.pri>,<key4.pri>

# PRI_KEY (Default private key :srk.pri) - [Optional]
PRI_KEY=srk.pri
# PUB_KEY (Default public key :srk.pub) - [Optional]
PUB_KEY=srk.pub
# Please provide KEY_SELECT(between 1 to 4) (Required for 1040/2080/C290/4860/4240 only) -
[Optional]
KEY_SELECT=

# Specify the file name of the extension keys seperated by comma.
# USAGE : IE_KEY = <key1.pub>,<key2.pub>,<key3.pub>,<key4.pub>,<key5.pub>
IE_KEY=<iekey1k_1.pub>,<iekey1k_2.pub>,<iekey1k_3.pub>,<iekey2k_1.pub>,<iekey2k_2.pub>,<iekey
2k_3.pub>,<iekey4k_1.pub>,<iekey4k_2.pub>

# Please provide Revoke keys. - [Optional]
# Provide key numbers from available ie keys to be revoked. Max n-1 keys can be revoked. n
```

```
is total number of IE keys.
# LSb represents key0 and MSb represents key 31. So total 32 keys are supported.
IE_REVOC=1,7
---------------------------------------------------
# Specify SG table address, only for (2041/3041/4080/5020/5040) with ESBC=0 - [Optional]
SG_TABLE_ADDR=
---------------------------------------------------
# Specify the target where image will be loaded. (Default is NOR_16B) - [Optional]
# Only required for Non-PBL Devices (1010/9131/9132/C290)
# Select from - NOR_8B/NOR_16B/NAND_8B_512/NAND_8B_2K/NAND_8B_4K/NAND_16B_512/NAND_16B_2K/
NAND_16B_4K/SD/MMC/SPI
IMAGE_TARGET=
---------------------------------------------------
# Specify IMAGE, Max 8 images are possible. DST_ADDR is required only for Non-PBL Platform.
[Mandatory]
# In case using IE_KEY, Max 7 images are possible. [Mandatory]
# USAGE : IMAGE_NO = {IMAGE_NAME, SRC_ADDR, DST_ADDR}
IMAGE_1={u-boot.bin,cff40000,ffffffff}
IMAGE_2={,,}
IMAGE_3={,,}
IMAGE_4={,,}
IMAGE_5={,,}
IMAGE_6={,,}
IMAGE_7={,,}
---------------------------------------------------
# Specify OEM AND FSL ID to be populated in header. [Optional]
# e.g FSL_UID=11111111
FSL_UID=
OEM_UID=
---------------------------------------------------
# Specify the file names of csf header and sg table. (Default :hdr.out) [Optional]
OUTPUT_HDR_FILENAME=hdr_uboot.out

# Specify the file names of hash file and sign file.
HASH_FILENAME=img_hash.out
INPUT_SIGN_FILENAME=sign.out

# Specify the signature size.It is mandatory when neither public key nor private key is
specified.
# Signature size would be [0x80 for 1k key, 0x100 for 2k key, and 0x200 for 4k key].
SIGN_SIZE=
---------------------------------------------------
# Specify the output file name of sg table. (Default :sg_table.out). [Optional]
# Please note that OUTPUT SG BIN is only required for 2041/3041/4080/5020/5040 when ESBC
flag is not set.
OUTPUT_SG_BIN=
---------------------------------------------------
# Following fields are Required for 4240/4860/1040/2080/C290 only

# Specify House keeping Area
# Required for 4240/4860/1040/2080/C290 only when ESBC flag is not set. [Mandatory]
HK_AREA_POINTER=bff00000
HK_AREA_SIZE=00010000
---------------------------------------------------
# Following field Required for 4240/4860/1040/2080/C290 only
# Specify Secondary Image Flag. (0 or 1) - [Optional]
# (Default is 0)
SEC_IMAGE=
---------------------------------------------------
```

Table 41. Description of new fields introduced.

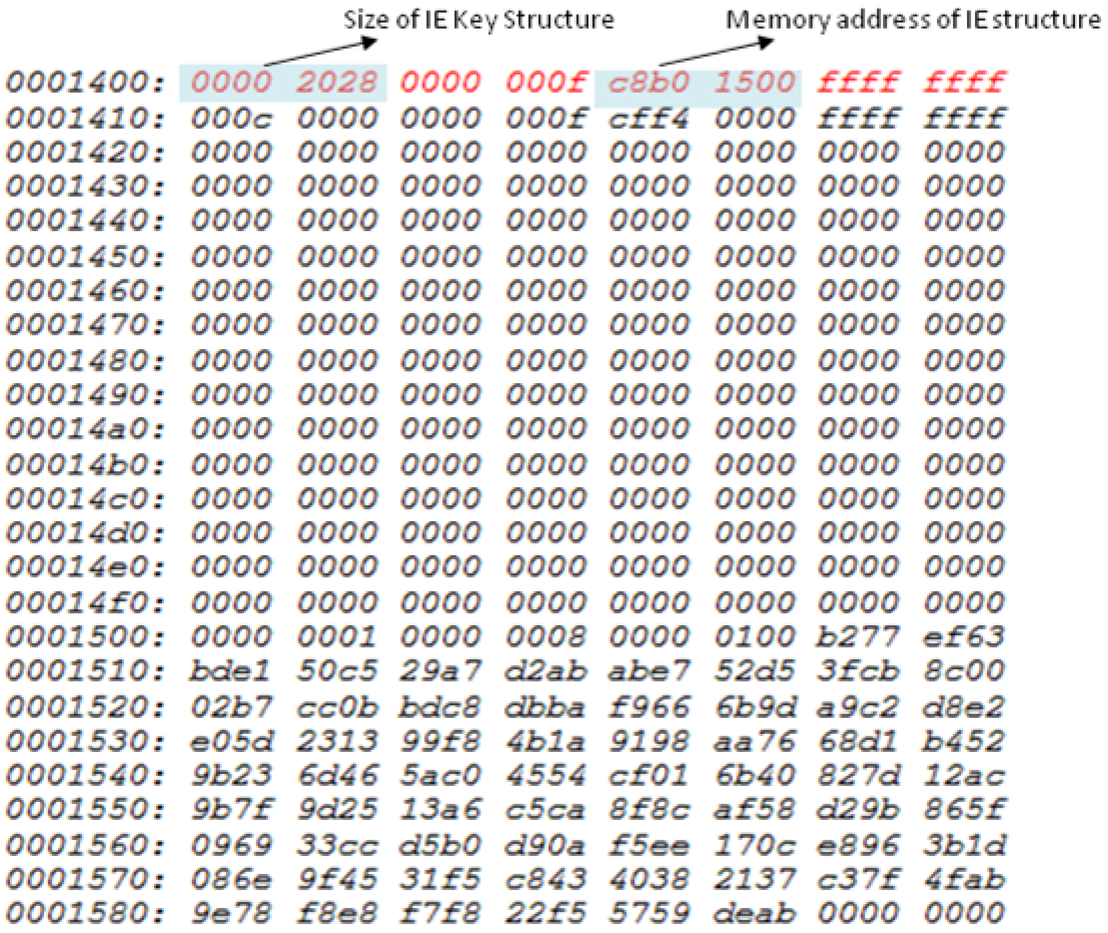
Field	Field Description
ESBC_HDRADDR	ESBC Header address. It contains location of ESBC header in the memory

Table continues on the next page...

Table 41. Description of new fields introduced. (continued)

Field	Field Description
IE_KEY	Extension Public key filenames to be used by further level images. (File has to be in PEM format) FILE1 [,FILE2, FILE3, FILE4].
IE_REVOC	Revoked keys numbers from available ie keys. If a key is compromised then this feature helps to avoid that key usage. Max n-1 keys can be revoked. n is total number of IE keys and less than equal to 32.Ex.[1,3,5]

OUTPUT



```
0001400: 0000 2028 0000 000f c8b0 1500 ffff ffff
0001410: 000c 0000 0000 000f cff4 0000 ffff ffff
0001420: 0000 0000 0000 0000 0000 0000 0000 0000
0001430: 0000 0000 0000 0000 0000 0000 0000 0000
0001440: 0000 0000 0000 0000 0000 0000 0000 0000
0001450: 0000 0000 0000 0000 0000 0000 0000 0000
0001460: 0000 0000 0000 0000 0000 0000 0000 0000
0001470: 0000 0000 0000 0000 0000 0000 0000 0000
0001480: 0000 0000 0000 0000 0000 0000 0000 0000
0001490: 0000 0000 0000 0000 0000 0000 0000 0000
00014a0: 0000 0000 0000 0000 0000 0000 0000 0000
00014b0: 0000 0000 0000 0000 0000 0000 0000 0000
00014c0: 0000 0000 0000 0000 0000 0000 0000 0000
00014d0: 0000 0000 0000 0000 0000 0000 0000 0000
00014e0: 0000 0000 0000 0000 0000 0000 0000 0000
00014f0: 0000 0000 0000 0000 0000 0000 0000 0000
0001500: 0000 0001 0000 0008 0000 0100 b277 ef63
0001510: bde1 50c5 29a7 d2ab abe7 52d5 3fcb 8c00
0001520: 02b7 cc0b bdc8 dbba f966 6b9d a9c2 d8e2
0001530: e05d 2313 99f8 4b1a 9198 aa76 68d1 b452
0001540: 9b23 6d46 5ac0 4554 cf01 6b40 827d 12ac
0001550: 9b7f 9d25 13a6 c5ca 8f8c af58 d29b 865f
0001560: 0969 33cc d5b0 d90a f5ee 170c e896 3b1d
0001570: 086e 9f45 31f5 c843 4038 2137 c37f 4fab
0001580: 9e78 f8e8 f7f8 22f5 5759 deab 0000 0000
```

Highlighted fields shows IE structure is embedded in the CSF header.

6.1.1.7.3.5 Generate Header for Next Level Images (bootscript, rootfs, dtb, linux).

IE key table generated in previous is embedded along with the CSF header for u-boot. Boot ROM code verifies these keys along with the bootloader. For the rest of the images in the chain of trust, user can use the keys in the IE key table. The IE Key Table is in the memory already, the sample input file needs to have the IE Key number to be used.(IE_KEY_SEL). The corresponding private key of the file needs to be provided for signature to be generated (PRI_KEY).

This sample file is same as file described above in <link to 4.1.2> except fields required for IE Key extension highlighted in red.

CSF Header for bootscript

```
---------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform - 1040/2080/2041/3041/4080/5020/5040/4860/4240/LS1
PLATFORM=1040
# ESBC Flag. Specify ESBC=0 to sign u-boot and ESBC=1 to sign ESBC images.(default is 0)
ESBC=1
---------------------------------------------------
# Entry Point/Image start address field in the header.[Mandatory]
# (default=ADDRESS of first file specified in images)
ENTRY_POINT=e8a00000
---------------------------------------------------
# Specify the file name of the keys seperated by comma.
# The number of files and key select should lie between 1 and 4 for 1040/2080 and C290.
# For rest of the platforms only one key is required and key select should not be provided.

# USAGE (for 4080/5020/5040/3041/2041/1010/913x): PRI_KEY = <key1.pri>
# USAGE (for 1040/2080/C290/4860/4240): PRI_KEY = <key1.pri>, <key2.pri>, <key3.pri>, 
<key4.pri>

# PRI_KEY (Default private key :srk.pri) - [Optional]
PRI_KEY=iekey4k_2.pri
# PUB_KEY (Default public key :srk.pub) - [Optional]
PUB_KEY=
# Please provide KEY_SELECT(between 1 to 4) (Required for 1040/2080/C290/9164/4240 only) - 
[Optional]
KEY_SELECT=
---------------------------------------------------
# Specify SG table address, only for (2041/3041/4080/5020/5040) with ESBC=0 - [Optional]
SG_TABLE_ADDR=
---------------------------------------------------
# Specify IE_KEY to be used for signature verification. [Mandatory]
IE_KEY_SEL=8
---------------------------------------------------
# Specify the target where image will be loaded. (Default is NOR_16B) - [Optional]
# Only required for Non-PBL Devices (1010/9131/9132/C290)
# Select from - NOR_8B/NOR_16B/NAND_8B_512/NAND_8B_2K/NAND_8B_4K/NAND_16B_512/NAND_16B_2K/
NAND_16B_4K/SD/MMC/SPI
IMAGE_TARGET=
---------------------------------------------------
# Specify IMAGE, Max 8 images are possible. DST_ADDR is required only for Non-PBL Platform. 
[Mandatory]
# In case using IE_KEY, Max 1 image is possible. [Mandatory]
# USAGE : IMAGE_NO = {IMAGE_NAME, SRC_ADDR, DST_ADDR}
IMAGE_1={bootscript,e8a00000,ffffffff}
IMAGE_2={,,}
IMAGE_3={,,}
IMAGE_4={,,}
IMAGE_5={,,}
IMAGE_6={,,}
IMAGE_7={,,}
IMAGE_8={,,}
---------------------------------------------------
# Specify OEM AND FSL ID to be populated in header. [Optional]
# e.g FSL_UID=11111111
FSL_UID=
OEM_UID=
---------------------------------------------------
# Specify the file names of csf header. (Default :hdr.out) [Optional]
OUTPUT_HDR_FILENAME=hdr_bs.out

# Specify the file names of hash file and sign file.
HASH_FILENAME=img_hash.out
INPUT_SIGN_FILENAME=sign.out
```

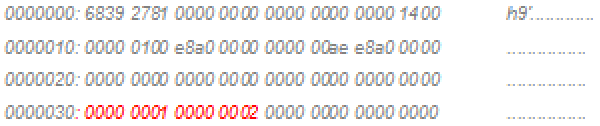
```
# Specify the signature size.It is mandatory when neither public key nor private key is
specified.
# Signature size would be [0x80 for 1k key, 0x100 for 2k key, and 0x200 for 4k key].
SIGN_SIZE=0x200
---------------------------------------------------
# Specify the output file name of sg table. (Default :sg_table.out). [Optional]
# Please note that OUTPUT SG BIN is only required for 2041/3041/4080/5020/5040 when ESBC
flag is not set.
OUTPUT_SG_BIN=

---------------------------------------------------
# Following fields are Required for 4240/9164/1040/2080/C290 only

# Specify House keeping Area
# Required for 42409164/1040/2080/C290 only when ESBC flag is not set. [Mandatory]
HK_AREA_POINTER=
HK_AREA_SIZE=
---------------------------------------------------
# Following field Required for 4240/9164/1040/2080/C290 only
# Specify Secondary Image Flag. (0 or 1) - [Optional]
# (Default is 0)
SEC_IMAGE=
---------------------------------------------------
```

Table 42. Description of new fields introduced.

Field	Field Description
IE_KEY_SEL	IE_KEY number for public key in IE Key table to be used for signature verification of ESBC image.

OUTPUT

Given below is a snapshot of header generated in which highlighted fields indicates IE flag is ON and IE KEY SELECT i.e. key to be used to verify image is embedded in header.



```
0000000: 6839 2781 0000 0000 0000 0000 0000 1400        h9'............
0000010: 0000 0100 e8a0 0000 0000 00ae e8a0 0000        ...............
0000020: 0000 0000 0000 0000 0000 0000 0000 0000        ...............
0000030: 0000 0001 0000 0002 0000 0000 0000 0000        ...............
```

Highlighted fields shows IE key select in CSF header.

6.1.1.8 Troubleshooting

Table 43. 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, OTPMK_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.

6.1.1.9 CSF Header Data Structure

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

LS1 Platform

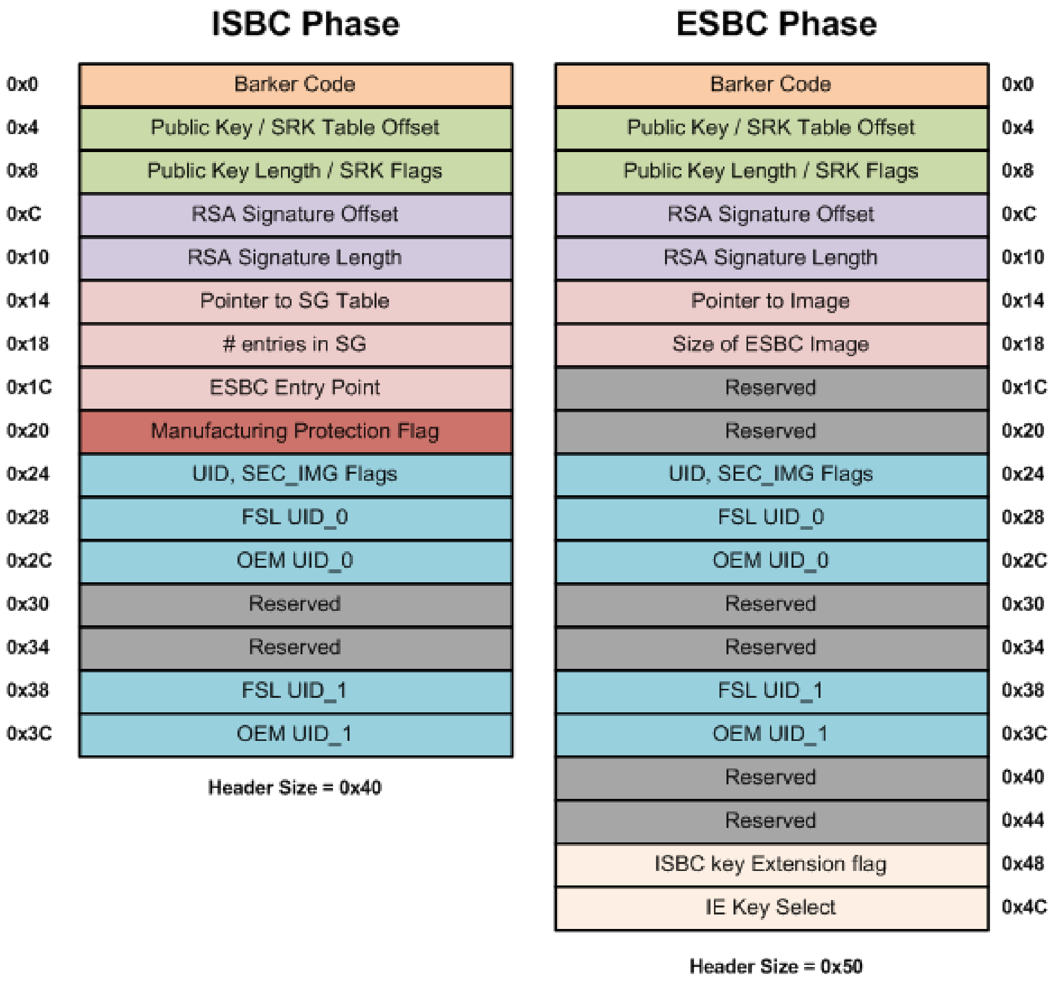


Figure 24. CSF Header for LS1 (ISBC and ESBC Phase)

Table 44. 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 44. 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 44. 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 44. 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 45. 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 46. Signature (LS1 Platform)

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

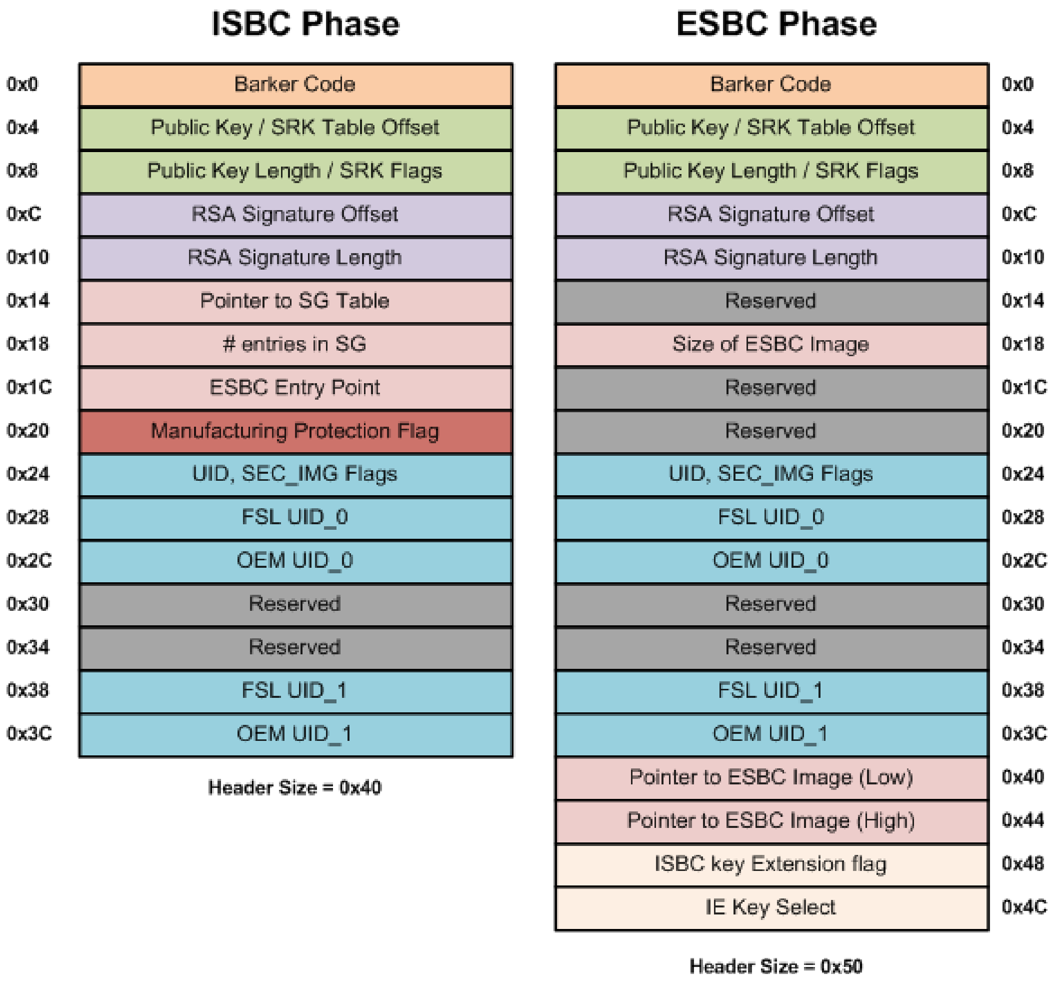
Table 47. Public key (LS1 Platform)

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 48. 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)

LS1043/LS1046/LS1012 Platforms



ISBC Phase

0x0	Barker Code
0x4	Public Key / SRK Table Offset
0x8	Public Key Length / SRK Flags
0xC	RSA Signature Offset
0x10	RSA Signature Length
0x14	Pointer to SG Table
0x18	# entries in SG
0x1C	ESBC Entry Point
0x20	Manufacturing Protection Flag
0x24	UID, SEC_IMG Flags
0x28	FSL UID_0
0x2C	OEM UID_0
0x30	Reserved
0x34	Reserved
0x38	FSL UID_1
0x3C	OEM UID_1

Header Size = 0x40

ESBC Phase

Barker Code	0x0
Public Key / SRK Table Offset	0x4
Public Key Length / SRK Flags	0x8
RSA Signature Offset	0xC
RSA Signature Length	0x10
Reserved	0x14
Size of ESBC Image	0x18
Reserved	0x1C
Reserved	0x20
UID, SEC_IMG Flags	0x24
FSL UID_0	0x28
OEM UID_0	0x2C
Reserved	0x30
Reserved	0x34
FSL UID_1	0x38
OEM UID_1	0x3C
Pointer to ESBC Image (Low)	0x40
Pointer to ESBC Image (High)	0x44
ISBC key Extension flag	0x48
IE Key Select	0x4C

Header Size = 0x50

Figure 25. CSF Header for LS1043/LS1046//LS1012 (ISBC and ESBC Phase)

Table 49. CSF Header Format (LS1043/LS1046/LS1012 Platforms)

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 49. CSF Header Format (LS1043/LS1046/LS1012 Platforms) (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 49. CSF Header Format (LS1043/LS1046/LS1012 Platforms) (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 50. Scatter Gather Table Format (LS1043/LS1046/LS1012 Platforms)

Offset	Data Bits [0:31]
0x00-0x03	Length. This location specifies the length in bytes of the ESBC image 1.
0x04-0x07	Target where the ESBC Image 1 can be found. This field is ignored in case of PBL based SOC's.
0x08-0x0b	Source Address of ESBC Image 1
0x0c-0x0f	Destination Address of ESBC Image 1 If the target address is 0xffffffff, the image is not copied to the target. This field is ignored in case of PBL based SOC's.
0x10-0x13	Length. This location specifies the length in bytes of the ESBC image 2.
0x14-0x17	Target where the ESBC Image 2 can be found. This field is ignored in case of PBL based SOC's.
0x18-0x1b	Source Address of ESBC Image 2
0x1c-0x1f	Destination Address of ESBC Image 2 If the target address is 0xffffffff, the image is not copied to the target. This field is ignored in case of PBL based SOC's.

Table 51. Signature (LS1043/LS1046/LS1012 Platforms)

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

Table 52. Public key (LS1043/LS1046/LS1012 Platforms)

Offset	Data Bits [0:31]
0x00-size	Public Key Value. The hash of this public key is compared with the hash stored in Secure Fuse Processor SRKH registers.

Table 53. SRK Table (LS1043/LS1046/LS1012 Platforms)

Offset	Data Bits [0:31]
0x00-0x03	Key 1 length
0x04-0x403	Key 1 value. (Remaining bytes will be padded with zero)
0x404-0x407	Key 2 length
0x408-0x807	Key 2 value. (Remaining bytes will be padded with zero)
0x808-0x80b	Key 3 length
0x80c-0xb0b	Key 3 value. (Remaining bytes will be padded with zero)
0xb0c-0xb0f	Key 4 length
0xb10-0xe10	Key 4 value. (Remaining bytes will be padded with zero)

6.1.1.10 ISBC Validation Error Codes

LS1/LS1043/LS1046/LS1012 platforms

Errors in the system can be of following types:

1. Core Exceptions
2. System State Failures
3. Header Checking Failures
    a. General Failures
    b. Key/Signature/UID related errors
4. Verification Failures
5. SEC/PAMU errors

Table 54. 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 55. Core Exceptions (LS1043/LS1046/LS1012 platforms)

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 56. System State Failures (LS1/LS1043/LS1046/LS1012 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 57. General Header Checking Failures (LS1/LS1043/LS1046/LS1012 platforms)

Value	Code	Definition
0x301	ERROR_ESBC_HDR_LOC	ESBC header location is not in 3.5G space
0x302	ERROR_ESBC_HEADER_BARKER	Barker code in the header is incorrect.
0x303	ERROR_ESBC_HEADER_SG_ENTRIES_NOT_IN_3_5G	SG table/ESBC image address (header address + image offset in sg table) is beyond 3.5G
0x303	ERROR_ESBC_HEADER_SG_ENTRIES_ON_OCRAM	One Entry in the SG table is on OCRAM
0x304	ERROR_ESBC_HEADER_SG_ESBC_EP	ESBC entry point in header not within ESBC address range
0x305	ERROR_SGL_ENTIRES_NOT_SUPPORTED	Number of entries in SG table exceeds maximum limit i.e 8
0x306	ERROR_ESBC_HEADER_HKAREA_LEN_ZERO	Houskeeping area not provided in header
0x307	ERROR_ESBC_HEADER_HKAREA_NOT_IN_3_5G	House keeping area not in 3.5G boundary
0x308	ERROR_ESBC_HEADER_HKAREA_LEN_INSUFFICIENT	Housekeeping area length provided is not sufficient.
0x309	ERROR_SG_TABLE_NOT_IN_3_5	SG Table is not in 3.5G boundary
0x309	ERROR_SG_TABLE_ON_OCRAM	SG table is on OCRAM
0x310	ERROR_ESBC_HEADER_HKAREA_NOT_4K_ALIGNED	House keeping area is not aligned to 4K boundary
0x311	ERROR_SGL_ENTRIES_SIZE_ZERO	SG table has entry with size zero.

Table 58. Key/Signature/UID related errors (LS1/LS1043/LS1046/LS1012 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 58. Key/Signature/UID related errors (LS1/LS1043/LS1046/LS1012 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 59. Verification Failures (LS1/LS1043/LS1046/LS1012 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 60. SEC/PAMU Failures (LS1/LS1043/LS1046/LS1012 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 60. SEC/PAMU Failures (LS1/LS1043/LS1046/LS1012 platforms) (continued)

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

6.1.1.11 ESBC Validation Error Codes

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

Table 61. 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 61. 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.

6.1.1.12 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	
LS1020A	2.1	3.3	1.89 V	A	SFP	2	SFP	
LS1043A	2.1	3.3	1.89 V	A	SFP	2	SFP	
LS1046A	2.1	3.3	1.89 V	A	SFP	2	SFP	
LS1012	2.1	3.3	1.89 V	A	SFP	2	SFP	

6.1.2 Service Processor (SP) Based Platforms

6.1.2.1 Secure Boot Introduction

There are three steps in the boot flow:

1. Service Processor Boot ROM

   SP provides pre-boot initialization and secure boot capabilities. The on-chip SP Boot ROM offers read-only, non-volatile storage for the Boot ROM code, including the internal secure boot code (ISBC) sub-routine for image validation. This Boot ROM is an integral part in the booting of the SOC in non-secure and secure boot modes.

2. GPP Boot ROM

   The on-chip GPP Boot ROM executes when the GPP cores are released from reset and is reponsible for passing control to next step in the boot flow i.e. the boot-loader validated by the SP.

3. Boot Loader

   The Boot Loader might further contain the External Secure Boot Code (ESBC) sub-routine for validation of next level images.

This document is intended for end-users to demonstrate the image validation process which happens in ISBC and ESBC.

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.

The Root of Trust is already established in the ISBC code residing in the Boot ROM which validates the Boot Loader 1.

Boot Loader 1 performs minimal SoC configuration before validating the Next Executable(s) which is/are known as ESBC image(s).

The flow includes validation of all ESBC images by a previously validated image before its execution to form a **Chain of TRUST**. The reference ESBC code also contains the functionality to form a **Chain of TRUST with confidentiality** where the next level images are kept on flash after encryption.

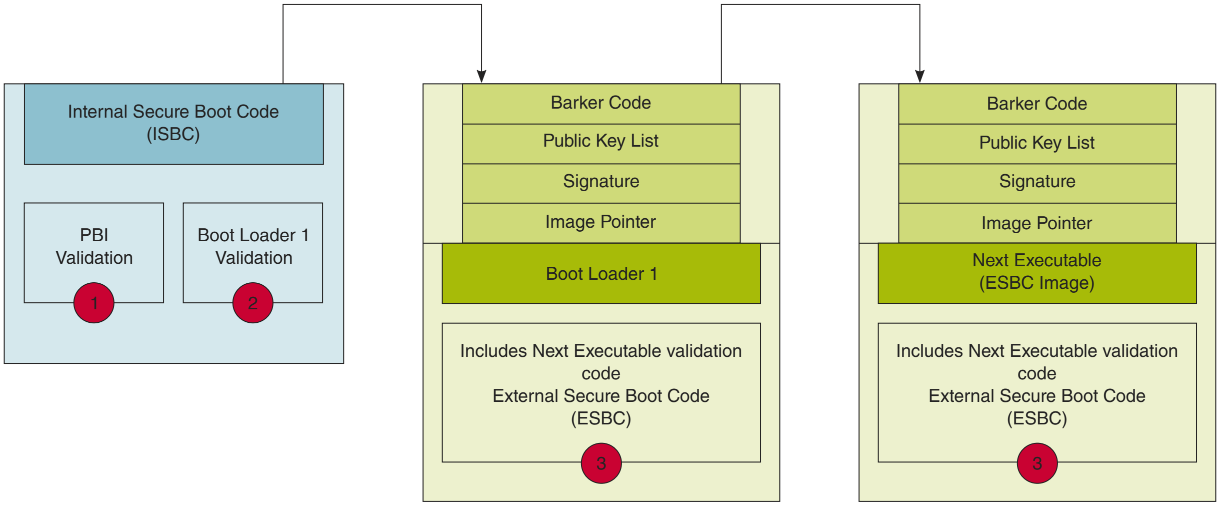

Internal Secure Boot Code (ISBC)
PBI Validation
1
Boot Loader 1 Validation
2
Barker Code
Public Key List
Signature
Image Pointer
Boot Loader 1
Includes Next Executable validation code
External Secure Boot Code (ESBC)
3
Barker Code
Public Key List
Signature
Image Pointer
Next Executable (ESBC Image)
Includes Next Executable validation code
External Secure Boot Code (ESBC)
3

Figure 26. Chain of Trust

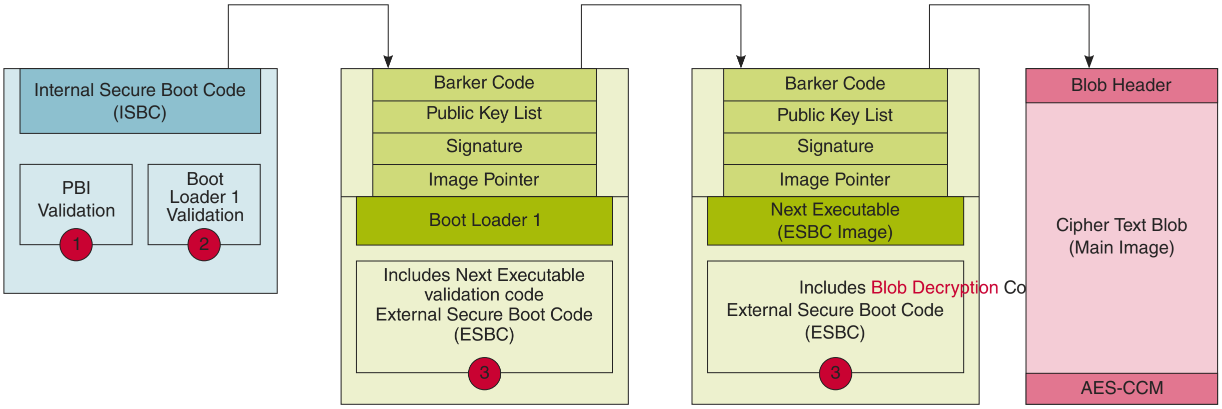

Internal Secure Boot Code (ISBC)
PBI Validation
1
Boot Loader 1 Validation
2
Barker Code
Public Key List
Signature
Image Pointer
Boot Loader 1
Includes Next Executable validation code External Secure Boot Code (ESBC)
3
Barker Code
Public Key List
Signature
Image Pointer
Next Executable (ESBC Image)
Includes Blob Decryption Co
External Secure Boot Code (ESBC)
3
Blob Header
Cipher Text Blob (Main Image)
AES-CCM

Figure 27. Chain of Trust with confidentiality

The validated ESBC image is allowed to use the One Time Programmable Master Key to decrypt system secrets. Cryptographic blob mechanism is used to establish Chain of trust with confidentiality.

The above is explained in detail in coming sections.

As depicted in figure(s) above, there are three types of images which need to be validated as part of Secure Boot.

1. PBI image by ISBC
2. Boot Loader 1 image by ISBC
3. Next level image(s) by ESBC

Typically ESBC images would include:

Boot Loader 2 - n In case Boot Loader is split in to multiple stages (Typical example is in case of NAND, SD Boot in which there is a mini-boot loader loaded on OCRAM which is Boot Loader stage 1 verified by ISBC. Boot Loader 2 is loaded on DDR, which must be validated by the Boot Loader 1.

MC/AIOP images Management Complex images

LINUX The operating system to be executed on the SoC.

6.1.2.1.1 Secure Boot process

Secure boot process uses a digital signature validation routine already present in ISBC residing in SP 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.

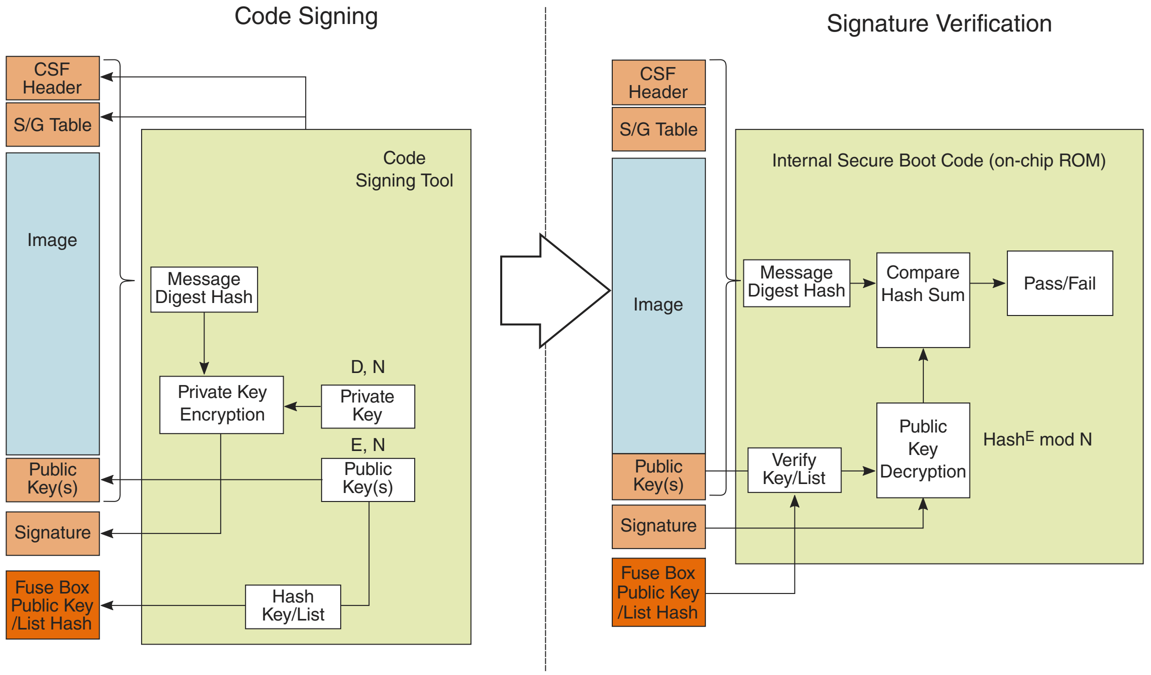


Figure 28. Secure Boot Process

As shown in the left side of the figure, the Code Signing Tool adds the following:

CSF Header Command Sequence File Header

This header provides the ISBC with flags, pointers, offsets, and lengths necessary to perform image validation.

S/G Table Scatter Gather Table

Optional (N/A for some stages which support only single image)

Allows support for multiple non-contiguous images.

Public Key list SRK (Super Root Key) Table

One or more public keys is appended to the image. The CSF header indicates which of the keys is to be used in signature validation.

Signature The SHA-256 hash of the CSF header + S/G table + Image + Public Key(s), encrypted with a RSA private key corresponding to one of the public keys in the key list.

As part of the code signing process, the CST also supports:

Generating RSA public and private key pairs The RSA private key is exported for the OEM to store securely

The CST also supports using public & private keys input by the OEM

Hashing the public key or public key list This hash becomes the Super Root Key Hash which is stored in the SFP

At a high level, the secure boot process runs code signing in reverse.

1. The ISBC locates the CSF header and S/G table to further locate the image, public key list, and signature
2. The public key (list) is hashed and compared to the SRKH
3. If the public key is good, it is used to decrypt the signature (recover the hash)
4. The CSF header + S/G table + Image + Public key list are hashed, with the result compared to above. If the two hashes match, image is considered to be authentic.

6.1.2.1.1.1 Super Root Key (SRK)

These are RSA public and private key pairs. Private keys are used to sign the images and public keys are used to validate the image during ISBC and ESBC phase.

Public keys are embedded in the header and the hash of SRK table is fused in SRKH register of SFP.

These are Hardware Bound Keys, once the hash is fused the public private key pair can not be modified.

Keys of sizes 1k, 2k and 4k are supported in NXP 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.

Key Revokation Trust Architecture provides 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. The SRK table supports maximum of 8 public keys and user has the option to revoke up to 7 keys.

During secure boot, 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.

6.1.2.2 ISBC Phase

At reset, Service Processor core is released and begins executing instructions from reset vector address 0x0 which is mapped to Internal Boot ROM. The Internal Boot ROM contains the code known as Internal Secure Boot Code (ISBC). The main steps in ISBC flow are defined below.

6.1.2.2.1 ISBC for PBI validation

1. **Sec_Mon check**: 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.
2. **PBI command check**: Verify that the first PBI command is ‘LOAD SEC HDR’. If not found, an error is raised.
3. **Valid header check**: Check for a valid preamble and correct B0/1 flag set as 0 in the header. If not, an error is raised.
4. **CSF parsing and public key check**; If ISBC finds a valid CSF header, it parses the CSF header to locate the public key from SRK (Super Root Key) table to be used to validate the code. There can be a table of maximum 8 public keys present in the header. The Secure Fuse Processor doesn’t actually store a public key, it stores a SHA-256 hash of the table. If the hash of the SRK table fails to match the stored hash, secure boot fails.
5. **Signature validation**: With the validated public key, ISBC decrypts the digital signature stored with the CSF header. The ISBC then uses the PBI length field in the RCW to calculate a hash over all PBI commands (CSF header is also a part of PBI commands) along with the SRK table. Optional flags in the CSF header tell the ISBC whether the FSL Unique ID and the OEM Unique ID (in the Secure Fuse Processor) are to be checked or not. Including these IDs allows the image to be bound to a single platform. If the decrypted hash and generated hash don ’ t match, secure boot fails.
6. **Sec_Mon Transition**: If ISS (Increment Security State) flag is set in the header, transition the SNVS state from Check to Trusted.

NOTE

1. PBI commands in Secure Boot must have a command ‘**Load Boot 1 CSF Header Ptr**’ to inform the ISBC about location of CSF Header for BOOT1 image (ESBC).
2. Boot1 image and header must be placed on an XIP memory before execution of next phase (ISBC validation of Boot1/ESBC). If these images are placed on memories like NAND/SD/eMMC, then they must be copied to an XIP memory like OCRAM, DDR via PBI commands.

6.1.2.2.2 ISBC for Boot1 (Boot Loader 1) validation

1. **Valid header check**: Check for a valid preamble and correct B0/1 flag set as 1 in the header. If not, an error is raised.
2. **CSF parsing and public key check**: If ISBC finds a valid CSF header, it parses the CSF header to locate the public key from SRK (Super Root Key) table to be used to validate the code. There can be a table of maximum 8 public keys present in the header. The Secure Fuse Processor doesn't actually store a public key, it stores a SHA-256 hash of the table. If the hash of the SRK table fails to match the stored hash, secure boot fails.
3. **Signature validation**: With the validated public key, ISBC decrypts the digital signature stored with the CSF header. The image information is stored in a SG (scatter gather) table with support of up to 8 discrete images. The ISBC calculates a hash over the CSF header, SRK table, SG table and all entries in SG table (i.e. images). Optional flags in the CSF header tell the ISBC whether the FSL Unique ID and the OEM Unique ID (in the Secure Fuse Processor) are to be checked or not. 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.
4. **Entry Point check**: One final check is performed by the ISBC. This check confirms that the Entry Point to be updated in Boot Location Pointer falls within one of the SG entries which have been validated by the ISBC.
5. **Sec_Mon Transition**: If ISS (Increment Security State) flag is set in the header, transition the SNVS state from Check to Trusted or Trusted (if transitioned in PB phase) to Secure.

NOTE

1. After End of ISBC, Entry Point parsed from header is written to Boot LOC PTR register.
2. GPP is waken up.
3. SP goes to sleep.

There are many reasons the ISBC could fail to validate the PBI or Boot1. Technicians with debug access can check the DCFG SCRATCHRW3 register to obtain an error code. For a list of error codes refer ISBC Validation Error Codes.

6.1.2.3 ESBC Phase

Unlike the ISBC, which is in an internal ROM and therefore unchangeable, the ESBC is reference code, and can be changed by OEMs. The remainder of this section is the description of a reasonable secure boot chain of trust based on NXP's reference software for secure boot. The reference ESBC code is also part of the Boot 1 image validated by ISBC and would be used to validate further ESBC images like MC, AIOP, and LINUX etc.

NXP provided ESBC consists of standard u-boot which has been signed using a private key. If the Boot Mode is secure, user can't reach to uboot prompt as the environment variable **bootdelay** is defined to 0.

There is default boot command for secure boot in the environment which executes on auto boot. This default **bootcmd** 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.

To establish the Secure Boot Chain of Trust, some U-Boot Commands have been added in the ESBC Code which will be discussed in detail in coming sections.

6.1.2.3.1 esbc_validate command

```
esbc_validate img_hdr [pub_key_hash]
```

Input arguments:

img_hdr – Location of CSF header of the image to be validated

pub_key_hash – hash of the public key used to verify the image. This is optional parameter. If not provided, code makes the assumption that the key pair used to sign the image is same as that used with ISBC. So the hash of the key in the header is checked against the hash available in SRK fuse for verification.

Description:

The command would do the following:

Perform CSF header validation on the address passed in the image header. During parsing of the header, image address in stored in an environment variable which is later used in source command in default secure boot command.

Signature checks on the image.

6.1.2.3.2 esbc_halt command

```
esbc_halt (no arguments)
```

Description:

This command puts core in spin loop.

6.1.2.3.3 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:

This command would create a cryptographic blob of the image placed at src location and place the blob at dst location.

6.1.2.3.4 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 a random number 16 bytes long(key modifier) is placed

Description:

This command would decapsulate the blob placed at src location and place the decapsulated data of expected size at dst location.

6.1.2.3.5 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.

1. Boot script can have any commands which u-boot supports. No checking is done 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 is 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.

Where to place the boot script?

ESBC u-boot expects the boot script to be loaded in flash . 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`

6.1.2.3.5.1 Chain of Trust

Boot script contains information about the next level images, e.g. MC, LINUX etc. ESBC validates these images as per their public keys. MC is started with validated MC images if required and finally bootm command is executed to pass control to LINUX 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. Figure below shows the Chain of Trust established for validation with this ESBC.

Sample Boot Script

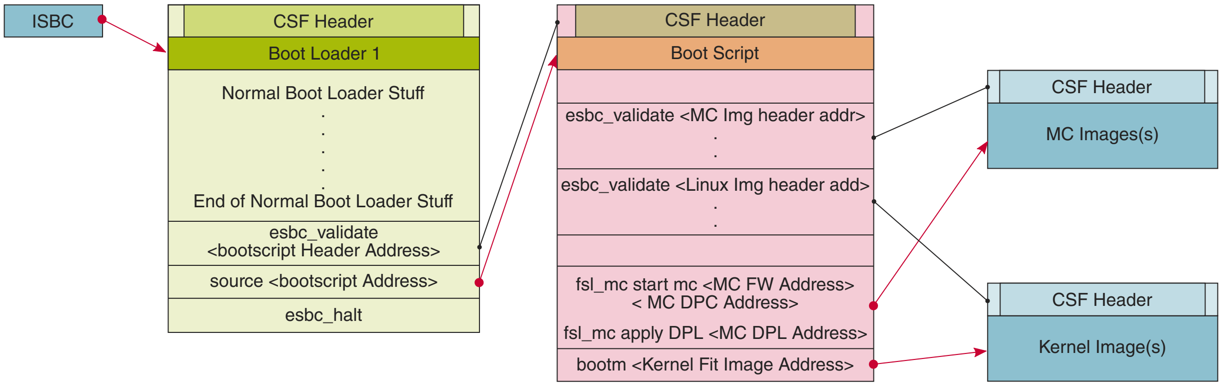

ISBC
CSF Header
Boot Loader 1
Normal Boot Loader Stuff
End of Normal Boot Loader Stuff
esbc_validate <bootscript Header Address>
source <bootscript Address>
esbc_halt
CSF Header
Boot Script
esbc_validate <MC Img header addr>
esbc_validate <Linux Img header add>
fsl_mc start mc <MC FW Address> < MC DPC Address>
fsl_mc apply DPL <MC DPL Address>
bootm <Kernel Fit Image Address>
CSF Header
MC Images(s)
CSF Header
Kernel Image(s)

Figure 29. Secure Boot Flow (Chain of Trust)

```
# Get Images and Headers on DDR
.
.
.
# Validate the Images. (<pub_key_hash> is optional)
esbc_validate <Image1 Header Address> <pub_key_hash>
esbc_validate <Image2 Header Address> <pub_key_hash>
.
.
.
# Start MC with validated images
fsl_mc start mc <MC FW Address> < MC DPC Address>
fsl_mc apply DPL <MC DPL Address>

# Boot the Linux
bootm <Kernel Fit Image Address>
```

6.1.2.3.5.2 Chain of Trust with confidentiality

To establish chain of trust with confidentiality, cryptographic blob mechanism can be used. In this chain of trust, validated image is allowed to use the One Time Programmable Master Key to decrypt system secrets. Two bootscripts are to be used. First encap bootscripts is used which creates a blob of the next level images(e.g. MC, LINUX etc.) and saves them on flash. After this the system is booted after replacing the encap bootscript with decap bootscript which decapsulates the blobs and start MC and LINUX.

Sample Encap Boot Script

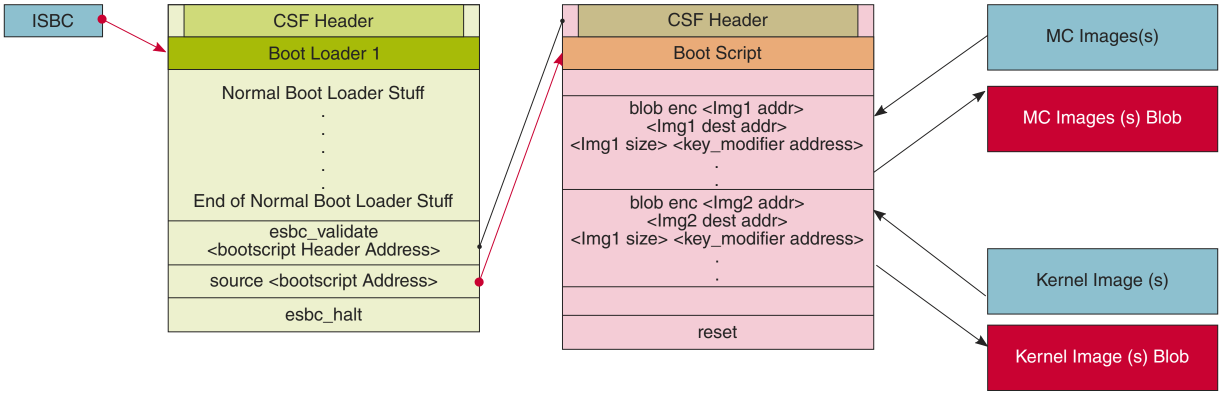

ISBC
CSF Header
Boot Loader 1
Normal Boot Loader Stuff
End of Normal Boot Loader Stuff
esbc_validate <bootscript Header Address>
source <bootscript Address>
esbc_halt
CSF Header
Boot Script
blob enc <Img1 addr> <Img1 dest addr> <Img1 size> <key_modifier address>
blob enc <Img2 addr> <Img2 dest addr> <Img1 size> <key_modifier address>
reset
MC Images(s)
MC Images (s) Blob
Kernel Image (s)
Kernel Image (s) Blob

Figure 30. Chain of Trust with Confidentiality (Encapsulation)

```
# Get Images on DDR
.
.
.
# Create the Blobs
blob enc <Img1 addr> <Img1 dest addr> <Img1 size> <key_modifier address>
blob enc <Img2 addr> <Img2 dest addr> <Img2 size> <key_modifier address>
blob enc <Img3 addr> <Img3 dest addr> <Img3 size> <key_modifier address>
.
.
.

Save The Blobs created on Flash
.
.
.

# End of Encap Boot Script (This is one time only and must be replaced with decap Boot Script)
```

Sample Decap Boot Script

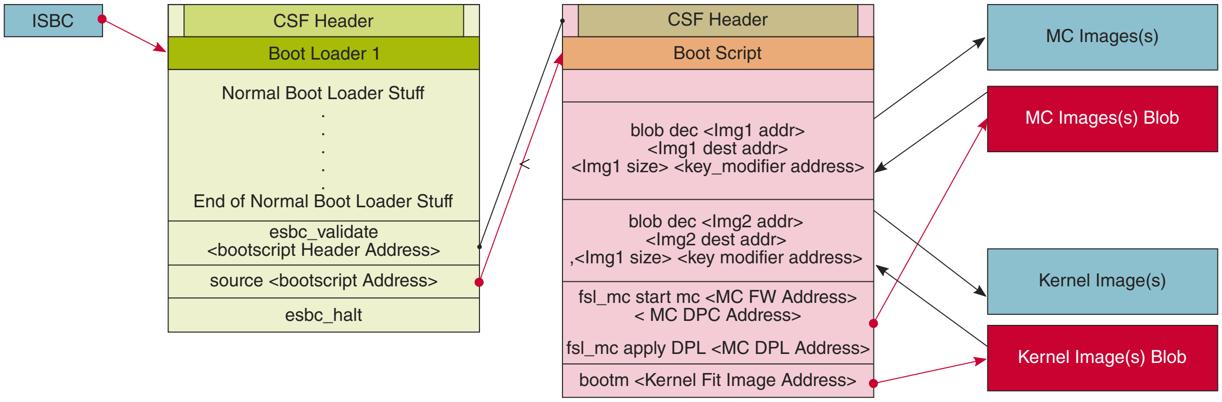

ISBC
CSF Header
Boot Loader 1
Normal Boot Loader Stuff
End of Normal Boot Loader Stuff
esbc_validate <bootscript Header Address>
source <bootscript Address>
esbc_halt
CSF Header
Boot Script
blob dec <Img1 addr> <Img1 dest addr> <Img1 size> <key_modifier address>
blob dec <Img2 addr> <Img2 dest addr> ,<Img1 size> <key modifier address>
fsl_mc start mc <MC FW Address> < MC DPC Address>
fsl_mc apply DPL <MC DPL Address>
bootm <Kernel Fit Image Address>
MC Images(s)
MC Images(s) Blob
Kernel Image(s)
Kernel Image(s) Blob

Figure 31. Chain of Trust with Confidentiality (Decapsulation)

```
# Get Images Blobs on DDR
.
.
.
# Decap the Blobs to get the actual images
```

```
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>
.
.
.
# Start MC with validated images
fsl_mc start mc <MC FW Address> < MC DPC Address>
fsl_mc apply DPL <MC DPL Address>

# Boot the Linux
bootm <Kernel Fit Image Address>
```

6.1.2.4 Next executable phase

The boot loader (ESBC) finishes the platform initialization and passed control to the Linux image. The boot chain can be further extended to be able to sign application which would be running on Linux prompt. Further RTIC can be integrated to verify memory regions using Security Engine (SEC) during run time.

6.1.2.5 Product Execution

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

Steps in the demo would be:

1. ISBC code would validate PBI and Boot Loader 1.
2. On Successful validation, PBI commands would be executed by SP BootROM.
3. Boot Loader 1 execution will begin on GPP.
4. The ESBC code in Boot Loader 1 would validate and execute and bootscript.[3]
5. The boot script would contain the commands to validate and execute next level images as described in Boot Script on page 215.

6.1.2.5.1 Introduction

Running secure boot (Chain of Trust

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

      Program the ITS fuse.

   b. **Flow B**

      For protyping phase, do not blow the ITS fuse, secure boot can be enabled by RCW with SB_EN = 1.
2. Blow other required fuses (TPMK and SRK Hash[4]) in the SFP in silicon. For more details regarding fuse blowing, CCS and Reset Pause, refer to *Platform Reference Manual* and *Trust Architecture User Guide*.

[3] In case the boot loader is split into two parts, the validation and execution of boot script would happen in the final boot loader i.e Boot Loader2. Boot Loader 1 will validate and transfer control to Boot Loader 2.

[4] 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. The SoC can be put in a **Reset Pause** state. This will pause the Reset State Machinery after RCW Loading. Then CCS can be connected via JTAG.

Write the SRK Hash value in SFP mirror registers and then get the system out of Reset Pause State.

NOTE

SRK hash in the fuse should be same as the hash of the key pair being used to sign the PBI and U-Boot image.

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.
   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 Bank1.
4. Give a power on cycle to the board.
   a. For Flow A and Flow B (If secure boot images flashed on default bank)
      - On power on, ISBC code in SP boot ROM would validate the PBI image followed by validation of Boot loader1 (U-Boot)
      - ESBC code in Boot loader 1 image would further validate the ESBC images (Boot Script, LINUX, MC, and so on)
      - MC and LINUX would be started.
   b. For Flow B (If secure boot images flashed on alternate bank), the user must first do the switch settings[5] for booting from alternate bank and also to enable reset pause.
      - On power on after the correct switch setting, Reset State Machinery will be paused after RCW loading.
      - Write the SRKH to SFP mirror registers and get the system out of Reset Pause via CCS.
      - 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

6.1.2.5.2 Chain of Trust with confidentiality

This section presents the steps to be followed in order to execute chain of trust with confidentiality.

The demo would be divided into two parts:

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

The execution steps remain same as specified above in Product Execution on page 218. In first phase the Boot Script would contain the commands to encrypt and create blobs of the images. After that the Boot Script is replaced and in second phase the Boot Script would contain commands to decrypt the blobs to get back the images and boot LINUX, AIOP using these images.

6.1.2.5.2.1 Other images required for demo

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

1. Encap boot script

   Sample Encap boot script

```
load \$devtype \$devnum:2 \$kernelheader_addr_r /secboot_hdrs/ls2088ardb/hdr_linux.out;
esbc_validate \$kernelheader_addr_r;
load \$devtype \$devnum:2 \$fdtheader_addr_r /secboot_hdrs/ls2088ardb/hdr_dtb.out;
```

[5] This may also be done via writing to FPGA registers from the U-Boot Prompt of U-Boot runing in Non-Secure Mode on Bank0. Refer the Platform FPGA guide for the same.

```
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 /ls2088ardb_dec_boot.scr;
echo replacing Bootscript; save \$devtype \$devnum:2 \$load_addr /ls2088ardb_boot.scr \
$filesize;size \$devtype \$devnum:2 /secboot_hdrs/ls2088ardb/hdr_ls2088ardb_bs_dec.out;
load \$devtype \$devnum:2 \$load_addr /secboot_hdrs/ls2088ardb/
hdr_ls2088ardb_bs_dec.out ;echo Replacing bootscript header; save \$devtype \$devnum:2 \
$load_addr /hdr_ls2088ardb_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-ls2088a-rdb.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 ;
```

6.1.2.5.2.2 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.

   b. **Flow B**

      For protyping phase, do not blow the ITS fuse, secure boot can be enabled by RCW with SB_EN = 1.
2. Blow other required fuses(OTPMK and SRK hash[6]) in the SFP in silicon. For more details regarding fuse blowing, CCS and Reset Pause, 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 PBI and U-Boot image.

   *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.

   b. If you are using **Flow B**, you can use alternate bank for demo purpose. This would mean flashing the images on alternate bank addresses from Bank0 and then switching to Bank4.
4. Give a power on cycle to the board.

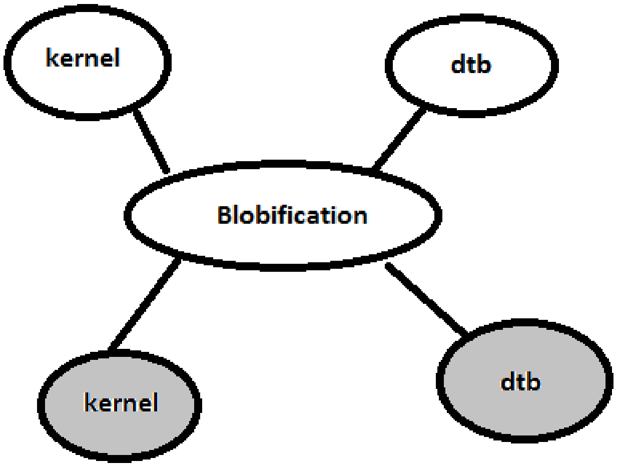
[6] Blowing of OTPMK is essential to run secure boot for both Production (Flow A) and Prototyping/ Development (Flow B).

For SRK Hash, in Development Mode (Flow B), there is a workaround to avoid blowing fuses. The SoC can be put in a **Reset Pause** state. This will pause the Reset State Machinery after RCW Loading. Then CCS can be connected via JTAG.

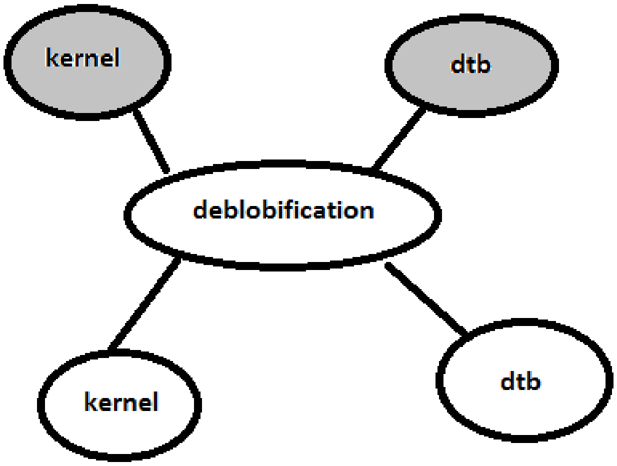
Write the SRK Hash value in SFP mirror registers and then get the system out of Reset Pause State.

a. For Flow A and Flow B (If Secure Boot images flashed on default bank)

- On power on, ISBC code in SP Boot ROM would validate the PBI image followed by validation of Boot Loader1 (U-Boot)
- First Boot: Encapsulaton Step (Should happen in OEM's premises)
    i. By defult the enacap and decap bootscripts will be installed in the bootpartition.
    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. For Flow B (If Secure Boot images flashed on alternate bank), the user must first do the switch settings[7] for booting from alternate bank and also to enable Reset Pause.

   - On power on after the correct switch setting, Reset State Machinery will be paused after RCW loading.
   - Write the SRKH to SFP mirror registers and get the system out of reset pause via CCS.
   - Secure Boot flow as mentioned above would execute.

6.1.2.5.3 ISBC Key Extension (IE)

The ISBC Key Extension feature allows the user to extend the number of keys available for signature validation. To use this feature, ISBC validates a table of **public** keys (IE keys) along with the ESBC image using one of the SRKs (Super Root Keys). These validated IE keys later become available for ESBC to validate downstream images, for example, Linux, MC, and so on.

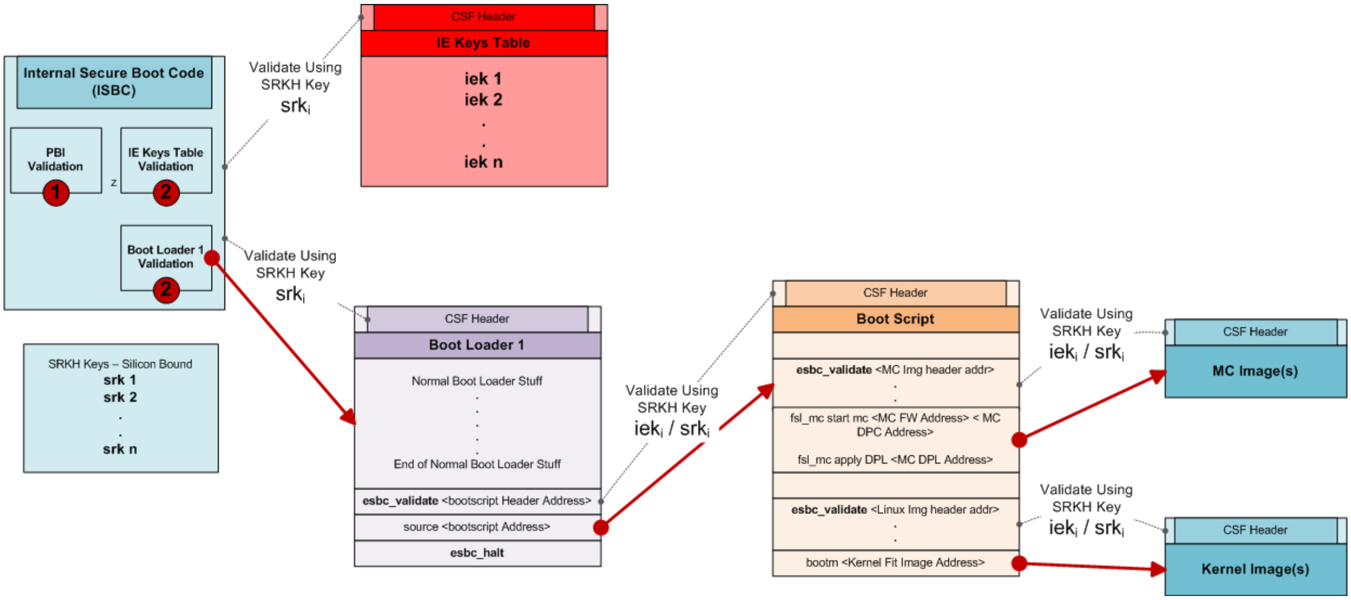

CSF Header
IE Keys Table
iek 1
iek 2
.
.
iek n
Internal Secure Boot Code (ISBC)
PBI Validation
1
z
IE Keys Table Validation
2
Boot Loader 1 Validation
2
Validate Using SRKH Key srk_i
Validate Using SRKH Key srk_i
SRKH Keys – Silicon Bound
srk 1
srk 2
.
.
srk n
CSF Header
Boot Loader 1
Normal Boot Loader Stuff
End of Normal Boot Loader Stuff
esbc_validate <bootscript Header Address>
source <bootscript Address>
esbc_halt
Validate Using SRKH Key iek_i / srk_i
CSF Header
Boot Script
esbc_validate <MC Img header addr>
fsl_mc start mc <MC FW Address> < MC DPC Address>
fsl_mc apply DPL <MC DPL Address>
esbc_validate <Linux Img header addr>
bootm <Kernel Fit Image Address>
Validate Using SRKH Key iek_i / srk_i
Validate Using SRKH Key iek_i / srk_i
CSF Header
MC Image(s)
CSF Header
Kernel Image(s)

Figure 32. Secure boot flow (with IE)

[7] This may also be done via writing to FPGA registers from the U-Boot Prompt of U-Boot runing in Non-Secure Mode on Bank0. Please refer the Platform FPGA guide for the same.

As shown in the Secure Boot Flow (with IE),

1. ISBC code validates IE Table and Boot loader (ESBC) using one of the SRKs (Silicon Bound).
2. After IE Table is successfully validated, it becomes available for use by future images.[8](SRKs are present in SRK table which is supplied with each image that is signed and validated using them. They are silicon bound as their hash, SRKH, is present in SoC's fuses.)
3. ESBC validates further images using either SRKs or IE Keys, which in turn can validate other images using these keys. Which key to use to verify image is decided based on CSF Header of the image to be validated.

Following are the advantages of IE Keys.

- Different images can be signed and validated using different IE keys. This is not possible with SRKs as they are significantly less in number than IE Keys.
- IE key table can be changed as opposed to SRKs which are fixed as they are tied to SRKH blown in fuses.

6.1.2.5.3.1 IE table format

Table 62. Table: IE table format

Offset (In Bytes)	Description
0x0 - 0x3	Keys revoked. Each bit represents a key. If the bit is set, it indicates that this key in the table has been revoked and cannot be used for verification purposes. Bit 0 represents Key 1, Bit 1 represent Key 2 ..., Bit 31 represents Key 32.
0x04-0x07	Total number of keys present in the table (Max 32).
0x08-0x0b	Key 1 length (In bytes. Max 1024)
0x0c-0x40b	Key 1 value (Zero padded if less than 1024 bytes)
0x40c-0x40f	Key 2 length
0x410-0x80f	Key 2 value
.....	....
0x8 + 0x404 * (n - 1) - 0x8 + 0x404 8 (n - 1) + 0x3	Key N length (N <= 32)
0x8 + 0x404 * (n - 1) + 0x4 - 0x8 + 0x404 * (n) - 0x1	Key N value

6.1.2.5.3.2 Enabling IE via the Code Signing Tool

Following are the steps that need to be executed during **runtime** to enable use of IE feature during secure boot:

1. During the PBI phase, scratch registers 13 and 14 (in the DCFG block) need to be populated with the address of the IE table (to be loaded later). Scratch register 14 contains the higher 32 bits of address of the IE table and scratch register 13 contains the lower 32 bits of address (SoC supports 48-bit address space).
2. Next, during the ISBC runtime, the IE table needs to be validated along with the ESBC image. For this, the address of this IE table is added as the first entry in SG table of CSF header.

[8] For allowing ESBC image to use IE Table, it is neccesary for the IE Table to be present in an XIP memory. This is trivial in case of XIP memories, such as NOR. In case of Non - XIP memories, IE table needs to be copied along with CSF Header to an XIP memory.

3. Now ESBC can use either one of the IE keys or the SRK keys to validate next images. Which key to use, depends on the various field of CSF header (CSF header structure definition on page 238CSF header structure definition on page 245) of the image to be validated.
    a. If IE Flag is not set, then one of the SRKs is used to verify the image. The key number is taken from the "Key No. for verification" flag. All SRKs would be present in the SRK table, whose offset is embedded in CSF header.
    b. If IE Flag is set, then one of the IE keys is used to verify the image. The key number is taken from the "IE Key Select" flag. The key is gathered from IE table, the address of which was already populated in Scratch registers 13 and 14.

Following are the steps that need to be executed during **build-time** to incorporate IE table in CSF header and sign images using IE keys using the Code Signing Tool:

1. Using uni_sign, sign ESBC image and IE table (uni_sign also generates IE table using IE keys). uni_sign also prints the IE table address.
2. Using uni_pbi, generate PBI commands which also includes the commands to write the address of IE table given by uni_sign to Scratch registers 13 and 14.
3. Sign next level images using one of the IE keys / SRKs.

6.1.2.5.3.2.1 uni_sign on ESBC

Sample input file for uni_sign

```
/* Copyright (c) 2015 Freescale Semiconductor, Inc.
 * Freescale Proprietary.
 */

---------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform -
# TRUST 3.1: LS2088, LS1088
PLATFORM=<platform>
---------------------------------------------------
# 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=30100000
---------------------------------------------------
# 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.x): (between 1 to 8)
KEY_SELECT=1
# PRI_KEY [Mandatory] Single Key Used for Signing
# USAGE: <srk.pri>
PRI_KEY=srk.pri
# IE_KEY [Mandatory] Comma Seperated List
#IE_REVOC [Optional] Comma Seperated List
IE_KEY=iekey1.pub,iekey2.pub,iekey3.pub
#ESBC_HDRADDR [Mandatory] 32bit Address of ESBC Header
#Used to Calculate IE Table Address
ESBC_HDRADDR=30020000
---------------------------------------------------
# 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-dtb.bin,30100000,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
```

Table 63. Description of fields in input file

Field in input file	Description
IE_KEY	Comma seperated list of names of files containing public keys (IE keys)
IE_REVOC	Comma seperated list of key numbers that are to be revoked from IE table
ESBC_HDRADDR	32-bit address where generated header will be placed

The above file is an example of input file for running uni_sign on ESBC image to enable use of IE table. Presence of field 'IE_KEY' indicates to the CST tool to embed IE table in the output header file. Field 'ESBC_HDRADDR' is needed to calculate the address of IE table. It states the address at which the output of uni_sign (output header file) will be placed on memory map. The 'PUB_KEY=srk.pub' and 'PRI_KEY=srk.pri' fields indicates that SRK key needs to be used to sign both ESBC and IE table.

Note: Here ESBC_HDRADDR needs to be 32-bit address as ISBC can only access 32-bit addresses.

Output of uni_sign

```
$ ./uni_sign --verbose input_files/uni_sign/<platform>/ie_keys/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>/ie_keys/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 : 2
-     Entry Point : 30100000
-     Entry 1 : ie_table.out (Size = 00000c14 SRC = 30020a00 DST = ffffffff)
-     Entry 2 : u-boot-dtb.bin (Size = 00090c16 SRC = 30100000 DST = ffffffff)
- RSA Signature Information
-     RSA Offset : 1800
-     RSA Size : 80
-----------------------------------------------

Image Hash:
fa2a5b7ba63375f18c94f2911687750e0e49c27eaf0c313617ed3ebfc006c6e6
IE Table Absolute Address: 30020a00


************************************************
* Header File is with Signature appended
************************************************

Header File Created: hdr_uboot.out

SRK (Public Key) Hash:
c9f90c9762e1693804421a4bd8193735b38ea03b83303d9529f7b1b5d2e4688f
     SFP SRKHR0 = c9f90c97
     SFP SRKHR1 = 62e16938
     SFP SRKHR2 = 04421a4b
     SFP SRKHR3 = d8193735
     SFP SRKHR4 = b38ea03b
     SFP SRKHR5 = 83303d95
     SFP SRKHR6 = 29f7b1b5
     SFP SRKHR7 = d2e4688f
```

As shown, the output prints the absolute address of IE Table. This address is equal to 'ESBC_HDRADDR + offset of IE Table within output header file' as IE table is embedded in the output header file by uni_sign.

Note: uni_sign needs to be executed with --verbose option to get the 'IE Table Absolute Address'

Partial view of hdr_uboot.out generated by uni_sign (In Hex)

```
......
......
0000800: 140c 0000 0000 0000 000a 0230 ffff ffff  ...........0....
0000810: 160c 0900 0000 0000 0000 1030 ffff ffff  ...........0....
......
......
0000a00: 0000 0000 0300 0000 0001 0000 cf9d 8fd0  ................
0000a10: 57a4 c7be e519 6ae1 b4db 2e97 8c79 9d3f  W....j......y.?
0000a20: 61a2 5538 f1af 58c0 31cf 3484 a54d 343a  a.U8..X.1.4..M4:
0000a30: 58ec 02ec e5e2 153f 9843 b0e0 7c0b f9b2  X......?.C..|...
0000a40: 76fa f87d 8780 98bd cf87 6079 3b34 87ad  v..}......`y;4..
```

```
0000a50: c9d3 6fe9 71fd 884d f531 be13 2d83 dfaa  ..o.q..M.1..-...
0000a60: 0ca1 0f9f 8ae1 5312 457a 251d 3fd2 0127  ......S.Ez%.?..'
0000a70: de51 3f3e 2da7 ae17 b203 42db a495 6c7a  .Q?>-.....B...lz
0000a80: d2b9 5671 bb77 b32d 2a62 9045 0000 0000  ..Vq.w.-*b.E....
......
......
```

As shown above, SG table is present at offset 0x800 in the hdr_uboot.out. There are two SG entries present. The first entry is the IE table (at address 0x30020a00). The second entry is the U-Boot image. Also, at 0xa00 offset of hdr_uboot.out, the IE table is present with 3 keys.

6.1.2.5.3.2.2 uni_sign On ESBC

Sample Input File for uni_sign

```
/* Copyright (c) 2015 Freescale Semiconductor, Inc.
 * Freescale Proprietary.
 */

---------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform -
# TRUST 3.0: LS2085
# TRUST 3.1: LS2088, LS1088
PLATFORM=<platform>
---------------------------------------------------
# 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=20100000
---------------------------------------------------
# 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.x): (between 1 to 8)
KEY_SELECT=1
# PRI_KEY [Mandatory] Single Key Used for Signing
# USAGE: <srk.pri>
PRI_KEY=srk.pri
# IE_KEY [Mandatory] Comma Seperated List
#IE_REVOC [Optional] Comma Seperated List
IE_KEY=iekey1.pub,iekey2.pub,iekey3.pub
#ESBC_HDRADDR [Mandatory] 32bit Address of ESBC Header
#Used to Calculate IE Table Address
ESBC_HDRADDR=20c00000
---------------------------------------------------
# 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-dtb.bin,20100000,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
```

Table 64. Description of Fields in Input File

Field in Input File	Description
IE_KEY	Comma seperated list of names of files containing public keys (IE Keys)
IE_REVOC	Comma seperated list of key number that are to be revoked from IE Table
ESBC_HDRADDR	32 Bit address where generated header shall be placed

The above file is an example of input file for running uni_sign on ESBC image to enable use of IE Table. Presence of field 'IE_KEY' indicates to the CST tool to embed IE Table in the output header file. Field 'ESBC_HDRADDR' is needed to calculate address of IE Table. It states the address at which the output of uni_sign (output header file) will be placed on memory map. 'PUB_KEY=srk.pub' and 'PRI_KEY=srk.pri' indicates that SRK Key needs to be used to sign both ESBC and IE Table.

Note: Here ESBC_HDRADDR needs to be 32 bit address as ISBC can only access 32 bit addresses.

Output of uni_sign

```
$ ./uni_sign --verbose input_files/uni_sign/ls2088_1088/ie_keys/qspi/input_uboot_qspi_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/ls2088_1088/ie_keys/qspi/input_uboot_qspi_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 : 2
-     Entry Point : 30100000
-     Entry 1 : ie_table.out (Size = 00000c14 SRC = 30020a00 DST = ffffffff)
-     Entry 2 : u-boot-dtb.bin (Size = 00090c16 SRC = 30100000 DST = ffffffff)
- RSA Signature Information
-     RSA Offset : 1800
-     RSA Size : 80
-----------------------------------------------

Image Hash:
fa2a5b7ba63375f18c94f2911687750e0e49c27eaf0c313617ed3ebfc006c6e6
IE Table Absolute Address: 410c00a00


************************************************
* Header File is with Signature appended
************************************************

Header File Created: hdr_uboot.out

SRK (Public Key) Hash:
c9f90c9762e1693804421a4bd8193735b38ea03b83303d9529f7b1b5d2e4688f
     SFP SRKHR0 = c9f90c97
     SFP SRKHR1 = 62e16938
     SFP SRKHR2 = 04421a4b
     SFP SRKHR3 = d8193735
     SFP SRKHR4 = b38ea03b
     SFP SRKHR5 = 83303d95
     SFP SRKHR6 = 29f7b1b5
     SFP SRKHR7 = d2e4688f
```

As shown, the output prints the absolute address of IE Table. This address is equal to 'ESBC_HDRADDR + offset of IE Table within output header file' as IE Table is embedded in the output header file by uni_sign.

Note: uni_sign needs to be executed with --verbose option to get the 'IE Table Absolute Address'

Partial view of hdr_uboot.out generated by uni_sign (In Hex)

```
......
......
0000800: 140c 0000 0000 0000 000a 0230 ffff ffff  ...........0....
0000810: 160c 0900 0000 0000 0000 1030 ffff ffff  ...........0....
......
......
0000a00: 0000 0000 0300 0000 0001 0000 cf9d 8fd0  ................
0000a10: 57a4 c7be e519 6ae1 b4db 2e97 8c79 9d3f  W.....j......y.?
0000a20: 61a2 5538 f1af 58c0 31cf 3484 a54d 343a  a.U8..X.1.4..M4:
0000a30: 58ec 02ec e5e2 153f 9843 b0e0 7c0b f9b2  X......?.C..|...
0000a40: 76fa f87d 8780 98bd cf87 6079 3b34 87ad  v..}......`y;4..
0000a50: c9d3 6fe9 71fd 884d f531 be13 2d83 dfaa  ..o.q..M.1..-...
0000a60: 0ca1 0f9f 8ae1 5312 457a 251d 3fd2 0127  ......S.Ez%.?..'
0000a70: de51 3f3e 2da7 ae17 b203 42db a495 6c7a  .Q?>-.....B...lz
0000a80: d2b9 5671 bb77 b32d 2a62 9045 0000 0000  ..Vq.w.-*b.E....
......
......
```

As shown above, at offset 0x800 in the hdr_uboot.out, SG Table is present. There are two SG Entries present there. The first entry is the IE Table (at address 0x30020a00). The second entry is the u-boot image. Also, at 0xa00 offset of hdr_uboot.out, the IE Table is present with 3 Keys.

6.1.2.5.3.2.3 uni_pbi

Sample input file for uni_pbi

```
/* Copyright (c) 2015 Freescale Semiconductor, Inc.
 * Freescale Proprietary.
 */

---------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform -
# TRUST 3.1: LS2088, LS1088
PLATFORM=<platform>
---------------------------------------------------
# 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.x): (between 1 to 8)
KEY_SELECT=1
# PRI_KEY [Mandatory] Single Key Used for Signing
# USAGE: <srk.pri>
PRI_KEY=srk.pri
#IE_TABLE_ADDR [optional] Used by Uboot to get IE Table Address
#Can be 64 bit
IE_TABLE_ADDR=580020a00
---------------------------------------------------
# For PBI Signing
# Name of RCW + PBI file [Mandatory]
RCW_PBI_FILENAME= rcw.bin
# Address of ISBC (Boot1) CSF Header [Mandatory]
BOOT1_PTR=30020000
---------------------------------------------------
# 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=rcw_sec.bin
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
```

Table 65. Description of fields in input file

Field in Input File	Description
IE_TABLE_ADDR	64 bit address of IE Table

As shown above, the field 'IE_TABLE_ADDR' is needed in the input file to uni_pbi. This field indicates the address of IE table, the address which is written to Scratch registers 13 and 14.

Note: This address (0x580020a00) is different from the address generated from uni_sign (0x30020a00) (though both address maps to same address. Refer to the memory map of the respective SoC). This is because ISBC can only access 32-bit addresses and U-Boot uses only 48-bit addresses for NOR mapping because U-Boot does not make MMU table entries for the 32-bit addresses of NOR. So uni_sign needs 32-bit address so that ISBC can access this address to validate IE table and uni_pbi needs 48-bit address so that U-Boot can access the address written in the Scratch register 13 and 14 to fetch the keys in IE table.

Output of uni_pbi

```
$ ./uni_pbi input_files/uni_pbi/<platform>/ie_keys/input_pbi_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_pbi/<platform>/ie_keys/input_pbi_nor_secure


************************************************
* Header File is with Signature appended
************************************************

Header File Created: rcw_sec.bin

SRK (Public Key) Hash:
c9f90c9762e1693804421a4bd8193735b38ea03b83303d9529f7b1b5d2e4688f
     SFP SRKHR0 = c9f90c97
     SFP SRKHR1 = 62e16938
     SFP SRKHR2 = 04421a4b
     SFP SRKHR3 = d8193735
     SFP SRKHR4 = b38ea03b
     SFP SRKHR5 = 83303d95
     SFP SRKHR6 = 29f7b1b5
     SFP SRKHR7 = d2e4688f
```

Partial view of rcw_sec.bin generated by uni_pbi (In Hex)

```
0000000: 55aa 55aa 0000 1080 2038 3038 3800 3838  U.U..... 8088.88
0000010: 0000 0000 0000 0000 0000 0000 0000 2000  .............. .
0000020: 0000 2000 0000 0000 802d e102 8025 0000  .. ......-...%..
0000030: 0000 0000 0000 0000 0b0e 0000 0000 0000  ................
0000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000070: 0070 0200 0000 0000 0000 2a41 0000 0000  .p........*A....
0000080: 0000 0000 0000 0000 b8b3 1bdf 0000 2080  .............. .
0000090: 1219 2001 0002 0000 0101 0020 0000 0000  .. ........ ....
00000a0: 0008 0000 8000 0000 0000 0000 0000 0000  ................
00000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000e0: 0000 2280 0000 0230 3002 e030 000a 0280   .."....00..0....
00000f0: 3402 e030 0500 0000 0404 e030 0000 0000  4..0.......0....
0000100: 0004 e030 0000 1030 0000 0833 0000 0000  ...0...0...3....
```

```
0000110: 0005 0833 d704 0000 2006 e030 0000 00a0  ...3.... ..0....
.....
.....
```

As shown above, the highlighted part are the PBI commands inserted in the rcw_sec.bin to write IE table address (64 bits) to Scratch Registers 13 and 14.

6.1.2.5.3.2.4 uni_pbi

Sample Input File for uni_pbi

```
/* Copyright (c) 2015 Freescale Semiconductor, Inc.
 * Freescale Proprietary.
 */

---------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform -
# TRUST 3.0: LS2085
# TRUST 3.1: LS2088, LS1088
PLATFORM=<platform>
---------------------------------------------------
# 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.x): (between 1 to 8)
KEY_SELECT=1
# PRI_KEY [Mandatory] Single Key Used for Signing
# USAGE: <srk.pri>
PRI_KEY=srk.pri
#IE_TABLE_ADDR [optional] Used by Uboot to get IE Table Address
#Can be 64 bit
IE_TABLE_ADDR=410c00a00
---------------------------------------------------
# For PBI Signing
# Name of RCW + PBI file [Mandatory]
RCW_PBI_FILENAME= rcw.bin
# Address of ISBC (Boot1) CSF Header [Mandatory]
BOOT1_PTR=20c00000
---------------------------------------------------
# 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=rcw_sec.bin
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
```

Table 66. Description of Fields in Input File

Field in Input File	Description
IE_TABLE_ADDR	64 bit address of IE Table

As shown above, in the input file to uni_pbi, the field 'IE_TABLE_ADDR' is needed. This field indicates the address of IE Table, the address which is written to Scratch registers 13 and 14.

Note: This address (0x580020a00) is different from the address generated from uni_sign (0x30020a00) (though both address maps to same address. Refer to the memory map of the respective SoC). This is because ISBC can only access 32 bit addresses and u-boot uses only 48 bit addresses for NOR mapping because u-boot doesn't make MMU Table entries for the 32 bit addresses of NOR. So uni_sign needs 32 bit address so that ISBC can access this address to validate IE Table and uni_pbi needs 48 bit address so that u-boot can access the address written in the Scratch register 13 and 14 to fetch the keys in IE Table.

Output of uni_pbi

```
$ ./uni_pbi input_files/uni_pbi/ls2088_1088/ie_keys/qspi/input_pbi_qspi_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_pbi/ls2088_1088/ie_keys/qspi/input_pbi_qspi_secure


************************************************
* Header File is with Signature appended
************************************************

Header File Created: rcw_sec.bin

SRK (Public Key) Hash:
c9f90c9762e1693804421a4bd8193735b38ea03b83303d9529f7b1b5d2e4688f
     SFP SRKHR0 = c9f90c97
     SFP SRKHR1 = 62e16938
     SFP SRKHR2 = 04421a4b
     SFP SRKHR3 = d8193735
     SFP SRKHR4 = b38ea03b
     SFP SRKHR5 = 83303d95
     SFP SRKHR6 = 29f7b1b5
     SFP SRKHR7 = d2e4688f
```

Partial view of rcw_sec.bin generated by uni_pbi (In Hex)

```
0000000: 55aa 55aa 0000 1080 2038 3038 3800 3838  U.U..... 8088.88
0000010: 0000 0000 0000 0000 0000 0000 0000 2000  .............. .
0000020: 0000 2000 0000 0000 802d e102 8025 0000  .. ......-...%..
0000030: 0000 0000 0000 0000 0b0e 0000 0000 0000  ................
0000040: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000050: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000060: 0000 0000 0000 0000 0000 0000 0000 0000  ................
0000070: 0070 0200 0000 0000 0000 2a41 0000 0000  .p........*A....
0000080: 0000 0000 0000 0000 b8b3 1bdf 0000 2080  .............. .
0000090: 1219 2001 0002 0000 0101 0020 0000 0000  .. ........ ....
```

```
00000a0: 0008 0000 8000 0000 0000 0000 0000 0000  ................
00000b0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000e0: 0000 2280 0000 0230 3002 e030 000a 0280   .."....00..0....
00000f0: 3402 e030 0500 0000 0404 e030 0000 0000  4..0.......0....
0000100: 0004 e030 0000 1030 0000 0833 0000 0000  ...0...0...3....
0000110: 0005 0833 d704 0000 2006 e030 0000 00a0  ...3.... ..0....
.....
.....
```

As shown above, the highlighted part are the PBI commands inserted in the rcw_sec.bin to write IE Table address (64 bits) to Scratch Registers 13 and 14.

6.1.2.5.3.2.5 uni_sign on next level images

Sample input file for uni_sign for signing Linux

```
/* Copyright (c) 2015 Freescale Semiconductor, Inc.
 * Freescale Proprietary.
 */

ESBC=1
---------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform -
# TRUST 3.1: LS2088, LS1088
PLATFORM=<platform>
---------------------------------------------------
# Specify the Key Information.
# PUB_KEY [Mandatory] Comma Seperated List
# Usage: <srk1.pub> <srk2.pub> .....
PUB_KEY=iekey3.pub
# IE_KEY_SEL [Mandatory]
# USAGE (for TRUST 3.x): (between 1 to 32)
IE_KEY_SEL=3
# PRI_KEY [Mandatory] Single Key Used for Signing
# USAGE: <srk.pri>
PRI_KEY=iekey3.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={kernel.itb,a1100000,fffffff}
---------------------------------------------------
# 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_kernel.out
IMAGE_HASH_FILENAME=
RSA_SIGN_FILENAME=
---------------------------------------------------
# Specify VERBOSE as 1, if you want to Display Header Information [Optional]
VERBOSE=0
```

Table 67. Description of fields in input file

Field in Input File	Description
IE_KEY_SEL	Number of the key in IE Table that is to be used to validate image

As shown above, in the input file to uni_sign, the 'IE_KEY_SEL' field indicates that a key from IE table is being used and it also indicates which key number from IE table needs to be used to validate this image during runtime. For build-time, CST tool needs the IE private key. This is indicated via the 'PRI_KEY' field.

Output of uni_sign

```
$ ./uni_sign input_files/uni_sign/<platform>/ie_keys/input_kernel_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>/ie_keys/input_kernel_secure

SRK/Public Key Hash not calculated.. IE = 1

************************************************
* Header File is with Signature appended
************************************************

Header File Created: hdr_kernel.out
SRK (Public Key) Hash Not Available
```

As shown above, SRK hash is not calculated as IE key is being used.

6.1.2.5.3.2.6 uni_sign on Next Level Images

Sample Input File for uni_sign for signing Linux

```
/* Copyright (c) 2015 Freescale Semiconductor, Inc.
 * Freescale Proprietary.
 */

ESBC=1
---------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform -
# TRUST 3.0: LS2085
# TRUST 3.1: LS2088, LS1088
PLATFORM=<platform>
---------------------------------------------------
# Specify the Key Information.
# PUB_KEY [Mandatory] Comma Seperated List
# Usage: <srk1.pub> <srk2.pub> .....
PUB_KEY=iekey3.pub
# IE_KEY_SEL [Mandatory]
# USAGE (for TRUST 3.x): (between 1 to 32)
IE_KEY_SEL=3
# PRI_KEY [Mandatory] Single Key Used for Signing
# USAGE: <srk.pri>
PRI_KEY=iekey3.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={kernel.itb,a1100000,fffffff}
---------------------------------------------------
# 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_kernel.out
IMAGE_HASH_FILENAME=
RSA_SIGN_FILENAME=
---------------------------------------------------
# Specify VERBOSE as 1, if you want to Display Header Information [Optional]
VERBOSE=0
```

Table 68. Description of Fields in Input File

Field in Input File	Description
IE_KEY_SEL	Number of the key in IE Table that is to be used to validate image

As shown above, in the input file to uni_sign, the field 'IE_KEY_SEL' indicates that a Key from IE Table is being used and it also indicates which key number from IE Table needs to be used to validate this image during run-time. For build-time, CST tool needs the IE private key. This is indicated via the field 'PRI_KEY'.

Output of uni_sign

```
$ ./uni_sign input_files/uni_sign/ls2088_1088/ie_keys/qspi/input_kernel_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/ls2088_1088/ie_keys/qspi/input_kernel_secure

SRK/Public Key Hash not calculated.. IE = 1

************************************************
* Header File is with Signature appended
************************************************

Header File Created: hdr_kernel.out
SRK (Public Key) Hash Not Available
```

As shown above, SRK Hash is not calculated as IE Key is being used.

6.1.2.6 PBI structure

	Fields	Offset	Size (In 32-bit word)
RCW	Preamble (RCW)	0x00	1
	Load RCW command	0x04	1
	RCW words	0x08 – 0x87	32
	RCW checksum	0x88	1
PBI commands	Load security header CSF header	0x8c 0x90 – 0xdf	1 20
	Load boot 1CSF header Boot 1 pointer	0xe0 0xe4	1 1
	Other PBI commands	0xe8	N
	STOP command (With/ Without CRC)	0xe8 + (4*N)	2
SRK table	SRK table	0x90 + SRK table offset in CSF header	(No. of keys * Key length)
RSA signature	Signature	0x90 + Sign offset in CSF header	Sign length

RCW

Preamble The preamble is always the first element in a PBI image. It contains a standard pattern that identifies the memory location as the beginning of a valid PBI image. The preamble is a 4-byte pattern defined as “0xaa55aa55”.

Load RCW command The next word is load RCW command. This command loads the 1024-bit Reset Configuration Word from the interface specified by Power-on-Reset (POR) configuration strapping pins. It has the following two formats.

1. Load RCW with Checksum (0x10): Read Reset Configuration Word performs simple 32-bit checksum, and update RCW registers.
2. Load RCW without Checksum (0x11): Read Reset Configuration Word and update RCW registers without performing checksum. The version without the checksum includes padding with zeroes in the place of the checksum value.

RCW words 1024 RCW bits that is 32 words of 32 bits.

RCW checksum It is calculated as a 32-bit unsigned integer summation of the RCW Preamble, the Load RCW with checksum command, and each of the 32 words (32-bit) of the RCW. A simple 32-bit checksum is used for the validation of the command.

```
checksum(RCW_WORD[]){
    unsigned_32 sum = 0xAA55AA55 + 0x80100000 + Load RCW Command;
    for(i=0; i<32; i++)
        sum+=RCW_WORD[i];
    return (sum);
}
```

NOTE

Checksum will have to be updated by CST tool as the fields like SB_EN, PBI_LEN in the RCW words are changed.

PBI commands

Load security header This command loads information required for authentication of the PBI image. The security header includes pointers to an SRK key table and RSA signatures as well as other flags and IDs. The CSF Header is part of the command. Please refer the CSF header structure in .

Load boot 1 CSF header This command loads a pointer to a CSF Header used for authentication of the Boot 1 Secondary Program Loader. This 32-bit value used by the Boot 0 ISBC and is required for secure boot.

Other PBI commands Other PBI commands input by user.

STOP command This command ends the PBI sequence and has two variants (with and without CRC).

The CRC check value covers all commands from the first command after the RCW up to and including this CRC and Stop command, regardless of whether any are skipped by Jump commands during execution.

In Stop command without CRC, it ends the PBI sequence immediately. It does not include a CRC value, but it instead has a 32-bit padding with zeroes so that it is the same size as the Stop with CRC command.

NOTE

CST tool updates the PBI commands by adding Load Security Header command and Load Boot 1 Security Header command. So, CRC must also be updated.

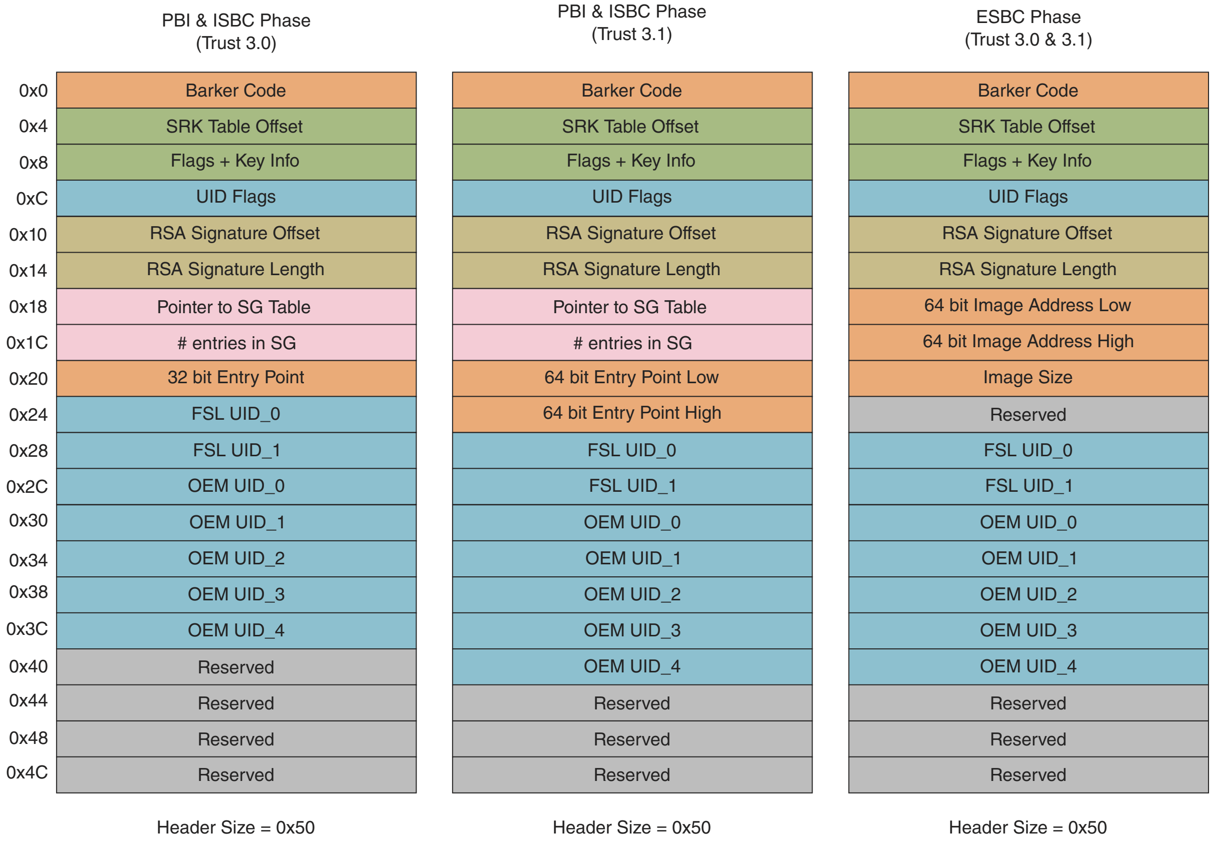
SRK table Table of public keys is used in secure boot validation. It is kept at an offset from the CSF header. The offset is specified in the CSF header.

RSA signature RSA signature is calculated over all PBI commands and SRK table. It is kept at an offset from the CSF header. The offset is specified in the CSF header.

6.1.2.7 CSF header structure definition

Table 69. Trust architecture and SFP information

SoC	Trust Arch. version	SFP version	POVDD	DRVR		OTPMK	
				Algo (CST)	Register to check Hamming Error	Algo (CST)	Register to check Hamming Error
LS1088A	3.1	3.5	1.89 V	A2	SFP Secret Value Hamming Error Status Register (SFP_SVHESR)	2	SFP Secret Value Hamming Error Status Register (SFP_SVHESR)
LS2088A (LS2 Rev2)	3.1	3.5	1.89 V	A2		2	



	PBI & ISBC Phase (Trust 3.0)	PBI & ISBC Phase (Trust 3.1)	ESBC Phase (Trust 3.0 & 3.1)
0x0	Barker Code	Barker Code	Barker Code
0x4	SRK Table Offset	SRK Table Offset	SRK Table Offset
0x8	Flags + Key Info	Flags + Key Info	Flags + Key Info
0xC	UID Flags	UID Flags	UID Flags
0x10	RSA Signature Offset	RSA Signature Offset	RSA Signature Offset
0x14	RSA Signature Length	RSA Signature Length	RSA Signature Length
0x18	Pointer to SG Table	Pointer to SG Table	64 bit Image Address Low
0x1C	# entries in SG	# entries in SG	64 bit Image Address High
0x20	32 bit Entry Point	64 bit Entry Point Low	Image Size
0x24	FSL UID_0	64 bit Entry Point High	Reserved
0x28	FSL UID_1	FSL UID_0	FSL UID_0
0x2C	OEM UID_0	FSL UID_1	FSL UID_1
0x30	OEM UID_1	OEM UID_0	OEM UID_0
0x34	OEM UID_2	OEM UID_1	OEM UID_1
0x38	OEM UID_3	OEM UID_2	OEM UID_2
0x3C	OEM UID_4	OEM UID_3	OEM UID_3
0x40	Reserved	OEM UID_4	OEM UID_4
0x44	Reserved	Reserved	Reserved
0x48	Reserved	Reserved	Reserved
0x4C	Reserved	Reserved	Reserved
	Header Size = 0x50	Header Size = 0x50	Header Size = 0x50

Figure 33. CSF header structure

Table 70. CSF header structure (ISBC trust 3.0)

Offset		Description
0x00		**Barker Code** Fixed value which the ISBC uses to confirm it has located the start of a CSF header. (12_19_20_01) 0x00 – 0x12 0x01 – 0x19 0x02 – 0x20 0x03 – 0x01 It is numeric encoding of LSTA (LS Series Trust Architecture)
0x04		**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 in SRK table, the SRK table is read. * Description of fields in SRK table is mentioned below.
0x08	0x08	**No. of keys** This field specifies the no. of keys in the SRK table

Table continues on the next page...

Table 70. CSF header structure (ISBC trust 3.0) (continued)

Offset		Description
	0x09	**Key No. for verification** Key # to use for verification; the key in the table which the ISBC uses to attempt signature verification
	0x0a	Field Reserved
	0x0b	**IE :** Reserved **MP :** Execute Manufacturing Protection Routine **ISS :** Increment Security State; indicates whether the ISBC should increment the SNVS SSM upon successful verification **B01 :** Identifies whether this is the CSF header of a boot 0 image (PBI) or a boot 1 image (SPL) **LW :** Leave Writeable; when set; ISBC does not set the SFP Write Disable
0x0C	0x0C	Reserved
	0x0D	Reserved
	0x0E	Reserved
	0x0F	**OIDx**: when set, the corresponding OEM UID field in the SFP is included in the digital signature verification. For each bit set, the corresponding OUID field is included in the CSF header. **FUID** : when set, the 64b FUID is included in the digital signature verification and the FUID is included in the CSF header Other bits are reserved.
0x10		**RSA signature offset** This location contains an address which is the offset 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.
0x14		**RSA signature length** This location contains the length of the RSA signature in bytes.
0x18		**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 in SG Table, the SG table is read. * Description of fields in SG table is mentioned below.
0x1C		**No. of entries** This field specifies the number of entries present in SG table.
0x20		**Entry point (32 bit)** ISBC transfers control to this location upon successful validation of ESBC image(s).
0x24		FSL UID 0
0x28		FSL UID 1

Table continues on the next page...

Table 70. CSF header structure (ISBC trust 3.0) (continued)

Offset	Description
0x2c	OEM UID 0
0x30	OEM UID 1
0x34	OEM UID 2
0x38	OEM UID 3
0x3c	OEM UID 4
0x40	Reserved
0x44	Reserved
0x48	Reserved
0x4C	Reserved

Table 71. CSF header structure (ISBC trust 3.1)

Offset		Description
0x00		**Barker Code** Fixed value which the ISBC uses to confirm it has located the start of a CSF header. (12_19_20_01) 0x00 – 0x12 0x01 – 0x19 0x02 – 0x20 0x03 – 0x01 It is numeric encoding of LSTA (LS Series Trust Architecture)
0x04		**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 in SRK table, the SRK table is read.
0x08	0x08	**No. of keys** This field specifies the no. of keys in the SRK table
	0x09	**Key No. for verification** Key # to use for verification; the key in the table which the ISBC uses to attempt signature verification
	0x0a	Field Reserved
	0x0b	**IE :** ISBC Extension (Reserved) **MP :** Execute Manufacturing Protection Routine **ISS :** Increment Security State; indicates whether the ISBC should increment the SNVS SSM upon successful verification **B01 :** identifies whether this is the CSF header of a boot 0 image (PBI) or a boot 1 image (SPL) **LW :** Leave Writeable; when set; ISBC does not set the SFP Write Disable

Table continues on the next page...

Table 71. CSF header structure (ISBC trust 3.1) (continued)

0x0C	0x0C	Reserved
	0x0D	Reserved
	0x0E	Reserved
	0x0F	**OIDx**: when set, the corresponding OEM UID field in the SFP is included in the digital signature verification. For each bit set, the corresponding OUID field is included in the CSF header. **FUID** : when set, the 64b FUID is included in the digital signature verification and the FUID is included in the CSF header Other bits are reserved.
0x10		**RSA signature offset** This location contains an address which is the offset 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.
0x14		**RSA signature length** This location contains the length of the RSA signature in bytes.
0x18		**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 in SG table, the SG table is read.
0x1C		**No. of entries** This field specifies the number of entries present in SG table.
0x20		**Entry point (64 bit)** ISBC transfers control to this location upon successful validation of ESBC image(s).
0x28		FSL UID 0
0x2c		FSL UID 1
0x30		OEM UID 0
0x34		OEM UID 1
0x38		OEM UID 2
0x3c		OEM UID 3
0x40		OEM UID 4
0x44		Reserved
0x48		Reserved
0x4C		Reserved

Table 72. CSF header structure (ESBC trust 3.0 and trust 3.1)

Offset	Description

Table continues on the next page...

Table 72. CSF header structure (ESBC trust 3.0 and trust 3.1) (continued)

0x00		**Barker Code** Fixed value which the ISBC uses to confirm it has located the start of a CSF header. (12_19_20_01) 0x00 – 0x12 0x01 – 0x19 0x02 – 0x20 0x03 – 0x01 It is numeric encoding of LSTA (LS Series Trust Architecture)
0x04		**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 in SRK Table, the SRK table is read.
0x08	0x08	**No. of keys** This field specifies the no. of keys in the SRK table
	0x09	**Key No. for verification** Key # to use for verification; the key in the table which the ISBC uses to attempt signature verification
	0x0a	Field Reserved
	0x0b	**IE :** ISBC Extension Flag
0x0C	0x0C	Reserved
	0x0D	Reserved
	0x0E	Reserved
	0x0F	**OIDx**: when set, the corresponding OEM UID field in the SFP is included in the digital signature verification. For each bit set, the corresponding OUID field is included in the CSF header. **FUID** : when set, the 64b FUID is included in the digital signature verification and the FUID is included in the CSF header Other bits are reserved.
0x10		**RSA signature offset** This location contains an address which is the offset 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.
0x14		**RSA signature length** This location contains the length of the RSA signature in bytes.
0x18		**Image address (64 bit)**
0x20		**Image size**
0x24		IE Key Select
0x28		FSL UID 0

Table continues on the next page...

Table 72. CSF header structure (ESBC trust 3.0 and trust 3.1) (continued)

0x2c	FSL UID 1
0x30	OEM UID 0
0x34	OEM UID 1
0x38	OEM UID 2
0x3c	OEM UID 3
0x40	OEM UID 4
0x44	Reserved
0x48	Reserved
0x4c	Reserved

Table 73. SRK table structure

Offset	Description
0x00	SRK 0 Length (Length of Modulus or Exponent; Modulus length always equals Exponent length)
0x04	SRK 0 Value (Modulus)
0x04 + K	SRK 0 Value (Exponent)
0x04 + 2K	SRK 0 (Padding; 8Kb - (Exponent+Modulus))
0x04 *1 + (10 * 1)K	SRK 1 Length (Length of Modulus or Exponent; Modulus length always equals Exponent length)
0x04 * 2 + (10 *1) K	SRK 1 Value (Modulus)
0x04 * 2+ (10 * 1) + 1k	SRK 1 Value (Exponent)
0x04 * 2 + (10 * 1) + 2K	SRK 1 (Padding; 8Kb - (Exponent+Modulus))
0x04 * (N-1) + (10 *(N-1))K	SRK N Length (Length of Modulus or Exponent; Modulus length always equals Exponent length)
0x04 * N + (10 *(N-1))K	SRK N Value (Modulus)
0x04 * N + (10 *(N-1)) + 1K	SRK N Value (Exponent)
0x04 * N + (10 *(N-1)) + 2K	SRK N (Padding; 8Kb - (Exponent+Modulus))

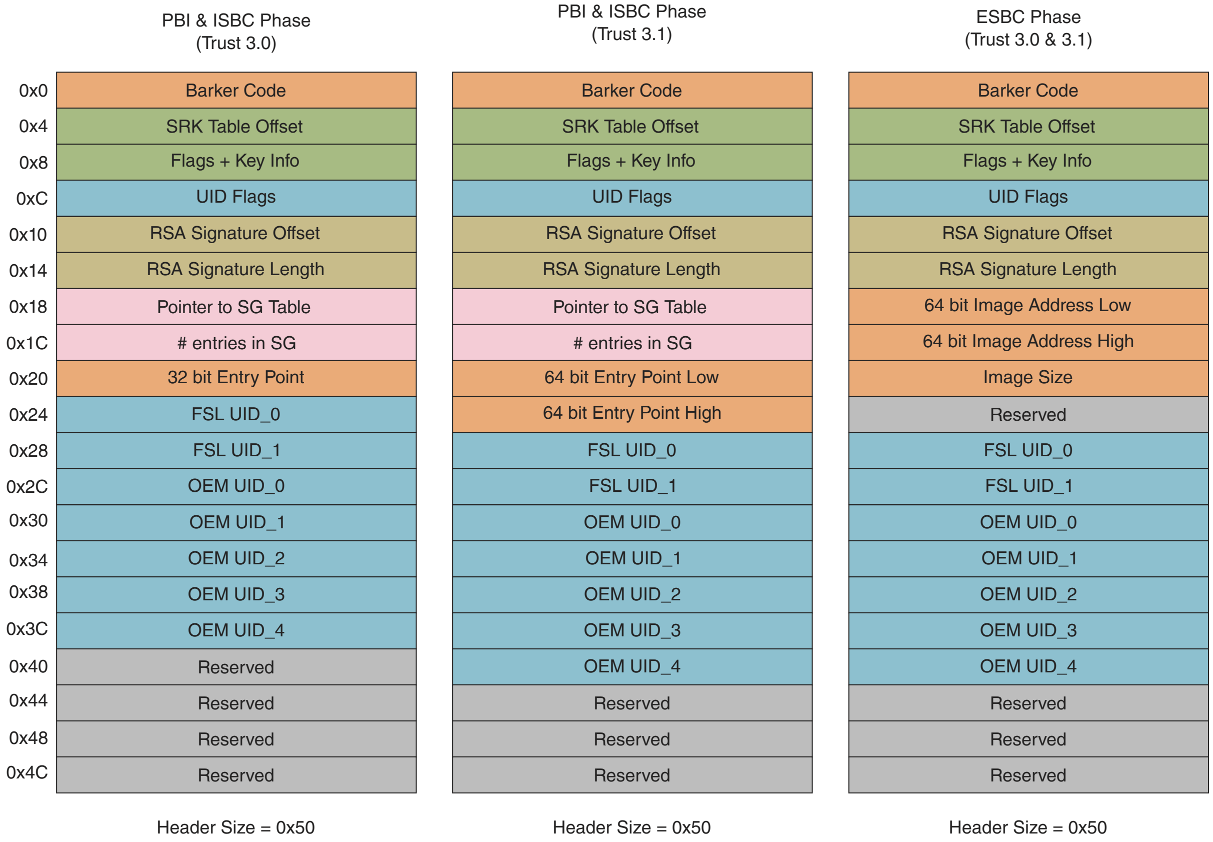
Table 74. SG Table Structure

Offset	Description
0x00	Length
0x04	Reserved
0x08	SRC Address Low
0x0C	SRC Address High

6.1.2.8 CSF header structure definition

Table 75. Trust architecture and SFP information

SoC	Trust Arch. version	SFP version	POVDD	DRVR		OTPMK	
				Algo (CST)	Register to check Hamming Error	Algo (CST)	Register to check Hamming Error
LS2080A (LS2 Rev1)	3.0	3.4	1.89 V	A2	SFP Secret Value Hamming Error Status Register (SFP_SVHESR)	2	SFP Secret Value Hamming Error Status Register (SFP_SVHESR)
LS1088A	3.1	3.5	1.89 V	A2		2	
LS2088A (LS2 Rev2)	3.1	3.5	1.89 V	A2		2	



	PBI & ISBC Phase (Trust 3.0)	PBI & ISBC Phase (Trust 3.1)	ESBC Phase (Trust 3.0 & 3.1)
0x0	Barker Code	Barker Code	Barker Code
0x4	SRK Table Offset	SRK Table Offset	SRK Table Offset
0x8	Flags + Key Info	Flags + Key Info	Flags + Key Info
0xC	UID Flags	UID Flags	UID Flags
0x10	RSA Signature Offset	RSA Signature Offset	RSA Signature Offset
0x14	RSA Signature Length	RSA Signature Length	RSA Signature Length
0x18	Pointer to SG Table	Pointer to SG Table	64 bit Image Address Low
0x1C	# entries in SG	# entries in SG	64 bit Image Address High
0x20	32 bit Entry Point	64 bit Entry Point Low	Image Size
0x24	FSL UID_0	64 bit Entry Point High	Reserved
0x28	FSL UID_1	FSL UID_0	FSL UID_0
0x2C	OEM UID_0	FSL UID_1	FSL UID_1
0x30	OEM UID_1	OEM UID_0	OEM UID_0
0x34	OEM UID_2	OEM UID_1	OEM UID_1
0x38	OEM UID_3	OEM UID_2	OEM UID_2
0x3C	OEM UID_4	OEM UID_3	OEM UID_3
0x40	Reserved	OEM UID_4	OEM UID_4
0x44	Reserved	Reserved	Reserved
0x48	Reserved	Reserved	Reserved
0x4C	Reserved	Reserved	Reserved
	Header Size = 0x50	Header Size = 0x50	Header Size = 0x50

Figure 34. CSF header structure

Table 76. CSF header structure (ISBC trust 3.0)

Offset		Description
0x00		**Barker code** Fixed value which the ISBC uses to confirm it has located the start of a CSF header. (12_19_20_01) 0x00 – 0x12 0x01 – 0x19 0x02 – 0x20 0x03 – 0x01 It is numeric encoding of LSTA (LS Series Trust Architecture)
0x04		**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	0x08	**No. of keys** This field specifies the no. of keys in the SRK table
	0x09	**Key No. for verification** Key # to use for verification; the key in the table which the ISBC uses to attempt signature verification
	0x0a	Field Reserved
	0x0b	**IE :** Reserved **MP :** Execute Manufacturing Protection Routine **ISS :** Increment Security State; indicates whether the ISBC should increment the SNVS SSM upon successful verification **B01 :** Identifies whether this is the CSF header of a boot 0 image (PBI) or a boot 1 image (SPL) **LW :** Leave Writeable; when set, ISBC does not set the SFP Write Disable
0x0C	0x0C	Reserved
	0x0D	Reserved
	0x0E	Reserved
	0x0F	**OIDx**: when set, the corresponding OEM UID field in the SFP is included in the digital signature verification. For each bit set, the corresponding OUID field is included in the CSF header. **FUID** : when set, the 64b FUID is included in the digital signature verification and the FUID is included in the CSF header Other bits are reserved.
0x10		**RSA signature offset** This location contains an address which is the offset 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.

Table continues on the next page...

Table 76. CSF header structure (ISBC trust 3.0) (continued)

Offset	Description
0x14	**RSA signature length** This location contains the length of the RSA signature in bytes.
0x18	**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.
0x1C	**No. of entries** This field specifies the number of entries present in SG table.
0x20	**Entry point (32 bit)** ISBC transfers control to this location upon successful validation of ESBC image(s).
0x24	FSL UID 0
0x28	FSL UID 1
0x2c	OEM UID 0
0x30	OEM UID 1
0x34	OEM UID 2
0x38	OEM UID 3
0x3c	OEM UID 4
0x40	Reserved
0x44	Reserved
0x48	Reserved
0x4C	Reserved

Table 77. CSF header structure (ISBC trust 3.1)

Offset	Description
0x00	**Barker Code** Fixed value which the ISBC uses to confirm it has located the start of a CSF header. (12_19_20_01) 0x00 – 0x12 0x01 – 0x19 0x02 – 0x20 0x03 – 0x01 It is numeric encoding of LSTA (LS Series Trust Architecture)
0x04	**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 77. CSF header structure (ISBC trust 3.1) (continued)

0x08	0x08	**No. of keys** This field specifies the no. of keys in the SRK Table
	0x09	**Key No. for verification** Key # to use for verification; the key in the table which the ISBC uses to attempt signature verification
	0x0a	Field Reserved
	0x0b	**IE :** ISBC Extension (Reserved) **MP :** Execute Manufacturing Protection Routine **ISS :** Increment Security State; indicates whether the ISBC should increment the SNVS SSM upon successful verification **B01 :** identifies whether this is the CSF header of a boot 0 image (PBI) or a boot 1 image (SPL) **LW :** Leave Writeable; when set, ISBC does not set the SFP Write Disable
0x0C	0x0C	Reserved
	0x0D	Reserved
	0x0E	Reserved
	0x0F	**OIDx**: when set, the corresponding OEM UID field in the SFP is included in the digital signature verification. For each bit set, the corresponding OUID field is included in the CSF header. **FUID** : when set, the 64b FUID is included in the digital signature verification and the FUID is included in the CSF header Other bits are reserved.
0x10		**RSA signature offset** This location contains an address which is the offset 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.
0x14		**RSA signature length** This location contains the length of the RSA signature in bytes.
0x18		**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 in SG Table, the SG table is read.
0x1C		**No. of entries** This field specifies the number of entries present in SG table.
0x20		**Entry point (64 bit)** ISBC transfers control to this location upon successful validation of ESBC image(s).
0x28		FSL UID 0
0x2c		FSL UID 1

Table continues on the next page...

Table 77. CSF header structure (ISBC trust 3.1) (continued)

0x30	OEM UID 0
0x34	OEM UID 1
0x38	OEM UID 2
0x3c	OEM UID 3
0x40	OEM UID 4
0x44	Reserved
0x48	Reserved
0x4C	Reserved

Table 78. CSF header structure (ESBC trust 3.0 and trust 3.1)

Offset		Description
0x00		**Barker code** Fixed value which the ISBC uses to confirm it has located the start of a CSF header. (12_19_20_01) 0x00 – 0x12 0x01 – 0x19 0x02 – 0x20 0x03 – 0x01 It is numeric encoding of LSTA (LS Series Trust Architecture)
0x04		**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 in SRK table, the SRK table is read.
0x08	0x08	**No. of keys** This field specifies the no. of keys in the SRK table
	0x09	**Key No. for verification** Key # to use for verification; the key in the table which the ISBC uses to attempt signature verification
	0x0a	Field Reserved
	0x0b	**IE :** ISBC Extension Flag
0x0C	0x0C	Reserved
	0x0D	Reserved
	0x0E	Reserved

Table continues on the next page...

Table 78. CSF header structure (ESBC trust 3.0 and trust 3.1) (continued)

	0x0F	**OIDx**: when set, the corresponding OEM UID field in the SFP is included in the digital signature verification. For each bit set, the corresponding OUID field is included in the CSF header. **FUID** : when set, the 64b FUID is included in the digital signature verification and the FUID is included in the CSF header Other bits are reserved.
0x10		**RSA signature offset** This location contains an address which is the offset 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.
0x14		**RSA signature length** This location contains the length of the RSA signature in bytes.
0x18		**Image address (64 bit)**
0x20		**Image size**
0x24		IE Key Select
0x28		FSL UID 0
0x2c		FSL UID 1
0x30		OEM UID 0
0x34		OEM UID 1
0x38		OEM UID 2
0x3c		OEM UID 3
0x40		OEM UID 4
0x44		Reserved
0x48		Reserved
0x4c		Reserved

6.1.2.9 Secure boot specific RCW fields

This section describes the various fields in RCW which are relevant to the ISBC code executed in the Service Processor Boot ROM.

SB_EN Secure Boot Enable

Bit(s): 266

- 0 Secure Boot is not enabled[9]
- 1 Secure Boot is enabled

PBI_LENGTH Pre-Boot Initialization Length

Bit(s): 287-276

[9] Secure Boot is enabled if either this RCW bit is set or the Intent to Secure fuse value is set.

Size in words of the PBI commands.

SDBGEN Secure Debug Enable

Bit(s): 288

Secure Debug (CoreSight SPIDEN) is enabled after RCW is loaded if this RCW bit is set and the 'Intent to Secure' fuse value is cleared.

- 0 Secure debug is not enabled
- 1 Secure debug is enabled if the ITS fuse is not burned to asserted

GPIO_LED_EN GPIO LED Enable

Bit(s): 311

If the OEM chooses to implement a LED to indicate secure boot failure, the LED will be connected to a GPIO. The SP Boot ROM code sequence turns on the LED (if RCW[GPIO_LED_EN] = 1) by configuring one GPIO direction (GPDIR) register bit as an output and writing the corresponding output in a GPIO block data (GPDAT) register.

GPIO_LED_NUM GPIO Number for LED

Bit(s): 310-304

If GPIO_LED_EN is set, these bits specify the GPIO number to which LED is connected.

- 0x1f - 0x00 : GPIO_1
- 0x3f - 0x20 : GPIO_2
- 0x5f - 0x40 : GPIO_3
- 0x7f - 0x60 : GPIO_4

NOTE

The GPIO output assigned to the LED is driven high to whatever VDD voltage is supplied by the integrated device for the chosen GPIO output.Since GPIO pins at the time of SoC reset are initially configured as inputs, and since there will be some indeterminate period of time from the assertion of SoC reset to when the GPIO pin is configured by SP Boot ROM as an output, the GPIO pin chosen must be terminated with a weak pulldown to ground.

6.1.2.10 ISBC error codes

Error handling in production environment (ITS = 1)

- Error code would be logged in DCFG SCARTCH register.
- SNVS would be transitioned to **soft fail** state.
- Activate the LED. If the OEM chooses to implement a LED to indicate secure boot failure, the LED will be connected to a GPIO. The information of GPIO is specified via bits in RCW.

GPIO_LED_EN Bit(s): 311

The SP Boot ROM code sequence turns on the LED (if RCW[GPIO_LED_EN] = 1) by configuring one GPIO direction (GPDIR) register bit as an output and writing the corresponding output in a GPIO block data (GPDAT) register.

GPIO_LED_NUM Bit(s): 310-304

If GPIO_LED_EN is set, these bits specify the GPIO number to which LED is connected.

— 0x1f - 0x00 : GPIO_1

- 0x3f - 0x20 : GPIO_2
- 0x5f - 0x40 : GPIO_3
- 0x7f - 0x60 : GPIO_4

- Soft reset would be issued
- Cores would then enter infinite loop (If Reset is disabled)[10]

Error handling in development environment (ITS = 0, SB_EN = 1)

- Error code would be logged in DCFG SCARTCH register.
- SNVS would be transitioned to **non-secure** state.
- Further actions depends on the type of failure

Fatal Errors	Core is put in infinite Loop
Non-Fatal Error	Application software is allowed to execute

Error codes

The Error codes reported by SP Boot ROM can be categorized in following sections.

1. Core exceptions
2. Device errors
3. RCW/PBI errors
4. Validation errors

Table 79. ISBC error codes

When error generated	Error code	Value	Description
Core exceptions			
Random	ERROR_UNDEFINED_INSTRUCTION	0x1	Occurs if neither the processor nor any attached co-processor recognizes the currently executing instruction.
Random	ERROR_SWI	0x2	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.
Random	ERROR_PREFETCH_ABORT	0x3	Occurs when the processor attempts to execute an instruction that has been prefetched from an illegal address.
Random	ERROR_DATA_ABORT	0x4	Occurs when a data transfer instruction attempts to load or store data at an illegal address.
Random	ERROR_IRQ	0x5	Occurs when the processor external interrupt request pin is asserted (LOW) and IRQ interrupts are enabled.

Table continues on the next page...

[10] To debug the root cause of failure and view the error code, Reset has to be disabled on the SoC.

Table 79. ISBC error codes (continued)

When error generated	Error code	Value	Description
Random	ERROR_FIQ	0x6	Occurs when the processor external fast interrupt request pin is asserted (LOW) and FIQ interrupts are enabled.
Device Errors – I2C			
Random	ERROR_I2C_TIMEOUT	0x11	
Random	ERROR_I2C_RESTART	0x12	
Random	ERROR_I2C_NODEV	0x13	
Random	ERROR_I2C_NOT_IDLE	0x14	
Random	ERROR_I2C_NOT_BUSY	0x15	
Random	ERROR_I2C_INVALID_OFFSET	0x16	
Random	ERROR_I2C_NO_WAKEUP_INIT	0x17	
Random	ERROR_I2C_NO_WAKEUP_READ	0x18	
Random	ERROR_I2C_NOACK	0x19	
Random	ERROR_READ_TIMEOUT	0x1a	
Random	ERROR_SLAVE_ADDR_TIMEOUT	0x1b	
Random	ERROR_MEM_ADDR_TIMEOUT	0x1c	
Device Errors – ESDHC			
Random	ERROR_ESDHC_CARD_DETECT_FAIL	0x31	
Random	ERROR_ESDHC_UNUSABLE_CARD	0x32	
Random	ERROR_ESDHC_COMMUNICATION_ERROR	0x33	
Random	ERROR_ESDHC_BLOCK_LENGTH	0x34	
Device Errors – QSPI			
Random	ERROR_QSPI_INVALID_OFFSET	0x41	
Phase – “**RCW**”			
RCW Phase	ERROR_PREAMBLE	0x50	Preamble not found.
RCW Phase	ERROR_RCW_CMD_NOT_FOUND	0x51	RCW command not found
RCW Phase	ERROR_RCW_CHECKSUM_MISMATCH	0x52	Checksum mismatch in RCW
RCW Phase	ERROR_RCW_SRC_INVALID	0x58	RCW_SRC is not a valid source
RCW Phase	ERROR_RCW_REQ_NOT_SET	0x59	RCW_REQ bit never set by Reset state machine (RSM)
RCW Phase	ERROR_PBI_REQ_NOT_SET	0x60	PBI_REQ bit never set (by RSM)
Phase = **PBI**			

Table continues on the next page...

Table 79. ISBC error codes (continued)

When error generated	Error code	Value	Description
PBI Phase	ERROR_SEC_CAAM_INIT	0x61	CAAM init failed (Would rarely occur)
PBI Phase	ERROR_SEC_CAAM_NOT_FOUND	0x62	CAAM block not found in case of secure boot
PBI Phase	ERROR_PBI_SRC_NOT_SAME_AS_RCW_SRC	0x64	Mismatch between RCW_SRC and PBI_SRC fields
PBI Phase	ERROR_PBI_LENGTH	0x65	PBI length defined in RCW[PBI_LEN] field is invalid
PBI Phase	ERROR_PBI_LAST _CMD_NOT_STOP	0x66	STOP or CRC&STOP not found at the end of the specified PBI Length.
PBI Phase	ERROR_PBI_ COMMAND_UNKNOWN	0x67	An invalid command parsed by PBI Parser
PBI Phase	ERROR_CAAM_SELF_TEST	0x6a	CAAM self-test failed
PBI Phase	ERROR_PBI_ COPY_INVALID_ SRC_TYPE	0x70	Copy command, src field does not match the RCW_SRC field
PBI Phase	ERROR_PBI_ COPY_INVALID_ DST_ADDR	0x71	Copy command, dest field is not 0x00
PBI Phase	ERROR_PBI_ COPY_INVALID_SRC_ADDR_ SRC_ADDR	0x72	SRC address is invalid (ROM/ OCRAM reserved for SP)
PBI Phase	ERROR_PBI_CCSR_BYTE_COUNT	0x74	Byte count in CCSR Write not valid
PBI Phase	ERROR_PBI_CCSR_4_BYTE_ALLIGNED	0x75	Offset is not 4 byte aligned
PBI Phase	ERROR_PBI_CCSR_OFFSET_INVALID	0x76	Offset is invalid that is less than allowed CCSR Base 0x0100_0000
PBI Phase	ERROR_PBI_ACSR_INVALID_ADDRESS	0x78	Source address in ACSR invalid (invalid addresses - OCRAM or ROM address)
PBI Phase	ERROR_PBI_ACSR_BYTE_COUNT	0x79	Byte count in ACSR write command not valid
PBI Phase	ERROR_PBI_ACSR_WINDOW_NOT_SET	0x7a	ATU Window is not configured
PBI Phase	ERROR_PBI_ACSR_OFFSET_ALLIGNED	0x7b	ACSR offset is invalid and trying to write to Reserved space on OCRAM.
PBI Phase	ERROR_PBI_ALTCFG_WNDW_INVALID	0x7c	ATU Window is invalid
PBI Phase	ERROR_PBI_JUMP_OUT_LENGTH	0x80	Offset specified in JUMP command does not lie in PBI length range
PBI Phase	ERROR_PBI_JUMP_4_BYTE_ALLIGNED	0x81	Offset specified in JUMP command is not 4 byte aligned
PBI Phase	ERROR_PBI_JUMP_OFFSET_0	0x82	Offset specified in JUMP command is 0
PBI Phase	ERROR_PBI_LOADC_4_BYTE_ALLIGNED	0x84	Address specified in LOAD condition command is not 4 byte aligned

Table continues on the next page...

Table 79. ISBC error codes (continued)

When error generated	Error code	Value	Description
PBI Phase	ERROR_PBI_JUMPC_OUT_LENGTH	0x88	Offset specified in JUMP command does not lie in PBI length range
PBI Phase	ERROR_PBI_JUMPC_4_BYTE_ALLIGNED	0x89	Offset specified in JUMP conditional command is not 4 byte aligned
PBI Phase	ERROR_PBI_JUMPC_CONDITION_NOT_SET	0x8a	Jump conditional command encountered before condition is set using Load Condition
PBI Phase	ERROR_PBI_CRC_MISMATCH	0x90	CRC mismatch
PBI Phase	ERROR_PBI_POLL	0x91	Poll timeout
PBI Phase	ERROR_PBI_POLL_4_BYTE_ALLIGNED	0x92	Address being polled is not 4 byte aligned
PBI Phase	ERROR_PBI_BOOT1_CSF_INVALID_ADDR	0x94	Address of CSF header is not valid
PBI Phase	ERROR_PBI_BOOT1_CSF_ALLIGNED	0x95	Address of CSF header is not 4 byte aligned
Phase = **Verify** (System State Errors (Secure boot))			
Before PBI verification	ERROR_STATE_NOT_CHECK	0xf0	SEC_MON State Machine not in CHECK state at start of ISBC in primary flow. Some Security violation could have occurred.
Before PBI verification	ERROR_STATE_NOT_CHECK_TRUSTED	0xf1	SEC_MON State Machine not in CHECK/Trusted state at start of ISBC in secondary flow.
Phase = **Verify** (Secure Boot Fatal errors)			
Verify PBI	ERROR_PBI_COMMANDS_NOT_FOUND	0xf4	Not having PBI commands in RCW is error scenario for secure boot
Verify PBI	ERROR_SEC_HDR_NOT_FOUND	0xf5	Error if security header command not found in RCW. Expected location of Security Header command • After Preamble for hard coded RCW • After preamble and rcw for other RCW sources
Phase = **Verify** (Secure Boot Fatal (Header parsing errors))			
Verify PBI	ERROR_HEADER_LOC	0xf8	Header location is invalid
Verify PBI	ERROR_HEADER_BARKER	0xf9	Barker code in the header is incorrect
Verify PBI	ERROR_HEADER_INVALID	0xfa	Flag B01 in the header identifies this as SPL header
Phase = **Verify** (Secure Boot Non Fatal (Key/UID related errors))			
Verify PBI	ERROR_INVALID_SRK_ENTRY_KEYLEN	0x210	Length of public key specified in one of the entries in srk table is not one of the supported values. (1k, 2k, or 4k)

Table continues on the next page...

Table 79. ISBC error codes (continued)

When error generated	Error code	Value	Description
Verify PBI	ERROR_ KEY_LEN_ NOT_TWICE_SIG_LEN	0x211	Public key is not twice the length of the RSA signature
Verify PBI	ERROR_ KEY_MOD_1	0x212	Most significant bit of modulus in header is zero.
Verify PBI	ERROR_ KEY_MOD_2	0x213	Modulus in header is even number
Verify PBI	ERROR_ SIG_KEY_MOD	0x214	Signature value is greater than modulus in header
Verify PBI	ERROR_ INVALID_SRK_NUM_ENTRY	0x215	Number of entries field in CSF Header is > 8 (This is when srk_flag in header is 1)
Verify PBI	ERROR_ INVALID_KEY_NUM	0x216	Key number to be used from srk table is not present in table. (This is when srk_flag in header is 1)
Verify PBI	ERROR_ KEY_REVOKED	0x217	Key selected from srk table has been revoked (This is when srk_flag in header is 1)
Verify PBI	ERROR_ FSL_UID	0x220	FSL_UID in ESBC header did not match the FSL_UID in SFP if fsl uid flag Is 1
Verify PBI	ERROR_ OEM_UID0	0x221	OEM_UID0 in ESBC header did not match the OEM_UID0 in SFP if oem uid0 flag is 1.
Verify PBI	ERROR_ OEM_UID1	0x222	OEM_UID1 in ESBC header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Verify PBI	ERROR_ OEM_UID2	0x223	OEM_UID1 in ESBC header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Verify PBI	ERROR_ OEM_UID3	0x224	OEM_UID1 in ESBC header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Verify PBI	ERROR_ OEM_UID4	0x225	OEM_UID1 in ESBC header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Phase = **Verify** (Header Verification Failure) Secure Boot Non Fatal			
Verify PBI	ERROR_ HASH_COMPARE_KEY	0x240	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.
Verify PBI	ERROR_ HASH_COMPARE_EM	0x241	RSA signature check failure. Signature provided by you in the header does not 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)
Phase = **Verify** (Secure Boot Fatal (Header parsing errors))			
Verify Boot1	ERROR_HEADER_LOC	0x100f8	Header location is invalid
Verify Boot1	ERROR_HEADER_BARKER	0x100f9	Barker code in the header is incorrect.

Table continues on the next page...

Table 79. ISBC error codes (continued)

When error generated	Error code	Value	Description
Verify Boot1	ERROR_HEADER_INVALID	0x100fa	Flag B01 in the header identifies this as SPL header.
Phase = Verify (Secure Boot Fatal (SG Table related errors))			
Verify Boot1	ERROR_SG_ENTRY_POINT	0x10200	Entry point is not within any of SG entries
Verify Boot1	ERROR_SG_NUM_ENTRY	0x10201	No. of entries in SG table is 0 or >8
Verify Boot1	ERROR_SG_SIZE_ZERO	0x10202	A SG entry has size 0
Phase = **Verify** (Secure Boot Non-Fatal (Key/UID related errors))			
Verify Boot1	ERROR_INVALID_SRK_ENTRY_KEYLEN	0x10210	Length of public key specified in one of the entries in srk table is not one of the supported values. (1k, 2k, or 4k)
Verify Boot1	ERROR_ KEY_LEN_ NOT_TWICE_SIG_LEN	0x10211	Public key is not twice the length of the RSA signature
Verify Boot1	ERROR_ KEY_MOD_1	0x10212	Most significant bit of modulus in header is zero
Verify Boot1	ERROR_ KEY_MOD_2	0x10213	Modulus in header is even number
Verify Boot1	ERROR_ SIG_KEY_MOD	0x10214	Signature value is greater than modulus in header
Verify Boot1	ERROR_ INVALID_SRK_NUM_ENTRY	0x10215	Number of entries field in CSF Header is > 8 (This is when srk_flag in header is 1)
Verify Boot1	ERROR_ INVALID_KEY_NUM	0x10216	Key number to be used from srk table is not present in table. (This is when srk_flag in header is 1)
Verify Boot1	ERROR_ KEY_REVOKED	0x10217	Key selected from srk table has been revoked (This is when srk_flag in header is 1)
Verify Boot1	ERROR_ FSL_UID	0x10220	FSL_UID in ESBC Header did not match the FSL_UID in SFP if fsl uid flag Is 1
Verify Boot1	ERROR_ OEM_UID0	0x10221	OEM_UID0 in ESBC Header did not match the OEM_UID0 in SFP if oem uid0 flag is 1.
Verify Boot1	ERROR_ OEM_UID1	0x10222	OEM_UID1 in ESBC Header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Verify Boot1	ERROR_ OEM_UID2	0x10223	OEM_UID1 in ESBC Header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Verify Boot1	ERROR_ OEM_UID3	0x10224	OEM_UID1 in ESBC Header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Verify Boot1	ERROR_ OEM_UID4	0x10225	OEM_UID1 in ESBC Header did not match the OEM_UID1 in SFP if oem uid1 flag is 1.
Phase = **Verify** (Header Verification Failure) Secure Boot Non-Fatal			

Table continues on the next page...

Table 79. ISBC error codes (continued)

When error generated	Error code	Value	Description
Verify Boot1	ERROR_ HASH_COMPARE_KEY	0x10240	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.
Verify Boot1	ERROR_ HASH_COMPARE_EM	0x10241	RSA signature check failure. Signature provided by you in the header does not 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)
Verify Boot1	ERROR_PRIVATE_KEY_DERIVATION	0x10250	Error in Private key derivation when enabling Manufacturing Protection.

6.1.2.11 ESBC error codes

Table 80. ESBC validation failures

Value	Code	Definition
0x4	ERROR_ESBC_CLIENT_HEADER_BARKER	Wrong barker code in header
0x8	ERROR_ESBC_CLIENT_HEADER_KEY_LEN	Wrong public key length in header
0x10	ERROR_ESBC_CLIENT_HEADER_SIG_LEN	Wrong signature length in header
0x20	ERROR_ESBC_CLIENT_HEADER_KEY_LEN_NOT_TWICE_SIG_LEN	Public key length not twice of signature length
0x40	ERROR_ESBC_CLIENT_HEADER_KEY_MOD_1	Public key Modulus most significant bit not set
0x80	ERROR_ESBC_CLIENT_HEADER_KEY_MOD_2	Public key Modulus in header not odd
0x100	ERROR_ESBC_CLIENT_HEADER_SIG_KEY_MOD	Signature not less than modulus
0x400	ERROR_ESBC_CLIENT_HASH_COMPARE_KEY	Public key hash comparison failed
0x800	ERROR_ESBC_CLIENT_HASH_COMPARE_EM	RSA verification failed
0x10000	ERROR_ESBC_CLIENT_HEADER_SG	No SG support

Table continues on the next page...

Table 80. ESBC validation failures (continued)

Value	Code	Definition
0x20000	ERROR_ESBC_WRONG_CMD	Failure in command/Unknown command/Wrong arguments of boot script.
0x40000	ERROR_ESBC_MISSING_BOOTM	Bootm command missing from boot script.

6.1.2.12 Troubleshooting

	Symptoms	Reasons and/or Recommended actions
1.	No print on UART console.	Check the status register of sec mon block. Refer to the details of the register from the Reference Manual. Bits OTPMK_ZERO, OTPMK_SYNDROME and PE should be 0 otherwise there is some error in the OTPMK fuse blown by you. If OTMPK fuse is correct (see Step 1), check the DCFG SCRATCHRW3 register for error code. For a list of error codes, see ISBC error codes on page 251 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 CSF header. If entry point is correct, ensure that U-Boot image has been signed with the correct input file.
2.	Instead of Linux prompt, you get a U-Boot command 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 running normal U-Boot.
3	U-Boot hangs or board resets	Some validation failure occurred in U-Boot. Error code and description would be printed on U-Boot console. See ESBC error codes on page 258for more details on errors.

6.1.3 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

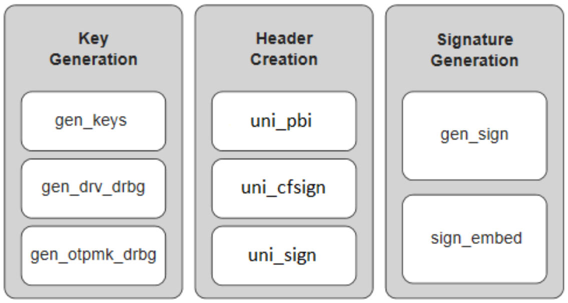


Figure 35. Tool in CST Package

6.1.3.1 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.

6.1.3.1.1 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
```

6.1.3.1.2 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
```

6.1.3.1.3 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
```

6.1.3.2 Header creation

6.1.3.2.1 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 175 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 81. 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

Table continues on the next page...

Table 81. Description of fields in input files of both type of platforms (TA2.x and TA3.x) (continued)

Field name	Description	Platform supported
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

TA 3.x : LS1088, LS2088. To know platforms under TA 2.x, refer Trust Architecture and SFP Information on page 209

6.1.3.2.1.1 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;}
----------------------------------------------------------------------------
```

Sample input file for SP based platforms

```
---------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform -
# TRUST 3.1: LS2088, LS1088
PLATFORM=LS1088
---------------------------------------------------
# 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
---------------------------------------------------
# For PBI Signing
# Name of RCW + PBI file [Mandatory]
RCW_PBI_FILENAME=rcw.bin
# Address of ISBC (Boot1) CSF Header [Mandatory]
BOOT1_PTR=20c00000
---------------------------------------------------
# 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=rcw_sec.bin
IMAGE_HASH_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
```

6.1.3.2.2 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.
    --help       Show the Help for Tool Usage.
```

Table 82. 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

Table continues on the next page...

Table 82. Description of fields (continued)

Field	Field description	Platform supported
HKAREA_SIZE	House Keeping Area Size (Required for Trust Arch v2.x devices only when esbc option is not provided)	TA2.x
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, you can refer to input_files/uni_sign/l<platform>/ie_ke

TA3.x: LS2088 and LS1088. To know platforms under TA1.x and TA 2.x, refer Trust Architecture and SFP Information on page 209

6.1.3.2.2.1 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=
```

6.1.3.3 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 264 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.

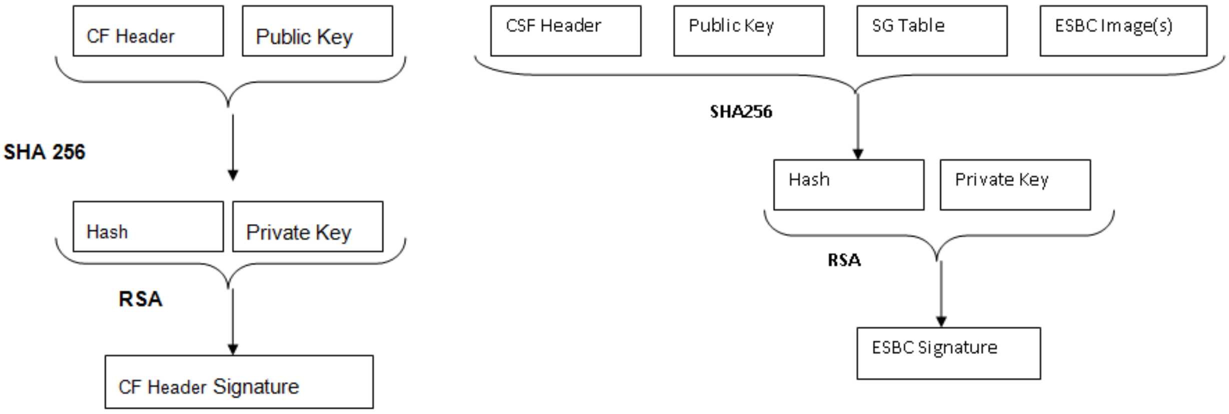


Figure 36. Dual signature generation

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.

6.1.3.3.1 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 270, 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
```

6.1.3.3.2 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[11] [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)

6.2 IMA-EVM on LS Trust Architecture SoCs

6.2.1 Introduction

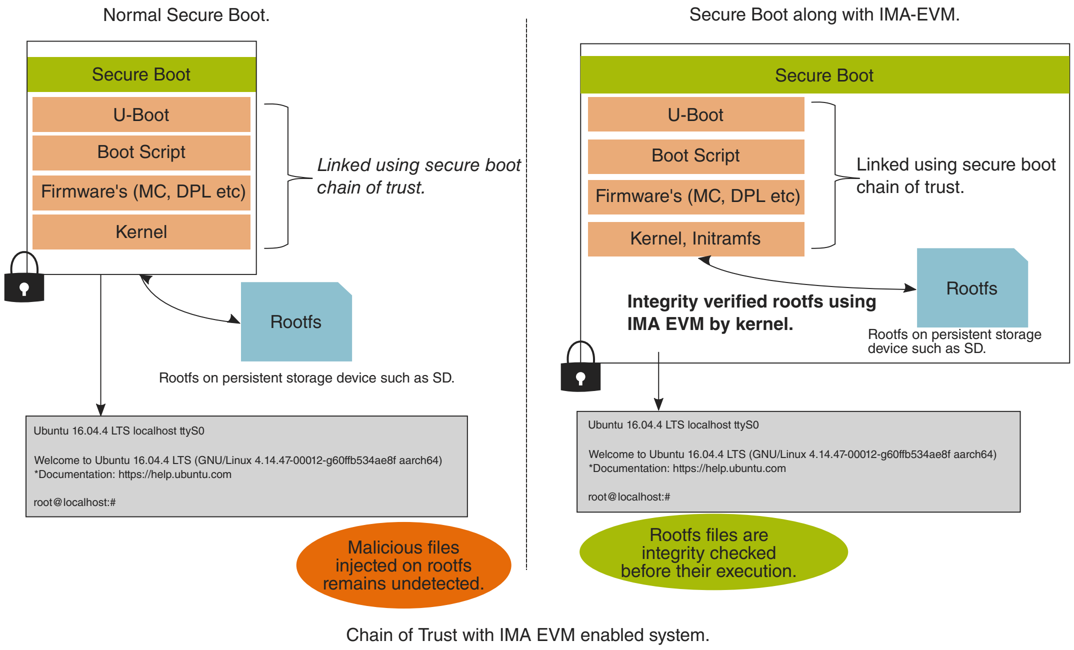
Layerscape trust architecture supports image validation using secure boot feature. Secure boot performs image validation in various stages and kernel image is the last one to get verified in chain of trust. Linux-based IMA-EVM feature helps in extending the chain of trust (COT) until rootfs installed over persistent storage devices.

Integrity Measurement Architecture (IMA): is the linux integrity subsystem used to detect if files have been accidentally or maliciously altered. It appraises a file's measurement against a "good" value stored as an extended attribute (security.ima) and enforces local file integrity checks. The extended attribute (security.ima) of a file is the hash value (SHA-1, SHA-256 or SHA-512) of its content. IMA maintains a list of hash values over all executables and other sensitive system files loaded at runtime into the system.

Extended Verification Module (EVM): protects a file's extended attributes against integrity attacks. The extended security attribute (security.evm) stores the HMAC value over other extended attributes associated with the file such as security.selinux, security.SMACK64, security.ima.

EVM depends on the kernel key retention system and requires an encrypted key named evm-key for the HMAC operation. The key is loaded onto the root user keyring using keyctl utility. EVM is enabled by setting an enable flag in securityfs/evm file.

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


Normal Secure Boot.
Secure Boot
U-Boot
Boot Script
Firmware's (MC, DPL etc)
Kernel
Linked using secure boot chain of trust.
Rootfs
Rootfs on persistent storage device such as SD.
Malicious files injected on rootfs remains undetected.
Secure Boot along with IMA-EVM.
Secure Boot
U-Boot
Boot Script
Firmware's (MC, DPL etc)
Kernel, Initramfs
Linked using secure boot chain of trust.
Integrity verified rootfs using IMA EVM by kernel.
Rootfs
Rootfs on persistent storage device such as SD.
Rootfs files are integrity checked before their execution.
Chain of Trust with IMA EVM enabled system.

```
Ubuntu 16.04.4 LTS localhost ttyS0

Welcome to Ubuntu 16.04.4 LTS (GNU/Linux 4.14.47-00012-g60ffb534ae8f aarch64)
*Documentation: https://help.ubuntu.com

root@localhost:#
```

```
Ubuntu 16.04.4 LTS localhost ttyS0

Welcome to Ubuntu 16.04.4 LTS (GNU/Linux 4.14.47-00012-g60ffb534ae8f aarch64)
*Documentation: https://help.ubuntu.com

root@localhost:#
```

Figure 37. Chain of Trust with IMA EVM enabled system

In normal secure boot process, contents of root file system mounted over persistent storage device are not validated by any mechanism and hence cannot be trusted. Any malicious changes in non-trusted rootfs contents are undetected. IMA EVM is the linux standard mechanism to verify the integrity of the rootfs. Integrity checks over file attributes and its contents are performed by linux IMA EVM module before its execution. IMA EVM depends on encrypted key loaded on user's keyring. Loading keys to root user keyring and enabling EVM is typically done using initramfs image. The initramfs image is validated using secure boot process and becomes the part of chain of trust. Initramfs switches control to main rootfs mounted over storage device, after EVM is successfully enabled on the system.

For secure boot process see, Secure boot process on page 176

6.2.2 Secure key and its blob

Secure key is generated using CAAM security engine which constitutes of random bytes. The key contents are stored in kernel space and not visible to user. User space will only be able to see the key blob.

Blobs are special data structures for holding encrypted data, along with an encrypted version of the key used to decrypt the data. Typically, blobs are used to hold data to be stored in external storage (such as flash memory or in an external file system), making the contents of a blob a semi-persistent secret. The secrecy of the blob depends on the device-specific 256-bit master key, which is derived from the OTMPK or ZMK on Layerscape trust architecture-based SoCs.

Secure key is loaded on the user keyring by the kernel keys framework and its blob is stored on the rootfs to be used during next bootup.

6.2.3 EVM Key on user keyrings

The EVM security attribute depends on an encrypted key (named evm-key) loaded on the user keyring. The encrypted key is derived by the kernel using the master key. The master key can be of following types:

- User-Key
- Secure-Key
- Trusted-Key

Secure and trusted keys are derived using a hardware security engine for greater security while the security of user-key depends upon the user-defined mechanisms irrespective of the hardware. The secure-key is derived using the Layerscape's SEC (aka CAAM). The trusted-key can be used on the platforms supporting TPM.

The encrypted key acts as an HMAC key, which is subsequently used to calculate the HMAC value (security.evm) over other security attributes. This key is stored internally by the kernel and user can only see its blob.

6.2.4 Enabling secure key and encrypted key in kernel config

The secure key is dependent on the Layerscape SoCs SEC (also known as: CAAM) hardware. To enable the secure key following SEC related configs needs to be enabled under **Hardware crypto devices** config option. See below spcified hierarchy for the config options:

```
Cryptographic API
 |____Hardware crypto devices
     |______ Freescale CAAM-Multicore platform driver backend
     |______ Freescale CAAM Job Ring driver backend
     |______ Register algorithm implementation with Crypto API
     |______ Queue Interface as Crypto API backend
     |______ Register hash algorithm implementations with Crypto API
     |______ Register CAAM device for hwrng API
```

After configuring basic CAAM options, enable the secure key and the encrypted key under Security options.

```
Security options
    |________ SECURE_KEYS
    |________ ENCRYPTED_KEYS
```

NOTE

1. To enable any option press Y.
2. If any option is enabled by default, there is no need to press Y.

6.2.5 Enabling IMA EVM integrity options in kernel image

Enable IMA EVM integrity options under Security options as shown below:

```
Security options
    |________ Integrity subsystem
    |________ Enables integrity auditing support
    |________ Integrity Measurement architecture (IMA)
    |________ Appraise integrity measurements
    |________ EVM support
    |________ FSUUID (version 2)
```

6.2.6 Modes of operation in IMA EVM

IMA EVM is enabled in two modes: fix mode, enforce mode. To enable a system with IMA EVM, both modes must be implemented in a sequence as described below.

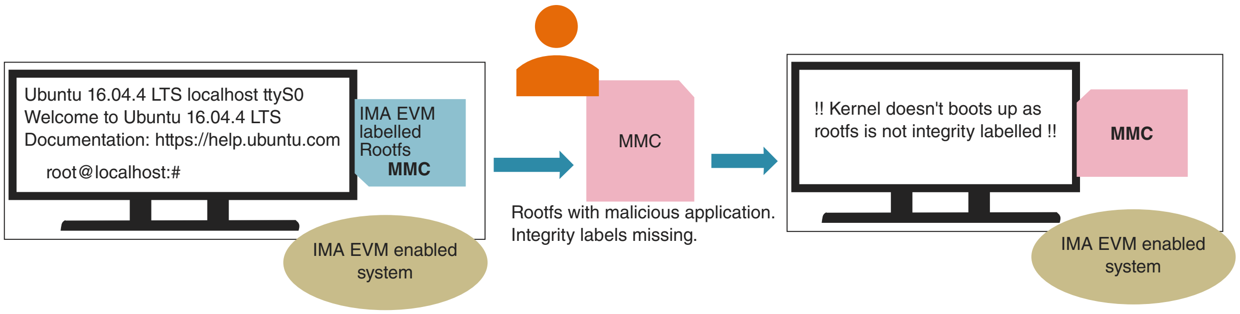
1. First, the system needs to be booted in fix mode with ima_appraise=fix and evm=fix bootargs. After loading the keys on the root keyring, the entire file system is labelled with security attributes. In fix mode any file with INTEGRITY_UNKNOWN is labelled with proper attribute values. This mode must be executed only once while preparing system for field deployment.
2. Second, after the fix mode execution is completed successfully, system needs to be booted IMA EVM in enforce mode. Enforce mode is enabled by setting ima_appraise=enforce bootargs. In enforce mode the files are measured against their “good” values. In case there is a mismatch between calculated security attribute value and stored value, access to that file is denied. While in field the system boot is done in enforce mode only.

6.2.7 IMA EVM feature use cases

IMA EVM prevents system from offline attack

Scenario: If attacker tries to boot the system using the rootfs mounted over persistent storage device for which the files are wrongly labelled or not labelled with security labels (such as security.ima, security.evm).

System behavior: In enforce mode, the IMA EVM labelled kernel stops booting due to mismatch in calculated security attribute values and labelled values.



```
Ubuntu 16.04.4 LTS localhost ttyS0
Welcome to Ubuntu 16.04.4 LTS
Documentation: https://help.ubuntu.com
    root@localhost:#
```

```
!! Kernel doesn't boots up as
rootfs is not integrity labelled !!
```

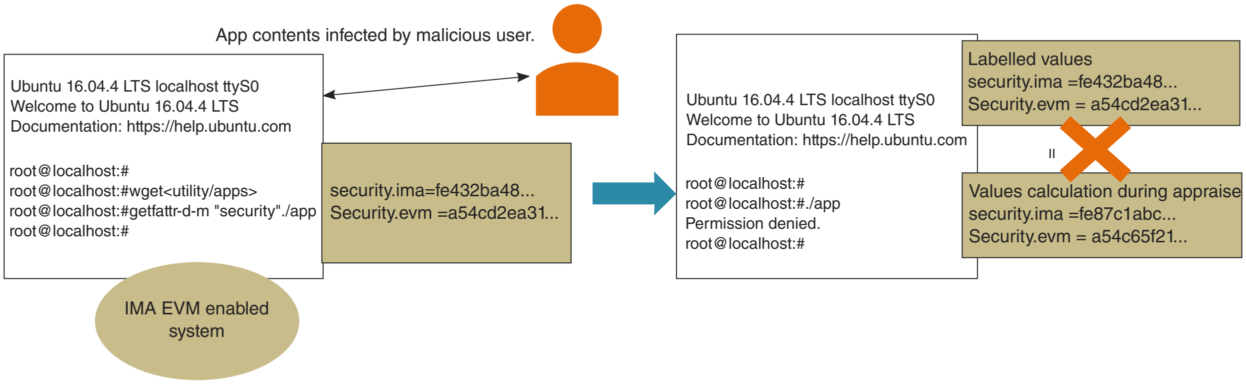
IMA EVM protect the system from offline attacks. System will not boot up if the files over rootfs are wrongly labelled or not labelled with security attributes.

Figure 38. IMA EVM protects system from offline attack

IMA EVM protects system against malicious file changes

Scenario: Any application loaded by the root user is changed maliciously. The labelled attribute values will be different from the calculated values on the next access.

System behavior: On next access, the file will not be accessible due to different security attribute values.



```
Ubuntu 16.04.4 LTS localhost ttyS0
Welcome to Ubuntu 16.04.4 LTS
Documentation: https://help.ubuntu.com

root@localhost:#
root@localhost:#wget<utility/apps>
root@localhost:#getfattr-d-m "security"./app
root@localhost:#
```

```
security.ima=fe432ba48...
Security.evm =a54cd2ea31...
```

```
Ubuntu 16.04.4 LTS localhost ttyS0
Welcome to Ubuntu 16.04.4 LTS
Documentation: https://help.ubuntu.com

root@localhost:#
root@localhost:#./app
Permission denied.
root@localhost:#
```

```
Labelled values
security.ima =fe432ba48...
Security.evm = a54cd2ea31...
```

```
Values calculation during appraise
security.ima =fe87c1abc...
Security.evm = a54c65f21...
```

Any app content changed by the malicious user would disturb the ima and evm attribute values. On next access the app will not be accessible

Figure 39. IMA EVM protect system from malicious file changes

NOTE

IMA EVM is independent of rootfs version i.e. it works with ubuntu rootfs 16.04 as well as ubuntu rootfs 18.04

6.2.8 Appendix A: Steps to enable IMA EVM using flex builder

To boot the SoC with the IMA EVM feature, various boot images can be generated using flex build. Secure boot process is mandatory to run the images generated using flex build.

Steps to generate images for IMA EVM along with secure boot:

1. Execute the commands below to enable configuration options for secure key and IMA-EVM feature in kernel as specified in Enabling secure key and encrypted key in kernel config on page 275 and Enabling IMA EVM integrity options in kernel image on page 275 respectively.

   ```
   $ flex-builder -c linux:custom -a <architecture>
   $ flex-builder -c linux -a <architecture>
   ```

2. Prepare initramfs images for arm32 or arm 64 architecture.

   ```
   $ flex-builder -i mkrfs -r buildroot:imaevm -a <architecture>
   ```

3. Prepare distro boot script images for the desired platform,

   ```
   $ flex-builder -i mkdistroscr -m <platform> -t
   ```

4. Sign the images (initramfs image, boot scripts and others) for secure boot process.

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

5. Create the firmware image to be used:

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

6. Build the boot partition tar ball:

   ```
   $ flex-builder -i mkbootpartition -a arm64 -t
   ```

7. Use the flex installer to program the boot partition and rootfs on the storage device.
8. System can be booted to run in secure mode.
9. On first boot up system will run IMA EVM in fix mode, during this mode the boot script in boot partition will be replaced with the enforce mode boot script. Upon first successful boot up reboot the system. Now on every bootup system will run in enforce mode.

NOTE

- Platform to be used: LS1088ARDB, LS1021ATWR, LS1046ARDB, LS2088ARDB. (LS2088ARDB Rev F does not support SD card.)
- Architecture to be used is: ARM32 or ARM64
- Boot type to be used is: qspi, sd, etc.
- -t flag is specified to indicate trust using IMA EVM.

6.2.9 Appendix B: Standalone steps to enable IMA EVM

To boot the system with IMA EVM, follow the steps below:

1. Clone the kernel (dash-lts) repository and checkout to the latest tag.
2. To compile the kernel set arch and path for cross-compilation:

   ```
   $: export CROSS_COMPILE=<aarch64-toolchain>
   $: export ARCH=arm64
   ```

3. Make the default config:

   ```
   $: make defconfig lsdk.config
   ```

   NOTE

   Above command is for ARM64 platform, use the necesaary defconfigs required to build the default config.

4. Enable the config options for secure key and IMA-EVM feature in kernel as mentioned in Enabling secure key and encrypted key in kernel config on page 275 and Enabling IMA EVM integrity options in kernel image on page 275 respectively.
5. Build the kernel image using make command:

   ```
   $:make -j<no. of jobs>
   ```

6. Prepare the SD card, ext4 type partition (say mmcblk0p3) with Ubuntu rootfs image extracted over it.
7. Boot the board to U-Boot prompt.
8. Prepare the initramfs for specific architecture using flex build as mentioned in step 2 at Appendix A: Steps to enable IMA EVM using flex builder on page 277.
9. Using tftp or by some other means, place the initramfs image on DDR (say at location 0xa0000000).
10. Enable the IMA EVM in fix mode by adding the following boot arguments to present bootargs.

    rootwait ima_tcb ima_appraise=fix ima_appraise_tcb evm=fix enforcing=0

    ```
    $: setenv bootargs=” console=ttyS0,115200 root=/dev/mmcblk0p3 rw rootwait ima_tcb
    ima_appraise=fix ima_appraise_tcb evm=fix enforcing=0
    ```

    NOTE

    a. The default mount partition is 3rd partition over mmc device (mmcblk0p3).

    b. Set the mount option accordingly if ROOTFS is not mounted on mmcblk0p3 partition e.g. mount="usb or sata or mmcblkpX"

    c. This step to be executed only once while preparing the board in factory for field deployment.

11. Run bootm command:

    ```
    $: bootm <uImage_addr> <initramfs_img_addr> <platform_dtb_img_addr>
    ```

12. Once the system gets booted up, immediately perform reboot. This ensure the system is ready to be used in enforce mode.
13. To start with enforce mode, boot the system to U-Boot Prompt and repeat the steps 10 and 11.
14. Enable the IMA EVM in enforce mode by adding the following boot arguments to present bootargs.

    “ rootwait ima_tcb ima_appraise=enforce ima_appraise_tcb enforcing=1 ”

    ```
    $: setenv bootargs=“console=ttyS0,115200 root=/dev/mmcblk0p3 rw rootwait
    ima_tcb ima_appraise=enforce ima_appraise_tcb enforcing=1
    ```

    NOTE

    - The default mount partition is 3rd partition over mmc device (mmcblk0p3).
    - Set the mount option accordingly if ROOTFS is not mounted on mmcblk0p3 partition e.g. mount="usb or sata or mmcblkpX"

15. Run bootm command

    ```
    $: bootm <uImage_addr> <initramfs_img_addr> <platform_dtb_img_addr>
    ```

16. System is booted up in enforce mode and ready with IMA EVM feature.

    NOTE

    Bootargs as mentioned in step 10 and 14 can be set using the boot script along with the secure boot chain.

6.2.10 Appendix C: Steps to verify IMA EVM feature

Perform the following checks to ensure that IMA EVM is successfully enabled in enforce mode.

1. Secure keys and encrypted keys are enabled in kernel image: The Following kernel logs ensures that secure key and encrypted key is successful registered.

   ```
   [    3.887255] Key type secure registered
   [    3.892526] Key type encrypted registered
   ```

2. IMA EVM is enabled in kernel image. The following kernel logs ensures IMA EVM is enabled.

   ```
   [    3.896537] ima: No TPM chip found, activating TPM-bypass! (rc=-19)
   [    3.902804] ima: Allocated hash algorithm: sha256
   [    3.907542] evm: HMAC attrs: 0x1
   ```

3. System is up in enforce mode. The following logs from initramfs and kernel ensures enforce mode is enabled.

   ```
   [   31.945199] EXT4-fs (mmcblk0p3): mounted filesystem with ordered data mode. Opts:
   (null)
   Loading blobs
   [   31.965094] evm: key initialized
   ```

4. EVM attributes over a file can be checked using getfattr utility.

   ```
   root@localhost# getfattr -d -m “security” /path/to/file
   ```

5. Security attribute are appraised successfully upon changing any file contents. Below steps will verify the appraise functionality.

   ```
   root@localhost# vim test_file  // wrtite contents "abc"
   root@localhost# getfattr -d -m “security” /path/to/test_file
   root@localhost# vim test_file  // write contents "abcd"
   root@localhost# getfattr -d -m “security” /path/to/test_file
   ```

NOTE

Security attributes values fetched in step d will be different from `root@localhost# getfattr -d -m “security” /path/to/file`.

6.2.11 Appendix - D: Points to remember

When preparing a system in fix mode, if labelling of the file system is interrupted (perhaps due to system power failure), remount the rootfs over the SD card partition before running fix mode again.

.

6.3 Trusted Execution (OP-TEE)

6.3.1 Introduction

Trusted Execution Environment (TEE), for ARM-based chips supporting TrustZone technology.

NXP Platforms are enabled with Open Portable TEE (OP-TEE), which is an open source project which contains a full implementation to make up a complete Trusted Execution Environment. This component meets the Global Platform TEE System Architecture specification. It also provides the TEE Internal core API v1.1 as defined by the Global Platform TEE Standard for the development of Trusted Applications.

OP-TEE consists of three components.

- OP-TEE Client, which is the client API running in normal world user space.
- OP-TEE Linux Kernel driver, which is the driver that handles the communication between normal world user space and secure world.
- OP-TEE Trusted OS, which is the Trusted OS running in secure world.

OP-TEE OS is made of 2 main components: the OP-TEE core and a collection of libraries designed for being used by Trusted Applications. While OP-TEE core executes in the ARM CPU privileged level (also referred to as 'kernel land'), the Trusted Applications execute in the non-privileged level (also referred to as the 'userland'). The static libraries provided by the OP-TEE OS enable Trusted Applications to call secure services executing at a more privileged level.

6.3.1.1 Support Platform

OP-TEE is supported on following NXP boards:

- **LS1046A-RDB**
- **LS1043A-RDB**
- **LS2088A-RDB**
- **LS1088A-RDB**
- **LS1012A-RDB**
- **LS1012A-FRWY**

6.3.1.2 Test Sequence

Execute the test sequence specified below on target machine:

Refer to the steps below to verify if the OP-TEE is successfully:

- Loaded (refer Appendix A for more details),
- Initialized (refer Appendix B for more details) &
- Running (refer Appendix C: Runtime flow of OP-TEE on page 283Appendix C for more details)

On the target NXP board:

- To check if the OP-TEE kernel driver is successfully initialized (after successfully communicating with OP-TEE OS running in OP-TEE), look for the following in Linux boot logs:

  ```
  optee: probing for conduit method from DT.
  optee: initialized driver
  ```

  Note: ppa.itb must be loaded with `loadable`

  ```
   node. Else, an error appears : optee: api uid mismatch
  ```

- Now, run the tee-supplicant (binary generated from optee_client repo) binary

```
$>: tee-supplicant & (press enter).
```

- Run the **xtest**(binary generated from optee_test repo) app as follows:

  ```
   $>: xtest -l 15 (press enter and look for the below logs to verify app runs successfully):
            47123 subtests of which 0 failed
            79 test cases of which 0 failed
            0 test case was skipped
            OP-TEE test application done!
  ```

6.3.2 Appendix A: Loading OP-TEE OS binary

1. OP-TEE binary is part of ppa.itb image as `loadables` node in ppa.itb image.
2. Corresponding `ppa.its` file content is as follows:

   ```
   printf "/dts-v1/;\n" > $1
   printf "\n" >> $1
   printf "/{\n" >> $1
   printf "   description = \"PPA Firmware\";\n" >> $1
   printf "   #address-cells = <1>;\n" >> $1
   printf "   images {\n" >> $1
   ```

```
printf "           firmware@1 {\n" >> $1
printf "                 description = \"PPA Firmware: Version %s\";\n" $VERSION >> $1
printf "                 data = /incbin/(\"../obj/monitor.bin\");\n" >> $1
printf "                 type = \"firmware\";\n" >> $1
printf "                 arch = \"arm64\";\n" >> $1
printf "                 compression = \"none\";\n" >> $1
printf "           };\n" >> $1
printf "           trustedOS@1 {\n" >> $1
printf "                 description = \"Trusted OS\";\n" >> $1
printf "                 data = /incbin/(\"../../tee.bin\");\n" >> $1
printf "                 type = \"OS\";\n" >> $1
printf "                 arch = \"arm64\";\n" >> $1
printf "                 compression = \"none\";\n" >> $1
printf "                 load = <0x00200000>;\n" >> $1
printf "           };\n" >> $1
printf "    };\n" >> $1
printf "\n" >> $1
printf "    configurations {\n" >> $1
printf "           default = \"config@1\";\n" >> $1
printf "           config@1 {\n" >> $1
printf "                 description = \"PPA Secure firmware\";\n" >> $1
printf "                 firmware = \"firmware@1\";\n" >> $1
printf "                loadables = \"trustedOS@1\";\n" >> $1
printf "           };\n" >> $1
printf "    };\n" >> $1
printf "};\n" >> $1
```

3. U-Boot parses the `ppa.itb` image and check if any `loadables` node is present in `ppa.itb` image.

   From U-Boot console logs, it can be seen if loadable present or not:

   ```
   Using SERDES1 Protocol: 5205 (0x1455)
              SEC0: RNG instantiated
              FSL_SDHC: 0
              PPA Firmware: Version LSDK-1712-TC1-dirty
              SEC Firmware: 'loadables' present in config
              MCFGR12201
              Resetting Job ring %p
              ….
   ```

4. If `loadables` is present, then U-Boot loads the OP-TEE OS binary `tee.bin`.
5. As part of loading U-Boot:

   Copy the binary `tee.bin` to DDR.

   OP-TEE binary is authenticated as part of ppa.itb image authentication using Secure Boot header
6. After loading OP-TEE, U-Boot passes the address where OP-TEE is loaded via SCRATCHRW registers to PPA.

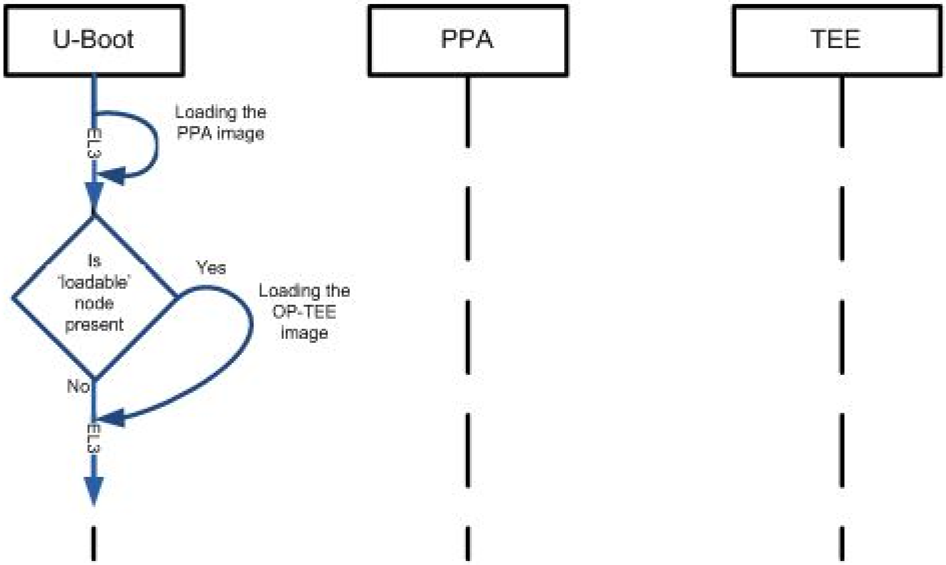


Figure 40. Loading Flow of OP-TEE OS

6.3.3 Appendix B: Initialization flow of OP-TEE OS

1. After the binary tee.bin is loaded to the DDR, U-Boot initiate the PPA initialization.
2. As part of OP-TEE initialization, PPA checks if `loadables` load address is not null, it initializes that OP-TEE.
3. Once the OP-TEE initialization is done, PPA initialization resumes. Before exiting to U-Boot after its initialization, PPA change the exception level from EL3 to EL2. Now, U-Boot run in EL2 mode.

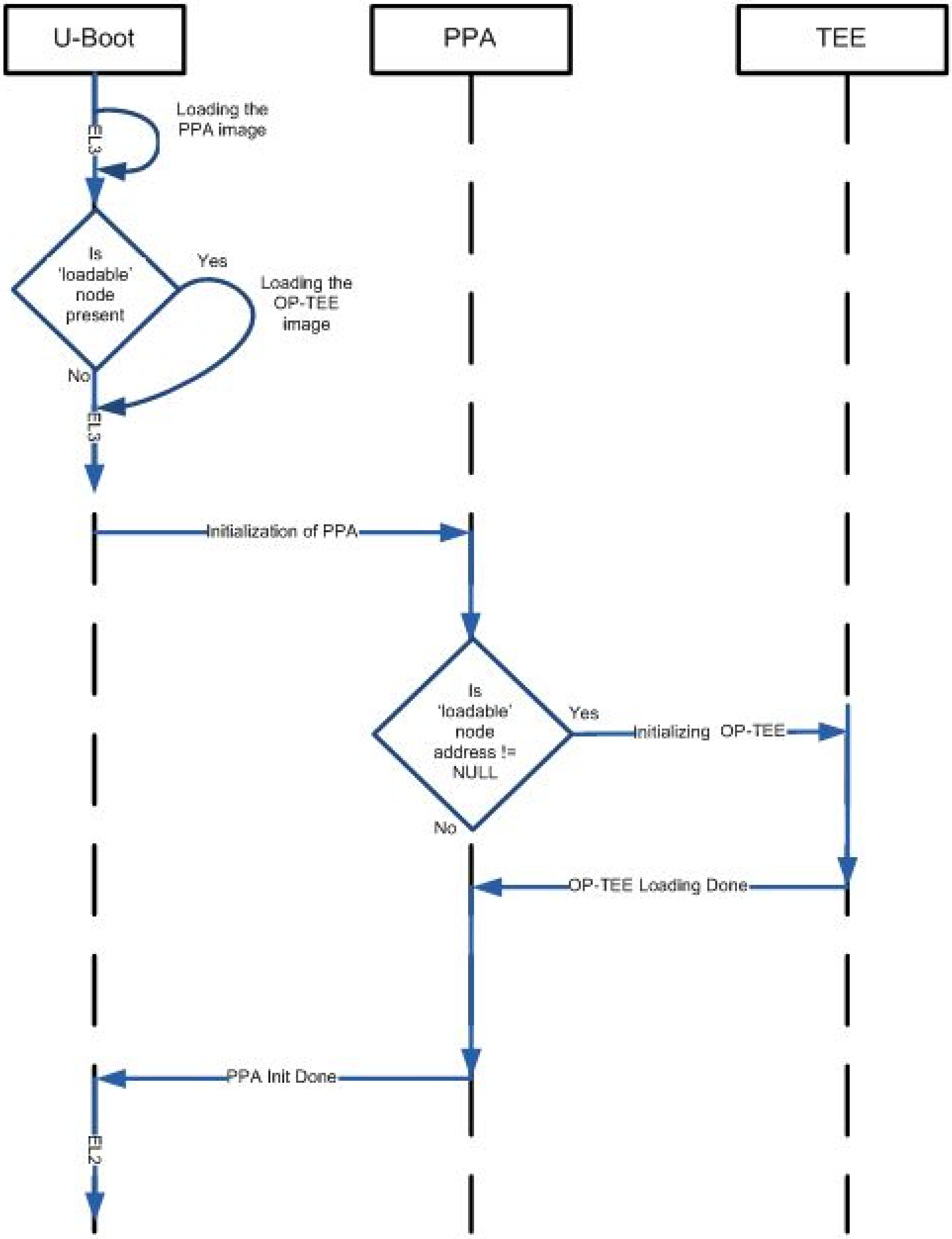
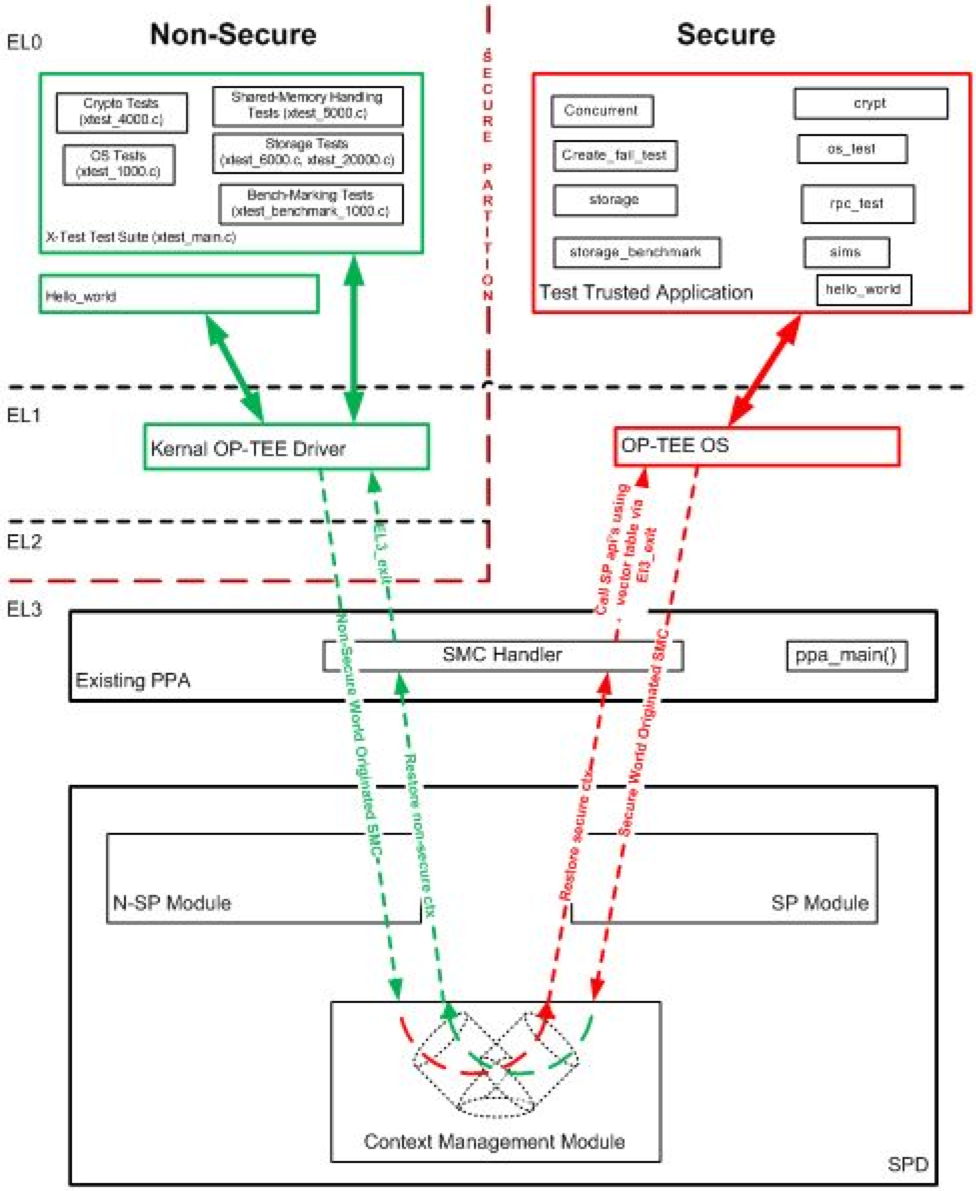


Figure 41. Loading and initialization flow of OP-TEE OS

6.3.4 Appendix C: Runtime flow of OP-TEE

To understand the runtime flow of OP-TEE, refer the figure below, which explains how the context switching is done between OP-TEE and Rich Execution Environment (Linux) (REE).



6.4 Fuse Provisioning User Guide

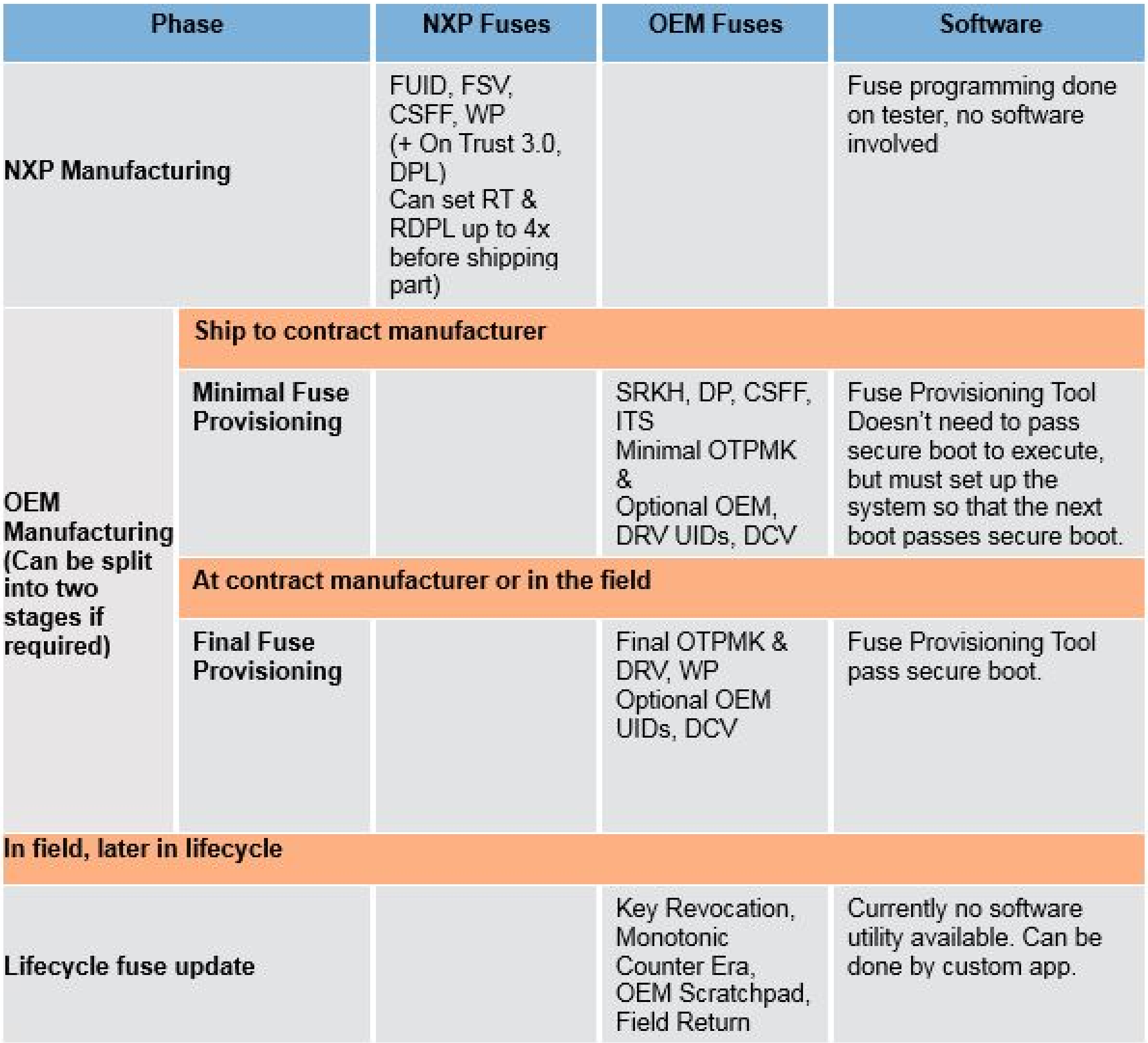
6.4.1 Introduction

NXP SoC's Trust Architecture provides non-volatile secure storage in form of on-chip fuse memory. Following information can be programmed into fuse memory via Security Fuse Processor (SFP):

- One Time Programmable Master Key Registers (OTPMKRs)
- Super Root Key Hash Registers (SRKHRs)
- Debug Challenge and Response Value Registers (DCVRs & DRVRs)
- OEM Security Policy Registers (OSPRs)

- OEM Unique ID/Scratch Pad Registers (OUIDRs)

6.4.2 Fuse Programming Scenarios



Phase		NXP Fuses	OEM Fuses	Software
NXP Manufacturing		FUID, FSV, CSFF, WP (+ On Trust 3.0, DPL) Can set RT & RDPL up to 4x before shipping part)		Fuse programming done on tester, no software involved
OEM Manufacturing (Can be split into two stages if required)	Ship to contract manufacturer			
	Minimal Fuse Provisioning		SRKH, DP, CSFF, ITS Minimal OTPMK & Optional OEM, DRV UIDs, DCV	Fuse Provisioning Tool Doesn't need to pass secure boot to execute, but must set up the system so that the next boot passes secure boot.
	At contract manufacturer or in the field			
	Final Fuse Provisioning		Final OTPMK & DRV, WP Optional OEM UIDs, DCV	Fuse Provisioning Tool pass secure boot.
In field, later in lifecycle				
Lifecycle fuse update			Key Revocation, Monotonic Counter Era, OEM Scratchpad, Field Return	Currently no software utility available. Can be done by custom app.

6.4.2.1 Fuse Provisioning during OEM Manufacturing

This stage may be split into two stages:

Stage 1 (Non-secure boot) – Minimal Fuse Provisioning

The following few fuses (Minimal Fuse File) programmed for secure boot to run:

- SRKH
- DP
- CSFF
- Minimal OTPMK
- ITS.

This stage does not pass secure boot to execute, but must set up the system so that the next boot passes secure boot. If this step happens in a trusted environment, OEM can choose to blow all the fuses in this stage itself.

Stage 2 (Secure Boot) – Final Fuse Provisioning

Rest of the fuses can be programmed after secure boot is up and running. This step ends with OEM WP fuse getting blown which renders most of the fuses as un- writable.

6.4.3 Fuse Provisioning Utility

PPA/secure firmware provides support to do the fuse provisioning. By default, the support is enabled and requires a built in. Steps to do so using flex build are available in Steps to build fuse provisioning firmware image on page 289.

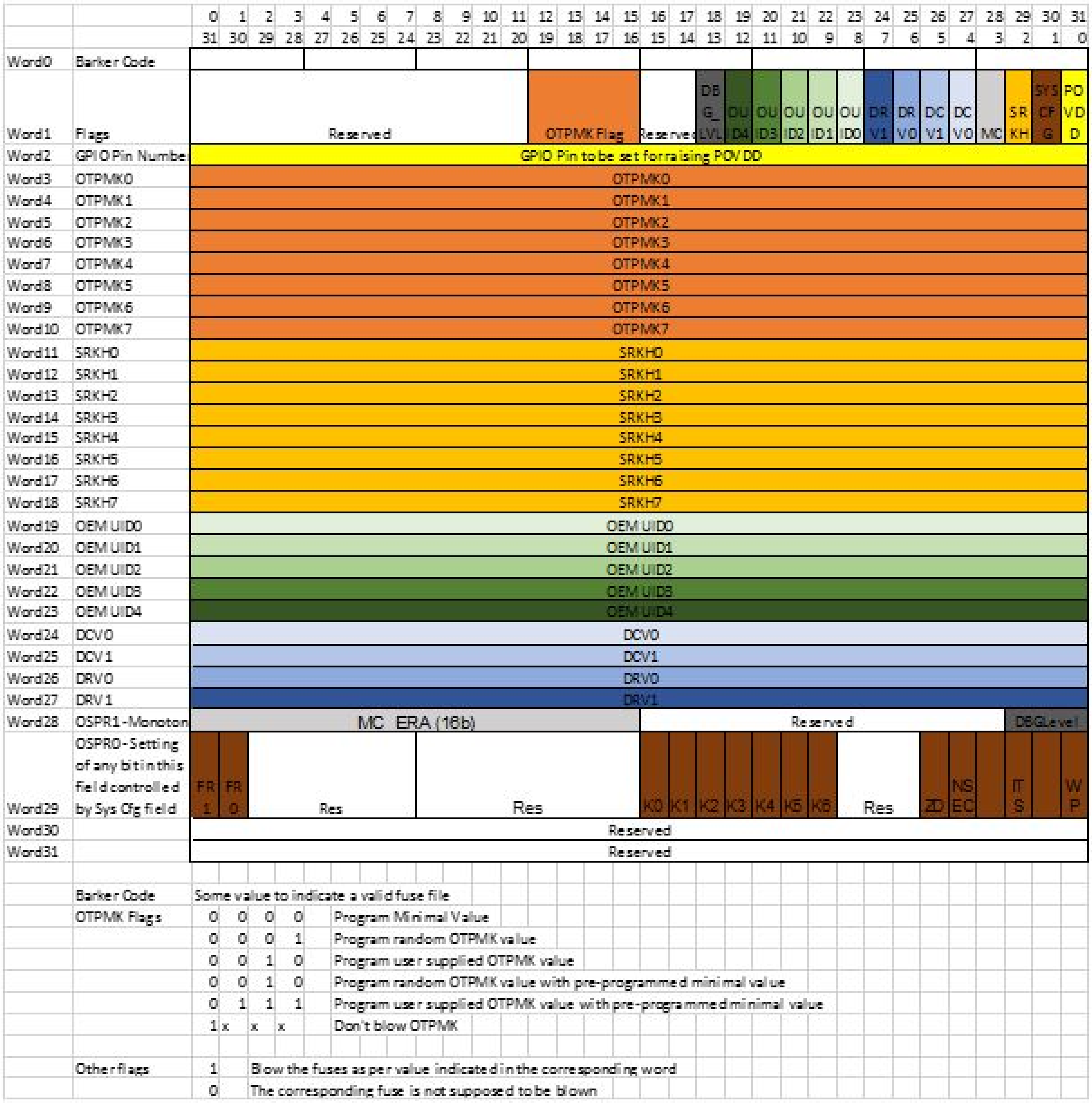
The information about the fuse values to be blown to be provided via a fuse file. The fuse file is a binary file with bits to indicate what fuses to be blown and their corresponding values.

CST provides an input file where user can enter the required values. Tool generates a Fuse file which is parsed in PPA to do fuse provisioning.

PPA would have the required checks to determine if the provided input values are correct or not.

For example, OTPMK, SRKH etc. cannot be programmed when OEM_WP is already set in SFP fuses.

6.4.3.1 Fuse file structure



		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
		31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Word0	Barker Code																																
Word1	Flags	Reserved												OTPMK Flag				Reserve		DBG_LVL	OUID4	OUID3	OUID2	OUID1	OUID0	DRV1	DRV0	DCV1	DCV0	MC	SRKH	SYSCFG	POVDD
Word2	GPIO Pin Numbe	GPIO Pin to be set for raising POVDD																															
Word3	OTPMK0	OTPMK0																															
Word4	OTPMK1	OTPMK1																															
Word5	OTPMK2	OTPMK2																															
Word6	OTPMK3	OTPMK3																															
Word7	OTPMK4	OTPMK4																															
Word8	OTPMK5	OTPMK5																															
Word9	OTPMK6	OTPMK6																															
Word10	OTPMK7	OTPMK7																															
Word11	SRKH0	SRKH0																															
Word12	SRKH1	SRKH1																															
Word13	SRKH2	SRKH2																															
Word14	SRKH3	SRKH3																															
Word15	SRKH4	SRKH4																															
Word16	SRKH5	SRKH5																															
Word17	SRKH6	SRKH6																															
Word18	SRKH7	SRKH7																															
Word19	OEM UID0	OEM UID0																															
Word20	OEM UID1	OEM UID1																															
Word21	OEM UID2	OEM UID2																															
Word22	OEM UID3	OEM UID3																															
Word23	OEM UID4	OEM UID4																															
Word24	DCV0	DCV0																															
Word25	DCV1	DCV1																															
Word26	DRV0	DRV0																															
Word27	DRV1	DRV1																															
Word28	OSPR1-Monoton	MC ERA (16b)																Reserved													DBGLevel		
Word29	OSPR0-Setting of any bit in this field controlled by Sys Cfg field	FR1	FR0	Res						Res								K0	K1	K2	K3	K4	K5	K6	Res			ZD	NSEC		ITS		WP
Word30		Reserved																															
Word31		Reserved																															

Barker Code	Some value to indicate a valid fuse file				
OTPMK Flags	0	0	0	0	Program Minimal Value
	0	0	0	1	Program random OTPMK value
	0	0	1	0	Program user supplied OTPMK value
	0	0	1	0	Program random OTPMK value with pre-programmed minimal value
	0	1	1	1	Program user supplied OTPMK value with pre-programmed minimal value
	1	x	x	x	Don't blow OTPMK
Other flags	1	Blow the fuses as per value indicated in the corresponding word			
	0	The corresponding fuse is not supposed to be blown			

6.4.3.2 Sample input file for fuse provisioning tool

```
---------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform - LS1/LS1043/LS1012/LS1046
PLATFORM=LS1046
---------------------------------------------------
# GPIO Pin to be set for raising POVDD [Optional]
POVDD_GPIO=
---------------------------------------------------
# One time programmable master key flags in binary form.[Mandatory]
# 0000 -> Program default minimal OTPMK value
# 0001 -> Program random OTPMK value
# 0010 -> Program user supplied OTPMK value
# 0101 -> Program random OTPMK value with pre-programmed minimal value
# 0110 -> Program user supplied OTPMK value with pre-programmed minimal value
# 1xxx -> Don't blow OTPMK
OTPMK_FLAGS=0000
# One time programmable master key value.
# [Optional dependent on flags, Mandatory in case OTPMK_FLAGS="0010" or "0110"]
OTPMK_0=
OTPMK_1=
OTPMK_2=
OTPMK_3=
OTPMK_4=
OTPMK_5=
OTPMK_6=
OTPMK_7=
---------------------------------------------------
# Super root key hash [Optional]
SRKH_0=
SRKH_1=
SRKH_2=
SRKH_3=
SRKH_4=
SRKH_5=
SRKH_6=
SRKH_7=
---------------------------------------------------
# Specify OEM UIDs. [Optional]
# e.g OEM_UID_0=11111111
OEM_UID_0=
OEM_UID_1=
OEM_UID_2=
OEM_UID_3=
OEM_UID_4=
---------------------------------------------------
# Specify Debug challenge and response values. [Optional]
# e.g DCV_0=11111111
DCV_0=
DCV_1=
DRV_0=
DRV_1=
---------------------------------------------------
# Specify Debug Level in binary form. [Optional]
# 000 -> Wide open: Debug portals are enabled unconditionally.
# 001 -> Conditionally open via challenge response, without notification.
# 01x -> Conditionally open via challenge response, with notification.
# 1xx -> Closed. All debug portals are disabled.
DBG_LVL=
---------------------------------------------------
# System Configuration register bits in binary form [Optional]
# WP (OEM write protect)
# ITS (Intent to Secure)
# NSEC (Non secure)
# ZD (ZUC Disable)
# K0,K1,K2 (Key revocation bits)
# FR0 (Field return 0)
# FR1 (Field return 1)
WP=
ITS=
NSEC=
ZD=
K0=
K1=
K2=
FR0=
FR1=
---------------------------------------------------
# Specify the output fuse provisioning file name. (Default:fuse_scr.bin) [Optional]
OUTPUT_FUSE_FILENAME=fuse_scr.bin
---------------------------------------------------
```

6.4.4 Steps to build fuse provisioning firmware image

Use following Flexbuild commands to build composite fuse provisioning firmware image. For detailed info regarding usage of Flexbuild, refer to Layerscape SDK user guide.

1. Command to build Code Signing Tool (CST):

   ```
   $:> flex-builder -c cst
   ```

2. Command to generate PPA image with fuse provisioning support:

   ```
   $:> flex-builder -c ppa-fuse -m ls1046ardb
   ```

3. Command to generate firmware image for SD boot source:

   Before running this command make CONFIG_BUILD_PPA_FUSE=y in configs/build_lsdk_xx.cfg in flexbuild to enable ppa-fuse.

   ```
   $:> flex-builder -i mkfw -m ls1046ardb -b sd -B uboot
   ```

4. Optional to edit input file used for fuse provisioning present here:

   "`<flexbuild_dir>/packages/apps/cst/input_files/gen_fusescr/ls104x_1012/input_fuse_file`

   ". Again, repeat above steps 2 & 3 to generate composite image.

5. Composite firmware image present here:

```
<flexbuild_dir>/build/images/firmware_ls1046ardb_uboot_sdboot.img
```

6.4.5 Deploy and run fuse provisioning

6.4.5.1 Enable POVDD for SFP

1. LS2088A RDB Board
   - Put J12 to enable PWR_PROG_SFP
2. LS1088A RDB Board
   - Put J10 to enable PWR_PROG_SFP.
3. LS1046A RDB Board
   - Put J21 to enable PWR_PROG_SFP

For more details to enable POVDD on board, refer to section Prepare board for Secure Boot on page 70 (point 1 only).

6.4.5.2 Deploy firmware image on board

1. Program composite firmware image (firmware_ls1046ardb_uboot_<boot-source>boot.img) built using Steps to build fuse provisioning firmware image on page 289, on corresponding boot-source using U-Boot commands as shown in below example for SD boot-source:

```
=> tftp a0000000 firmware_ls1046ardb_uboot_sdboot.img
 => mmc write a0000000 8 1fff8
```

6.4.5.3 Run firmware image on board

1. Execute the following U-Boot command to switch to boot-source. Below is example to switch to SD boot-source on LS1046ARDB:

   ```
   => cpld reset sd
   ```

2. Check for following logs as part of U-Boot boot logs:

```
PPA Firmware:<Version Info>
SEC Firmware: 'loadables' present in config
loadables:'fuse_scr'
```

3. On U-Boot prompt, check for any error code in DCFG scratch 4 register for any Error Codes on page 290 as follows:

   ```
   => md 1ee020c 1
   ```

4. If above “md” command shows that no error as follows, then fuse provisioning is successful:

```
01ee020c: 00000000
```

6.4.6 Validation

The procedure specified above is fully validated and verified on LS1046ARDB, LS1088ARDB, and LS2088ARDB platforms.

6.4.7 Error Codes

Table 1: Error Codes

Error Code	Value	Description
ERROR_FUSE_BARKER	0x1	Occurs if fuse script not found.
ERROR_READFB_CMD	0x2	Occurs if SFP Read Fuse Box (READFB) command fails.
ERROR_PROGFB_CMD	0x3	Occurs if SFP Program Fuse Box (PROGFB) command fails.
ERROR_SRKH_ALREADY_BLOWN	0x4	Occurs if SRKH is already blown.
ERROR_SRKH_WRITE	0x5	Occurs if write to SRKH mirror registers fails.
ERROR_OEMUID_ALREADY_BLOWN	0x6	Occurs if OEMUID is already blown.
ERROR_OEMUID_WRITE	0x7	Occurs if write to OEMUID mirror registers fails.
ERROR_DCV_ALREADY_BLOWN	0x8	Occurs if DCV is already blown.
ERROR_DCV_WRITE	0x9	Occurs if write to DCV mirror registers fails.
ERROR_DRV_ALREADY_BLOWN	0xa	Occurs if DRV is already blown.
ERROR_DRV_HAMMING_ERROR	0xb	Occurs if write to DRV mirror registers gives hamming error.
ERROR_OTPMK_ALREADY_BLOWN	0xc	Occurs if OTPMK is already blown.
ERROR_OTPMK_HAMMING_ERROR	0xd	Occurs if write to OTPMK mirror registers gives hamming error.
ERROR_OTPMK_USER_MIN	0xe	Occurs if user supplied OTPMK does not have minimal OTPMK bits set in case where OTPMK flags represents to program user supplied OTPMK value with pre-programmed minimal value.
ERROR_OSPR1_ALREADY_BLOWN	0xf	Occurs if OSPR1 is already blown.
ERROR_OSPR1_WRITE	0x10	Occurs if write to OSPR1 mirror register fails.
ERROR_SC_ALREADY_BLOWN	0x11	Occurs if SysCfg is already blown.

Table continues on the next page...

Table continued from the previous page...

ERROR_SC_WRITE	0x12	Occurs if write to SysCfg mirror register fails.
ERROR_POVDD_GPIO_FAIL	0x13	Occurs if gpio number configured is incorrect.
ERROR_GPIO_SET_FAIL	0x14	Occurs if the gpio bit is not set correctly
ERROR_GPIO_RESET_FAIL	0x15	Occurs if the gpio bit reset is not reset to inital state.

Appendix A Manual steps to build PPA fuse image

CST

1. Clone cst from LSDK components.
2. Now make.

   ```
   $:> make
   ```
3. Default sample input file programs minimal OTPMK values only in fuse memory. Edit “*input_files/gen_fusescr/ls104x_1012/input_fuse_file*” file to select/change values to be programmed in fuses for LS1046 or LS1012. Edit "*input_files/gen_fusescr/ls2088_1088/input_fuse_file*" file to select/change values to be programmed in fuses for LS1088A and LS2088A.
4. To generate fuse_scr.bin, execute the following command:

```
$:> ./gen_fusescr input_files/gen_fusescr/<platform>/input_fuse_file
platform: ls104x_1012 for LS1046A or LS1012A
platform: ls2088_1088 for LS1088A or LS2088A
```

PPA

1. Clone ppa from LSDK components.
2. Set path for the following:

   ```
   $:> export CROSS_COMPILE=<aarch64-toolchain-path->
   ```
3. Copy fuse_scr.bin file

   ```
   $:> cp <cst-clone-dir>/fuse_scr.bin
               <ppa-clone-dir>/ppa/<soc folder>/
   ```
4. Now build.

   $:> ./build rdb-fit fuse <platform>

   NOTE

   platform is ls1046, ls1012, ls1088 or ls2088
5. Fuse provisioning PPA image is built in “`<ppa-clone-dir>/ppa/<soc folder>/build/obj/ppa.itb`” directory.

Appendix B Loading Fuse Script binary

1. Fuse script binary is a part of ppa.itb image as “loadables” node in ppa.itb image.
2. Corresponding ***ppa.its*** file content is as follows for better clarity.

   ```
   printf "/dts-v1/;\n" > $1
   printf "\n" >> $1
   printf "/{\n" >> $1
   printf "        description = \"PPA Firmware\";\n" >> $1
   printf "        #address-cells = < 1 >\n" >> $1
   printf "        images {\n" >> $1
   printf "                firmware@1 {\n" >> $1
   printf "                        description = \"PPA Firmware: Version %s\";\n" $VERSION
   >> $1
   ```

```
printf "                                data = /incbin/(\"../obj/monitor.bin\");\n" >> $1
printf "                                type = \"firmware\";\n" >> $1
printf "                                arch = \"arm64\";\n" >> $1
printf "                                compression = \"none\";\n" >> $1
printf "                        };\n" >> $1
printf "                        fuse_scr {\n" >> $1
printf "                                description = \"Fuse Script\";\n" >> $1
printf "                                data = /incbin/(\"../../fuse_scr.bin\");\n" >> $1
printf "                                type = \"firmware\";\n" >> $1
printf "                                arch = \"arm64\";\n" >> $1
printf "                                compression = \"none\";\n" >> $1
printf "                                load = <0x00180000>;\n" >> $1
printf "                        };\n" >> $1
printf "                };\n" >> $1
printf "\n" >> $1
printf "                configurations {\n" >> $1
printf "                        default = \"config@1\";\n" >> $1
printf "                        config@1 {\n" >> $1
printf "                                description = \"PPA Secure firmware\";\n" >> $1
printf "                                firmware = \"firmware@1\";\n" >> $1
printf "                                loadables = \"fuse_scr\";\n" >> $1
printf "                        };\n" >> $1
printf "                };\n" >> $1
printf "};\n" >> $1
```

3. U-Boot parses the ppa.itb image and check if any “loadables” node is present in ppa.itb image.
4. From U-Boot console logs, it can be seen if loadable present or not:

```
….
PPA Firmware: <Version Info>
SEC Firmware: 'loadables' present in config
loadables: 'fuse_scr'
….
```

5. If ‘loadable’ is present, then U-Boot loads the fuse script binary ‘fuse_scr.bin’.
6. As part of loading U-Boot:
   a. Copy the binary ‘fuse_scr.bin’ to DDR.
   b. Fuse script binary is authenticated as part of ppa.itb image authentication using Secure Boot header.

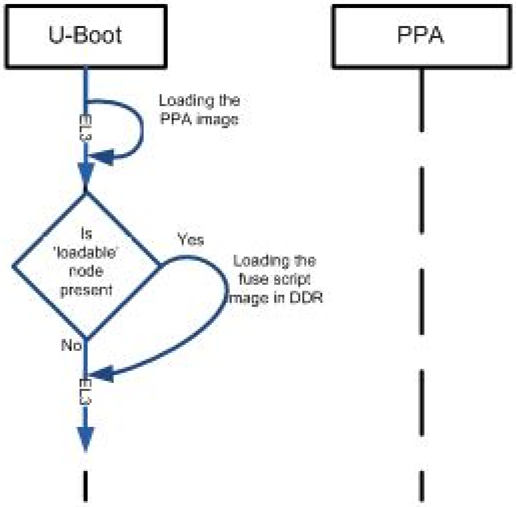

U-Boot
PPA
Loading the PPA image
EL3
Is 'loadable' node present
Yes
Loading the fuse script image in DDR
No
EL3

Figure 44. Loading Flow of Fuse Script

6.5 PKCS#11 and Secure Object Library

6.5.1 Introduction

NXP SoCs such as LS1046A can store keys securely using built-in SoC capabilities - virtual HSM. With such devices, sensitive private keys never leave the device and cryptographic operations are performed on this virtual HSM.

The PKCS#11 is a standard programming interface to communicate with HSMs. This standard specifies an application programming interface (API), called “Cryptoki” to devices which hold cryptographic information and perform cryptographic functions.

Proprietary interfaces using Secure Object Library are provided to interact with the HSM for:

- Generating key pair within the HSM.
- Installing existing key in the HSM.
- Manufacturing Protection key operations. (MPKey)

The private keys are never visible to normal world.

Sensitive Cryptographic operations using these keys can only be done using PKCS#11 cryptographic token standard.

An OpenSSL engine on Secure Object Library is also provided to interface directly with OpenSSL APIs

The PKCS#11 library release is compliant to v2.40. It is targeted for LS1046ARDB and supports

- RSA keys of size 1K and 2K.
- ECDSA keys curve prime256v1 and secp384r1.

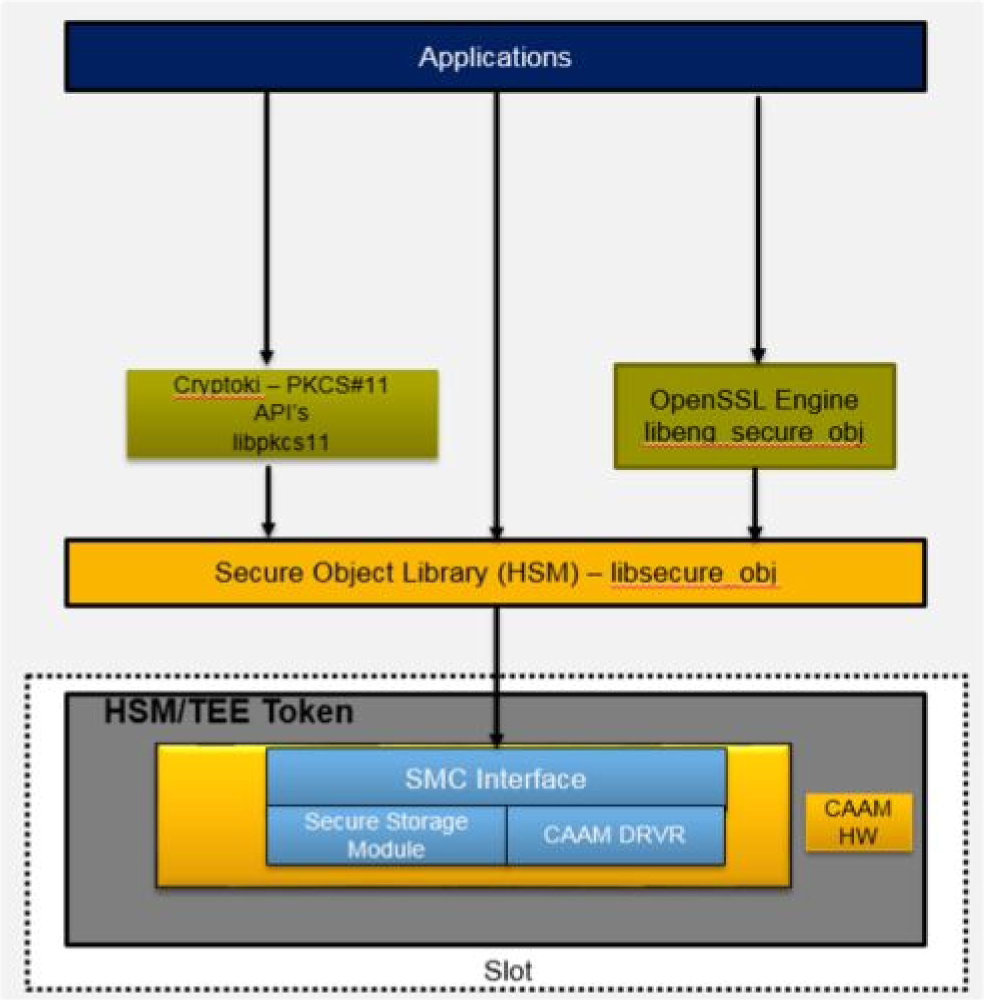


Figure 45. Block Diagram

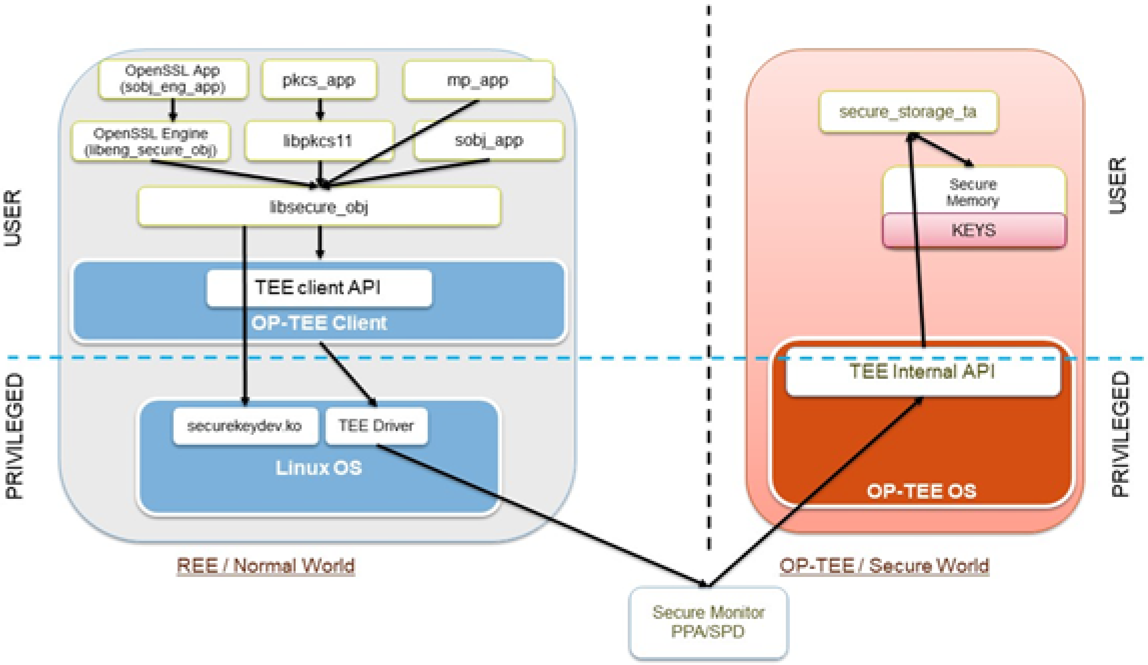


Figure 46. Details of HSM

6.5.2 Supported APIs

6.5.2.1 PKCS#11 Library – libpkcs11

The PKCS11 interfaces are exposed and implemented via a shared library with a name called libpkcs11.so (Cryptoki Library). Any PKCS11 library has a static CK_FUNCTION_LIST structure, and a pointer to it may be obtained by the C_GetFunctionList() function.

The table below, summarizes the list of supported PKCS11 interfaces. The return values and API behaviors are compliant with the PKCS11 standard v2.40. Library expects the caller to use them in a standard way.

API	Description
C_Initialize	Initialize Cryptoki library
C_Finalize	Clean up cryptoki related resources
C_GetFunctionList	Obtains entry points of Cryptoki library functions.
C_GetInfo	Obtains general information about Cryptoki
C_GetSlotInfo	Obtains information about a particular slot
C_GetTokenInfo	Obtains information about a particular token
C_GetSlotList	Obtain list of slots in the system. Only a fixed slot with fixed token is supported. Dynamic slot or token addition is not supported.

Table continues on the next page...

Table continued from the previous page...

C_OpenSession C_CloseSession C_CloseAllSessions	Opens/Closes a session. • "Read Only (R/O)" Public sessions are only supported. • This PKCS#11 library provides read-only access to token's public objects. This means that keys cannot be created/deleted/modified via the PKCS#11 interface. • R/O SO or R/W User/SO sessions are not supported.
C_FindObjectsInit C_FindObjects C_FindObjectsFinal	Objects search operations. RSA Public and Private key objects of size up to 2048bits are supported. ECDSA Public and Private key objects of size 256 & 384 bits are supported.
C_SignInit C_Sign	Initialize signature operation and signing single part data. Mechanisms supported: • RSA-based Mechanisms — CKM_RSA_PKCS — CKM_MD5_RSA_PKCS — CKM_SHA1_RSA_PKCS — CKM_SHA256_RSA_PKCS — CKM_SHA384_RSA_PKCS — CKM_SHA512_RSA_PKCS • ECDSA-based Mechanisms — CKM_ECDSA — CKM_ECDSA_SHA1
C_GetAttributeValue	Obtains the value of one or more attributes of the objects.
C_GetMechanismList	Obtains List of mechanism supported by token.
C_GetMechanismInfo	Obtains the information about a mechanism.

6.5.2.2 Secure Object Library – libsecure_obj

The following are the details of the supported interfaces to generate/import keys using the Secure Object library.

1. **Import Keys**

   SK_RET_CODE SK_CreateObject(SK_ATTRIBUTE *attr, uint16_t attrCount, SK_OBJECT_HANDLE *phObject);

   The API creates an Object on the HSM, and returns a handle to it. API always succeeds even if an object with same attributes exists in HSM. Duplicate object is created. Application needs to take care that duplicate objects should not be created.

   attr is an array of attributes that the object should be created with. Some of the attributes may be mandatory, such as SK_ATTR_OBJECT_TYPE and SK_ATTR_OBJECT_INDEX (the id of the object), and some are optional.

Application needs to take care that valid attributes are passed, library does not return any error on receiving inconsistent/ incompatible attributes.

param[in] attr: The array of attributes to be used in creating the Object.

param[in] attrCount: The number of attributes in attr

param[in, out] phObjectIN: A pointer to a handle (must not be NULL);

OUT: The handle of the created Object

Return Values:

SKR_OK Successful execution, phObject filled with created object handle.

SKR_ERR_BAD_PARAMETERS Invalid function arguments

SKR_ERR_OUT_OF_MEMORY Memory allocation failed.

SKR_ERR_NOT_SUPPORTED The function and/or parameters are not supported by the library.

-- Some internal error code other than mentioned above can be returned. Refer to securekey_api_types.h for error code description.

2. **Generate Key.**

   SK_RET_CODE SK_GenerateKeyPair(SK_MECHANISM_INFO *pMechanism, SK_ATTRIBUTE *attr, uint16_t attrCount, SK_OBJECT_HANDLE *phKey);

   This API generates key pair on the HSM, and returns a handle to it. API always succeeds even if an object with same attributes exists in HSM. Duplicate object is created. Application needs to take care that duplicate objects should not be created.

   pMechanism is mechanism for key pair generation. For example: SKM_RSA_PKCS_KEY_PAIR_GEN.

   attr is an array of attributes that the object should be created with. Some of the attributes may be mandatory, such as SK_ATTR_OBJECT_INDEX (the id of the object), and some are optional.

   Application needs to take care that valid attributes are passed, library does not return any error on receiving inconsistent/ incompatible attributes.

   param[in] pMechanism Mechanism for key pair generation

   param[in] attr The array of attributes to be used in creating the Object.

   param[in] attrCount The number of attributes in attr

   param[in, out] phKey IN: A pointer to a handle (must not be NULL);

   OUT: The handle of the created Object

   Return Values:

   SKR_OK Successful execution, phObject is filled with created object handle.

   SKR_ERR_BAD_PARAMETERS Invalid function arguments

   SKR_ERR_OUT_OF_MEMORY Memory allocation failed.

   SKR_ERR_NOT_SUPPORTED The function and/or parameters are not supported by the library.

   --Some internal error code other than mentioned above can be returned. Refer to securekey_api_types.h for error code description.

3. **Erase Object.**

   SK_RET_CODE SK_EraseObject(SK_OBJECT_HANDLE hObject);

   Erases an object from the HSM. This means that the object with the specified handle can no longer be used.

   param[in] hObject

The handle of the Object to be erased.

Return Values:

SKR_OK Successful execution SKR_ERR_BAD_PARAMETERS Invalid function arguments -- Some internal error code other than mentioned above to be returned. Refer to securekey_api_types.h for error code description.

Further details of the APIs and its types are available in the files <securekey_api.h> and <securekey_api_types.h> in folder **secure_obj.**

NOTE

1. Maximum of 50 objects can be created/generated as of now.
2. Secure Object Library will not be throwing any error if multiple objects having same attributes are being created. It is applications responsibility to take care of attributes that are passed during creation/generation of objects.

Manufacturing Key APIs:

Following secure boot, the system runs the key generation routine producing an ECC Public and Private Key pair, referred to as Manufacturing Protection Key Pair(MPKey).

Key Generation is performed by BootRom. APIs for getting MP Public key, signing using MP Private key and for getting the MP Tag are described below.

For complete documentation on how to perform the key generation, public key export, and signing with the ECC private key, refer to the Manufacturing-protection chip authentication

process section in the SoC’s Security (SEC) Reference Manual 5.6

NOTE: For this feature to work board must be booted in Secure Boot mode, with ITS bit set to 1.

1. **Get MP Public key.**

   enum sk_status_code sk_mp_get_pub_key(struct sk_EC_point *pub_key);

   Get Manufacturing Protection(MP) Public Key (ECC P256 Key).

   param[in,out] pub_key: This is MP Public Key to be returned. Application needs to allocate memory for sk_EC_point. Each of the coordinate x & y needs to allocate sk_EC_point.len memory. sk_EC_point.len can be obtained using **sk_mp_get_pub_key_len()**.

   Return Values:

   SK_SUCCESS on success, error value otherwise.

2. **Sign using MP Private Key**

   enum sk_status_code sk_mp_sign(unsigned char * msg, uint8_t msglen,

   struct sk_EC_sig * sig, uint8_t * digest, uint8_t digest_len)

   Sign the msg using MP Priv Key. While signing MP Message, it will be prepended to message. Message over which signature will be calculated = MP message + msg.

   param[in] msg: Pointer to the message to be signed.

   param[in] msglen: Length of the message to be signed.

   param[in,out] sig: This is Signature calculated. Application needs to allocate memory for sk_EC_sig. Each of the parts r & s needs to be allocated sk_EC_sig.len memory. sk_EC_sig.len can be obtained using sk_mp_get_sig_len().

   param[in, out] digest: Digest(SHA256) of the message to be signed. Digest is calculated by prepending MP Message to the msg.

   param[in] digest_len: Length of digest. Application needs to allocate memory for sk_EC_point. Each of the coordinate x & y needs to allocate sk_EC_point.len memory. sk_EC_point.len can be obtained using sk_mp_get_pub_key_len().

   Return Values:

SK_SUCCESS on success, error value otherwise.

3. **Get MP Tag.**

   enum sk_status_code sk_mp_get_mp_tag(uint8_t *mp_tag_ptr,uint8_t mp_tag_len);

   Get the MP Message. While signing, MP Message is prepended to message automatically. User can call this function to get MP message tag during verification operation.

   param[in, out] mp_tag_ptr: Pointer to the message to be signed. Application needs to allocate memory of length returned by **sk_mp_get_tag_len().**

   param[in] mp_tag_len: Length of the mp_tag_ptr buffer

   Return Values:

   SK_SUCCESS on success, error value otherwise.

   The API definition can be found in file **securekey_mp.h**. Sample applications have also been provided which demonstrate how to use APIs.

6.5.3 Integrating applications with Secure Object

There are following ways in which applications can interact with Secure Objects stored in HSM/Token.

- Using PKCS#11 APIs.
- Using Secure Object APIs.
- Applications using OpenSSL APIs
  - — Secure Object Library based OpenSSL Engine (libeng_secure_obj)
  - — PKCS#11 based OpenSSL Engine (Third party OpenSC/libp11)

6.5.3.1 Using PKCS#11 APIs

Applications can directly use the PKCS#11 APIs to interact with the Secure Objects stored in HSM/Token. Currently we support only PKCS#11 APIs mentioned in PKCS#11 APIs.

We have also created a reference application pkcs11_app for showing how to use the PKCS#11 APIs for writing your own application.

Commands to run **pkcs11_app** are shown here

6.5.3.2 Using Secure Object APIs

Applications can directly use the Secure Object Library APIs to interact with the Secure Objects stored in HSM/Token. Currently we support APIs mentioned in Secure Object APIs.

We have also created a reference application sobj_app for showing how to use the Secure Object APIs.

Commands to run **sobj_app** are shown here.

6.5.3.3 Applications using OpenSSL APIs

This topic provides examples of usage with OpenSSL. It is recommended that you should familiarize yourself with Open SSL.

Refer to the appropriate documents for Open SSL commands at the following location:

http://www.openssl.org/docs/

Open SSL provides the support of engine (basically hardware devices) to store the keys on hardware devices to make keys more secure.

There are 2 ways in which applications using the OpenSSL APIs can access the Secure Objects stored in HSM/Token.

- Secure Object Library based OpenSSL Engine (libeng_secure_obj).
- PKCS#11 based OpenSSL Engine (Third party OpenSC/libp11).

6.5.3.3.1 Secure Object Library based OpenSSL Engine (libeng_secure_obj)

NXP provides the Secure Object Library based OpenSSL Engine that is used to communicate with underlying HSM.

This engine is based on Secure Object Library, It does following things:

1. RSA Private Encryption.
2. RSA Private Decryption.
3. ECDSA Signing Operation.

All other RSA/ECDSA operations will be done by OpenSSL itself.

This engine does not support generation of RSA Keys. Keys will be generated via another app **sobj_app** and these keys are used in the applications using this OpenSSL Engine.

Refer **Running the sobj_eng_app** section for screenshots of app using OpenSSL engine.

Example Usage with OpenSSL

This topic provides examples of usage with OpenSSL:

- Using the engine from command Line.Change the following in `openssl.cnf` (often in `/etc/ssl/openssl.cnf`).

  This line must be placed at the top, before any sections are defined:

  openssl_conf = conf_section

  Add following section at bottom of file:

  ```
  [conf_section]
  engines = engine_section
  [engine_section]
  secure_obj = sobj_section
  [sobj_section]
  engine_id = eng_secure_obj
  dynamic_path = <path where lib_eng_secure_obj.so is placed>
  default_algorithms = RSA
  init = 1
  ```

  Note: This sections shows only RSA examples, same can be done for EC by changing *default_algorithms* in openssl.cnf as shown below:

  `default_algorithms = RSA, EC`

  Testing the engine operation:

  To verify that the engine is properly operating, you can use the following example:

  ```
  root@Ubuntu:~#
  root@Ubuntu:~# openssl engine
  (dynamic) Dynamic engine loading support
  (eng_secure_obj) secure object OpenSSL Engine.
  root@Ubuntu:~#
  root@Ubuntu:~#
  ```

  If you do not update the OpenSSL configuration file, specify the engine configuration explicitly.

  ```
  $: openssl engine -t dynamic -pre SO_PATH:<path-to-libeng_secure_obj.so> -pre
  ID:eng_secure_obj -pre LIST_ADD:1 -pre LOAD
  ```

  ```
  root@Ubuntu:~#
  root@Ubuntu:~# openssl engine -t dynamic -pre SO_PATH:/usr/lib/aarch64-linux-gnu/
  openssl-1.0.0/
  engines/libeng_secure_obj.so -pre ID:eng_secure_obj -pre LIST_ADD:1 -pre LOAD
  (dynamic) Dynamic engine loading support
  ```

```
[Success] : SO_PATH:/usr/lib/aarch64-linux-gnu/openssl-1.0.0/engines/libeng_secure_obj.SO
[Success] : ID:eng_secure_obj
[Success] : LIST_ADD:1
[Success] : LOAD
LOADED: (eng_secure_obj) Secure object OpenSSL Engine.
    [available]
root@Ubuntu:~#
```

- Using OpenSSL from the command line.

  Following commands can be used to generate RSA/ECDSA key-pair and use them in signing any data and verifying the signatures generated.

  ```
  $: sobj_app -G -m rsa-pair -s 2048 -l "rsa_gen_2048" -i 1 -w rsa_2048.pem ## Generating RSA
  key-pair ##
  $: openssl rsa -in rsa_2048.pem -pubout -out rsa_pub_2048.pem  ## Taking out Public Key for
  verifying signature ##
  $: openssl dgst -sha1 -sign rsa_2048.pem -out sig.data data  ## Generating Signature
  "sig.data" of "data" ##
  $: openssl dgst -sha1 -verify rsa_pub_2048.pem -signature sig.data data  ## Verifying the
  signature using Public Key ##
  ```

  Same thing can be done for ECDSA keys of prime256v1 by using following commands:

  ```
  $: sobj_app -G -m ec-pair -c prime256v1 -l "ecc_256" -i 2 -w ec256.pem
  $: openssl ec -in ec256.pem -pubout -out ec_pub_256.pem
  $: openssl dgst -sha1 -sign ec256.pem -out sig.data data
  $: openssl dgst -sha1 -verify ec_pub_256.pem -signature sig.data data
  ```

  For ECDSA secp384r1 curve us following commands:

  ```
  $: sobj_app -G -m ec-pair -c secp384r1 -l "ecc_384" -i 3 -w ec384.pem
  $: openssl ec -in ec384.pem -pubout -out ec_pub_384.pem
  $: openssl dgst -sha1 -sign ec384.pem -out sig.data data
  $: openssl dgst -sha1 -verify ec_pub_384.pem -signature sig.data data
  ```

- This section describes how to use the command line to create a self-signed certificate for "NXP Semiconductor". The key of the certificate is generated in the Secure Object HSM and will not exportable.

  As per the following examples, generate a private key in the HSM with sobj_app, This will also create a fake PEM file **“dev_key.pem”** having information to get the required key from HSM.

  Following command is generating RSA key-pair.

  ```
  $: sobj_app -G -m rsa-pair -s 2048 -l "Test_Key" -i 1 -w dev_key.pem
  ```

  ECDSA key-pair can also be generated using following command:

  ```
  $: sobj_app -G -m ec-pair -c prime256v1 -l "ecc_256" -i 30 -w dev_key.pem
  ```

  To generate a certificate with key in the Secure Object module, the following commands can be used:

  ```
  $ openssl req -new -key dev_key.pem -out req.pem -text -x509 -subj "/CN=NXP Semiconductor"
  $ openssl x509 -signkey dev_key.pem -in req.pem -out cert.pem
  ```

  The first command creates a self-signed Certificate for “NXP Semiconductor". The signing is done using the key specified by the fake PEM file.

  The second command creates a self-signed certificate for the request, the private key used to sign the certificate is the same private key used to create the request.

6.5.3.3.2 PKCS#11 based OpenSSL Engine (Third party OpenSC/libp11)

libp11 is a library implementing a thin layer on top of PKCS#11 API to make using PKCS#11 implementations easier.

You can get library from: https://github.com/OpenSC/libp11 .

This code repository produces two libraries:

- libp11 provides a higher-level (compared to the PKCS#11 library) interface to access PKCS#11 objects. It is designed to integrate with applications that use OpenSSL.

- pkcs11 engine plugin for the OpenSSL library allows accessing PKCS#11 modules in a semi-transparent way.

pkcs11 engine for OpenSSL can be installed on board using command **sudo apt-get install libengine-pkcs11-openssl**

Above command will install the libpkcs11.so (pkcs11 engine) in /usr/lib/aarch64-linux-gnu/engines-1.1/libpkcs11.so and this will be *dynamic_path* in OpenSSL onfiguration file.

For running the PKCS#11 OpenSSL Engine with our PKCS#11 Library add following into your global OpenSSL configuration file (often in /etc/ssl/openssl.cnf). This line must be placed at the top, before any sections are defined:

```
openssl_conf = openssl_init
```

This should be added to the bottom of the file:

```
[openssl_init]
engines=engine_section

[engine_section]
pkcs11 = pkcs11_section

[pkcs11_section]
engine_id = pkcs11
dynamic_path = <path-to-pkcs11-engine>/libpkcs11.so
MODULE_PATH = <path-to-NXP-pkcs11-library>/libpkcs11.so
init = 0
```

The dynamic_path value is the pkcs11 engine plug-in, the MODULE_PATH value is the NXP PKCS#11 library. The engine_id value is an arbitrary identifier for OpenSSL applications to select the engine by the identifier.

Testing the engine operation

To verify that the engine is properly operating you can use the following example.

```
$ openssl engine pkcs11 -t
(pkcs11) pkcs11 engine
[ available ]
```

Using p11tool and OpenSSL from the command line:

This section demonstrates how to use the command line to create a self-signed certificate for "NXP Semiconductor". The key of the certificate will be generated in the token and will not exportable.

p11tool from GnuTLS and this engine with OpenSSL work in combination.

p11tool is a tool that manipulate PKCS #11 tokens. Export/import data from PKCS #11 tokens. To use PKCS #11 tokens with gnutls the configuration file /etc/gnutls/pkcs11.conf must exist and contain number lines of the form "load=<pkcs-library-path>" or this PKCS#11 module can be provided directly as –provider in command line as argument.

p11tool can be installed by running command **sudo apt-get install gnutls-bin**

For more configuration options check: https://www.gnutls.org/manual/html_node/p11tool-Invocation.html.

Check for key which is already created from **sobj_app** via p11tool.

The following commands utilize p11tool for that.

```
$ p11tool --provider <path-to-NXP-PKCSC-library>/libpkcs11.so --list-privkeys
```

```
root@localhost:~# p11tool --provider /root/libpkcs11.so --list-privkeys
Object 0:
    URL: pkcs11:model=;manufacturer=NXP;serial=1;token=TEE_BASED_TOKEN;
%01%00%00%00;object=Device_Key3;type=private
    Type: Private Key
    Label: Device Key3
    Flags: CKA_NEVER_EXTRACTABLE; CKA_SENSITIVE;
    ID: 01:00:00:00
Object 1:
    URL: pkcs11:model=;manufacturer=NXP;serial=1;token=TEE_BASED_TOKEN;
%01%00%00%00;object=Device_Key2;type=private
    Type: Private Key
    Label: Device Key2
    Flags: CKA_NEVER_EXTRACTABLE; CKA_SENSITIVE;
    ID: 01:00:00:00

Object 0:
    URL: pkcs11:model=;manufacturer=NXP;serial=1;token=TEE_BASED_TOKEN;
%01%00%00%00;object=Device_Key3;type=private
    Type: Private Key
    Label: Device Key
    Flags: CKA_NEVER_EXTRACTABLE; CKA_SENSITIVE;
    ID: 01:00:00:00
root@localhost:~#
```

Note the PKCS #11 URL shown above and use it in the commands below.

To generate a certificate with its key in the PKCS #11 module, the following command can be used.

Following command creates a self-signed Certificate for "NXP Semiconductor". The signing is done using the key specified by the URL.

```
$ openssl req -engine pkcs11 -new -key
"pkcs11:model=;manufacturer=NXP;serial=1;token=TEE_BASED_TOKEN;id=
%01%00%00%00;object=Device_Key3
;type=private" -keyform engine -out req.pem -text -x509 -subj "/CN=NXP Semiconductor"
```

6.5.4 Board Bootup & Running applications

6.5.4.1 Board Bootup

1. Prepare the images using the LSDK documentation and bootup the board with **secure-boot and ITS set to 1.** ITS = 1 is required for bootrom to generate the Manufacturing Protection Private Key.

   For setting ITS bit to 1 run following command after programming SRK Hash and before removing the boot hold-off. The test is performed on LS1046ARDB.

   #To do ITS=1

   ccs::write_mem 32 0x1e80200 4 0 0x00000004

   You can refer here for documentation - https://lsdk.github.io/document.html

2. After booting up the board with LSDK1809 images, Check if following images are placed in corresponding places.

Binary	Place in rootfs
b05bcf48-9732-4efa-a9e0-141c7c888c34.ta	/lib/optee_armtz/
libsecure_obj.so	/usr/local/lib
sobj_app	/usr/local/bin
mp_app	/usr/local/bin
mp_verify	/usr/local/bin
libeng_secure_obj.so	/usr/lib/aarch64-linux-gnu/openssl-1.0.0/engines/
sobj_eng_app	/usr/local/bin
securekeydev.ko	This path depends on Linux Kernel Version:
	Linux Kernel 4.9 - /lib/modules/4.9.xx<commit-id>/extra/
	Linux Kernel 4.14 - /lib/modules/4.14.xx<commit-id>/extra/
libpkcs11.so	/usr/local/lib

Table continues on the next page...

Table continued from the previous page...

pkcs11_app	/usr/local/bin
thread_test	/usr/local/bin

For Compilation steps, refer **Appendix Section** at end of this document.

3. Run “**tee-supplicant &**” command from linux prompt.
4. Depending on linux kernel version used **“insmod securekeydev.ko”** from right folder**.**
5. Run the applications as described in Running the applications.

6.5.4.2 Running applications

Two applications are available with the package.

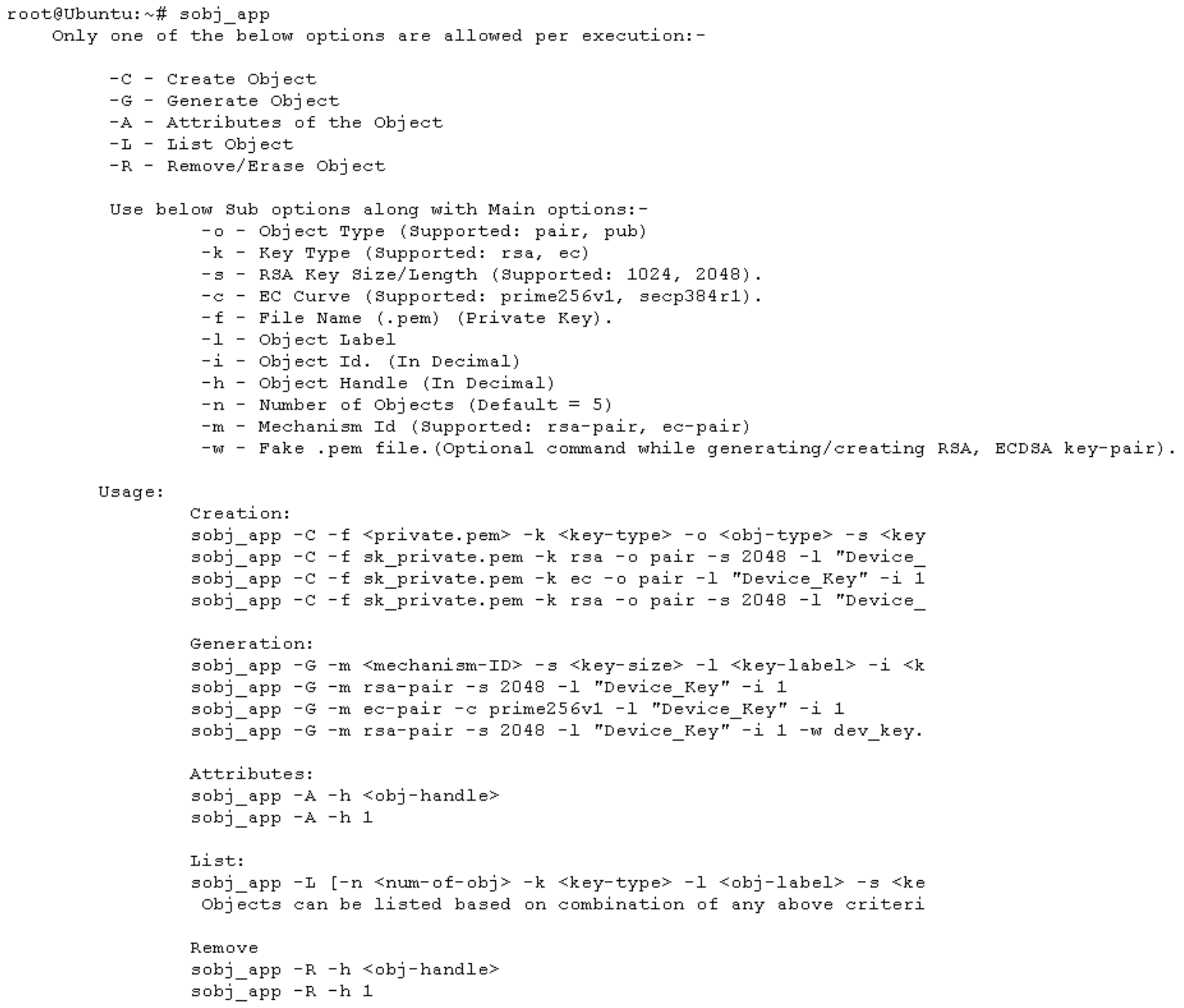
- **sobj_app** - Provides interface to generate/import key objects via Secure Object Library
- **pkcs11_app** – Provides interface to enumerate objects in the HSM and perform cryptographic operations.
- **mp_app** - This application demonstrates how to Get MP Public Key, sign a message using MP Private Key, Get Message tag.
- **mp_verify** - This app uses OpenSSL APIs to verify the signature obtained by using mp_app application.
- **sobj_eng_app** – This app uses OpenSSL APIs to show how to use Secure Object based OpenSSL Engine. This application is loading the private key and then doing cryptographic operations using this key.
- **thread_test** - PKCS#11 application to test multithreading feature of PKCS#11 library.

NOTE: These are reference applications to demonstrate the usage of APIs as described in Supported APIs

6.5.4.2.1 sobj_app

To create/generate objects, run **sobj_app** application.

- **sobj_app** – This command shows help related to sobj_app.



```
root@Ubuntu:~# sobj_app
    Only one of the below options are allowed per execution:-

         -C - Create Object
         -G - Generate Object
         -A - Attributes of the Object
         -L - List Object
         -R - Remove/Erase Object

         Use below Sub options along with Main options:-
                 -o - Object Type (Supported: pair, pub)
                 -k - Key Type (Supported: rsa, ec)
                 -s - RSA Key Size/Length (Supported: 1024, 2048).
                 -c - EC Curve (Supported: prime256v1, secp384r1).
                 -f - File Name (.pem) (Private Key).
                 -l - Object Label
                 -i - Object Id. (In Decimal)
                 -h - Object Handle (In Decimal)
                 -n - Number of Objects (Default = 5)
                 -m - Mechanism Id (Supported: rsa-pair, ec-pair)
                 -w - Fake .pem file.(Optional command while generating/creating RSA, ECDSA key-pair).

        Usage:
                Creation:
                sobj_app -C -f <private.pem> -k <key-type> -o <obj-type> -s <key
                sobj_app -C -f sk_private.pem -k rsa -o pair -s 2048 -l "Device_
                sobj_app -C -f sk_private.pem -k ec -o pair -l "Device_Key" -i 1
                sobj_app -C -f sk_private.pem -k rsa -o pair -s 2048 -l "Device_

                Generation:
                sobj_app -G -m <mechanism-ID> -s <key-size> -l <key-label> -i <k
                sobj_app -G -m rsa-pair -s 2048 -l "Device_Key" -i 1
                sobj_app -G -m ec-pair -c prime256v1 -l "Device_Key" -i 1
                sobj_app -G -m rsa-pair -s 2048 -l "Device_Key" -i 1 -w dev_key.

                Attributes:
                sobj_app -A -h <obj-handle>
                sobj_app -A -h 1

                List:
                sobj_app -L [-n <num-of-obj> -k <key-type> -l <obj-label> -s <ke
                 Objects can be listed based on combination of any above criteri

                Remove
                sobj_app -R -h <obj-handle>
                sobj_app -R -h 1
```

- **Importing an RSA key pair to HSM**

  sobj_app -C -f <private.pem> -k <key-type> -o <obj-type> -s <key-size> -l <obj-label> -i <obj-ID>

  This command helps in importing a key to the HSM. It creates an object in HSM reading key from <private.pem> with object label <obj-label> and object ID <obj-ID>. This private.pem can be generated by openssl using the command below:

  openssl genpkey -algorithm RSA -out sk_private.pem -pkeyopt rsa_keygen_bits:2048

  Handle of the object created in the HSM is printed as an output to the command. This handle can be used for further operations on the created object (for example, delete, printing attributes and so on)

```
root@Ubuntu:~# sobj_app -C -f sk_private.pem -k rsa -o pair -s 2048 -l "rsa_create_2048" -i 0
Creating the Object.
Import Key from sk_private.pem
Key Length = 2048
Object created successfully handle = 0
root@Ubuntu:~#
root@Ubuntu:~#
```

- **Importing an ECDSA key pair to HSM**

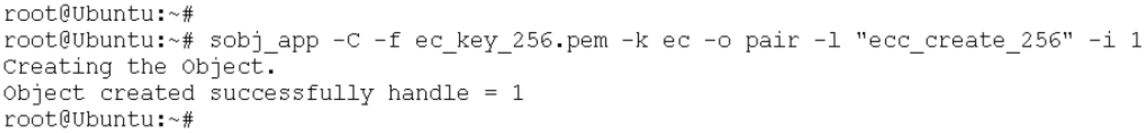
  sobj_app -C -f <private.pem> -k <key-type> -o <obj-type> -l <obj-label> -i <obj-ID>

  This command helps in importing a key to the HSM. It will create an object in HSM reading key from <private.pem> with object label <obj-label> and object ID <obj-ID>.

This private.pem can be generated by openssl using below command:

openssl ecparam -genkey -name prime256v1 -noout -out ec_key_256.pem

Handle of the object created in the HSM is printed as an output to the command. This handle can be used for further operations on the created object (eg delete, printing attributes etc)



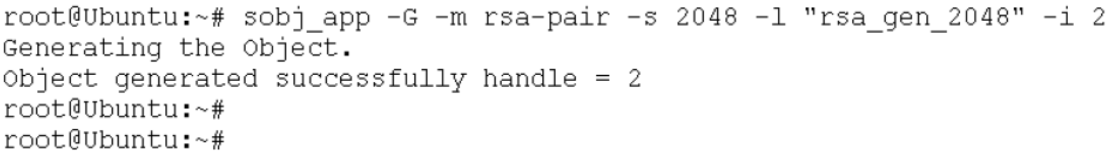
```
root@Ubuntu:~#
root@Ubuntu:~# sobj_app -C -f ec_key_256.pem -k ec -o pair -l "ecc_create_256" -i 1
Creating the Object.
Object created successfully handle = 1
root@Ubuntu:~#
```

- **Generating an RSA key pair in HSM**

  sobj_app -G -m <mechanism-ID> -s <key-size> -l <key-label> -i <key-ID>

  This command generates an object of type derived from mechanism-ID of size <key-size> with label <key-label> and ID <key-ID>

  Handle of the object created is printed as an output to the command. This handle can be used for further operations on the created object (for example, delete, printing attributes and so on)



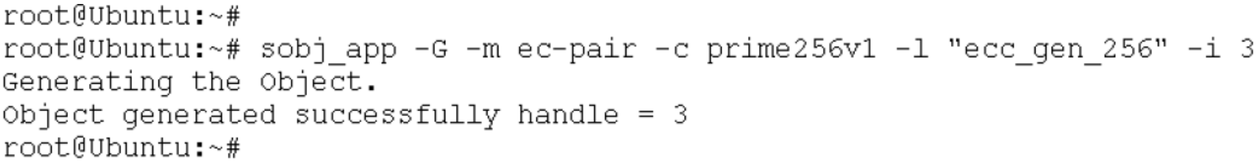
```
root@Ubuntu:~# sobj_app -G -m rsa-pair -s 2048 -l "rsa_gen_2048" -i 2
Generating the Object.
Object generated successfully handle = 2
root@Ubuntu:~#
root@Ubuntu:~#
```

- **Generating ECDSA key pair in HSM**

  sobj_app -G -m <mechanism-ID> -c <curve> -l <key-label> -i <key-ID>

  This command will generate an object of type derived from mechanism-ID of size <key-size> with label <key-label> and ID <key-ID>

  Handle of the object created is printed as an output to the command. This handle can be used for further operations on the created object (eg delete, printing attributes etc)

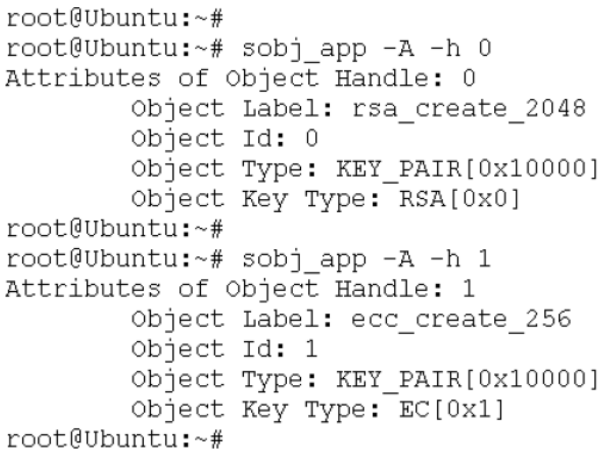


```
root@Ubuntu:~#
root@Ubuntu:~# sobj_app -G -m ec-pair -c prime256v1 -l "ecc_gen_256" -i 3
Generating the Object.
Object generated successfully handle = 3
root@Ubuntu:~#
```

- **Display attributes of an object in the HSM**

  sobj_app -A -h <obj-handle>

  This command shows some attributes related to object created. Pass the object handle <obj-handle> to the command. This <obj-handle> is printed during generation/import of objects to HSM.

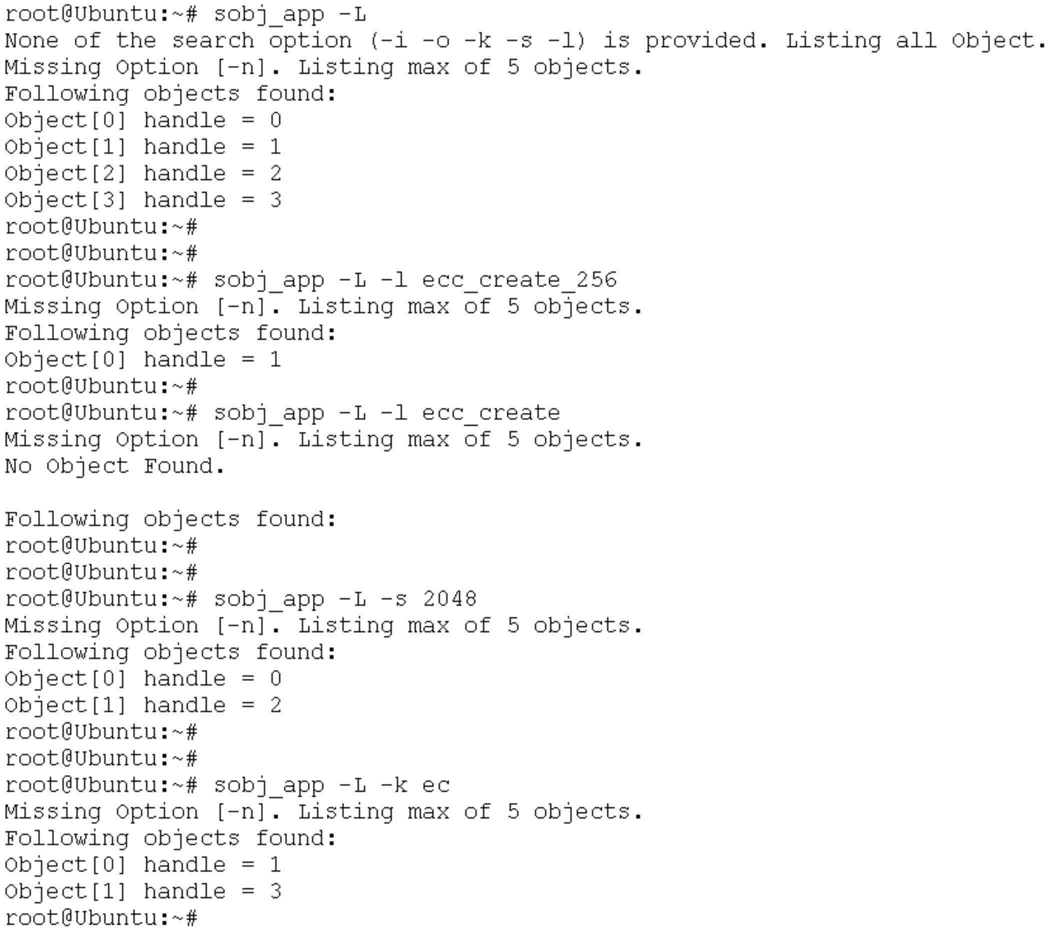


```
root@Ubuntu:~#
root@Ubuntu:~# sobj_app -A -h 0
Attributes of Object Handle: 0
        Object Label: rsa_create_2048
        Object Id: 0
        Object Type: KEY_PAIR[0x10000]
        Object Key Type: RSA[0x0]
root@Ubuntu:~#
root@Ubuntu:~# sobj_app -A -h 1
Attributes of Object Handle: 1
        Object Label: ecc_create_256
        Object Id: 1
        Object Type: KEY_PAIR[0x10000]
        Object Key Type: EC[0x1]
root@Ubuntu:~#
```

- **List handles of the objects available in the HSM**

  sobj_app -L [-n <num-of-obj> -k <key-type> -l <obj-label> -s <key-size> -i <obj-id>]

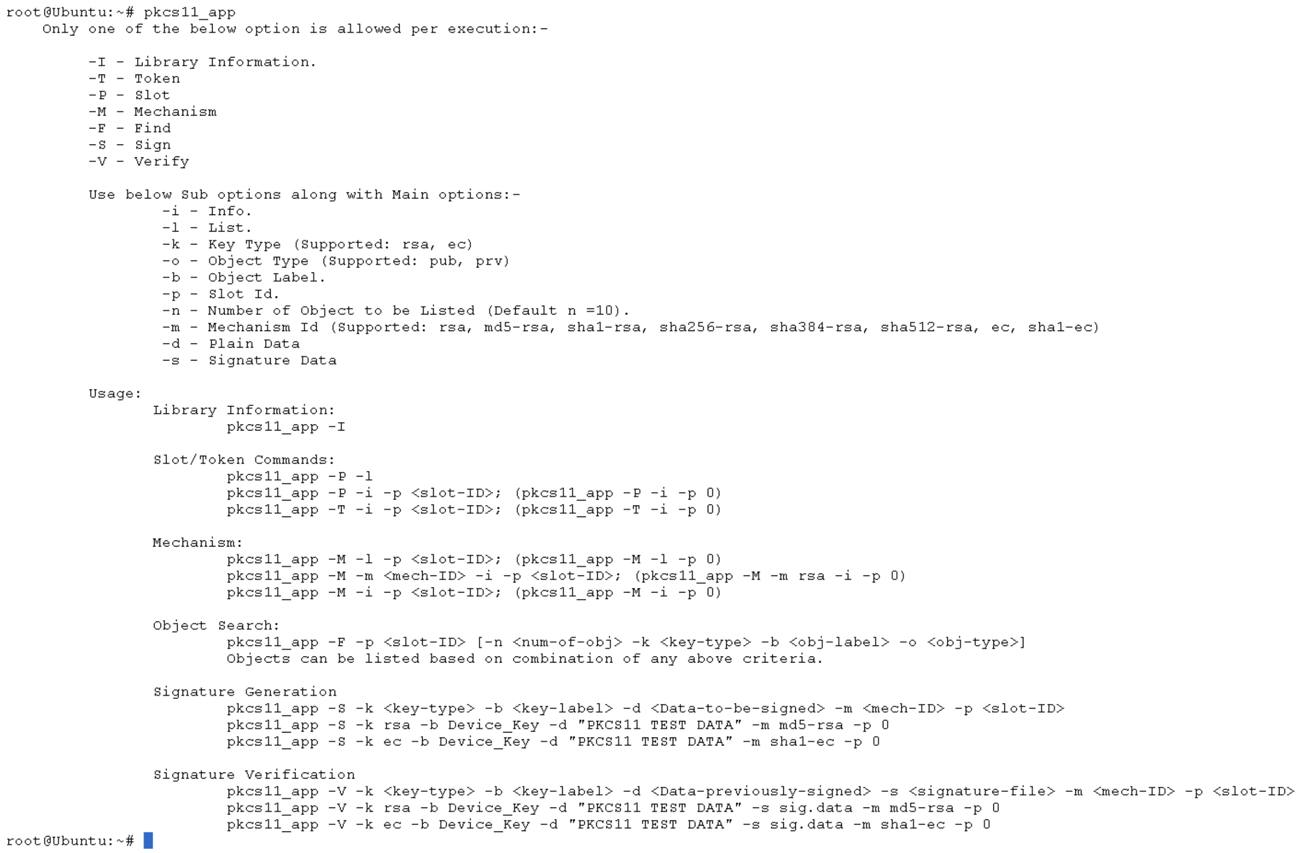
  This command lists handles of the objects already created/generated based on some search criteria (if given). User can then use this handle to print the rest of the attributes. (See above command)



```
root@Ubuntu:~# sobj_app -L
None of the search option (-i -o -k -s -l) is provided. Listing all Object.
Missing Option [-n]. Listing max of 5 objects.
Following objects found:
Object[0] handle = 0
Object[1] handle = 1
Object[2] handle = 2
Object[3] handle = 3
root@Ubuntu:~#
root@Ubuntu:~#
root@Ubuntu:~# sobj_app -L -l ecc_create_256
Missing Option [-n]. Listing max of 5 objects.
Following objects found:
Object[0] handle = 1
root@Ubuntu:~#
root@Ubuntu:~# sobj_app -L -l ecc_create
Missing Option [-n]. Listing max of 5 objects.
No Object Found.

Following objects found:
root@Ubuntu:~#
root@Ubuntu:~#
root@Ubuntu:~# sobj_app -L -s 2048
Missing Option [-n]. Listing max of 5 objects.
Following objects found:
Object[0] handle = 0
Object[1] handle = 2
root@Ubuntu:~#
root@Ubuntu:~#
root@Ubuntu:~# sobj_app -L -k ec
Missing Option [-n]. Listing max of 5 objects.
Following objects found:
Object[0] handle = 1
Object[1] handle = 3
root@Ubuntu:~#
```

6.5.4.2.2 pkcs11_app

- **pkcs11_app –** This command shows commands available.



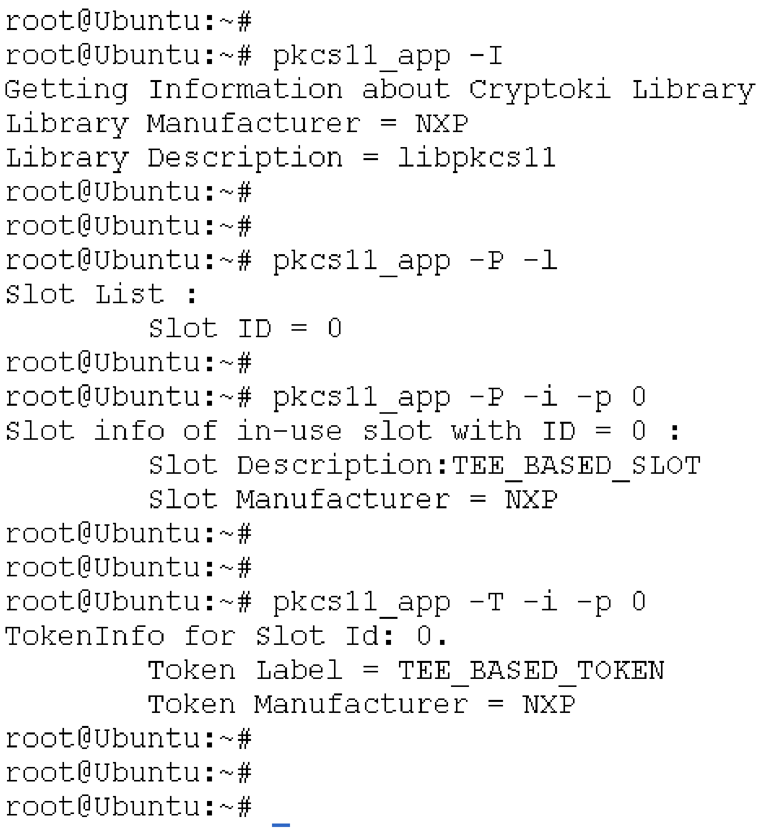
```
root@Ubuntu:~# pkcs11_app
    Only one of the below option is allowed per execution:-

         -I - Library Information.
         -T - Token
         -P - Slot
         -M - Mechanism
         -F - Find
         -S - Sign
         -V - Verify

         Use below Sub options along with Main options:-
                 -i - Info.
                 -l - List.
                 -k - Key Type (Supported: rsa, ec)
                 -o - Object Type (Supported: pub, prv)
                 -b - Object Label.
                 -p - Slot Id.
                 -n - Number of Object to be Listed (Default n =10).
                 -m - Mechanism Id (Supported: rsa, md5-rsa, sha1-rsa, sha256-rsa, sha384-rsa, sha512-rsa, ec, sha1-ec)
                 -d - Plain Data
                 -s - Signature Data

         Usage:
                Library Information:
                        pkcs11_app -I

                Slot/Token Commands:
                        pkcs11_app -P -l
                        pkcs11_app -P -i -p <slot-ID>; (pkcs11_app -P -i -p 0)
                        pkcs11_app -T -i -p <slot-ID>; (pkcs11_app -T -i -p 0)

                Mechanism:
                        pkcs11_app -M -l -p <slot-ID>; (pkcs11_app -M -l -p 0)
                        pkcs11_app -M -m <mech-ID> -i -p <slot-ID>; (pkcs11_app -M -m rsa -i -p 0)
                        pkcs11_app -M -i -p <slot-ID>; (pkcs11_app -M -i -p 0)

                Object Search:
                        pkcs11_app -F -p <slot-ID> [-n <num-of-obj> -k <key-type> -b <obj-label> -o <obj-type>]
                        Objects can be listed based on combination of any above criteria.

                Signature Generation
                        pkcs11_app -S -k <key-type> -b <key-label> -d <Data-to-be-signed> -m <mech-ID> -p <slot-ID>
                        pkcs11_app -S -k rsa -b Device_Key -d "PKCS11 TEST DATA" -m md5-rsa -p 0
                        pkcs11_app -S -k ec -b Device_Key -d "PKCS11 TEST DATA" -m sha1-ec -p 0

                Signature Verification
                        pkcs11_app -V -k <key-type> -b <key-label> -d <Data-previously-signed> -s <signature-file> -m <mech-ID> -p <slot-ID>
                        pkcs11_app -V -k rsa -b Device_Key -d "PKCS11 TEST DATA" -s sig.data -m md5-rsa -p 0
                        pkcs11_app -V -k ec -b Device_Key -d "PKCS11 TEST DATA" -s sig.data -m sha1-ec -p 0
root@Ubuntu:~#
```

- **pkcs11_app -I:** Library Information

  pkcs11_app -P -l: List the all available slots

  pkcs11_app -P -i -p <slot-ID> : Provides the information about Slot with <slot-ID>

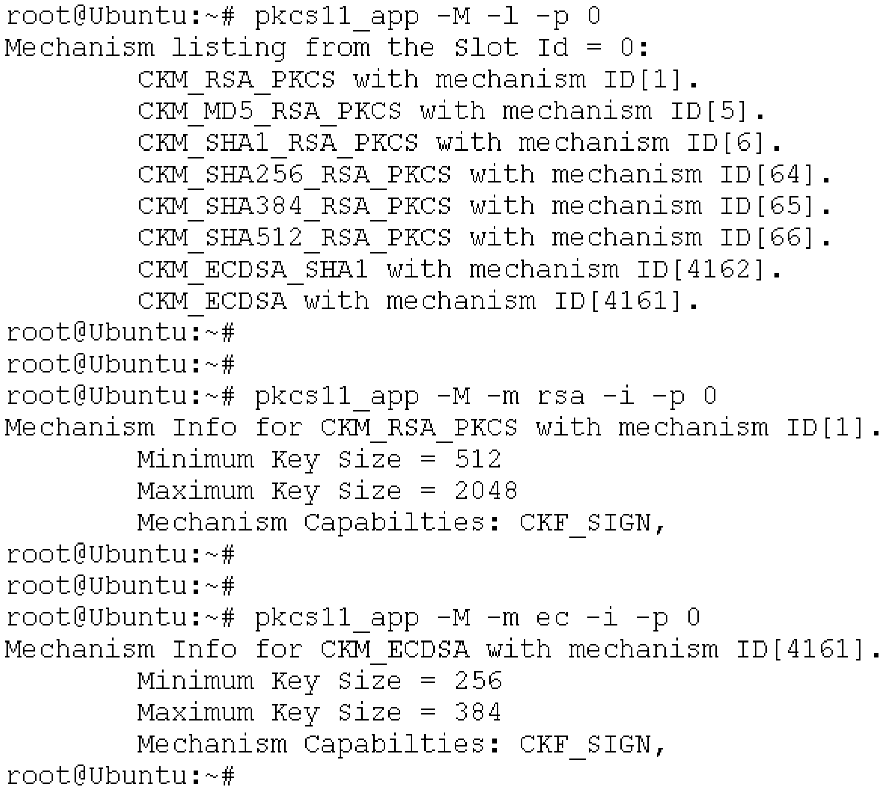
  pkcs11_app -T -i -p <slot-ID> : Provides the information about Token inserted in Slot <slot-ID>



```
root@Ubuntu:~#
root@Ubuntu:~# pkcs11_app -I
Getting Information about Cryptoki Library
Library Manufacturer = NXP
Library Description = libpkcs11
root@Ubuntu:~#
root@Ubuntu:~#
root@Ubuntu:~# pkcs11_app -P -l
Slot List :
        Slot ID = 0
root@Ubuntu:~#
root@Ubuntu:~# pkcs11_app -P -i -p 0
Slot info of in-use slot with ID = 0 :
        Slot Description:TEE_BASED_SLOT
        Slot Manufacturer = NXP
root@Ubuntu:~#
root@Ubuntu:~#
root@Ubuntu:~# pkcs11_app -T -i -p 0
TokenInfo for Slot Id: 0.
        Token Label = TEE_BASED_TOKEN
        Token Manufacturer = NXP
root@Ubuntu:~#
root@Ubuntu:~#
root@Ubuntu:~# _
```

- **pkcs11_app -M -l -p <slot-ID> :** Lists the Mechanism List supported by token in Slot <slot-ID>

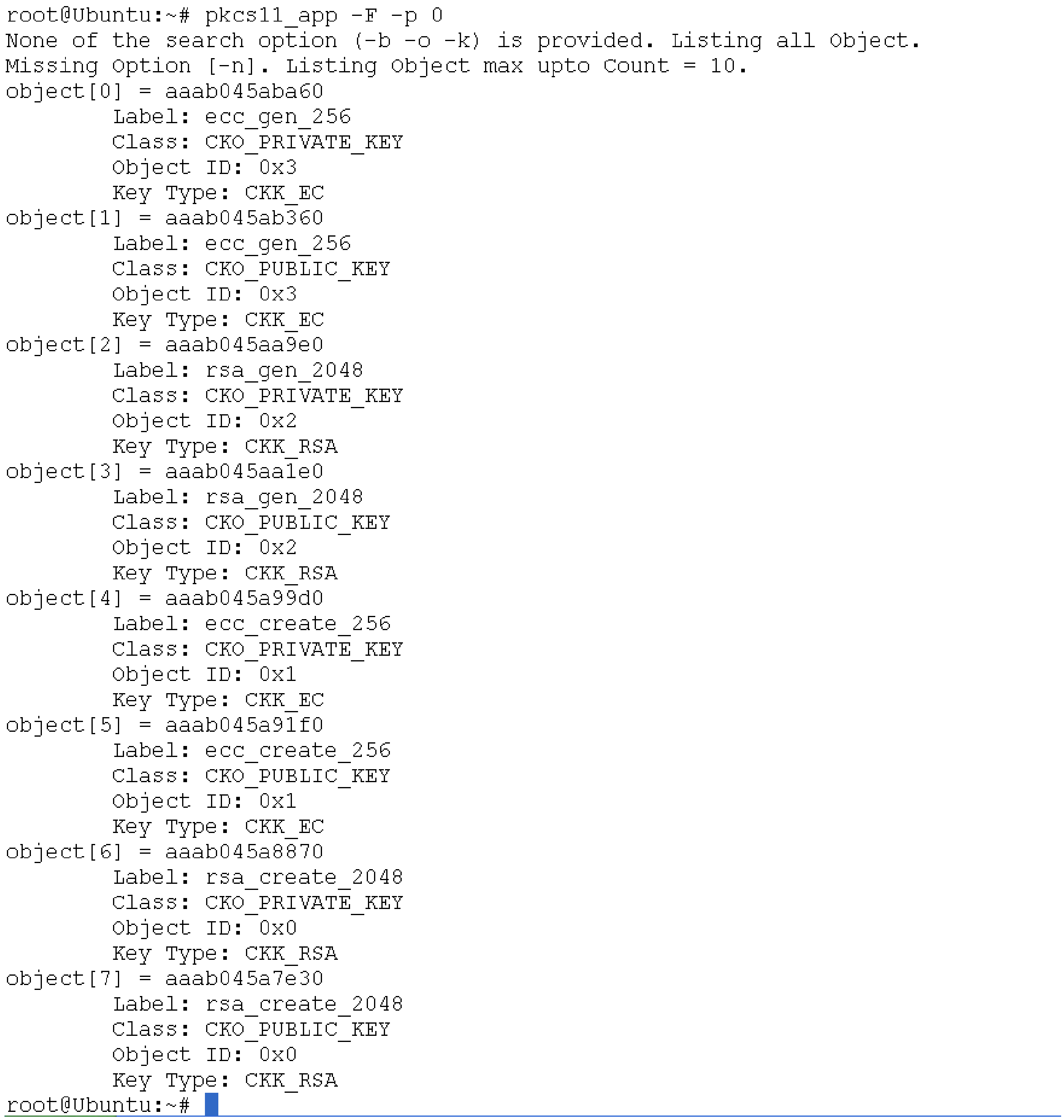
  pkcs11_app -M -m <mech-ID> -i -p <slot-ID> : Gives information about the mechanism with <mech-ID> for Slot <slot-ID>



```
root@Ubuntu:~# pkcs11_app -M -l -p 0
Mechanism listing from the Slot Id = 0:
        CKM_RSA_PKCS with mechanism ID[1].
        CKM_MD5_RSA_PKCS with mechanism ID[5].
        CKM_SHA1_RSA_PKCS with mechanism ID[6].
        CKM_SHA256_RSA_PKCS with mechanism ID[64].
        CKM_SHA384_RSA_PKCS with mechanism ID[65].
        CKM_SHA512_RSA_PKCS with mechanism ID[66].
        CKM_ECDSA_SHA1 with mechanism ID[4162].
        CKM_ECDSA with mechanism ID[4161].
root@Ubuntu:~#
root@Ubuntu:~#
root@Ubuntu:~# pkcs11_app -M -m rsa -i -p 0
Mechanism Info for CKM_RSA_PKCS with mechanism ID[1].
        Minimum Key Size = 512
        Maximum Key Size = 2048
        Mechanism Capabilties: CKF_SIGN,
root@Ubuntu:~#
root@Ubuntu:~#
root@Ubuntu:~# pkcs11_app -M -m ec -i -p 0
Mechanism Info for CKM_ECDSA with mechanism ID[4161].
        Minimum Key Size = 256
        Maximum Key Size = 384
        Mechanism Capabilties: CKF_SIGN,
root@Ubuntu:~#
```

- **pkcs11_app -F -p <slot-ID>:** List all objects associated with token present in slot <slot-ID>

  We have 2 objects already created via the **sobj_app**, which will be shown here through **pkcs11_app** find operation.



```
root@Ubuntu:~# pkcs11_app -F -p 0
None of the search option (-b -o -k) is provided. Listing all Object.
Missing Option [-n]. Listing Object max upto Count = 10.
object[0] = aaab045aba60
        Label: ecc_gen_256
        Class: CKO_PRIVATE_KEY
        Object ID: 0x3
        Key Type: CKK_EC
object[1] = aaab045ab360
        Label: ecc_gen_256
        Class: CKO_PUBLIC_KEY
        Object ID: 0x3
        Key Type: CKK_EC
object[2] = aaab045aa9e0
        Label: rsa_gen_2048
        Class: CKO_PRIVATE_KEY
        Object ID: 0x2
        Key Type: CKK_RSA
object[3] = aaab045aa1e0
        Label: rsa_gen_2048
        Class: CKO_PUBLIC_KEY
        Object ID: 0x2
        Key Type: CKK_RSA
object[4] = aaab045a99d0
        Label: ecc_create_256
        Class: CKO_PRIVATE_KEY
        Object ID: 0x1
        Key Type: CKK_EC
object[5] = aaab045a91f0
        Label: ecc_create_256
        Class: CKO_PUBLIC_KEY
        Object ID: 0x1
        Key Type: CKK_EC
object[6] = aaab045a8870
        Label: rsa_create_2048
        Class: CKO_PRIVATE_KEY
        Object ID: 0x0
        Key Type: CKK_RSA
object[7] = aaab045a7e30
        Label: rsa_create_2048
        Class: CKO_PUBLIC_KEY
        Object ID: 0x0
        Key Type: CKK_RSA
root@Ubuntu:~#
```

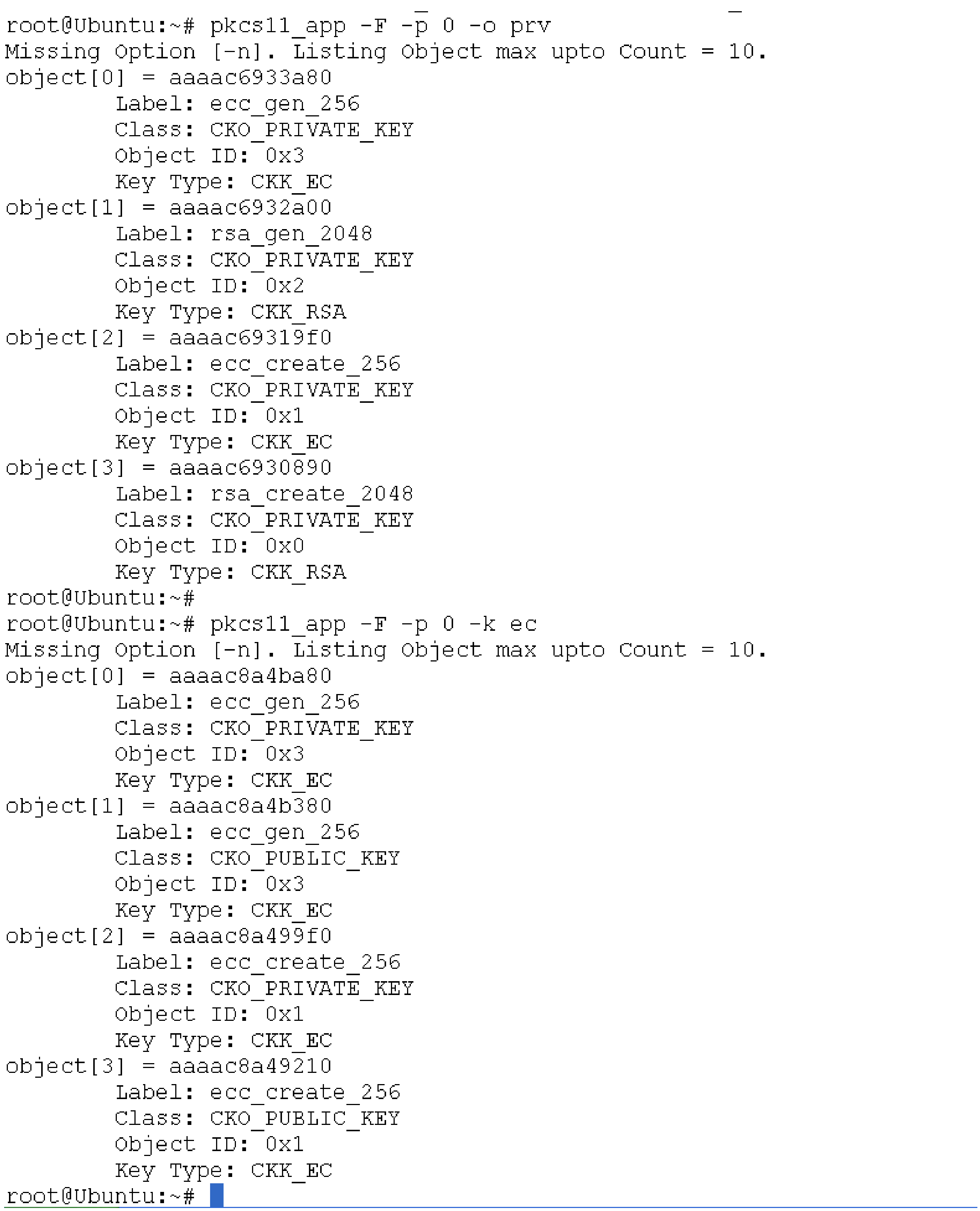
- **Currently search can be made based on 3 criteria via this app:**

  -o: Object type (Can be public key, private key, certificates and so on)(For now supports only Public and Private keys)

  -k: Key type (Can be RSA, EC, AES and so on)(For now supports only RSA)

  -b: Object Label associated with object while creating/generating.

  pkcs11_app -F -o <obj-type> -k <key-type> -b <label> -p <slot-ID> : List all objects which are having object type <obj-type> of key type <key-type> and with label < label> on token present in slot <slot-ID>



```
root@Ubuntu:~# pkcs11_app -F -p 0 -o prv
Missing Option [-n]. Listing Object max upto Count = 10.
object[0] = aaaac6933a80
        Label: ecc_gen_256
        Class: CKO_PRIVATE_KEY
        Object ID: 0x3
        Key Type: CKK_EC
object[1] = aaaac6932a00
        Label: rsa_gen_2048
        Class: CKO_PRIVATE_KEY
        Object ID: 0x2
        Key Type: CKK_RSA
object[2] = aaaac69319f0
        Label: ecc_create_256
        Class: CKO_PRIVATE_KEY
        Object ID: 0x1
        Key Type: CKK_EC
object[3] = aaaac6930890
        Label: rsa_create_2048
        Class: CKO_PRIVATE_KEY
        Object ID: 0x0
        Key Type: CKK_RSA
root@Ubuntu:~#
root@Ubuntu:~# pkcs11_app -F -p 0 -k ec
Missing Option [-n]. Listing Object max upto Count = 10.
object[0] = aaaac8a4ba80
        Label: ecc_gen_256
        Class: CKO_PRIVATE_KEY
        Object ID: 0x3
        Key Type: CKK_EC
object[1] = aaaac8a4b380
        Label: ecc_gen_256
        Class: CKO_PUBLIC_KEY
        Object ID: 0x3
        Key Type: CKK_EC
object[2] = aaaac8a499f0
        Label: ecc_create_256
        Class: CKO_PRIVATE_KEY
        Object ID: 0x1
        Key Type: CKK_EC
object[3] = aaaac8a49210
        Label: ecc_create_256
        Class: CKO_PUBLIC_KEY
        Object ID: 0x1
        Key Type: CKK_EC
root@Ubuntu:~#
```

- **pkcs11_app -S -k <key-type> -b <key-label> -d <Data-to-be-signed> -m <mech-ID> -p <slot-ID>**

  This command will sign the <Data> with private key of type <key-type> having label <key-label> using mechanism specified by <mech-ID> with functions provided by token in slot <slot-ID>

After successful signing, the signature will be saved in file “sig.data”

RSA signing:

```
root@Ubuntu:~#
root@Ubuntu:~# pkcs11_app -S -k rsa -b Device_Key -d "PKCS11 TEST DATA" -m md5-rsa -p 0
Signing...
Size of Unsigned data = 16
Signature size: 256
Signature is saved in the file sig.data:
root@Ubuntu:~#
```

ECDSA signing

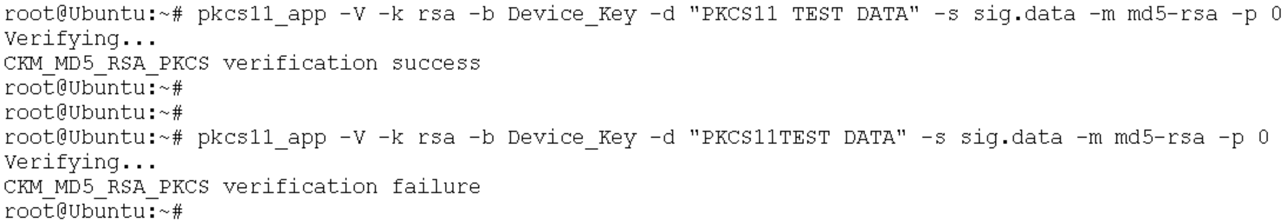
```
root@Ubuntu:~#
root@Ubuntu:~# pkcs11_app -S -k ec -b ecc_gen_256 -d "PKCS11 TEST DATA" -m sha1-ec -p 0
Signing...
Size of Unsigned data = 16
Signature size: 512
Signature is saved in the file sig.data:
```

- **pkcs11_app -V -k <key-type> -b <key-label> -d <Data-previously-signed> -s <signature-file> -m <mech-ID> -p <slot-ID>**

  This command verifies the signature <signature-file> with public key of type <key-type> having label <key-label> using mechanism specified by <mech-ID> with functions provided by token in slot <slot-ID> by comparing the data recovered from signature to <Data-previously-signed>. This command uses openssl APIs to do the verification. Refer to the application code for more details.

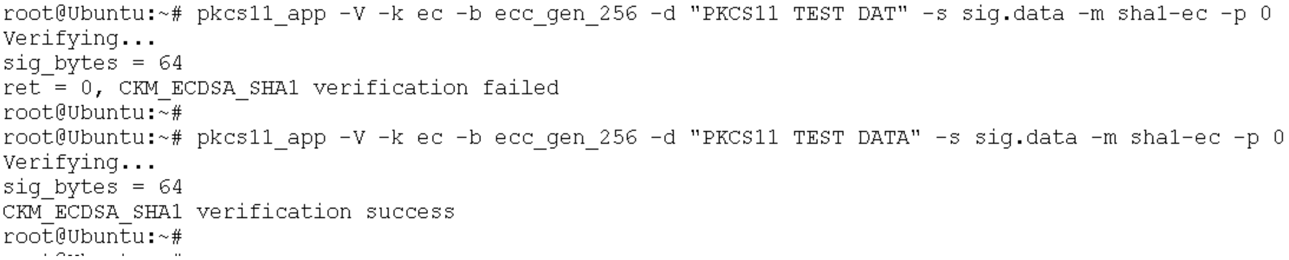
  <mech-ID> passed must match with the <mech-ID> passed during signature otherwise verification fails, as shown in following picture.

RSA Verification:



```
root@Ubuntu:~# pkcs11_app -V -k rsa -b Device_Key -d "PKCS11 TEST DATA" -s sig.data -m md5-rsa -p 0
Verifying...
CKM_MD5_RSA_PKCS verification success
root@Ubuntu:~#
root@Ubuntu:~#
root@Ubuntu:~# pkcs11_app -V -k rsa -b Device_Key -d "PKCS11TEST DATA" -s sig.data -m md5-rsa -p 0
Verifying...
CKM_MD5_RSA_PKCS verification failure
root@Ubuntu:~#
```

ECDSA Verification:



```
root@Ubuntu:~# pkcs11_app -V -k ec -b ecc_gen_256 -d "PKCS11 TEST DAT" -s sig.data -m sha1-ec -p 0
Verifying...
sig_bytes = 64
ret = 0, CKM_ECDSA_SHA1 verification failed
root@Ubuntu:~#
root@Ubuntu:~# pkcs11_app -V -k ec -b ecc_gen_256 -d "PKCS11 TEST DATA" -s sig.data -m sha1-ec -p 0
Verifying...
sig_bytes = 64
CKM_ECDSA_SHA1 verification success
root@Ubuntu:~#
```

6.5.4.2.3 mp_app

This application demonstrates how to use the following APIs:

- Get MP Public Key.
- Sign a message using MP Private Key.
- Get Message tag.

The application source code at location “**secure_obj/securekey_lib/app/mp_app.c**” can be used as reference for integration of these APIs.

mp_app - This application gives 3 options.

Usage:

- **mp_app -p:** Get the MP Public key and store it in a file "pub_key"
- **mp_app -s <MSG>:** Sign <MSG> with MP Priv key and store signature in file "signature"
- **mp_app -m:** Get the MP Message tag and store it in file "mtag"

```
root@Ubuntu:~#
root@Ubuntu:~# mp_app -p
Generating the MP Public Key
Public key x part = 94d9548963b8fd4f4ebd253320cc8a7ad3342b16288c18eccddc2fd1482a9b0b
Public key y part = e2d4bd22c40a7c2a41a8ca68204c0a346dde9721bb710bc3a5df31a50099c572
Public key in form of x followed by y is saved in pub_key file
root@Ubuntu:~#
root@Ubuntu:~# mp_app -s "HELLO"

Signing message  'HELLO' with MP Priv Key
HELLO in Hex = 48454c4c4f
Generated Hash = 504eb5a99c0ab4a6b27678cb6dd36bd2c365d9d5a666cc16ac5a2fbd5b5049a9
Signature part r = 37a3bac5c27a981f6826c6c3e6841ff687a838e24fc5fb9dd8809b567a5fc194
Signature part s = 2babeea4da0192d2e6363eebc179e49dbcf933865e13a0215eb06acb380a6cdd
Signature in form of r followed by s is saved in signature file
root@Ubuntu:~#
root@Ubuntu:~#
root@Ubuntu:~# mp_app -m

MP Tag = 9bac8c5220dcbec0114d651879e5d5891160fcf14e3d5a4d79403c5a1252ded9
MP Tag is saved in mptag file
root@Ubuntu:~#
root@Ubuntu:~#
```

6.5.4.2.4 mp_verify

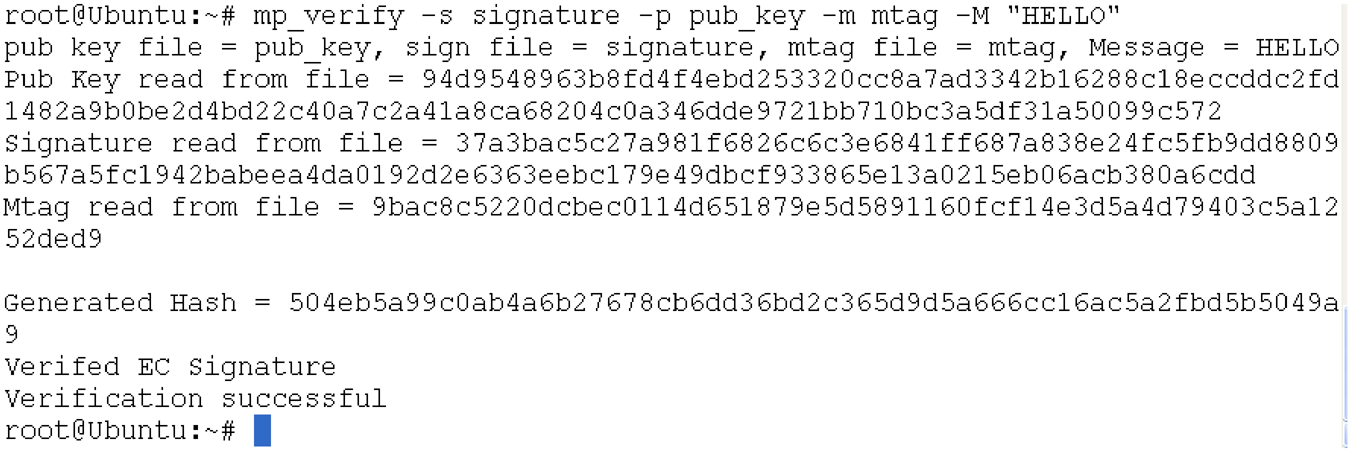
This app uses OpenSSL APIs to verify the signature obtained by using **"mp_app"** application. The application source code at location "**secure_obj/securekey_lib/app/mp_verify.c**" can be used as reference.

mp_verify - This application verifies the signature generated by **mp_app -s**.

Usage:

mp_verify -p <pubkeyfile> -s <signaturefile> -m <mtagfile> -M <MSG>

This <MSG> must be same which is used in **mp_app -s <MSG>**



```
root@Ubuntu:~# mp_verify -s signature -p pub_key -m mtag -M "HELLO"
pub key file = pub_key, sign file = signature, mtag file = mtag, Message = HELLO
Pub Key read from file = 94d9548963b8fd4f4ebd253320cc8a7ad3342b16288c18eccddc2fd
1482a9b0be2d4bd22c40a7c2a41a8ca68204c0a346dde9721bb710bc3a5df31a50099c572
Signature read from file = 37a3bac5c27a981f6826c6c3e6841ff687a838e24fc5fb9dd8809
b567a5fc1942babeea4da0192d2e6363eebc179e49dbcf933865e13a0215eb06acb380a6cdd
Mtag read from file = 9bac8c5220dcbec0114d651879e5d5891160fcf14e3d5a4d79403c5a12
52ded9

Generated Hash = 504eb5a99c0ab4a6b27678cb6dd36bd2c365d9d5a666cc16ac5a2fbd5b5049a
9
Verifed EC Signature
Verification successful
root@Ubuntu:~#
```

6.5.4.2.5 sobj_eng_app

This app uses OpenSSL APIs to show how to use Secure Object based OpenSSL Engine.

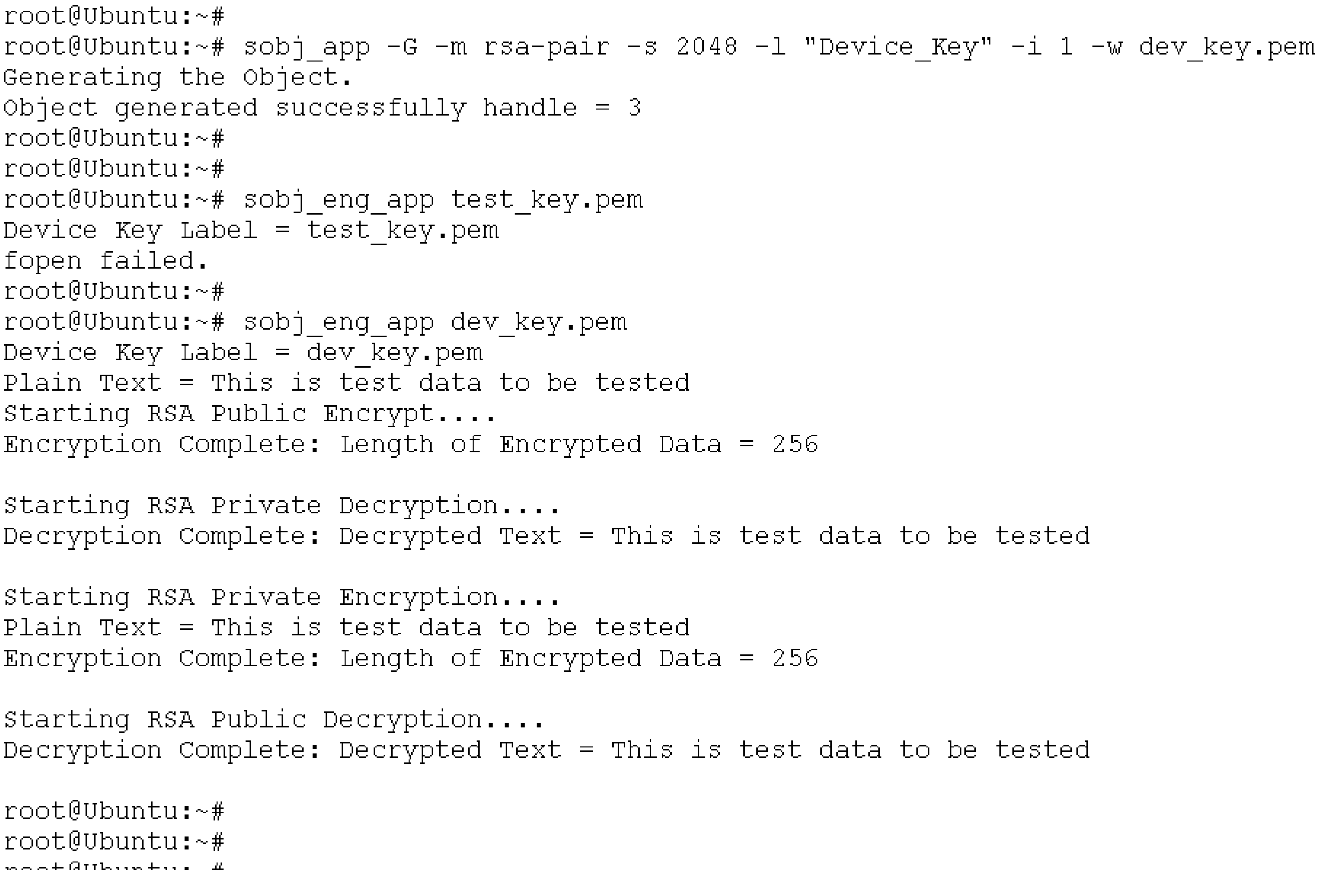
Code for this app is at **“secure_obj/secure_obj-openssl-engine/app/sobj_eng_app.c “.**

This application is internally loading RSA private key and then doing cryptographic operations using this key.

Private Key operations are offloaded to Secure Object via this engine, and Public Key operations are done through OpenSSL itself.

In Following screenshot, see creating a key via **sobj_app**. It will be used by **sobj_eng_app** (using OpenSSL APIs) to do the cryptographic operations.

This **sobj_eng_app** is internally offloading the cryptographic operation to Secure Object Library using the OpenSSL Engine based on Secure Object Library.



```
root@Ubuntu:~#
root@Ubuntu:~# sobj_app -G -m rsa-pair -s 2048 -l "Device_Key" -i 1 -w dev_key.pem
Generating the Object.
Object generated successfully handle = 3
root@Ubuntu:~#
root@Ubuntu:~#
root@Ubuntu:~# sobj_eng_app test_key.pem
Device Key Label = test_key.pem
fopen failed.
root@Ubuntu:~#
root@Ubuntu:~# sobj_eng_app dev_key.pem
Device Key Label = dev_key.pem
Plain Text = This is test data to be tested
Starting RSA Public Encrypt....
Encryption Complete: Length of Encrypted Data = 256

Starting RSA Private Decryption....
Decryption Complete: Decrypted Text = This is test data to be tested

Starting RSA Private Encryption....
Plain Text = This is test data to be tested
Encryption Complete: Length of Encrypted Data = 256

Starting RSA Public Decryption....
Decryption Complete: Decrypted Text = This is test data to be tested

root@Ubuntu:~#
root@Ubuntu:~#
root@Ubuntu:~#
```

6.5.4.2.6 thread_test

PKCS#11 based application to test the multithreading support in PKCS#11 Library.

This application will be taking the number of threads to create as an argument, if not given by default it will create 10 threads.

thread_test <num-of-threads>

This application is making threads and each thread is doing the signing operation.

As part of signing operation each thread is doing following operations:

- Opening a R/O session with token.
- Find a RSA Private Key from token.
- Sign data using this RSA Private Key.

- Get the Public part of the RSA Private Key.
- Verify the generated signature using OpenSSL.

All threads try to do this in parallel, but if one of the thread finished its work and “Finalized” the library, all other threads will terminate, because library is not in initialized state now.

NOTE: This sequence of operations are used only for test purpose.

6.5.5 Validation

Above steps are fully validated and verified on LS1046ARDB platform.

6.5.6 Appendix

Appendix A: Steps to build the PKCS#11 Library

PKCS Library is using Secure Object Library. For steps compiling Secure Object Library, see section **Appendix B: Steps to build the Secure Object Library**.

From flexbuild environment:

flex-builder -c libpkcs11 -m ls1046ardb

Standalone Build:

1. Clone the libpkcs11 from: https://source.codeaurora.org/external/qoriq/qoriq-components/libpkcs11
2. Checkout tag **“LSDK-18.09”**.
3. Set path for cross-compile:
   ```
   $:> export CROSS_COMPILE=<aarch64-toolchain>
   ```
4. Set path for Secure Object:
   ```
   $:> export SECURE_OBJ_PATH=<path-to-secure_obj>/secure_obj/securekey_lib/out/export/
   ```
5. Set path for OpenSSL:

   Note: For interoperability, we are verifying the signature generated by PKCS Library via OpenSSL, so reference application needs OpenSSL library, so exporting OPENSSL_PATH.

   We have cloned and compiled the OpenSSL in “Steps to build the Secure Object Library”, hence only give path of that folder in OPENSSL_PATH.
   ```
   $:> export OPENSSL_PATH=<openssl-folder>
   ```
6. Run make:
   ```
   $:> make
   ```
   This compiles the libpkcs11 and reference applications and put it into “images” folder in libpkcs11. Following images are generated:

   - **libpkcs11.so** – PKCS#11 User space library.
   - **pkcs11_app** – PKCS#11 Test App.
   - **thread_test** - PKCS#11 application to test multithreading feature of PKCS#11 library

Appendix B: Steps to build the Secure Object Library

From flexbuild environment:

flex-builder -c secure_obj -m ls1046ardb

Standalone Build:

Order of repo compilation for Secure Object Library.

1. OP-TEE OS
    a. Clone optee_os from: https://source.codeaurora.org/external/qoriq/qoriq-components/optee_os
    b. Checkout tag “**LSDK-18.09**”
    c. Set the path for the following:

    ```
    $:> export CROSS_COMPILE64=<aarch64-toolchain>
    ```

    d. Now make.

    ```
    $:> make CFG_ARM64_core=y PLATFORM=ls-ls1046ardb ARCH=arm
    ```

2. OP-TEE Client
    a. Clone optee_client from: https://source.codeaurora.org/external/qoriq/qoriq-components/optee_client
    b. Checkout tag “**LSDK-18.09**”
    c. Set path for the following:

    ```
    $:> export CROSS_COMPILE=<aarch64-toolchain-path->
    ```

    d. Now make.

    ```
    $:> make
    ```

3. OpenSSL:
    a. Clone openssl from: https://source.codeaurora.org/external/qoriq/qoriq-components/openssl
    b. Checkout tag “**LSDK-18.09**”
    c. Set path for the following:

    ```
     $:> export CROSS_COMPILE=<aarch64-toolchain-path->
    ```

    d. Run configure as follows:

    ```
    $:>. /Configure shared linux-aarch64
    ```

    e. Run make

    ```
     $:> make
    ```

4. Secure Object:
    a. Clone secure_obj from: https://source.codeaurora.org/external/qoriq/qoriq-components/secure_obj
    b. Checkout tag “**LSDK-18.09**”.
        - Secure Object Library code - securekey_lib
        - Secure Object Trusted Application code - secure_storage_ta
        - Secure Key Dev Kernel Module - securekeydev
        - Secure Object OpenSSL Engine - secure_obj-openssl-engine

        There is script “compile.sh” which compiles all above components and put all binaries in “images”.
    c. Follow the below compilation steps:
        - export CROSS_COMPILE path:

        ```
        $:> export CROSS_COMPILE= <aarch64-toolchain-path->
        ```

        - export ARCH path:

        ```
        $:> export ARCH=arm64
        ```

        - Set the paths from OP-TEE OS:

        ```
        $:> export TA_DEV_KIT_DIR=<path-to-optee-os>/optee_os/out/arm-plat-ls/export-
        ta_arm64/
        ```

        - Set path for OP-TEE Client

```
$:> export OPTEE_CLIENT_EXPORT=<path-to-optee-client>/optee_client/out/export/
```

- Set path for Secure Storage:

```
$:> export SECURE_STORAGE_PATH=<path-to-secure_obj>/secure_obj/
secure_storage_ta/ta/
```

- Set path for OpenSSL:

```
$:> export OPENSSL_PATH=<openssl-folder-path>
```

- Set path for Linux Code (Used Flexbuild kernel for this):

```
$:> export KERNEL_SRC=<path-in-flexbuild-containing-kernel-source-code>
For example,
$:> export KERNEL_SRC=/home/b42224/flexbuild_1809/flexbuild/build/linux/linux/
arm64/lib/modules/4.9.62/source
```

- Set path for Linux Build Directory (Used Flexbuild kernel for this):

```
$:> export KERNEL_BUILD=<path-in-flexbuild-containing-kernel-build>
For example,
$:> export KERNEL_BUILD=/home/b42224/flexbuild_1809/flexbuild/build/linux/linux/
arm64/lib/modules/4.9.62/build
```

- Run "./compile.sh" (It will compile TA, library and Kernel Module).

```
$:> ./compile.sh
```

This will compile all the binaries and put them into the images folder in `secure_obj`. After compilation, images folder have the following:

- **b05bcf48-9732-4efa-a9e0-141c7c888c34.ta** - Trusted application for Secure Object library.
- **libsecure_obj.so** - User space Secure Object Library
- **sobj_app** - Application for creating and erasing objects.
- **mp_app** - Application for getting MP Public Key, signing using MP Private key and getting the MP tag.
- **mp_verify** - Application for verifying the signature generated through mp_app.
- **securekeydev.ko** - Kernel Module for offloading MP Key feature to CAAM. Binaries to be placed at following locations in rootfs.
- **libeng_secure_obj** - Secure Object based OpenSSL engine offloading Private Key Operations to the Secure Object Library.
- **sobj_eng_app** - This app uses OpenSSL APIs to show how to use Secure Object based OpenSSL Engine. This application is loading the private key and then doing cryptographic operations using this key.

Chapter 7
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:

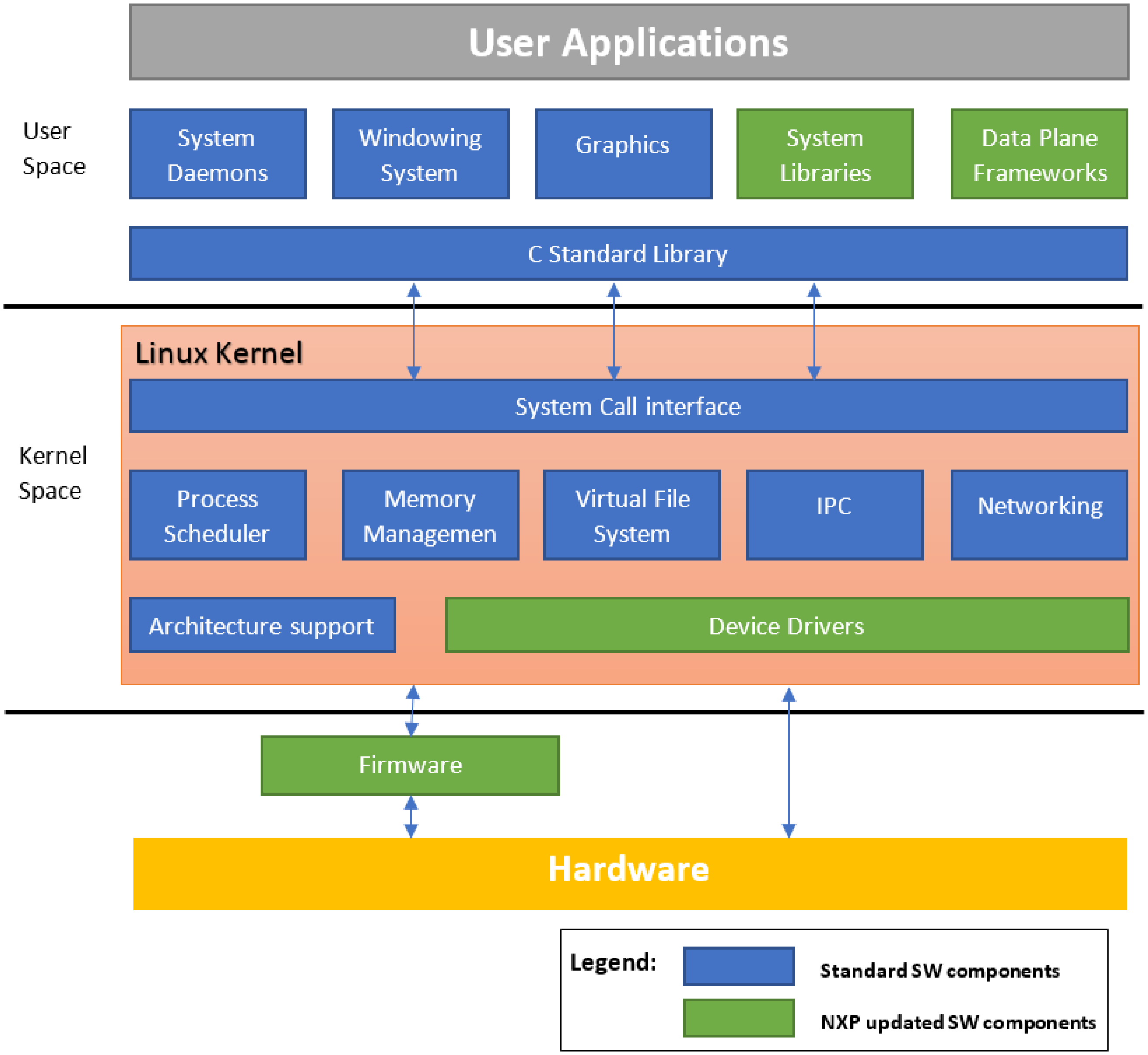


Figure 59. Typical Linux System

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
  ```

7.1 Configuring and building

Configuring and building the Linux kernel is controlled by the Kbuild sub-system. You can find documents describing the internal of Kbuild sub-system under the Documentation/kbuild/ folder in the Linux source code tree if you are adding new files or new configure options to the kernel. Otherwise as a user of Linux kernel, you probably only want to know how to fine tune the kernel configuration base on your system requirements and build new kernel image with updated configuration. These are done through *make* commands, below we will talk about `make` commands you probably need to know as a kernel user.

Environment setting for cross-compiling

This following settings are applicable when you are configuring and building kernel on a different architecture from the target. For example, compiling an ARMv8 kernel on an X86 computer. If you are compiling the kernel natively on a machine of the same architecture as the target, you should skip this chapter.

- Install the cross compiler of your distribution
- Specify the target architecture in ARCH environment variable
- Specify the prefix (and path) of a cross compiler in `CROSS_COMPILE` environment variable

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

  Or just 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 like:

- 64-bit ARM:

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

- 32-bit ARM (ARMv7 / 32-bit mode of ARMv8):

  ```
  $ sudo apt-get install gcc-arm-linux-gnueabihf
  $ export CROSS_COMPILE=arm-linux-gnueabihf-
  $ export ARCH=arm
  ```

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

Configuring kernel

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

The following targets from the Linux kernel Kbuild framework are used to load the default kernel configuration for LSDK:

- `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 different platforms for LSDK will be:

- For Layerscape ARMv8 platforms in 64bit mode:

  ```
  $ make defconfig lsdk.config
  ```

- For Layerscape ARMv7 platforms:

  ```
  $ make multi_v7_defconfig multi_v7_lpae.config lsdk.config
  ```

- For Layerscape ARMv8 platforms in 32bit mode:

  ```
  $ make multi_v7_defconfig multi_v7_lpae.config multi_v8.config lsdk.config
  ```

To further fine tune the configuration base on your system need you can use the following make commands.

- `$ make menuconfig`

  Choose configure options in text based color menus, radiolists & dialogs. It is a good way to navigate through all the selectable kernel configure options in a well-organized human-readable hierarchy and you can get a description of every option when it is highlighted by selecting the `<Help>` button. In the device driver part of this User's Manual we also provided the path to the configure 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 without selecting each one of them in the menuconfig. In the device driver part of this User's Manual, we listed the `CONFIG_*` symbols needed by a specific feature/driver. 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 `$ make ${FEATURE}.config` command, it will enable all these listed kernel configure options together.

Building kernel

Building the kernel is simple.

- To build kernel images and device tree images.

  ```
  make
  ```

- To build loadable kernel modules:

  ```
  make modules
  ```

You can supply `-j <NUM>` option 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 (e.g. /folder/to/install) by using 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 LSDK.

Using the flex-build scripts

- 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 *flexbuild -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/
  ```

  — **For 32-bit ARM:**

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

- 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}
```

- Use flex-installer to deploy the updated boot partition and rootfs to the device following the Layerscape SDK user guide.

Update the target filesystem directly

This 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 (e.g. using scp, tftp, or etc.).

- Copy your Image file to `/boot` folder on the target using `cp` if compiled locally; Use any available remote update approach if compiled remotely.
- Copy dtb files to `/boot` folder on the target using `cp` if compiled locally; Use any available remote update approach to do the same if compiled remotely.
- Update kernel modules. (Note: kernel modules are required to be updated when you updated the kernel image).
  - If you compiled the kernel on the target device locally. Use the command below:

    ```
    $ make modules_install
    ```
  - If you compiled the kernel remotely. Do the following:
    - Install the modules into a temporary folder (e.g. `/tmp/lsdk/`).

      ```
      $ make modules_install INSTALL_MOD_PATH=/tmp/lsdk/
      ```
    - Transfer the `lib/` directory from the temporary location above to the target device using any file transfer approach and put it in the / path of the filesystem.

7.2 Device Drivers

7.2.1 Enhanced Direct Memory Access (eDMA)

Description

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

Kernel Configure Options

Tree View

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

Kernel Configure Tree View Options	Description

`Device Drivers --->`
`    [*] DMA Engine support  ---> --->`
`        <*>   Freescale eDMA engine support`
	DMA engine subsystem driver and eDMA driver support

Identifier

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

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

Device Tree Binding

Device Tree Node

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

```
edma0: edma@2c00000 {
                        #dma-cells = <2>;
                        compatible = "fsl,vf610-edma";
                        reg = <0x0 0x2c00000 0x0 0x10000>,
                                <0x0 0x2c10000 0x0 0x10000>,
                                <0x0 0x2c20000 0x0 0x10000>;
                        interrupts = <GIC_SPI 135 IRQ_TYPE_LEVEL_HIGH>,
                                        <GIC_SPI 135 IRQ_TYPE_LEVEL_HIGH>;
                        interrupt-names = "edma-tx", "edma-err";
                        dma-channels = <32>;
                        big-endian;
                        clock-names = "dmamux0", "dmamux1";
                        clocks = <&platform_clk 1>,
                                <&platform_clk 1>;
                };
```

Device Tree Node Binding for Slave Device

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

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

Source Files

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

Table 83. Source Files

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

Verification in Linux

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

   ```
   root@ls1021aqds:~# i2cdetect 0
   WARNING! This program can confuse your I2C bus, cause data loss and worse!
   I will probe file /dev/i2c-0.
   I will probe address range 0x03-0x77.
   Continue? [Y/n]
   ```

```
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- 69 -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
root@ls1021aqds:~# i2cdump 0 0x69 i
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f    0123456789abcdef
00: 05 07 ff ff 5d 55 10 55 11 05 1e 00 e8 03 b5 ff    ??..]U?U???.???.
10: ff e8 03 95 00 00 00 00 aa fe 9a 00 00 00 00 78    .???....???....x
20: 05 12 04 ff 00 7f 40 14 1d 60 3c 83 05 00 40 00    ???..?@??`<??.@.
30: fe 80 c6 29 00 00 00 7a 00 ff ff ff ff ff ff ff    ???)...z........
40: 05 07 ff ff 5d 55 10 55 11 05 1e 00 e8 03 b5 ff    ??..]U?U???.???.
50: ff e8 03 95 00 00 00 00 aa fe 9a 00 00 00 00 78    .???....???....x
60: 05 12 04 ff 00 7f 40 14 1d 60 3c 83 05 00 40 00    ???..?@??`<??.@.
70: fe 80 c6 29 00 00 00 7a 00 ff ff ff ff ff ff ff    ???)...z........
80: 07 ff ff 5d 55 10 55 11 05 1e 00 e8 03 b5 ff ff    ?..]U?U???.???..
90: e8 03 95 00 00 00 00 aa fe 9a 00 00 00 00 78 00    ???....???....x.
a0: 12 04 ff 00 7f 40 14 1d 60 3c 83 05 00 40 00 fe    ??..?@??`<??.@.?
b0: 80 c6 29 00 00 00 7a 00 ff ff ff ff ff ff ff ff    ??)...z.........
c0: 07 ff ff 5d 55 10 55 11 05 1e 00 e8 03 b5 ff ff    ?..]U?U???.???..
d0: e8 03 95 00 00 00 00 aa fe 9a 00 00 00 00 78 00    ???....???....x.
e0: 12 04 ff 00 7f 40 14 1d 60 3c 83 05 00 40 00 fe    ??..?@??`<??.@.?
f0: 80 c6 29 00 00 00 7a 00 ff ff ff ff ff ff ff ff    ??)...z.........
root@ls1021aqds:~# cat /proc/interrupts
           CPU0       CPU1
 29:          0          0       GIC  29  arch_timer
 30:       5563       5567       GIC  30  arch_timer
112:        260          0       GIC 112  fsl-lpuart
120:         32          0       GIC 120  2180000.i2c
121:          0          0       GIC 121  2190000.i2c
167:          8          0       GIC 167  eDMA
IPI0:          0          1  CPU wakeup interrupts
IPI1:          0          0  Timer broadcast interrupts
IPI2:       1388       1653  Rescheduling interrupts
IPI3:          0          0  Function call interrupts
IPI4:          2          4  Single function call interrupts
IPI5:          0          0  CPU stop interrupts
Err:          0
root@ls1021aqds:~#
```

7.2.2 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)
```

7.2.3 DCU Display Device Driver User Manual

Description

This manual describes how to use the Two Dimensional Animation and Compositing Engine (2D-ACE or DCU) and frame buffer on TWR-LS1021A board.

Module Loading

The DCU device driver supports kernel built-in and module.

U-Boot Configuration

Please use ‘ls1021atwr_lpuart_config’ to build the uboot.

Runtime options.

Env Variable	Description	Sub Option		Option Description
bootargs	Kernel command line argument passed to kernel	HDMI	console=ttyLP0,115200 hdmi	select LPUART0 as the system console
		LCD	console=ttyLP0,115200	

Kernel Configure Options

Tree View

Below are the Kernel Configure Tree View options need to be set/unset while doing "make menuconfig" for kernel and enable DCU/HDMI drivers and Linux Penguin Logo picture.

```
Device Drivers  --->

   < > Multimedia support  ----
       Graphics support  --->
           <*> Support for frame buffer devices  --->
               <*> Si Image SII9022 DVI/HDMI Interface Chip
               <*> Freescale DCU framebuffer support
                   …
           [ ] Exynos Video driver support  ----
           [ ] Backlight & LCD device support  ----
               Console display driver support  --->
                   <*> Framebuffer Console support
                   [*] Map the console to the primary display device
                   [*] Framebuffer Console Rotation
           [*] Bootup logo  --->
               --- Bootup logo
               [*] Standard black and white Linux logo
               [*] Standard 16-color Linux logo
               [*] Standard 224-color Linux logo
   < > Sound card support  ----
```

Identifier

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

Special Configure needs to be enabled ("Y") for LS1021A. Please find in below table with default value as "N"

Option	Values	Default Value	Description
CONFIG_ FB_FSL_SII902X	y/m/n	y	Si Image SII9022 DVI/HDMI Interface Chip
CONFIG_FB_FSL_DCU	y/m/n	y	NXP DCU framebuffer supportt
CONFIG_LOGO	y/m/n	y	Bootup logo
CONFIG_LOGO_LINUX_MONO	y/m/n	y	Standard black and white Linux logo
CONFIG_LOGO_LINUX_VGA16	y/m/n	y	Standard 16-color Linux logo
CONFIG_LOGO_LINUX_CLUT224	y/m/n	y	Standard 224-color Linux logo
CONFIG_FRAMEBUFFER_CONSOLE	y/m/n	y	Framebuffer Console support

Device Tree Binding

Special Configure needs to be enabled ("Y") for LS1021A. Please find in below table with default value as "N"

The default configuration dsiplay through LCD, like below specified .

arch/arm/boot/dts/ls1021a.dtsi

```
dcu0: dcu@2ce0000 {
            compatible = "fsl,vf610-dcu";
             reg = <0x0 0x2ce0000 0x0 0x10000>;
             interrupts = <GIC_SPI 172 IRQ_TYPE_LEVEL_HIGH>;
             clocks = <&platform_clk 0>;
             clock-names = "dcu";
             scfg-controller = <&scfg>;
              big-endian;
              status = "disabled";
};
```

arch/arm/boot/dts/ls1021a-twr.dts

```
&dcu0 {
        display = <&display>;
        status = "okay";

        display: display@0 {
                bits-per-pixel = <24>;

                display-timings {
                        native-mode = <&timing0>;
                        timing0: nl4827hc19 {
                                clock-frequency = <10870000>;
                                hactive = <480>;
                                vactive = <272>;
                                hback-porch = <2>;
                                hfront-porch = <2>;
                                vback-porch = <2>;
                                vfront-porch = <2>;
                                hsync-len = <41>;
                                vsync-len = <4>;
                                hsync-active = <1>;
                                vsync-active = <1>;
                        };
                };
        };
};
```

Ramdisk:

Please use the 'fsl-image-x11-ls1021a(XXXXX)rootfs.ext2.gz.gz' ramdisk from each release images, or you can just use the ramdisk image which has 'x11' label.

If you want to HDMI display, change the following configuration:

```
arch/arm/boot/dts/ls1021a-twr.dtscan
diff --git
a/arch/arm/boot/dts/ls1021a-twr.dts b/arch/arm/boot/dts/ls1021a-twr.dtsindex
cc351e3..928d376 100644---
a/arch/arm/boot/dts/ls1021a-twr.dts+++
b/arch/arm/boot/dts/ls1021a-twr.dts@@ -122,7 +122,7
@@
port {
         dcu_out: endpoint {
-            remote-endpoint = <&panel_in>;
+            remote-endpoint = <&sii9022a_out>;
         };
     };
 };
@@ -204,6 +204,18 @@
         VDDIO-supply = <&reg_3p3v>;
         clocks = <&sys_mclk>;
     };
+
+    sii9022a: hdmi@39 {
+        compatible = "sil,sii9022";
+        reg = <0x39>;
+        interrupts = <GIC_SPI 167 IRQ_TYPE_EDGE_RISING>;
+
+        port@0 {
+            sii9022a_out: endpoint {
+                remote-endpoint = <&dcu_out>;
+            };
+        };
+    };
 };

 &ifc {
```

Source Files

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

Source File	Description
drivers/video/fsl-dcu-fb.c	NXP DCU driver

Testing LCD/DHMI at Uboot Level

1. Display with LCD:

```
=> setenv video-mode  "fslfb:480x272-32@60,monitor=twr_lcd"
=> save
=> setenv stdout vga
```

2. Display with HDMI:

```
=> setenv video-mode  "fslfb:640x480-32@60,monitor=hdmi"
=> save
=> setenv stdout vga
```

Testing LCD at Kernel Level

1. Configure and rebuild the kernel as configuration list above, let the DCU driver built into the Kernel Image.
2. Boot up Linux kernel, upon the kernel has been uncompressed, the TFT Panel will display the Linux Penguin Logo.
3. And then after the root filesystem has been monted, and the Xwindows Desktop will be display.
4. Or also you can start the Xwindow using:

```
root@ls1021atwr:~# killall matchbox-window-manager
root@ls1021atwr:~# xinit /etc/init.d/xserver-nodm restart
```

5. Just plug out and plug in the HDMI to test the hot plug.

Testing HDMI at Kernel Level

1. 1.Configure and rebuild the kernel as configuration list above, let the HDMI and DCU drivers built into the Kernel Image.
2. Boot up Linux kernel, upon the kernel has been uncompressed, the TFT Panel won't display any picture correctly.
3. And then after the root filesystem has been monted, and the Xwindows Desktop will be displayed on the HDMI Monitor.
4. Or also you can start the Xwindow using:

```
root@ls1021atwr:~# killall matchbox-window-manager
root@ls1021atwr:~# xinit /etc/init.d/xserver-nodm restart

Note: Please unplugged the TWR-LDC_RGB daughter board when testing the
HDMI.
```

Known Bugs, Limitations, or Technical Issues

Unplug the SD card before testing the DCU/HDMI, or the system will hang.

7.2.4 Enhanced Secured Digital Host Controller (eSDHC)

Description

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

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`
`(8)          Number of minors per block device`
`[*]          Use bounce buffer for simple hosts`
	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	8	Number of minors per block device
CONFIG_MMC_BLOCK_BOUNCE	y/n	y	Enable continuous physical memory for transmit
CONFIG_MMC_SDHCI	y/n	y	Enable generic sdhc interface
CONFIG_MMC_SDHCI_PLTFM	y/n	y	Enable common helper function support for sdhci platform and OF drivers
CONFIG_MMC_SDHCI_OF_ESDHC	y/n	y	Enable NXP eSDHC support

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>;
};
```

Verification in U-Boot

The U-Boot log

```
=> mmcinfo
Device: FSL_SDHC
Manufacturer ID: 74
OEM: 4a45
Name: SDC
Tran Speed: 50000000
Rd Block Len: 512
SD version 3.0
High Capacity: Yes
Capacity: 7.5 GiB
Bus Width: 4-bit
Erase Group Size: 512 Bytes
=> mw.l 81000000 11111111 100
=> mw.l 82000000 22222222 100
=> cmp.l 81000000 82000000 100
word at 0x0000000081000000 (0x11111111) != word at 0x0000000082000000 (0x22222222)
Total of 0 word(s) were the same
=> mmc write 81000000 0 2

MMC write: dev # 0, block # 0, count 2 ... 2 blocks written: OK
=> mmc read 82000000 0 2

MMC read: dev # 0, block # 0, count 2 ... 2 blocks read: OK
=> cmp.l 81000000 82000000 100
Total of 256 word(s) were the same
=>
```

Verification in Linux

Set U-Boot environment

```
=> setenv hwconfig sdhc
```

The linux booting log

```
......
[    3.913163] sdhci: Secure Digital Host Controller Interface driver
[    3.919339] sdhci: Copyright(c) Pierre Ossman
[    3.931467] sdhci-pltfm: SDHCI platform and OF driver helper
[    3.938900] sdhci-esdhc 1560000.esdhc: No vmmc regulator found
[    3.944728] sdhci-esdhc 1560000.esdhc: No vqmmc regulator found
[    3.978676] mmc0: SDHCI controller on 1560000.esdhc [1560000.esdhc] using ADMA 64-bit
[    4.197784] mmc0: new high speed SDHC card at address b368
[    4.203502] mmcblk0: mmc0:b368 SDC   7.45 GiB
```

Partition the card with fdisk

```
~# fdisk /dev/mmcblk0

Welcome to fdisk (util-linux 2.26.2).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.
```

```
Device does not contain a recognized partition table.
Created a new DOS disklabel with disk identifier 0x5a5f34b3.

Command (m for help): n
Partition type
   p   primary (0 primary, 0 extended, 4 free)
   e   extended (container for logical partitions)
Select (default p):

Using default response p.
Partition number (1-4, default 1):
First sector (2048-15628287, default 2048):
Last sector, +sectors or +size{K,M,G,T,P} (2048-15628287, default 15628287):

Created a new partition 1 of type 'Linux' and of size 7.5 GiB.

Command (m for help): w
The partition table has been altered.
Calling ioctl() [  410.501876]  mmcblk0: p1
to re-read partition table.
Syncing disks.

~#
~# fdisk -l /dev/mmcblk0
Disk /dev/mmcblk0: 7.5 GiB, 8001683456 bytes, 15628288 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x5a5f34b3

Device         Boot Start      End  Sectors  Size Id Type
/dev/mmcblk0p1       2048 15628287 15626240  7.5G 83 Linux
```

Format the card with mkfs

```
~# mkfs.ext2 /dev/mmcblk0p1
mke2fs 1.42.9 (28-Dec-2013)
Discarding device blocks: [   37.611042] random: nonblocking pool is initialized
done
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
488640 inodes, 1953280 blocks
97664 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=2000683008
60 block groups
32768 blocks per group, 32768 fragments per group
8144 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632

Allocating group tables: done
Writing inode tables: done
Writing superblocks and filesystem accounting information: done

~#
```

Mount, read, and write

```
~# mount /dev/mmcblk0p1 /mnt/
~# ls /mnt/
lost+found
~# cp -r /lib /mnt/
~# sync
~# ls /mnt/
```

```
lib  lost+found
~# umount /dev/mmcblk0p1
~# ls /mnt/
~#
```

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 LS1046ARDB, 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.
6. For LS1012ARDB RevD and later versions, I2C reading for DIP switch setting is not reliable so U-Boot could not enable/disable SDHC2 automatically. If SDHC2 is used, "esdhc1" should be set in U-Boot hwconfig environment to enable it manually.

7.2.5 IEEE 1588

Description

From IEEE-1588 perspective, the components required are:

1. IEEE-1588 extensions to the gianfar driver or DPAA/DPAA2 driver.
2. A stack application for IEEE-1588 protocol.

IEEE 1588 device driver supports either kernel built-in or module.

Kernel Configure Options

Tree View

1. eTSEC - Using PTPd stack

Kernel Configure Tree View Options	Description

`Device Drivers  --->`
`  PTP clock support  --->`
`     <*> Freescale eTSEC as PTP clock`
	Enable 1588 driver for PTPd stack

2. DPAA - Using PTPd stack

Kernel Configure Tree View Options	Description

`Device Drivers  --->`
`  PTP clock support  --->`
`     <*> Freescale DPAA as PTP clock`
	Enable 1588 driver for PTPd stack

3. DPAA2 - Using PTPd stack

Kernel Configure Tree View Options	Description

`Device Drivers  --->`
`  PTP clock support  --->`
`     <*> Freescale DPAA2 as PTP clock`
	Enable 1588 driver for PTPd stack

Compile-time Configuration Options

1. eTSEC - Using PTPd stack

Option	Values	Default Value	Description
CONFIG_GIANFAR	y/n	y	Enable eTSEC driver support
CONFIG_PTP_1588_CLOCK_GIANFAR	y/n	y	Enables 1588 driver support

2. DPAA - Using PTPd stack

Option	Values	Default Value	Description
CONFIG_PTP_1588_CLOCK_DPAA	y/n	n	Enable IEEE 1588 support
CONFIG_FSL_SDK_DPAA_ETH	y/n	y	Enables DPAA driver support

3. DPAA2 - Using PTPd stack

Option	Values	Default Value	Description
CONFIG_PTP_1588_CLOCK_DPAA2	y/n	y	Enable IEEE 1588 support
CONFIG_FSL_DPAA2_ETH	y/n	y	Enables DPAA2 driver support

Source Files

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

1. eTSEC (for PTPd)

Source File	Description
drivers/net/ethernet/freescale/gianfar.c	eTSEC Ethernet driver
drivers/net/ethernet/freescale/gianfar_ptp.c	IEEE 1588 driver

2. DPAA (for PTPd)

Source File	Description
drivers/net/ethernet/freescale/sdk_dpaa/dpaa_ptp.c	IEEE 1588 driver
drivers/net/ethernet/freescale/sdk_dpaa/dpaa_eth.c	DPAA Ethernet driver

3. DPAA2 (for PTPd)

Source File	Description
drivers/staging/fsl-dpaa2/rtc/rtc.c	IEEE 1588 driver
drivers/staging/fsl-dpaa2/ethernet/dpaa2-eth.c	DPAA2 Ethernet driver

Device Tree Binding

1. eTSEC (for PTPd)

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

Example:

```
ptp_clock@2d10e00 {
    compatible = "fsl,etsec-ptp";
    reg = <0x0 0x2d10e00 0x0 0xb0>;
    interrupts = <GIC_SPI 173 IRQ_TYPE_LEVEL_HIGH>;
    fsl,tclk-period = <5>;
    fsl,tmr-prsc    = <2>;
    fsl,tmr-add     = <0xaaaaaaab>;
    fsl,tmr-fiper1  = <999999990>;
    fsl,tmr-fiper2  = <99990>;
    fsl,max-adj     = <499999999>;
};
```

2. DPAA (For PTPd)

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

Example:

```
ptp_timer0: ptp-timer@fe000 {
    compatible = "fsl,fman-ptp-timer", "fsl,fman-rtc";
    reg = <0xfe000 0x1000>;
};
```

3. DPAA2

   NA.

Verification in Linux

Connect Ethernet interfaces of two boards with back-to-back method (for example, eth0 to eth0).

One board runs as master and the other one runs as slave.

- **The linux booting log**

  ```
  ...
  pps pps0: new PPS source ptp0
  ...
  ```

- **On the master side**

  ```
  # ifconfig eth0 up
  # ifconfig eth0 192.168.1.100
  # ptpd2 -i eth0 -MV
  ```

- **On the slave side**

  ```
  # ifconfig eth0 up
  # ifconfig eth0 192.168.1.200
  # ptpd2 -i eth0 -sV --servo:kp=0.32 --servo:ki=0.05
  ```

  The slave side would print synchronization messages.

- **Note:**

  ptpd2 stack would use /dev/ptp0 in default. If 1588 timer is initialized as ptp1 or others, please use '-o' option to clarify that such as,

  ```
  -o /dev/ptp1
  ```

Known Bugs, Limitations, or Technical Issues

- Packet loss issue could be observed on LS1021ATWR when Ethernet interfaces are connected in back-to-back way. The root cause is that the PHY supports IEEE 802.11az EEE mode by default. The low speed traffic will make it go into low power mode. It affects 1588 synchronization performance greatly. Use the workaround below to disable the feature.

  ```
  # ifconfig eth0 up
  # ethtool --set-eee eth0 advertise 0
  # ifconfig eth0 down
  # ifconfig eth0 up
  ```

7.2.6 Integrated Flash Controller (IFC)

7.2.6.1 Integrated Flash Controller NOR Flash User Manual

Description

NXP’s Integrated Flash Controller can be used to connect various types of flashes e.g. NOR/NAND on board for boot functionality as well as data storage.

U-Boot Configuration

Compile time options

Below are major u-boot configuration options related to this feature defined in platform specific config files under include/configs/ directory.

Option Identifier	Description
CONFIG_FSL_IFC	Enable IFC support
CONFIG_FLASH_CFI_DRIVER CONFIG_SYS_FLASH_CFI CONFIG_SYS_FLASH_EMPTY_INFO	Enable CFI Driver for NOR Flash devices

Source Files

The following source files are related to this feature in u-boot.

Source File	Description
./drivers/misc/fsl_ifc.c	Set up the different chip select parameters from board header file
drivers/mtd/cfi_flash.c	CFI driver support for NOR flash devices

Kernel Configure Options

Tree View

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

Kernel Configure Tree View Options	Description
Device Drivers ---> <*> Memory Technology Device (MTD) support ---> [*] MTD partitioning support [*] Command line partition table parsing <*> Flash partition map based on OF description <*> Direct char device access to MTD devices -*- Common interface to block layer for MTD 'translation layers' <*> Caching block device access to MTD devices < > FTL (Flash Translation Layer) support RAM/ROM/Flash chip drivers ---> <*> Detect flash chips by Common Flash Interface (CFI) probe <*> Support for Intel/Sharp flash chips <*> Support for AMD/Fujitsu/Spansion flash chips Mapping drivers for chip access --->	These options enable CFI support for NOR Flash under MTD subsystem and Integrated Flash Controller support on Linux

Table continues on the next page...

Table continued from the previous page...

Kernel Configure Tree View Options	**Description**
`<*> Flash device in physical memory map based on OF description`	

`File systems --->`
`[*] Miscellaneous filesystems --->`
`<*> Journalling Flash File System v2 (JFFS2) support`
	This option enables JFFS2 file system support for MTD Devices

Identifier

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

Special Configure needs to be enabled("Y") for LS1021. Please find in below table with default value as "N"

Option	Values	Default Value	Description
CONFIG_FSL_IFC	Y/N	Y	Integrated Flash Controller support
CONFIG_MTD	Y/N	Y	Memory Technology Device (MTD) support
CONFIG_MTD_PARTITIONS	Y/N	Y	MTD partitioning support
CONFIG_MTD_CMDLINE_PARTS	Y/N	Y	Allow generic configuration of the MTD partition tables via the kernel command line.
CONFIG_MTD_OF_PARTS	Y/N	Y	This provides a partition parsing function which derives the partition map from the children of the flash nodes described in Documentation/powerpc/booting-without-of.txt
CONFIG_MTD_CHAR	Y/N	Y	Direct char device access to MTD devices
CONFIG_MTD_BLOCK	Y/N	Y	Caching block device access to MTD devices
CONFIG_MTD_CFI	Y/N	Y	Detect flash chips by Common Flash Interface (CFI) probe
CONFIG_MTD_GEN_PROBE	Y/N	Y	NA
CONFIG_MTD_MAP_BANK_WIDTH_1	Y/N	Y	Support 8-bit buswidth
CONFIG_MTD_MAP_BANK_WIDTH_2	Y/N	Y	Support 16-bit buswidth
CONFIG_MTD_MAP_BANK_WIDTH_4	Y/N	Y	Support 32-bit buswidth
CONFIG_MTD_PHYSMAP_OF	Y/N	Y	Flash device in physical memory map based on OF description
CONFIG_FTL	Y/N	N	FTL (Flash Translation Layer) support
CONFIG_MTD_CFI_INTELEXT	Y/N	Y	Support for Intel/Sharp flash chips
CONFIG_MTD_CFI_AMDSTD	Y/N	Y	Support for AMD/Fujitsu/Spansion flash chips

Device Tree Binding

Documentation/devicetree/bindings/memory-controllers/fsl/ifc.txt

Flash partitions are specified by platform device tree.

Source Files

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

Source File	Description
drivers/memory/fsl_ifc.c	Integrated Flash Controller driver to handle error interrupts
drivers/mtd/mtdpart.c	Simple MTD partitioning layer
drivers/mtd/mtdblock.c	Direct MTD block device access
drivers/mtd/mtdchar.c	Character-device access to raw MTD devices.
drivers/mtd/ofpart.c	Flash partitions described by the OF (or flattened) device tree
drivers/mtd/ftl.c	FTL (Flash Translation Layer) support
drivers/mtd/chips/cfi_probe.c	Common Flash Interface probe
drivers/mtd/chips/cfi_util.c	Common Flash Interface support
drivers/mtd/chips/cfi_cmdset_0001.c	Support for Intel/Sharp flash chips
drivers/mtd/chips/cfi_cmdset_0002.c	Support for AMD/Fujitsu/Spansion flash chips

Verification in U-Boot

Test the Read/Write/Erase functionality of NOR Flash

1. Boot the u-boot with above config options to get NOR Flash access enabled. Check this in boot log,

   ```
   FLASH: * MiB
   ```

   where * is the size of NOR Flash
2. Erase NOR Flash
3. Make test pattern on memory e.g. DDR
4. Write test pattern on NOR Flash
5. Read the test pattern from NOR Flash to memory e.g DDR
6. Compare the test pattern data to verify functionality.

Test Log :

Test log with initial u-boot log removed

```
--
--

FLASH: 128 MiB

--
--
/* u-boot prompt */
=> mw.b 80000000 0xa5 10000
=> md 80000000
80000000: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
80000010: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
80000020: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
80000030: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
=> protect off all
Un-Protect Flash Bank # 1
=> erase 0x584100000 +0x10000
```

```
. done
Erased 1 sectors
=> cp.b 80000000 0x584100000 10000
Copy to Flash... 9....8....7....6....5....4....3....2....1....done
=> cmp.b 80000000 0x584100000 10000
Total of 65536 bytes were the same
=>
```

Verification in Linux

To cross check whether IFC NOR driver has been configured in the kernel or not, see the kernel boot log with following entries. Please note mtd partition number can be changed depending upon device tree.

```
[    2.368207] 60000000.nor: Found 1 x16 devices at 0x0 in 16-bit bank. Manufacturer ID
0x000001 Chip ID 0x002801
[    2.378219] Amd/Fujitsu Extended Query Table at 0x0040
[    2.383374]   Amd/Fujitsu Extended Query version 1.3.
[    2.388427] number of CFI chips: 1
[    2.391835] 8 cmdlinepart partitions found on MTD device 60000000.nor
[    2.398277] Creating 8 MTD partitions on "60000000.nor":
[    2.403591] 0x000000000000-0x000000100000 : "nor_bank0_rcw"
[    2.409553] 0x000000100000-0x000001000000 : "nor_bank0_uboot"
[    2.415653] 0x000001000000-0x000002000000 : "nor_bank0_kernel"
[    2.421839] 0x000002000000-0x000004000000 : "nor_bank0_rootfs"
[    2.428027] 0x000004000000-0x000004100000 : "nor_bank4_rcw"
[    2.433948] 0x000004100000-0x000005000000 : "nor_bank4_uboot"
[    2.440043] 0x000005000000-0x000006000000 : "nor_bank4_kernel"
[    2.446228] 0x000006000000-0x000008000000 : "nor_bank4_rootfs"
```

Note: NOR address and number of partition will vary from SoC to SoC supported in LSDK

To verify NOR flash device accesses see the following test,

```
[root@ root]# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00100000 00020000 "nor_bank0_rcw"
mtd1: 00f00000 00020000 "nor_bank0_uboot"
mtd2: 01000000 00020000 "nor_bank0_kernel"
mtd3: 02000000 00020000 "nor_bank0_rootfs"
mtd4: 00100000 00020000 "nor_bank4_rcw"
mtd5: 00f00000 00020000 "nor_bank4_uboot"
mtd6: 01000000 00020000 "nor_bank4_kernel"
mtd7: 02000000 00020000 "nor_bank4_rootfs"
mtd8: 01000000 00040000 "nand_uboot"
mtd9: 01000000 00040000 "nand_kernel"
mtd10: 02000000 00040000 "nand_free"
mtd11: 00600000 00001000 "uboot"
mtd12: 00a00000 00001000 "free"
mtd13: 00080000 00001000 "spi0.1"
mtd14: 00800000 00001000 "spi0.2"

[root@ root]# flash_eraseall -j /dev/mtd2

Erasing 128 Kibyte @ 1400000 -- 100% complete. Cleanmarker written at 13e0000.

[root@P1010RDB root]# mount -t jffs2 /dev/mtdblock2 /mnt/

JFFS2 notice: (1202) jffs2_build_xattr_subsystem: complete building xattr subsystem, 0 of
xdatum (0 unchecked, 0 orphan) and 0 of xref (0 dead, 0 orphan) found.

[root@ root]# cd /mnt/

[root@ mnt]# ls -l

[root@ mnt]# touch flash_file

[root@ root]# umount mnt
```

```
//ls must list local_file
[root@ root]# ls mnt
//mount again
[root@ root]# mount -t  jffs2 /dev/mtdblock2 /mnt/
JFFS2 notice: (1219) jffs2_build_xattr_subsystem: complete building xattr subsystem, 0 of
xdatum (0 unchecked, 0 orphan) and 0 of xref (0 dead, 0 orphan) found.
//use ls ; it must show the created file
[root@ root]# ls /mnt/
flash_file
//unmount
[root@ root]# umount /mnt/
```

7.2.6.2 Integrated Flash Controller NAND Flash User Manual

Description

NXP's Integrated Flash Controller can be used to connect various types of flashes (e.g. NOR/NAND) on board for boot functionality as well as data storage.

U-Boot Configuration

Compile time options

Below are major U-Boot configuration options related to this feature defined in platform specific config files under include/configs/ directory.

Option Identifier	Description
CONFIG_FSL_IFC	Enable IFC support
CONFIG_NAND_FSL_IFC	Enable NAND Machine support on IFC
CONFIG_SYS_MAX_NAND_DEVICE	No of NAND Flash chips on platform
CONFIG_MTD_NAND_VERIFY_WRITE	Verify NAND flash writes
CONFIG_CMD_NAND	Enable various commands support for NAND Flash
CONFIG_SYS_NAND_BLOCK_SIZE	Block size of the NAND flash connected on Platform

Source Files

The following source files are related to this feature in u-boot.

Source File	Description
./drivers/misc/fsl_ifc.c	Set up the different chip select parameters from board header file
drivers/mtd/nand/fsl_ifc_nand.c	IFC nand flash machine driver file

Kernel Configure Options

Tree View

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

Kernel Configure Tree View Options	Description
	These options enable Integrated Flash Controller

Kernel Configure Tree View Options	Description
Device Drivers ---> <*> Memory Technology Device (MTD) support ---> [*] MTD partitioning support [*] Command line partition table parsing= <*> Flash partition map based on OF description <*> Direct char device access to MTD devices -*- Common interface to block layer for MTD 'translation layers' <*> Caching block device access to MTD devices <*> NAND Device Support ---> <*> NAND support for Freescale IFC controller **Enable UBIFS filesystem in linux configuration** Device Drivers ---> <*> Memory Technology Device (MTD) support ---> UBI - Unsorted block images ---> <*> Enable UBI (4096) UBI wear-leveling threshold (1) Percentage of reserved eraseblocks for bad eraseblocks handling < > MTD devices emulation driver (gluebi) *** UBI debugging options *** [ ] UBI debugging File systems ---> [*] Miscellaneous filesystems ---> <*> UBIFS file system support	NAND support to work with MTD subsystem available on Linux. Also UBIFS support needs to be enabled.

Kernel Configure Tree View Options	Description
[*] Extended attributes support [ ] Advanced compression options [ ] Enable debugging	

Identifier

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

Option	Values	Default Value	Description
CONFIG_FSL_IFC	y/n	y	Enable Integrated Flash Controller support
CONFIG_MTD_NAND_FSL_IFC	y/n	Y	Enable Integrated Flash Controller NAND Machine support
CONFIG_MTD_PARTITIONS	y/n	Y	MTD partitioning support
CONFIG_MTD_CMDLINE_PARTS	y/n	Y	Allow generic configuration of the MTD partition tables via the kernel command line.
CONFIG_MTD_OF_PARTS	y/n	Y	This provides a partition parsing function which derives the partition map from the children of the flash nodes described in Documentation/powerpc/booting-without-of.txt
CONFIG_MTD_CHAR	y/n	Y	Direct char device access to MTD devices
CONFIG_MTD_BLOCK	y/n	Y	Caching block device access to MTD devices
CONFIG_MTD_GEN_PROBE	y/n	Y	NA
CONFIG_MTD_PHYSMAP_OF	y/n	Y	Flash device in physical memory map based on OF description

Device Tree Binding

`Documentation/devicetree/bindings/memory-controllers/fsl/ifc.txt`

Flash partitions are specified by platform device tree.

Source Files

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

Source File	Description
drivers/memory/fsl_ifc.c	Integrated Flash Controller driver to handle error interrupts
drivers/mtd/nand/fsl_ifc_nand.c	Integrated Flash Controller NAND Machine driver
include/linux/fsl_ifc.h	IFC Memory Mapped Registers

Verification in U-Boot

Test the Read/Write/Erase functionality of NAND Flash

1. Boot the u-boot with above config options to get NAND Flash driver enabled. Check this in boot log,

   ```
   NAND: * MiB
   ```

   ```
   Where * is NAND flash size
   ```

2. Erase NAND Flash

3. Make test pattern on memory e.g. DDR
4. Write test pattern on NAND Flash
5. Read the test pattern from NAND Flash to memory e.g DDR
6. Compare the test pattern data to verify functionality.

Test Log :

```
...
...

NAND:  512 MiB

...
...

/* U-boot prompt */
=> nand erase.chip



NAND erase.chip: device 0 whole chip

Bad block table found at page 65504, version 0x01 Bad block table found at page 65472,
version 0x01

Skipping bad block at  0x01ff0000


Skipping bad block at  0x01ff4000


Skipping bad block at  0x01ff8000


Skipping bad block at  0x01ffc000




OK

=> mw.b 80000000 0xa5 100000

=> md 80000000

80000000: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

80000010: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

80000020: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

80000030: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................

=> nand write 80000000 0 100000


NAND write: device 0 offset 0x0, size 0x100000

 1048576 bytes written: OK

=> nand read 90000000 0 100000


NAND read: device 0 offset 0x0, size 0x100000
```

```
 1048576 bytes read: OK

=>  cmp.b 80000000 90000000 100000

Total of 1048576 bytes were the same
```

Verification in Linux

To cross check whether IFC NAND driver has been configured in the kernel or not, check the following. Please note mtd partition numbers can be changed depending upon board device tree

```
[root@(none) root]# cat /proc/mtd
dev:    size   erasesize name
mtd0: 00100000 00020000 "nor_bank0_rcw"
mtd1: 00f00000 00020000 "nor_bank0_uboot"
mtd2: 01000000 00020000 "nor_bank0_kernel"
mtd3: 02000000 00020000 "nor_bank0_rootfs"
mtd4: 01000000 00040000 "nand_uboot"
mtd5: 01000000 00040000 "nand_kernel"
mtd6: 02000000 00040000 "nand_free"


[root@(none) root]# flash_eraseall /dev/mtd4 Erasing 16 Kibyte @ f00000 -- 100% complete.


[root@(none) root]# ubiattach /dev/ubi_ctrl -m 4

UBI: attaching mtd4 to ubi0

UBI: physical eraseblock size:   16384 bytes (16 KiB)

UBI: logical eraseblock size:    15360 bytes

UBI: smallest flash I/O unit:    512

UBI: VID header offset:          512 (aligned 512)

UBI: data offset:                1024

UBI: empty MTD device detected

UBI: create volume table (copy #1)

UBI: create volume table (copy #2)

UBI: attached mtd4 to ubi0

UBI: MTD device name:            "NAND Root File System"

UBI: MTD device size:            15 MiB

UBI: number of good PEBs:        960

UBI: number of bad PEBs:         0

UBI: max. allowed volumes:       89

UBI: wear-leveling threshold:    4096

UBI: number of internal volumes: 1

UBI: number of user volumes:     0

UBI: available PEBs:            947

UBI: total number of reserved PEBs: 13
```

```
UBI: number of PEBs reserved for bad PEB handling: 9

UBI: max/mean erase counter: 0/0

UBI: image sequence number: 0

UBI: background thread "ubi_bgt0d" started, PID 7541 UBI device number 0, total 960 LEBs
(14745600 bytes, 14.1 MiB), available 947 LEBs (14545920 bytes, 13.9 MiB), LEB size 15360
bytes (15.0 KiB)


[root@(none) root]# ubimkvol /dev/ubi0 -N rootfs -s 14205KiB Volume ID 0, size 947 LEBs
(14545920 bytes, 13.9 MiB), LEB size 15360 bytes (15.0 KiB), dynamic, name "rootfs",
alignment 1


[root@(none) root]# mount -t ubifs /dev/ubi0_0 /mnt/

UBIFS: default file-system created

UBIFS: mounted UBI device 0, volume 0, name "rootfs"

UBIFS: file system size:   14361600 bytes (14025 KiB, 13 MiB, 935 LEBs)
UBIFS: journal size:       721920 bytes (705 KiB, 0 MiB, 47 LEBs)

UBIFS: media format:       w4/r0 (latest is w4/r0)

UBIFS: default compressor: lzo

UBIFS: reserved for root:  678333 bytes (662 KiB)


[root@(none) root]# cd /mnt/

[root@(none) mnt]# ls

[root@(none) mnt]# touch flash_file

[root@(none) mnt]# ls -l

total 0

-rw-r--r-- 1 root root 0 Jul  6 14:45 flash_file

[root@(none) mnt]# cd

[root@(none) root]# umount /mnt/

UBIFS: un-mount UBI device 0, volume 0


[root@(none) root]# mount -t ubifs /dev/ubi0_0 /mnt/

UBIFS: mounted UBI device 0, volume 0, name "rootfs"

UBIFS: file system size:   14361600 bytes (14025 KiB, 13 MiB, 935 LEBs)
UBIFS: journal size:       721920 bytes (705 KiB, 0 MiB, 47 LEBs)

UBIFS: media format:       w4/r0 (latest is w4/r0)

UBIFS: default compressor: lzo

UBIFS: reserved for root:  678333 bytes (662 KiB)
```

```
[root@(none) root]# ls -l /mnt/

total 0

-rw-r--r-- 1 root root 0 Jul  6 14:45 flash_file
```

Known Bugs, Limitations, or Technical Issues

Boards which have NAND Flash with 512byte page size, JFFS2 cannot be supported using H/W ECC support of IFC , as there is not enough remaining space in the OOB area.

To use JFFS2 use SOFT ECC.

7.2.7 Low Power Universal Asynchronous Receiver/Transmitter (LPUART)

Description

Low Power Universal asynchronous receiver/transmitter (LPUART) is a high speed and low power uart. Refer to below table for the NXP soc can support LPUART.

SOC	Num of LPUART module
LS1021A	6
LS1043A	6

U-boot ConfigurationCompile time options

Below are major u-boot configuration options related to this feature defined in platform specific config files under include/configs/ directory.

Option Identifier	Description
CONFIG_LPUART	Enable lpuart support
CONFIG_FSL_LPUART	Enable NXP lpuart support
CONFIG_LPUART_32B_REG	Select 32-bit lpuart register mode

Choosing predefined u-boot board configs:

Please make the defconfig include 'lpuart', like: ls1021atwr_nor_lpuart_defconfig. That's will supoort lpuart.

Runtime options

Env Variable	Env Description	Sub option	Option Description
bootargs	Kernel command line argument passed to kernel	console=ttyLP0,1152000	select LPUART0 as the system console

Kernel Configure Options

Tree View

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

Kernel Configure Tree View Options	Description

`Device Drivers --->`
`Character devices --->`
`Serial drivers --->`
`<*> Freescale lpuart serial port support`
`[*] Console on Freescale lpuart serial port`
	LPUART driver and enable console support

Identifier

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

Option	Values	Default Value	Description
CONFIG_SERIAL_FSL_LPUART	y/m/n	n	LPUART Driver

Device Tree Binding

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

```
lpuart0: serial@2950000 {
                        compatible = "fsl,vf610-lpuart";
                        reg = <0x0 0x2950000 0x0 0x1000>;
                        interrupts = <GIC_SPI 80 IRQ_TYPE_LEVEL_HIGH>;
                        clocks = <&sysclk>;
                        clock-names = "ipg";
                        fsl,lpuart32;
                        status = "okay";
```

Source Files

The following source file are related the this feature in u-boot.

Source File	Description
drivers/serial/serial_lpuart.c	The LPUART driver file

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

Source File	Description
drivers/tty/serial/fsl_lpuart.c	The LPUART driver file

Verification in U-Boot

1. Boot up U-Boot from bank0, and update rcw and u-boot for lpuart support to bank4, first copy the rcw and U-Boot binary to the tftp directory.
2. Please refer to the platform depoly document to update the rcw and uboot.
3. After all is updated, run u-boot command to switch to alt bank, then will bring up the new U-Boot to the lpuart console.

```
CPU:   Freescale LayerScape LS1020E, Version: 1.0, (0x87081010)
Clock Configuration:
       CPU0(ARMV7):1000 MHz,
       Bus:300  MHz, DDR:800  MHz (1600 MT/s data rate),
Reset Configuration Word (RCW):
       00000000: 0608000a 00000000 00000000 00000000
       00000010: 60000000 00407900 e0025a00 21046000
       00000020: 00000000 00000000 00000000 08038000
       00000030: 00000000 001b7200 00000000 00000000
I2C:   ready
Board: LS1021ATWR
CPLD:  V2.0
PCBA:  V1.0
VBank: 0
DRAM:  1 GiB
Using SERDES1 Protocol: 48 (0x30)
Flash: 0 Bytes
MMC:   FSL_SDHC: 0
EEPROM: NXID v16777216
PCIe1: Root Complex no link, regs @ 0x3400000
PCIe2: disabled
In:    serial
Out:   serial
Err:   serial
SATA link 0 timeout.
AHCI 0001.0300 1 slots 1 ports ? Gbps 0x1 impl SATA mode
flags: 64bit ncq pm clo only pmp fbss pio slum part ccc
Found 0 device(s).
SCSI:  Net:   eTSEC1 is in sgmii mode.
eTSEC2 is in sgmii mode.
eTSEC1, eTSEC2 [PRIME], eTSEC3
=>
```

Verification in Linux

1. After uboot startup, set the command line parameter to pass to the linux kernel including console=ttyLP0,115200 in boootargs. For deploy the ramdisk as rootfs, the bootargs can be set as: "set bootargs root=/dev/ram0 rw console=ttyLP0,115200"

   ```
   => set bootargs root=/dev/ram0 rw console=ttyLP0,115200

   => dhcp 81000000 <tftpboot dir>/zImage.ls1021a;tftp 88000000 <tftpboot dir>/
   initrd.ls1.uboot;tftp 8f000000 <tftpboot dir>/ls1021atwr.dtb;bootz 81000000 88000000
   8f000000

   [...]

   Starting kernel ...

   Uncompressing Linux... done, booting the kernel.
   Booting Linux on physical CPU 0xf00
   Linux version 3.12.0+ (xxx@rock) (gcc version 4.8.3 20131202 (prerelease) (crosstool-NG
   linaro-1.13.1-4.8-2013.12 - LinaroGCC 2013.11) ) #664 SMP Tue Jun 24 15:30:45 CST 2014
   CPU: ARMv7 Processor [410fc075] revision 5 (ARMv7), cr=30c73c7d
   CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
   Machine: Freescale Layerscape LS1021A, model: LS1021A TWR Board
   Memory policy: ECC disabled, Data cache writealloc
   PERCPU: Embedded 7 pages/cpu @8901c000 s7936 r8192 d12544 u32768
   Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 520720
   Kernel command line: root=/dev/ram rw console=ttyLP0,115200
   PID hash table entries: 4096 (order: 2, 16384 bytes)

   [...]

   ls1021atwr login: root
   root@ls1021atwr:~#
   ```

2. After the kernel boot up to the console, You can type any shell command in the LPUART TERMINAL.

7.2.8 PCI Express Interface Controller

7.2.8.1 PCIe Linux Driver

Module Loading

The MPC85xx/Layerscape PCIE host bridge support code is compiled into the kernel. It is not available as a module.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`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
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;
```

7.2.8.2 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.

LS1088A and LS1012A fail to recover from fatal errors.

7.2.8.3 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

7.2.9 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.
```

7.2.10 Queue Direct Memory Access Controller (qDMA)

The qDMA controller transfers blocks of data between one source and one destination. The blocks of data transferred can be represented in memory as contiguous or noncontiguous using scatter/gather table(s). Channel virtualization is supported through enqueuing of DMA jobs to, or dequeuing DMA jobs from, different work queues.

QDMA can support Layerscape platform with DPAA1 or DPAA2.

QDMA for platform with DPAA1

Kernel Configure Options

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

Kernel Configure Tree View Options	Description

`Device Drivers --->`
`[*] DMA Engine support ---> ---`
`>`
`<*> Freescale qDMA engine support`
	Support the Freescale qDMA engine with command queue and legacy mode.
	Channel virtualization is supported through enqueuing of DMA jobs to,
	or dequeuing DMA jobs from, different work queues.
	This module can be found on Freescale LS SoCs.

Identifier

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

Option	Values	Default Value	Description
CONFIG_FSL_QDMA	y/m/n	n	qDMA driver

Device Tree Binding

Device Tree Node

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

```
qdma: qdma@8380000 {
        compatible = "fsl,ls1046a-qdma", "fsl,ls1021a-qdma";
        reg = <0x0 0x8380000 0x0 0x1000>, /* Controller regs */
              <0x0 0x8390000 0x0 0x10000>, /* Status regs */
```

```
                              <0x0 0x83a0000 0x0 0x40000>; /* Block regs */
                        interrupts = <0 153 0x4>,
                                     <0 39 0x4>;
                        interrupt-names = "qdma-error", "qdma-queue";
                        channels = <8>;
                        queues = <2>;
                        status-sizes = <64>;
                        queue-sizes = <64 64>;
                        big-endian;
                };
```

Source File

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

Source File	Description
drivers/dma/fsl-qdma.c	The qDMA driver file

Verification in Linux

```
root@ls1043ardb:~# echo 1024 > /sys/module/dmatest/parameters/test_buf_size;
root@ls1043ardb:~# echo 4 > /sys/module/dmatest/parameters/threads_per_chan;
root@ls1043ardb:~# echo 2 > /sys/module/dmatest/parameters/max_channels;
root@ls1043ardb:~# echo 100 > /sys/module/dmatest/parameters/iterations;
root@ls1043ardb:~# echo 1 > /sys/module/dmatest/parameters/run

[   32.498138] dmatest: Started 4 threads using dma0chan0
[   32.503430] dmatest: Started 4 threads using dma0chan1
[   32.508939] dmatest: Started 4 threads using dma0chan2
[   32.520073] dmatest: dma0chan0-copy0: summary 100 tests, 0 failures 4904 iops 2452 KB/s (0)
[   32.520076] dmatest: dma0chan0-copy2: summary 100 tests, 0 failures 4923 iops 2461 KB/s (0)
[   32.520079] dmatest: dma0chan0-copy3: summary 100 tests, 0 failures 4928 iops 2661 KB/s (0)
[   32.520176] dmatest: dma0chan0-copy1: summary 100 tests, 0 failures 4892 iops 2446 KB/s (0)
[   32.526438] dmatest: dma0chan1-copy0: summary 100 tests, 0 failures 4666 iops 2240 KB/s (0)
[   32.526441] dmatest: dma0chan1-copy2: summary 100 tests, 0 failures 4675 iops 2291 KB/s (0)
[   32.526469] dmatest: dma0chan1-copy3: summary 100 tests, 0 failures 4674 iops 2197 KB/s (0)
[   32.529610] dmatest: dma0chan2-copy1: summary 100 tests, 0 failures 5168 iops 2791 KB/s (0)
[   32.529613] dmatest: dma0chan2-copy0: summary 100 tests, 0 failures 5164 iops 2478 KB/s (0)
[   32.529754] dmatest: dma0chan2-copy3: summary 100 tests, 0 failures 5215 iops 2555 KB/s (0)
[   32.529756] dmatest: dma0chan2-copy2: summary 100 tests, 0 failures 5211 iops 2709 KB/s (0)
[   32.537881] dmatest: dma0chan1-copy1: summary 100 tests, 0 failures 3044 iops 1461 KB/s (0)
(0)
dmatest: dma0chan0-copy3: summary 1000 tests, 0 failures 4078 iops 33474 KB/s (0)
dmatest: dma0chan0-copy0: summary 1000 tests, 0 failures 3024 iops 24486 KB/s (0)
dmatest: dma0chan0-copy2: summary 1000 tests, 0 failures 2881 iops 23588 KB/s (0)
```

QDMA for platform with DPAA1

Kernel Configure Options

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

Kernel Configure Tree View Options	Description

`Device Drivers --->`
`    [*] DMA Engine support  ---> --->`
`        <*>    NXP DPAA2 QDMA`
	NXP Data Path Acceleration
	Architecture 2 QDMA driver, using the NXP MC bus driver.

Identifier

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

Option	Values	Default Value	Description
CONFIG_FSL_DPAA2_QDMA	y/m/n	n	qDMA driver

Source Files

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

Source File	Description
drivers/dma/dpaa2-qdma/*	The qDMA driver file

Verification in Linux

```
Create DPDMAI object using restool:

restool dpdmai create --priorities=2,5
restool dprc assign dprc.1 --object=dpdmai.0 --plugged=1

Configure parameters for dmatest and run it:

echo 8 > /sys/module/dmatest/parameters/test_flag
echo 100 > /sys/module/dmatest/parameters/sg_size
echo 10000 > /sys/module/dmatest/parameters/iterations
echo 1 > /sys/module/dmatest/parameters/threads_per_chan
echo 8 > /sys/module/dmatest/parameters/max_channels
echo 64 > /sys/module/dmatest/parameters/test_buf_size
echo 1 > /sys/module/dmatest/parameters/run

Example log:

root@ls2085ardb:~# echo 8 > /sys/module/dmatest/parameters/test_flag
root@ls2085ardb:~# echo 10 > /sys/module/dmatest/parameters/iterations
root@ls2085ardb:~# echo 2 > /sys/module/dmatest/parameters/threads_per_chan
root@ls2085ardb:~# echo 32384 > /sys/module/dmatest/parameters/test_buf_size
root@ls2085ardb:~# echo 4 > /sys/module/dmatest/parameters/max_channels
root@ls2085ardb:~# echo 1 > /sys/module/dmatest/parameters/run
[   68.460353] dmatest: Started 2 threads using dma0chan0
[   68.465549] dmatest: Started 2 threads using dma0chan1
[   68.465755] dmatest: dma0chan0-sg0: summary 10 tests, 0 failures 1847 iops 422686 KB/s (0)
[   68.465963] dmatest: dma0chan0-sg1: summary 10 tests, 0 failures 1786 iops 367095 KB/s (0)
[   68.470694] dmatest: dma0chan1-sg0: summary 10 tests, 0 failures 1938 iops 608838 KB/s (0)
[   68.470987] dmatest: dma0chan1-sg1: summary 10 tests, 0 failures 1843 iops 517419 KB/s (0)
[   68.503858] dmatest: Started 2 threads using dma0chan2
[   68.509042] dmatest: Started 2 threads using dma0chan3
[   68.509255] dmatest: dma0chan2-sg0: summary 10 tests, 0 failures 1849 iops 549944 KB/s (0)
[   68.509454] dmatest: dma0chan2-sg1: summary 10 tests, 0 failures 1789 iops 473514 KB/s (0)
[   68.514518] dmatest: dma0chan3-sg1: summary 10 tests, 0 failures 1830 iops 414714 KB/s (0)
[   68.515016] dmatest: dma0chan3-sg0: summary 10 tests, 0 failures 1670 iops 512859 KB/s (0)
```

7.2.11 Real Time Clock (RTC)

Linux SDK for QorIQ Processors

Description

Provides the RTC function.

Kernel Configure Tree View Options

Kernel Configure 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 "dallas,ds3232"

Default node:

```
i2c@3000 {
            #address-cells = <1>;
            #size-cells = <0>;
            compatible = "fsl-i2c";
            reg = <0x3000 0x100>;
            interrupts = <43 2>;
            interrupt-parent = <&mpic>;
            dfsrr;
            rtc@68 {
                    compatible = "dallas,ds3232";
                    reg = <0x68>;
```

```
                };
            };
```

Verification in Linux

Here is the rtc booting 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)
      ...

NOTE: Please refer to the related DTS file to enable the RTC driver before building.
      For example, LS2080AQDS board, should enable the below option:
      <*> Dallas/Maxim DS3232
```

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
```

7.2.12 Synchronous Audio Interface (SAI)

Description

This document describes how to configure and test SAI audio driver for TWR-LS1021A. The integrated I2S module is NXP's Synchronous Audio Interface (SAI). The codec is SGTL5000 stereo audio codec.

RCW configuration

Refer to the below table for the RCW for Audio on the TWR-LS1021A.

Board	RCW
TWR-LS1021A	Bit 364, EC1_EXT_SAI2_TX = 1; Bit 365, EC1_EXT_SAI2_RX =1; Bit 366-367, EC1_BASE = 00

Kernel Configure Options Tree View

<table>
<tr><th>Kernel Configure Tree View Options</th><th>Description</th></tr>
<tr><td>

```
Device Drivers --->
<*> I2C support  --->
    [*] Enable compatibility bits for old user-space
    [*] I2C device interface
    [*] I2C bus multiplexing support
        Multiplexer I2C Chip support  --->
        <*> Philips PCA954x I2C Mux/switches
    [*] Autoselect pertinent helper modules
    I2C Hardware Bus support  --->
        <*> IMX I2C interface

<*> Voltage and Current Regulator Support --->
    [*] Regulator debug support
    [*] Provide a dummy regulator if regulator lookups fail
    [*] Fixed voltage regulator support

<*> Sound card support
    <*> Advanced Linux Sound Architecture ->
        [*] OSS PCM (digital audio) API
        [*]     OSS PCM (digital audio) API - Include plugin system
        [*] Support old ALSA API
        [*] Verbose procfs contents
            ALSA for SoC audio support  --->
            SoC Audio for Freescale CPUs  --->
            <*> Synchronous Audio Interface (SAI) module support
            CODEC drivers --->
            <*> Freescale SGTL5000 CODEC
            <*> ASoC Simple sound card support
<*> DMA Engine support   --->
    <*> Freescale eDMA engine support support
```

</td><td>Enable ALSA SOC driver, I2C driver and EDMA driver.</td></tr>
</table>

Identifier

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

Option	Values	Default Value	Description
CONFIG_I2C_IMX	y/m/n	y	I2C driver needed for configuring SGTL5000
CONFIG_SOUND	y/m/n	y	Enable sound card support
CONFIG_SND	y/m/n	y	Enable advanced Linux sound architecture supports
CONFIG_SND_PCM_OSS	y/m/n	y	Enable OSS digital audio
CONFIG_SND_PCM_OSS_PLUGINS	y/m/n	y	Support conversion of channels, formats and rates
CONFIG_SND_SUPPORT_OLD_API	y/m/n	y	Enable support old ALSA API
CONFIG_SND_SOC_FSL_SAI	y/m/n	y	Enable SAI module support

Table continues on the next page...

Table continued from the previous page...

CONFIG_SND_SOC_GENERIC_DMAENGINE_PCM	y/m/n	y	Enable generic dma engine for PCM
CONFIG_SND_SIMPLE_CARD	y/m/n	y	Enable generic simple sound card support
CONFIG_SND_SOC_SGTL5000	y/m/n	y	Enable codec sgtl5000 support
CONFIG_FSL_EMDA	y/m/n	y	Enable EDMA engine support

Source Files

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

Source File	Description
sound/soc/fsl	ALSA SOC driver source

Verification in Linux

1. The following messages will be shown in the kernel boot process:

```
sgtl5000 5-000a: sgtl5000 revision 0x11
sgtl5000 5-000a: Using internal LDO instead of VDDD
......
asoc-simple-card sound:  sgtl5000 <-> 2b60000.sai mapping ok
......
ALSA device list:
  #0: 2b60000.sai-sgtl5000
```

2. If the device nodes don't already exist, create directory /dev/snd/, and create device nodes with the following commands in /dev/snd/ directory.

```
mknod controlC0 c 116 0
mknod pcmC0D0c c 116 24
mknod pcmC0D0p c 116 16
```

3. On TWR-LS1021A, the LineOut interface is J8 and the LineIn interface is J13

4. Run the following aplay commands to test playback. Run the following arecord command to test record.

```
aplay -f S16_LE -r 44100 -t wav -c 2 44k-16bit-stereo.wav

arecord -d 10 -f S16_LE -r 44100 -t wav -c 2 44k-16bit-stereo-10s.wav
aplay -f S16_LE -r 44100 -t wav -c 2 44k-16bit-stereo-10s.wav
```

5. Use alsamixer to adjust the volume for playing by the option “PCM” and recording gain by the option "Mic" . Use alsamixer to choose LINE IN or MIC.

7.2.13 Serial Advanced Technology Attachment (SATA)

Description

The driver supports NXP native SATA controller.

Module Loading

SATA driver supports either kernel built-in or module.

Kernel Configure Tree View Options	Description
Device Drivers---> <*> Serial ATA and Parallel ATA drivers ---> --- Serial ATA and Parallel ATA drivers <*> AHCI SATA support <*> Freescale QorIQ AHCI SATA support	Enables SATA controller support on ARM-based SoCs

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_SATA_AHCI=y	y/m/n	y	Enables SATA controller
CONFIG_SATA_AHCI_QORIQ=y	y/m/n	y	Enables SATA controller

Source Files

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

Source File	Description
drivers/ata/ahci_qoriq.c	Platform AHCI SATA support

Test Procedure

```
Please follow the following steps to use USB in Simics
(1) Boot up the kernel
...
fsl-sata ffe18000.sata: Sata FSL Platform/CSB Driver init
scsi0 : sata_fsl
ata1: SATA max UDMA/133 irq 74
fsl-sata ffe19000.sata: Sata FSL Platform/CSB Driver init
scsi1 : sata_fsl
ata2: SATA max UDMA/133 irq 41
...
(2) The disk will be found by kernel.
...
ata1: Signature Update detected @ 504 msecs
ata2: No Device OR PHYRDY change,Hstatus = 0xa0000000
ata2: SATA link down (SStatus 0 SControl 300)
ata1: SATA link up 1.5 Gbps (SStatus 113 SControl 300)
ata1.00: ATA-8: WDC WD1600AAJS-22WAA0, 58.01D58, max UDMA/133
ata1.00: 312581808 sectors, multi 0: LBA48 NCQ (depth 16/32)
ata1.00: configured for UDMA/133
scsi 0:0:0:0: Direct-Access     ATA      WDC WD1600AAJS-2 58.0 PQ: 0 ANSI: 5
sd 0:0:0:0: [sda] 312581808 512-byte logical blocks: (160 GB/149 GiB)
sd 0:0:0:0: Attached scsi generic sg0 type 0
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't support DPO
 or FUA
 sda: sda1 sda2 sda3 sda4 < sda5 sda6 >
sd 0:0:0:0: [sda] Attached SCSI disk

(3)play with the disk according to the following log.
[root@ls1046 root]# fdisk -l /dev/sda
Disk /dev/sda: 160.0 GB, 160041885696 bytes
255 heads, 63 sectors/track, 19457 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes
```

```
    Device Boot      Start         End      Blocks  Id System
/dev/sda1               1         237     1903671  83 Linux
/dev/sda2             238         480     1951897+ 82 Linux swap
/dev/sda3             481        9852    75280590  83 Linux
/dev/sda4            9853       19457    77152162+  f Win95 Ext'd (LBA)
/dev/sda5            9853       14655    38580066  83 Linux
/dev/sda6           14656       19457    38572033+ 83 Linux
[root@ls1046 root]#
[root@ls1046 root]# mke2fs /dev/sda1
mke2fs 1.41.4 (27-Jan-2009)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
65280 inodes, 261048 blocks
13052 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=268435456
8 block groups
32768 blocks per group, 32768 fragments per group
8160 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 22 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
[root@ls1046 root]#
[root@ls1046 root]# mkdir sata
[root@ls1046 root]# mount /dev/sda1 sata
[root@ls1046 root]# ls sata/
lost+found
[root@ls1046 root]# cp /bin/busybox sata/
[root@ls1046 root]# umount sata/
[root@ls1046 root]# mount /dev/sda1 sata/
[root@ls1046 root]# ls sata/
busybox     lost+found
[root@ls1046 root]# umount sata/
[root@ls1046 root]# mount /dev/sda3 /mnt
[root@ls1046 root]# df
Filesystem           1K-blocks      Used Available Use% Mounted on
rootfs               852019676 794801552  13937948  99% /
/dev/root            852019676 794801552  13937948  99% /
tmpfs                  1036480        52   1036428   1% /dev
shm                    1036480         0   1036480   0% /dev/shm
/dev/sda3             74098076   4033092  66300956   6% /mnt
```

Known Bugs, Limitations, or Technical Issues

- CDROM is not supported due to the silicon limitation

7.2.14 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 Primary Protected Application (PPA) 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
  ```

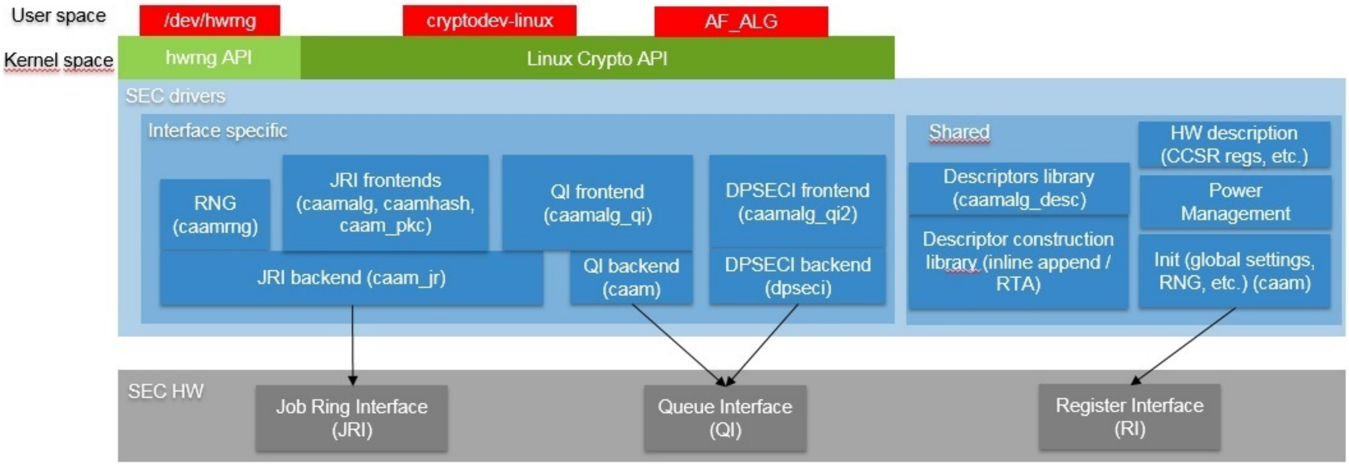


Figure 60. Linux kernel - SEC device drivers overview

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
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 367 for the list of module names and section Kernel Configuration on page 368 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:

Kernel Configure Tree View Options	Description

`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`
`<*>       Register public key cryptography implementations with Crypto API`
`<*>       Register caam device for hwrng API`
`<M>   QorIQ DPAA2 CAAM (DPSECI) driver`
	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* - 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

`Networking support --->`
`Network option --->`
`<*> TCP/IP networking`
`<*>   IP: AH transformation`
`<*>   IP: ESP transformation`
`<*>   IP: IPsec transport mode`
`<*>   IP: IPsec tunnel mode`
	For IPsec support the TCP/IP networking option and corresponding sub-options should be enabled.

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 84. 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)
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)

Table continues on the next page...

Table 84. Algorithms supported by each interface / backend (continued)

Algorithm name / Backend	Job Ring Interface	Queue Interface	DPSEC Interface
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)
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

Table continues on the next page...

Table 84. Algorithms supported by each interface / backend (continued)

Algorithm name / Backend	Job Ring Interface	Queue Interface	DPSEC Interface
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
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)

7.2.15 Time Division Multiplexing (TDM)

Description

Time Division Multiplexing (TDM) is a type of digital or analog multiplexing in which two or more signals or bit streams are transferred apparently simultaneously as sub-channels in one communication channel, but are physically taking turns on the channel. The time domain is divided into several recurrent timeslots of fixed length, one for each sub-channel. A sample byte or data block of sub-channel 1 is transmitted during timeslot 1, sub-channel 2 during timeslot 2, etc. One TDM frame consists of one timeslot per sub-channel. After the last sub-channel the cycle starts all over again with a new frame, starting with the second sample, byte or data block from sub-channel 1, etc.

TDM or Time Division Multiplexing is an essential component to run VoIP applications on NXP Platforms. Its function is to receive and send time division multiplexed voice samples on the physical TDM lines.

This document explains the procedure to test the TDM on FSL MPC85xx platforms.

The test procedure shows the method to run a small TDM demo application which transfers voice from one TDM channel to the other.

The overall TDM software stack and the data flow is depicted below. On the top is a generic TDM framework layer which can ideally integrate with any TDM driver beneath it.

Generally NXP platforms offer two types of TDM interfaces:

1. NXP TDM
2. QE based TDM

This manual specifically talks about NXP TDM

U-Boot Configuration

Compile time options

Please check the platform specific document to check if any specific u-boot configuration is required for TDM feature.

Also please ensure if there is any requirement from pin mux perspective to enable TDM.

Runtime options

Refer to platform specific document for any specific hwconfig or environment variables which may be required for TDM functionality.

Also the FXS ports location will be mentioned in the platform specific document.

Kernel Configure Options

Tree View

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

Kernel Configure Tree view options	Description

`Device Drivers  --->`
`      <*> TDM support  --->`
`                --- TDM support`
`                [ ] TDM Core debugging messages (NEW)`
`                <M> TDM test Module`
	Enable TDM Framework
	Enable TDM test as Module
	Enable TDM driver
	Enable SLIC driver

Kernel Configure Tree view options	Description

`TDM Device support --->`
`<*> Driver for Freescale TDM controller`
`Line Control Devices --->`
`<*> Zarlink Slic intialization Module`	

Identifier

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

Option	Value	Default value in BSP	Description
CONFIG_TDM	y/n/m	N	Enable / Disable TDM Framework support
CONFIG_TDM_FSL	y/n/m	N	Enable / Disable TDM driver, depends on TDM framework and CONFIG_FSL_SOC
CONFIG_SLIC_ZARLINK	y/n/m	N	Enable / Disable SLIC driver , depends on TDM driver and TDM framework, and CONFIG_FSL_ESPI
CONFIG_TDM_TEST	y/n/m	N	Enable / disable TDM test module

Device Tree Binding

Below is the definition of the device tree node required by this feature

TDM device dts entries.(as many entries as the number of TDM controllers on the platform)

Property	Type	Status	Description
compatible = "fsl,tdm1.0";	<string>		Should contain "fsl,tdm1.0"
reg = <0x16000 0x200 0x2c000 0x2000>;	<tdm-reg-offset tdm-reg-size dmac-reg-offset dmac-reg-size>		Offset and length of the register set for the NXP TDM and TDM-DMAC
interrupts = <16 8 62 8>;	<tdm-err-intr tdm-err-intr-type dmac-intr dmac-intr-type>		Defines two interrupt specifiers namely interrupt + number and interrupt type for TDM error and TDM DMAC
fsl-max-time-slots = <128>	<u32>		Maximum number of 8-bit time slots in one TDM frame that hardware supports.

SLIC device dts entries (As many entries as the number of SLICs on the platform)

Please note that the below mentioned SLIC entry is for the Legerity SLIC which is connected to the chip through SPI interface.

Property	Type	Status	Description
compatible = "zarlink,le88266";			Should be "zarlink,le88266"
reg = <1>;			Chip select number of the SPI bus SLIC is connected to
spi-max-frequency =<8000000>;			The maximum frequency the SLIC can operate at.

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

```
       tdm@16000 {
               compatible = "fsl,tdm1.0";
               reg = <0x16000 0x200 0x2c000 0x2000>;
               clock-frequency = <0>;
               interrupts = <16 8 62 8>;
               phy-handle = <zarlink1>
               fsl-max-time-slots = <128>
       };


spi@7000 {
    cell-index = <0>;
    #address-cells = <1>;
    #size-cells = <0>;
    compatible = "fsl,espi";
    reg = <0x7000 0x1000>;
    interrupts = <59 0x2>;
    interrupt-parent = <&mpic>;
    mode = "cpu";
            ………….
            …………..
            …………..
               legerity@0{
                       compatible = "zarlink,le88266";
                       reg = <1>;
                       spi-max-frequency = <8000000>;
               };

               legerity@1{
                       compatible = "zarlink,le88266";
                       reg = <2>;
                       spi-max-frequency = <8000000>;
               };
};
```

Source Files

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

Source file	Purpose
include/linux/tdm.h	Header file for TDM framework
drivers/tdm/tdm-core.c	Source file for TDM framework
drivers/tdm/device/tdm_fsl.h	Header file for TDM driver
drivers/tdm/device/tdm_fsl.c	Source file for TDM driver
drivers/tdm/line_ctrl/slic_zarlink.h	Header file for SLIC driver
drivers/tdm/line_ctrl/slic_zarlink.c	Source file for SLIC driver
drivers/tdm/test/tdm_test.c	Source file for TDM test module

Verification in U-Boot

N/A

Verification in Linux

1. Attach two analog phones at the two FXS ports of the board. (Incase there are two SLIC devices there would be 4 FXS ports available).

NOTE

Please refer to the platform documentation for specific information on FXS ports.

2. Bring up the platform with the kernel image and dts configured as explained above.

   Look for the below mentioned messages in the kernel boot log.

   This will ensure TDM and SLIC initialization.

```
...
...

EDAC MC: Ver: 2.1.0
fsl_tdm: Freescale TDM Driver Adapter:Init
adapter [fsl_tdm] registered
SLIC: Freescale DEVELOPED ZARLINK SLIC DRIVER
####################################################
# This driver was created solely by Freescale,     #
# without the assistance, support or intellectual  #
# property of Zarlink Semiconductor. No            #
# maintenance or support will be provided by       #
# Zarlink  Semiconductor regarding this driver.    #
####################################################
SLIC probed!
SLIC config success
SLIC: product code 1 read is  4
SLIC: product code 2 read is  b3
SLIC: config read is  ff
SLIC: config read is  8a
DEV reg is 82
DEV reg after is 2
Mask reg before setting is 3f bf
Mask reg after setting is f6 f6
Read Tx Timeslot for CH1 is 0
Read Tx Timeslot for CH2 is 2
Read Rx Timeslot for CH1 is 0
Read Rx Timeslot for CH2 is 2
Operating Fun for channel 1 is 82
Cadence Timer Reg for CH1 before is 7 ff0 0
Cadence Timer Reg for CH1 after is 1 903 20
Switching control for channel 1 is 20
Operating Fun for channel 2 is a0
Cadence Timer Reg for CH2 before is 7 ff0 0
Cadence Timer Reg for CH2 after is 1 903 20
Switching control for channel 2 is 20
SLIC 1 configuration success
TDM_TEST: Test Module for Freescale Platforms with TDM support
TDM_TEST module installed
...
...
```

3. Check /proc/device-tree/soc for tdm and slic nodes.
4. Run cat /proc/interrupts to check for TDM interrupts. Following is an example log details may vary over different platforms.

```
[root@ /root]# insmod tdm_test.ko
TDM_TEST: Test Module for Freescale Platforms with TDM support
TDM Driver(ID=1)is attached with Adapterfsl_tdm(ID = 0) drv_count=1
TDM_TEST module installed
[root@ /root]# cat /proc/interrupts
           CPU0
 20:          0   OpenPIC   Level     fsldma-chan
 21:          0   OpenPIC   Level     fsldma-chan
 22:          0   OpenPIC   Level     fsldma-chan
 23:          0   OpenPIC   Level     fsldma-chan
 28:          0   OpenPIC   Level     ehci_hcd:usb1
 42:         57   OpenPIC   Level     serial
 43:          0   OpenPIC   Level     i2c-mpc, i2c-mpc
 59:          0   OpenPIC   Level     fsl_espi
 62:        993   OpenPIC   Edge      dmac_done_isr
LOC:        698   Local timer interrupts
SPU:          0   Spurious interrupts
CNT:          0   Performance monitoring interrupts
MCE:          0   Machine check exceptions
```

5. To test the TDM functionality Pick up both the phones. Anything spoken on one phone will be heard on the other.

Benchmarking

Voice must be clearly audible and must not break.

Known Bugs, Limitations, or Technical Issues

1. TDM functionality is not supported in 36bit Physical address mode. This is because of hardware limitation on current FSL platforms.
2. TDM_TEST is for demo purpose only and hence runs only for a small duration.

7.2.16 Universal Serial Bus Interfaces

7.2.16.1 USB 2.0 Host Driver User Manual

Description

The driver supports USB controller in host mode

Module Loading

The USB Host driver in linux supports either kernel built-in or module driver. U-boot USB driver is always built-in

U-Boot Compile Time Configuration Options

U-Boot Configure Options	Description

`#define CONFIG_HAS_FSL_DR_USB`
 	
`#ifdef CONFIG_HAS_FSL_DR_USB`
`#define CONFIG_USB_EHCI`
 	
`#ifdef CONFIG_USB_EHCI`
`#define CONFIG_CMD_USB`
`#define CONFIG_USB_STORAGE`
`#define CONFIG_USB_EHCI_FSL`
`#define CONFIG_EHCI_HCD_INIT_AFTER_RESET`
`#define CONFIG_CMD_EXT2`
	Enables USB host Dual Role controller support. Defined inside platform config file: include/configs/<platform.h>.

`CONFIG_SYS_FSL_USB_INTERNAL_UTMI_PHY`	Enable internal UTMI Phy support. Required only for SoCs having internal UTMI PHY. Defined inside file: arch/powerpc/include/asm/config_mpc85xx.h
`CONFIG_USB_MAX_CONTROLLER_COUNT`	Tell maximum no. of USB controllers in this SoC. Defined inside file: arch/powerpc/include/asm/config_mpc85xx.h

NOTE

For LS1012A, due to the USB2.0 controller and the Ethernet controller being pin muxed, you need to remove `CONFIG_FSL_PPFE` from the `include/configs/ls1012a_common.h` directory if it exists. In the `include/configs/ls1012aqds.h` directory, enable `CONFIG_HAS_FSL_XHCI_USB` and remove `CONFIG_HAS_FSL_XHCI_USB` in the U-Boot source code.

U-Boot Source Files

The driver source is maintained in the U-boot source in following files

Source File	Description
drivers/usb/host/ehci-fsl.c	EHCI FSL USB host controller driver
common/cmd_usb.c	Common usb command file
drivers/usb/host/ehci-hcd.c	EHCI USB host controller driver

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`Device Drivers--->`
`USB support --->`
`[*] Support for Host-side USB`
	Enables USB host controller support

`Device Drivers--->`
`USB support --->`
`<*> EHCI HCD (USB 2.0) support`
`-*- Root Hub Transaction Translators`
`[ ] Improved Transaction Translator scheduling (EXPERIMENTAL)`
`[*] Support for Freescale on-chip EHCI USB controller`
`[*] EHCI support for PPC USB controller on OF platform bus`
`<*> OHCI HCD support`
`[*] OHCI support for PPC USB controller on OF platform bus`
`[*] Support big endian HC`
`[*] Support little endian HC`
`[*] OHCI support for PCI-bus USB controllers`
	Enables EHCI Host Controller Driver and transaction translator.

NOTE

For LS1012A, due to the USB2.0 controller and the Ethernet controller being pin muxed, you need to disable `CONFIG_FSL_PPFE` in Linux kernel `.config` file.

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_USB	y/m/n	y	Enables USB host controller
CONFIG_USB_EHCI_HCD	y/m/n	y	Enables EHCI HCD
CONFIG_USB_EHCI_ROOT_HUB_TT	y/n	y	Enables EHCI to support USB1.1 device
CONFIG_USB_OHCI_HCD	y/m/n	y	Enables OHCI HCD

Kernel Source Files

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

Source File	Description
drivers/usb/host/ehci-fsl.c	EHCI USB host controller driver
arch/powerpc/sysdev/fsl_soc.c	Hook between OF tree and platform device
drivers/usb/host/ohci_hcd.c	OHCI USB host controller driver

Device Tree Binding

```
usb@22000 {

 #address-cells = <1>;

 #size-cells = <0>;

 compatible = "fsl-usb2-<controller-type>-v<controller version>",

              "fsl-usb2-<controller-type>";

 reg = <0x22000 0x1000>;

 interrupt-parent = <&mpic>;

 interrupts = <28 0x2>;

 phy_type = "ulpi";                         /* ulpi/utmi/utmi_dual */

 dr_mode = "host"                            /* host, peripheral */

 };
```

NOTE

controller-type: dr(dual-role), mph(multi-port-host) controller-version: 1.6, 2.2, or earlier default mode is always host

Verification in U-Boot

U-boot environment to specify usb phy and usb mode type

```
=> setenv hwconfig 'usb<controller-no>:dr_mode=<mode>,phy_type=<phy_type>;<next usb 
controller>'

For example:
For socs having single usb controller and ULPI phy
```

```
=> setenv hwconfig 'usb1:dr_mode=host,phy_type=ulpi'

For socs having single usb controller and UTMI phy
=> setenv hwconfig 'usb1:dr_mode=host,phy_type=utmi'

For socs having two usb controllers and ULPI phys only
=> setenv hwconfig 'usb1:dr_mode=host,phy_type=ulpi;usb2:dr_mode=host,phy_type=ulpi'
```

Then use usb start to start the usb device

```
=> usb start

(Re)start USB...

USB: Register 10011 NbrPorts 1

USB EHCI 1.00

scanning bus for devices... 2 USB Device(s) found

 scanning bus for storage devices... 1 Storage Device(s) found

=> usb dev



USB device 0: Vendor: SanDisk Rev: 8.02 Prod: Cruzer Colors+

 Type: Removable Hard Disk

 Capacity: 7663.9 MB = 7.4 GB (15695871 x 512)

=> usb tree

Device Tree:
  1  Hub (480 Mb/s, 0mA)
  |  u-boot EHCI Host Controller
  |
  |+-2  Mass Storage (480 Mb/s, 500mA)
       JetFlash Mass Storage Device 63ZOA56O8TZFZ0AC

=> usb info
1: Hub,  USB Revision 2.0
 - u-boot EHCI Host Controller
 - Class: Hub
 - PacketSize: 64  Configurations: 1
 - Vendor: 0x0000  Product 0x0000 Version 1.0
   Configuration: 1
   - Interfaces: 1 Self Powered 0mA
     Interface: 0
     - Alternate Setting 0, Endpoints: 1
     - Class Hub
     - Endpoint 1 In Interrupt MaxPacket 2048 Interval 255ms

2: Mass Storage,  USB Revision 2.0
 - JetFlash Mass Storage Device 63ZOA56O8TZFZ0AC
 - Class: (from Interface) Mass Storage
 - PacketSize: 64  Configurations: 1
 - Vendor: 0x8564  Product 0x1000 Version 17.0
   Configuration: 1
   - Interfaces: 1 Bus Powered 500mA
     Interface: 0
     - Alternate Setting 0, Endpoints: 2
     - Class Mass Storage, Transp. SCSI, Bulk only
     - Endpoint 1 In Bulk MaxPacket 512
     - Endpoint 2 Out Bulk MaxPacket 512
```

```
=> md 2000000
02000000: 02992004 02060002 08462cc0 84990329    .. ......F,....)
02000010: 00c48e24 82181008 06501810 01a80004    ...$.....P......
02000020: 083d3881 00808270 40a00000 b012a502    .=8....p@.......
02000030: d4000088 28840b45 80028200 40244400    ....(..E....@$D.
02000040: 022b1004 04842482 20610b81 0494d020    .+....$. a.....
02000050: 8012b628 08200100 010c6300 0411b880    ...(. ....c.....
02000060: 42400200 8004a4c8 29802818 904000c0    B@......).(..@..
02000070: 08210200 2040a8c0 448aae00 a0000000    .!.. @..D.......
02000080: 2800c800 04b62080 60199885 02a62324    (..... .`.....#$
02000090: 04870a08 a0008000 18020003 0281a232    ...............2
020000a0: 50414020 4000850b 02044c00 10013018    PA@ @.....L...0.
020000b0: 00208810 000c2280 081805a8 88800010    . ....".........
020000c0: 000c020a 0012b024 01282c02 00808181    .......$.(,.....
020000d0: 00010824 0160b602 81621008 00828082    ...$.`...b......
020000e0: 38d0f028 42010e03 1d242290 02000120    8..(B....$"....
020000f0: 6c217230 00920800 20200d40 41c08011    l!r0....  .@A...
=> mw 2000000 ffffaaaa 100
=> md 2000000
02000000: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000010: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000020: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000030: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000040: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000050: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000060: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000070: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000080: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
02000090: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
020000a0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
020000b0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
020000c0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
020000d0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
020000e0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
020000f0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
=> usb write 2000000 0 100

USB write: device 0 block # 0, count 256 ... 256 blocks write: OK
=> md 1000000
01000000: fbf3eae6 2feeffbf dbf7775d ff5bebf7    ..../.....w].[..
01000010: abefefaf 7dbb3e3b bfffb5bb bfb86a7f    ....}.>;......j.
01000020: eff7b68f deaadfff eebf7bff bd7fed1f    ..........{.....
01000030: ffef7deb e7bfbbef dfeff7df 7f3ffcba    ..}..........?..
01000040: ab3bfbfe dfdee69b ffe18fd5 ff3e777f    .;...........>w.
01000050: da7effef bfabff7f f58ef768 9ffffeff    .~.........h....
01000060: cfebf8b0 f1b7dfef e9eefbfe f37bbadb    .............{..
01000070: b7f34ea2 da7efbff dfdff7ff f7effde3    ..N..~..........
01000080: fffbebff fe56ff5d 6ffd7ffd ff87efdf    .....V.]o.......
01000090: b6bfafac ddebfbfb ffacebfd f87bff9f    .............{..
010000a0: ffebffff ff7e7ff9 aefefd7f 5f7f5ebf    .....~......_.^.
010000b0: 6fe87e7b fabfdbcf d3faefad 6fbb5e7a    o.~{........o.^z
010000c0: f6af86de ffdb7bbf ff5ff6ba bfa4bfdf    ......{.._......
010000d0: ffbfa87f ffe67fcd efeffb9a 9b7e6a6f    .............~jo
010000e0: fffcf76f efbfeebb ffaceab1 5cfbfffe    ...o........\...
010000f0: edebffde e29fefff deaeafdb f97bdff5    .............{..
=> usb read 1000000 0 100

USB read: device 0 block # 0, count 256 ... 256 blocks read: OK
=> md 1000000
01000000: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000010: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000020: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000030: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000040: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000050: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000060: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000070: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000080: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
01000090: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
```

```
010000a0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
010000b0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
010000c0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
010000d0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
010000e0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
010000f0: ffffaaaa ffffaaaa ffffaaaa ffffaaaa    ................
=>
```

Verification in Linux

· Kernel configuration for USB memory stick support

```
 * Device Drivers---> SCSI Device Support---> SCSI Device Support ---> <*> SCSI disk support

* Device Drivers---> SCSI Device Support---> SCSI Device Support ---> <*> SCSI generic
support

 * Device Drivers---> USB Support---> [*] USB Mass Storage Support

 (The user can enable it either in kernel mode (set option as True) or as module (set option
as Module)).

* File Systems---> DOS/FAT/NT Filesystems---> <*> MSDOS fs support

* File Systems---> DOS/FAT/NT Filesystems---> <*> VFAT(Window-95)fs support

 * File Systems---> DOS/FAT/NT Filesystems---> VFAT(Window-95)fs support---> Default
codepage for FAT - 437

 * File Systems---> DOS/FAT/NT Filesystems---> VFAT(Window-95)fs support---> Default
iocharset for FAT - "iso8859-1"

 * File Systems---> Partition Types---> [*] Advanced partition selection

 * File Systems---> Partition Types---> [*] PC BIOS (MSDOS partition tables) support

 * File Systems---> Native Language Support---> Base native language support---> (iso8859-1)
Default NLS Option

* File Systems---> Native Language Support---> Base native language support---> <*> Codepage
437 (United States, Canada)

* File Systems---> Native Language Support---> Base native language support---> <*> NLS ISO
8859-1 (Latin 1; Western European Languages)
```

· plug in memory stick

```
~ # usb 1-1: new high speed USB device using fsl-ehci and address 2

usb 1-1: configuration #1 chosen from 1 choice

scsi6 : SCSI emulation for USB Mass Storage devices

scsi 6:0:0:0: Direct-Access SanDisk Cruzer 7.01 PQ: 0 ANSI: 0 CCS

sd 6:0:0:0: [sda] 3907583 512-byte hardware sectors: (2.00 GB/1.86 GiB)

sd 6:0:0:0: [sda] Write Protect is off

sd 6:0:0:0: [sda] Assuming drive cache: write through

sd 6:0:0:0: [sda] 3907583 512-byte hardware sectors: (2.00 GB/1.86 GiB)

sd 6:0:0:0: [sda] Write Protect is off

sd 6:0:0:0: [sda] Assuming drive cache: write through
```

```
 sda: sda1

sd 6:0:0:0: [sda] Attached SCSI removable disk

sd 6:0:0:0: Attached scsi generic sg0 type 0

scsi 6:0:0:1: CD-ROM SanDisk Cruzer 7.01 PQ: 0 ANSI: 0

sr0: scsi3-mmc drive: 48x/48x tray

Uniform CD-ROM driver Revision: 3.20

sr 6:0:0:1: Attached scsi generic sg1 type 5

~ # fdisk -l


Disk /dev/sda: 2000 MB, 2000682496 bytes

64 heads, 63 sectors/track, 969 cylinders

Units = cylinders of 4032 * 512 = 2064384 bytes


 Device Boot Start End Blocks Id System

/dev/sda1 1 969 1953439+ b Win95 FAT32

~ # mount -t vfat /dev/sda1 /mnt/cdrom/

~ # cd /mnt/cdrom/

/mnt/cdrom # ls

/mnt/cdrom # cp /usr/sbin/wd_keepalive .

/mnt/cdrom # ls

wd_keepalive

/mnt/cdrom # cd ..

/mnt # umount /mnt/cdrom/
```

```
===================================================================================
To create ext2 file-system on USB flash drive, follow below steps:
# fdisk /dev/sda
Device contains neither a valid DOS partition table, nor Sun, SGI or OSF disklab
el
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that the previous content
won't be recoverable.


Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-1011, default 1): Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-1011, default 1011): Using default
 value 1011

Command (m for help): w
The partition table has been altered!
```

```
Calling ioctl() to re-read partition table
sd 2:0:0:0: [sda] Test WP failed, assume Write Enabled
sd 2:0:0:0: [sda] Assuming drive cache: write through
 sda: sda1
[root@p5020 root]# mke2fs /dev/sda1
mke2fs 1.41.4
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
65536 inodes, 262094 blocks
13104 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=268435456
8 block groups
32768 blocks per group, 32768 fragments per group
8192 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 38 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
[root@p5020 /root]# mount /dev/sda1 /mnt
```

· Kernel configuration for USB Human Input Devices support

```
 * Device Drivers--->

        [*] HID Devices --->

        -*- Generic HID support

        <*> USB Human Interface Device (full HID) support

        Input device support --->

        -*- Generic input layer (needed for keyboard, mouse, ...)

        <*> Mouse interface

        [*] Provide legacy /dev/psaux device

        (1024) Horizontal screen resolution

        (768) Vertical screen resolution

        [*] Keyboards --->

        <*> AT keyboard

        [*] Mice --->

        <*> PS/2 mouse

        [*] ALPS PS/2 mouse protocol extension

        [*] Logitech PS/2++ mouse protocol extension

        [*] Synaptics PS/2 mouse protocol extension

        [*] IBM Trackpoint PS/2 mouse protocol extension
```

· plug in USB keyboard

```
~ # usb 1-1: new full speed USB device using fsl-ehci and address 3
```

```
usb 1-1: configuration #1 chosen from 1 choice

hub 1-1:1.0: USB hub found

hub 1-1:1.0: 3 ports detected

usb 1-1.1: new full speed USB device using fsl-ehci and address 4

usb 1-1.1: configuration #1 chosen from 1 choice

input: Dell Dell USB Keyboard Hub as /class/input/input1

generic-usb 0003:413C:2002.0002: input: USB HID v1.10 Keyboard [Dell Dell USB Ke yboard Hub]
on usb-fsl-ehci.0-1.1/input0

input: Dell Dell USB Keyboard Hub as /class/input/input2

generic-usb 0003:413C:2002.0003: input: USB HID v1.10 Device [Dell Dell USB Keyb
```

· plug in USB mouse

```
~ # usb 1-1: new low speed USB device using fsl-ehci and address 2

usb 1-1: configuration #1 chosen from 1 choice

input: HID 413c:3010 as /class/input/input0

generic-usb 0003:413C:3010.0001: input: USB HID v1.00 Mouse [HID 413c:3010] on u
```

Power Management

Following Pwr. Mgmt. features are supported:

1) Sleep

2) Deep Sleep

Pwr. Mgmt. Kernel Configuration Option(s)

Kernel Configure Tree View Options	Description

`Kernel options--->`
`[*] Suspend to RAM and standby`
`[*] Hibernation (aka 'suspend to disk')`
`() Default resume partition (NEW)`
	Enable Power Management feature

`Platform support -->`
`CPU Frequency scaling -->`
`[*] CPU Frequency scaling`
`<*> CPU frequency translation statistics`
`Default CPUFreq governor (userspace) -->`
`-*- 'userspace' governor for userspace frequency scaling`
`CPU Frequency drivers -->`
`[*] Support for Freescale MPC85xx CPU freq`
	Enable the CPU frequency driver

Verification in Linux

Sleep Capability

A system can be put into Suspend state, and can also be Resumed (woken-up) by USB. For this the following needs to be done:

1. Enable USB remote wake-up capability before putting the system into Suspend state

```
~ # echo enabled >/sys/bus/usb/devices/usb1/power/wakeup
```

2. Insert/Remove a USB flash drive into USB port after the system is put into SUSPEND state. This will bring the system out of the SUSPEND state

```
# echo standby > /sys/power/state
PM: Syncing filesystems ... done.
Freezing user space processes ... (elapsed 0.01 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.01 seconds) done.
Suspending console(s) (use no_console_suspend to debug)
sd 0:0:0:0: [sda] Synchronizing SCSI cache
sd 0:0:0:0: [sda] Stopping disk
PM: suspend of devices complete after 519.108 msecs
PM: late suspend of devices complete after 0.489 msecs
PM: noirq suspend of devices complete after 0.555 msecs
Disabling non-boot CPUs ...

USB Flash drive inserted --->

Enabling non-boot CPUs ...
CPU1 is up
PM: noirq resume of devices complete after 0.513 msecs
PM: early resume of devices complete after 0.349 msecs
fsl-lbc ffe05000.localbus: Chip select error: LTESR 0x00080000
/pcie@ffe09000: PCICSRBAR @ 0xfff00000
/pcie@ffe0a000: PCICSRBAR @ 0x0
/pcie@ffe0a000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the memory
map could reduce unnecessary bounce buffering.
/pcie@ffe0a000: DMA window size is 0x0
/pcie@ffe0b000: PCICSRBAR @ 0x0
/pcie@ffe0b000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the memory
map could reduce unnecessary bounce buffering.
/pcie@ffe0b000: DMA window size is 0x0
pci 0000:00:00.0: enabling device (0106 -> 0107)
pci 0001:02:00.0: enabling device (0106 -> 0107)
pci 0002:04:00.0: enabling device (0106 -> 0107)
ata2: No Device OR PHYRDY change,Hstatus = 0xa0000000
ata2: SATA link down (SStatus 0 SControl 300)
ata1: Signature Update detected @ 504 msecs
ata1: SATA link up 3.0 Gbps (SStatus 123 SControl 300)
ata1.00: configured for UDMA/133
sd 0:0:0:0: [sda] Starting disk
PM: resume of devices complete after 5419.653 msecs
Restarting tasks ... done.
root@p1022ds:~# usb 1-1: new high-speed USB device number 2 using fsl-ehci
scsi2 : usb-storage 1-1:1.0
scsi 2:0:0:0: Direct-Access     SRT      USB              1100 PQ: 0 ANSI: 4
sd 2:0:0:0: Attached scsi generic sg1 type 0
sd 2:0:0:0: [sdb] 15568896 512-byte logical blocks: (7.97 GB/7.42 GiB)
sd 2:0:0:0: [sdb] Write Protect is off
sd 2:0:0:0: [sdb] Mode Sense: 43 00 00 00
sd 2:0:0:0: [sdb] No Caching mode page present
sd 2:0:0:0: [sdb] Assuming drive cache: write through
sd 2:0:0:0: [sdb] No Caching mode page present
sd 2:0:0:0: [sdb] Assuming drive cache: write through
 sdb: sdb1
sd 2:0:0:0: [sdb] No Caching mode page present
sd 2:0:0:0: [sdb] Assuming drive cache: write through
sd 2:0:0:0: [sdb] Attached SCSI removable disk
FAT-fs (sdb): error, fat_get_cluster: invalid cluster chain (i_pos 0)
FAT-fs (sdb): Filesystem has been set read-only
```

Deep Sleep Capability

USB working across Deep sleep using Timer Interrupt

System is put into deep sleep using the following command :

```
~# echo 30 > /sys/devices/system/mpic/timer_wakeup;echo mem > /sys/power/state
PM: Syncing filesystems ... done.
mmc0: card e624 removed
Freezing user space processes ... (elapsed 0.001 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
Suspending console(s) (use no_console_suspend to debug)
sd 0:0:0:0: [sda] Synchronizing SCSI cache
sd 0:0:0:0: [sda] Stopping disk
PM: suspend of devices complete after 316.061 msecs
PM: late suspend of devices complete after 0.217 msecs
PM: noirq suspend of devices complete after 31.099 msecs
Disabling non-boot CPUs ...
/pcie@ffe240000: PCICSRBAR @ 0xff000000
/pcie@ffe240000: Setup 64-bit PCI DMA window
/pcie@ffe240000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the
memory map could reduce unnecessary bounce buffering.
/pcie@ffe240000: DMA window size is 0xe0000000
/pcie@ffe250000: PCICSRBAR @ 0xff000000
/pcie@ffe250000: Setup 64-bit PCI DMA window
/pcie@ffe250000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the
memory map could reduce unnecessary bounce buffering.
/pcie@ffe250000: DMA window size is 0xe0000000
/pcie@ffe260000: PCICSRBAR @ 0x0
/pcie@ffe260000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the
memory map could reduce unnecessary bounce buffering.
/pcie@ffe260000: DMA window size is 0x0
/pcie@ffe270000: PCICSRBAR @ 0xff000000
/pcie@ffe270000: Setup 64-bit PCI DMA window
/pcie@ffe270000: WARNING: Outbound window cfg leaves gaps in memory map. Adjusting the
memory map could reduce unnecessary bounce buffering.
/pcie@ffe270000: DMA window size is 0xe0000000
```

After 30 seconds, system comes out of deep sleep and usb storage device is successfully detected

```
Enabling non-boot CPUs ...
CPU1 is up
CPU2 is up
CPU3 is up
PM: noirq resume of devices complete after 63.844 msecs
PM: early resume of devices complete after 0.166 msecs
caam ffe300000.crypto: Instantiated RNG4 SH0
caam ffe300000.crypto: Instantiated RNG4 SH1
ata2: No Device OR PHYRDY change,Hstatus = 0x80000000
ata2: SATA link down (SStatus 10 SControl 300)
ata1: Signature Update detected @ 504 msecs
ata1: SATA link up 3.0 Gbps (SStatus 123 SControl 300)
ata1.00: configured for UDMA/133
sd 0:0:0:0: [sda] Starting disk
PM: resume of devices complete after 4461.429 msecs
Restarting tasks ... done.
usb 1-1: USB disconnect, device number 3
root@t1040rdb:~# EXT2-fs (sdb1): previous I/O error to superblock detected

EXT2-fs (sdb1): previous I/O error to superblock detected

EXT2-fs (sdb1): previous I/O error to superblock detected

EXT2-fs (sdb1): previous I/O error to superblock detected

mmc0: new high speed SDHC card at address e624
mmcblk0: mmc0:e624 SU08G 7.40 GiB
 mmcblk0: p1
usb 1-1: new high-speed USB device number 4 using fsl-ehci
usb-storage 1-1:1.0: USB Mass Storage device detected
scsi4 : usb-storage 1-1:1.0
scsi 4:0:0:0: Direct-Access     JetFlash Transcend 4GB    8.07 PQ: 0 ANSI: 4
sd 4:0:0:0: Attached scsi generic sg1 type 0
sd 4:0:0:0: [sdb] 7843200 512-byte logical blocks: (4.01 GB/3.73 GiB)
sd 4:0:0:0: [sdb] Write Protect is off
sd 4:0:0:0: [sdb] Write cache: disabled, read cache: enabled, doesn't support DPO or FUA
 sdb: sdb1
sd 4:0:0:0: [sdb] Attached SCSI removable disk
```

Deep Sleep using USB wake-up interrupt

This feature is not yet supported.

Known Bugs, Limitations, or Technical Issues

1. Across system Deep Sleep, if a device is already mounted, it may get umounted automatically. Hence, to use it again, the user needs to re-mount the device
2. USB remote wake-up during system Deep-Sleep is not yet supported
3. On some platforms where USB2 controller is muxed with some other IP, USB2 is disabled in default platform configurations inside both U-boot and Linux. For more details, please refer Platform BSP/Board User Manuals
4. Dual-Utmi Phy HW register restoration requirement for System Deep-Sleep feature: Some SoCs have a new utmi phy version called "dual-utmi" phy (for example: T1040, T1042, T1020, T1022, T2080, T2081: rev1.0 and rev1.1). This dual-phy hw registers need to be saved and restored across system Deep-Sleep. Hence, a code is added in u-boot usb driver that identifies all socs having this dual-utmi phy, and adds "dual_utmi" in phy_type property. This is used to determine if all phy registers are to be saved (during system-suspend) and restored (during system-resume). In absence of restoration of dual-phy hw registers, system restore during deep-sleep is going to fail - system hangs and goes into non-recoverable state.

7.2.16.2 USB 2.0 Gadget Network Driver User Manual

Description
The NXP processor has a High speed Dual-Role(DR) USB controller, which supports device mode

Module Loading

USB device controller driver can be built in kernel or compiled as a module.

Gadget drivers are recommended to be built as modules, because parameters will be passed as module parameter

Table 85. Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`Device Drivers --->`
`   USB support --->`
`     <*>   Support for Host-side USB`
`     <*>   EHCI HCD (USB 2.0) support`
`     -*-   Root Hub Transaction  Translators`
`     [ ]   Use Xilinx usb host EHCI controller core`
`     [*]   Support for Freescale PPC on-chip EHCI USB controller`
	Need to enable CONFIG_USB_FSL_MPH_DR_OF

Table continues on the next page...

Table 85. Kernel Configure Tree View Options (continued)

Kernel Configure Tree View Options	Description

`Device Drivers --->`
`USB support --->`
`USB Gadget Support --->`
`< M > Support for USB Gadgets`
`USB Peripheral Controller (Freescale Highspeed USB DR Peripheral Controller) --->`
`Freescale Highspeed USB DR Peripheral Controller`
	Enable NXP USB Device Controller support

`Device Drivers --->`
`USB support --->`
`USB Gadget Support --->`
`< M > Support for USB Gadgets`
`USB Gadget Drivers`
`<M> Ethernet Gadget (with CDC Ethernet support)`
`[*]RNDIS support (NEW)`
`[ ]Ethernet Emulation Model (EEM) support (NEW)`
	Enable USB Gadget support

Table 86. Compile-time Configuration Options

Options	Values	Default	Description
CONFIG_USB_SUPPORT	y/n/m	Ym	Enable USB Support
CONFIG_USB_FSL_MPH_DR_OF	y/n/m	Y	Enable NXP EHCI USB controller
CONFIG_USB_GADGET	y/n/m	m	Enable USB Gadget modules
CONFIG_USB_GADGET_FSL_USB2	y/n/m	m	Enable NXP USB peripheral controller
CONFIG_USB_ETH	y/n/m	m	Enable Ethernet Gadget
CONFIG_USB_ETH_RNDIS	y/n	y	Enable Ethernet Gadget

Source Files

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

Table 87. Source Files

Source File	Description
drivers/usb/gadget/fsl_usb2_udc.c	NXP USB peripheral controller driver

Table continues on the next page...

Table 87. Source Files (continued)

Source File	Description
drivers/usb/host/fsl-mph-dr-of.c	Hook between OF tree and platform device
drivers/usb/gadget/ether.c	Ethernet gadget driver
Drivers/usb/gadget/rndis.[ch]	Microsoft's RNDIS support

Device Tree Entry

```
usb@22000 {

 #address-cells = <1>;
 #size-cells = <0>;
 compatible = "fsl-usb2-<controller-type>-v<controller version>",
              "fsl-usb2-<controller-type>";
 reg = <0x22000 0x1000>;    /* specifies register base addr, soc dependent */
 interrupt-parent = <&mpic>;
 interrupts = <28 0x2>;     /* specifies usb interrupt line, soc dependent */
 phy_type = "ulpi";         /* phy can be ulpi(external)/utmi(internal) */
 dr_mode = "peripheral"     /* this entry specifies usb mode */
 };
```

> **NOTE**
>
> Controller-type: dr(dual-role), mph(multi-port-host) controller-version: 1.6, 2.2, or earlier default mode is always host. It can be either changed to peripheral inside the dts entry like above. In this case re-compilation of dts is required. DR mode can also be changed to peripheral via u-boot command line. This won't require DTS recompilation, and can work with default DTS For USB1 controller.

```
=> setenv hwconfig 'usb1:dr_mode=peripheral,phy_type=<ulpi/utmi>
```

Test Procedure

For board specific changes (required for USB Gadget mode), please refer to the board BSP User Manual.

1. Bring all USB Gadget modules (driver/usb/gadget/*.ko including fs/configfs/configfs.ko) onto the target board.
2. Load device controller driver and test ethernet gadget

   Load FSL gadget driver module **udc-core.ko** & ***fsl_usb2_udc.ko***

   ```
   bash# insmod udc-core.ko
   bash# insmod fsl_usb2_udc.ko
   ```

   Load Ethernet modules

   ```
   bash# insmod configfs.ko
   bash# insmod libcomposite.ko
   bash# insmod u_ether.ko
   bash# insmod u_rndis.ko
   bash# insmod usb_f_rndis.ko
   bash# insmod usb_f_ecm.ko
   bash# insmod usb_f_ecm_subset.ko
   bash# insmod g_ether.ko
   using random self ethernet address
   using random host ethernet address
   usb0: HOST MAC 82:14:b4:63:d1:85
   usb0: MAC 4a:b1:59:3b:b3:bd
   using random self ethernet address
   using random host ethernet address
   g_ether gadget: Ethernet Gadget, version: Memorial Day 2008
   g_ether gadget: g_ether ready
   ```

```
bash# ifconfig usb0
usb0      Link encap:Ethernet  HWaddr 5e:0c:de:2f:f9:0f
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

3. Assign an IP to usb0

```
bash# ifconfig usb0 10.232.1.11 netmask 255.255.255.0 up
IPv6: ADDRCONF(NETDEV_UP): usb0: link is not ready

bash# ifconfig usb0

usb0      usb0      Link encap:Ethernet  HWaddr 5e:0c:de:2f:f9:0f
          inet addr:10.232.1.11  Bcast:10.232.1.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

4. Connect a USB cable between target board USB port and the USB port on Windows host machine.

   As soon as USB cable is plugged into a Windows XP host, the following message displays:

```
http://embedded.seattle.intel-research.net/wiki/index.php?
title=Setting_up_USBnet#Install_the_RNDIS_Driver
```

5. Download linux.inf from either of the following, and install the Windows XP RNDIS driver as mentioned in the previous step:

```
http://www.davehylands.com/linux/gumstix/usbnet/linux.inf
http://embedded.seattle.intel-research.net/wiki/files/linux.inf
```

   For Windows 7, driver will automatically install.

6. As soon as driver installed on host, the following message displays on the target:

```
bash# g_ether gadget: high-speed config #2: RNDIS
IPv6: ADDRCONF(NETDEV_CHANGE): usb0: link becomes ready
```

7. Once the RNDIS driver is installed, configured, and loaded, configure the IP address for the new network device.

   For example, assign 10.232.1.10 as IP to the RNDIS device and run ipconfig to verify the network configuration.

8. Now run ping both ways to check the connectivity between RNDIS@Windows and usb0@linux

```
D:\Profiles>ping 10.232.1.11

Pinging 10.232.1.11 with 32 bytes of data:

Reply from 10.232.1.11: bytes=32 time<1ms TTL=64
Reply from 10.232.1.11: bytes=32 time<1ms TTL=64
Reply from 10.232.1.11: bytes=32 time<1ms TTL=64

Ping statistics for 10.232.1.11:
    Packets: Sent = 3, Received = 3, Lost = 0 (0% loss),

Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms

bash# ping 10.232.1.10

PING 10.232.1.10 (10.232.1.10): 56 data bytes

64 bytes from 10.232.1.10: seq=0 ttl=128 time=4.352 ms
64 bytes from 10.232.1.10: seq=1 ttl=128 time=1.015 ms
```

```
64 bytes from 10.232.1.10: seq=2 ttl=128 time=0.974 ms
64 bytes from 10.232.1.10: seq=3 ttl=128 time=0.935 ms
64 bytes from 10.232.1.10: seq=4 ttl=128 time=1.021 ms

--- 10.232.1.10 ping statistics ---

5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.935/1.659/4.352 ms
```

Known Bugs, Limitations, or Technical Issues

If a board/platform is having multiple USB controller, they cannot be simultaneously used in "gadget/peripheral" mode. Please do not set dr_mode as “peripheral” for both the controllers at the same time.

Supporting Documentation

Linux USB gadget framework - http://www.linux-usb.org/gadget/

Please refer to http://embedded.seattle.intel-research.net/wiki/index.php?title=Setting_up_USBnet for setting up the RNDIS on Windows XP.

Linux USBnet @ http://www.linux-usb.org/usbnet/

7.2.16.3 USB 3.0 Host/Peripheral Linux Driver User Manual

Description

The driver supports xHCI SuperSpeed (SS) Dual-Role-Device (DRD) controller

Main features of xHCI controller

- Supports operation as a standalone USB xHCI host controller
- USB dual-role operation and can be configured as host or device
- Super-speed (5 GT/s), High-speed (480 Mbps), full-speed (12 Mbps), and low-speed (1.5 Mbps) operations
- Supports operation as a standalone single port USB
- Supports eight programmable, bidirectional USB endpoints

Modes of Operation

- Host Mode: SS/HS/FS/LS
- Device Mode: SS/HS/FS

NOTE

Super-speed operation is not supported when OTG is enabled

NOTE

This document explains working of **HS Host and HS Device** in Linux

Module Loading

The default kernel configuration enables support for USB_DWC3 as built-in kernel module.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`Device Drivers--->`
`USB support --->`
`[*] Support for Host-side USB`
	Enables USB host controller support

`Device Drivers--->`
`USB support --->`
`<*>     xHCI HCD (USB 3.0) support`
	Enables XHCI Host Controller Driver and transaction translator

`Device Drivers--->`
`USB support --->`
`<*>     USB Mass Storage support`
`[ ]       USB Mass Storage verbose debug`
	Enable support for USB mass storage devices. This is the driver needed for USB flash devices, and memory sticks

`<*> Sound card support  --->`
`<*>   Advanced Linux Sound Architecture   --->`
`   <*>   OSS Mixer API`
`   <*>   OSS PCM (digital audio) API`
`   [*]     OSS PCM (digital audio) API - Include plugin`
`system`
`   [*]   Support old ALSA API`
`   [*]   USB sound devices   --->`
`     <*>   USB Audio/MIDI driver`
	Enables support for USB Audio devices. This driver is needed for USB microphone.

`Device Drivers--->`
`USB support --->`
` <*>   USB Gadget Support --->`
`    <M>   USB Gadget Drivers`
`    < >     USB functions configurable through configfs`
`    < >     Gadget Zero (DEVELOPMENT)`
`    <M>     Ethernet Gadget (with CDC Ethernet support)`
`    [*]       RNDIS support`
`    [ ]       Ethernet Emulation Model (EEM) support`
`    < >     Network Control Model (NCM) support`
`    < >     Gadget Filesystem`
`    < >     Function Filesystem`
`    <M>     Mass Storage Gadget`
`    < >     Serial Gadget (with CDC ACM and CDC OBEX`
`support)`
	Note: Required only for USB Gadget/ Peripheral Support
	• Enable driver for peripheral/device controller
	• Enable Ethernet Gadget Client driver
	• Enable Mass Storage Client driver

`Device Drivers--->`
`<*>   DesignWare USB3 DRD Core Support`
`         DWC3 Mode Selection (Dual Role mode)  --->`
	Enable XHCI DRD Core Support

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_USB	y/m/n	y	Enables USB host controller
CONFIG_USB_XHCI_HCD	y/m/n	y	Enables XHCI HCD
CONFIG_USB_DWC3	y/m/n	y	Enables DWC3 Controller
CONFIG_USB_GADGET	y/m/n	n	Enables USB peripheral device
CONFIG_USB_ETH	y/m/n	n	Enable Ethernet style communication
CONFIG_USB_MASS_STORAGE	m/n	n	Enable USB Mass Storage disk drive
CONFIG_SOUND	y/m/n	y	Enables Sound Card Support
CONFIG_SND	y/m/n	y	Enables ALSA (Advanced Linux Sound Architecture)
CONFIG_SND_MIXER_OSS	y/m/n	y	Enables OSS Mixer API
CONFIG_SND_PCM_OSS	y/m/n	y	Enables OSS PCM (digital audio) API
CONFIG_SND_PCM_OSS_PLUGINS	y/n	y	Enables OSS PCM (digital audio) API - Include plugin system
CONFIG_SND_SUPPORT_OLD_API	y/n	y	Enables old ALSA API
CONFIG_SND_USB	y/n	n	Enables USB sound devices
CONFIG_SND_USB_AUDIO	y/m/n	n	Enables USB Audio/MIDI driver

NOTE: USB Audio configuration options default value is listed for LS1021A platform.

Source Files

The driver source is maintained in the Linux kernel source tree in below files

Table continued from the previous page...

Source File	Description
drivers/usb/host/xhci-*	xhci platform driver
drivers/usb/gadget/mass_storage.c	USB Mass Storage
drivers/usb/gadget/ether.c	Ethernet gadget driver

Device Tree Binding for Host

```
usb@3100000 {
               compatible = "snps,dwc3";
               reg = <0x0 0x3100000 0x0 0x10000>;
               interrupts = <GIC_SPI 93 IRQ_TYPE_LEVEL_HIGH>;
              dr_mode = "host;
            };
```

Device Tree Binding for Peripheral

Note: with multiple USB controller, just one can be peripheral mode at a time.

```
usb@3100000 {
               compatible = "snps,dwc3";
               reg = <0x0 0x3100000 0x0 0x10000>;
               interrupts = <GIC_SPI 93 IRQ_TYPE_LEVEL_HIGH>;
              dr_mode = "peripheral;
               maximum-speed = "super-speed";
          };
```

Host Testing

Following are serial console logs that appear during bootup if dr_mode set to host in device-tree

```
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb
xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 1
xhci-hcd xhci-hcd.0.auto: irq 125, io mem 0x03100000
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 1 port detected
xhci-hcd xhci-hcd.0.auto: xHCI Host Controller
xhci-hcd xhci-hcd.0.auto: new USB bus registered, assigned bus number 2
hub 2-0:1.0: USB hub found
hub 2-0:1.0: 1 port detected
usbcore: registered new interface driver usb-storage
```

Following are serial-console logs after connecting a USB flash drive

```
For High-Speed Device attach
usb 1-1.2: new high-speed USB device number 3 using xhci-hcd
usb-storage 1-1.2:1.0: USB Mass Storage device detected
scsi0 : usb-storage 1-1.2:1.0
scsi 0:0:0:0: Direct-Access     SanDisk  Cruzer           7.01 PQ: 0 ANSI: 0 CCS
sd 0:0:0:0: [sda] 1957887 512-byte logical blocks: (1.00 GB/955 MiB)
sd 0:0:0:0: Attached scsi generic sg0 type 0
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] No Caching mode page found
sd 0:0:0:0: [sda] Assuming drive cache: write through
sd 0:0:0:0: [sda] No Caching mode page found
sd 0:0:0:0: [sda] Assuming drive cache: write through
 sda: sda1
sd 0:0:0:0: [sda] No Caching mode page found
sd 0:0:0:0: [sda] Assuming drive cache: write through
sd 0:0:0:0: [sda] Attached SCSI removable disk

For Super-Speed Device attach
# usb 2-1: new SuperSpeed USB device number 2 using xhci-hcd
usb 2-1: Parent hub missing LPM exit latency info.  Power management will be impacted.
usb-storage 2-1:1.0: USB Mass Storage device detected
scsi0 : usb-storage 2-1:1.0
scsi 0:0:0:0: Direct-Access     SanDisk  Extreme          0001 PQ: 0 ANSI: 6
sd 0:0:0:0: [sda] 31277232 512-byte logical blocks: (16.0 GB/14.9 GiB)
sd 0:0:0:0: Attached scsi generic sg0 type 0
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Write cache: disabled, read cache: enabled, doesn't support DPO or FUA
 sda:
sd 0:0:0:0: [sda] Attached SCSI removable disk
FAT-fs (sda): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
```

Make filesystem and mount connected USB flash drive using below commands

```
root@freescale /$ fdisk -l

Disk /dev/sda: 16.0 GB, 16013942784 bytes
255 heads, 63 sectors/track, 1946 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks  Id System
/dev/sda1               1        1946    15631213+ 83 Linux
root@freescale /$
```

```
root@freescale /$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
shm                     516684         0    516684   0% /dev/shm
rwfs                       512         0       512   0% /mnt/rwfs
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$ fdisk /dev/sda

The number of cylinders for this disk is set to 1946.
There is nothing wrong with that, but this is larger than 1024,
and could in certain setups cause problems with:
1) software that runs at boot time (e.g., old versions of LILO)
2) booting and partitioning software from other OSs
   (e.g., DOS FDISK, OS/2 FDISK)

Command (m for help): d
Selected partition 1

Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-1946, default 1): Using default value 1
Last cylinder or +size or +sizeM or +sizeK (1-1946, default 1946): Using default value 1946

Command (m for help): w
The partition table has been alter sda: sda1
ed!

Calling ioctl() to re-read partition table
root@freescale /$
root@freescale /$
root@freescale /$ fdisk -l

Disk /dev/sda: 16.0 GB, 16013942784 bytes
255 heads, 63 sectors/track, 1946 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks  Id System
/dev/sda1               1        1946    15631213+ 83 Linux
root@freescale /$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
shm                     516684         0    516684   0% /dev/shm
rwfs                       512         0       512   0% /mnt/rwfs
root@freescale /$ mkdir my_mnt
root@freescale /$
root@freescale /$
root@freescale /$ mkfs.ext2 /dev/sda1
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
977280 inodes, 3907803 blocks
195390 blocks (5%) reserved for the super user
First data block=0
Maximum filesystem blocks=4194304
120 block groups
32768 blocks per group, 32768 fragments per group
8144 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632, 2654208
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$
```

```
root@freescale /$
root@freescale /$ mount /dev/sda1 my_mnt/
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
shm                     516684         0    516684   0% /dev/shm
rwfs                       512         0       512   0% /mnt/rwfs
/dev/sda1             15385852        20  14604272   0% /my_mnt
root@freescale /$
```

Test by wring/reading data on mount drive

```
root@freescale /$ dd if=/dev/urandom of=/tmp/123 bs=1M count=100
100+0 records in
100+0 records out
104857600 bytes (100.0MB) copied, 54.535026 seconds, 1.8MB/s
root@freescale /$
root@freescale /$
root@freescale /$
root@freescale /$ cp /tmp/123 /my_mnt/.
root@freescale /$ sync
root@freescale /$ ls /my_mnt/
123         lost+found
root@freescale /$
```

Peripheral testing with Win7 as Host

NOTE

In gadget mode standard USB cables with micro plug should be used.

Below Message will appear during bootup if dr_mode set as peripheral in device-tree

```
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb

usbcore: registered new interface driver usb-storage
```

Make sure "dr_mode" contains "peripheral" string

```
root@freescale /$# cat /proc/device-tree/soc/usb\@3100000/dwc3/dr_mode
peripheral root@freescale /$
```

Move all below modeules to platform

```
fs/configfs/configfs.ko
driver/usb/gadget/libcomposite.ko
driver/usb/gadget/g_mass_storage.ko
driver/usb/gadget/u_rndis.ko
driver/usb/gadget/u_ether.ko
driver/usb/gadget/usb_f_ecm.ko
driver/usb/gadget/usb_f_ecm_subset.ko
driver/usb/gadget/usb_f_rndis.ko
driver/usb/gadget/g_ether.ko
```

Mass Storage Gadget

```
To use ramdisk as a backing store use the following

root@freescale /$ mkdir /mnt/ramdrive
root@freescale /$ mount -t tmpfs tmpfs /mnt/ramdrive -o size=600M
root@freescale /$ dd if=/dev/zero of=/mnt/ramdrive/vfat-file bs=1M count=500
root@freescale /$ mke2fs -F /mnt/ramdrive/vfat-file
root@freescale /$ insmod configfs.ko
root@freescale /$ insmod libcomposite.ko
root@freescale /$ insmod usb_f_mass_storage.ko
root@freescale /$ insmod g_mass_storage.ko file=/mnt/ramdrive/vfat-file stall=n
```

We will get below messages

```
[   39.987594] g_mass_storage gadget: Mass Storage Function, version: 2009/09/11
[   39.994822] g_mass_storage gadget: Number of LUNs=1
[   39.989240]  lun0: LUN: file: /home/backing_file_20mb
[   39.994367] g_mass_storage gadget: Mass Storage Gadget, version: 2009/09/11
[   39.990902] g_mass_storage gadget: userspace failed to provide iSerialNumber
[   39.987547] g_mass_storage gadget: g_mass_storage ready
```

Attached ***USB3.0 only*** gadget cable to host and you will get below message. Now Storage is ready to use.

```
g_mass_storage gadget: super-speed config #1: Linux File-Backed Storage
```

Speaker and Microphone

1. Aplay utility can be used to list the available sound cards e.g. Here Jabra 410 USB speaker is detected as a second sound card and can be addressed as **–D hw:1,0 OR –c1**:

```
[root@freescale ~]$ aplay –l**** List of PLAYBACK Hardware Devices ****
  card 0: FSLVF610TWRBOAR [FSL-VF610-TWR-BOARD], device 0: HiFi sgtl5000-0 [ ]
  Subdevices: 1/1
  Subdevice #0: subdevice #0
  card 1: USB [Jabra SPEAK 410 USB], device 0: USB Audio [USB Audio]  Subdevices: 1/1
Subdevice #0: subdevice #0
```

2. Sample wav file can be played using the below command:

```
[root@freescale ~]$ aplay -D hw:1,0 LYNC_fsringing.wav
Playing WAVE 'LYNC_fsringing.wav' : Signed 16 bit Little Endian, Rate 48000 Hz, Stereo
```

3. Sample wav file can be recorded using the below command:

```
[root@freescale ~]$ arecord -f S16_LE -t wav -Dhw:1,0 -r 16000 foobar.wav -d 5
Recording WAVE 'foobar.wav' : Signed 16 bit Little Endian, Rate 16000 Hz, Mono
```

NOTE: If recorded audio is not played, try to use "-D plughw:1,0" in above command.

4. Audio controls can be checked using the below command, control details and name of the controls can be checked from output of “amixer –c1” as below:

```
[root@freescale ~]$ amixer -c1 controls
numid=3,iface=MIXER,name='PCM Playback Switch'
numid=4,iface=MIXER,name='PCM Playback Volume'
numid=5,iface=MIXER,name='Headset Capture Switch'
numid=6,iface=MIXER,name='Headset Capture Volume'
numid=2,iface=PCM,name='Capture Channel Map'
numid=1,iface=PCM,name='Playback Channel Map'

[root@freescale ~]$ amixer -c1
Simple mixer control 'PCM',0  Capabilities: pvolume pvolume-joined pswitch pswitch-joined
penum
  Playback channels: Mono
  Limits: Playback 0 - 11
  Mono: Playback 4 [36%] [-20.00dB] [on]

Simple mixer control 'Headset',0  Capabilities: cvolume cvolume-joined cswitch cswitch-
joined penum
  Capture channels: Mono
  Limits: Capture 0 - 7
  Mono: Capture 5 [71%] [0.00dB] [on]
```

For Example, in above output there are two controls named “PCM” and “Headset” for Speaker and microphone respectively.

Sample Audio controls Usage:

a. mute/unmute

```
[root@freescale ~]$ amixer -c1 set PCM  mute
Simple mixer control 'PCM',0
  Capabilities: pvolume pvolume-joined pswitch pswitch-joined
  Playback channels: Mono
  Limits: Playback 0 - 11
  Mono: Playback 2 [18%] [-28.00dB] [off]
[root@freescale ~]$ amixer -c1 set PCM  unmute
Simple mixer control 'PCM',0
  Capabilities: pvolume pvolume-joined pswitch pswitch-joined
  Playback channels: Mono
  Limits: Playback 0 - 11
  Mono: Playback 2 [18%] [-28.00dB] [on]
```

b. volume up/down – Below commands are trying to set volume to 11 and 2 performing volume up and down respectively.

```
root@freescale ~]$ amixer -c1 set PCM 11
Simple mixer control 'PCM',0
Capabilities: pvolume pvolume-joined pswitch pswitch-joined
Playback channels: Mono
Limits: Playback 0 - 11
Mono: Playback 11 [100%] [8.00dB] [on]
[root@freescale ~]$ amixer -c1 set PCM 2
Simple mixer control 'PCM',0
Capabilities: pvolume pvolume-joined pswitch pswitch-joined
Playback channels: Mono
Limits: Playback 0 - 11
Mono: Playback 2 [18%] [-28.00dB] [on]
```

Ethernet Gadget

```
To use Ethernet gadget use the following

root@freescale /$ insmod configfs.ko
root@freescale /$ insmod libcomposite.ko
root@freescale /$ insmod u_ether.ko
root@freescale /$ insmod u_rndis.ko
root@freescale /$ insmod usb_f_ecm.ko
root@freescale /$ insmod usb_f_ecm_subset.ko
root@freescale /$ insmod usb_f_rndis.ko
root@freescale /$ insmod g_ether.ko
```

We will get below messages

```
[   28.692611] using random self ethernet address
[   28.697156] using random host ethernet address
[   28.694271] usb0: HOST MAC 82:96:69:7e:a5:7d
[   28.698928] usb0: MAC 72:00:a5:80:2b:e8
[   28.692586] using random self ethernet address
[   28.697080] using random host ethernet address
[   28.691368] g_ether gadget: Ethernet Gadget, version: Memorial Day 2008
[   28.698028] g_ether gadget: g_ether ready
```

Make sure USB0 ethernet interface is available after this

```
root@freescale /$ ifconfig -a
can0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          NOARP  MTU:16  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:10
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:158

can1      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          NOARP  MTU:16  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:10
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:159

eth0      Link encap:Ethernet  HWaddr 00:E0:0C:BC:E5:60
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth1      Link encap:Ethernet  HWaddr 00:E0:0C:BC:E5:61
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

eth2      Link encap:Ethernet  HWaddr 00:E0:0C:BC:E5:62
          inet addr:10.232.132.212  Bcast:10.232.135.255  Mask:255.255.252.0
          inet6 addr: fe80::2e0:cff:febc:e562/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2311 errors:0 dropped:3 overruns:0 frame:0
          TX packets:66 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:290810 (283.9 KiB)  TX bytes:8976 (8.7 KiB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:100 (100.0 B)  TX bytes:100 (100.0 B)

sit0      Link encap:IPv6-in-IPv4
          NOARP  MTU:1480  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

usb0      Link encap:Ethernet  HWaddr 72:00:A5:80:2B:E8
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Attached the cable with Win7 and Configure RNDS interface in windows under "Control Panel -> Network and Internet -> Network Connections" and set IP Address

Set IP Address in Platform and start Ping

```
root@freescale /$ ifconfig usb0 10.232.1.11
root@freescale /$
root@freescale /$
root@freescale /$ ping usb   10.232.1.10
PING 10.232.1.10 (10.232.1.10): 56 data bytes
64 bytes from 10.232.1.10: seq=0 ttl=128 time=5.294 ms
64 bytes from 10.232.1.10: seq=1 ttl=128 time=6.101 ms
64 bytes from 10.232.1.10: seq=2 ttl=128 time=4.170 ms
64 bytes from 10.232.1.10: seq=3 ttl=128 time=4.233 ms
```

Known Bugs, Limitations, or Technical Issues

- Linux only allows one peripheral at one time. Make sure that when one of DWC3 controller 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 a OTG3.0 cable compatibility issue. An OTG 2.0 cable and USB standard port will work fine.

7.2.17 Watchdog

Module Loading

Watchdog device driver support kernel built-in mode.

U-Boot Configuration

Runtime options

Env Variable	Env Description	Sub option	Option Description
bootargs	Kernel command line argument passed to kernel	setenv othbootargs wdt_period=35	Sets the watchdog timer period timeout

Kernel Configure Options

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`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 applications will be used during functional or performance testing. Please refer to the SDK UM document for the detailed build procedure.

Command Name	Description	Package Name
watch	watchdog is a daemon for watchdog feeding	watchdog

Verification in Linux

- Set NFS rootfs. Build a rootfs image which includes watchdog daemon.
- 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 ie. the higher the `wdt_period`, the lower the watchdog expiry time. So 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`

7.2.18 QUICC Engine HDLC/TDM User Manual

Linux SDK for QorIQ Processors

Description

HDLC, standing for High-level Data Link Control, is one of the most common protocols of the Layer 2 (Data Link Layer) of the seven-layer OSI model. HDLC uses a zero insertion/deletion process (commonly known as bit stuffing) to ensure that the bit pattern of the delimiter flag does not occur in the fields between flags. The HDLC frame is synchronous and therefore relies on the physical layer for a method of clocking and of synchronizing the transmitter/receiver.

The HDLC/TDM driver is implemented by UCC and TSA(HDLC is upper layer protocol of TDM). It enables UCC1/3 to work in hdlc protocol, connected with X-TDM-DS26522 card to support T1/E1 function. It can work in normal or loopback mode both for tdm controller and phy. connect X-TDM-DS26522 card to TDM Riser slot, it can transmit data and receive data.

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

Choosing predefined u-boot modes:

make ls1043ardb_deconfig

before doing the actually build

Runtime options

Env Variable	Env Description	Sub option	Option Description
hwconfig	Hardware configuration for u-boot	qe-hdlc	Assgin pins for HDLC; QUICC Engine TDM enabled in DTB
bootargs	Kernel command line argument passed to kernel		

Kernel Configure Options

Tree View

LS1043ARDB and X-TDM-DS26522 card:

<table>
<tr><th>Kernel Configure Tree View Options</th><th>Description</th></tr>
<tr><td><pre>
Device Drivers --->
    SOC (System On Chip) specific Drivers --->
        [*] Freescale QUICC Engine (QE) Support
    [*] Network device support --->
        [*]   Wan interfaces support --->
            <*>   Generic HDLC layer
            <*>   Raw HDLC support
            <*>   Freescale QUICC Engine HDLC support
            <*>   SLIC MAXIM DS26522 CARD SUPPORT
</pre></td><td>Enable the QE TDM driver and X-TDM-DS26522 card driver.</td></tr>
</table>

Identifier

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

Option	Values	Default Value	Description
CONFIG_QUICC_ENGINE	y/n	n	QUICC Engine enabled
CONFIG_FSL_UCC_TDM	y/n	n	QUICC Engine TDM lib
CONFIG_SLIC_MAXIM	y/m/n	n	Enable x-tdm-ds26522 card support
FSL_UCC_HDLC	y/m/n	n	QUICC Engine driver driver

Device Tree Binding

Below is the definition of the device tree node required by this feature

Property	Type	Status	Description
qe	qe	enable	QUICC Engine node
ucc	hdlc	enable	QE UCC HDLC node.
si	si	si	QE TSA node

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

LS1040ARDB and X-TDM-DS26522 card:

```
ucc_hdlc: ucc@2000 {
                compatible = "fsl,ucc-hdlc";
                rx-clock-name = "clk8";
                tx-clock-name = "clk9";
                fsl,rx-sync-clock = "rsync_pin";
                fsl,tx-sync-clock = "tsync_pin";
                fsl,tx-timeslot-mask = <0xfffffffe>;
                fsl,rx-timeslot-mask = <0xfffffffe>;
                fsl,tdm-framer-type = "e1";
                fsl,tdm-id = <0>;
                fsl,siram-entry-id = <0>;
                fsl,tdm-interface;
        };
        slic@3 {
                compatible = "maxim,ds26522";
                reg = <3>;
                spi-max-frequency = <2000000>;
                fsl,spi-cs-sck-delay = <100>;
                fsl,spi-sck-cs-delay = <50>;
        };
```

Source Files

The following source file are related the this feature in Linux.

T1040RDB and X-TDM-DS26522 card:

Source File	Description
drivers/soc/fsl/qe/qe_tdm.c	QE UCC TDM lib
include/soc/fsl/qe/qe_tdm.h	QE UCC TDM lib head file.
drivers/net/tdm/slic_ds26522.c	X-TDM-DS26522 card driver.
drivers/net/wan/fsl_ucc_hdlc.*	QE HDLC driver
arch/arm64/boot/dts/freescale/fsl-ls1043a.dtsi	Define the device tree nodes for LS1043ARDB QE
arch/arm64/boot/dts/freescale/fsl-ls1043a-rdb.dts	Define the device tree nodes for LS1043ARDB ds26522

User Space Application

The following applications will be used during functional or performance testing. Please refer to the SDK UM document for the detailed build procedure.

Command Name	Description	Package Name

Verification in U-boot

N/A

Verification in Linux

1. After u-boot startup,set "qe-hdlc" parameter in hwconfig.
2. After bootup kernel, Kernel boot log for hdlc:

   ```
   hdlc: HDLC support module revision 1.22
   ```

3. QE HDLC T1/E1 test
   a. Make X-TDM-DS26522 card connected to T1040RDB board Slot.

b. To test tdm external ports, please plugin tdm t1/e1 loopback cable in the related port.

   The following is HDLC port mapping with X-TDM-DS26522 card:

```
HDLC Port                 X-TDM-DS26522 Port
Port A                         CH1;
Port B                         CH2;
```

c. HDLC test using E1.

   Use the default dts to test E1 function. Test module can receive ucc_num as parameter. This number should be 1/3 related to the tdm port.

```
ls1043ardb login: root
root@ls1043ardb:~# ./sethdlc hdlc0 hdlc;
root@ls1043ardb:~# ifconfig hdlc0 192.168.0.1 up
[   41.072590] hdlc0: Carrier detected
root@ls1043ardb:~# route add -net 192.168.0.0 netmask 255.255.255.0 gw 192.168.0.1
hdlc0;
root@ls1043ardb:~# ping 192.168.0.2;
PING 192.168.0.2[   52.208784] Tx data skb->len:86  (192.168.0.2) 56(84) bytes of
d[   52.213119]
[   52.213119] Transmitted data:
ata.
[   52.220324] ff
[   52.222491] 44
[   52.224154] 45
[   52.225810] 00
[   52.227472] 00
[   52.229125] 54
[   52.230778] c3
[   52.232440] 89
[   52.234094] 40
[   52.235755] 00
[   52.237408] 40
[   52.239069] 01
[   52.240722] f5
[   52.242375] cb
[   52.244038] c0
[   52.245691] a8
[   52.247844] irq ucce:20000
[   52.250543] TxBD: 1c00
[   52.252900] Received data length:88[   52.256206] while entry times:0
[   52.259338]
[   52.259338] Received data:
[   52.263512] ff
[   52.265165] 44
[   52.266818] 45
[   52.268474] 00
[   52.270127] 00
[   52.271782] 54
[   52.273435] c3
[   52.275091] 89
[   52.276744] 40
[   52.278397] 00
[   52.280052] 40
[   52.281705] 01
[   52.283361] f5
[   52.285014] cb
[   52.286667] c0
[   52.288322] a8
[   52.289980] skb->protocol:8
[   52.292784] irq ucce:80000
[   53.262909] Tx data skb->len:86 [   53.265951]
```

Chapter 8
QorIQ networking technologies

8.1 Summary of networking technologies

NXP provides several different hardware networking architectures. Each SoC incorporates one of them. The hardware architectures are:

HW networking architectures	Blocks
DPAA1	QMan, BMan, and FMan
DPAA2	QBMan, WRIOP, and optionally AIOP
DPAA2 and DPAA1 are relatives in that they both use generic hardware-based queues. Also, each supports additional accelerators such as SEC via these queues.	
PFE	PFE package engine block
veTSEC	veTSEC traditional Ethernet controller block

The following table shows which SoCs supported by LSDK use which networking hardware architecture.

HW networking architectures	SoCs
DPAA1	LS1023A, LS1043A, LS1026A, LS1046A
DPAA2	LS1044A, LS1048A, LS1084A, LS1088A, LS2044A, LS2048A, LS2084A, LS2088A
PFE	LS1012A
veTSEC	L1021A

8.2 DPAA1-specific Software

8.2.1 DPAA1 software architecture overview

8.2.1.1 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.

8.2.1.1.1 General architectural considerations

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

8.2.1.1.2 Multicore design

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

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

8.2.1.1.3 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.

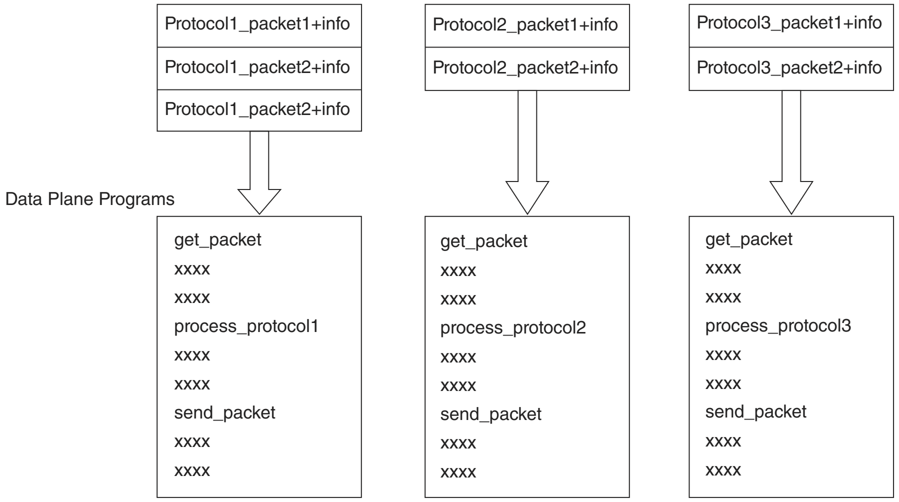


Figure 61. Hardware-sorted protocol flow

8.2.1.1.4 Flow order considerations

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

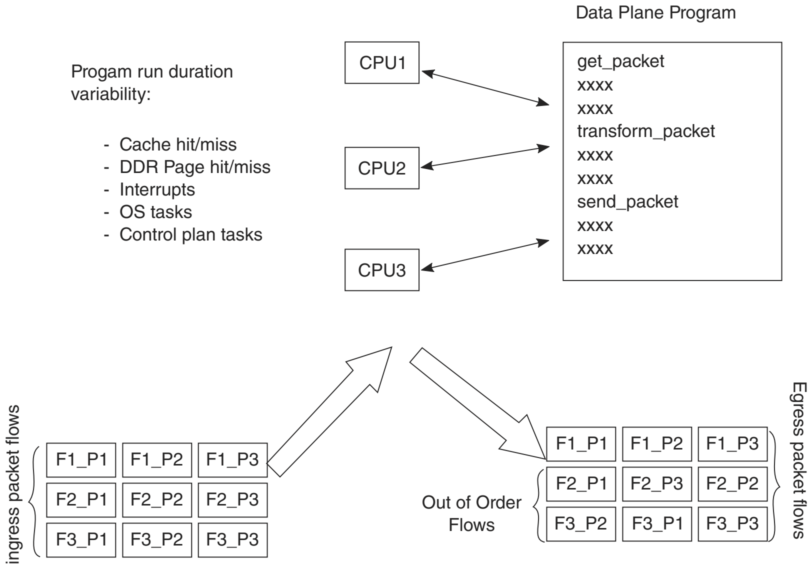


Figure 62. Multicore Flow Reordering

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.

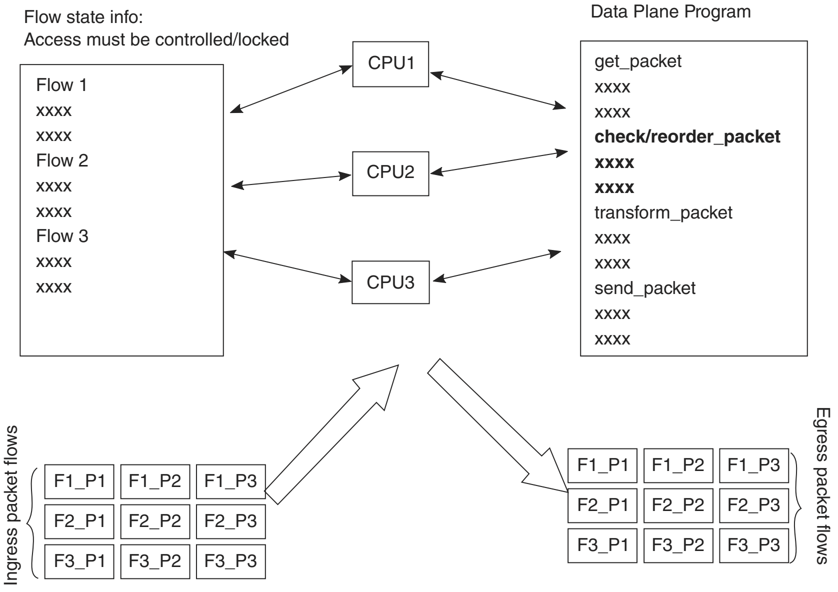


Figure 63. Implementing Order in Software

8.2.1.1.5 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.

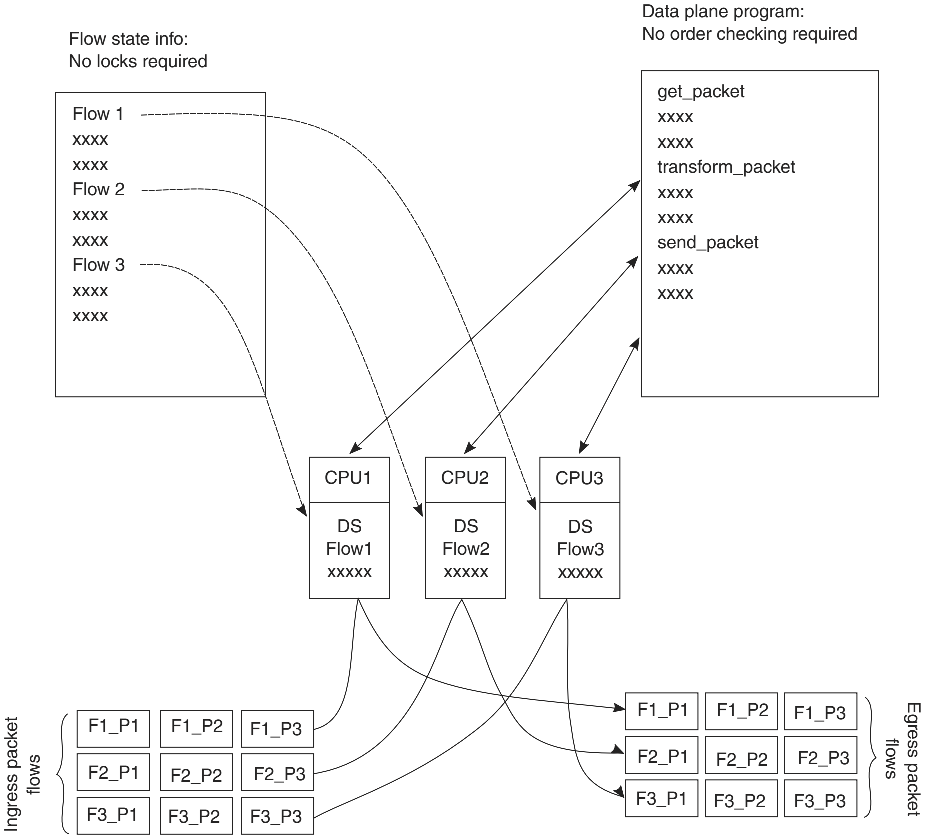


Figure 64. Managing flow-to-core affinity

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.

8.2.1.2 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.

8.2.1.3 FMan Overview

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

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

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

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

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

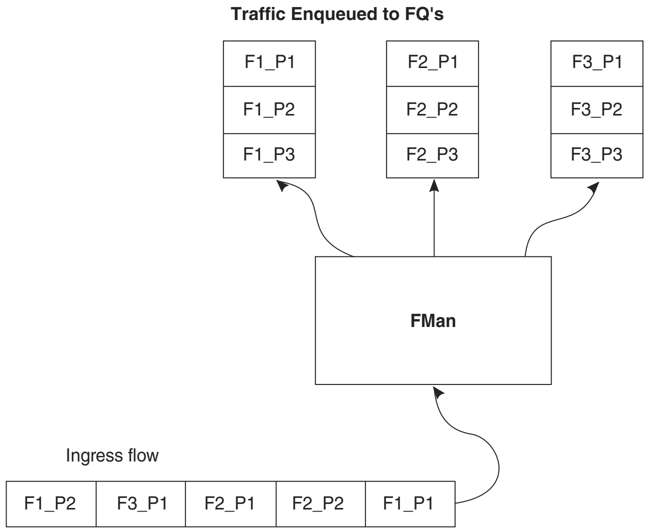


Figure 65. Ingress FMan Flow

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.

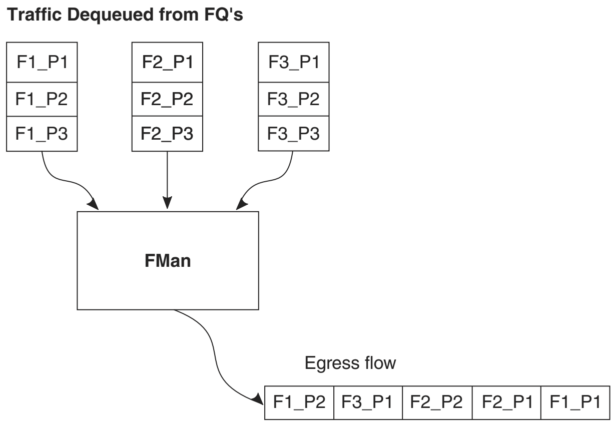


Figure 66. FMan Egress Flow

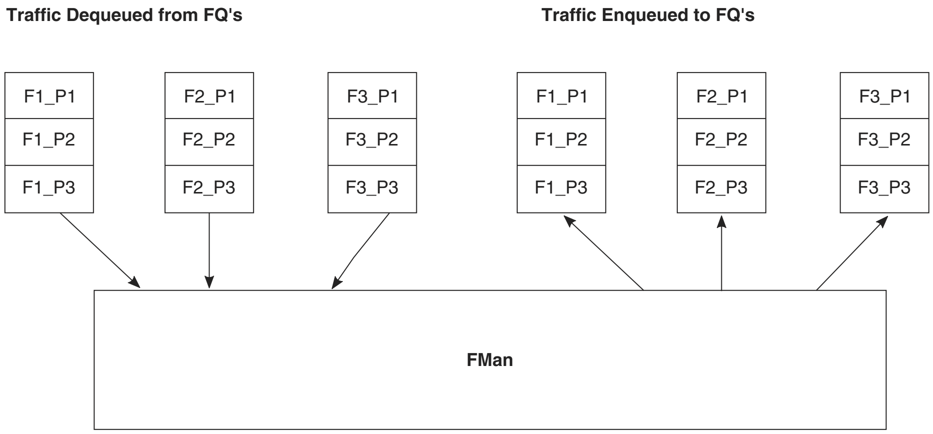


Figure 67. FMan Offline Flow

8.2.1.4 QMan Overview

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

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

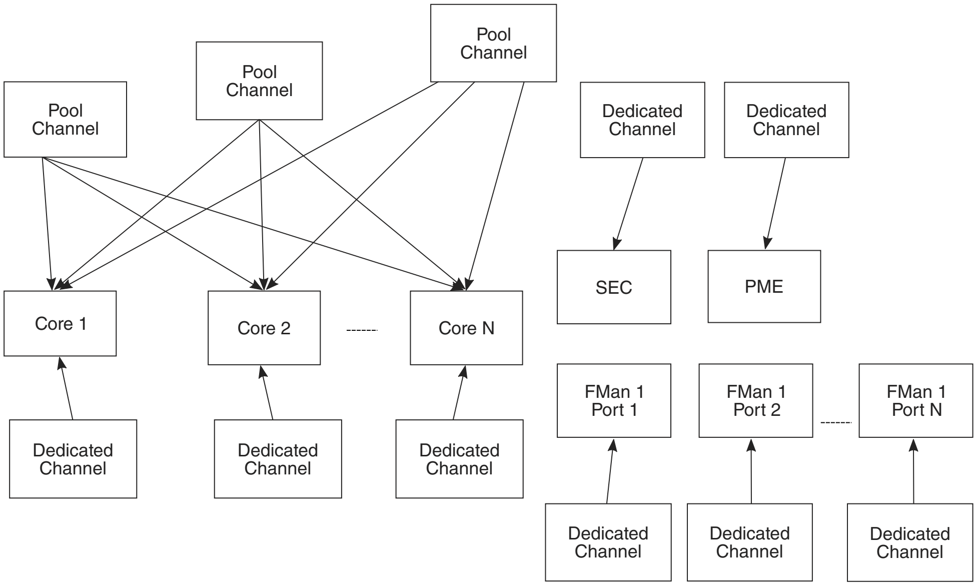
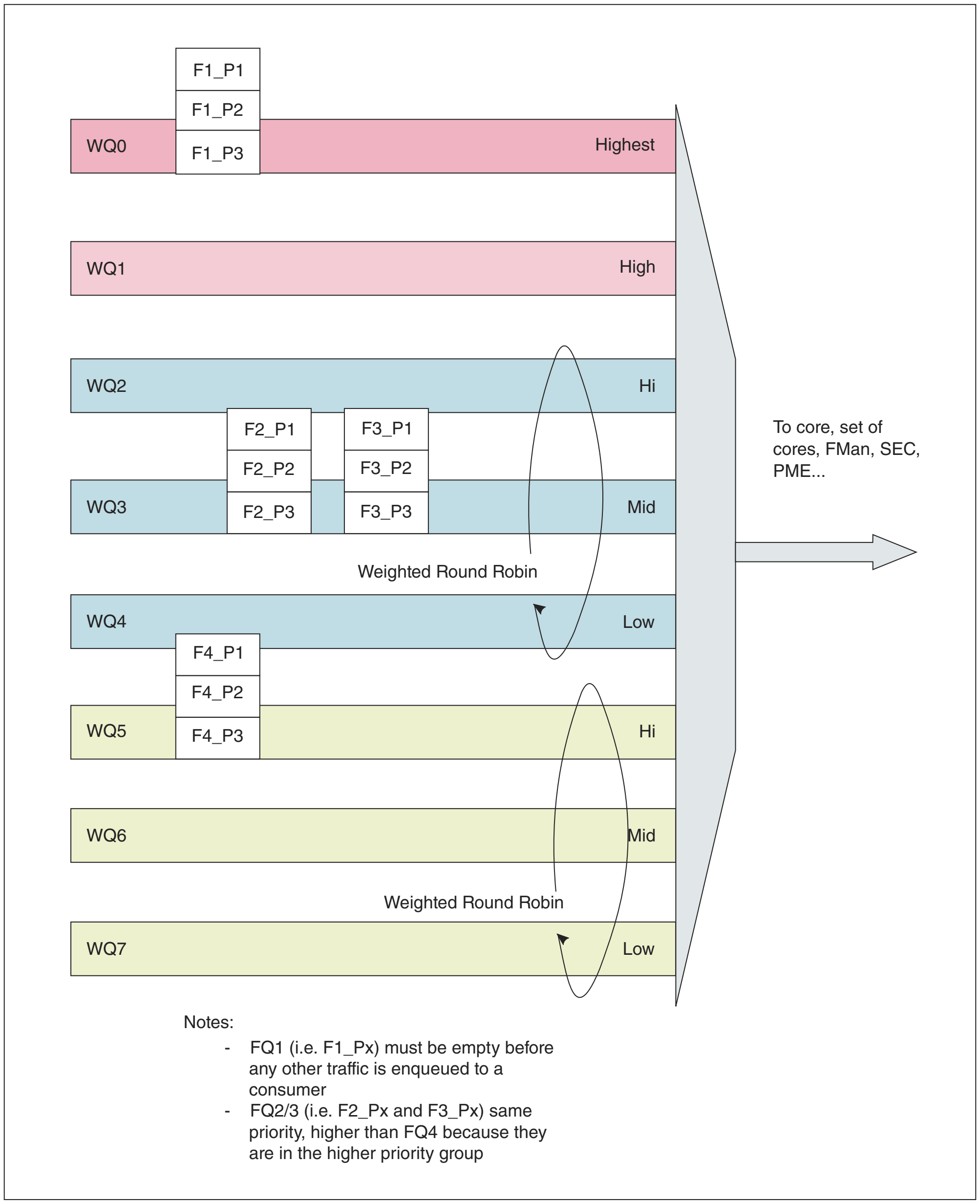


Figure 68. DPAA1 Channel Types

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.



Notes:

- FQ1 (i.e. F1_Px) must be empty before any other traffic is enqueued to a consumer
- FQ2/3 (i.e. F2_Px and F3_Px) same priority, higher than FQ4 because they are in the higher priority group

Figure 69. 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.

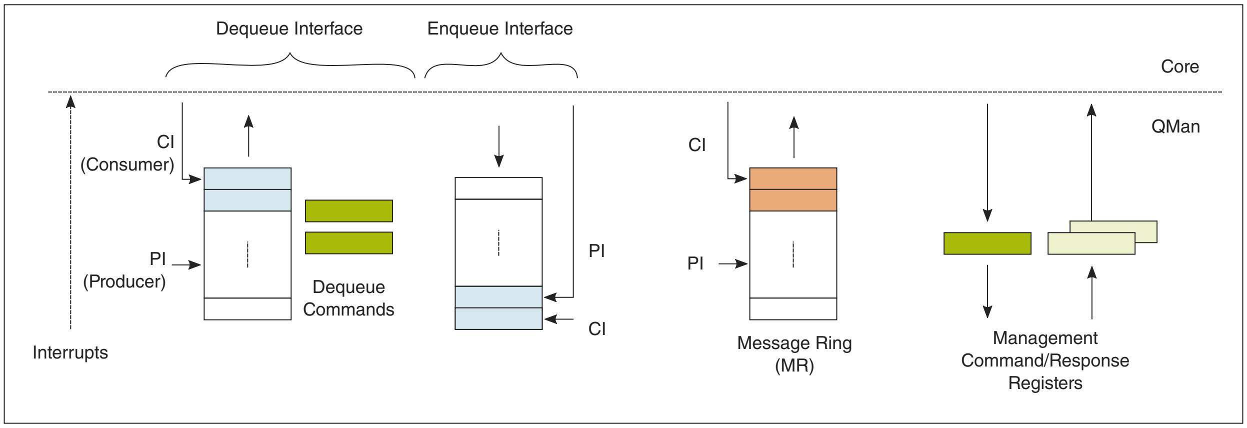


Figure 70. Processor Core Portal

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.

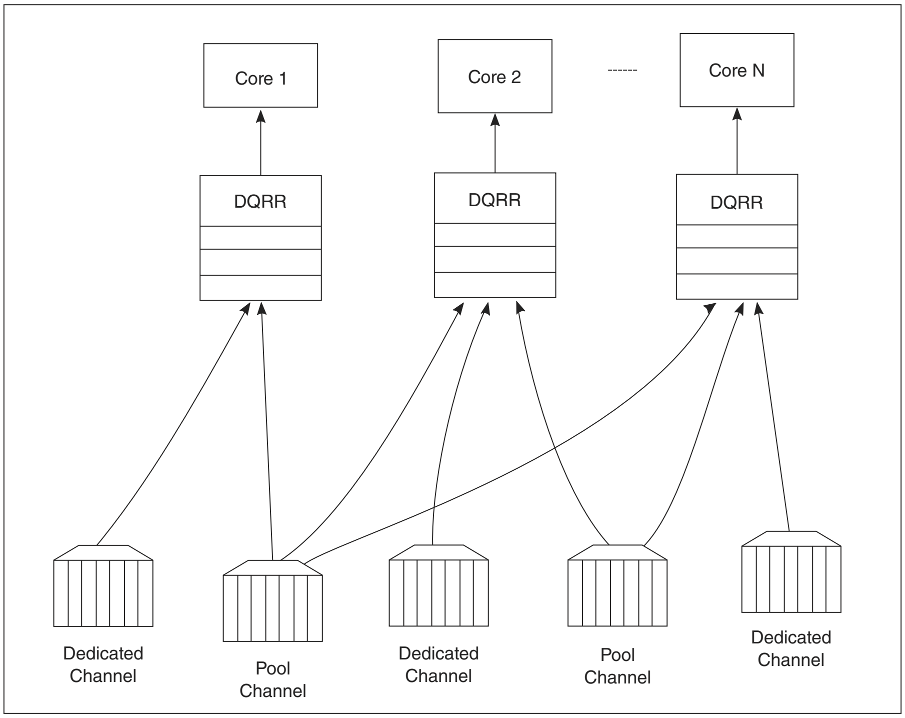


Figure 71. Ingress Channel to Portal Options

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.

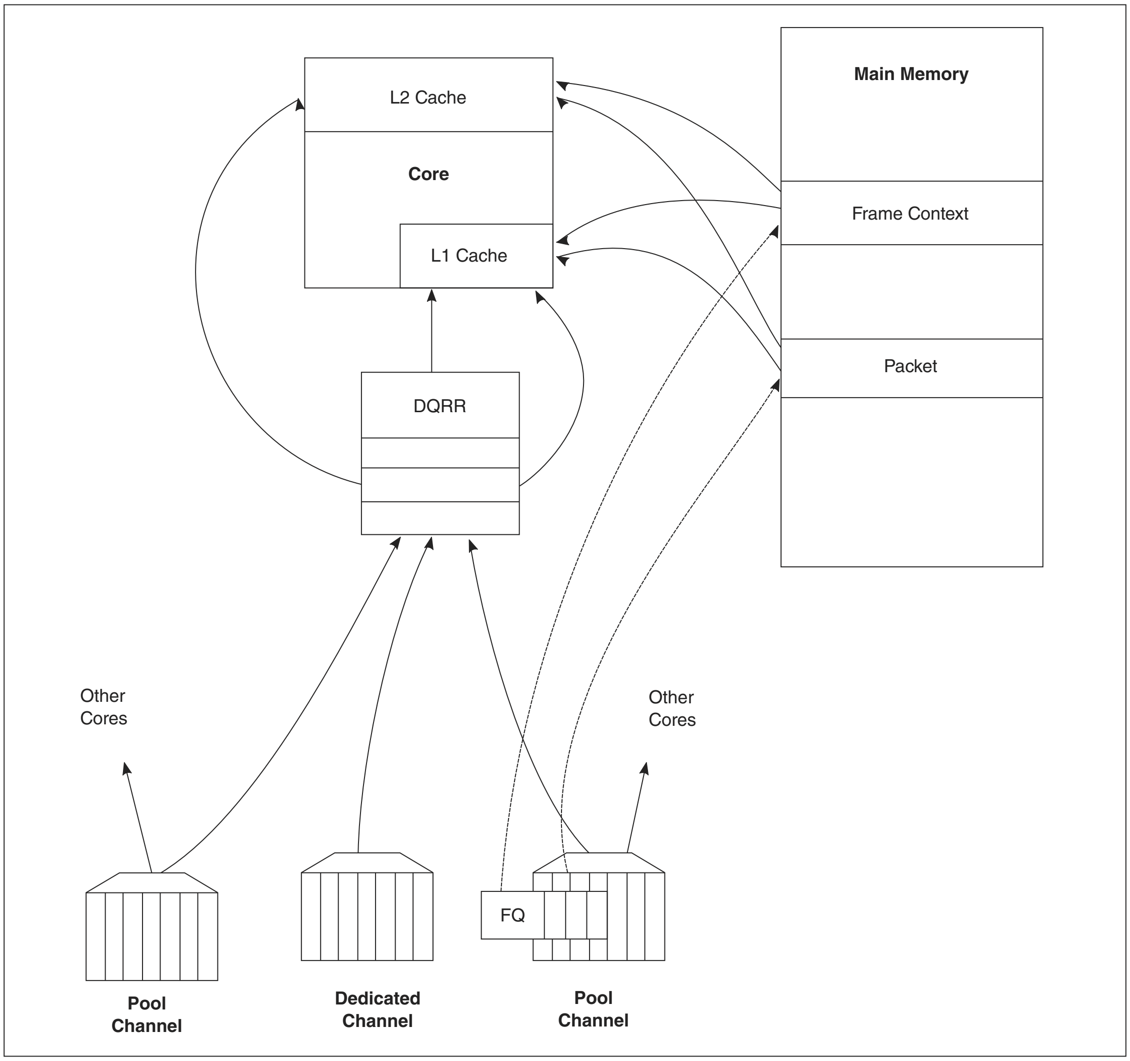


Figure 72. Cache Warming Options

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.

8.2.1.5 QMan Scheduling

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

QMan: Queue schedule options

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

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

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

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

QMan: Default Scheduling

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

NOTE

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

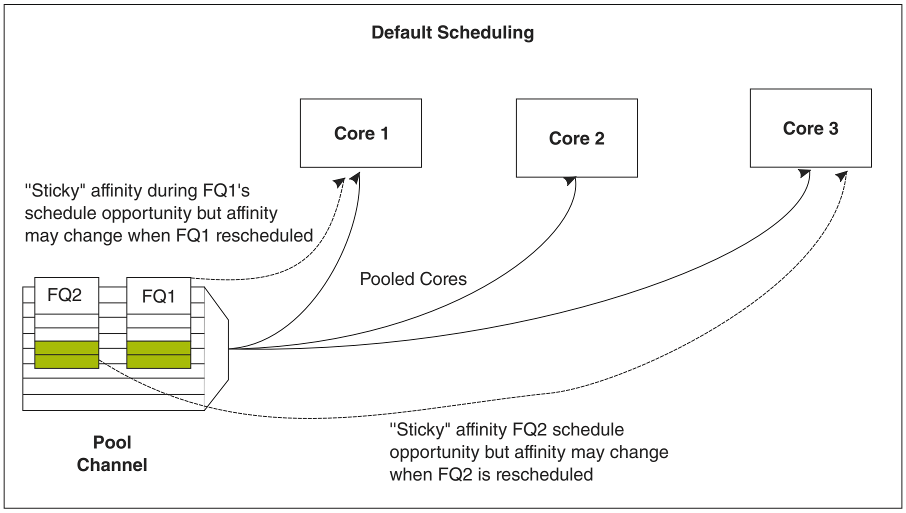

Default Scheduling
Core 1
Core 2
Core 3
"Sticky" affinity during FQ1's schedule opportunity but affinity may change when FQ1 rescheduled
Pooled Cores
FQ2
FQ1
Pool Channel
"Sticky" affinity FQ2 schedule opportunity but affinity may change when FQ2 is rescheduled

Figure 73. 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.

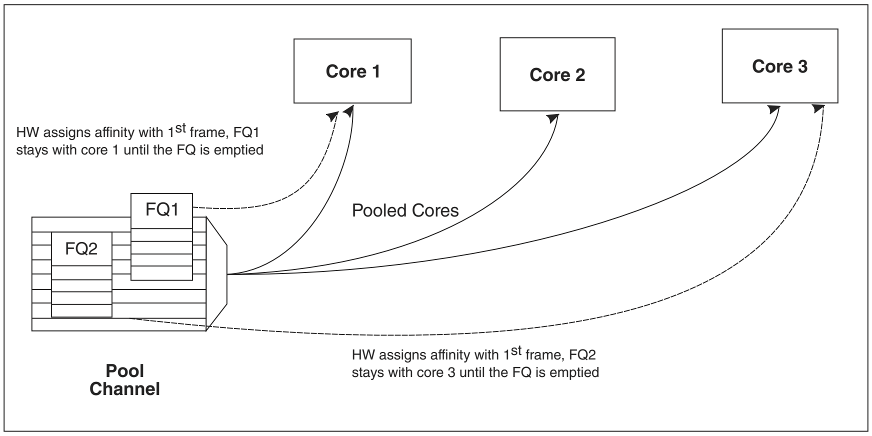

Core 1
Core 2
Core 3
HW assigns affinity with 1st frame, FQ1 stays with core 1 until the FQ is emptied
FQ1
FQ2
Pooled Cores
HW assigns affinity with 1st frame, FQ2 stays with core 3 until the FQ is emptied
Pool Channel

Figure 74. 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.

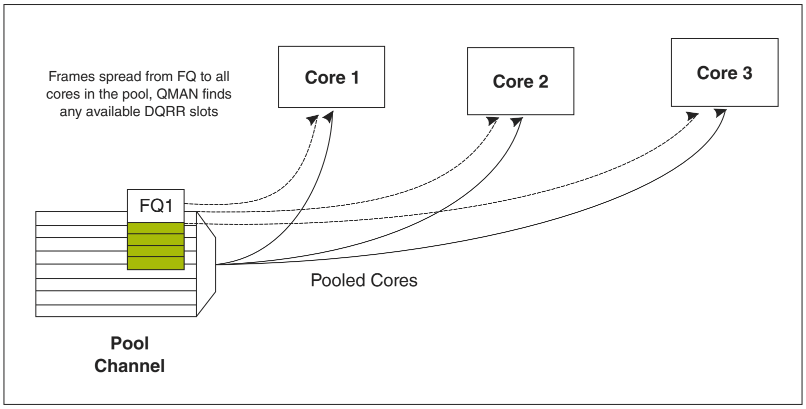

Frames spread from FQ to all cores in the pool, QMAN finds any available DQRR slots
Core 1
Core 2
Core 3
FQ1
Pooled Cores
Pool Channel

Figure 75. 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.

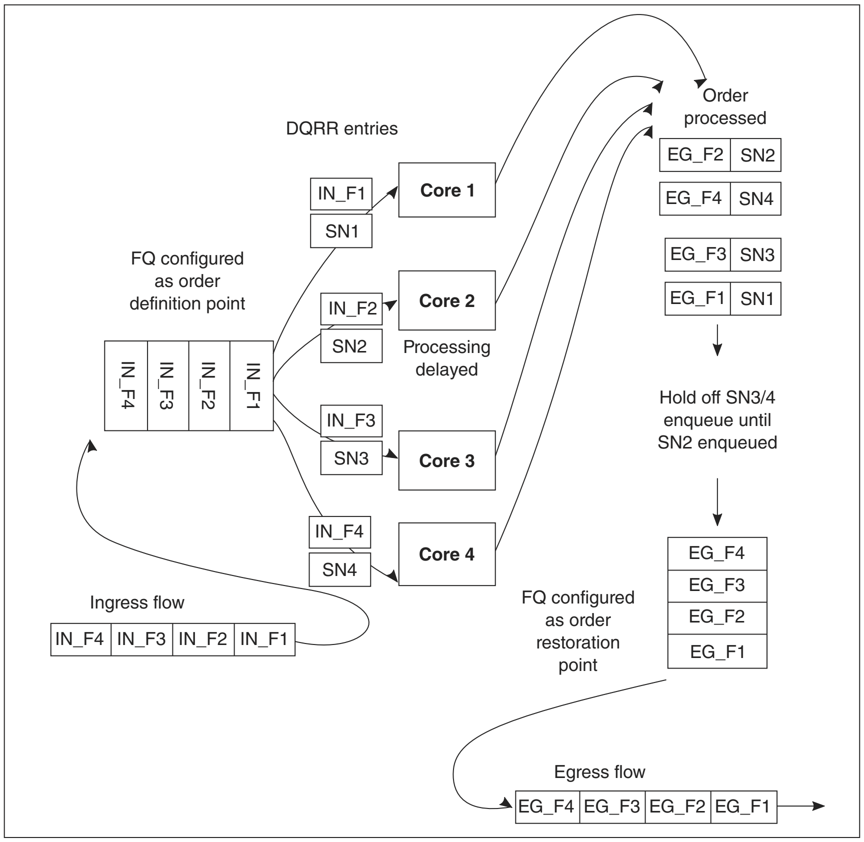


Figure 76. Order Definition/Restoration

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.

8.2.1.6 BMan

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

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

The BMan also supports single or multi-buffer frames. Single buffer frames generally require the adequately defined (or allocated) buffer size to contain the largest data frame and minimize system overhead. Multi-buffer frames potentially allow better memory utilization, but the entity passed between the producers/consumers is a scatter-gather table (that then points to the buffers within the frame) rather than the pointer to the entire frame, which adds an extra processing requirement to the processing element.

The software defines pools of buffers when the system is initialized. The BMan unit itself manages the pointers to the buffers provided by the 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).

8.2.1.7 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.

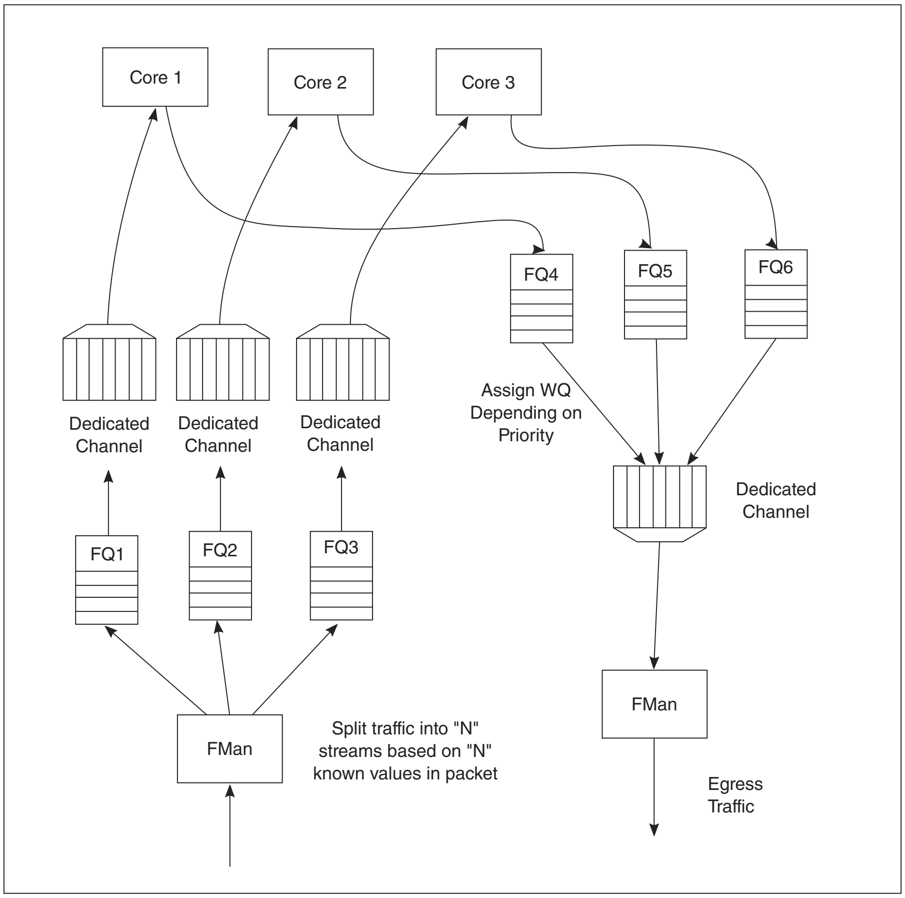


Figure 77. Direct Flow-to-Core Mapping (Order Preserved)

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.

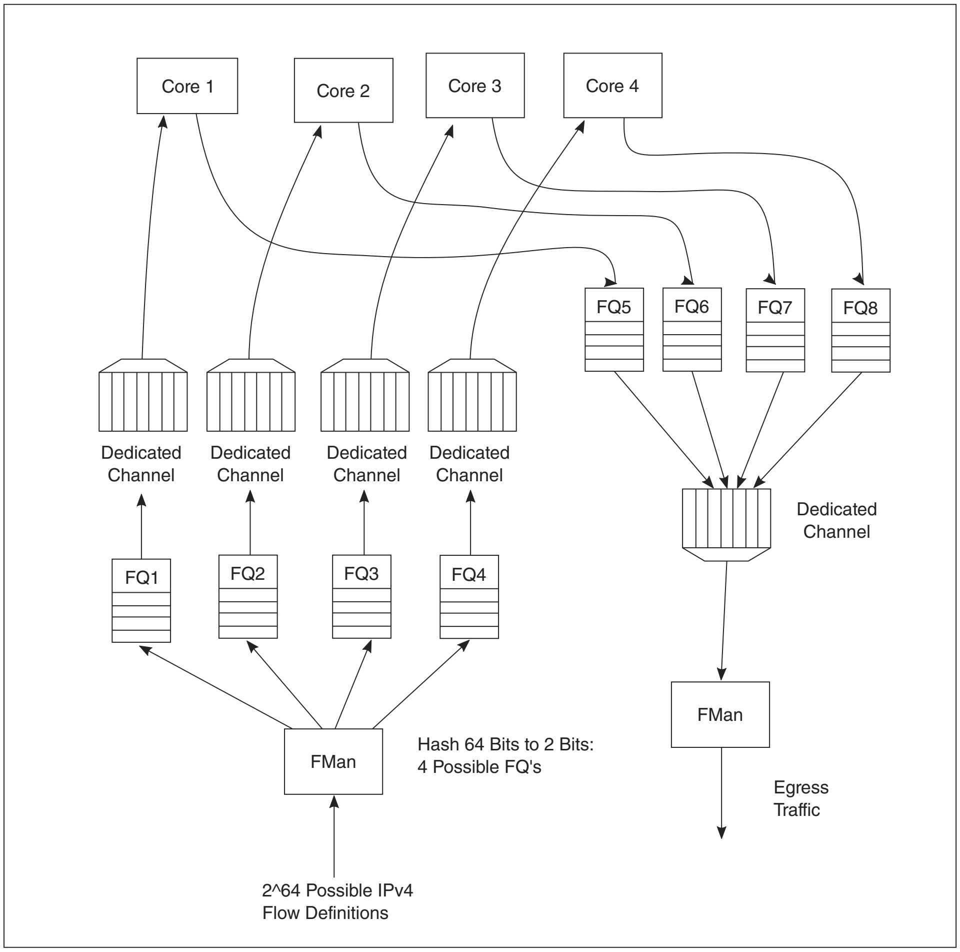


Figure 78. Simple flow distribution via hash (order preserved)

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.

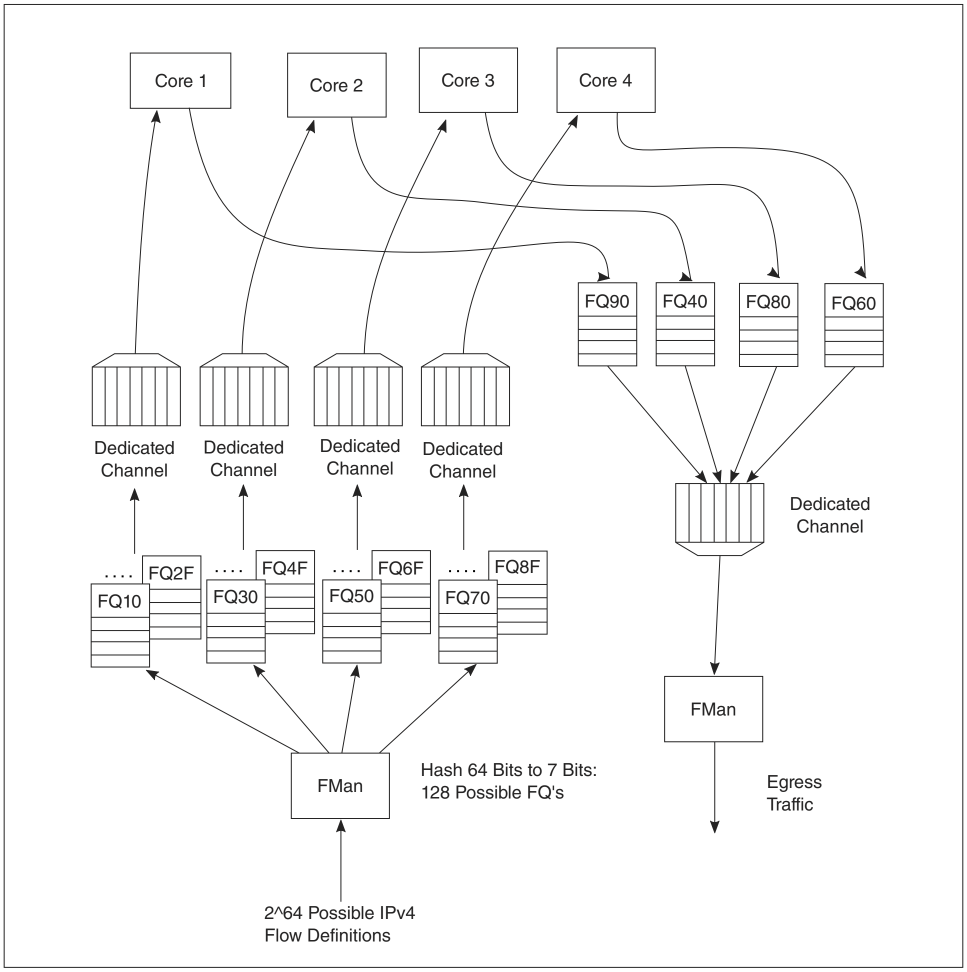


Figure 79. Using hash to assign one flow per FQ (order preserved and cache stashing effective)

8.2.1.8 Pool Channels

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

So far, the techniques discussed in this white paper have involved assigning specific flows to the same core to ensure that the same core always processes the same flow or set of flows, thereby preserving flow order. However, depending on the nature of the flows being processed (that is, variable frame sizes, difficulty efficiently spreading due to the nature of the flow contents, and so on), this may not effectively balance the processing load among the cores. Alternatively, a user may employ a pool channel approach where multiple cores may pool together to service a specific set of flows. This alternative approach allows potentially better processing balance, but increases the likelihood that packets may be processed out of order allowing egress packets to pass ingress packets. When the application does not require flows to be processed in order, the pool channel approach allows the easiest method for balancing the processing load. When a pool channel is used and order is required, the software must maintain order. The hardware order preservation may be used by the software to implement order without requiring locked access to shared state information. When the system uses a software lock to handle order then the default scheduling and hold active scheduling tends to minimize lock contention.

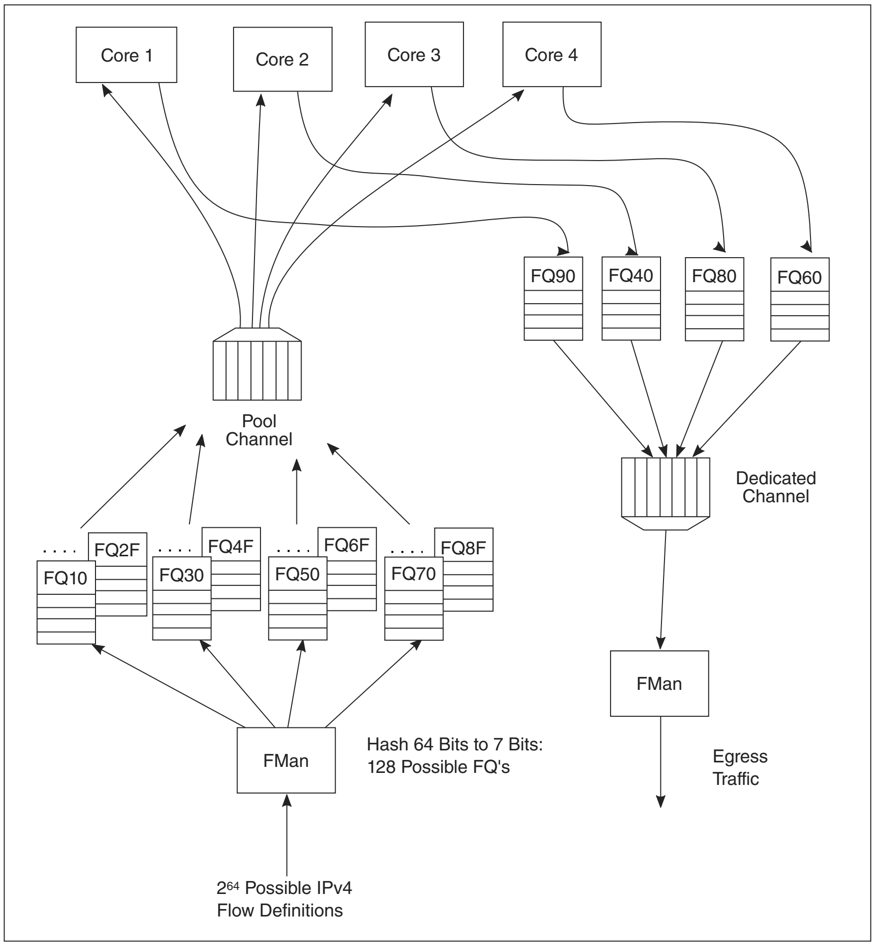


Figure 80. Using pool channel to balance processing

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.

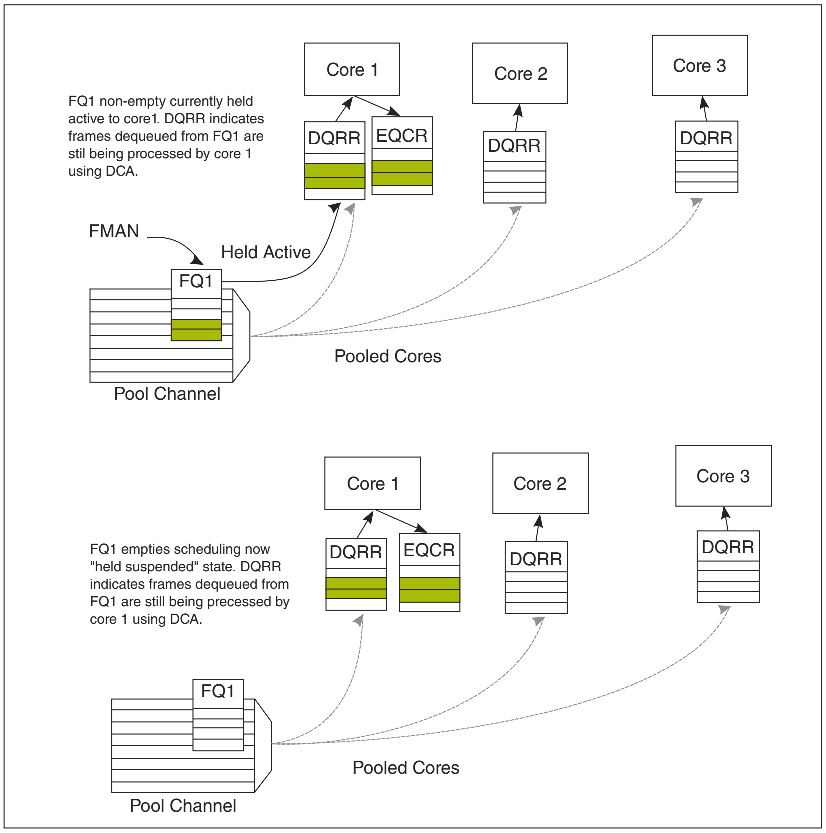

Core 1
Core 2
Core 3
FQ1 non-empty currently held active to core1. DQRR indicates frames dequeued from FQ1 are stil being processed by core 1 using DCA.
DQRR
EQCR
DQRR
DQRR
FMAN
Held Active
FQ1
Pooled Cores
Pool Channel
Core 1
Core 2
Core 3
FQ1 empties scheduling now "held suspended" state. DQRR indicates frames dequeued from FQ1 are still being precessed by core 1 using DCA.
DQRR
EQCR
DQRR
DQRR
FQ1
Pooled Cores
Pool Channel

Figure 81. Hold active to held suspended mode

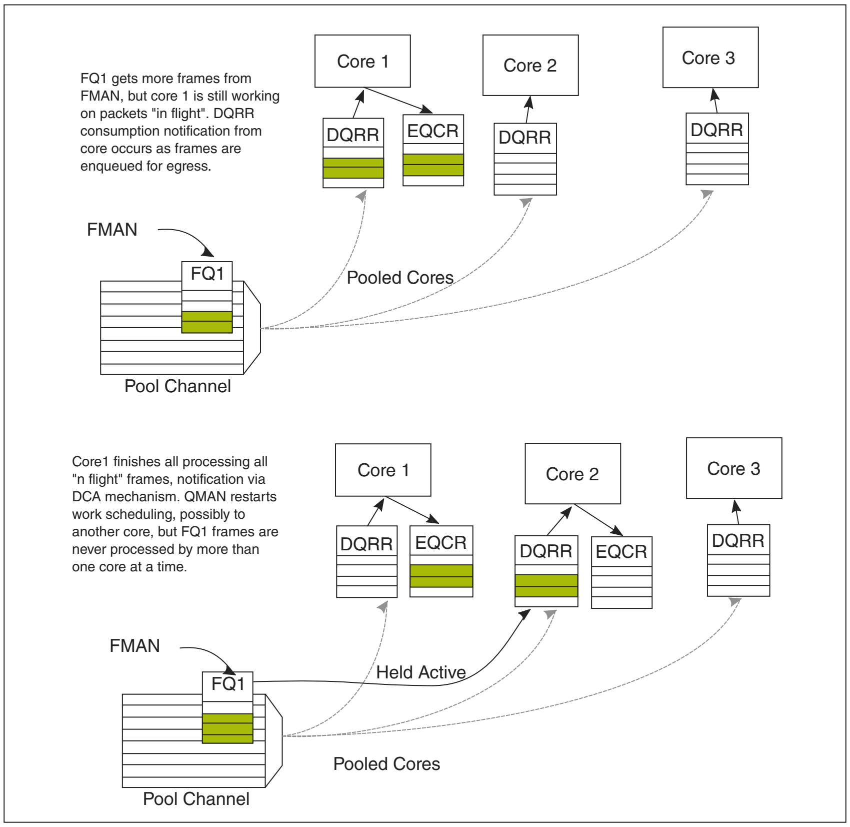

FQ1 gets more frames from FMAN, but core 1 is still working on packets "in flight". DQRR consumption notification from core occurs as frames are enqueued for egress.
Core 1
Core 2
Core 3
DQRR
EQCR
DQRR
DQRR
FMAN
FQ1
Pooled Cores
Pool Channel
Core1 finishes all processing all "n flight" frames, notification via DCA mechanism. QMAN restarts work scheduling, possibly to another core, but FQ1 frames are never processed by more than one core at a time.
Core 1
Core 2
Core 3
DQRR
EQCR
DQRR
EQCR
DQRR
FMAN
FQ1
Held Active
Pooled Cores
Pool Channel

Figure 82. 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.

8.2.1.9 Application Mapping

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

Processor core assignment

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

Define flows

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

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

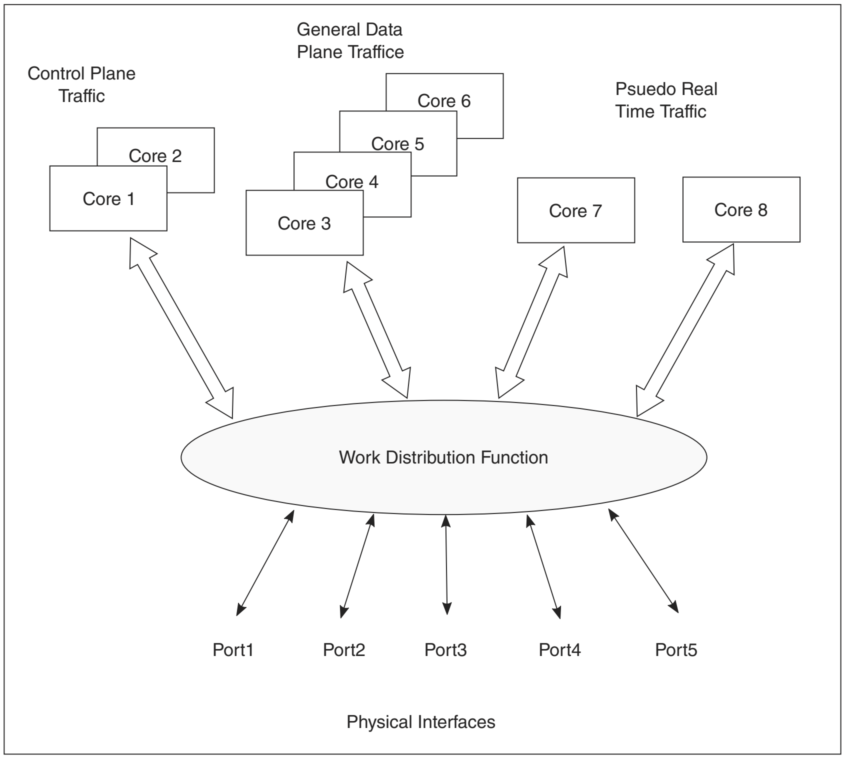


Figure 83. Example Application with Three Classes

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 88. 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 88. 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.

8.2.1.10 FQ/WQ/Channel

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

Consider the following by class of traffic:

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

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

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

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

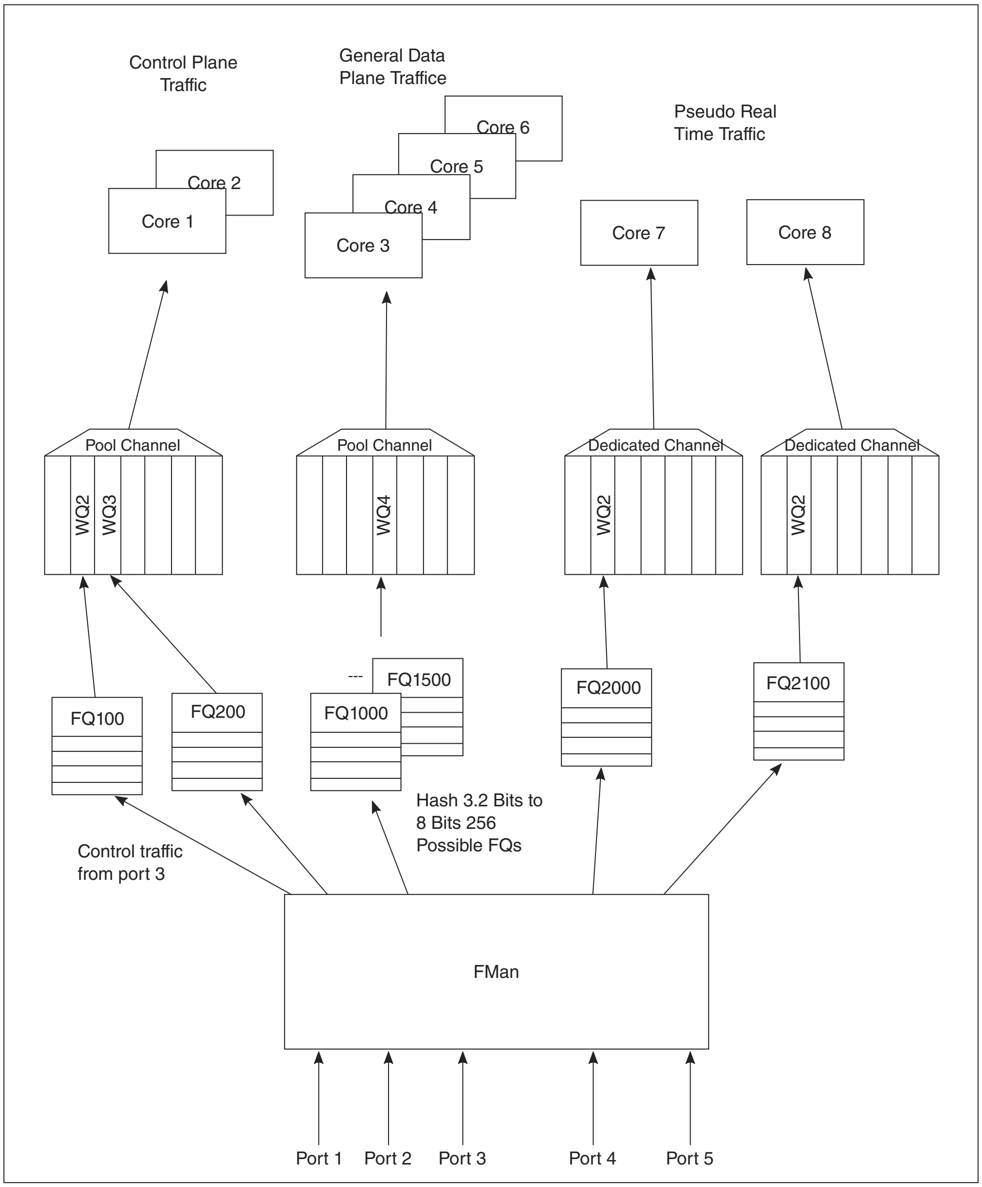


Figure 84. Ingress application map

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.

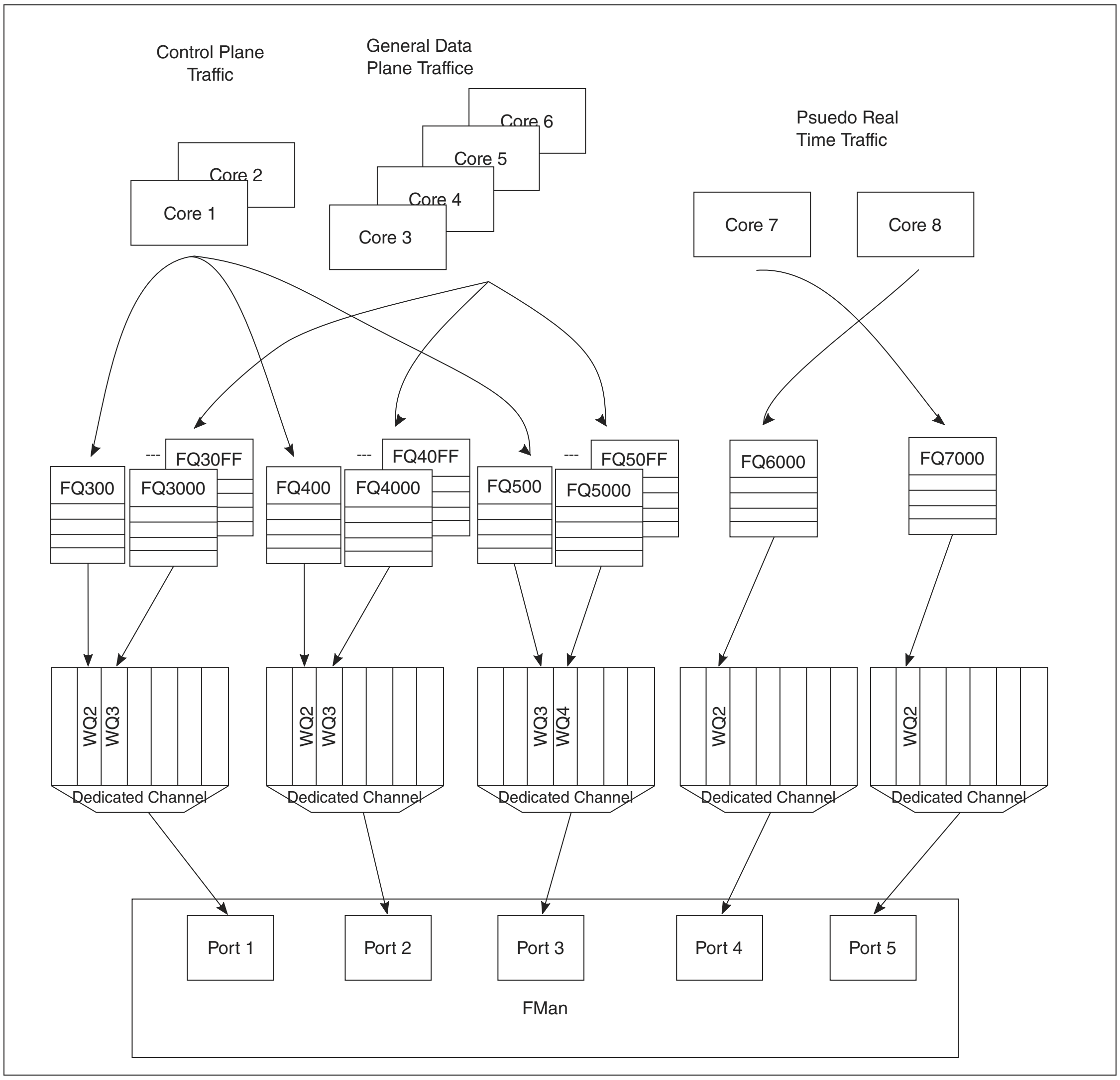


Figure 85. Egress application map

End of Document

8.2.2 Linux Ethernet

8.2.2.1 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.

8.2.2.2 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.

8.2.2.2.1 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.

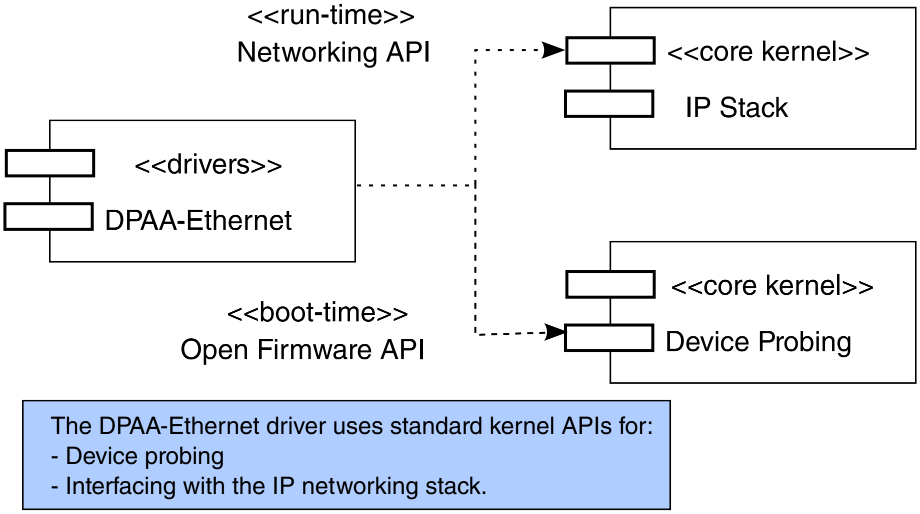


Figure 86. Platform device tree

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

8.2.2.2.2 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

8.2.2.2.2.1 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:

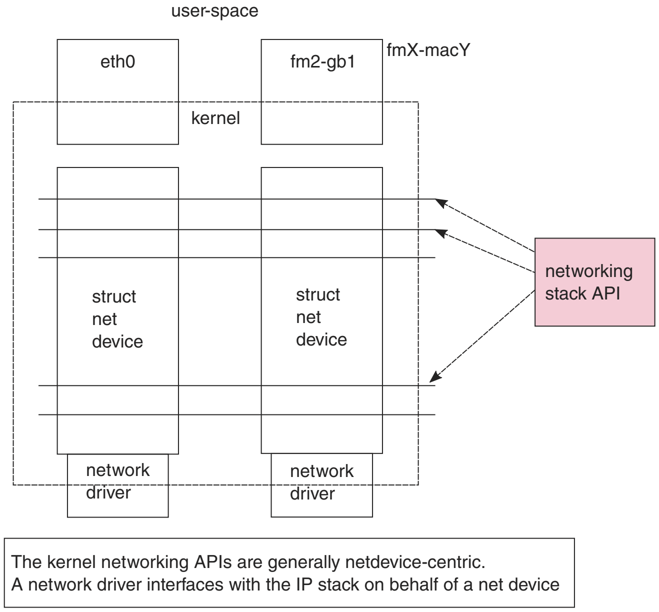

user-space
eth0
fm2-gb1
fmX-macY
kernel
struct net device
struct net device
networking stack API
network driver
network driver
The kernel networking APIs are generally netdevice-centric.
A network driver interfaces with the IP stack on behalf of a net device

Figure 87. Every Linux Interface is Represented by a Net Device

8.2.2.2.2.2 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.

8.2.2.2.2.3 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.

8.2.2.3 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.

8.2.2.3.1 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`.

8.2.2.3.2 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 455 section.

8.2.2.4 The (Simplified) Life of a packet

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

8.2.2.4.1 Private net device: Tx

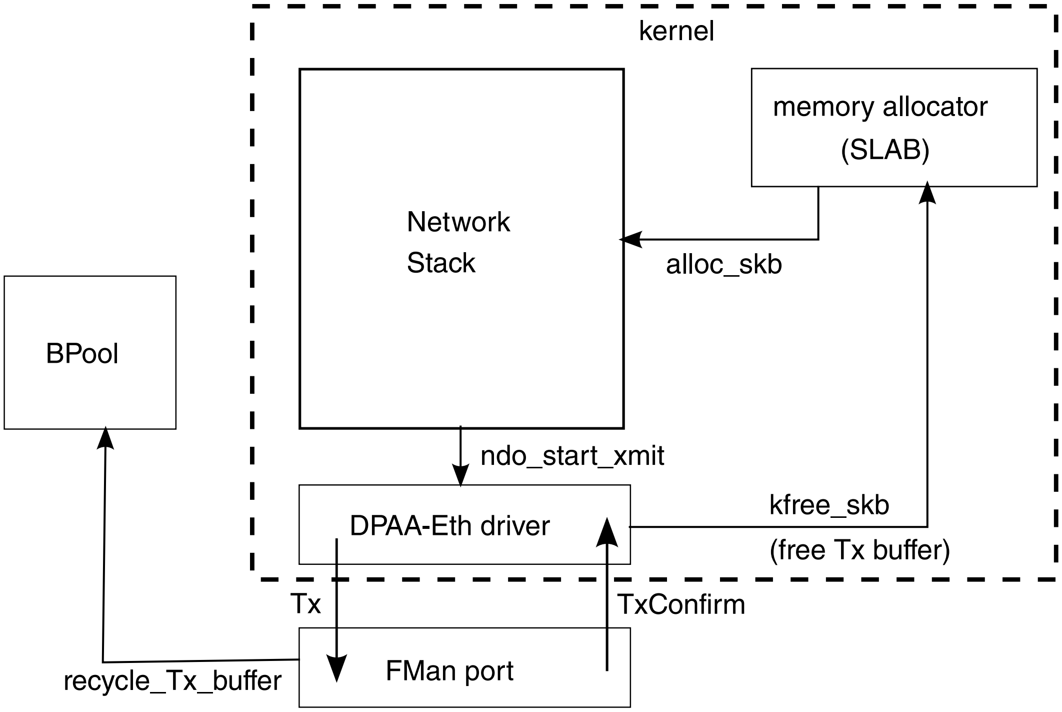


Figure 88. Buffers on the egress path

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.

8.2.2.4.2 Private net device: Rx

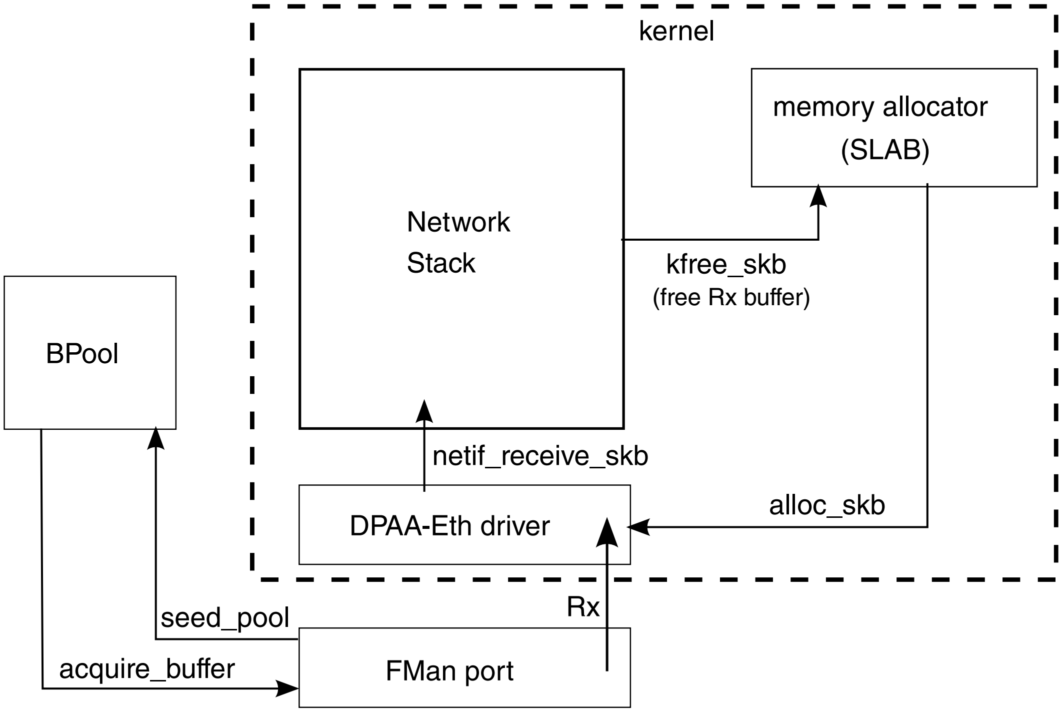


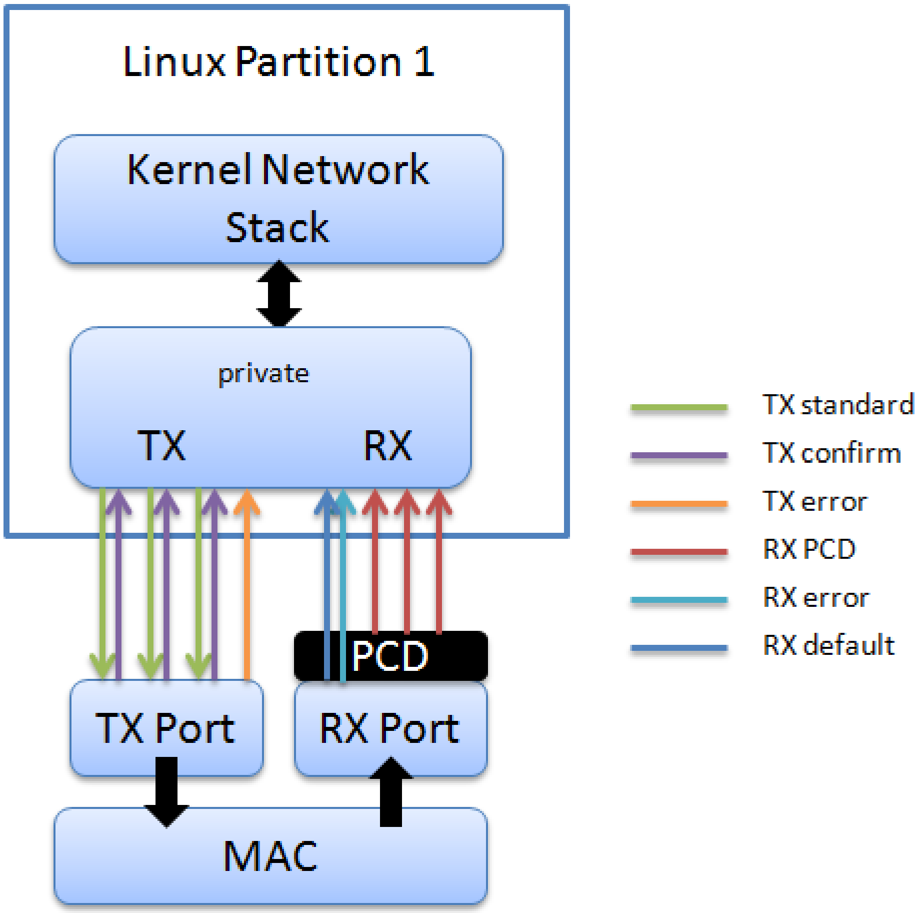
Figure 89. Buffers on the ingress path

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.

8.2.2.5 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.

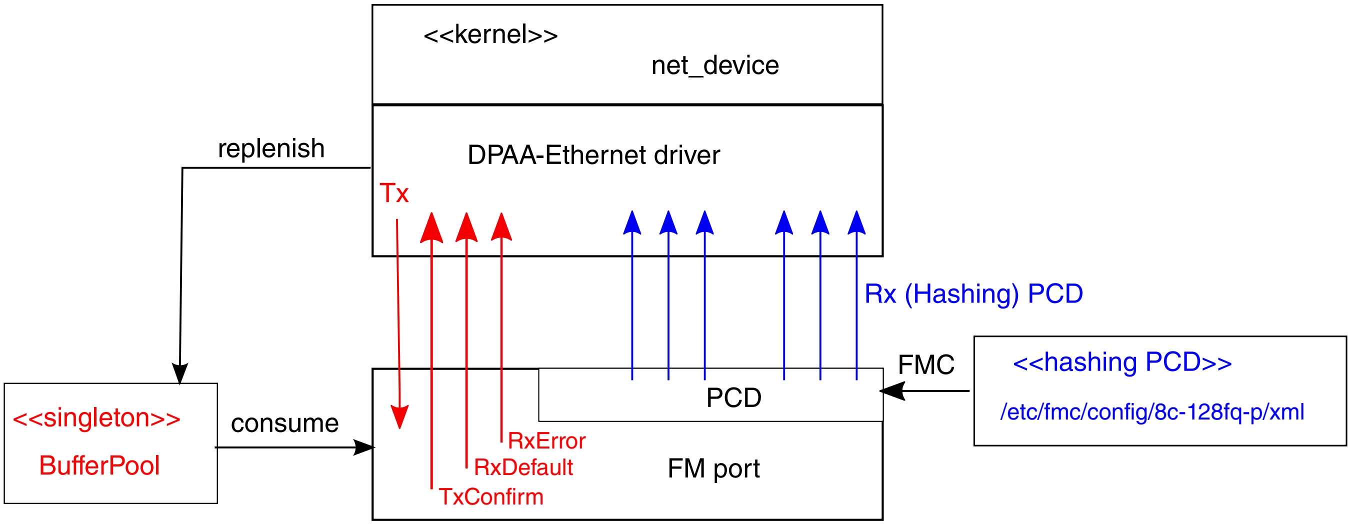


8.2.2.5.1 Network driver

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

8.2.2.5.2 Configuration

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

8.2.2.5.2.1 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.

8.2.2.5.2.2 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.

8.2.2.5.2.3 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.

8.2.2.5.2.4 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.
```

8.2.2.5.3 Features

This section present the private DPAA1 ethernet driver features.

8.2.2.5.3.1 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 447.

8.2.2.5.3.2 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.

8.2.2.5.3.3 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 448.

8.2.2.5.3.4 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.

8.2.2.5.3.5 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.

8.2.2.5.3.6 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 89. 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.

8.2.2.5.3.7 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.

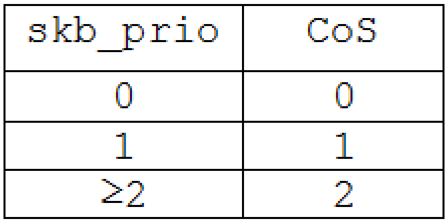
Enabling and disabling CoS and their pause time is unavailable at runtime. It is only possible at compile time in this release.

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
```

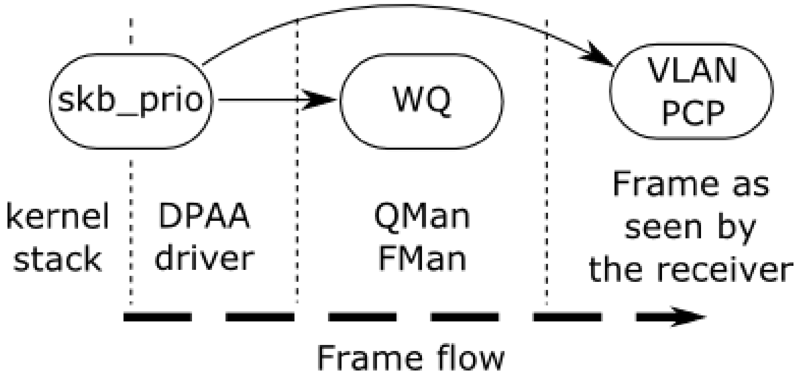
VLAN tagging

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

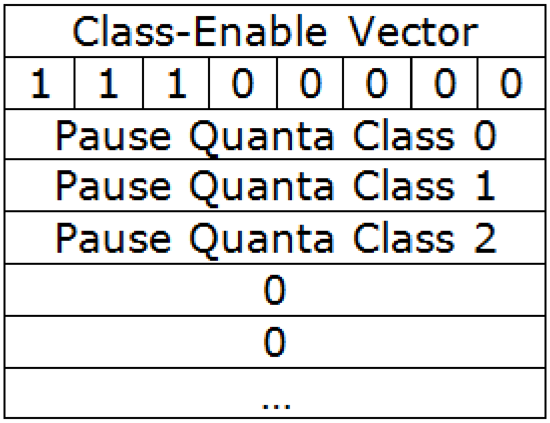
All DPAA1 Private interfaces share a single buffer pool which accounts for the buffers in which the frames are stored upon receiving.

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
```

8.2.2.5.3.8 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 90. 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 91. 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

Table continues on the next page...

Table 91. FMan devices high priority core affined queues (continued)

Interface	FQID base	LS1043A	LS1046A
fm1-mac10	0x13c80		Y

8.2.2.5.4 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.

8.2.2.5.4.1 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 611.

8.2.2.5.4.2 Scheduling and Shaping

8.2.2.5.4.2.1 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.

8.2.2.5.4.2.1.1 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.

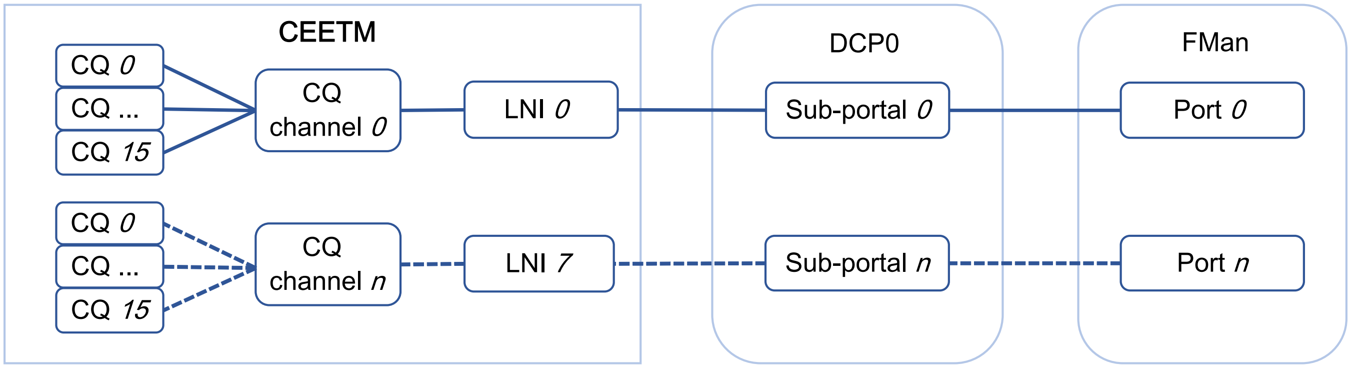


Figure 91. CEETM block

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*.

8.2.2.5.4.2.1.2 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:

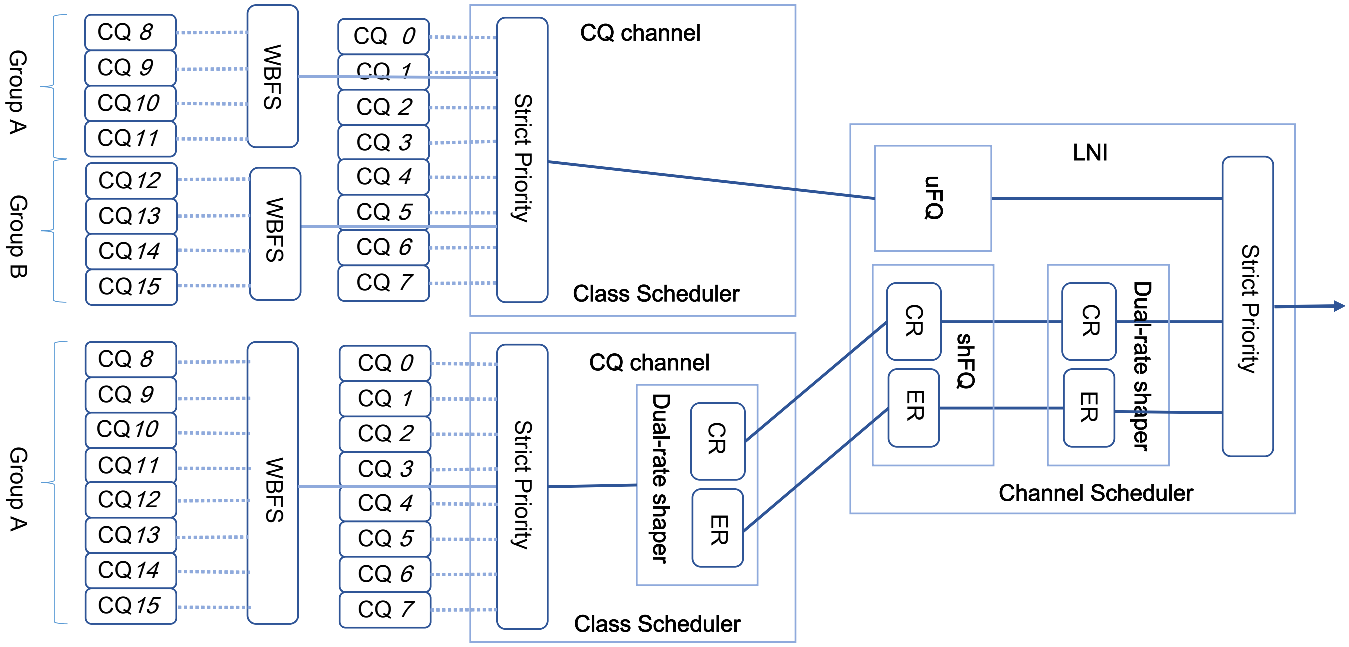


Figure 92. CEETM architecture

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.

8.2.2.5.4.2.1.3 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:

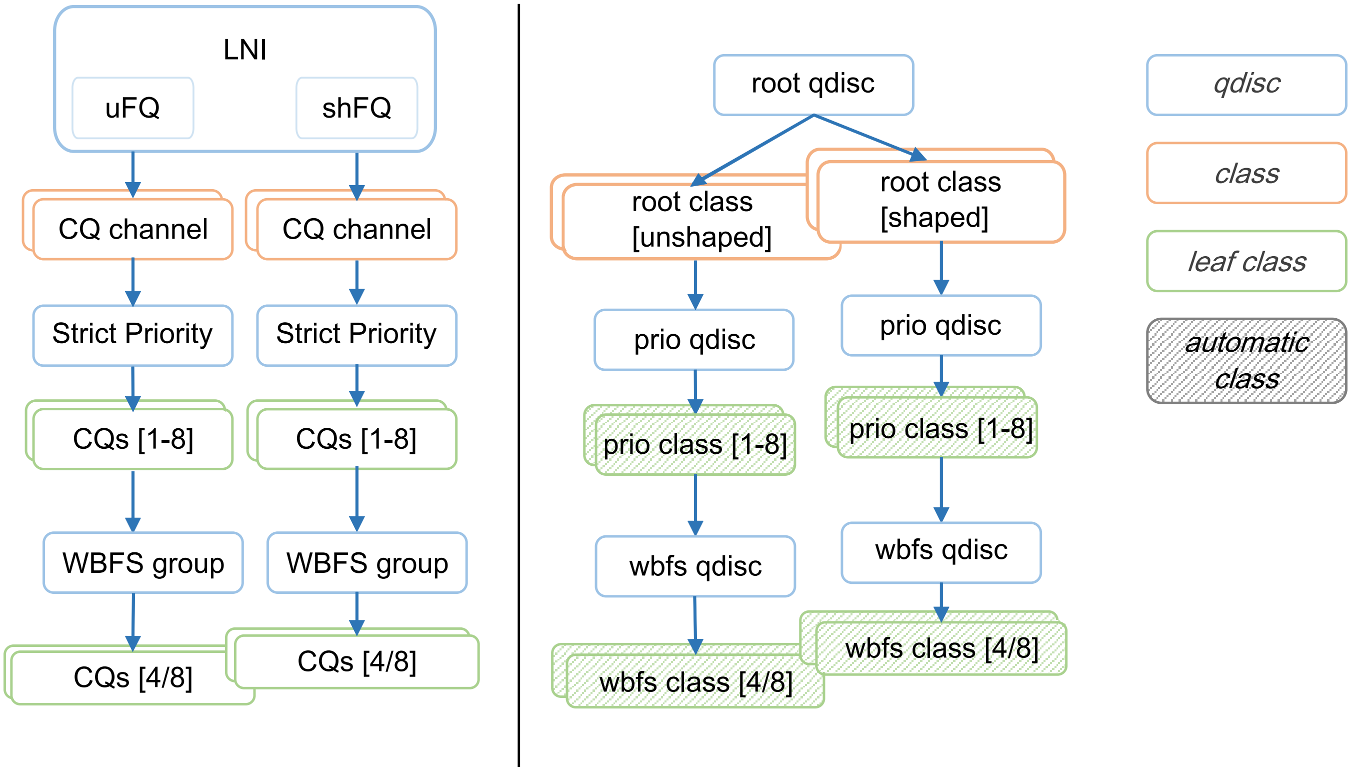


Figure 93. Comparison between CEETM and tc terminology

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 460 section.

8.2.2.5.4.2.2 User guide

8.2.2.5.4.2.2.1 Supported platforms

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

8.2.2.5.4.2.2.2 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
```

8.2.2.5.4.2.2.3 Limitations

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

8.2.2.5.4.2.2.4 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.

8.2.2.5.4.2.3 Examples

8.2.2.5.4.2.3.1 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 94. on page 461.

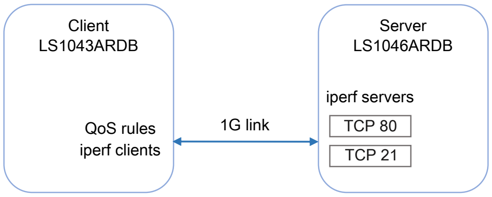


Figure 94. Rate example setup

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 95. on page 462.

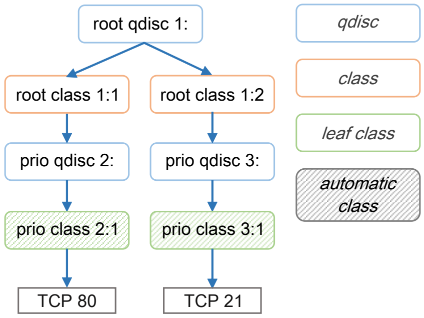


Figure 95. Rate example class hierarchy

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
```

8.2.2.5.4.2.3.2 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:

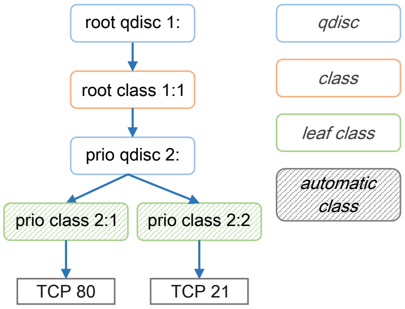


Figure 96. Prioritization example class hierarchy

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
```

8.2.2.5.4.2.3.3 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:

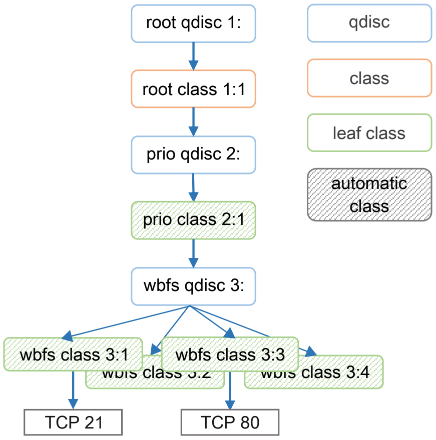


Figure 97. WBFS example class hierarchy

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
```

8.2.2.5.4.2.3.4 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:

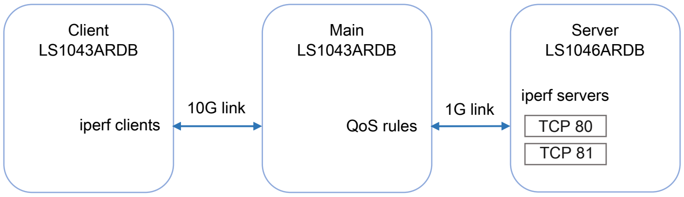


Figure 98. Unshaped Fair Queuing example setup

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 99. on page 465.

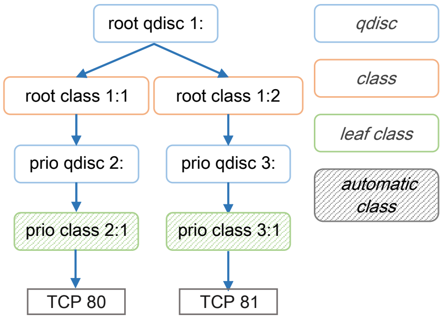


Figure 99. Unshaped Fair Queuing example class hierarchy

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
```

8.2.2.5.5 Debugging

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

8.2.2.5.5.1 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
```

8.2.2.5.5.2 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.

8.2.2.5.5.3 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
[...]
```

8.2.2.5.6 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
   ```

8.2.2.5.7 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.

8.2.2.6 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-ls1043a-rdb.dts - used by the Upstream Ethernet driver
fsl-ls1043a-rdb-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-ls1043a-rdb-sdk.dtb - build the DTB for the Private Ethernet driver
make freescale/fsl-ls1043a-rdb.dtb - build the DTB for the Upstream Ethernet driver
```

Limitations

A workaround for the LS1043A FMan A010022 errata is integrated into the Upstream Ethernet driver. Egress Scatter Gather frames can not be used on this platform.

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

8.2.3.1 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 477”).

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.

Table continues on the next page...

Table continued from the previous page...

Common Kernel Configure Options	Description
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.
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
CONFIG_FSL_BMAN_CONFIG	Handles config/CCSR nodes in the device-tree and initialises the corresponding devices

Table continues on the next page...

Table continued from the previous page...

BMan Kernel Configure Options	Description
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

Table continues on the next page...

Table continued from the previous page...

Source Files	Description
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.

8.2.3.2 QMan BMan API Reference

8.2.3.2.1 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 521for more details.

8.2.3.2.2 Buffer Manager

8.2.3.2.2.1 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.

8.2.3.2.2.2 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.

8.2.3.2.2.2.1 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.

8.2.3.2.2.2.1.1 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.

8.2.3.2.2.2.1.2 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.

8.2.3.2.2.2.2 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>;
        };
        [...]
        };
```

8.2.3.2.2.2.2.1 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.

8.2.3.2.2.2.2.2 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].

8.2.3.2.2.2.2.3 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.

8.2.3.2.2.2.3 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.

8.2.3.2.2.2.3.1 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.

8.2.3.2.2.2.3.2 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.

8.2.3.2.3 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 480.

8.2.3.2.3.1 BMan High-Level Portal Interface

8.2.3.2.3.1.1 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.

8.2.3.2.3.1.2 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 480. 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);
```

8.2.3.2.3.1.2.1 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 481. 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);
```

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

If portal-sharing is in effect, refer to Portal sharing on page 481. 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);
```

8.2.3.2.3.1.2.3 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);
```

8.2.3.2.3.1.2.4 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);
```

8.2.3.2.3.1.3 Pool Management

To work with BMan buffer pools, a pool object must be created. As explained in Depletion State on page 485, 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);
```

8.2.3.2.3.1.4 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);
```

8.2.3.2.3.1.5 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 483, 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) [...]
```

8.2.3.2.4 Queue Manager

8.2.3.2.4.1 QMan Overview

8.2.3.2.4.1.1 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.

8.2.3.2.4.1.2 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.

8.2.3.2.4.1.3 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.

8.2.3.2.4.1.4 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.

8.2.3.2.4.1.5 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.

8.2.3.2.4.1.6 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.

8.2.3.2.4.1.7 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).

8.2.3.2.4.1.8 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.

8.2.3.2.4.1.9 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".

8.2.3.2.4.1.10 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.

8.2.3.2.4.1.10.1 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 489).

8.2.3.2.4.1.10.2 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";

8.2.3.2.4.1.10.3 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.

8.2.3.2.4.1.10.4 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.

8.2.3.2.4.1.10.5 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.

8.2.3.2.4.1.11 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.

8.2.3.2.4.1.12 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.

8.2.3.2.4.1.12.1 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".

8.2.3.2.4.1.12.2 Parking Scheduled FQs

As noted above in Dequeue Atomicity on page 490, 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.

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

8.2.3.2.4.1.13 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.

8.2.3.2.4.1.14 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 501, 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 497, 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.

8.2.3.2.4.2 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.

8.2.3.2.4.2.1 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.

8.2.3.2.4.2.1.1 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 478.

8.2.3.2.4.2.1.2 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 478.

8.2.3.2.4.2.1.3 Logical I/O Device Number (QMan)

This property is the same as described in Logical I/O Device Number (BMan) on page 479, but for use by QMan when accessing FQD and PFDR memory (rather than BMan's FBPR memory).

8.2.3.2.4.2.2 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 487. 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>;
        };
```

8.2.3.2.4.2.2.1 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>;
```

8.2.3.2.4.2.3 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 487 and Portal Sub-Interfaces on page 487. 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 493, the other QMan-specific portal properties are described below.

8.2.3.2.4.2.3.1 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>;
```

8.2.3.2.4.2.3.2 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 489. 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>;
```

8.2.3.2.4.2.3.3 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 481 for an explanation of this behaviour in the BMan documentation, the QMan behaviour is identical.

8.2.3.2.4.2.3.4 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 481 for an explanation of this behavior in the BMan documentation. The QMan behavior is identical.

8.2.3.2.5 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 493, and described in Portals on page 487 and Portal Sub-Interfaces on page 487.

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.

8.2.3.2.5.1 QMan High-Level Portal Interface

8.2.3.2.5.1.1 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.

8.2.3.2.5.1.2 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
};
```

8.2.3.2.5.1.3 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 493. 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);
```

8.2.3.2.5.1.3.1 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 481), 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);
```

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

If portal-sharing is in effect (see Portal sharing on page 481), 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);
```

8.2.3.2.5.1.3.3 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);
```

8.2.3.2.5.1.3.4 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);
```

8.2.3.2.5.1.3.5 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);
```

8.2.3.2.5.1.3.6 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);
```

8.2.3.2.5.1.4 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);
```

8.2.3.2.5.1.4.1 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);
```

8.2.3.2.5.1.4.2 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);
```

8.2.3.2.5.1.4.3 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);
```

8.2.3.2.5.1.4.4 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);
```

8.2.3.2.5.1.4.5 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);
```

8.2.3.2.5.1.4.6 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);
```

8.2.3.2.5.1.4.7 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)
```

8.2.3.2.5.1.4.8 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);
```

8.2.3.2.5.1.5 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);
```

8.2.3.2.5.1.6 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);
```

8.2.3.2.5.1.7 DCA Mode

As described in Order Preservation & Discrete Consumption Acknowledgement on page 491, 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 501, 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);
```

8.2.3.2.5.1.8 Congestion Management Records

QMan supports a fixed number[12] 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.

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;
};
```

[12] 256 for P4080/P5020/P3041

```
/* 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);
```

8.2.3.2.5.1.9 Zero-Configuration Messaging

As described in Overview (QMan) on page 494, 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 499).

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);
```

8.2.3.2.5.1.10 FQ allocation

8.2.3.2.5.1.10.1 Ad-hoc FQ allocator

As described in Seeding Buffer Pools on page 480>, BMan buffer pool ID zero is currently reserved for use as an ad-hoc FQ allocator. As seen in Frame queue management on page 497, 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).

8.2.3.2.5.1.10.2 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);
```

8.2.3.2.5.1.10.3 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.

8.2.3.2.5.1.11 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;
}
```

8.2.3.2.6 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.

8.2.3.2.6.1 QMan sysfs

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

Description:

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

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

Description:

This directory contains a snapshot of error related qman attributes.

8.2.3.2.6.1.3 /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.

8.2.3.2.6.1.4 /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.

8.2.3.2.6.1.5 /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

8.2.3.2.6.1.6 /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.

8.2.3.2.6.1.7 /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.

8.2.3.2.6.1.8 /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.

8.2.3.2.6.1.9 /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

8.2.3.2.6.1.10 /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

8.2.3.2.6.2 BMan sysfs

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

Description:

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

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

Description:

This directory contains a snapshot of error related BMan attributes.

8.2.3.2.6.2.3 /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.

8.2.3.2.6.2.4 /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.

8.2.3.2.6.2.5 /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

8.2.3.2.6.2.6 /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.

8.2.3.2.6.3 QMan debugfs

8.2.3.2.6.3.1 /sys/kernel/debug/qman

Description:

This directory contains various QMan device debugging attributes.

8.2.3.2.6.3.2 /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
```

8.2.3.2.6.3.3 /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.
```

8.2.3.2.6.3.4 /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
```

8.2.3.2.6.3.5 /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
```

8.2.3.2.6.3.6 /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
```

8.2.3.2.6.3.7 /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

8.2.3.2.6.3.8 /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

8.2.3.2.6.3.9 /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
```

8.2.3.2.6.3.10 /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
```

8.2.3.2.6.3.11 /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
```

8.2.3.2.6.3.12 /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
```

8.2.3.2.6.3.13 /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
```

8.2.3.2.6.3.14 /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
```

8.2.3.2.6.3.15 /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
```

8.2.3.2.6.3.16 /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
```

8.2.3.2.6.3.17 /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
```

8.2.3.2.6.3.18 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
```

8.2.3.2.6.3.19 /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
```

8.2.3.2.6.3.20 /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
```

8.2.3.2.6.3.21 /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
```

8.2.3.2.6.3.22 /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
```

8.2.3.2.6.3.23 /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
```

8.2.3.2.6.4 BMan debugfs

8.2.3.2.6.4.1 /sys/kernel/debug/bman

Description:

This directory contains various BMan device debugging attributes.

8.2.3.2.6.4.2 /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

8.2.3.2.7 Error handling and reporting

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

8.2.3.2.7.1 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).

8.2.3.2.8 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.

8.2.3.2.8.1 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 482 (for BMan) and Processing non-interrupt-driven portal duties (QMan) on page 496 (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.

8.2.3.2.8.2 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().

8.2.3.2.8.3 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.

8.2.4 Configuring DPAA1 Frame Queues

8.2.4.1 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.

8.2.4.2 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.

8.2.4.3 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.

8.2.4.4 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.

8.2.4.5 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.

8.2.4.6 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.

8.2.4.7 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.

8.2.4.8 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.

.

8.2.4.9 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.

8.2.4.10 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 92. 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 93. 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 93. 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 94. 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 95. 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 .

8.2.5 Frame Manager

8.2.5.1 Frame Manager Linux Driver User Guide

8.2.5.1.1 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:

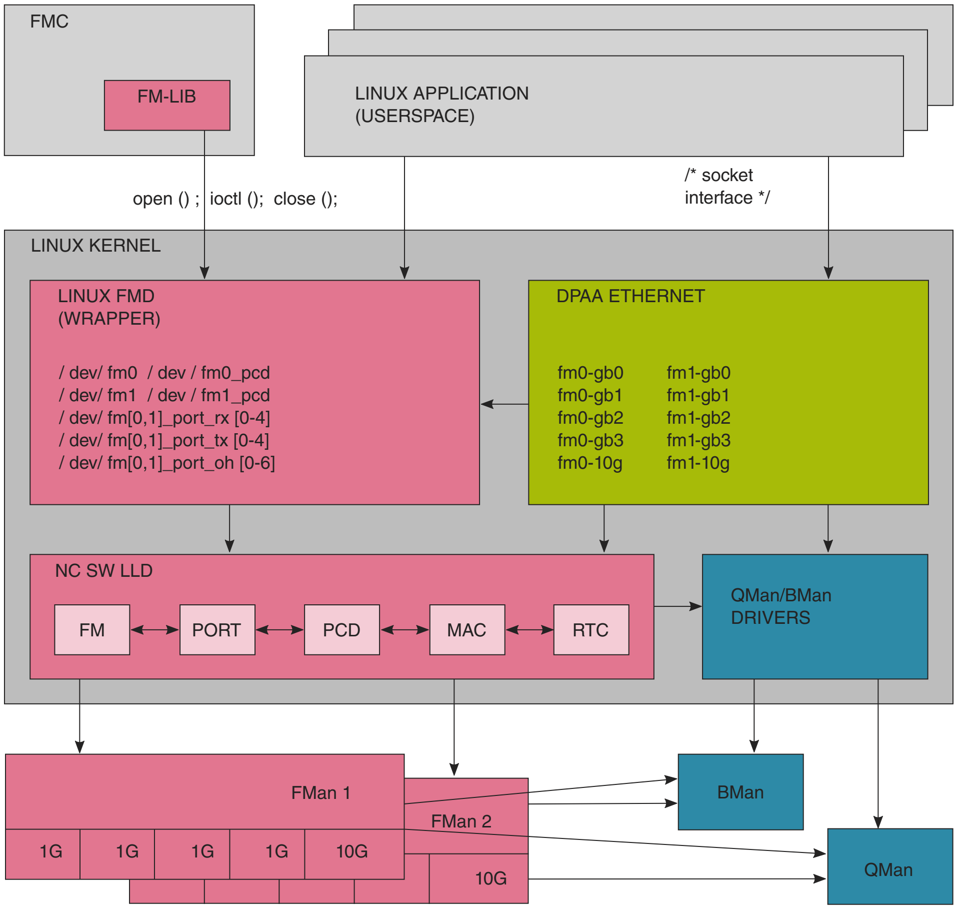


Figure 100. FMan-centric view of relationships between DPAA software and hardware blocks in the Linux environment.

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 439 section.

8.2.5.1.2 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 96. 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 96. 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.

8.2.5.1.3 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.

8.2.5.1.4 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.

8.2.5.1.4.1 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 97. 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 97. 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.

8.2.5.1.4.2 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 98. 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 98. 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 98. 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 98. 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 99. 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;

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;
	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;

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;
	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;

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.

8.2.5.1.4.3 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 100. 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 101. 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 101. 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 101. 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.

8.2.5.2 Frame Manager Driver User's Guide

8.2.5.2.1 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, then delivered to SoC memory via the FMan DMA. The packet header (max size 256 bytes) is stored and the modules common database structure is allocated. Then the packet is parsed by the parser or by the FMan controller. According to parsing results a key may be extracted by KeyGen, a destination frame-queue-id may be set, the packet may be classified by the FMan controller. in that stage, some offloads may be done like re-assembly, fragmentation, header-manipulation and frame-replication. At the end of the classification and manipulations stage, the packet may be colored by policer. At the end of this process, packets are delivered to SoC memory via the FMan DMA and then are enqueued to a frame queue or dropped. The processing order is Parse-Classify-Distribute (PCD) flow dependant, based on user configurations. Each step is dependant on previous state results. This structure enables flexibility, which efficiently supports many flows.

On Tx the frames are transmitted via the desired MAC with optional checksum generation.

8.2.5.2.2 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.

8.2.5.2.3 Frame Manager Driver Components

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

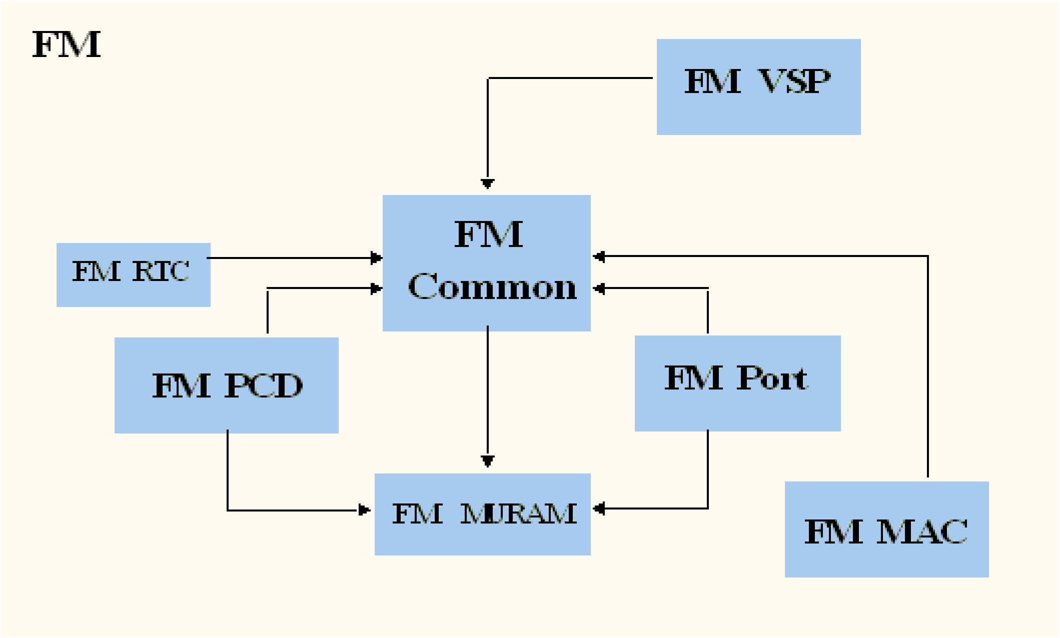


Figure 101. FMan Driver Modules (from a partition point of view)

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 mEMAC dTSEC and the 10G MAC controllers.

  This module can be run by each core or partition independently.
- Frame Manager Virtual-Storage-Profile (FMan-VSP)-This module is responsible for allocating and managing virtual storage profiles that may used for virtualization purposes. More of the VSP is described in FMan VSP Driver on page 586.

  This module can be run by each core or partition independently.

8.2.5.2.4 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).

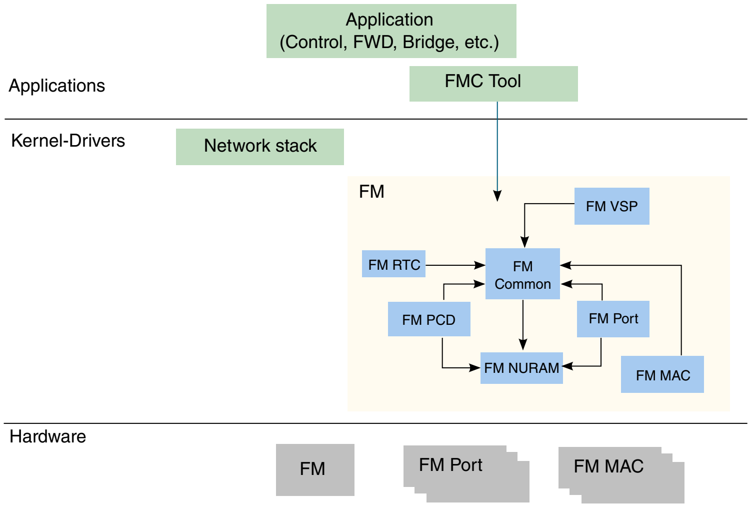


Figure 102. Single-Partition FM Building Blocks

8.2.5.2.4.1 Multicore Approach

The driver supports the Symmetric Multi-Processing (SMP) opperation method.

8.2.5.2.4.1.1 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.

8.2.5.2.5 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. [optional] If the FMan Port required to be virtualized, a set of VSPs need to be allocated and one of them should be set as the default.
    c. [optional] If VSPs were allocated in previous step, the default VSP need to be configured & initialized
    d. in that stage, user should configure and intialize everything that is needed for the operation of a port outside the fman; e.g. buffer-pools, frame-queues, etc.
    e. Port Enablement
    f. MAC Enablement
    g. Calling 'AdjustLink' MAC API routine with the relevant link parameters

NOTE

Now, the FMan is operational. The ports operate in independent mode or BMI-to-BMI mode. From that point, all the following steps are optional.

6. FMan PCD Configuration & Initialization
7. If a physical port is being "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. If VSP is enabled, in that stage, user should configure and initialize the relevant profile.
9. FMan PCD Graph initialization:
    a. Calling restricted runtime routines (that may be called only when PCD is disabled)
    b. Calling the PCD enable routine
    c. Initialization of a all PCD Graph objects (i.e. KG-schemes, Match-Tables, etc)
10. FMan port-PCD related initialization; calling the run-time control routines to set the PCD related parameters

NOTE

In case the PCD is "set" to a FMan OP port, it should be disabled first (i.e. before calling 'FM_PORT_SetPCD' routine).

11. FMan runtime routines
12. FMan Free sequence - in reverse order from initialization

8.2.5.2.6 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.

8.2.5.2.6.1 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.

8.2.5.2.6.1.1 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.

8.2.5.2.6.2 How to use the Global FMan Driver?

The following sections provide practical information for using the software drivers.

8.2.5.2.6.2.1 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

8.2.5.2.6.2.2 Global FMan Driver Sequence

- FMan config routine
- [Optional] FMan advance configuration routines
- FMan Init routine
- FMan runtime routines
- FMan free routine

8.2.5.2.6.2.3 Global FMan Driver Functional Description

The following sections describe main driver functionalities and their usage.

8.2.5.2.6.2.3.1 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.

8.2.5.2.6.2.3.2 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.

8.2.5.2.6.2.3.3 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.

8.2.5.2.6.2.3.4 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.

8.2.5.2.7 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.

8.2.5.2.7.1 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), a Storage-Profile ID (SPID) and policer profile (PP). One main function of the Keygen module is to separate network data into different flows, each requiring different processing. Another function of the Keygen, is the Classification Plan. This is a mechanism provided in order to mask LCV bits according to per-port definition.

The Classification Plan is implemented as a table of SoC dependent number of entries, logically divided or shared between the FMan Ports.

- Custom Classifier-The Frame Manager (FMan) Custom Classifier module performs a look-up using a specific key from the received frame or internal frame context according to Parser results. The FMan Custom Classifier logically occurs after the Keygen processing has completed and can be operational in both the MAC receive flow and the offline parsing flow. The look-up produces an action descriptor which contains the necessary information for the continuation of the frame processing in the next module or the next look-up table.
- 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.

8.2.5.2.7.1.1 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.

8.2.5.2.7.1.1.1 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):

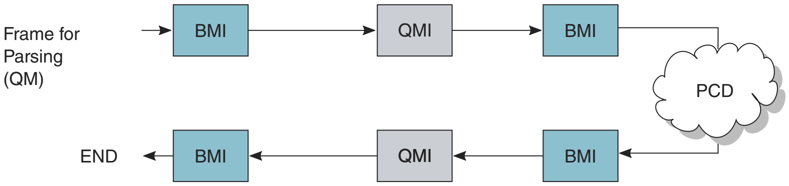


Figure 103. Default Rx Flow

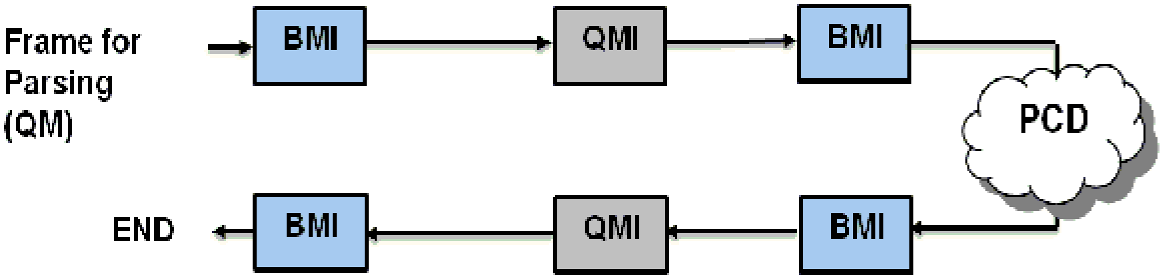


Figure 104. Default Offline Parsing Flow

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 103. on page 553, 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.

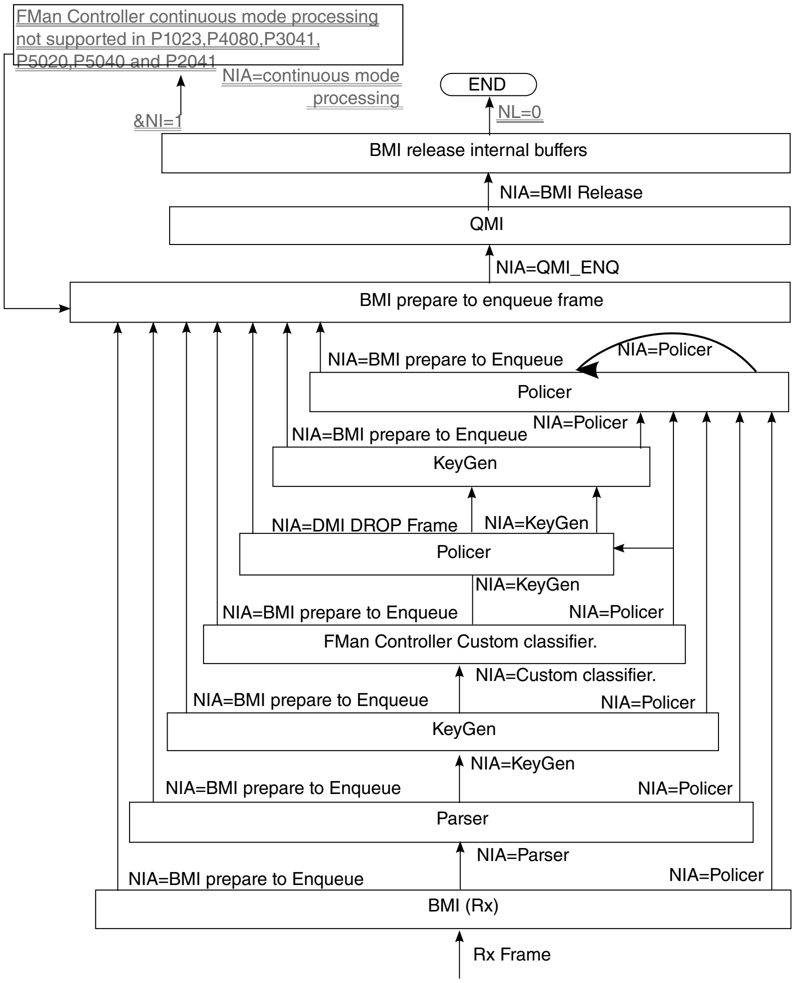

FMan Controller continuous mode processing not supported in P1023,P4080,P3041, P5020,P5040 and P2041
NIA=continuous mode processing
&NI=1
END
NL=0
BMI release internal buffers
NIA=BMI Release
QMI
NIA=QMI_ENQ
BMI prepare to enqueue frame
NIA=BMI prepare to Enqueue
NIA=Policer
Policer
NIA=BMI prepare to Enqueue
NIA=Policer
KeyGen
NIA=DMI DROP Frame
NIA=KeyGen
Policer
NIA=KeyGen
NIA=BMI prepare to Enqueue
NIA=Policer
FMan Controller Custom classifier.
NIA=Custom classifier.
NIA=BMI prepare to Enqueue
NIA=Policer
KeyGen
NIA=KeyGen
NIA=BMI prepare to Enqueue
NIA=Policer
Parser
NIA=Parser
NIA=BMI prepare to Enqueue
NIA=Policer
BMI (Rx)
Rx Frame

Figure 105. Available flows

8.2.5.2.7.1.1.2 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.

8.2.5.2.7.1.1.3 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 581.

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.

8.2.5.2.7.1.1.4 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...

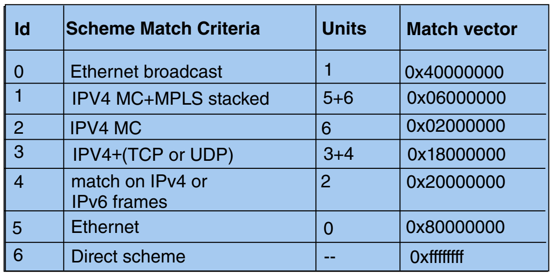
5	1	1	1	1	1	0	1	1...1	1+4
6	1	0	1	1	1	1	1	1...1	2+4
7	1	1	1	1	1	1	1	1...1	1+2+4

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.



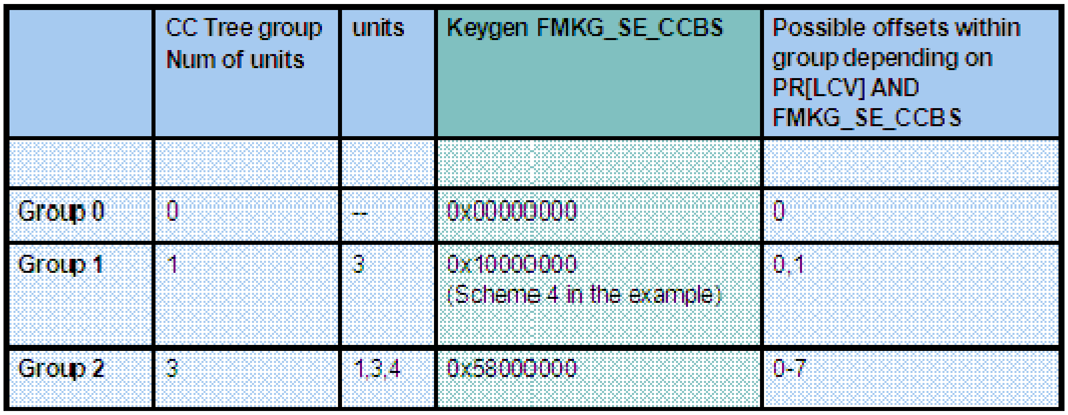
Id	Scheme Match Criteria	Units	Match vector
0	Ethernet broadcast	1	0x40000000
1	IPV4 MC+MPLS stacked	5+6	0x06000000
2	IPV4 MC	6	0x02000000
3	IPV4+(TCP or UDP)	3+4	0x18000000
4	match on IPv4 or IPv6 frames	2	0x20000000
5	Ethernet	0	0x80000000
6	Direct scheme	--	0xffffffff

Figure 106. 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 563).

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 107. 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.

8.2.5.2.7.1.1.5 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.

8.2.5.2.7.1.1.6 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 563). 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.

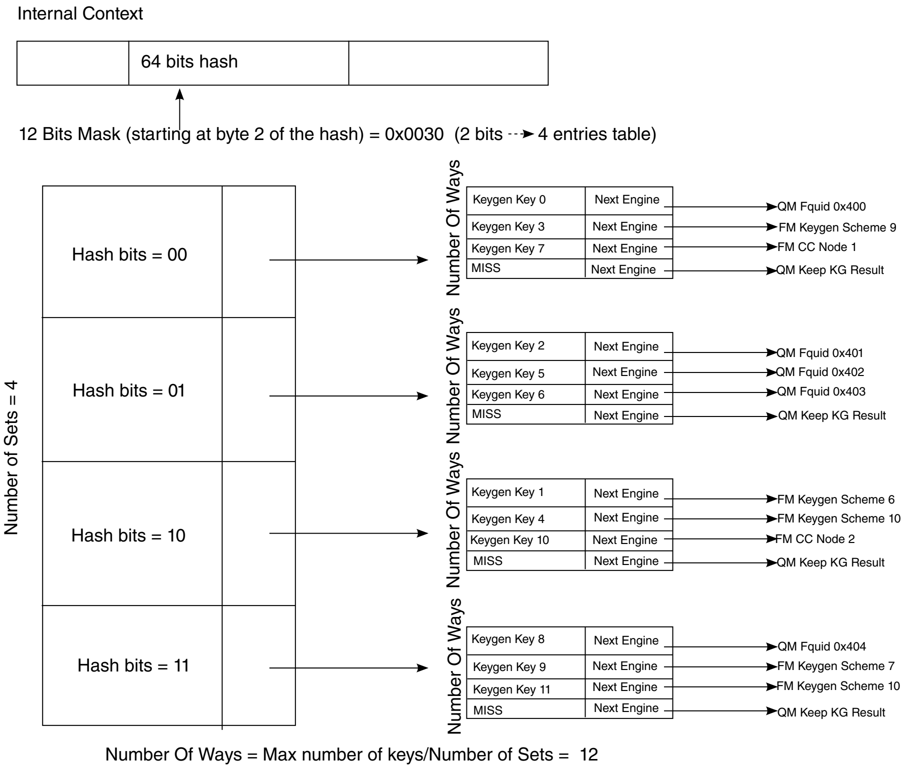


Figure 108. Hash_Table node example

8.2.5.2.7.2 How to use the FMan PCD Driver?

The following sections provide practical information for using the software drivers.

8.2.5.2.7.2.1 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

8.2.5.2.7.2.2 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

8.2.5.2.7.2.3 FMan PCD Driver Functional Description

The following sections describe main driver functionalities and their usage.

8.2.5.2.7.2.3.1 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 581 for more details on port-PCD binding.

8.2.5.2.7.2.3.2 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:

8.2.5.2.7.2.3.3 **Network Environment Characteristics**

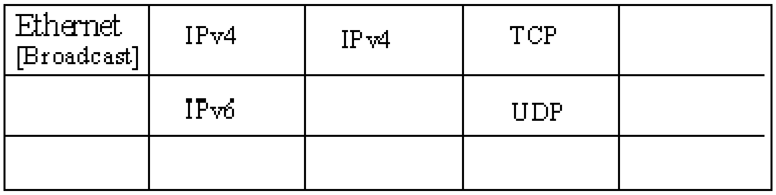
The Network Environment (NetEnv) is a software entity that lists the network protocols used by the FM-PCD for classification and distribution. The total number of NetEnvs defined depends on the system configuration. A NetEnv may be defined per port or shared among some or all ports. The definition of 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 590).

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 109. 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 555.

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`

8.2.5.2.7.2.3.4 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`

8.2.5.2.7.2.3.5 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 590.

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 560.

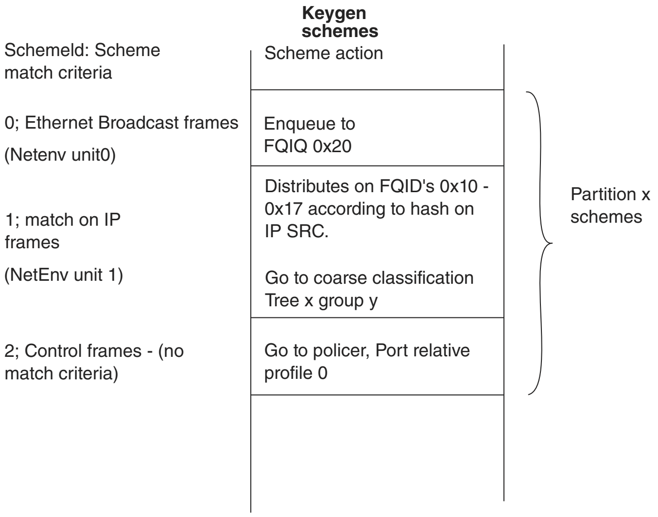

Keygen schemes
SchemeId: Scheme match criteria
Scheme action
0; Ethernet Broadcast frames
(Netenv unit0)
Enqueue to FQIQ 0x20
1; match on IP frames
(NetEnv unit 1)
Distributes on FQID's 0x10 - 0x17 according to hash on IP SRC.
Go to coarse classification Tree x group y
2; Control frames - (no match criteria)
Go to policer, Port relative profile 0
Partition x schemes

Figure 110. 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`

8.2.5.2.7.2.3.6 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 102. CC Root Nodes Selection

Unit0	Unit1	Unit2	Unit3	Selected Node
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
1	1	0	0	12
1	1	0	1	13
1	1	1	0	14

To allow more than 4 units to be involved in the selection, the 16 entries may be divided into groups. The table above demonstrates an organization of one group of 16 nodes, but other organizations are possible:

2 groups of 8 -> each group selected by 3 units (to select nodes 0-7 relative to this group's base)

4 groups of 4 -> each group selected by 2 units (to select nodes 0-3 relative to this group's base)

8 groups of 2 -> each group selected by 1 units (to select nodes 0-1 relative to this group's base)

16 groups of 1 -> indifferent to units (single node group always selected)

2-8 groups of varied sizes (8-1)

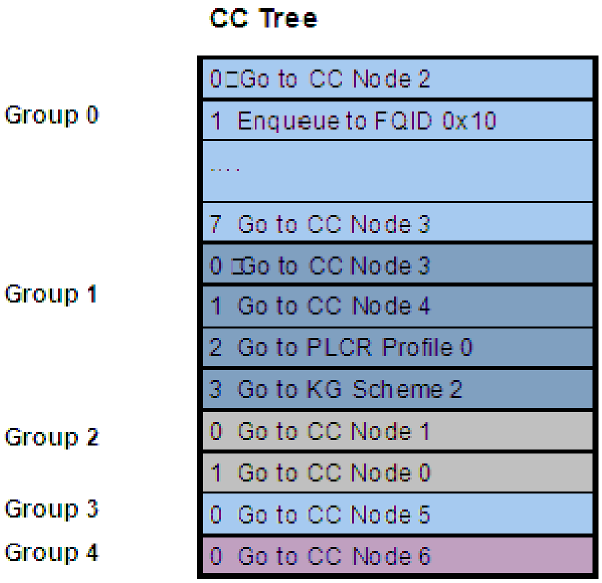


Figure 111. CC Root - 5 groups example

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.

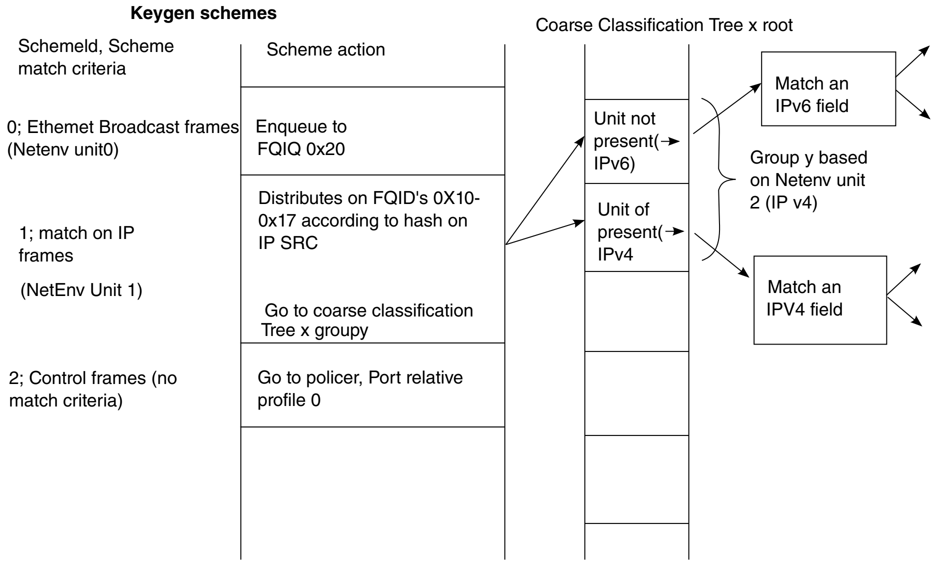

Keygen schemes
SchemeId, Scheme match criteria
Scheme action
0; Ethemet Broadcast frames (Netenv unit0)
Enqueue to FQIQ 0x20
1; match on IP frames
(NetEnv Unit 1)
Distributes on FQID's 0X10-0x17 according to hash on IP SRC
Go to coarse classification Tree x groupy
2; Control frames (no match criteria)
Go to policer, Port relative profile 0
Coarse Classification Tree x root
Unit not present(→ IPv6)
Unit of present(→ IPv4
Group y based on Netenv unit 2 (IP v4)
Match an IPv6 field
Match an IPV4 field

Figure 112. 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`

8.2.5.2.7.2.3.7 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.

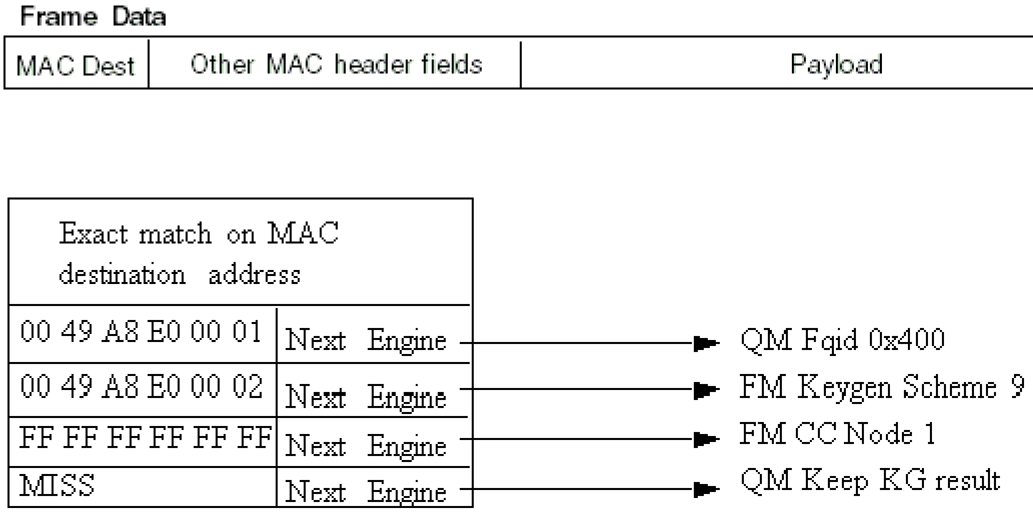


Figure 113. Exact-Match Node Example

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.

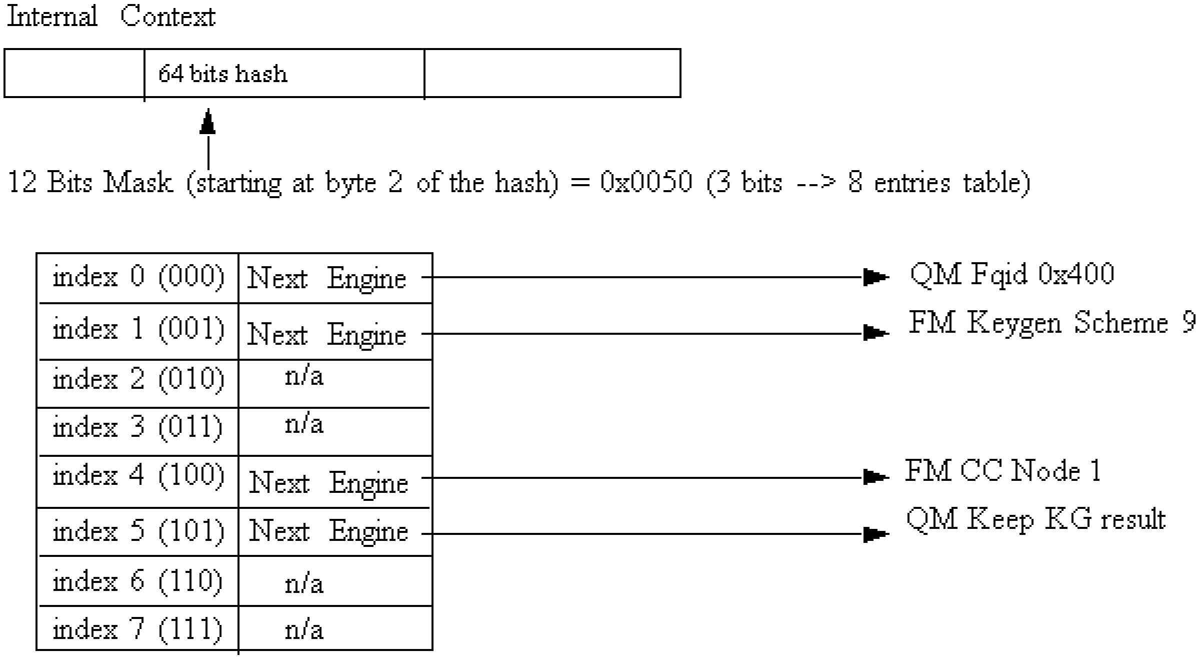


Figure 114. Indexed-Lookup node example

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.

8.2.5.2.7.2.3.8 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).

8.2.5.2.7.2.3.9 **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.

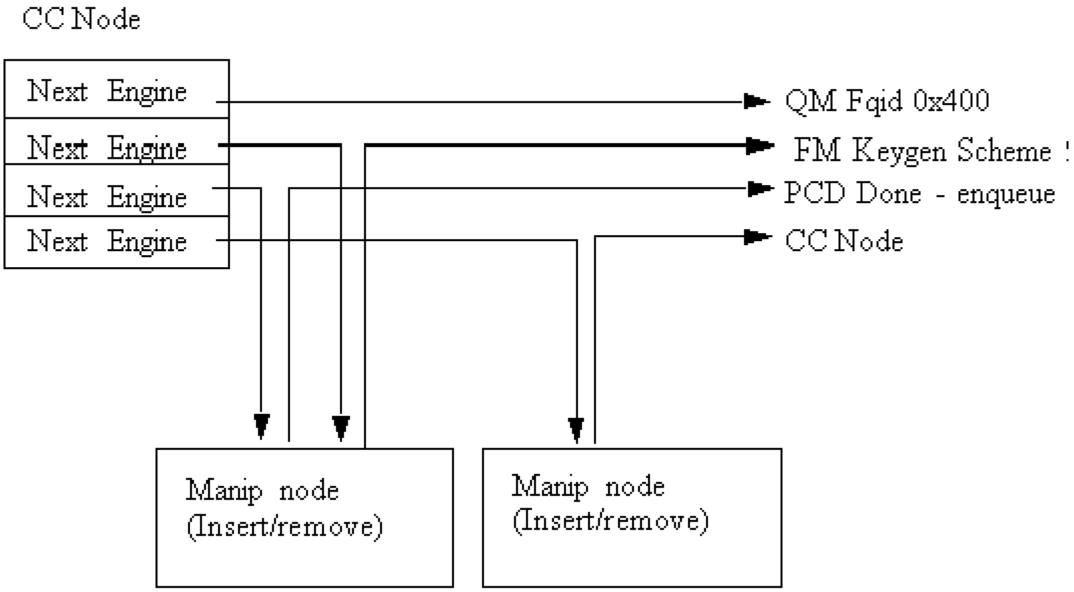


Figure 115. CC Node With Manipulation

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`'

8.2.5.2.7.2.3.9.1 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.

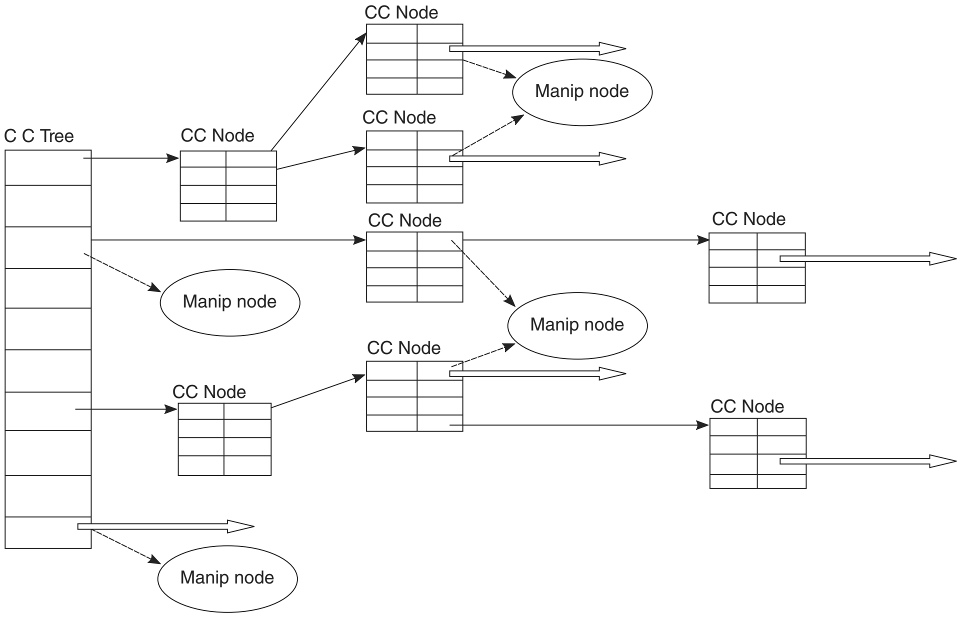


Figure 116. Header Manipulation CC Perspective

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.

8.2.5.2.7.2.3.9.2 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 direct schemes the Keygen and caught by dedicated schemes that pass them to the Custom Classifier. The CC Root object is configured so that the IP fragments will reach a dedicated root entry node that contains a CC manipulation node. At this point, the IP Reassembly is performed. When a full frame is gathered, it is passed by the FMan controller back to the parser as a full reassembled frame. It is then passed to the Keygen and may be distributed and classified as any other frame.

What should the user do?

The following sequence describes the steps the user must take in order for the flow above to work.

- Initialize general DPAA (BM, BM Portal, BM Pools, QM, QM Portal, FMan and FMan PCD)
- Initialize the Rx/Offline FMan Port on which reassembly should run
- Define PCD as follows:
    - Set a Network Environment with one of the following options:
        - `HEADER_TYPE_IPV4` unit with `IPV4_FRAG_1` option for IPv4 reassembly manipulation.
        - `HEADER_TYPE_IPV6` unit with `IPV6_FRAG_1` option for IPv6 reassembly manipulation.

Note that if the user needs IPv4 or IPv6 units for other use, the fragmentation units may not be shared and dedicated units must be defined.

- Allocate the first one or two schemes - one if only IPv4 is used, 2 if IPv6 is also used. The user should not configure those schemes, just save these schemes from other usage. The driver will use the first scheme for IPv4, and if needed, it will use the second for IPv6.
- Create reassembly manipulation using `FM_PCD_ManipNodeSet` routine. Pass the relative id's of the schemes allocated above (A single manipulation module should be created for both IPv4 and IPv6 fragmented frames, passing all relevant parameters).
- If CC is used, it is user's responsibility to leave two unused entries when building the CC root nodes (i.e. the total number of entries between all groups should not exceed 14).
- Set at least one scheme to catch regular/reassembled frames.

- When binding the Rx/Offline FMan Port to the PCD properties (i.e. calling `FM_PCD_SetPCD`), pass a handle to the created Reassembly Manipulation node.

Note that in order to perform distribution or classification on IPv4/IPv6 frames (unrelated to reassembly of IPv4/IPv6 fragments), independent IPv4/IPv6 units with no option must be explicitly defined.

What does the driver do?

In order to provide the required support for IP Reassembly, the driver performs some internal actions triggered by the user configuration. The following information describes the actions the driver internally performs and has no functional relevance to the user:

- When reassembly is required, the driver internally enables parser recognition of IPv4/IPv6 and shim2 - which is the IP Reassembly extension. This is triggered by the user defining NetEnv units with options: `IPV4_FRAG_1/IPV6_FRAG_1`.
- The driver loads the software parser that identifies IP fragments and enables its operation for the required FMan Port.
- The driver defines one or two (one for each IP version) Keygen schemes that recognize IP fragments and are programmed to generate an IP Reassembly key. When a frame is recognized as an IP fragment (by the Parser), it is steered to these Keygen schemes. The user should allocate the first one or two (for IPv4 and/or IPv6) schemes and pass their relative id's to the driver. The driver will internally initialize the relevant reassembly schemes when required.
- Each of the schemes above is programmed by the driver to point to a group in the Custom Classifier Root. If the user did not create a CC Root, the driver internally creates a new one. In both cases, the driver creates the needed group/s in the CC Root. It always uses the last two groups. It is user's responsibility to have at least two empty entries (one for a single IP version, two for both).
- The driver attaches the Manipulation sequence (created by the user) to the appropriate root entry node in the CC Root, causing the reassembly of IP fragments.

NOTE

The software parser code required to support reassembly may not coexist with user software parser code. If the user supplies IPv4 or IPv6 software parser code, it must include the code for handling IPv4/IPv6 reassembly according to the FMan controller spec.

Suggestions of how to use IPR in a system

The PCD with the IPR should identify frames up to L3; i.e. if the frame is IP or not.

In case it isn't an IP frame it should pass the desire PCD. IP frames should pass the reassembly process and than be directed to OP-Port to be classified according to their L3 and above.

8.2.5.2.7.2.3.9.3 **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 563). All relevant parameters such as MTU are defined during this module creation.

Following is the sequence that should be followed:

- Initialize general DPAA (BM, BM Portal, BM Pools, QM, QM Portal, FMan and FMan PCD)
- Initialize FMan Port of type Offline Parsing
- Define fragmentation PCD as follows:
    - Initialize an empty Network Environment (without any units)
    - Create fragmentation manipulation using `FM_PCD_ManipNodeSet` routine.
    - Create CC Node by calling `FM_PCD_MatchTableSet`/`FM_PCD_HashTableSet` and attached the fragmentation manipulation previously created to the desired key.
    - Build a CC Root with 1 group that points to the previously defined CC Node .
- Bind the Offline Parsing FMan Port to the PCD properties by calling `FM_PORT_SetPCD`

Manipulation parameters

- MTU of the fragmentation manipulation.
- Scratch Buffer Pool ID is a buffer pool that is required by the fragmentation process in order to ensure correct release operation of the frames and fragments. All IP Fragmentation Table Descriptors should use the same Scratch Buffer Pool ID. This pool must not be used by any other process or engine in the system.
- Don't Fragment Action - by setting this parameter the user can determine the action to be taken in case the IP packet is larger than the defined MTU and the 'Don't Fragment' (DF) bit of the frame is set.

NOTE

The software parser code required to support fragmentation may not coexist with user software parser code. If the user supplies IPv6 software parser code, it must include the code for handling IPv6 fragmentation according to the FMan controller spec.

Restrictions:

1. Tx confirmation is not supported.
2. Only Bman buffers shall be used for frames to be fragmented.
3. IP-Fragmentation will not work on OP-Port with VSP enabled.
4. fragmentation of IP-fragments is not supported
5. IPv4 packets containing header option field are fragments by copying all option fields to each fragment, regardless of the copy bit value.
6. Maximum number of fragments per frame is 16.

Suggestions of how to use IPF in a system:

In case one of the #1-#2 3 restrictions above is critical than it is suggested not to use IPF on OP-Ports that receive frames from the GPP and to do it on the GPP itself. We also suggest to put the IPF on a OP-Port just before the TX-Port.

8.2.5.2.7.2.3.9.4 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

8.2.5.2.7.2.3.10 **Frame Replicator**

The Frame Replicator (FR) is designed to duplicate incoming packets and route them to separate destinations. It is defined as a next engine and may follow other CC nodes, i.e. Match-table key, Hash-Table key or a CC-Root entry.

A Frame Replicator is realized by a group of members, where each member defines a replication of the incoming frame and a route to continue.

The next engine after FR is restricted to one of the following:

- Enqueue (PCD Termination)
- Policer
- Keygen (Direct scheme that leads to either Policer or PCD Termination)

When initializing an FR node, the user must define the maximum number of members this node may contain. The actual number of members may be modified on runtime by adding and removing FR group members.

Runtime modifications of add/remove members to/from the group can be done at any point in the system and in any location of the members group (first, middle or last). Note that runtime-modifications require the use of Host Command.

The order of the members in the group is of significance as the implementation of the replication is serial.

Manipulation may be applied to:

1. The whole group. The manipulation node should be placed before the replication group. That means that the FR is the next-engine of the Manip node. The Manip node is the next-engine of a key in a Match-table or Hash-table.
2. The last member of a FR group. That means that the manip node is the next-engine of the last member of the FR group.

NOTE

No support of Manip node after the "non-last" members.

The driver supports sharing of FR nodes means that FR group may be shared by more than one source.

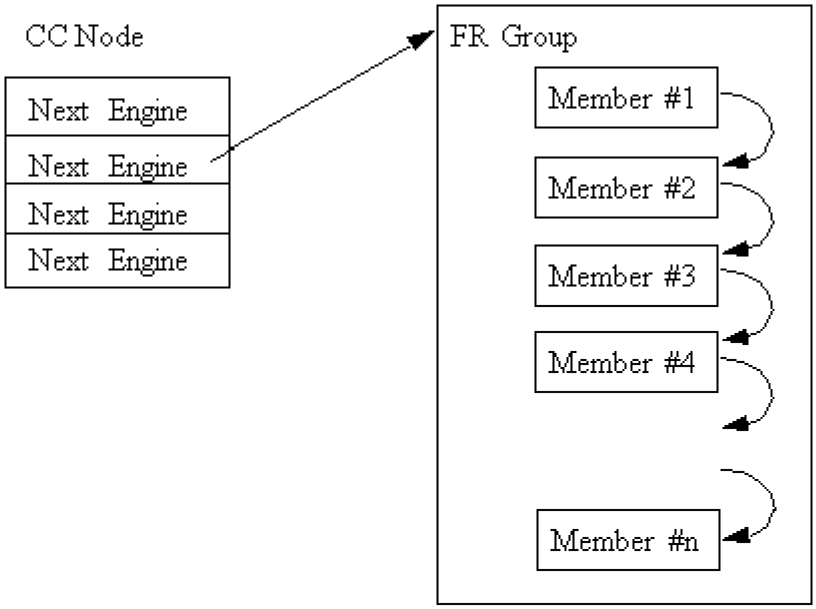


Figure 117. Frame Replicator Following a CC Node

Available API:

- `FM_PCD_FrmReplicSetGroup`
- `FM_PCD_FrmReplicDeleteGroup`

Specific runtime API:

- `FM_PCD_FrmReplicAddMember`
- `FM_PCD_FrmReplicRemoveMember`

NOTE

- For all manipulation types below, the user must call `'FM_PCD_SetAdvancedOffloadSupport'` before calling `'FM_PCD_Enable'`.
- For each RX/OP-Ports that will work with the above FM-PCD, the user should have at least 16 tnums (num of tasks). In order to set the tnums, the user should call '`FM_PORT_ConfigNumOfTasks`'.
- It is also required to set the DMA transactions to be per port by calling '`FM_ConfigDmaAidOverride`' with 'FALSE' and calling '`FM_ConfigDmaAidMode`' with '`e_FM_DMA_AID_OUT_PORT_ID`'

8.2.5.2.7.2.3.11 Policer Profiles

The policer profile entity relies on the hardware entity. It defines rules for policing for a certain flow. There are 256 different profiles in a frame manager that may be organized in per port windows. Some profiles may be shared between ports on the same PCD. By default, the number of shared profiles is set by the driver, but the user can also configure it to a different value. Shared profiles are typically used for aggregation.

When a PCD is defined in a single partition environment, it is the owner of all 256 profiles. When a PCD is defined in a multipartition environment, it is the owner of its shared profiles along with all the profiles that will be allocated per port for ports on this partition. The user must explicitly allocate per-port profiles for each port (if required), after PCD is initialized and prior to the profile initialization. Note that per-port profiles are the only PCD resource that is explicitly allocated and initialized for a specific port.

After profiles are mapped, the user may initialize each of the profiles by stating the following:

- Type
    - Shared

    - Per-port

- Offset relative to the port or to the shared group of profiles
- Characteristics

Once initialized, a handle is assigned to the profile for later use.

The Policer may be used after the Parser, Keygen or Custom Classifier, or solely - without activating any of the other PCD engines. It is not dependant on any previous output such as parser result. The policer may be used more than once in a frame flow. The next action after a police profile is either to pass the frame to a direct Keygen scheme for a new distribution (typically for control frames coming from the Custom Classifier), to pass the frame to another profile (always a shared profile, typically an aggregators), or to enqueue the frame to an FQID.

When other engines select a policer profile as the next engine, its handle must be passed. An exception is when a per-port profile is specified as the next engine of a scheme or of a "overrideParams" CC key. In these cases a port-relative index is required instead. The reason for this is that the required Policer Profile may not be initialized at this stage and hence have no handle. This irregular behavior is because CC Roots and KG schemes may be shared by ports, and at the time of scheme/root initialization, they are not yet bound to specific ports. In this context, the profile selected may in fact be uninitialized and therefore can't be verified by the driver. It is therefor user's responsibility to make sure it is set prior to port- PCD binding.

Runtime Modifications: Valid profiles may be modified at runtime by calling the profile initialization routine for an existing profile, passing the profile handle as 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`

8.2.5.2.7.2.3.12 PCD Organization

By initializing PCD resources, the user creates a directed graph in which the parser is the source of the graph and the FQIDs are its endpoints. Following figure shows a generalized example of a basic PCD graph.

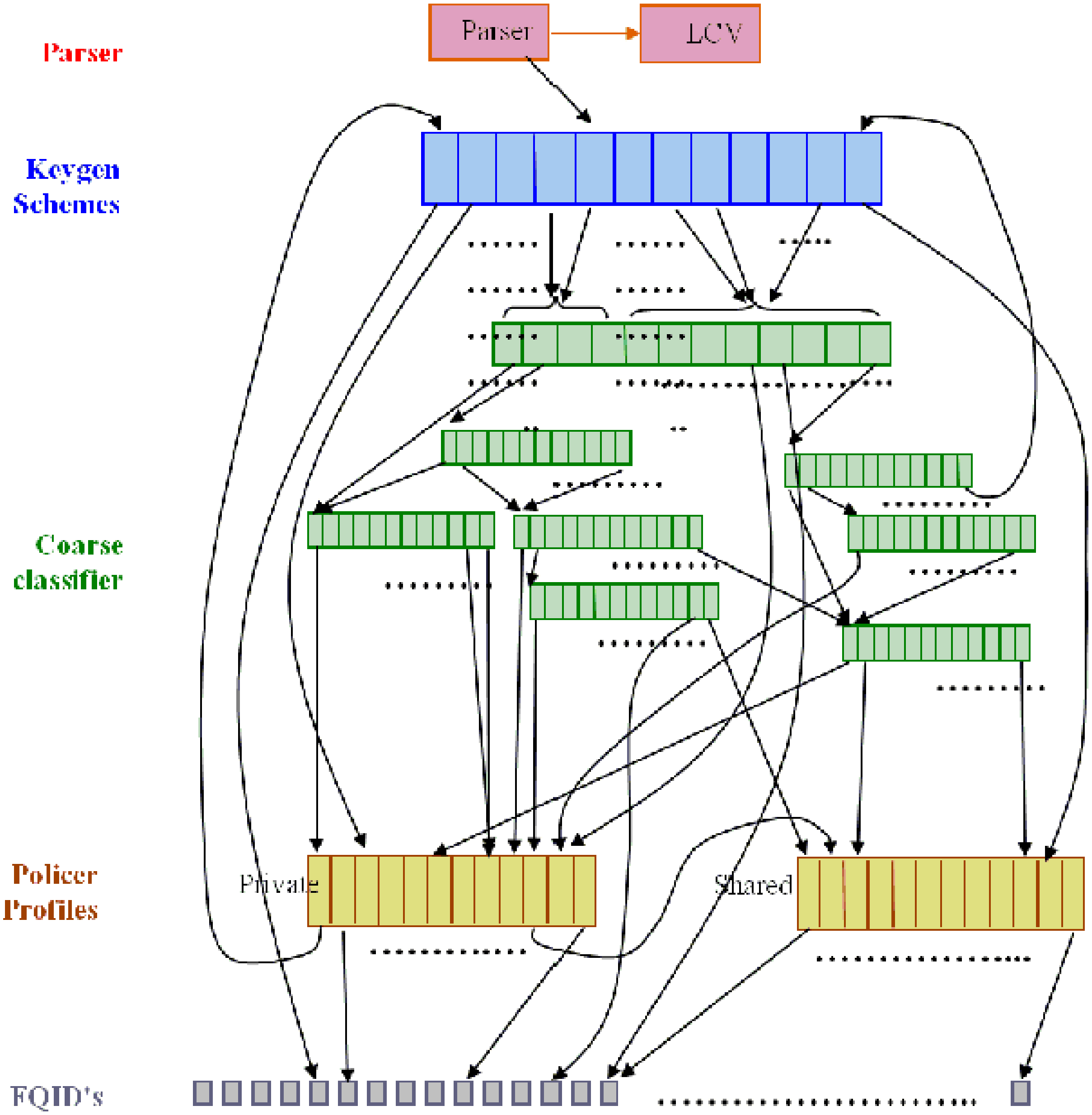


Figure 118. PCD Organization

8.2.5.2.7.2.3.13 PCD Definition Sequence

When a PCD graph is defined, its resources must be initialized bottom up when there's a dependency between them. Following figure shows the order of initialization (starting at the top of the figure) in a specific sequence.

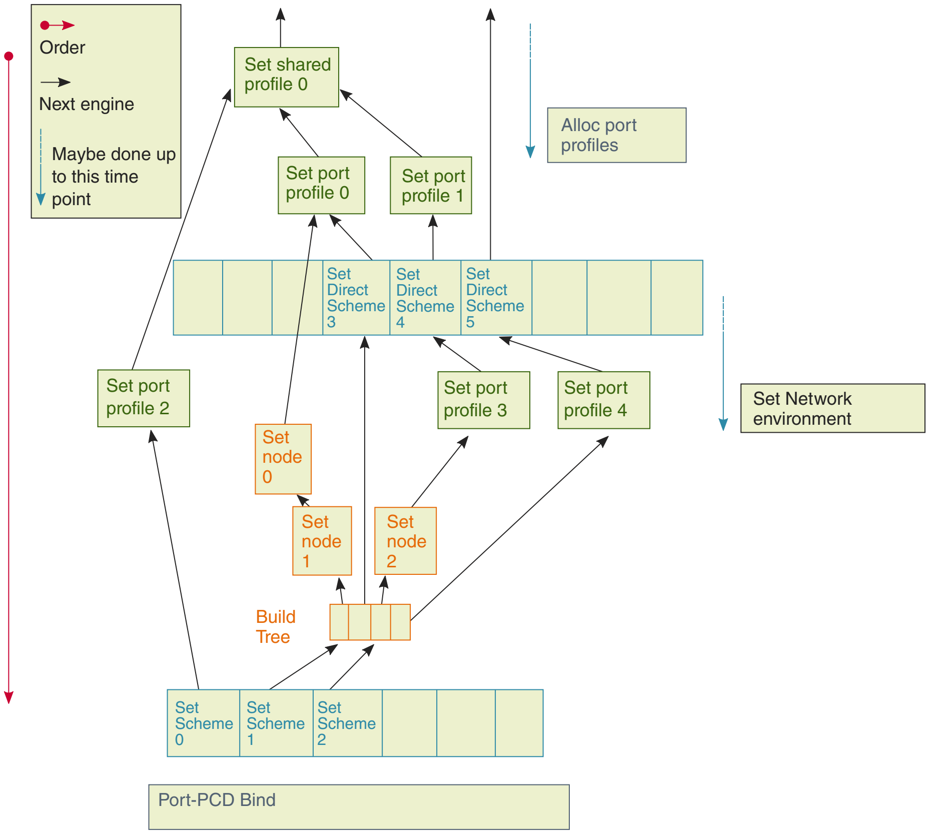


Figure 119. Definition Sequence

8.2.5.2.7.2.3.14 Host Command

Some PCD functionalities may be managed by either memory-mapped registers or by the host command mechanism to allow independent programming in a multipartition environment. In a single partition environment in the FMan driver, the host command mechanism is optionally used, but in a multipartition environment, wherever available, only the host command is used to prevent a risk of racing. The host command driver is a part of the PCD driver and is initialized internally by the driver, using user parameters.

When PCD is first initialized in a single-partition environment, the user must specify whether the host command should be used, and if so, host command parameters are required. In a multipartition environment, the use of the host command is forced and all host command parameters are required. When PCD initialization routine is called by the master/single partition driver, the user parameters include host command port parameters (such as port id, virtual address, and default queues) and the FMan Port for the host command is internally initialized.

8.2.5.2.7.2.3.15 PCD Statistics

The FMan PCD API provides access to all the statistics gathered by the FMan PCD engines hardware. Statistics is enabled by default but may be disabled/enabled at runtime using the dedicated API.

The following API routines may be called at any time after initialization to retrieve any of the following FMan PCD counters:

- `FM_PCD_GetCounter`
- `FM_PCD_KgSchemeGetCounter`
- `FM_PCD_PlcrProfileGetCounter`

8.2.5.2.7.2.3.15.1 Custom Classifier Statistics

A CC node supports statistics gathering on per-key basis. In order to enable statistics gathering by a CC node (Match table or Hash table), statistics mode must be provided upon initialization of that node and this will determine the statistics mode for all keys of the CC node.

Next, statistics should be enabled per-key, meaning statistics should be enabled for every key that the user wishes to monitor.

After these steps, the following API routines may be called to retrieve the statistics:

- `FM_PCD_MatchTableGetKeyCounter`
- `FM_PCD_MatchTableGetKeyStatistics`
- `FM_PCD_MatchTableFindNGetKeyStatistics`
- `FM_PCD_HashTableFindNGetKeyStatistics`

8.2.5.2.8 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.

8.2.5.2.8.1 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.

8.2.5.2.8.1.1 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.

8.2.5.2.8.2 How to use the FMan Port Driver?

The following sections provide practical information for using the software drivers.

8.2.5.2.8.2.1 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

8.2.5.2.8.2.2 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

8.2.5.2.8.2.3 FMan Port Driver Functional Description

The following sections describe main driver functionalities and their usage.

8.2.5.2.8.2.3.1 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.

8.2.5.2.8.2.3.2 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.

8.2.5.2.8.2.3.3 Independent-Mode

Dpaa-im is an Ethernet driver using Dpaa Fman, this implements on independent mode. For details on usage, refer README in source code.

8.2.5.2.8.2.3.4 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.

8.2.5.2.8.2.3.5 Virtual Storage Profiles Support

An FMan Port may use the legacy Physical Storage Profile or the Virtual Storage Profiles (VSP). This section will discuss the usage of VSP by an FMan port, while more information about the VSP mechanism which is implemented by the driver as separate entity FM_VSP, can be found in FMan VSP Driver on page 586.

When a user wants to set an Rx or OP port to work in virtualization mode using VSP's rather than the physical SP, user should call the function which allocates a storage profile window (range of VSPs allocated in continuously manner) to a port. The user should also define which profile in this range should be used as default SP; note that the default profile should be a relative index within the allocated window. Upon calling the window allocation routine, the driver enables virtual mode (i.e. using VSPs) for this port, allocates its profiles and defines default SP. In order to redirect a packet into a certain VSP, user may set the 'relative-VSP-id' within the PCD graph nodes (e.g. in the match-table entries). The value in the PCD graph nodes is port relative so if two ports are sharing the same PCD graph node (e.g. a match-table), the actual VSP will be selected by the 'relative-VSP-id' plus the port's base VSP as shown in the figure below.

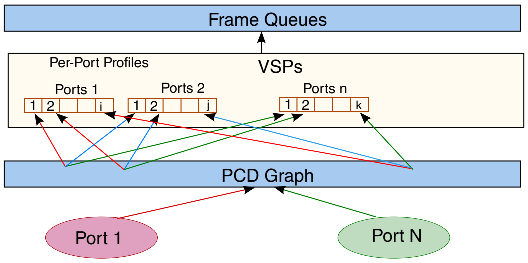


Figure 120. FMan PCD graph and the VSP selection

Rules and restrictions regarding the use of VSP:

- When called for Rx ports, the allocation routine expects also the handle of coupled Tx port as a parameter; the driver sets automatically the Tx port to work in VSP mode also and use the same default profile for this port.
- Storage Profiles windows may not overlap; i.e. sharing of VSPs between FM ports is not allowed by the driver.
- A call to the allocation routine requires that the FM port will be disabled. In the case of Rx port, coupled Tx port should also be disabled. When an FM Port (that has VSP mechanism enabled) is enabled, at least the default profile must be initialized.
- A call to the allocation routine may not be reverted, i.e. it's impossible to disable virtualization mode.
- Number of profiles to be allocated must be a power-of-2. In addition, the "base-profile" that will be allocated by the driver will be aligned to the number-of-profiles provided by the user.

- For FM-Port that works with VSP, its classification should also use VSP; i.e. classification (e.g. KG scheme or CC-node) should NOT try to revert from VSP to the FM-Port "physical" SP.
- When user frees all resources of FM port, the driver frees automatically VSP window which have been allocated for this port.

Initialization Sequence:

- Initialize FM Tx Port
- Initialize FM Rx Port
- Allocate VSP for FM Rx Port (thus enabling virtualization mode)
- Initialize default VSP (See FMan VSP Driver on page 586)
- Enable FM Ports

Free Sequence

- Disable Ports
- Free the default VSP
- Free FM Tx Port
- Free FM Rx Port

8.2.5.2.8.2.3.6 Rate Limiting

The driver supports the hardware mechanism of rate limiting for Tx ports. The runtime API consists of a number of parameters including a definition of the required rate (in KB/sec for Tx ports, in frame/sec for offline parsing ports) and refers to data rate rather than line-rate.

8.2.5.2.8.2.3.7 Simple BMI-to-BMI (regular) mode

This is the default FMan Rx/Offline Parsing Port mode. After Port initialization and prior to Port-PCD binding, all traffic will be received on the default Rx queue. This mode is called "BMI-To-BMI" as no PCD is involved in the data reception.

This mode is useful for the early state of a port as well as when major runtime PCD modification takes place. In such a case, sometimes the whole PCD functionality needs to be manipulated and the user should temporarily detach the Port from the PCD, receive all frames on the default Rx queue and only re-attach it to the PCD after the modifications have completed.

8.2.5.2.8.2.3.8 Port LIODN

An FMan Port LIODN is constructed out of a base and offset.

Upon FMan Port configuration, the user must specify the port's base LIODN.

For Rx ports, the user must also specify the LIODN offset for each port. No such configuration is required for Tx and Offline Parsing ports since on transmission, the offset LIODN is taken from the frames' FD. The FD is set according to the source of the frame - if transmitted by CPU, it is dynamically set by the QM SW portal. Another scenario is frames forwarded by other engines, in such a case their FD must contain the correct LIODN offset.

8.2.5.2.8.2.3.9 Port-PCD Binding

Ports may be linked to the PCD graph according to their PCD binding specifications and considering partition and Network Environment restrictions.

Following figure shows a schematic demonstration of possible port > PCD binding.

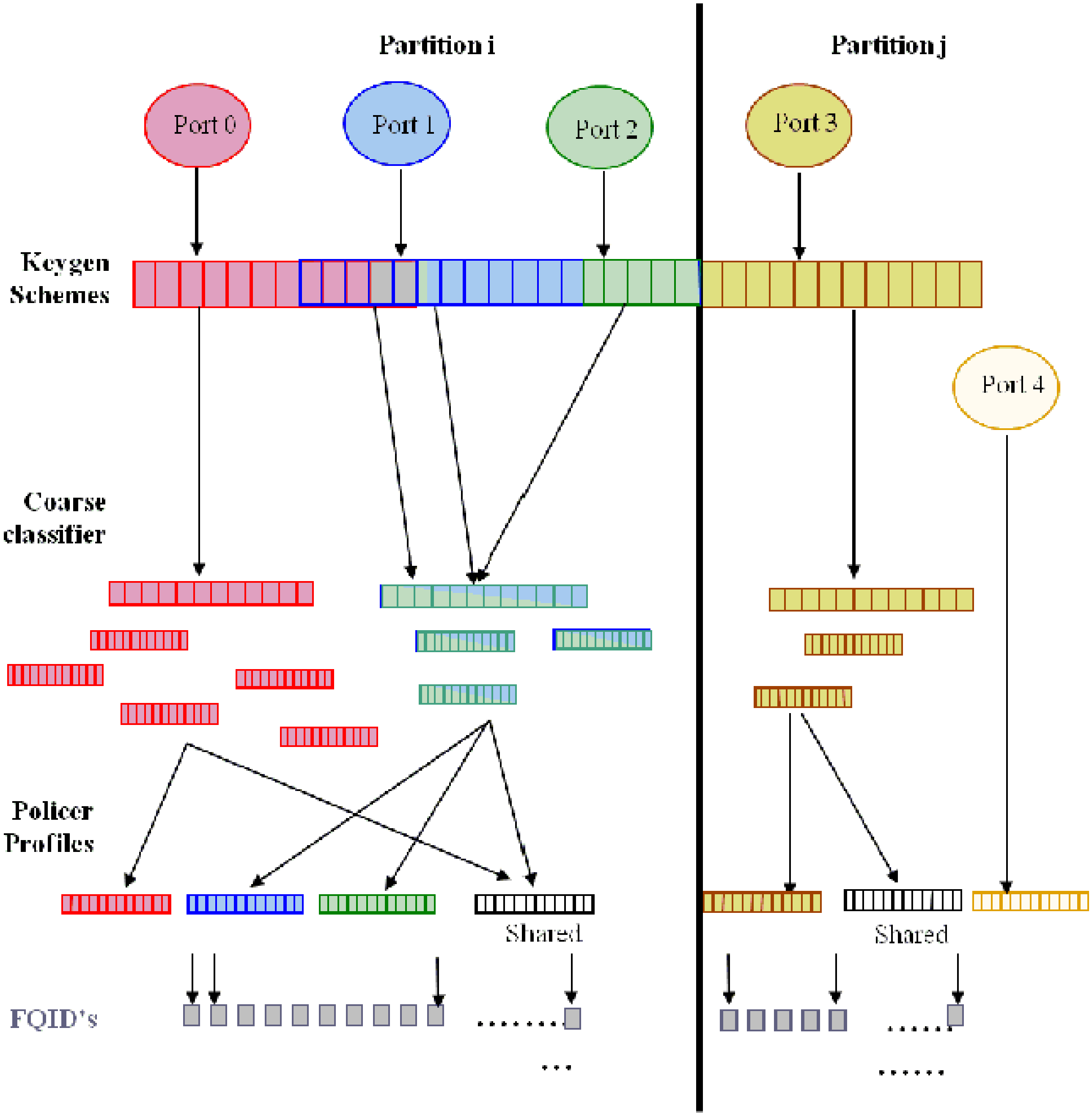


Figure 121. Port-To-PCD binding example

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`

8.2.5.2.8.2.3.10 Port-PCD Binding Changes

There are three levels of Port-PCD binding changes:

- **Basic Runtime Modifications-**May be invoked while PCD is active and on enabled ports using PCD.
    - — Port routines responsible for binding/unbinding to/from the modified resources.
        - `FM_PORT_PcdKgBindScheme`
        - `FM_PORT_PcdKgUnbindScheme`
    - — Port routines responsible for PCD change of behavior.

  - `FM_PORT_PcdKgModifyInitialScheme`
  - `FM_PORT_PcdPlcrModifyInitialProfile`
  - `FM_PORT_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`

8.2.5.2.9 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.

8.2.5.2.9.1 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.

8.2.5.2.9.1.1 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.

8.2.5.2.9.2 How To Use The FMan MAC Driver?

The following sections provide practical information for using the software drivers.

8.2.5.2.9.2.1 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

8.2.5.2.9.2.2 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

8.2.5.2.9.2.3 FMan MAC Driver Functional Description

The following sections describe main driver functionalities and their usage.

8.2.5.2.9.2.3.1 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.

8.2.5.2.9.2.3.2 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.

8.2.5.2.9.2.3.3 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. IEEE1588 functionality is always enabled on FM-MAC. Thus, no additional settings are required for the MAC. and the FM-MAC and only then they can enable this feature by calling `FM_MAC_Enable1588TimeStamp` routine. Once enabled, the user may also set the exception for receiving 1588 relevant interrupts on the MAC.

8.2.5.2.9.2.3.4 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.

8.2.5.2.10 FMan VSP Driver

The FMan VSP driver module refers to the software support provided for the Virtual Storage Profile mechanism.

8.2.5.2.10.1 FMan VSP Hardware Overview

VSPs may be used by user for virtualization. If a user is running with a multi-partitioned (or with a multiple software entities) system where a single MAC may be used by several software partitions/entities simultaneously, except for using a different FQID (that is already available in DPAA1.0), user may use a different VSP for each SW partition/entity; that way, the buffer may be private (rather than being shared as in DPAA1.0). It allows the virtualization of the buffer pool selection for frame storage (and other parameters related to storage in external memory) from the physical hardware ports. Using this mechanism, different packets received on the same physical port may be stored in different BM pools based on the frame header, in a similar way to FQID selection. VSPs are replacing the legacy, "physical", per-port BM Pool selection. A backward compatible mode exists and it is possible to use the original BM Pool selection, now referred to as "Physical SP".

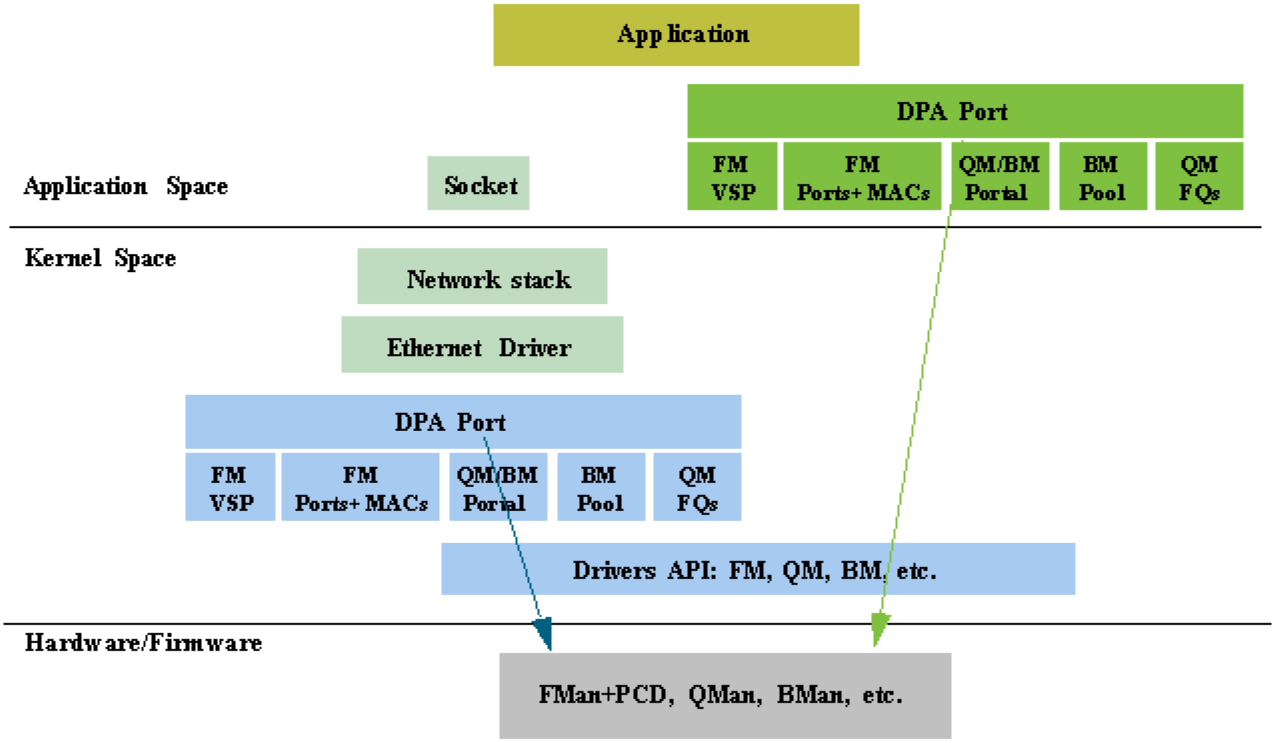


Figure 122. Virtualization Using VSPs

The global FMan module is in charge of the Virtual Storage Profiles entries management. On FMan initialization, the first VSP index dedicated to this partition must be defined (it should be an absolute index), and so is the total number of VSP's for this partition. Later, for each port using VSP's, a window of entries should be defined. VSPs may not be shared among FMan ports.

Each port has a default VSP. On each PCD classification, a VSP may be selected. Received packets will be written into the destination buffer according to the VSP parameters, while the VSP is selected according to the frame headers and the PCD configuration.

The VSP is implemented by the driver as separate entity, however, other modules of the FM driver are aware of this entity and interact with it. An FM VSP module represents a single storage profile.

The global FMan module is in charge of the Virtual Storage Profiles entries management. On FMan port initialization, if using VSP mode, it should allocate and bind to a range of VSP's. On the PCD, A decision is being taken by user on every node of the PCD graph whether to continue to work with previously defined VSP or to override with a new profile.

8.2.5.2.10.2 How To Use The FMan VSP Driver?

The VSP is implemented by the driver as separate entity, however, other modules of the FM driver are aware of this entity and interact with it. An FM VSP module represents a single storage profile.

The global FMan module is in charge of the Virtual Storage Profiles entries management. On FMan port initialization, if using VSP mode, it should allocate and bind to a range of VSP's. On the PCD, A decision is being taken by user on every node of the PCD graph whether to continue to work with previously defined VSP or to override with a new profile.

8.2.5.2.10.2.1 FMan VSP Driver Scope

This module represents the FMan VSP driver. It includes:

- FMan VSP hardware structures configuration and enablement
- Parsing of the buffer
- Statistics

8.2.5.2.10.2.2 FMan VSP Driver Sequence

This sequence includes other modules required for the VSP

- Definition of general VSP parameters on global FMan initialization
- FM Port initialization
- FM Port VSP window allocation
- FM Port enablement
- FMan VSP Config routine (for specific VSP's)
- [Optional] FMan VSP advance configuration routines (for specific VSP's)
- FMan VSP Init routine (for specific VSP's)

8.2.5.2.10.2.3 FMan VSP Driver Functional Description

The following sections describe main driver functionalities and their usage.

8.2.5.2.10.2.3.1 Virtual Storage Profile Initialization

The VSP's must be initialized prior to their usage. It is user's responsibility to initialize at least the default VSP for each port before enabling it. Similarly, it is their responsibility to initialize all other VSPs before a classification that may use some VSP is enabled.

Initializing a VSP defines the destination BM Pool buffer for a specific type of packets. It also defines the structure of the buffer - i.e. the data offset, the prefix content, etc.

8.2.5.2.10.2.3.2 Virtual Storage Profile Parsing

On VSP initialization, the user defines the buffer prefix content. Based on these requirements, the driver then defines the buffer prefix structure, i.e. data offset, whether certain information such as parse result should be copied to the external buffer and where it will be located. On buffer reception, the user may call VSP routines in order to get the data, as well as the buffer prefix sections such as parse result, time stamp, or Keygen output.

8.2.5.2.11 FMan RTC (IEEE 1588) Driver

The FMan RTC driver module refers to the software support provided for the IEEE 1588 hardware of the FMan.

8.2.5.2.11.1 FMan RTC Hardware Overview

The 1588 timer module interfaces to up to four 10/100/1000 or one 10G Ethernet MACs, providing current time, 2 alarms, and 2 fiper periodic pulse generators.

8.2.5.2.11.2 How To Use The RTC Driver?

The following sections provide practical information for using the software drivers.

8.2.5.2.11.2.1 RTC Driver Scope

This module represents the FMan 1588 driver. It includes:

- IEEE 1588 hardware configuration and enablement
- Support for alarm mechanism
- Support for periodic pulse
- Support for external trigger
- Runtime compensation tuning
- Interrupt handling

8.2.5.2.11.2.2 RTC Driver Sequence

- FMan RTC Config routine
- [Optional] FMan RTC advance configuration routines
- FMan RTC Init routine
- FMan RTC Enable routine
- FMan RTC runtime routines
- FMan RTC Free routine

8.2.5.2.11.2.3 RTC Driver Functional Description

The following sections describe main driver functionalities and their usage.

8.2.5.2.11.2.3.1 FMan RTC 1588 module utilization

The driver API provides interface to the 1588 hardware module. It initializes its registers to define the clock period and it supports the definition of the alarms and periodic pulses. Note that When setting periodic pulse, the RTC module must be disabled.

8.2.5.2.11.2.3.2 Utilizing IEEE1588 for MAC frames time stamping

Several FMan driver modules are involved in having the 1588 time stamping functionality activated: FMan-RTC, FMan-MAC, FMan-Port and FMan-PCD.

The initialization sequence is as described below:

After the Frame Manager is initialized, the FMan-RTC needs to be initialized by calling (with the appropriate parameters):

- `FM_RTC_Config`
- `FM_RTC_Init`

Next, the following routine should be called, only after MAC is initialized.

- `FM_MAC_Enable1588TimeStamp`

From this point and on all the Ethernet frames on this MAC are time-stamped. In order to obtain the timestamp, during the FMan Port configuration, the user must call the advance config routine:

- `FM_PORT_ConfigBufferPrefixContent` (with 'passTimeStamp' parameter set).

At run-time, for each received/confirmed frame, the user should call the following routine, passing it the frame's data pointer:

- `FM_PORT_GetBufferTimeStamp`

The routine will return the pointer to the time stamp.

8.2.5.2.11.2.3.3 Utilizing IEEE1588 for PTP

The sequence described in the previous section causes all the frames that are being received or transmitted by FMan to be time-stamped. However, if the user wants to distinguish PTP frames from other frames on a specific port, PCD rules need to be applied on the PCD graph for this port; i.e using the parser to recognize the PTP frame and then using an appropriate scheme to distinguish PTP frames and route them to the desired destination queues.

8.2.5.2.12 FMan MURAM Driver

The FMan MURAM driver module refers to the memory management of the FMan Multi User RAM.

8.2.5.2.12.1 FMan MURAM Hardware Overview

The MURAM is the internal memory of the FMan.

8.2.5.2.12.1.1 FMan MURAM Driver Software Abstraction

The FMan MURAM driver is a memory manager that allows partitioning of the MURAM. Upon initialization the user receives a handle that may be used by other modules in order to allocate and de-allocate memory blocks out of that MURAM partition.

8.2.5.2.12.2 How To Use The FMan MURAM Driver?

The following sections provide practical information for using the software drivers.

8.2.5.2.12.2.1 FMan MURAM Driver Scope

This module manages the FMan MURAM. It includes MURAM allocation and de-allocation of different sizes of required memory blocks.

8.2.5.2.12.2.2 FMan MURAM Driver Sequence

- FMan MURAM config and init routine
- FMan MURAM allot and free runtime routines
- FMan MURAM free routine

8.2.5.2.12.2.3 FMan MURAM Driver Functional Description

The FMan MURAM drivers supports MURAM memory blocks allocation and de-allocation. After initializing an MURAM partition, the user is normally required to pass its handle to other FMan driver modules. In this way, these modules may allocate and de-allocate memory blocks from this partition.

8.2.5.2.13 Supported Network Protocols

The following sections show the protocols that may be selected when defining NetEnv characteristics.

8.2.5.2.13.1 L2 Protocols

The following list shows the L2 protocols:

- `HEADER_TYPE_ETH`, with the following two options
    - `ETH_BROADCAST`
    - `ETH_MULTICAST`
- `HEADER_TYPE_VLAN`, with the following option
    - `VLAN_STACKED`
- `HEADER_TYPE_MPLS`, with the following option
    - `MPLS_STACKED`
- `HEADER_TYPE_PPPoE`

- HEADER_TYPE_LLC_SNAP

8.2.5.2.13.2 L3 Protocols

The following list shows the L3 protocols:

- HEADER_TYPE_IPV4, with the following options
    - IPV4_BROADCAST_1
    - IPV4_MULTICAST_1
    - IPV4_UNICAST_2
    - IPV4_MULTICAST_BROADCAST_2
    - IPV4_FRAG_1
- HEADER_TYPE_IPV6, with the following options
    - IPV6_MULTICAST_1
    - IPV6_UNICAST_2
    - IPV6_MULTICAST_2
    - IPV6_FRAG_1
- HEADER_TYPE_GRE
- HEADER_TYPE_MINENCAP
- HEADER_TYPE_USER_DEFINED_L3

8.2.5.2.13.3 L4 Protocols

The following list shows the L4 protocols:

- HEADER_TYPE_TCP
- HEADER_TYPE_UDP
- HEADER_TYPE_SCTP
- HEADER_TYPE_DCCP
- HEADER_TYPE_IPSEC_AH
- HEADER_TYPE_IPSEC_ESP
- HEADER_TYPE_USER_DEFINED_L4

8.2.5.2.13.4 Private Headers

- HEADER_TYPE_USER_DEFINED_SHIM1
- HEADER_TYPE_USER_DEFINED_SHIM2

8.2.5.2.13.5 Fields Supported By Driver for Keygen Extraction

Fields supported as "full fields":

- HEADER_TYPE_ETH
    - NET_HEADER_FIELD_ETH_DA
    - NET_HEADER_FIELD_ETH_SA
    - NET_HEADER_FIELD_ETH_TYPE

- HEADER_TYPE_LLC_SNAP
  - NET_HEADER_FIELD_LLC_SNAP_TYPE
- HEADER_TYPE_VLAN
  - NET_HEADER_FIELD_VLAN_TCI

    (index may apply:

    - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1,
    - e_FM_PCD_HDR_INDEX_LAST)
- HEADER_TYPE_MPLS
  - NET_HEADER_FIELD_MPLS_LABEL_STACK

    (index may apply:

    - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1,
    - e_FM_PCD_HDR_INDEX_2,
    - e_FM_PCD_HDR_INDEX_LAST)
- HEADER_TYPE_IPv4
  - NET_HEADER_FIELD_IPv4_SRC_IP
  - NET_HEADER_FIELD_IPv4_DST_IP
  - NET_HEADER_FIELD_IPv4_PROTO
  - NET_HEADER_FIELD_IPv4_TOS

    (index may apply:

    - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1,
    - e_FM_PCD_HDR_INDEX_2/e_FM_PCD_HDR_INDEX_LAST)
- HEADER_TYPE_IPv6
  - NET_HEADER_FIELD_IPv6_SRC_IP
  - NET_HEADER_FIELD_IPv6_DST_IP
  - NET_HEADER_FIELD_IPv6_NEXT_HDR
  - NET_HEADER_FIELD_IPv6_VER | NET_HEADER_FIELD_IPv6_FL | NET_HEADER_FIELD_IPv6_TC (must come together!)

    (index may apply:

    - e_FM_PCD_HDR_INDEX_NONE/e_FM_PCD_HDR_INDEX_1,
    - e_FM_PCD_HDR_INDEX_2/e_FM_PCD_HDR_INDEX_LAST)

    NOTE

    NET_HEADER_FIELD_IPv6_NEXT_HDR with e_FM_PCD_HDR_INDEX_LAST indication, applies to the very last next header indication, meaning the next L4, which may be present at the Ipv6 last extension. On earlier revisions this field applies to the Next-Header field of the main IPv6 header)
- HEADER_TYPE_IP
  - NET_HEADER_FIELD_IP_PROTO

    (index may apply:

    - 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)

8.2.6 Frame Manager Configuration Tool User's Guide

8.2.6.1 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.

8.2.6.2 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 595.)
- Generate C source files containing this information that you can include in your application. (See FMC Tool - Host Mode on page 596.)

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 595.)
- 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 596.)

8.2.6.3 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 613.

8.2.6.4 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 123. on page 596shows, 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 599 for more information.
- Custom Protocol file - Optional; user written; see Custom Protocol File on page 600 for more information.
- Policy file - Required; user written; see Policy file on page 601 for more information.
- Configuration file - Required; user written; see Configuration File on page 613 for more information.

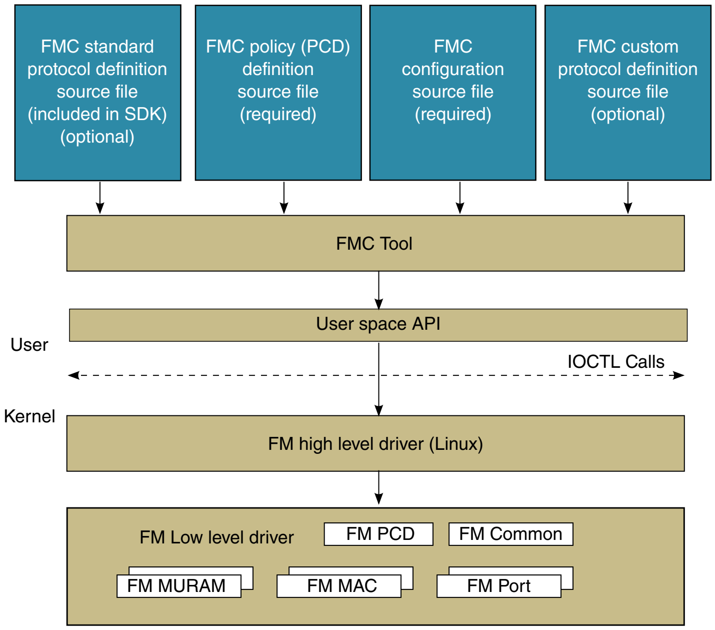

FMC standard protocol definition source file (included in SDK) (optional)
FMC policy (PCD) definition source file (required)
FMC configuration source file (required)
FMC custom protocol definition source file (optional)
FMC Tool
User space API
User
IOCTL Calls
Kernel
FM high level driver (Linux)
FM Low level driver
FM PCD
FM Common
FM MURAM
FM MAC
FM Port

Figure 123. FMC Tool, Runtime Environment - Input XML Files / FMan Driver API Calls

See FMC Tool Command-Line Arguments on page 598 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.

8.2.6.5 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 124. on page 597 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 597 for more information.)

- Standard Protocol File - Optional; included in LSDK; see Standard Protocol File on page 599 for more information.
- Custom Protocol File - Optional; user written; see Custom Protocol File on page 600 for more information.
- Policy File - Required; user written; see Policy file on page 601 for more information.
- Configuration File - Required; user written; see Configuration File on page 613 for more information.

You pass these files to the FMC Tool as command-line arguments.

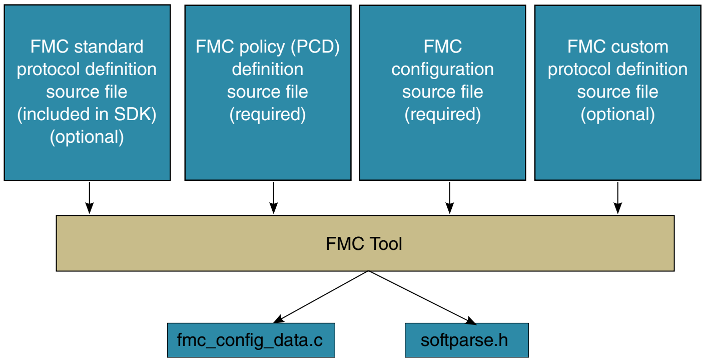

FMC standard protocol definition source file (included in SDK) (optional)
FMC policy (PCD) definition source file (required)
FMC configuration source file (required)
FMC custom protocol definition source file (optional)
FMC Tool
fmc_config_data.c
softparse.h

Figure 124. FMC Tool, Host Mode - Input XML Files / Generated C Source Code Files

See FMC Tool Command-Line Arguments on page 598 for documentation of each of the tool's command-line arguments.

8.2.6.5.1 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.

8.2.6.6 FMC Tool Command-Line Arguments

The table below lists and describes the FMC Tool's command-line arguments.

Table 103. 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 599 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 601 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 613 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 600 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)*
-x, --clean	Cleanup current configuration existing on target. This command cannot be used together with other commands and must be used alone. (Optional; valid only when FMC Tool is executed in runtime environment)

Table continues on the next page...

Table 103. FMC Tool Command-Line Arguments (continued)

Command-Line Argument Syntax ***(Both the verbose and abbreviated command forms are shown)***	**Description**
-f, --force	Applies the new configuration, without regard for the previous configuration stored by FMC tool. When used in runtime environment mode (that is, on the target), the information stored about the previous configuration is used by the tool to properly cleanup current configuration before applying the new one. There are cases such as a reboot when such cleanup is not necessary. In such cases, use the --force switch. (*Optional valid only when FMC Tool is executed in runtime environment*).
-w	Do not report warnings. *(Optional)*
--version	Display version information, then exit. *(Optional)*
-h, --help	Display usage information, then exit. *(Optional)*

8.2.6.7 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.

8.2.6.8 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.

8.2.6.8.1 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 662 topic to see a larger portion of the Standard Protocol file.

8.2.6.8.2 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 613 topic, which fully documents the custom version of NetPDL used in custom protocol definitions.

See Custom Protocol File - GTP Protocol Example on page 668, 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>
```

8.2.6.9 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 634 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 125. 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>
```

8.2.6.9.1 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 126. 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 637 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 127. on page 603 shows the algorithm the KeyGen sub block uses to calculate a FQID.

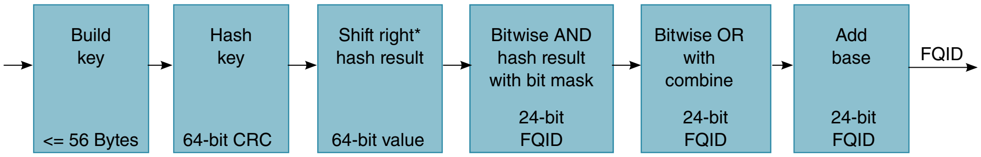


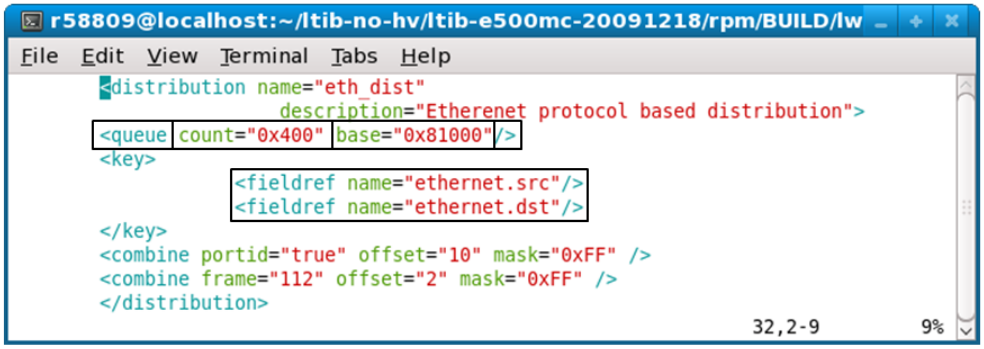
Figure 127. KeyGen Algorithm for FQID Calculation

* 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 128. on page 603 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
32,2-9
9%

```
        <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>
```

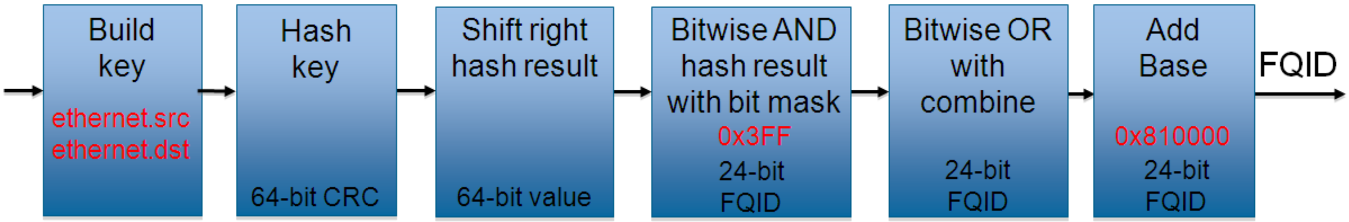
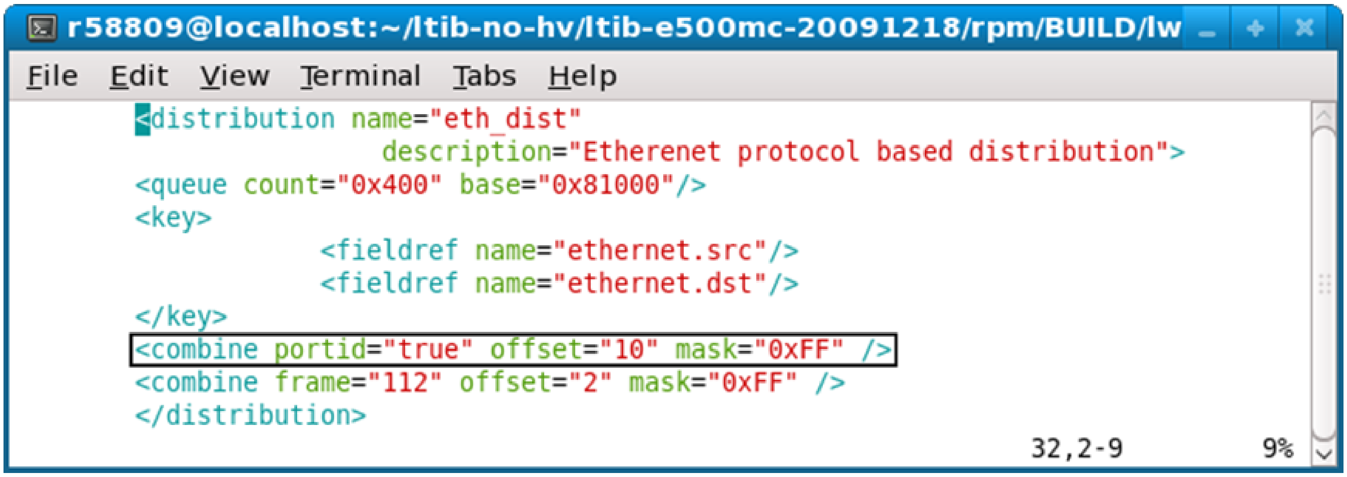


Figure 128. FQID Calculation - Elements/Attributes Used for Key, Bit Mask, and Base FQID

Figure 129. on page 604shows a 'combine' element that includes a 'portid' attribute that is set to "true". In addition, the element's 'offset' attribute is "10", and its 'mask' is "0xFF". This markup instructs the KeyGen sub block to perform the "bitwise OR" part of the FQID calculation. In more detail, for this markup, the KeyGen does these things:

- Bitwise ANDs the 8-bit logical port ID (defined in the Configuration file) of the port on which the current frame arrived with the 8-bit mask in the 'combine' element.
- Bitwise ORs (inserts) the 8-bit result at the specified offset (10 bits) within the 24-bit FQID (where offset 0 signifies the FQID's most significant bit).

Note: Each FMan port can be assigned an 8-bit logical port ID by adding markup to the Configuration file. To do this, assign an 8-bit value to the 'portid' attribute of each 'port' element to which you want to assign a logical port ID. The Hard Parser puts this value (if defined) in the parse results array, where the a KeyGen sub block can get it.



r58809@localhost:~/ltib-no-hv/ltib-e500mc-20091218/rpm/BUILD/lw
File Edit View Terminal Tabs Help

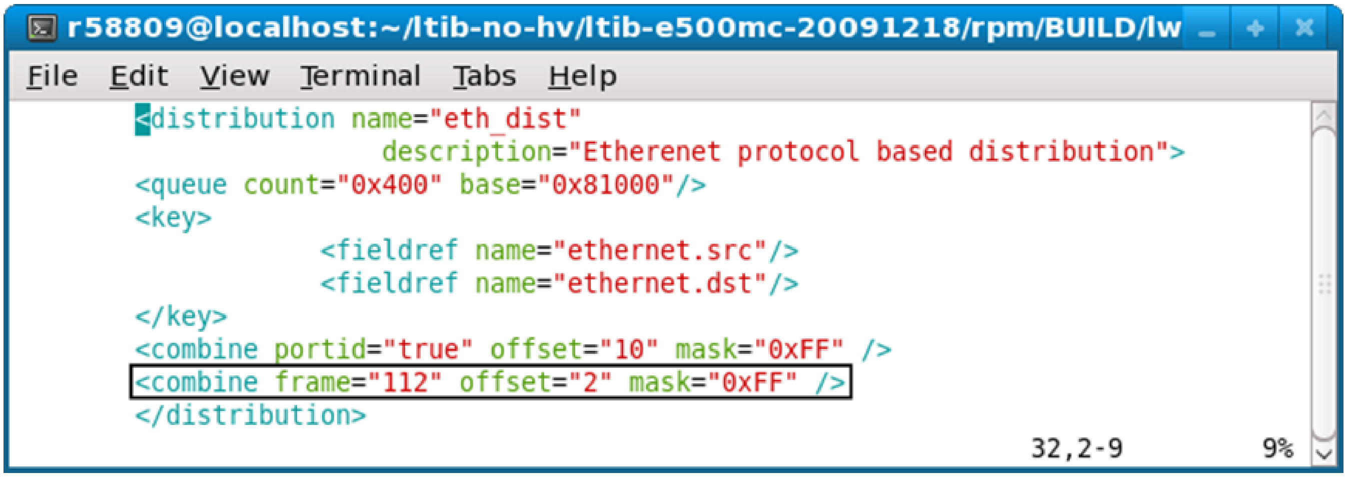
```
        <distribution name="eth_dist"
                        description="Etherenet protocol based distribution">
        <queue count="0x400" base="0x81000"/>
        <key>
                    <fieldref name="ethernet.src"/>
                    <fieldref name="ethernet.dst"/>
        </key>
        <combine portid="true" offset="10" mask="0xFF" />
        <combine frame="112" offset="2" mask="0xFF" />
        </distribution>
                                                                   32,2-9        9%
```

Figure 129. FQID Calculation - A 'combine' Element that Uses the 'portid' Attribute

Figure 130. on page 604 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.

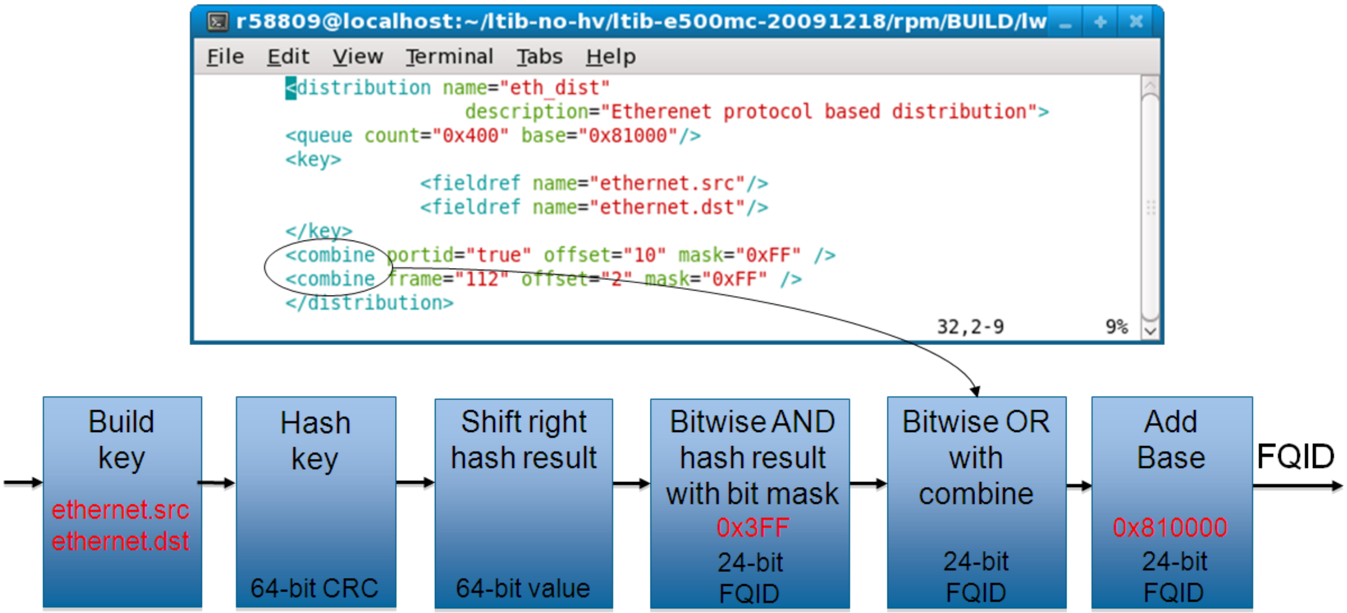


r58809@localhost:~/ltib-no-hv/ltib-e500mc-20091218/rpm/BUILD/lw
File Edit View Terminal Tabs Help

```
        <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 130. FQID Calculation - A 'combine' Element that Uses the 'frame' Attribute

Finally, Figure 131. on page 605 shows where the KeyGen sub block plugs the values from each of the combine elements into the bitwise OR part of the FQID calculation.



```
<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 131. 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.

8.2.6.9.2 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 635 for complete documentation of this element.

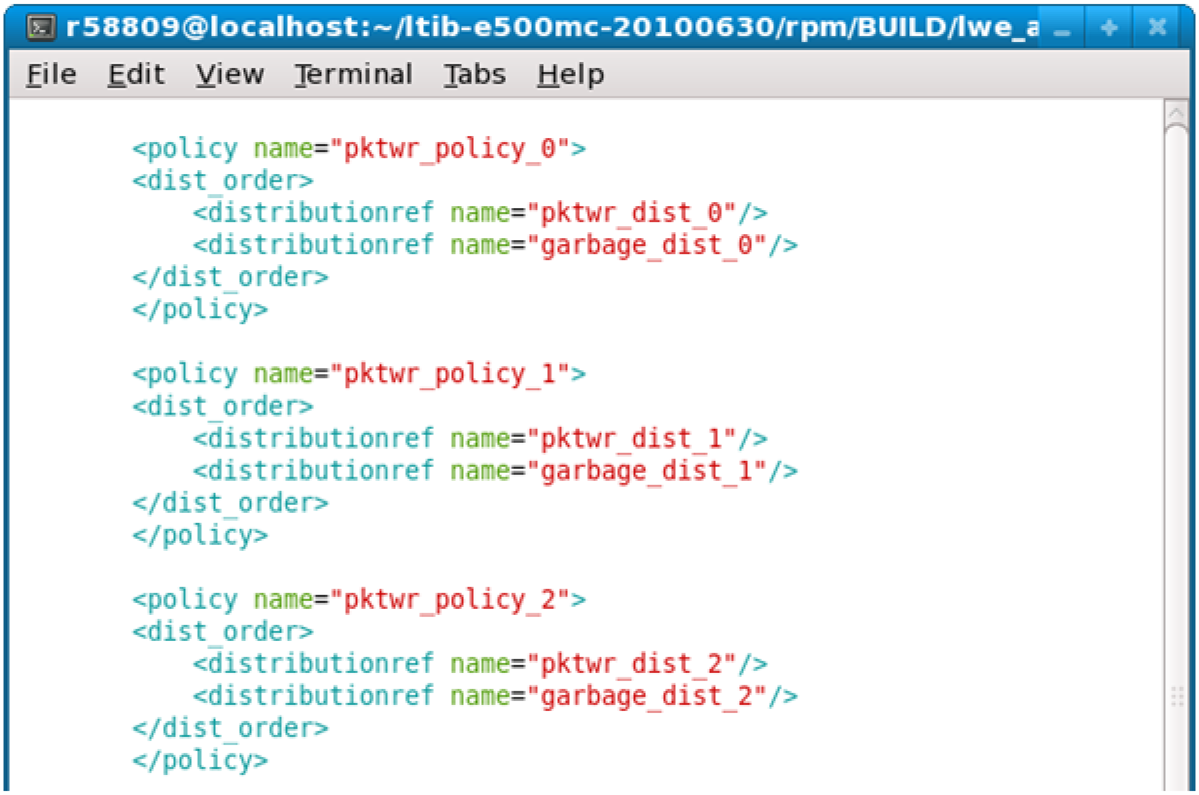
Example 2 - More Complex Use of the Policy Element

Figure 132. on page 607 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.


r58809@localhost:~/ltib-e500mc-20100630/rpm/BUILD/lwe_a
File Edit View Terminal Tabs Help

```
        <policy name="pktwr_policy_0">
        <dist_order>
            <distributionref name="pktwr_dist_0"/>
            <distributionref name="garbage_dist_0"/>
        </dist_order>
        </policy>

        <policy name="pktwr_policy_1">
        <dist_order>
            <distributionref name="pktwr_dist_1"/>
            <distributionref name="garbage_dist_1"/>
        </dist_order>
        </policy>

        <policy name="pktwr_policy_2">
        <dist_order>
            <distributionref name="pktwr_dist_2"/>
            <distributionref name="garbage_dist_2"/>
        </dist_order>
        </policy>
```

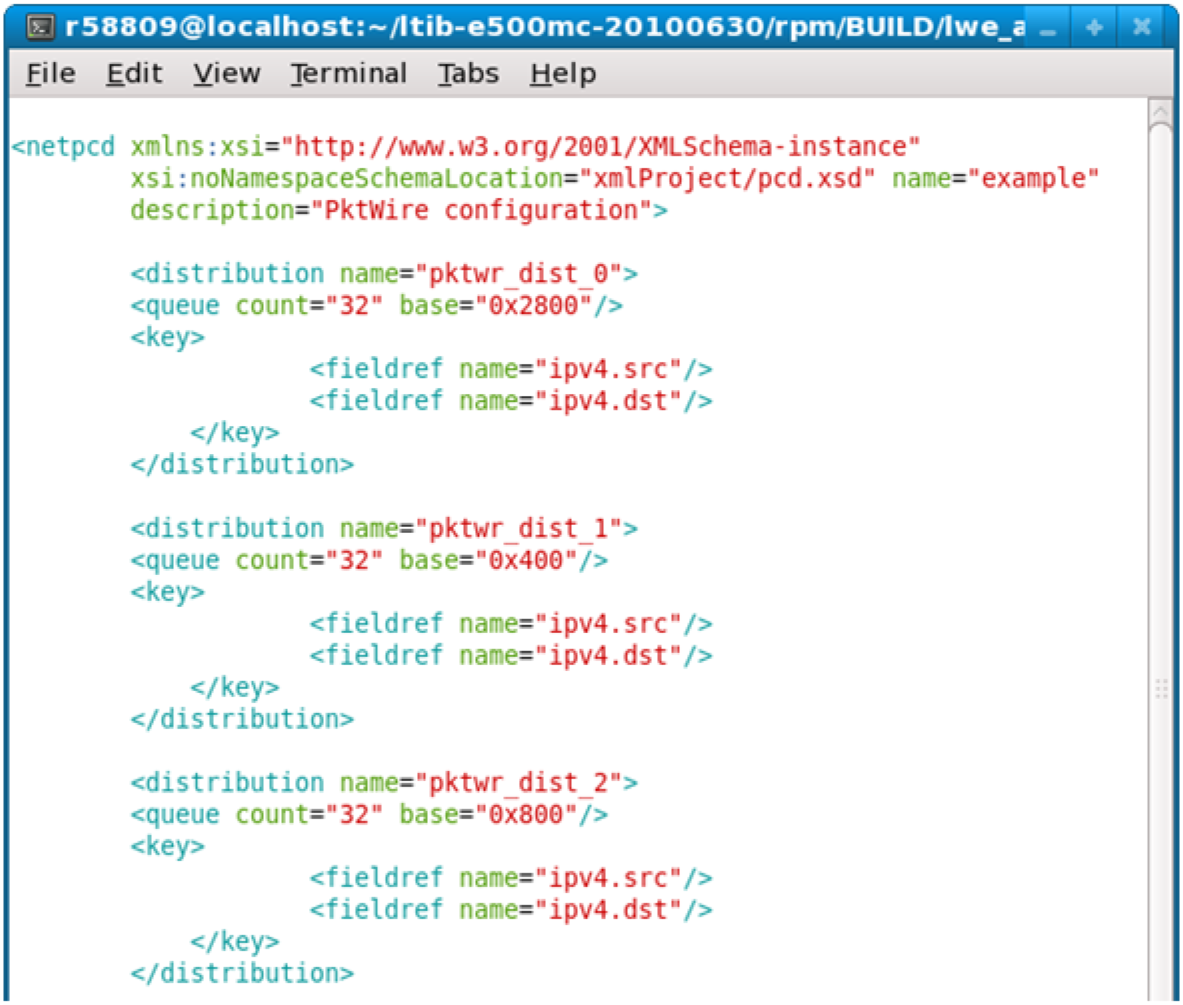
Figure 132. More Complex Policy File - 1

The Policy file also defines ten distributions - pktwr_dist_0, pktwr_dist_1, … pktwr_dist_9 - some of which are shown in Figure 133. on page 608.

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 602 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.



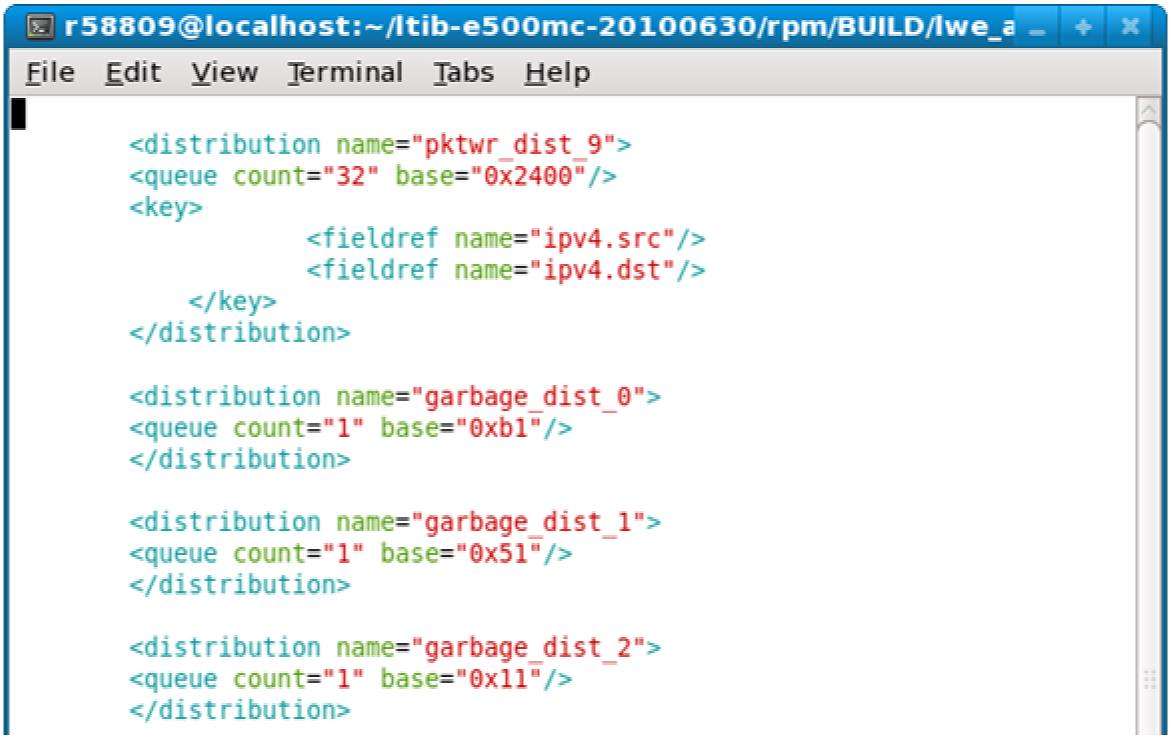
r58809@localhost:~/ltib-e500mc-20100630/rpm/BUILD/lwe_a
File Edit View Terminal Tabs Help

```
<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 133. 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 134. on page 609.

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>
```

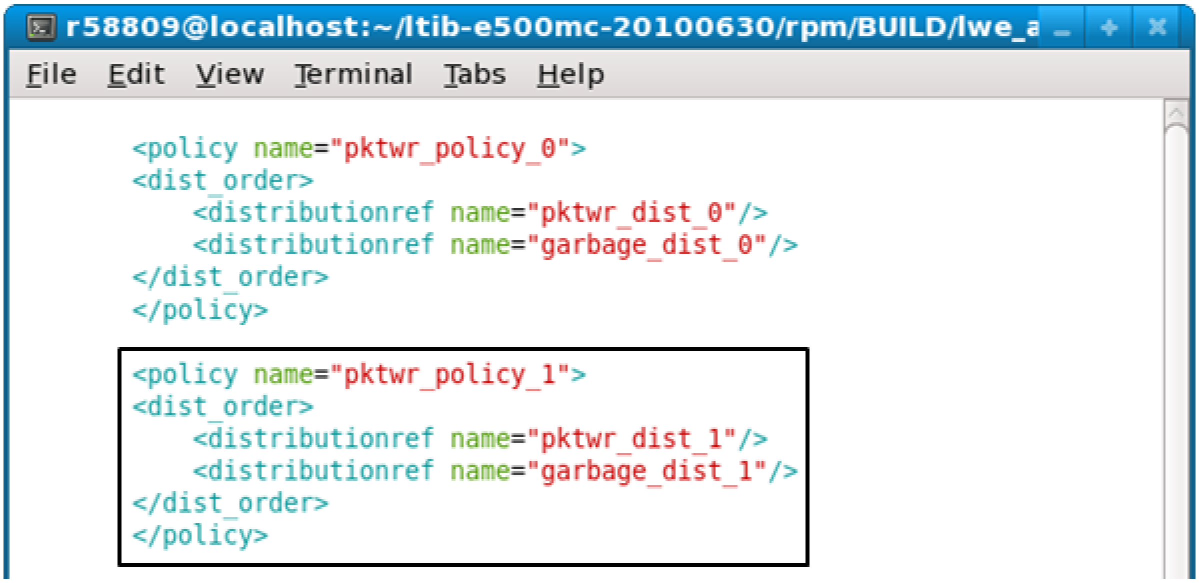
Figure 134. More Complex Policy File - 3

Let’s say that an FMan port is tied to policy 'pktwr_policy_1' - highlighted in Figure 135. on page 610.

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 135. More Complex Policy File - 4

8.2.6.9.3 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 645 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%--
```

8.2.6.9.4 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[13]
- CBS, Committed Burst Size[14]

[13] If "unit" attribute is "packet" specify CIR and PIR in packets/second. If "unit" attribute is "byte" specify CIR and PIR in Kbits/second

- PIR, Peak Information Rate[15]
- PBS, Peak Burst Size[16]

In all three modes, you can specify the next invoked action (NIA) for each color result (drop the frame, proceed to the specified distribution, etc.)

Example 1 - Policer Markup for RFC2698 Mode

```
<policer name="policer2">
  <algorithm>rfc2698</algorithm>

  <color_mode>color_aware</color_mode>

  <CIR>12000</CIR>
  <EIR>34000</EIR>
  <CBS>56000</CBS>
  <EBS>78000</EBS>

  <unit>byte</unit>

  <action condition="on-green"  type="distribution" name="green_dist"/>
  <action condition="on-yellow" type="distribution" name="yellow_dist"/>
  <action condition="on-red"    type="drop"/>
</policer>
```

Example 2 - Policer Markup for Pass-through Mode

```
<policer name="“vlan_congestion_control_green">
  <algorithm>pass_through</algorithm>

  <color_mode>color_blind</color_mode>

  <default_color>green</default_color>

  <action condition="on-green" type="distribution name="default_dist"/>
</policer>

<policer name="“vlan_congestion_control_yellow">
  <algorithm>pass_through</algorithm>

  <color_mode>color_blind</color_mode>

  <default_color>yellow</default_color>

  <action condition="on-yellow" type="“drop"/>
</policer>

<policer name="“vlan_congestion_control_red">
  <algorithm>pass_through</algorithm>

  <color_mode>color_blind</color_mode>

  <default_color>red</default_color>

  <action condition="on-red" type="“drop"/>
</policer>
```

[14] If "unit" attribute is "packet" specify CBS and PBS in packets. If "unit" attribute is "byte" specify CBS and PBS in bytes.

[15] If "unit" attribute is "packet" specify CIR and PIR in packets/second. If "unit" attribute is "byte" specify CIR and PIR in Kbits/second

[16] If "unit" attribute is "packet" specify CBS and PBS in packets. If "unit" attribute is "byte" specify CBS and PBS in bytes.

8.2.6.10 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 136. on page 613 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 136. on page 613 shows an example configuration file. It uses the optional 'portid' attribute for the 1 Gbps ports.

Figure 136. 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>
```

8.2.6.11 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.

8.2.6.11.1 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"/>
```

8.2.6.11.2 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>
```

8.2.6.11.3 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 104. Valid values for the prevproto attribute on page 614 lists the values that you can assign to the 'prevproto' attribute.

Table 104. Valid values for the prevproto attribute

Protocol	Layer
ethernet	2
llc_snap	2
vlan	2
pppoe	2
mpls	2
ipv4	3

Table continues on the next page...

Table 104. Valid values for the prevproto attribute (continued)

Protocol	Layer
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 616.	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>
```

8.2.6.11.3.1 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.

8.2.6.11.4 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

8.2.6.11.4.1 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

8.2.6.11.4.2 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

8.2.6.11.5 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>
```

8.2.6.11.5.1 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 616 for more information.

8.2.6.11.5.2 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 633) 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>
```

8.2.6.11.5.3 Child Elements of the before and after Elements

8.2.6.11.5.3.1 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"/>
```

8.2.6.11.5.3.2 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 632 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>
```

8.2.6.11.5.3.2.1 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>
```

8.2.6.11.5.3.2.2 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>
```

8.2.6.11.5.3.3 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>
```

8.2.6.11.5.3.3.1 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>
```

8.2.6.11.5.3.3.2 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>
```

8.2.6.11.5.3.4 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 643 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 105. Parse Action for each Value of the nextproto Attribute

If nextproto is ...	The parse action is ...
ethernet	Jump to the Ethernet header and continue hard parsing
llc_snap	Jump to the LLC_SNAP header and continue hard parsing
vlan	Jump to the VLAN header and continue hard parsing
pppoe	Jump to the PPPoE header and continue hard parsing
mpls	Jump to the MPLS header and continue hard parsing
ipv4	Jump to the IPv4 header and continue hard parsing
ipv6	Jump to the IPV6 header and continue hard parsing
gre	Jump to the GRE header and continue hard parsing
minencap	Jump to the MinEncap header and continue hard parsing
otherl3	Jump to the otherl3 header and continue hard parsing
tcp	Jump to the TCP header and continue hard parsing
udp	Jump to the UDP header and continue hard parsing
ipsec_ah	Jump to the IPsec_ah header and continue hard parsing
ipsec_esp	Jump to the IPsec_esp header and continue hard parsing
sctp	Jump to the SCTP header and continue hard parsing
dccp	Jump to the DCCP header and continue hard parsing
otherl4	Jump to the otherl4 header and continue hard parsing
after_ethernet	Jump to the protocol that should follow the Ethernet header. The next protocol is determined from the value of the $nxtHdr variable. See Table 106. Next Protocol for each $nxtHdr Value if nextproto is 'after_ethernet' on page 623to 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'.

Table continues on the next page...

Table 105. Parse Action for each Value of the nextproto Attribute (continued)

If nextproto is ...	The parse action is ...
return (default value)	Return to the Hard Parser without advancing the frame window. In this case, the Hard Parser starts parsing the frame header at the same position at which the Soft Parser began. The 'advance' attribute cannot be 'yes' when 'nextproto is set to return.
none/end_parse	Finish parsing the frame header; do not return to the Hard Parser.

Table 106. 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 107. 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"/>
```

8.2.6.11.6 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').

8.2.6.11.6.1 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).

8.2.6.11.6.1.1 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).

8.2.6.11.6.1.2 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>
```

8.2.6.11.6.1.3 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.

8.2.6.11.6.1.3.1 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 108. Result Array Variables on page 625shows the available result array variables .

Table 108. 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

Table continues on the next page...

Table 108. Result Array Variables (continued)

Variable Name	Result Array Bytes Referenced
rhp	31-31
ipvalid	31-31
shimoffset_1	32-32
shimoffset_2	33-33
ip_pidoffset	34-34
ethoffset	35-35
llcs_napoffset	36-36
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

Table continues on the next page...

Table 108. Result Array Variables (continued)

Variable Name	Result Array Bytes Referenced
hpnia	77-79
sperc	80-80
ipver	85-85
iplength	86-87
icp	90-91
attr	92-92
nia	93-95
ipv4sa	96-99
ipv4da	100-103
ipv6sa1	96-103
ipv6sa2	104-111
ipv6da1	112-119
ipv6da2	120-127

Note: The $GPR2 variable is used internally by the FMC Tool to calculate complex expressions, including checksum calculations. Using $GPR2 for other purposes is possible, but is not supported or recommended.

8.2.6.11.6.1.3.2 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.

8.2.6.11.6.1.3.3 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.

8.2.6.11.6.1.3.4 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>
```

8.2.6.11.6.1.3.5 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 109. 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

Table continues on the next page...

Table 109. Previous Protocol RA Return Values (continued)

If the previous protocol is ...	The value returned from result array is ...
pppoe	$pppoeoffset
tcp, udp, sctp, dccp, ipsec_ah, ipsec_esp	$l4offset
vlan	$vlanoffset_n
otherl3, otherl4	$NxtHdrOffset - When the previous protocol is otherl3 or other l4, the custom protocol and the previous protocol have the same offset. See Effect of Setting prevproto Attribute to otherl3 or otherl4 on page 616.

8.2.6.11.6.2 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 633 and Logical Expressions on page 632 for more information.)

Table 110. Supported Operators and their Properties on page 629describes 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 110. 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	+

Table continues on the next page...

Table 110. Supported Operators and their Properties (continued)

Name	Parameters	Description	Symbol
Subtract	32-bit arithmetic (32-bit Arithmetic, 32-bit arithmetic)	Return the difference between the two expressions (result of subtraction)	-
Add carry	16-bit arithmetic (16-bit arithmetic, 16-bit arithmetic)	Return the sum of the two expressions summed with the carry after 32bit	addc
Bitwise OR	Arithmetic (Arithmetic, Arithmetic)	Returns the result of a bitwise OR operation on the two expressions	bitwor
Bitwise XOR	Arithmetic (Arithmetic, Arithmetic)	Returns the result of a bitwise XOR operation on the two expressions	bitwxor
Bitwise AND	Arithmetic (Arithmetic, Arithmetic)	Returns the result of a bitwise AND operation on the two expressions	bitwand
Bitwise NOT	Arithmetic (Arithmetic)	Returns the result of a bitwise NOT operation on the expression	bitwnot
Shift left	Arithmetic (Arithmetic, Integer - value up to 64 bits)	Return the left expression shifted left by the right expression	shl
Shift right	Arithmetic (Arithmetic, Integer - value up to 64 bits)	Return the left expression shifted right by the right expression	shr
Concat	Arithmetic (Arithmetic, Variable or Integer)	Special operator See The concat Operator on page 630 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 631 for full documentation	checksum

8.2.6.11.6.2.1 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">
```

8.2.6.11.6.2.2 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)
```

8.2.6.11.6.2.3 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.

8.2.6.11.6.2.4 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

8.2.6.11.6.2.5 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">`

8.2.6.11.6.3 Expression Types

There are two main types of expressions: Logical expressions, which return "true" or "false", and arithmetic expressions, which return a numeric result.

8.2.6.11.6.3.1 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)`

8.2.6.11.6.3.2 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`

8.2.6.11.7 Tips and Recommendations

8.2.6.11.7.1 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 617, The after Element on page 618, and The action Element (for use in a Custom Protocol file) on page 621 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.

8.2.6.11.7.2 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.

8.2.6.11.7.3 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">
```

8.2.6.11.8 Limitations

This section discusses limitations you should consider when working with the FMC Tool's Soft Parser functionality.

8.2.6.11.8.1 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.

8.2.6.12 NetPCD Reference

8.2.6.12.1 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.

8.2.6.12.1.1 netpcd Attribute Definitions

Table 111. 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

8.2.6.12.1.2 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>
```

8.2.6.12.2 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.

8.2.6.12.2.1 policy Attribute Definitions

Table 112. 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.

8.2.6.12.2.2 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>
```

8.2.6.12.3 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.

8.2.6.12.3.1 dist_order Attribute Definitions

Table 113. dist_order Attribute Definitions

Attribute	Requirement	Description
none	n/a	n/a

8.2.6.12.3.2 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.

8.2.6.12.4 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.

8.2.6.12.4.1 distributionref Attribute Definitions

Table 114. distributionref Attribute Definitions

Attribute	Requirement	Description
name	required	Name of the referenced 'distribution' element

8.2.6.12.4.2 distributionref Example

```
<policy name="ipv4">
  <dist_order>
    <distributionref name="eth_dist"/>
    <distributionref name="default_dist"/>
  </dist_order>
</policy>
```

8.2.6.12.5 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.

8.2.6.12.5.1 distribution Attribute Definitions

Table 115. 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.

8.2.6.12.5.2 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>
```

8.2.6.12.5.3 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 116. '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 117. '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

Table continues on the next page...

Table 117. 'default' Element Attributes: (continued)

Attribute	Requirement	Description
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>
```

8.2.6.12.6 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.

8.2.6.12.6.1 key Attribute Definitions

Table 118. 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.

8.2.6.12.6.2 key Example

```
<key shift="16">
  <fieldref name="ethernet.src"/>
  <fieldref name="ethernet.dst"/>
</key>
```

8.2.6.12.7 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/.

8.2.6.12.7.1 fieldref Attribute Definitions

Table 119. 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".

8.2.6.12.7.2 fieldref Example

```
<key>
  <fieldref name="ethernet.src"/>
  <fieldref name="ethernet.dst"/>
</key>
```

8.2.6.12.8 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.

8.2.6.12.8.1 queue Attribute Definitions

Table 120. 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.

8.2.6.12.8.2 queue Example

```
<distribution name="eth_dist">
  <queue count="0x400" base="0x810000"/>
  <key>
    <fieldref name="ethernet.src"/>
    <fieldref name="ethernet.dst"/>
  </key>
</distribution>
```

8.2.6.12.9 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.

8.2.6.12.9.1 protocols and protocolref Attribute Definitions

Table 121. 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 122. Protocol options. Groups are separated by empty rows.

Value	Description
0x80000000	Ethernet Broadcast
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

Table continues on the next page...

Table 122. Protocol options. Groups are separated by empty rows. (continued)

Value	Description
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).

8.2.6.12.9.2 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>
```

8.2.6.12.10 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.

8.2.6.12.10.1 combine Attribute Definitions

Table 123. 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.

8.2.6.12.10.2 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>
```

8.2.6.12.11 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.

8.2.6.12.11.1 action Attribute Definitions

Table 124. 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"

8.2.6.12.11.2 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 125. 'statistics' Element Attributes:

Attribute	Requirement	Description
statistics	optional	Enable statistics for a particular action. Possible values are: • enable/yes/true – to enable it. • disable

8.2.6.12.11.3 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>
```

8.2.6.12.12 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

8.2.6.12.12.1 classification Attribute Definitions

Table 126. classification Attribute Definitions

Attribute	Requirement	Description
name	required	The name of the classification

8.2.6.12.12.2 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 127. 'statistics' Element Attributes:

Attribute	Requirement	Description
statistics	optional	Choose statistic mode for the particular entry. Possible values are: • none • frame • byteframe • rmon

8.2.6.12.12.3 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>
```

8.2.6.12.12.4 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 128. '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>
```

8.2.6.12.12.5 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 129. '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).

8.2.6.12.12.6 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>
```

8.2.6.12.12.7 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 130. 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 131. '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

8.2.6.12.13 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

8.2.6.12.13.1 entry Attribute Definitions

Table 132. entry Attribute Definitions

Attribute	Requirement	Description
none	n/a	n/a

8.2.6.12.13.2 entry Example

```
<classification name="eth_dest_clsf">
  <key>
    <fieldref name="ethernet.dst"/>
  </key>

  <entry>
    <data>0x1234567890AB1234567890AB</data>
    <queue base="0x550000"/>
  </entry>
</classification>
```

8.2.6.12.14 The policer element

The 'policer' element is a container whose child elements define a policer profile that performs network bandwidth management.

8.2.6.12.14.1 policer Attribute Definitions

Table 133. 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".

Table continues on the next page...

Table 133. policer Attribute Definitions (continued)

Attribute	Requirement	Description
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.

8.2.6.12.14.2 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>
```

8.2.6.12.15 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.

8.2.6.12.15.1 nonheader Attribute Definitions

Table 134. 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
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.

8.2.6.12.15.2 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>
```

8.2.6.12.16 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 135. '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>
```

8.2.6.12.17 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.

8.2.6.12.17.1 vsp Attributes

Table 136. '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.	--

Table continues on the next page...

Table 136. 'vsp' Element Attributes: (continued)

Attribute	Requirement	Description
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.

8.2.6.12.17.2 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>
```

8.2.6.12.18 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.

8.2.6.12.18.1 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 137. 'fragmentation' Element Attributes:

Attribute	Requirement	Description
name	required	Name of the element. The name is used to refer the manipulation algorithm.

Table 138. '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

Table continues on the next page...

Table 138. 'fragmentation' Child Elements: (continued)

Attribute	Requirement	Description
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>
```

8.2.6.12.18.2 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 139. 'reassembly' Element Attributes:

Attribute	Requirement	Description
Name	required	Name of the element. The name is used to refer the manipulation algorithm

Table 140. '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

Table continues on the next page...

Table 140. 'reassembly' Child Elements: (continued)

Attribute	Requirement	Description
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.
numOfFramesPerHashEntry (numOfFramesPerHashEntry1)	required	Number of frames per hash entry needed for reassembly process – for ipv4. Possible values are: numeric values from 1 to 8.
numOfFramesPerHashEntry2	optional	Number of frames per hash entry needed for reassembly process – for ipv6. Possible values are: numeric values from 1 to 6.
timeoutThreshold	required	Represents the time interval in microseconds which defines if opened frame (at least one fragment was processed but not all the fragments)is found as too old
nonConsistentSpFqid	optional	Handles the case when other fragments of the frame corresponds to a different storage profile than the opening fragment. (DPAA >= 1.1 only). Default is 0

Here is an example of possible IP reassembly definition:

```
<manipulations>
    <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>
```

8.2.6.12.18.3 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 141. 'header' Element Attributes:

Attribute	Requirement	Description
name	required	Name of the element. The name is used to refer the manipulation algorithm
parse	optional	Activate the parser a second time after completing the manipulation of the frame (if ‘yes’)
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.

8.2.6.12.18.3.1 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 142. '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

8.2.6.12.18.3.2 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 143. '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.

8.2.6.12.18.3.3 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 144. 'insert_header' Element Attributes

Element	Requirement	Description
type	required	The type of the header inserted. Only ‘mpls’ is valid at this time.
header_index	optional	The header index of the header has possible values "1" and "2". The restrictions on this attribute are: • 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 145. '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>
```

8.2.6.12.18.3.4 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 146. '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>
```

8.2.6.12.18.3.5 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 147. 'remove_header' Child Elements

Element	Requirement	Description
type	required	The type of the update. Possible values: • "vlan" • "ipv4" • "ipv6" • "tcpudp"

update Child Elements:

Table 148. '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 149. '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>
```

8.2.6.12.18.3.6 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 150. '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 151. 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>0x45000000123400000001000010111213141516 17</data>
     </custom>
</header>
```

8.2.6.12.18.3.7 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 152. Nextmanip element attributes

Element	Requirement	Description
name	required	The name of the next header manipulation

8.2.6.12.18.3.8 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>
```

8.2.6.13 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 -->
```

8.2.6.14 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>
```

8.2.7 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 365, 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 365

DPAA2-specific SEC details - Data Path SEC Interface (DPSECI):Security Engine (SEC) on page 731

8.2.8 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.

8.3 DPAA2-specific Software

8.3.1 DPAA2 Software Overview

8.3.1.1 Introduction

The following section provides an overview of the software and tools for the DPAA2 networking hardware that is provided on NXP SoCs such as LS2088A and LS1088A. These SoCs are called "DPAA2 SoCs" because they contain the hardware that is required to support the DPAA2 networking architecture. This hardware includes Queue Manager/Buffer Manager (QBMan), the Wire Rate I/O Processor (WRIOP), and the Management Complex (MC).

DPAA2 is an architecture in which some facilities (and thus the hardware that supports them) are optional. For this reason, this document may describe features that are not available on all DPAA2 SoCs.

DPAA2 in the NXP Layerscape SDK

NXP provides a Linux-based software development kit (SDK) for SoCs. The core of the SDK is an embedded-oriented Linux distribution containing components such as:

- U-Boot boot loader
- Linux kernel with networking support
- GNU tool chain for ARMv8™
- Large set of standard Linux user space packages including shells, initialization scripts, servers, etc.
- Yocto-based package management in an embedded-style source-based Linux distribution

NXP supports and builds upon standard Linux with drivers and additional packages and capabilities including support for the DPAA2 networking hardware such as:

- Management complex firmware for the DPAA2 architecture. DPAA2 is a networking peripheral subsystem architecture and will be discussed at length in later sections.
- Restool: a DPAA2 object management tool
- A DPAA2 Linux Ethernet driver
- Linux kernel support for treating DPAA2 containers as plug-and-play buses with VFIO support
- Integrated kernel-based control of DPAA2 L2 switch objects
- Kernel support for DPAA2 acceleration objects including cryptographic offload
- And more

8.3.1.2 DPAA2 Hardware

8.3.1.2.1 Introduction

This section introduces the DPAA2 hardware components and explains their relationship to the DPAA hardware found on previous NXP SoCs. Finally, it shows the DPAA2 hardware blocks in the context of a specific SoC, LS2088A, LS1088A.

Note that the DPAA2 hardware is configured via DPAA2 objects as will be described below. This section on hardware provides background information to give context to the discussion of the DPAA2 objects. Most developers will deal with the DPAA2 objects and not directly with all aspects of the DPAA2 hardware blocks.

8.3.1.2.2 DPAA2 hardware

The DPAA2 hardware provides network interfaces, hardware-based queuing, layer 2 switching, more general switching, networking-related accelerators, and also memory dedicated to packet processing.

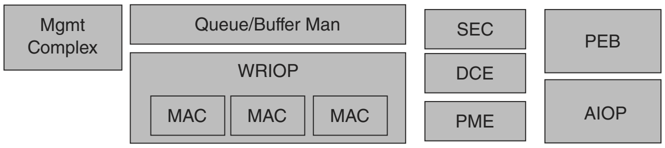


Figure 137. DPAA2 hardware components

The DPAA2 hardware contains the following components:

Management Complex (MC)

The DPAA2 hardware is abstracted by DPAA2 objects with the help of the Management Complex. This means that users need not study the details of the DPAA2 hardware blocks in order to develop drivers for or use DPAA2 capabilities. This software and solution oriented focus is one of the key differences between the first DPAA and DPAA2.

Queue and Buffer Manager (QBMan)

QBMan provides hardware-based buffer and queue management.

WRIOP

WRIOP provides hardware that serves as the basis for network interfaces. It includes Ethernet MACs, packet header key generators, parsers, table look up units, and an interface to the buffer and queue managers.

Accelerators (optional)

Accelerators that interface to QBMan are a key part of DPAA2. They include a cryptographic and security accelerator (SEC), a pattern matching accelerator (PME), a data compression/decompression accelerator (DCE), and a generic DMA engine. The set of accelerators may vary from SoC to SoC and new types of accelerators may be added.

PEB (optional)

PEB is a memory devoted to high-performance packet processing. It can be used to store in-flight packets and other items.

AIOP (optional)

AIOP is a fully programmable multicore engine with tightly coupled hardware accelerators that is specialized for efficient packet processing. It uses techniques somewhat similar to hardware multithreading to provide multiple "tasks" per core. The hardware supports efficient task switching to hide latencies associated with using accelerators and other hardware. The AIOP supports C language programming. It is optional in the DPAA2 architecture and thus is not available on all DPAA2 SoCs.

DPAA2 versus DPAA

DPAA2 is the latest generation of the Datapath Acceleration Architecture (DPAA) hardware. It is an evolution of the DPAA present in previous SoCs.

DPAA2 changes relative to DPAA include:

- DPAA2 contains a hardware block called the Management Complex. It facilitates and simplifies hardware resource allocation and hardware configuration.
- The hardware buffer and queue managers (QMan and BMan) are integrated into a single hardware block called QBMan.
- DPAA2 session context can be maintained per frame, rather than per frame queue, which allows multiple accelerator sessions to share a single frame queue pair. This single frame queue pair then reduces the number of frame queues needed, making session establishment more efficient because frame queues do not need to be initialized per session.
- Software portals are enhanced to make it easier and more efficient for General Purpose Processing (GPP) core software to share them.
- WRIOP in DPAA2 replaces FMan as the hardware block that provides Ethernet interfaces. WRIOP is designed to be more partitionable, in that it allows GPP software to more independently manage separate network interfaces.
- WRIOP and QBMan contain new features that support autonomous L2 switching functionality:
    - — WRIOP: L2 address learning and forwarding unit.
    - — QBMan: packet replication facility.
- WRIOP does not contain a generic programmable engine like the one present in FMan, and instead DPAA2 has a new hardware block called AIOP that is specifically designed to perform this function.

8.3.1.2.3 LS2088A block diagram

The LS2088A is an ARMv8-A 64-bit SoC. It contains eight ARM Cortex-A57 cores and numerous peripherals. The LS2088A is an example of a DPAA2 SoC because it contains the required DPAA2 hardware blocks: WRIOP, QBMan, and MC.

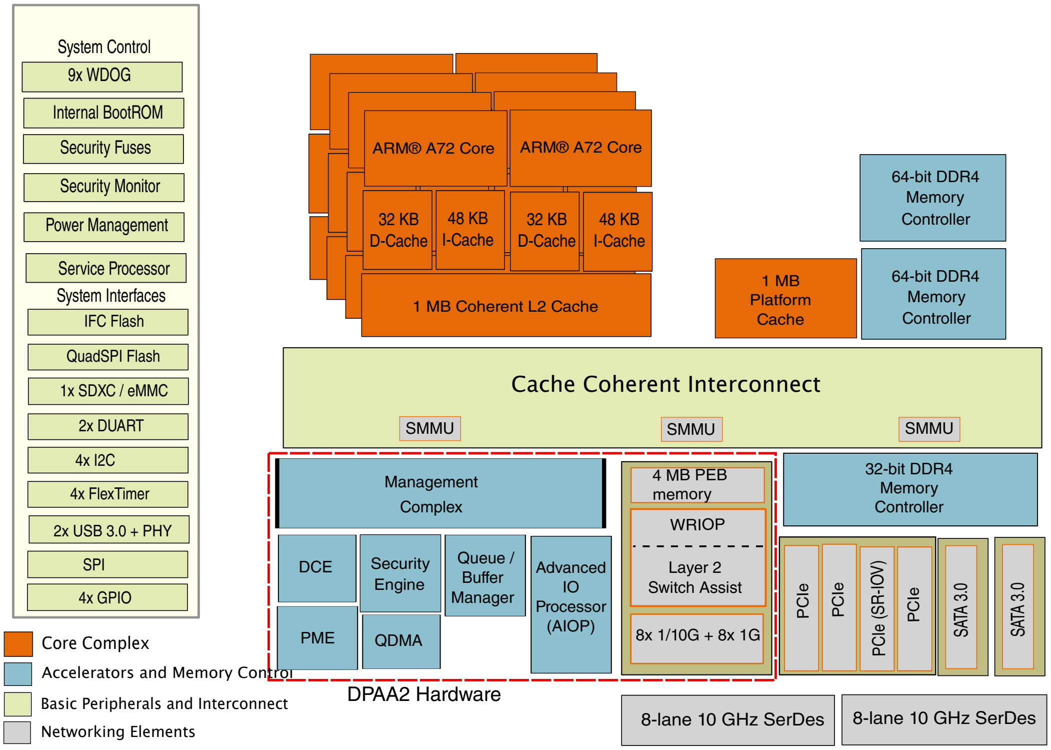


Figure 138. LS2088A SoC

The LS2088A contains standard-ARM components in addition to the cores, such as:

- ARM generic timer
- GIC-500 interrupt controller
- MMU-500 System Memory Management Unit (I/O MMU)

It also contains conventional hardware blocks including:

- DDR controllers
- Flash controller
- SDxC/eMMC controller
- USB controller
- PCIe controller
- SATA controller
- Other blocks visible in the diagram.

Finally, the following DPAA2 components are highlighted in the figure:

- QMan/BMan: hardware queue and buffer management
- WRIOP: Ethernet interfaces
- Management complex: DPAA2 objects and their management
- Accelerators: SEC, PME, and DCE

8.3.1.3 DPAA2 Linux Software

8.3.1.3.1 Introduction

This section provides a high-level summary of the most important DPAA2 software associated with the Linux operating system.

8.3.1.3.2 Linux and DPAA2

This section summarizes major Linux DPAA2 software. See Linux DPAA2 software which shows the software in relation to some standard Linux software.

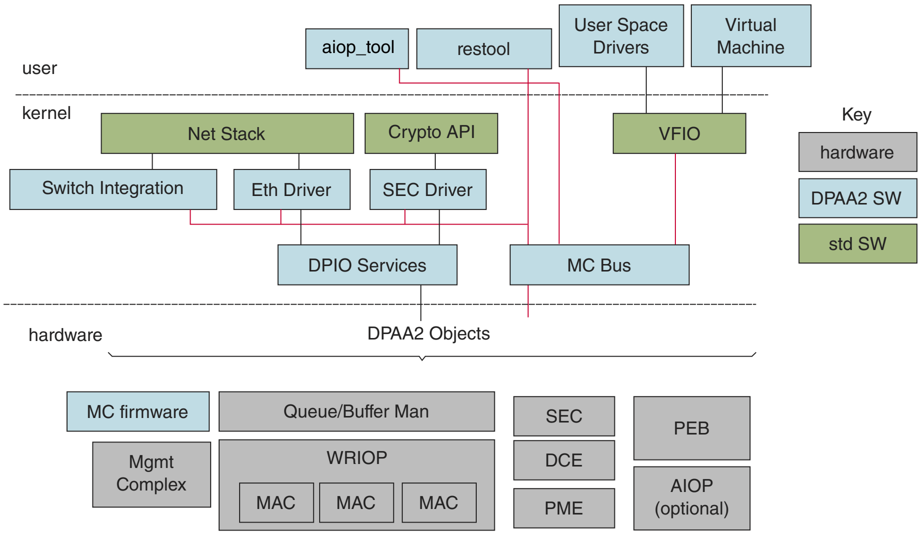


Figure 139. Linux DPAA2 software

Ethernet Driver

DPAA2 software includes a conventional Ethernet driver for use by the Linux network stack. This driver is controlled via standard Linux means such as the "ifconfig" or "ip" commands and also "ethtool". It operates in a manner that will be familiar to Linux users. Drivers in DPAA2 manage DPAA2 "objects" as will be described below. These objects are best regarded as hardware. They are formed from hardware resources.

DPIO Services

DPAA2 drivers such as the Ethernet driver in the Linux kernel use the DPIO services Linux component to do I/O. The DPIO services layer manages the kernel's DPIO objects. DPIO objects contain DPAA2 software portals (which are hardware components). The software portals can be shared by multiple higher-level drivers.

DPAA2 Objects and Management Complex (MC) Firmware

The DPAA2 hardware is presented to software in terms of DPAA2 objects that are realized by means of firmware running on the Management Complex. This will be explained in depth in DPAA2 Networking Subsystem Deeper Dive on page 679 and also immediately below.

MC Bus and Restool

The DPAA2 objects appear as devices on a special software-defined bus called the MC bus. Linux has a driver for this bus (and interactions with VFIO). This software is analogous to PCIe bus software. Like PCIe, the MC bus supports plug and play.

The "restool" utility is a Linux user space command that allows DPAA2 objects to be managed: created, destroyed, queried for status, etc.

SEC Driver

The SEC driver provides the standard Linux kernel cryptographic API but implemented by the SEC hardware by means of a special DPAA2 object. Other accelerators can be handled in the same way, but Linux tends to not provide standard (hardware-independent) kernel-level APIs for them so they are not discussed here.

Switch Integration

Finally, DPAA2 objects exist that perform L2 and more general network switching. These hardware elements can be configured using standard Linux mechanisms such as "bridge". As will be discussed later, there are two types of switch-related DPAA2 objects: DPSW and DPDMUX. There is kernel-based management support for both.

AIOP Tool

AIOP Tool (aiop_tool) is a user space command line utility that allows programs (images) to be loaded onto the AIOP and started. It also supports stopping and resetting the AIOP. Of course, aiop_tool is used only on DPAA2 SoCs that have an AIOP.

It is also possible for user space programs to manage these aspects of AIOP via programmatic means. The aiop_tool utility is most useful during development of AIOP software and for AIOP applications that happen not to be tightly integrated with control software running in user space on the GPP cores.

From the point of view of software running on the GPP cores, AIOP programs may be thought of as firmware that defines the functionality that the AIOP will provide to an overall system.

.

8.3.1.3.3 DPAA2, Management Complex, and drivers

DPAA2 is the architecture that describes network interfaces and other networking services for an SoC with DPAA2 hardware. It is discussed in depth in DPAA2 Networking Subsystem Deeper Dive on page 679. For now, think of DPAA2 as hardware for networking that is presented in terms of DPAA2 objects. The objects provide specific high-level features or services such as network interfaces or L2 switches.

The objects are managed by means of firmware running on a hardware block called the Management Complex. Software on general purpose cores must load firmware onto the Management Complex before networking can be done using DPAA2 hardware.

Normally, the MC firmware is loaded early in the boot process so that boot loaders can make use of DPAA2 objects and peform networking operations such as network-based booting.

Since the objects represent hardware, they require driver software on general purpose cores. NXP provides drivers for U-Boot and standard Linux and thus both support Ethernet networking out of the box. For example, one can use Linux networking without delving into the details of DPAA2 and its objects just as one can use Linux networking via a PCIe Ethernet card (whose manufacturer provides a driver) without delving into the design of the card.

DPAA2 and its objects are fully documented so it is possible to write drivers for other operating systems, applications, or boot loaders, e.g. DPDK, UEFI firmware, etc. Many of these drivers exist or are roadmap items.

8.3.1.3.4 DPAA2 and plug-and-play

There is another analogy between DPAA2 objects and PCIe devices. PCIe devices appear to operating systems as plug-and-play devices on a bus. The operating system can scan the bus to discover and identify the devices on it. It can then use the device identities to associate drivers with devices and bring them into service.

DPAA2 objects work in a similar way. They are placed into datapath containers (DPRC) that can be scanned in an analogous manner. Then objects are associated with drivers and placed into service.

The Linux kernel is provided with a container with its DPAA2 objects. Containers can also be provided to other software including virtual machines and even arbitrary user space processes. This is how the hardware that objects encapsulate can be directly assigned to virtual machines and user space processes. This allows them highly efficient access to hardware but in a secure fashion due to the involvement of the SoC IO-MMU.

This, also, is analogous to PCIe devices in standard Linux; DPAA2 objects can be directly assigned to virtual machines and user space processes using a standard Linux architecture called VFIO which allows devices to be mapped into the address space of user space processes and also enables IO-MMU configuration to constrain the memory to which devices can read and write data via their DMA engines.

Like PCIe devices, DPAA2 objects are also mapped using VFIO. NXP supplies the extensions to VFIO in Linux that makes this possible.

8.3.1.3.5 Datapath layout files and restool

As mentioned elsewhere, DPAA2 containers are like PCIe busses in that they can be scanned for objects/devices. But containers and PCIe busses are populated very differently. PCIe busses are populated physically, e.g. by plugging a card into a slot.

Objects are encapsulations of DPAA2 hardware resources that must be created via management complex firmware and then assigned to a container. There are several ways to do this:

1. the datapath layout file
2. restool
3. Management Complex commands

Datapath layout (DPL) file

Containers and objects can be defined statically in a file called a datapath layout file (DPL) that is passed to the management complex when it is initially booted. The DPL can specify containers, objects, and connections between objects. When an OS such as Linux boots, it will discover the populated containers.

restool

The utility called “restool” is a NXP-created Linux user space command that allows inspection and dynamic management of containers and objects. With it, one can

- Display the current set of containers and objects
- Create and destroy containers
- Create and destroy objects
- Assign objects to containers
- Create links among objects

One can use a sequence of restool command invocations to create the same container and object state that a DPL might specify. The difference is that restool is dynamic.

Management Complex commands

Finally, objects and containers can be manipulated by software running on general purpose cores by sending commands to the Management Complex. This is, in fact, what restool does. Command line arguments to restool define an operation. The restool utility simply forms a command and passes it to the Management Complex. Other drives can also do this.

8.3.1.4 DPAA2 Networking Subsystem Deeper Dive

This section provides additional detail on the DPAA2 architecture and the DPAA2 object services paradigm.

This paradigm simplifies using the DPAA2 hardware IP blocks through abstraction and encapsulation. DPAA2 objects are objects in the sense that they:

- Encapsulate specific abstract functionality, e.g. L2 switching.
- Are composed of allocated hardware sub-components of the DPAA2 hardware peripherals, and then mostly abstract their functionality .
- Present functionality in terms of specific attributes and methods, meaning operations on the objects.

NOTE

DPAA2 objects are not associated with object-oriented programming languages, instead they are collections of hardware resources allocated for a specific purpose. General purpose processing (GPP) core software can configure objects by sending them commands expressed in terms of hardware-level descriptors. GPP software can also include C language functions that prepare and interpret the descriptors. No use of object-oriented programming languages is required. For the most part, Linux drivers are written in C as usual

This section:

- Presents the DPAA2 object model at a concept level and describes how objects are created, destroyed, conveyed, configured, and used
- Lists the objects types and their purposes
- Outlines how the Management Complex implements and provides the objects
- Explains what software components use the various DPAA2 object types, and how they use them. The users are often application software running on general purpose processors (cores) or on the optional AIOP.

Driver-level software on GPPs works with the abstracted objects, rather than directly with the hardware. For example, the GPP software deals with L2 switch and network interface objects rather than WRIOPs .

DPAA2 objects express and abstract the DPAA2 hardware into software-managed objects that are:

- Application-oriented in terminology and use, rather than hardware-oriented
- Based on concepts that are generally familiar to programmers and system architects
- Simpler than direct management of the hardware
- Indicate the architectural intent of the hardware blocks

DPAA2 object services are provided by software that runs as firmware on a DPAA2 hardware block called the Management Complex. Users do not need to program the Management Complex in order to use the Network Object Services; they simply use the NXP-supplied firmware. This firmware runs on the Management Complex instead of a general purpose core in order to simplify the integration of the NXP software with customer software. DPAA2 object concept below shows at a concept level how the Management Complex provides objects that perform specific services; the objects have attributes and interfaces that appear as hardware.

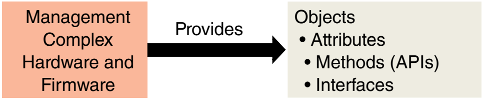


Figure 140. DPAA2 object concept

8.3.1.4.1 DPAA2 hardware abstraction example

This section introduces the DPAA2 objects and the abstractions they provide by means of an example. Example scenario shows a system constructed using the DPAA2 hardware on a DPAA2 SoC such as the LS2088A. The goal is to run two KVM virtual machines (VMs) on the SoC. The two virtual machines each have a hardware network interface that they can directly access (i.e. a dedicated interface) connected to a DPAA2 L2 switch. These VMs can communicate with each other via the L2 switch, and they can communicate externally via the MAC on the L2 switch. So, the L2 switch has three ports, one for an off-SoC connection (connected to a MAC), and two for the VMs.

In addition, there are two network interfaces with MAC addresses for off-SoC communication that are used by the host Linux. The host Linux instance and the virtual machines all run on the Cortex-A72 cores on the LS2088A. In this example, each network interface is associated with an Ethernet driver working with Linux.

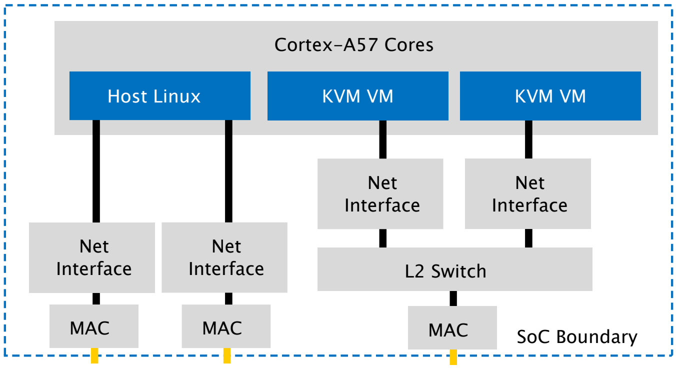


Figure 141. Example scenario

DPAA2 hardware shows the DPAA2 hardware blocks. This figure bears little resemblance to Example scenario. It provides little guidance to how the example scenario could be realized because the hardware blocks are conceptually distant from a natural statement of what is desired in the example. The DPAA2 objects are much closer, as will be seen below.

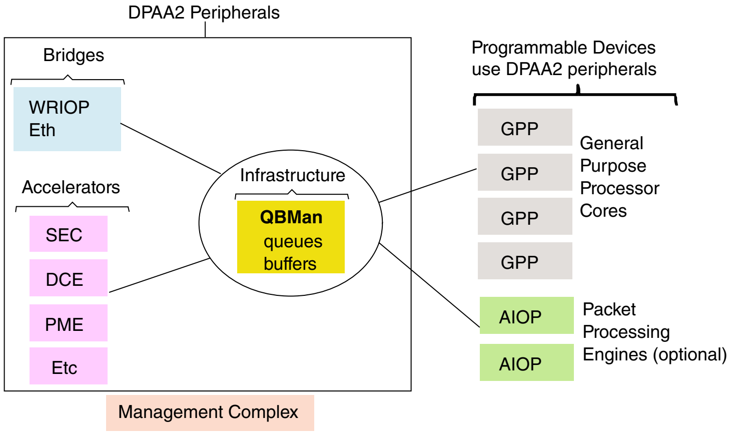


Figure 142. DPAA2 hardware

Example scenario based on DPAA2 objects shows how the example can be realized using the DPAA2 object abstractions of the DPAA2 hardware; this figure is much closer to the goal expressed in Example scenario and its components are described below:

- The host Linux is shown in more detail on the left. The network stack and two instances of the Ethernet drivers appear in the figure above the hardware boundary. Also, the figure shows the stacks and drivers for the two virtual machines.
- The DPAA2 objects appear below the hardware boundary
- The DPNI (Datapath Network Interface) objects correspond directly to the network interfaces in Example scenario. The DPSW (Datapath Switch) object corresponds to the L2 switch.
- The DPMAC (Datapath MAC) objects represent Ethernet MACs within WRIOP. These are hardware components that connect to PHY hardware, and provide Ethernet physical layer termination, i.e. Ethernet connections to the SoC.
- The DPIO (Datapath I/O) objects include QBMan software portals, and they allow GPP core software to read and write packets from the DPNIs. DPIOs are described in more detail later in this document.

See Object summary on page 684 for a summary of the DPAA2 objects.

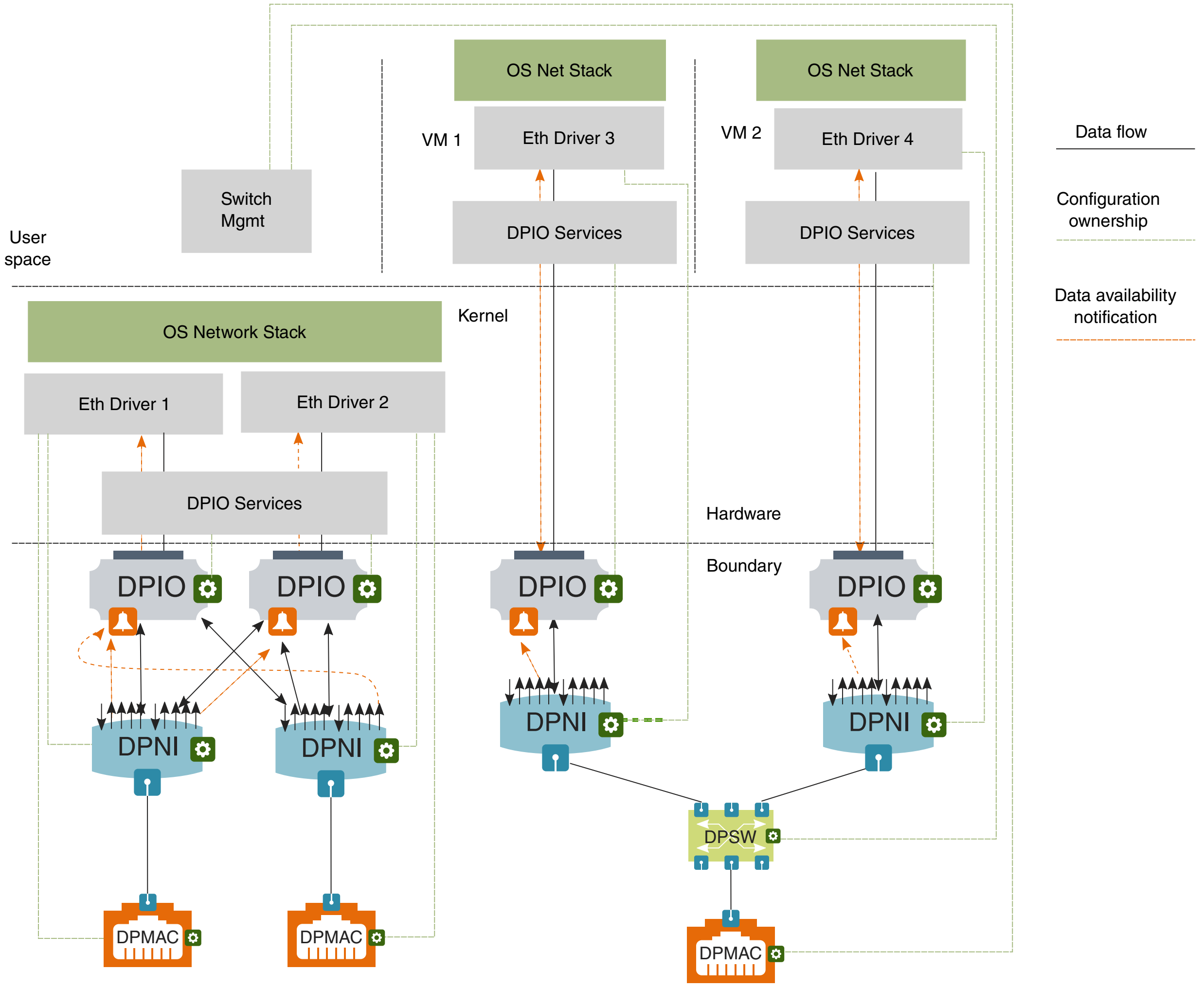


Figure 143. Example scenario based on DPAA2 objects

Objects are partitioned among software owners

Software management of DPAA2 objects is distributed. Software components that use a particular set of objects independently manage the objects in their set. The green boxes on the object icons in Example scenario based on DPAA2 objects represent management interfaces, and the green dashed lines show what software component owns the management of each object. For example, the DPSW is shown as managed by switch management software running on the general purpose processing cores.

Objects can be directly assigned

The virtual machines directly access and manage the objects their software uses, and they do this with minimal host kernel involvement; this enhances efficiency while preserving access isolation. In the figure, the virtual machines have directly assigned hardware-based network interfaces.

DPNI objects provide network interfaces

DPNI objects interact with drivers to allow software to send and receive network frames, usually Ethernet frames. DPNIs are central to DPAA2's concept of network interfaces, but they do not act alone. In general, network drivers manage several objects as part of managing network interfaces. DPNI ingress shows a high-level outline of DPNI ingress frame processing, and the following steps give insight into how objects work together.

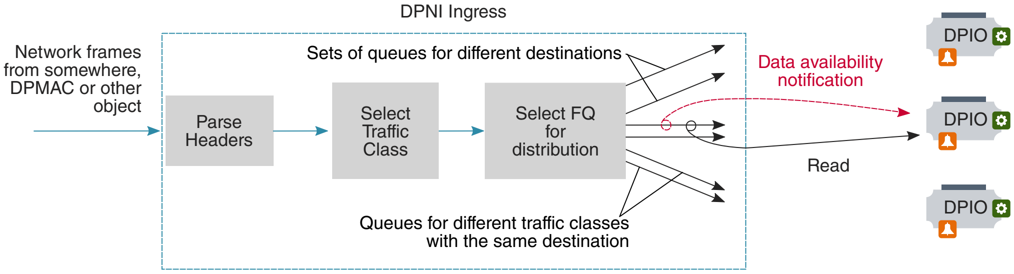

DPNI Ingress
Network frames from somewhere, DPMAC or other object
Parse Headers
Select Traffic Class
Select FQ for distribution
Sets of queues for different destinations
Queues for different traffic classes with the same destination
Data availability notification
Read
DPIO
DPIO
DPIO

Figure 144. DPNI ingress

1. A frame arrives at DPNI from another object, a MAC (DPMAC), a switch (DPSW) or other object.
2. DPNI parses the packet to locate the header from which lookup keys can be generated.
3. A lookup selects a traffic class (priority) for the frame; this priority causes a specific set of queues (implemented as QMan frame queues) to be selected.
4. DPNI must select a destination for the frame, using either another lookup or an RSS-style hashing operation; this lookup causes a specific queue within the previously selected set to be selected.
5. The frame is enqueued onto the queue, and the queue represents the destination indirectly. At this point, DPIO objects enter the process.
6. Every queue is configured to deliver data availability notifications to a specific DPIO, and these notifications tell the driver software using the DPIO that one or more frames are available to read from a specific queue.
7. Driver software responds by using a DPIO (actually any of its DPIOs) to read a burst of one or more frames from the queue.

Egress is simpler. The driver software uses a DPIO to enqueue a frame to a specific egress queue within DPNI; the queue is selected based on the desired traffic class.

Multiple DPIOs provide parallelism

It is common to assign queues in network interfaces to specific cores, and then to distribute the traffic between them using techniques like RSS or explicit flow steering. DPAA2 supports this process by using multiple DPIOs. See DPIO parallelism for an example involving a single network interface and two cores.

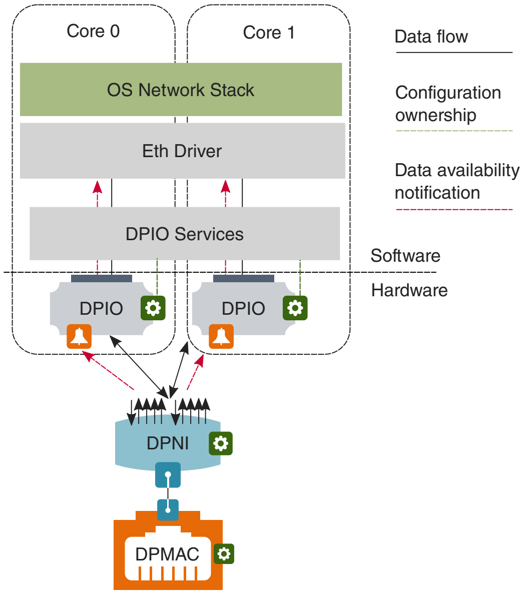


Figure 145. DPIO parallelism

The DPNI is configured so that each of its egress queues send its data availability notifications to one DPIO or another in a balanced way. A core receives an interrupt from its DPIO telling it to read a data availability notification, and then it then uses its DPIO to read a burst of one or more frames. In Linux terms, it starts a NAPI burst.

DPIO services

Notice in Example scenario based on DPAA2 objects that the host operating system on the left has two network interfaces. It has two DPIOs also, but either DPIO can be used for I/O to either of the interfaces. DPIOs are designed to be shared across network interfaces that belong to the same software component, such as the Linux kernel. For this reason, the Linux kernel contains a software layer called DPIO Services that facilitates driver instances performing I/O from a resource that might be shared across a network interface, and also might be shared across cores or software threads. Giving more DPIOs to the DPIO Services layer can increase performance, and using the same DPIO on a core for more than one network interface need not decrease performance because each core is physically able to do only one thing at a time.

8.3.1.4.1.1 Object summary

This section summaries the DPAA2 objects and shows a standard icon for each used in the illustrations that follow. See DPAA2 object summary and icons.

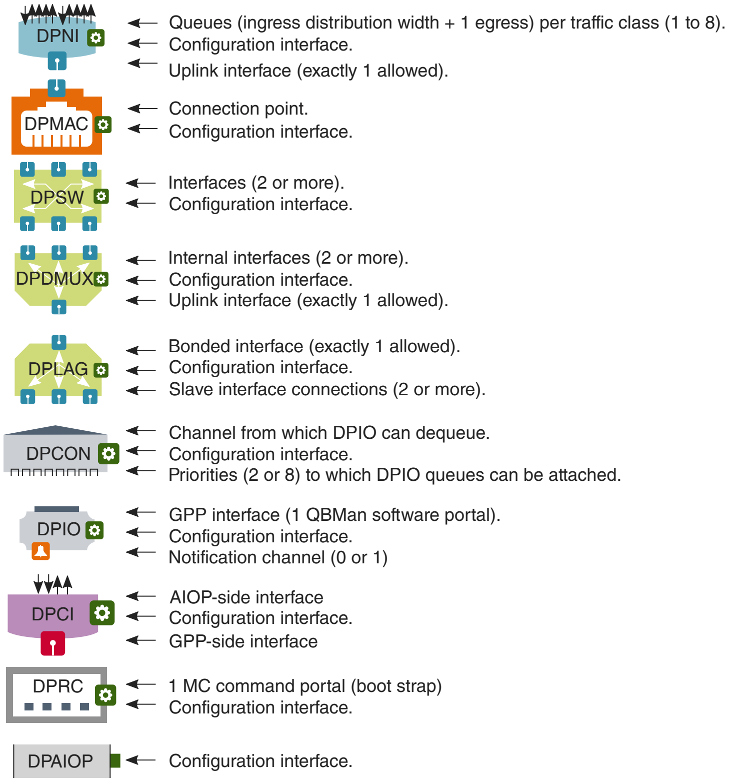

DPNI
Queues (ingress distribution width + 1 egress) per traffic class (1 to 8).
Configuration interface.
Uplink interface (exactly 1 allowed).
DPMAC
Connection point.
Configuration interface.
DPSW
Interfaces (2 or more).
Configuration interface.
DPDMUX
Internal interfaces (2 or more).
Configuration interface.
Uplink interface (exactly 1 allowed).
DPLAG
Bonded interface (exactly 1 allowed).
Configuration interface.
Slave interface connections (2 or more).
DPCON
Channel from which DPIO can dequeue.
Configuration interface.
Priorities (2 or 8) to which DPIO queues can be attached.
DPIO
GPP interface (1 QBMan software portal).
Configuration interface.
Notification channel (0 or 1)
DPCI
AIOP-side interface
Configuration interface.
GPP-side interface
DPRC
1 MC command portal (boot strap)
Configuration interface.
DPAIOP
Configuration interface.

Figure 146. DPAA2 object summary and icons

DPNI

A DPNI object is the key to network interfaces. On ingress, it receives frames from a DPMAC or another object such as a DPSW, parses headers, determines the frame's traffic class, and enqueues the frame onto a frame queue selected based on the traffic class and other header values. This supports both hash-based distribution of frames to multiple cores, and also direct flow steering of frames to specific cores.

DPNI can generate a per-queue data availability notification when a frame is enqueued. On egress, the DPNI dequeues frames from frame queues and transmits them to an external port using a DPMAC, or to another DPAA2 object such as a DPSW.

DPMAC

The DPMAC object represents an Ethernet MAC, a hardware device that connects to a PHY and allows physical transmission and reception of Ethernet frames.

DPSW

The DPSW object provides the functionality of a general layer 2 switch. It receives packets on one port and sends them on another. It can also send packets out on multiple ports for the purposes of broadcast, multi-cast, or mirroring.

DPDMUX

The DPDMUX is another type of switch. It differs from a DPSW in several ways. A DPDMUX may have only a single uplink port. Also, it can be programmed to direct packets based on header values above layer 2.

DPLAG

The DPLAG object provides link aggregation. It combines two or more uplinks into a single downlink.

DPCON

The DPCON object allows multiple DPNIs to be aggregated into a single device that appears to a GPP core or AIOP software as single interface that carries frames from multiple DPNIs; it combines two or more network interfaces into one. It provides a hardware-based scheduling off load because the hardware selects the order based on the priority in which frames from the multiple DPNIs are provided to software on GPP cores or AIOP.

DPCON is also useful for software that polls for input frames; it allows a single interface to be polled instead of multiple interfaces.

DPCON objects are also used by Linux Ethernet drivers for priority-based frame delivery.

DPIO

General purpose processing core software uses a DPIO object to perform hardware queuing operations, such as enqueue and dequeue, and hardware buffer management operations, such as acquire and release. It also allows data availability notifications to be received. DPIOs can generate interrupts. The DPIO object is unusual in that GPP core software is expected to directly access portions of the DPIO's hardware (QBMan software portals) for run time operations, in addition to supporting configuration operations from the management complex.

Note that AIOP software does not rely on DPIO objects; they are used only by software on the general purpose processing cores.

DPBP

The Datapath Buffer Pool object represents a QBMan buffer pool. It is used mainly as a resource by network drivers, but it is an active entity because it can send buffer pool depletion notifications to GPP core software.

DPCI

The Datapath Communication Interface provides general purpose processing core software with a transport mechanism typically for control and configuration command interfaces to AIOP applications. The AIOP service layer implements the AIOP-side of the transport, but the commands are application-specific. Note that AIOP is optional in DPAA2 and is not present on some SoCs.

DPRC

The DPRC object allows the Management Complex to track sets of objects in use by the same software component. The objects in the set are said to be in the same container. It also facilitates the assignment of sets of objects to specific software components, such as a virtual machine or a user space application using user space drivers. The software component can query containers in order to discover objects at run time, and this enables plug-and-play drivers that interface to objects.

Some objects include DMA-capable hardware. All objects in the same DPRC share a common ICID, and a common set of IO-MMU mappings. A number of key features of DPRCs include:

- **Direct access.** All the objects and resources in a container are private to the container, and software components get direct access to the registers (as abstracted by the Management Complex) of the hardware objects.
- **Dynamic discovery.** A software context that is given a DPRC can dynamically discover the objects and resources placed in the container using MC commands.
- **Hot plug/unplug.** Objects can be dynamically plugged and unplugged into DPRCs.
- **Security.** A software context can only see the objects in its DPRC, and cannot affect other containers or the proper operation of other software contexts. DMA transactions from MC objects are isolated using the system IOM-MU.

DPMCP

The DPMCP object represents a Management Complex command portal and is used by drivers to send commands to manage objects.

DPAIOP

DPAIOP is a configuration object that aids in loading programs onto the AIOP, running the AIOP, resetting the AIOP, and receiving status, error, and log information from AIOP programs. Note that the AIOP is optional in DPAA2 and is not present on some SoCs.

Objects for accelerators

There are also objects associated with accelerators such as SEC, PME, and DCE. These objects provide software with interfaces to the accelerator hardware. For this reason, the accelerator interface objects end in "I".

- DPSECI - SEC (security/cryptographic coprocessor) interface.
- DPDCEI - DCE (data compression engine) interface.
- DPDMAI - DMA engine interface.

Software uses queues associated with an object to send a buffer to an accelerator for processing and to receive the result.

New types of objects

NXP will create new types of objects over time to address new needs and use cases as they arise.

8.3.1.4.2 Management Complex: How DPAA2 objects are created and managed

This section outlines how the Management Complex creates and manages DPAA2 objects.

The best way to think of DPAA2 hardware, in particular WRIOP and QBMan, is that it provides many low-level resources ranging from Ethernet MACs to look up tables to frame queues and so on. Software's mission is to assemble the right set of these low-level resources, and configure them collectively to achieve a goal.

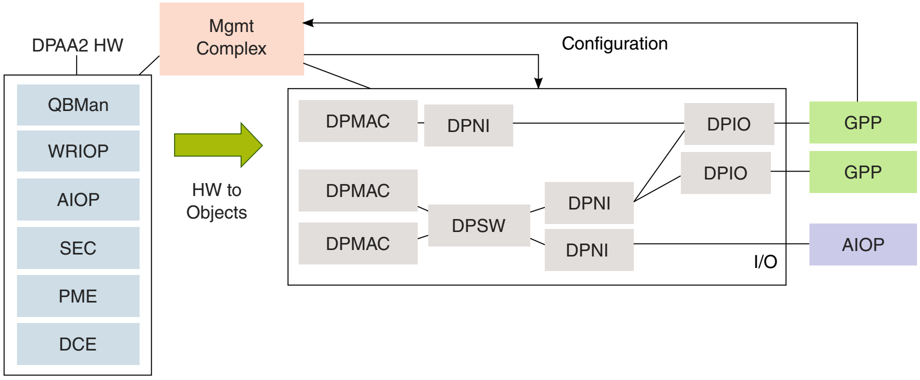


Figure 147. Management Complex creates objects from hardware sub-components

Think of the low-level resources as “atom resources” because they are always allocated as a unit. DPAA2 objects are then “composite resources,” or collections of atom resources that are then configured to achieve a common goal, like being an L2 switch as shown in Realizing an L2 switch.

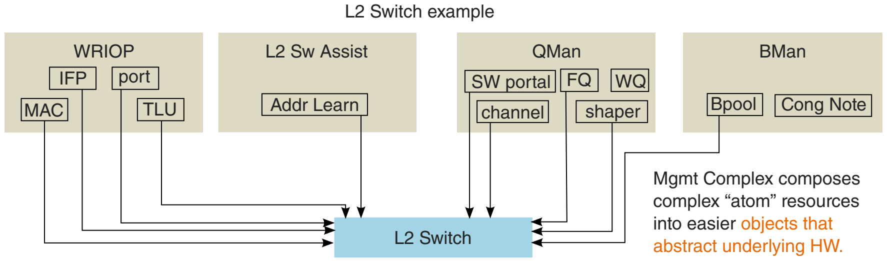


Figure 148. Realizing an L2 switch

The creation method for a DPAA2 object involves allocating the necessary atom resources and configuring them enough to place the object in an initial idle state. Object methods and other interfaces then allow it to be further configured and used. For example, forming an L2 switch from DPAA2 atom resources is quite complex. The NXP firmware running on the Management Complex implements the methods necessary, and hides this complexity from GPP (and AIOP) developers.

Continuing the example, an L2 switch object can also be shutdown and disassembled by its methods. Its atom-resources are then placed back into the pools of atom resources that the Management Complex firmware manages.

Hardware directly visible to software

Clearly, DPAA2 provides abstractions. The objects are best thought of as being hardware, and most actually are collections or encapsulations of hardware resources that are allocated and configured to achieve a higher-level and more abstract purpose than would be clear from a direct view of the hardware resources. An example of an abstract purpose is “be an L2 switch” (DPSW).

It can be helpful to focus on exactly what is visible to driver-level software running on the general purpose cores and AIOP, especially since what is visible is a mixture of direct access to hardware and indirect access to hardware via abstractions. This discussion will be biased towards the view of objects from drivers running on general purpose processing cores (such as in U-Boot and Linux).

Also the discussion will avoid details of individual objects since this is an overview with the purpose of clarifying objects in general.

DPAA2 visibility boundary describes in one diagram what is directly visible to the driver layer software.

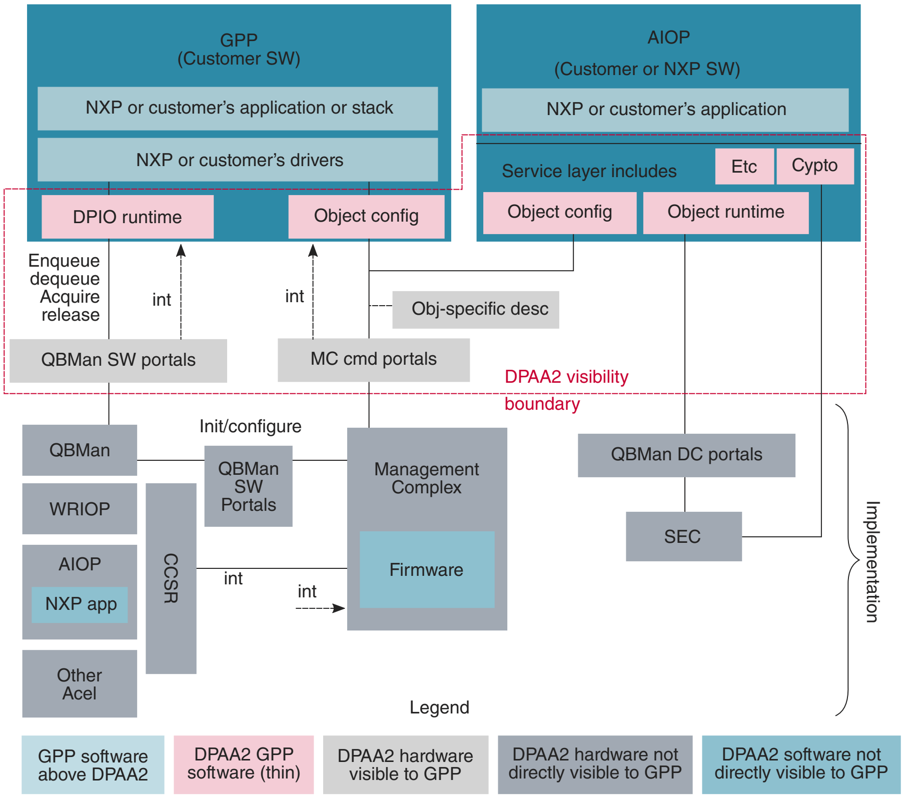


Figure 149. DPAA2 visibility boundary (AIOP not present on all DPAA2 SoCs)

There is quite a lot in the figure above, so it is best to break it down. What driver level software can see and do is dictated by its function.

This begins with the Management Complex (MC) itself. The discussion below will focus on the services that the MC provides to other software in the system. There will be no discussion of MC firmware's internal design.

See Management complex visibility in DPAA2. The first step is that general purpose processing core software (usually a boot loader) must load the opaque firmware image onto the Management Complex and then start it running. This involves direct access to portions of the Management Complex hardware: registers defining the location of the Management Complex's portion of DDR, image location, address translation, and run state control.

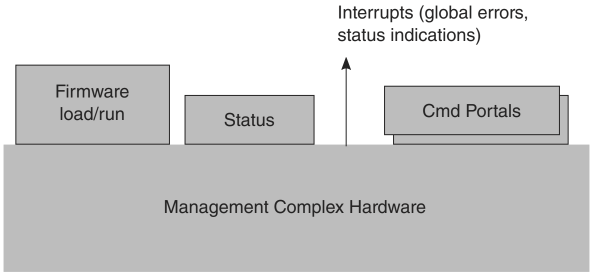


Figure 150. Management Complex visibility in DPAA2

The driver software also requires visibility to global status, particularly to status for global errors. Changes in the state of this status can be signaled by interrupts to the general purpose processing cores so the Management Complex can produce these interrupts.

Finally, the Management Complex exists to serve its masters, the general purpose processor core (and AIOP when present) software that "owns" objects, i.e. has been allocated access rights to them via container ownership and hierarchy. The service is provided by responding to commands so driver software needs a way to deliver commands to the Management Complex. In addition, this process must be secure in that the Management Complex must know, in a way that cannot be spoofed, an ID of the software sending the command. This is to allow the Management Complex to enforce object access rights.

Driver software delivers commands to the Management Complex via hardware called Management Complex command portals. SoC hardware provides significant numbers (at least 10s) of these portals because:

- They can be directly assigned to multiple different drivers, all of which independently use the Management Complex's services. If they each have their own command portal, they do not have to coordinate with each other.
- Each independent driver instance has its own ID (ICID) that is securely associated with the command portal to prevent spoofing. This prevents a driver from being able to access for configuration an object that it does not "own".

To send a command to the Management Complex, driver software creates a descriptor and enqueues a pointer to it to the command portal.

Next consider objects. See DPAA2 objects.

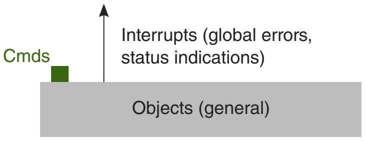


Figure 151. DPAA2 objects

Objects are created either via the DPL file or driver software sending a command to the Management Complex instructing it to create an object (as in restool). The Management Complex supplies a globally unique ID for the new object.

Object command interfaces are abstractions. There is no hardware that directly represents object command portals. Objects are usually hardware, but in most cases that hardware does not directly expose a hardware-level programming model to driver software. Instead, driver software configures objects via an indirect mechanism; it sends a command to the Management Complex. The command is a descriptor that includes the ID of the object as well as the definition of the operation to be performed.

The Management Complex automatically gets the ID of the requestor when it reads the command. The command portal securely adds it. The Management Complex then checks that the requestor is authorized to configure the object and, if so, performs the configuration on behalf of the requestor.

So, object configuration is a visible part of DPAA2, but the configuration of the individual hardware subcomponents that make up an object is not.

The fundamental programming model for object configuration is the commands that can be sent to the Management Complex to configure the object. Each object type has a different purpose so each object type's configuration programming model is defined by the descriptor set that describes the commands to configure the particular type of object.

NXP also provides C callable APIs that basically allocate and populate descriptors and pass them as commands to the Management Complex. The APIs bear a close relationship to the more fundamental descriptors.

Many object types have nothing but a configuration space, but this is not always true. Some objects also provide I/O interfaces. The DPIO object is a prime example. See DPIO object and I/O interfaces.

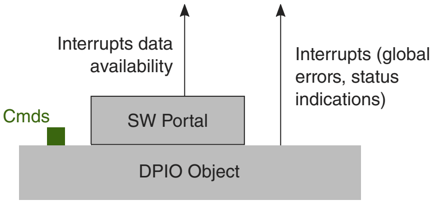

Interrupts data availability
Interrupts (global errors, status indications)
Cmds
SW Portal
DPIO Object

Figure 152. DPIO object and I/O interfaces

As has been stated before, DPAA2 objects are usually opaque bundles of hardware sub-resources allocated and configured to achieve a more abstract purpose. A DPIO object includes a hardware sub-resource called a QBMan software portal but this hardware is not opaque to the driver software running on the general purpose processing cores. The reason is performance. Software portals are the hardware mechanism for actually doing I/O with DPAA2 peripherals so driver software must directly access them. There are also data availability interrupts associated with DPIOs. These indicate availability of data to read using the software portals.

NOTE

Software portals actually support more than I/O (enqueue onto queues and dequeue from them). They also support commands. The simplest examples are buffer acquires and releases. Without going into full detail, software portals actually support commands that require privilege (example: initialize a frame queue) and commands that do not (example: acquire a buffer). Driver software on general purpose processing cores (and AIOP) uses only the unprivileged commands. The privileged commands are not part of the visible architecture. They are used only by Management Complex firmware.

In summary, the visible architecture includes both hardware and abstractions as follows:

- Management Complex hardware associated with loading and running images
- Management Complex hardware associated with accessing global status
- Management Complex global interrupt
- Management Complex hardware command portals
- Objects themselves (abstraction):
    - — Object configuration interface and command set as defined by descriptors (abstraction)
    - — Object error interrupts
    - — Some objects (like DPIO) also have additional interfaces that are hardware directly accessed by driver software. DPIO’s QBMan software portals are an example. They can produce interrupts.

8.3.1.4.2.1 Object creation, the datapath layout file, and restool

DPAA2 objects can be created in multiple ways. First, they can be specified in a Datapath Layout (DPL) file that the Management Complex reads and applies before Linux boots. This file contains the specific list of objects that are to be automatically created as the system initializes.

DPAA2 objects also can be created and destroyed dynamically by sending commands to the Management Complex through its command portals via a kernel driver. For Linux, a user space command line tool called “restool” uses this interface to allow interactive and dynamic creation of objects. It also allows destruction and some additional configurations to be done.

Restool also shows information about objects and what they are connected to.

8.3.1.4.2.2 DPRC objects, plug and play, and the fsl-mc Linux “bus”

As mentioned previously, it is common for a GPP software component to manage multiple objects. The DPIO parallelism diagram shows a simple example of the Linux kernel managing a set of objects to provide a pair of network interfaces. The DPRC (Datapath Resource Container) is a special object that serves to organize other objects, and also the hardware sub-components from which objects can be dynamically created; the hardware sub-components include frame queues, channels, buffer pools, etc. Containers can be created and filled with objects and resources and then passed to the software component, such as a virtual machine, that will use them.

The software that was assigned a DPRC can enumerate the objects inside it; this is a form of dynamic hardware discovery that relates to plug-and-play. For example, an operating system can scan a DPRC and associate all DPNI objects found within with an Ethernet driver that will use them to form network interfaces. The Ethernet driver then uses a dynamic allocator within the kernel to aquire other objects such as DPBPs that it needs to operate.

The device discovery analogy is strong enough that Linux exposes DPRCs assigned to it as a bus in sysfs-- much like physical buses like PCI. The same sysfs mechanism that allow a physical PCI device to be assigned (bound) to virtual machines are also used to assign containers to virtual machines. Objects can even be dynamically added and removed from DPRCs. This is analogous to hot plug and unplug on a bus.

Many DPAA2 objects are DMA-capable so that they can autonomously read and write memory. SoCs like the LS2088A contain an IO-MMU, so objects must express an identifier (that they cannot control) when they perform DMA operations. This identifier is called an ICID in DPAA2, and it serves as a key for the IO-MMU to associate I/O virtual addresses with I/O physical addresses. In DPAA2, ICIDs are attributes of DPRCs, and all objects in a DPRC express the same ICID value.

A GPP software context (a virtual machine or application) will typically be assigned a single DPRC that contains all the fsl-mc resources that the software context can access or use. As mentioned elsewhere, there are two general types of resources that can be in a container:

- **Resources:** Resources are primitive resources that can't be further decomposed, and are uninitialized and unpurposed. Some examples are MC portals, QBman portals, frame queues, buffer pools, etc. Generally primitives are “fungible,” in that there is nothing distinctive among the same kind of primitives. However, some primitives may be non-fungible, such as an external port or MAC.
- **Objects:** Objects are created and configured with a purpose, typically constructed of multiple resources. Some examples of objects are network interfaces, an L2 switch, or a crypto instance. A DPRC is itself an fsl-mc object.

NOTE

See documentation of the Linux restool facility for more information related to this topic.

Management Complex (MC) initialization and boot

The MC is normally enabled and initialized by system boot firmware such as U-Boot. The boot firmware is responsible for reserving a region of memory (DDR) for the fsl-mc, and then loading the MC firmware into memory, loading a datapath layout file (see below for DPL overview info), and writing a bit to enable/start the MC. See Management Complex initialization and boot.

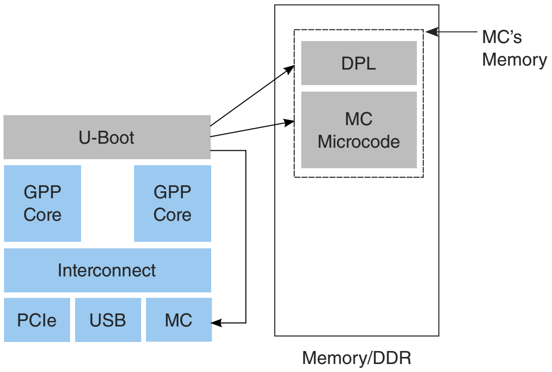


Figure 153. Management Complex initialization and boot

Management Complex datapath layout file (DPL)

As mentioned above, a datapath layout file (DPL) must be supplied to the Management Complex when it is booted. The DPL contains the definitions of initial objects and containers/DPRCs to create.

A DPL is defined in a text file in device tree syntax (DTS) format and then compiled into a standardized DTB binary format (used by ePAPR compliant device trees).

See the *DPAA2 User Manual* for more information and examples on the datapath layout file.

Boot loader use of the MC

In typical usage, the boot loader loads the MC firmware image and starts the MC running. At this time, it supplies a data path control (DPC) file that supplies the MC image with basic configuration information that allows it to operate.

The boot loader can now use the services of the MC in order to access network devices. It is a good approach to have the boot loader dynamically create the objects it needs and destroy them (releasing resources) before starting the operating system. This way, the operating system is not forced to operate with the constraints of objects and DPRCs established by the boot loader. The OS can see a "green field".

Optionally, the boot loader can apply the data path layout (DPL) file mentioned above just before starting this OS. This approach allows the DPL to be written only to serve the operating system's needs and not the boot loader's, which tend to be much simpler.

DPRCs are hierarchical

The MC manages DPRCs in a hierarchical relationship. There is a single root DPRC at the root of the hierarchy. That DPRC can have child DPRCs, children can have grandchildren, and so on. The root DPRC belongs to the root software context of the system, usually an OS or hypervisor and it should never be unbound from the corresponding driver. The root DPRC can further allocate its resources to its child DPRCs and assign them to other entities such as user space applications or virtual machines.

In this example there are 3 DPRCs/containers managed by the Management Complex: a root container “root” with 2 children “foo” and “bar”. The DPRCs all contain 3 objects, a DPNI, DPBP, and DPIO. There are 3 software contexts: the host Linux, a user space application, and Linux in a KVM virtual machine. Each software context is assigned a DPRC that it can use and manage; see DPRC hierarchy for a figure that illustrates this example.

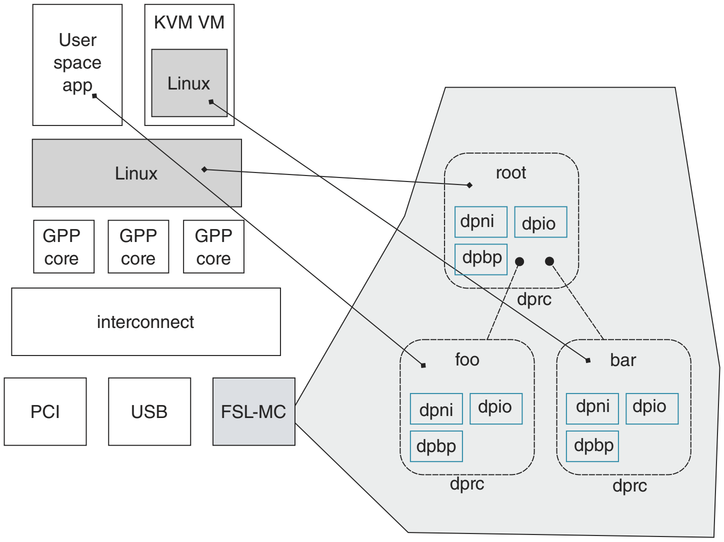


Figure 154. DPRC hierarchy

The container hierarchy allows the parent to manage the resources of the children. If the OS in the KVM VM crashes, the parent (Linux) can reset and clean up the VM's DPRC. If the user space application terminates, the parent (Linux) has the option of destroying the container.

8.3.1.4.3 Objects and topology

As mentioned elsewhere, objects have a topological relationship with each other. See Object topology example for an example.

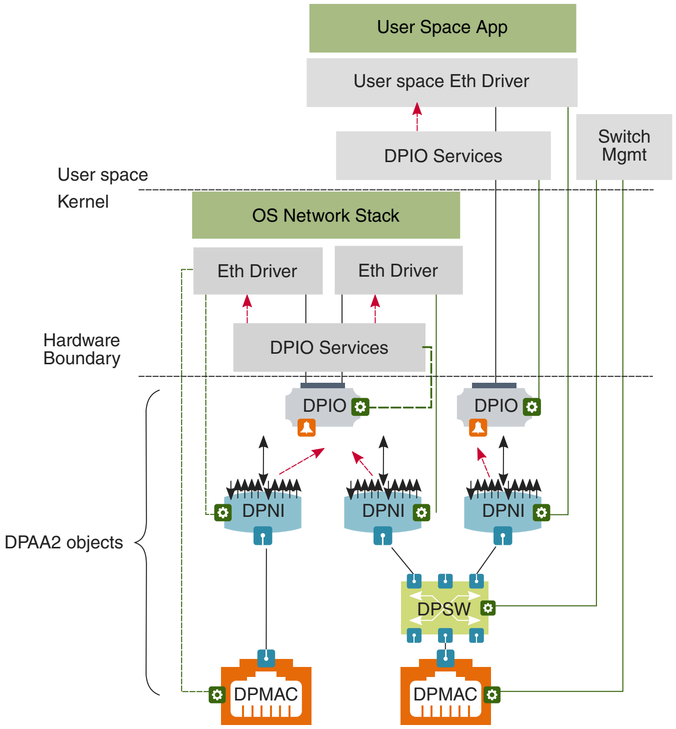


Figure 155. Object topology example

- There are three network interfaces managed by the DPAA2 Linux Ethernet driver. Each of the network interfaces uses a DPNI object.
- All of the Ethernet drivers happen to have a distribution width of one (an example), so they cannot load balance to multiple cores or threads; this was done to simplify the diagram and discussion. If a network interface has a distribution width greater than one, then many times it is connected to more than one DPIO but this is not required.
- Two of the network interfaces are connected to a switch; two DPNIs are connected to a DPSW. This allows both network interfaces to communicate outside of the SoC using the DPMAC that is also connected to the DPSW, and they can also communicate with each other using the DPSW.
- One of the network interfaces is directly assigned to a user space process, and has a user space Ethernet driver. This network interface could also be directly assigned to a KVM virtual machine under Linux.
- Two of the DPNIs have Linux network stack drivers; they interface to the Linux network stack. One of them has its own DPMAC, and a traditional type of controller represented by its DPNI being directly connected to a DPMAC.
- The two DPNIs connected to the Linux network stack share a single DPIO; this is possible when they can cooperatively use a layer of GPP software that provides DPIO services. The hardware that makes up a DPIO is a QBMan software portal and, optionally, a QMan channel for data availability notifications. QBMan software portals are a relatively scarce hardware resource, so they are designed to be sharable, in particular for NAPI-compliant Linux Ethernet drivers.
- It is a key assumption of DPAA2 that objects are managed (or “owned”) by a single software entity. Independent software entities can independently manage the objects they own, and this allows software to be decoupled from other entities.
- The management relationship between objects and software entities is not defined or imposed by DPAA2; DPAA2 defines the objects and what they do, and not what software uses them. Customer GPP core software is allowed to determine the management relationship; a single monolithic software entity that manages all of the objects can be created.
- The Linux DPAA2 Ethernet driver design defines the set of objects needed to provide a network interface. The green lines show the management relationships for Linux network interfaces and switches. Note that switches are managed independently from the network interfaces that connect to it.

The *DPAA2 User Manual* provides a complete description of the rules that govern object topology.

8.3.1.4.4 AIOP in DPAA2

This section describes AIOP in the DPAA2 architecture and how it uses DPAA2 objects. The figure below shows an example in which the Linux kernel network stack has a single Ethernet interface, a user space application has a single directly assigned Ethernet interface, and AIOP has two Ethernet interfaces.

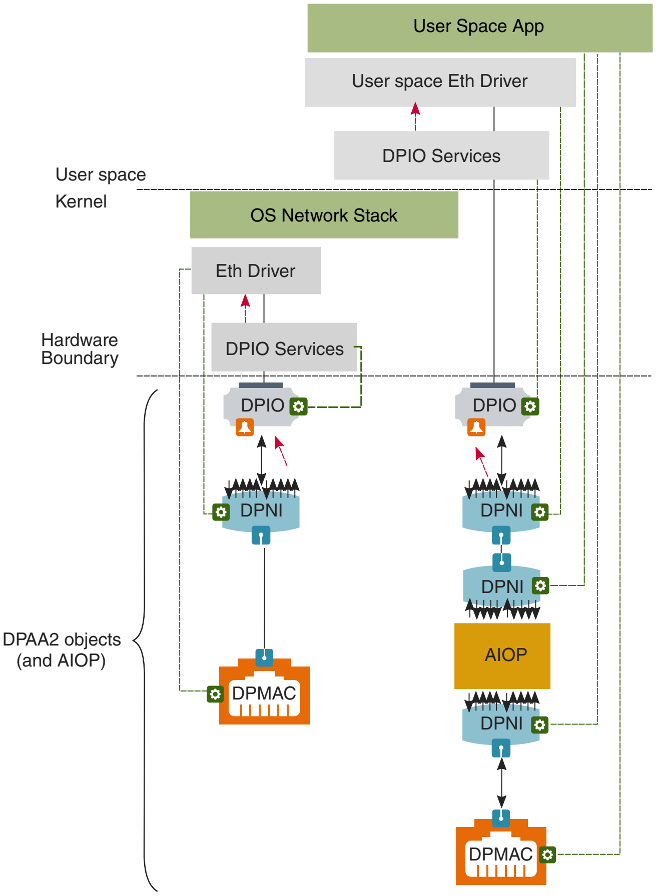


Figure 156. AIOP in DPAA2

AIOP runs software and uses objects in a manner that is similar to general purpose processing cores. In this example configuration, the AIOP's software has access to two Ethernet interfaces. One is connected to a MAC for an external connection to outside the SoC. The other is connected point-to-point to the general purpose processing core user space application's Ethernet interface.

So, there are two Ethernet interfaces involved on the path between the AIOP and the user space application. This is logical because there is software running in both places. Thus, each software component should see and control its own Ethernet Interface. Both software components do Ethernet I/O without being coupled to what their Ethernet interface is connected to.

One difference is that AIOP software is focused on packet processing. It does not actively manage or configure its own Ethernet interfaces. Usually, a control application on the general purpose processing cores takes that role. Also, the AIOP does not use DPIO objects.

Note that AIOP itself is not a DPAA2 object. It is an active entitiy that uses other DPAA2 objects. However, there is a DPAIOP object that general purpose processing core software can use to manage the AIOP, e.g. start it, stop it, load images onto it, get error status from it, etc.

In addition, there are DPAA2 objects (not shown) that facilitate passing commands (rather than packets) between general purpose processing core software and AIOP software.

AIOP Service Layer

NXP provides an AIOP software library called the "AIOP Service Layer". This library's main purpose is to provide very lightweight drivers for hardware components that AIOP software must access. These include components such as Ethernet intefaces, the QBMan buffer manager, timers, table lookup units, etc.

8.3.2 DPAA2 Quick Start Guide

8.3.2.1 Data Path Resource Containers

Many sections refer to Data Path Resource Containers (DPRC), so a brief introduction to the concept may be helpful. DPRCs are part of the DPAA2 object architecture that is described in the DPAA2 Software Overview.

DPRCs are communicated to software entities as a part of their start up process; this is true for software entities such as:

- The host Linux kernel (that may provide KVM services to virtual machines)
- Linux kernel instances that run in virtual machines
- DPDK applications
- AIOP applications

DPRCs contain DPAA2 objects that are used by the software entity that owns the DPRC. For example, DPNI objects are used as network interfaces.

As an example, see RDB DPL on page 698RDB DPL on page 698. The DPRC called “dprc@1” is supplied to the host Linux kernel. It contains one DPNI object. The DPNI object binds to the DPAA2 Linux kernel Ethernet driver, and causes two standard Linux Ethernet interfaces to exist and be visible using “ifconfig”. See later sections in this document for additional details and explanations on the use of objects by various types of software entities.

As mentioned previously, DPRCs must be created and populated with the initial set of DPAA2 objects prior to the startup of the software entity that will use the DPRC.

8.3.2.1.1 Creating DPRCs

There are two ways to create and populate DPRCs:

1. Statically: by means of a control file called a datapath layout (DPL) file. Key Release Files: RCW, DPC and DPL on page 697 describes the DPL files that are supplied as examples with the Linux SDK.
2. Dynamically: by means of the Linux command line utility called restool. See DPRCs and restool section, and also the document titled *Standard Linux Documentation.*

A software entity behaves exactly the same way on startup regardless of whether its DPRC was created statically using a DPL or dynamically using restool. The DPL method is convenient for situations when the desired DPRCs are known in advance. DPRCs defined within the DPL are created and populated automatically with no need for a subsequent use of restool.

8.3.2.1.2 DPRCs and Hot Plug

A DPRC must be supplied to a software entity when the software entity is started; this implies prior creation of the DPRC. However, it is also possible to dynamically alter the contents of a DPRC *after* the software entity that is using it is already running; this is a form of hot plug.

For example, restool can be used to dynamically create a DPNI object and then assign it to a DPRC. If that DPRC is being used by a Linux kernel instance, this will cause that kernel to dynamically detect a new network interface and bind it to the Linux kernel Ethernet driver; the “ifconfig” command will now show a newly created network interface.

It is also possible to dynamically destroy or unassign objects within an in-use DPRC; this is a form of hot unplug. Hot plug and unplug are advanced topics, and not covered further here. The key take-away is that dynamically creating and populating a DPRC *before* supplying it to software entity when it is started is a very different use case than hot plug/unplug. The latter use case involves changing the contents of a DPRC *while it is in use* by a software entity.

8.3.2.2 Key Release Files: RCW, DPC and DPL

This section describes the key binaries that are available on the RDB. These include the reset configuration word (RCW), the data path configuration (DPC) and the data path layout (DPL) files.

8.3.2.2.1 RCW

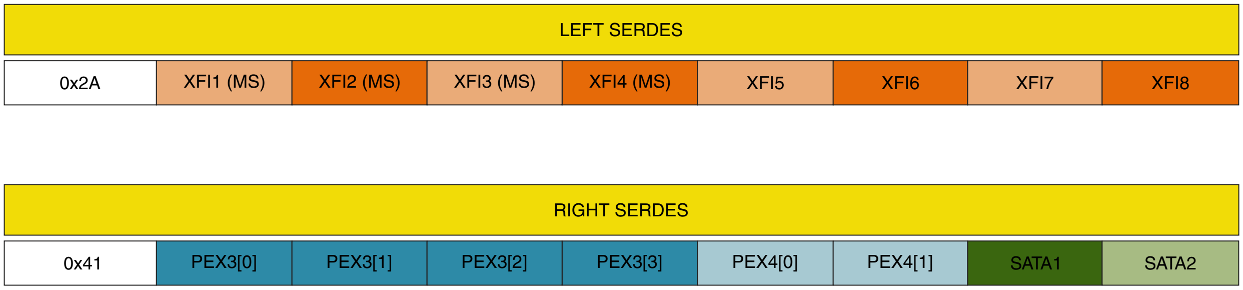
The reset configuration word (RCW) resides in non-volatile memories (e.g. NOR, QSPI, SDHC). It gives flexibility to accommodate a large number of configuration parameters to support a high degree of configurability of the SoC. Configuration parameters generally include:

- Frequencies of various blocks including cores/DDR/interconnect.
- IP pin-muxing configurations
- Other SoC configurations

The RCW's provided with the release enable the following features:

- Boot location as NOR flash
- Enables 4 UART without flow control
- Enables I2C1, I2C2, I2C3, I2C4, SDHC, IFC, PCIe, SATA

The figure below shows the SERDES configuration supported for DPAA2 platforms. Note that each platform supports upto 5 ports out of the 8 available on the RDB at a time.



LEFT SERDES								
0x2A	XFI1 (MS)	XFI2 (MS)	XFI3 (MS)	XFI4 (MS)	XFI5	XFI6	XFI7	XFI8

RIGHT SERDES								
0x41	PEX3[0]	PEX3[1]	PEX3[2]	PEX3[3]	PEX4[0]	PEX4[1]	SATA1	SATA2

Figure 157. SERDES

8.3.2.2.2 Data path configuration file (DPC)

The data path configuration (DPC) contains board-specific and system-specific information that overrides the default DPAA hardware configuration.

The release provides one data path configuration (DPC) file per board type. This file specifies the following information:

- default logging mode for the Management Complaex (MC)
- default board MACs
- default number of DPAA channels with 2 and 8 work-queues

The DPC is based on a text source file similar to a device tree source file (DTS) and should be compiled using the DTC utility to form a binary structure (blob, similar to DTB).

8.3.2.2.3 Data path layout file (DPL)

The data path layout file (DPL) defines the containers created during MC initialization. In order to compile the DPL, the device tree compiler (DTC) tool needs to be installed on the host system.

As described in Creating DPRCs on page 696, the example DPL source code is provided in the *dpl-examples* package.

The DPL file specifies the basic resources needed for a simple usecase; other resources are created and managed dynamically using restool capabilities. For each of the use cases included in this document, there is a diagram that depicts the objects that are necessary for that usecase.

The DPL is based on a text source file (similar to a device tree source file (DTS)) and compiled with the DTC utility to form a binary structure (blob, similar to DTB). The DPL file should be compiled to a binary blob using standard DTC tool.

Using a static DPL is not a requirement since restool can be used to dynamically create/manage objects and resources.

8.3.2.2.3.1 RDB DPL

The source for the RDB DPL is in the *dpl-examples* package:

- dpl-examples/LS2088a/RDB/dpl-eth.0x2A_0x41.dts

The figure below shows a graphical view of the container configuration:

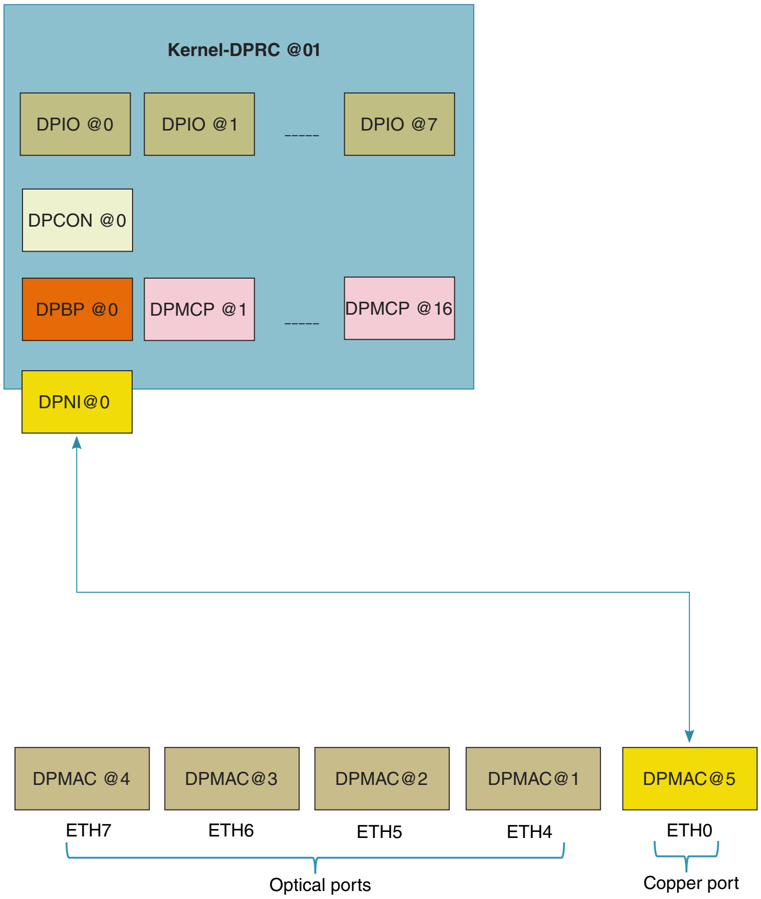


Figure 158. RDB DPL container configuration

8.3.2.2.3.2 RDB DPL

The source for the RDB DPL is in the *dpl-examples* package:

- dpl-examples/ls1088a/RDB/dpl-eth.0x1D_0x0D.dts

The figure below shows a graphical view of the container configuration:

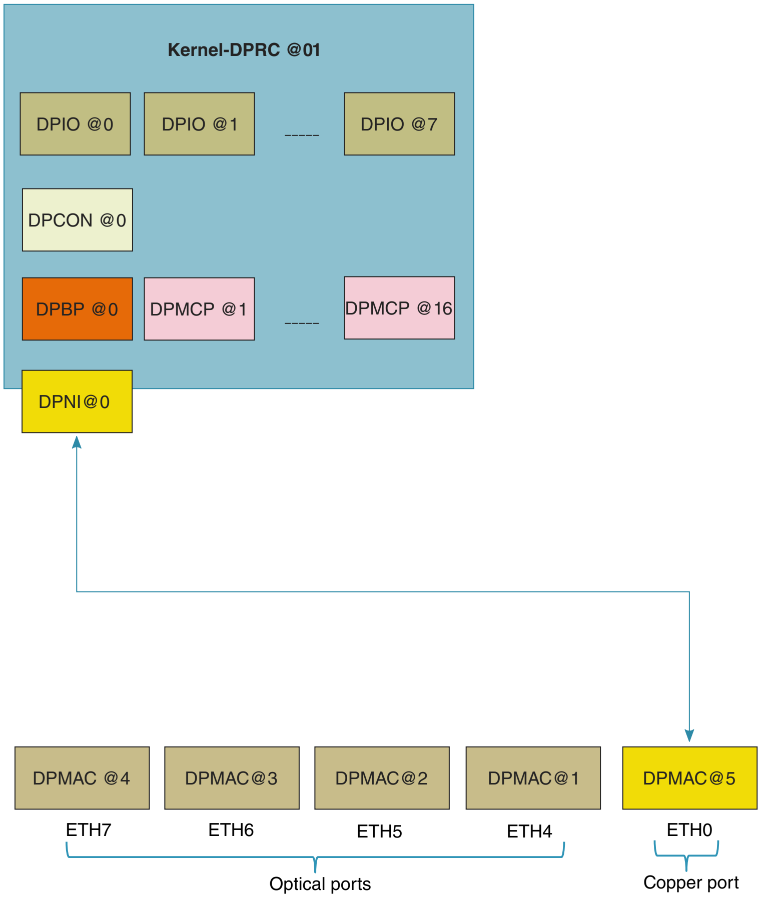


Figure 159. RDB DPL container configuration

8.3.2.2.3.3 DPRCs and restool

The release provides a Linux command-line tool called restool that can be used for examining the resource containers used for managing DPAA2 objects and resources. See the DPAA2 Overview for an overview of DPAA2 and the data path resource containers (DPRCs). Also see the Stardard Linux Documentation for details about functionality and use of restool.

Given below is an example of using restool:

List dprc:

```
$ restool dprc list
dprc.1
```

List all objects in container dprc.1:

```
$ restool dprc show dprc.1
dprc.1 contains 33 objects:
object          label           plugged-state
dpio.7                          plugged
dpio.6                          plugged
dpio.5                          plugged
dpio.4                          plugged
dpio.3                          plugged
dpio.2                          plugged
dpio.1                          plugged
dpio.0                          plugged
```

```
dpni.0                          plugged
dpbp.0                          plugged
dpmac.5                         plugged
dpmac.4                         plugged
dpmac.3                         plugged
dpmac.2                         plugged
dpmac.1                         plugged
dpcon.0                         plugged
dpmcp.0                         plugged
dpmcp.16                        plugged
dpmcp.15                        plugged
dpmcp.14                        plugged
dpmcp.13                        plugged
dpmcp.12                        plugged
dpmcp.11                        plugged
dpmcp.10                        plugged
dpmcp.9                         plugged
dpmcp.8                         plugged
dpmcp.7                         plugged
dpmcp.6                         plugged
dpmcp.5                         plugged
dpmcp.4                         plugged
dpmcp.3                         plugged
dpmcp.2                         plugged
dpmcp.1                         plugged
```

Get information about dpni.0:

```
dpni version: 7.1
dpni id: 0
plugged state: unplugged
endpoint state: -1
endpoint: No object associated
link status: 0 - down
mac address: 00:00:00:00:00:00
dpni_attr.options value is: 0
num_queues: 1
num_tcs: 1
mac_entries: 16
vlan_entries: 0
qos_entries: 0
fs_entries: 64
qos_key_size: 0
fs_key_size: 56
ingress_all_frames: 0
ingress_all_bytes: 0
ingress_multicast_frames: 0
ingress_multicast_bytes: 0
ingress_broadcast_frames: 0
ingress_broadcast_bytes: 0
egress_all_frames: 0
egress_all_bytes: 0
egress_multicast_frames: 0
egress_multicast_bytes: 0
egress_broadcast_frames: 0
egress_broadcast_bytes: 0
ingress_filtered_frames: 0
ingress_discarded_frames: 0
ingress_nobuffer_discards: 0
egress_discarded_frames: 0
egress_confirmed_frames: 0
```

8.3.2.3 Linux Ethernet

This chapter provides guidelines on exercising creation, functionality and statistics of Linux DPAA2 Ethernet interfaces.

8.3.2.3.1 Features overview

The following is an overview of the functionality of the Linux DPAA2 Ethernet driver that will be described in this chapter:

- Primary MAC address change
- Scatter-gather support
- Checksum offload
- MAC filtering
- Large frame support
- GRO – generic receive offload
- Egress traffic shaping
- Rx hashing
- Rx flow steering (LS2088A only)
- Flow control pause frames
- Interface statistics

8.3.2.3.2 Compiling and selecting Kconfig options

The DPAA2 Ethernet driver is by default selected by the kernel configuration shipped with the SDK, with a set of sensible compile-time defaults. The driver path in the kernel config file is: `" Device Drivers -> Staging drivers -> Freescale DPAA2 devices -> Freescale DPAA2 Ethernet" (CONFIG_FSL_DPAA2_ETH).`

The following Kconfig selects are also available, but not checked by default:

- "Enable Rx error queue" `(CONFIG_FSL_DPAA2_ETH_USE_ERR_QUEUE)` - This configures a separate queue for presenting Rx Error frames to the Ethernet driver running on the GPP. By default this option is disabled and Rx Error frames are dropped in hardware and counted for statistics inspection. Rx Error processing increases the GPP load and so it is only necessary in debugging situations when inspection of actual frames is required.
- "Enable debugfs support" `(CONFIG_FSL_DPAA2_ETH_DEBUGFS)`. This option depends on "Debug Filesystem" `(CONFIG_DEBUG_FS)` being enabled. If selected with its dependencies, this option enables a number of Ethernet driver statistics under DebugFS nodes `(/sys/kernel/debug/dpaa2-eth/*)` that are not normally visible to the user.
- "Data Center Bridging (DCB) Support" `(CONFIG_FSL_DPAA2_ETH_DCB)`. This option depends on "Data Center Bridging support" `(CONFIG_DCB)`. It is required when configuring Priority-based Flow Control (PFC) scenarios.
- "DPAA2 Ethernet CEETM QoS" `(CONFIG_FSL_DPAA2_ETH_CEETM)`. This option enables the use of a custom CEETM qdics to offload egress Qos support.

8.3.2.3.3 Creating a DPAA2 network interface

This section documents the resource utilization and the necessary steps for creating a DPAA2 network interface (DPNI) in Linux.

A DPNI can be created either statically through the DPL file or dynamically using the ‘restool’ utility.

8.3.2.3.3.1 DPAA2 objects dependencies

This section documents the steps to create a DPNI and related objects in order to have a fully functional network interface. It describes the dependencies a DPNI has on other DPAA2 objects.

This is of interest to anyone who changes a static DPL file or uses *restool* commands to dynamically create: Using restool to create a DPNI on page 705 (For LS1088A see Using restool to create a DPNI).

The DPAA2 object definition allows for flexibile software architectures. The Linux drivers, in particular the Ethernet driver, are additionally bound by requirements from the kernel architecture. This enforces a certain usage model of the DPAA2 objects that the DPNI interacts with; in particular, it affects the number of various other DPAA2 objects that a DPNI need s.

Generally, the Linux Ethernet driver requires private DPAA2 resources (e.g. Frame Queues) and objects (e.g. DPCON objects), distinct from other DPNIs. There are exceptions, such as the DPIO or DPMAC, which are not owned by the DPNI. To create a DPNI, either statically in DPL or dynamically using ‘restool’, the following types of objects may need to be instantiated in the current container (i.e., made available if they are not already):

- DPBP
- DPMCP
- DPCON
- DPIO
- DPMAC

The significance and number of these objects per DPNI are detailed in the following table:

Object	**Private to DPNI?**	**Cardinality**	**Comments**
DPBP	Yes	1 per DPNI	Each network interface (NI) has private buffer pools, not shared with other NIs.
DPMCP	Yes	1 per DPNI 1 per DPMAC	MC command portals (MCPs) are used to send commands to, and receive responses from, the MC firmware. One such example is configuring DPNI functionality like hashing or checksumming, but DPNI statistics are also queried via the MCPs. Like the DPNI, each DPMAC also requires one private DPMCP.

Table continues on the next page...

Table continued from the previous page...

Object	Private to DPNI?	Cardinality	Comments
DPCON	Yes	Rx hash size/ number of transmitter queues ("num_queues") of the DPNI.	DPCONs are used to distribute Rx or Tx Confirmation traffic to different GPPs, via affine DPIO objects. The implication is that one DPCON must be available for each GPP we want to distribute Rx or Tx Confirmation traffic to. Rx and Tx Confirmation share the same DPCONs if they are available. (If for example GPP #0 processes both types of traffic, one DPCON is enough. If in addition GPP #1 processes only Tx Confirmation traffic, then a second DPCON is necessary.) Since we must be able to distinguish between traffic from different NIs arriving on the same GPP, the DPCONs must be private to the DPNIs. These design constraints may cause a relatively large consumption of DPCONs by DPNIs with large Rx distribution width. The DPNI's Rx distribution width is implemented by the "num_queues" property (see Rx hashing on page 709 for extra information). Notes: DPCONs' main hardware resource are the Work Queues (WQs). The DPCONs come in 2 flavours: 2-WQ and 8-WQ DPCONs, depending on the number of traffic class priorities the object is going to support. (Note: the Ethernet driver only supports one traffic class at the moment, so using 2-WQ DPCONs is safe and enough.) The MC firmware can convert any number of 8-WQ DPCONs to four times as many 2-WQ DPCONs, depending on the static configuration provided at boot. Since WQs are a limited hardware resource, DPCONs tend to be limited, too, especially the 8-WQ flavor. The DPNIs being one of the major consumers of DPCONs, the current SDK ships with a default configuration where a number of the 8-WQ DPCONs are converted to 2-WQ DPCONs, thereby increasing their availability. Note that DPIO objects themselves transparently consume DPCONs (one per DPIO object), which therefore must be substracted from the total number available to the DPNIs (they need not be explicitly declared in the DPL, but they are simply not available to the rest of the system). The system can provide up to 64 8-WQ DPCONs (and up to 256 2-WQ DPCONs and combinations thereof). So for a container with 8 DPIOs, only up to 56 8-WQ DPCONs will be in fact available for DPNI configuration.
DPIO	No	One per running GPP	DPIOs are used to provide data availability notifications to the GPPs. For each GPP that we want to distribute traffic to, there must be an affine DPIO. While DPIOs are the source of data availability interrupts, the DPCONs are used (among other things) to identify the NI that has produced ingress data to that GPP. Due to a known limitation, the number of DPIOs in a container must not be less than the number of running GPPs. The static DPL in this release defensively provides 8 DPIOs at boot-time, one for each running GPP.

Table continues on the next page...

Table continued from the previous page...

Object	Private to DPNI?	Cardinality	Comments
DPMAC	No	User-defined.	DPMACs are proxy objects which link DPNIs to external PHYs on the board. DPMACs effectively decouple DPNIs from the PHYs they are linked to (if they are indeed linked to an external PHY, which is in fact transparent to the DPNI). As such, the DPMACs are not "owned" by a DPNI, which is unaware of their presence, but they can be "connected" to the DPNI, via the DPL file or 'restool'. DPMACs can be connected to other types of objects, too, such as the EVB. Having DPMACs connected to external PHYs depends on the board wiring and is strictly confined to the SerDes configuration. See also: DPMAC configuration on page 705.

8.3.2.3.3.2 Static DPNI definition

The default DPL provides a simple DPNI object definition, under the dpni@0 node as follows:

```
dpni@0 {
   options = "";
   num_queues = <1>;
   num_tcs = <1>;
};
```

The DPNI object is linked to a DPMAC object, also created in the DPL, via the “connections” node as follows:

```
connections {
   connection@5{
       endpoint1 = "dpni@0";
       endpoint2 = "dpmac@5";
   };
};
```

The DPNI object can be more complex, as in the following enhanced example of a DPNI node:

```
dpni@1 {
     options = "DPNI_OPT_HAS_KEY_MASKING";
     num_tcs = <1>;
     num_queues = <8>;
     mac_filter_entries = <64>;
};
```

In this example, dpni@1 has more options declared than dpni@0 in the previous example. In addition, it can distribute traffic to more GPPs than dpni@0, as declared by the “num_queues” attribute.

The following section describes the DPNI bindings in the DPL file.

8.3.2.3.3.2.1 DPNI bindings

.

- The `num_queues` attribute indicates the number of queues to be used for transmission as well as the number of Rx queues (hash distribution size). This also implicitly defines the number of queues used for Tx Confirmation, since each "sender" uses a dedicated queue for confirmations. This may impact the number of necessary DPCON objects - see "DPAA2 objects dependencies" chapter for details on resourcing the DPNI.
- `num_tcs` represents the number of traffic classes; maximum supported value is 8.
- `options` allows the creation of a DPNI object with non-default options
- Other possible attributes are listed below. Unless otherwise stated, attributes with value <0> receive a default, non-trivial, value from the MC firmware and can be skipped from the DPL altogether.

- fs_entries
- vlan_filter_entries
- mac_filter_entries

Note: See MC documentation for all available options and supported values.

8.3.2.3.3.3 DPMAC configuration

This section is a brief introduction to DPMAC objects and their relation to the DPNIs.

DPMACs are essentially proxies to external PHYs, which are board-level components and therefore not managed by the MC firmware.

DPMACs can be connected to other DPAA2 objects, such as DPNI, DPDMUX and DPSW. For example, to statically connect a DPMAC to a DPNI in the DPL file, the “connections” node is used:

```
connection@5{
     endpoint1 = "dpni@0";
     endpoint2 = "dpmac@5";
};
```

In this DPL example, DPNI0 is connected to DPMAC5, which the MC thereon connects to a lane depending on the current SerDes. Unlike most other DPAA2 objects, the id of the DPMAC (in this example, “5”) is relevant as the MC uses it to identify a physical MAC (at the moment, there is no other property of the DPMAC object to do that).

8.3.2.3.3.4 Dynamically creating a DPNI

This chapter documents the steps to create a DPNI using the *restool* utility and the *restool* wrapper scripts.

8.3.2.3.3.4.1 Using restool to create a DPNI

DPNIs can be dynamically created and plugged into the Linux container using the restool utility. Before creating a DPNI, one must create a number of DPAA2 objects (dependencies), for which multiple *restool* commands are needed. This section provides simple examples of commands that should be used to create a working DPNI (Linux network interface) and its dependencies. Usage of the Restool Wrapper Script bundled with the SDK is encouraged, because of their better ease-of-use.

In order to create an object, the “restool create” command must be executed and then the new object can be assigned to a container. For example, to create a DPBP object:

```
$ restool dpbp create
dpbp.1 is created under dprc.1
$ restool dprc assign dprc.1 --object=dpbp.1 --plugged=1
```

For automation purposes, the “--script" flag can be used, reducing the verbosity of the command output. Object properties can be specified at creation time as follows:

```
$ restool --script dpio create --channel-mode="DPIO_LOCAL_CHANNEL" --num-priorities=8
dpio.8
```

To create a DPNI, a number of DPMCP, DPBP and other dependencies are required, if they do not exist already in the container - refer to the DPAA2 objects dependencies on page 701 chapter for details on the types and number of DPNI dependencies. The static DPL from the current release already defines 8 DPIO objects, one for each running GPP, so adding new DPIOs is not normally required. Also, the maximum number of DPMACs supported on LS2088A and LS1088A are already created in the static (default) DPLs, so adding new ones is not necessary in the default configuration.

The general steps to create and configure a DPNI using restool are:

1. Create DPAA2 object dependencies (DPBPs, DPCONs, DPMCPs, etc);
2. Create and parametrize the DPNI;
3. Connect the DPNI to another object (typically but not necessarily a DPMAC).

The Restool Wrapper Script automatically take care of the DPNI resourcing and parametrization, therefore we encourage their use instead of bare *restool* for complex objects like the DPNI.

8.3.2.3.3.4.2 Restool Wrapper Scripts

User-friendly scripts are provided in the release rootfs to assist dynamic creation of DPNIs and associated dependencies. They also implement parameter restrictions and workarounds related to known limitations of the DPAA2 objects in the current SDK release.

The following scripts are available to interact with DPNI and DPMAC objects, respectively Linux network interfaces: ls-addni, ls-listni, ls-listmac.

1. **ls-addni**

   This script creates a new DPNI object, required dependencies (potentially DPBP, DPMCP, DPCON, DPMAC, depending on the options being passed to the script) and an associated Linux network interface. The script can be used to connect the newly-created DPNI to another DPNI, DPMAC or DPDMUX, which must be already created and not currently connected.

   The script supports a multitude of parameters to fine-tune configuration of the DPNI; in fact, it is intended to support every parameter as *restool* itself for creating DPNIs. An empty list of options will choose sensible defaults for maximal performance of the new DPNI, such as Rx hashing to the maximum number of cores.

   Adding a new DPNI has the effect of discovering the new object on the Linux mc-bus and probing it as a new Ethernet device. This results in a new Linux network interface becoming available. The new interface has the name `eth<X>`, where `X` depends on the order in which the interfaces are probed and on what other interfaces (e.g. PCIe NIC) are present. The mapping between the DPNI object and the interface name is shown by the `ls-listni` command.

   Utilization examples:

   -
     ```
     # ls-addni dpmac.6
     [70218.813064] fsl_dpaa2_eth dpni.4: Probed interface eth2
     Created interface: eth2 (object:dpni.4, endpoint: dpmac.6)
     ```

     This is probably the most typical usage example. It creates a network interface (eth2) and the underlying dpni (dpni.4) and connects it to an external MAC (dpmac.6).

     Connecting DPNIs to DPMACs is not the only option, though:

   -
     ```
     # ls-addni -n
     [70270.944458] fsl_dpaa2_eth dpni.5: Probed interface eth3
     Created interface: eth3 (object:dpni.5, endpoint: none)
     ```

     This command creates the unconnected object dpni.5 and the respective Linux interface eth3. Not being connected to anything, there is little practical use for this interface; therefore, a command such as the following would be used:

   -
     ```
     # ls-addni dpni.5
     [70312.255487] fsl_dpaa2_eth dpni.6: Probed interface eth4
     Created interface: eth4 (object:dpni.6, endpoint: dpni.5)
     ```

     This command creates another network interface eth4 (and the underlying dpni.6 object) and connects it with the previously created eth3 (dpni.5) interface.

     Notes:

     ls-addni --help list all supported options.

     Although it is technically possible to connect a DPNI to itself, the wrapper scripts do not support this;

2. **ls-listni**

   This script lists all the dpni objects available in the root and child containers, the associated network interface name, the end point and the label.

   Output after running the above examples (dpni.0 through dpni.3 had been statically defined in the DPL):

   ```
   # ls-listni
   dprc.1/dpni.6 (interface: eth4, end point: dpni.5)
   dprc.1/dpni.5 (interface: eth3, end point: dpni.6)
   dprc.1/dpni.4 (interface: eth2, end point: dpmac.6)
   dprc.1/dpni.3
   dprc.1/dpni.2
   ```

```
dprc.1/dpni.1
dprc.1/dpni.0 (interface: eth1, end point: dpmac.2)
```

3. **ls-listmac**

   This script lists all the dpmac objects available in the root and child containers, the associated network interface name, the end point and the label.

   Output after running the above examples (dpni.0 had been connected to dpmac.2 in the static DPL):

```
# ls-listmac
dprc.1/dpmac.10
dprc.1/dpmac.9
dprc.1/dpmac.8
dprc.1/dpmac.7
dprc.1/dpmac.6 (end point: dpni.4)
dprc.1/dpmac.5
dprc.1/dpmac.4
dprc.1/dpmac.3
dprc.1/dpmac.2 (end point: dpni.0)
dprc.1/dpmac.1
```

8.3.2.3.4 DPAA2 Ethernet features

This section presents the individual functions of the Linux DPAA2 Ethernet driver.

8.3.2.3.4.1 Bring up the bootstrap DPNI interface

From Linux, interfaces are visible through the '`ifconfig`' command. The DPAA2 interfaces are named as " `eth<X>` ", where X depends on the order in which the interfaces are probed.

The default DPL file shipped with the current BSP release contains one statically-defined DPNI object (DPNI.0).

DPNI.0 is configured with a minimal set of resources – e.g. it can only receive traffic on GPP0 – and its intended uses are network boot and low-bandwidth traffic. For fully-featured DPNI objects, dynamic configuration is recommended (see Dynamically creating a DPNI on page 705).

For IP connectivity between the default interface and an external host, first assign a valid IP address to it as in the following example:

```
$ ifconfig eth1 192.168.1.2
```

Assuming the remote peer has address 192.168.1.1, ping to test as shown:

```
$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=87.0 ms
```

8.3.2.3.4.2 Primary MAC address change

Changing the primary MAC address of a Linux Ethernet interface is supported without the need to bring down the interface.

For example:

```
$ ifconfig eth1 hw ether 02:00:C0:01:02:0a
```

```
$ ip link set dev eth1 address 02:00:C0:01:02:0b
```

8.3.2.3.4.3 Scatter/gather configuration

The Ethernet driver supports scatter/gather (S/G) on both the transmit and receive side. The S/G option can be configured through ethtool on Tx; on Rx, S/G support is always on. For example, in order to see the current state of the device features and hardware offloads for device ni0 :

```
$ ethtool -k eth1
Features for eth1:
[…]
scatter-gather: on
```

```
        tx-scatter-gather: on
[…]
```

In order to change the S/G status of the Linux Ethernet interface ni0:

```
$ ethtool -K eth1 sg off
Actual changes:
scatter-gather: off
        tx-scatter-gather: off
generic-segmentation-offload: off [requested on]

$ ethtool -K eth1 sg on
Actual changes:
scatter-gather: on
        tx-scatter-gather: on
generic-segmentation-offload: on
```

Notes:

- S/G support on the *egress* path, together with High DMA, which is also supported, allows for efficiently transmitting TCP segments from user-space, without copying them to kernel-space first ("Tx zero-copy").
- Egress S/G is necessary for other kernel features such as generic segmentation offload (GSO, implicitly turned on).
- The Ethernet driver support for S/G frames on the *ingress* path is transparent to the user.

8.3.2.3.4.4 Checksum offload configuration

The Ethernet driver supports hardware offloading of both Rx checksum validation and Tx checksum generation for TCP and UDP over IPv4/IPv6. The hardware checksum offload can be configured through ethtool.

For viewing the current state of the feature use the "-k" flag:

```
$ ethtool –k eth1
Features for eth1:
[…]
rx-checksumming: on
tx-checksumming: on
        tx-checksum-ipv4: on
        tx-checksum-ipv6: on
[…]
```

The checksum offload can be controlled separately on Rx and Tx paths as follows:

```
$ ethtool –K eth1 rx on|off
$ ethtool –K eth1 tx on|off

$ ethtool -k eth1 | grep tx-checksumming
tx-checksumming: off
```

8.3.2.3.4.5 MAC filtering

The DPAA2 hardware supports unicast and multicast MAC filters on the ingress path. In Linux, MAC unicast filtering can be accomplished with the help of MACVLAN interfaces. Kernel configuration and DPNI configuration are required to enable this feature, as follows:

- To enable support in the kernel, CONFIG_MACVLAN must be selected at compile-time, from the kernel menuconfig: `"Device Drivers -> Network device support -> Network core driver support -> MAC-VLAN support" .`

The Linux Ethernet driver allows adding and deleting of MAC filters via the standard "ip" command. An example of adding/deleting a MAC unicast address is the following:

```
$ ip link add link eth1 address <macvlan_mac_addr> eth1.1 type macvlan
$ ifconfig eth1.1 up
[…]
$ ip link delete eth1.1 type macvlan
```

Adding a multicast address is also possible using the "ip" command as follows:

```
$ ip maddr add 01:00:00:00:00:01 dev eth1
```

8.3.2.3.4.6 Large frame support

The DPAA2 hardware supports large frames. The Ethernet driver correlates between the Layer-2 maximum frame length (MFL) and Layer-3 MTUs. The maximum MTU that a Linux user can request on a DPAA2 Ethernet interface is 10222 bytes.

Notes:

- Outgoing packets larger than the current MTU are going to be fragmented by the kernel stack.
- All Ethernet devices on the same LAN must have the same MTU
- Ingress frames larger than MTU are accepted by the Ethernet driver

8.3.2.3.4.7 Generic receive offload

The DPAA2 Ethernet driver is integrated with the kernel's generic receive offload (GRO) support. GRO is enabled by default and is configurable via "ethtool":

```
$ ethtool -k eth1 | grep generic-receive-offload
 generic-receive-offload: on
$ ethtool -K eth1 gro off
$ ethtool -k eth1 | grep generic-receive-offload
 generic-receive-offload: off
```

NOTE

For better performance, GRO should be disabled on the receiving interfaces in certain scenarios such as IP Forwarding.

8.3.2.3.4.8 Egress traffic shaping

The DPAA2 Ethernet interface supports traffic shaping in the egress path. Egress shaping can be set or cleared via a per-interface entry in SysFS:

```
$ echo M N > /sys/class/net/eth1/tx_shaping
```

where:

M is the maximum throughput, expressed in Mbps.

N is the maximum burst size, expressed in bytes, at most 63487.

To remove shaping, use M=0, N=0.

8.3.2.3.4.9 Rx hashing

The DPAA2 Ethernet driver supports hash distribution of ingress flows, based on some of the common L2/L3/L4 fields. Configuration is done via standard "ethtool" support as follows:

```
$ ethtool –N <ethX> rx-flow-hash <proto_type> <header_fields>
```

The set of header fields from which the hash key is extracted is configured globally for all protocols and the protocol type parameter is ignored.

The following fields are supported:

- m - Ethernet destination address
- v - VLAN tag
- t - L3 protocol
- s - IPv4 source address
- d - IPv4 destination address

- f - L4 bytes 0 & 1 [TCP/UDP source port]
- n - L4 bytes 2 & 3 [TCP/UDP destination port]

The “r” flag (discard all packets of this flow type) is not supported.

For example, Rx hashing based on IP source and destination address can be configured with the following command:

```
$ ethtool -N <ethX> rx-flow-hash udp4 sd
```

The current hashing configuration can be viewed using the “-n” flag:

```
$ ethtool -n <ethX> rx-flow-hash udp4
```

The protocol type parameter is ignored; the configuration applies to both UDPv4 and TCPv4.

Note: By default, Ethernet interfaces start with hashing enabled on a 5-tuple key (IP proto, IP src/dst addresses, L4 src/dst ports). If an Ethernet interface is created with a single queue then hashing is not supported.

Interfaces created dynamically with "ls-addni" have a number of queues equal to the number of available CPUs, unless explicitly requested otherwise, so they'll have hashing enabled by default. For DPNIs statically defined inside a DPL file, in order to allow hashing the "num_queues" property must have a value larger than 1 and there must also be a sufficient number of DPCON objects available.

For full details and examples on dynamic DPNI creation, refer to this chapter.

8.3.2.3.4.10 Rx flow steering

The DPAA2 Ethernet driver supports steering of ingress traffic, directing flows to specific GPPs based on exact-match operations on some of the common L2/L3/L4 fields. The advantage versus Rx hashing on page 709 is cache locality of ingress data: the user-space applications that actually process the traffic make better use of the local GPP's cache than if the traffic were processed on another GPP. The disadvantage stems from the static configuration of flow affinity and from the fact that flow characteristics (e.g. L4 ports) must be known in advance, which is not always possible in real scenarios.

Configuration is done via standard "ethtool" support as follows:

```
$ ethtool -N eth1 flow-type <proto_type> <header_field> <value> [m <mask>] action <cpu_id>
```

Steering is supported for the following protocols:

- ethernet (flow-type `ether`)
- IPv4 (flow-type `ip4`)
- TCP, UDP over IPv4 (flow-type `tcp4`, `udp4`)

Supported fields are as follows:

- `src`, `dst` (L2 source/destination address; only for `ether` flow type)
- `dst-mac` (only for `ip4`, `udp4`, `tcp4` flow types)
- `vlan` (all flow types)
- `l4proto` (only for `ip4`)
- `src-ip`, `dst-ip`, `src-port`, `dst-port` (for `ip4`, `udp4`, `tcp4`)

Masking of header fields is also supported.

For example, in order to set up flow steering based on destination IP:

```
$ ethtool -N eth1 flow-type ip4 dst-ip 192.168.1.0 action 0
```

or subnet:

```
$ ethtool -N eth1 flow-type ip4 dst-ip 192.168.2.0 m 0.0.0.255 action 1
```

NOTE

The MC firmware and Linux Ethernet driver will only configure flow-steering if `DPNI_OPT_HAS_KEY_MASKING` is set in the "options" list of the DPNI object either via the DPL node or via restool - e.g.:

```
dpni@1 {
[…]
    options = "DPNI_OPT_HAS_KEY_MASKING";
[…]
};
```

respectively:

```
restool dpni create [...] --options=DPNI_OPT_HAS_KEY_MASKING
```

NOTE

Flow steering is currently supported on LS2088A, but not on LS2080A or LS1088A.

8.3.2.3.4.11 Flow Control Pause Frames

The DPAA2 Ethernet interfaces support sending and responding to pause frames, as part of the Ethernet flow control mechanism. The behavior of the pause frames is described in the IEEE 802.3x standard. In a nutshell, in a scenario involving a full duplex link, if the sender is sending at a higher rate than the receiver can process frames, the receiver can choose to send a special kind of frame, called pause frame, which asks the sender to halt the transmission of traffic for a specified period of time.

Pause frame control is integrated into ethtool. By default it's enabled for both Tx and Rx.

```
~# ethtool -a eth1
Pause parameters for eth1:
Autonegotiate:  on
RX:             on
TX:             on

~# ethtool -A eth1 rx off
~# ethtool -a eth1
Pause parameters for eth1:
Autonegotiate:  on
RX:             off
TX:             on
```

Currently, there's no support in configuring pause frame autonegotiation independently from link general autonegotiation. Hence, if link autonegotiation (e.g. rate and duplex) is on, it is on pause frames as well, and viceversa.

Pause frames are interpreted at the MAC level. Therefore, the counters for pause frames are visible when configuring netdevice objects for DPMAC objects using the `CONFIG_FSL_DPAA2_MAC_NETDEVS=y` kernel option. This option will bring up a set of additional Linux network interfaces named `macX`, one for each probed DPMAC object in the system.

```
~# ethtool -S mac1 | grep pause
     rx pause: 106731474
     tx b-pause: 15287253
```

If the driver is configured with Tx pause frames on, the hardware will start sending pause frames when the interface enters a congestion state on the Rx side.

If the driver is configured with Rx pause frames on, it will respond to any pause frames received on the line by reducing the send rate.

NOTE

Interrogating the current state of the flow control support and changing it through ethtool works when the interface is up.

8.3.2.3.4.12 Ethernet Priority-based Flow Control

The DPAA2 Ethernet interfaces support sending and responding to 802.1Qbb PFC (Priority-based Flow Control) frames, also known as CBFC (Class Based Flow Control) frames. PFC is a function of the 802.1 DCB (Data Center Bridging) standard, enabling lossless semantics at L2 on the Ethernet medium. Eight different classes of service (802.1p Ethernet priorities) are available as expressed through the 3-bit PCP field in an IEEE 802.1Q (VLAN) header added to the frame.

The DPAA2 Ethernet driver supports enabling PFC for a subset of the traffic classes. This configuration is done using a higher level protocol, LLDP - Link Layer Discovery Protocol. In Ubuntu, this protocol is implemented by the lldpad package, containing lldpad - the agent daemon - and lldptool - the client program.

Before attempting to configure PFC, make sure lldpad is installed and running:

```
~# apt-get update
~# apt-get install -y lldpad
~# service --status-all
~# service lldpad status
  lldpad.service - LSB: Start and stop the lldp agent daemon
Loaded: loaded (/etc/init.d/lldpad; bad; vendor preset: enabled)
Active: active (running) since Mon 2017-08-21 11:43:45 UTC; 2h 33min ago
Docs: man:systemd-sysv-generator(8)
CGroup: /system.slice/lldpad.service
—4542 /usr/sbin/lldpad -d
```

The LLDP agent daemon will register all the active interfaces in the system. In order to configure PFC for an interface (e.g. eth1) please run the following commands:

- Set the LLDP operation mode - in this case, to send and receive LLDP packets. This is required in order for PFC changes to take effect.

  ```
  lldptool –L –i eth1 adminStatus=rxtx
  ```

- Enable PFC for priorities 1, 2, and 4 on ni0.

  ```
  lldptool -T -i eth1 -V PFC -c enabled=1,2,4
  ```

- Display priorities enabled for PFC on ni0.

  ```
  lldptool -t -i eth1 -V PFC -c enabled
  ```

In order to disable PFC, you can run the following command:

```
lldptool -T -i eth1 -V PFC -c enabled=none
```

When setting PFC for the first time since boot, the DPAA2 Ethernet driver will configure static ingress traffic classification based on VLAN PCP. In order for this to work, you need to configure the DPNI with a number of traffic classes that's greater than 1 - preferably `num_tcs=8`, since there are a total of 8 priorities handled by the 3-bit PCP VLAN field.

The LLDP interface configuration is persistent across reboot and stored in the lldpad configuration files.

It's not advised to change the PFC configuration when the interface is handling heavy traffic.

There's a current **known limitation** for PFC to work only with DPNIs created using the DPL. DPNIs created with restool will not behave as expected.

8.3.2.3.4.13 XDP support

The DPAA2 Ethernet driver offers support for XDP (eXpress Data Path) programs. Support is enabled by default and no special configuration is needed.

XDP is a high performance data path in the Linux kernel, which allows for fast and programmable frame processing.

XDP programs are based on eBPF (extended Berkeley Packet Filter) and some basic examples can be found in the kernel source tree, in samples/bpf. Samples of the associated userspace apps that load the XDP program and attach it to the desired network interface can also be found there. For example, the "xdp1" sample program can be loaded by running:

```
./samples/bpf/xdp1 -N <interface_index>
```

The userspace applications that load XDP programs have to be built for arm64. XDP programs are compiled using clang/llvm, with minimum required version being 6.0. We recommend building natively, following the steps described in samples/bpf/README.rst.

The currently supported actions are:

- *XDP_DROP* : any frame for which this action is selected is dropped immediately by the driver
- *XDP_PASS* : frame follows the standard processing path and is sent to the network stack
- *XDP_TX* : frame is forwarded back to the same interface
- *XDP_REDIRECT* : frame is forwarded to another interface; only available on the 4.14 kernel

The driver also supports header updates that change the frame header size, but only on the 4.14 kernel.

Scatter/gather frames are not handled by the XDP program and will go through the regular path to the stack.

8.3.2.3.4.13.1 Building XDP Kernel Samples

In order to use XDP programs from the kernel `bpf/samples` folder, these are the steps for building them natively:

1. Prerequisites:

   Use a Layerscape board with LSDK1806 images, that has external network conectivity at least 6GB of disk space.

2. Install dependent packages:

   ```
   apt-get install git
   git apt-get install make
   apt-get install gcc
   apt-get install bc
   apt-get install elfutils
   apt-get install libelf-dev
   apt-get install bison
   apt-get install flex
   apt-get install cmake
   ```

3. Build the latest version of LLVM and clang (required to be >= 7.0):

   ```
   git clone https://git.llvm.org/git/llvm.git/ LLVM
   cd LLVM/tools
   git clone https://git.llvm.org/git/clang.git/
   cd ../..
   mkdir <llvm-build-dir>; cd <llvm-build-dir>
   cmake -DCMAKE_BUILD_TYPE=Release -DLLVM_TARGETS_TO_BUILD="BPF" ../LLVM/
   make -j 8
   ```

4. Download the 4.14 kernel sources from the LSDK release

5. Build the bpf samples:

   ```
   cd <kernel-src-dir>
   make mrproper
   make defconfig; make lsdk.config
   make headers_install
   make samples/bpf/ LLC=<llvm-build-dir>/bin/llc CLANG=<llvm-build-dir>/bin/clang
   ```

The resulting binaries will be located in `<kernel-src-dir>/samples/bpf`.

8.3.2.3.4.14 MQPRIO qdisc support

The DPAA2 Ethernet driver supports the MQPRIO qdisc, configurable through the tc tool.

MQPRIO (Multiqueue Priority Qdisc) is a simple queuing discipline that allows mapping traffic flows to hardware queue ranges, using priorities and a configurable priority to traffic class mapping. When creating the qdisc, the user can pass the number of traffic classes handled by the netdevice, the skb priority to traffic class map, and the hardware offloading flag.

For example:

```
$ tc qdisc add dev <ethX> root handle 1: mqprio num_tc 2 map 0 0 1 1 hw 1
```

The above translates to:

- The mqprio qdisc has 2 traffic classes (num_tc 2)
- The qdisc depends on hw offloading (hw 1)

- The skb prio to traffic class map is as follows:
  - — skb prio 0 -> tc 0
  - — skb prio 1 -> tc 0
  - — skb prio 2 -> tc 1
  - — skb prio 3 -> tc 1

Note: We only suppport the hardware offloading mode. Setting the "hw" param to 0 is not supported.

For setting the skb priority, the clsact qdisc can be used. Then we use the u32 filter to assign the skb priority based on traffic flow characteristics. This requires a recent iproute2-tc with clsact support compiled in:

```
$ tc qdisc add dev <ethX> clsact
$ tc filter add dev <ethX> egress prio 1 u32 match ip dport 7776 0xffff action skbedit priority
0
$ tc filter add dev <ethX> egress prio 1 u32 match ip dport 7777 0xffff action skbedit priority
1
$ tc filter add dev <ethX> egress prio 1 u32 match ip dport 7778 0xffff action skbedit priority
2
$ tc filter add dev <ethX> egress prio 1 u32 match ip dport 7779 0xffff action skbedit priority
3
```

In the above example, the destination port is used to assign an skb priority level. Outgoing IPv4 frames with port id 7778 and 7779 will be treated with highest priority.

Note: In order to use tc mqprio with the DPAA2 Ethernet driver, make sure the following kernel options are enabled:

```
CONFIG_NET_SCHED=y
CONFIG_NET_SCH_MQPRIO=y
```

Also, in order to run above examples, the following kernel options are also needed:

```
CONFIG_NET_CLS=y
CONFIG_NET_CLS_ACT=y
CONFIG_NET_ACT_SKBEDIT=y
CONFIG_NET_CLS_U32=y
CONFIG_CLS_U32_PERF=y
CONFIG_CLS_U32_MARK=y
CONFIG_NET_EMATCH_U32=y
```

Note: The DPNI has to be configured with a number of traffic classes greater than one - the maximum supported is `num_tcs`=8. Make sure the `num_tc` parameter passed at mqprio qdisc creation is not higher than the number of traffic classes supported by the DPNI.

8.3.2.3.4.15 CEETM support

DPAA2 platforms offer scheduling, shaping and prioritization capabilities through CEETM (Customer Edge Egress Traffic Management). The purpose of the CEETM block is to enhance networking performances by moving the egress QoS logic from software to hardware.

This section briefly describes what is supported and how CEETM can be configured through the Linux traffic control tool (tc) by using a custom queuing discipline.

8.3.2.3.4.15.1 Features

Each network interface (DPNI) can be associated with a LNI (logical network interface) containing a class queue channel. We don't support more than one channel per LNI. The LNI channel allows dual-rate shaping, which can be configured by specifying the CIR (committed information rate) and/or EIR (excess information rate). CBS (committed burst size) and EBS (excess burst size) values can also be configured.

We also support scheduling of class queues inside the channel; the number of queues cannot be larger than the configured number of traffic classes ("num_tcs" DPNI option), with a maximum value of 8.

Queues can be independent or part of a group:

- inside a group, queues are selected based on the WBFS (weighted bandwidth fair scheduling) algorithm. We support at most two class queue groups (referred to as group A and group B) with up to 4 queues each; if a single group is used (group A), up to 8 queues can be configured to be part of it.
- independent queues have fixed priorities and are subject to a strict priority scheduling (i.e. queue 1 will always be higher prio than queue 2)

Weighted queues share the priority of the group they belong to. Groups have configurable strict priorities relative to the independent queues. See the next section for an example on how to configure both weighted and independent queues.

We consider 0 to be the highest priority level.

8.3.2.3.4.15.2 Prerequisites

In order to use the CEETM feature, it must first be enabled in the kernel config file:

```
CONFIG_FSL_DPAA2_ETH_CEETM=y
```

Also, the following kernel option is needed:

```
CONFIG_NET_SCHED=y
```

The CEETM TC library (*q_ceetm.so*) should be located under /usr/lib/tc. It is built and deployed by default, without any user action needed.

8.3.2.3.4.15.3 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
... class add ... ceetm type root [cir CIR] [eir EIR] [cbs CBS] [ebs EBS] [coupled C]
... qdisc add ... ceetm type prio [prioA PRIO] [prioB PRIO] [separate SEPARATE]
... class add ... ceetm type prio [mode MODE] [weight W]
Update configurations:
... class change ... ceetm type root [cir CIR] [eir EIR] [cbs CBS] [ebs EBS] [coupled C]

Qdisc types:
root - associate a LNI to the DPNI
prio - configure the LNI channel's Priority Scheduler with up to eight classes

Class types:
root - configure the LNI channel
prio - configure an independent or weighted class queue

Options:
CIR - the committed information rate of the LNI channel
        dual-rate shaper (required for shaping scenarios)
EIR - the excess information rate of the LNI channel
        dual-rate shaper (optional for shaping scenarios, default 0)
CBS - the committed burst size of the LNI channel
        dual-rate shaper (required for shaping scenarios)
EBS - the excess of the LNI channel
        dual-rate shaper (optional for shaping scenarios, default 0)
C - shaper coupled, if both CIR and EIR are finite, once the
        CR token bucket is full, additional CR tokens are instead
        added to the ER token bucket
PRIO - priority of the weighted group A / B of queues
SEPARATE - groups A and B are separate
MODE - scheduling mode of class queue, can be:
        STRICT_PRIORITY
        WEIGHTED_A
        WEIGHTED_B
W - the weight of the class queue in the weighted group
```

8.3.2.3.4.15.4 Example

We present here an example of how tc ceetm qdisc can be used to create a complex egress shaping and scheduling configuration.

We start by configuring the LNI channel to allow a maximum egress rate of 1Gbps:

```
tc qdisc add dev <ethX> root handle 1: ceetm type root
tc class add dev <ethX> parent 1: classid 1:1 ceetm type root cir 1000mibit
```

We configure queue_1 and queue_2 to be part of group A, with a group priority of 3, and queue_4 and queue_5 to be part of group B with prio 1. Independent queues queue_0 and queue_3 are also configured. The resulting order of priorities is as follows (highest to lowest): {queue_0, group_B, queue_3, group_A}

Inside group A, queue_1 and queue_2 have equal weights; inside group B, queue_5 is given three times more bandwidth than queue_4. The weights are not absolute values, the relevant information is the ratio between them; it's recommended to use the value 100 for the queue with the lowest bandwidth.

```
tc qdisc add dev <ethX> parent 1:1 handle 2: ceetm type prio prioA 3 prioB 1 separate 1
tc class add dev <ethX> parent 2: classid 2:1 ceetm type prio mode STRICT_PRIORITY
tc class add dev <ethX> parent 2: classid 2:2 ceetm type prio mode WEIGHTED_A weight 100
tc class add dev <ethX> parent 2: classid 2:3 ceetm type prio mode WEIGHTED_A weight 100
tc class add dev <ethX> parent 2: classid 2:4 ceetm type prio mode STRICT_PRIORITY
tc class add dev <ethX> parent 2: classid 2:5 ceetm type prio mode WEIGHTED_B weight 100
tc class add dev <ethX> parent 2: classid 2:6 ceetm type prio mode WEIGHTED_B weight 300
```

Additionally, we define flows based on IP destination address and match them to the class queues:

```
# Flow 1 - queue 0
tc filter add dev <ethX> parent 1: protocol ip u32 match ip dst 192.85.2.2/32 flowid 1:1
tc filter add dev <ethX> parent 2: protocol ip u32 match ip dst 192.85.2.2/32 flowid 2:1

# Flow 2 - queue 1
tc filter add dev <ethX> parent 1: protocol ip u32 match ip dst 192.85.2.3/32 flowid 1:1
tc filter add dev <ethX> parent 2: protocol ip u32 match ip dst 192.85.2.3/32 flowid 2:2

# Flow 3 - queue 2
tc filter add dev <ethX> parent 1: protocol ip u32 match ip dst 192.85.2.4/32 flowid 1:1
tc filter add dev <ethX> parent 2: protocol ip u32 match ip dst 192.85.2.4/32 flowid 2:3

# Flow 4 - queue 3
tc filter add dev <ethX> parent 1: protocol ip u32 match ip dst 192.85.2.5/32 flowid 1:1
tc filter add dev <ethX> parent 2: protocol ip u32 match ip dst 192.85.2.5/32 flowid 2:4

# Flow 5 - queue 4
tc filter add dev <ethX> parent 1: protocol ip u32 match ip dst 192.85.2.6/32 flowid 1:1
tc filter add dev <ethX> parent 2: protocol ip u32 match ip dst 192.85.2.6/32 flowid 2:5

# Flow 6 - queue 5
tc filter add dev <ethX> parent 1: protocol ip u32 match ip dst 192.85.2.7/32 flowid 1:1
tc filter add dev <ethX> parent 2: protocol ip u32 match ip dst 192.85.2.7/32 flowid 2:6
```

Assuming the initial throughput of each flow was 200Mbps, the final output is:

```
flow 1 (queue_0) - 200Mbps
flow 2 (queue_1) - 100Mbps
flow 3 (queue_2) - 100Mbps
flow 4 (queue_3) - 200Mbps
flow 5 (queue_4) - 200Mbps
flow 6 (queue_5) - 200Mbps
```

If initial throughput per flow was 600Mbps, the final output is:

```
flow 1 (queue_0) - 600Mbps
flow 2 (queue_1) - 0Mbps
flow 3 (queue_2) - 0Mbps
flow 4 (queue_3) - 0Mbps
flow 5 (queue_4) - 100Mbps
flow 6 (queue_5) - 300Mbps
```

Note: In order to run this example, the following kernel configs are also needed:

```
CONFIG_NET_CLS=y
CONFIG_NET_CLS_ACT=y
CONFIG_NET_CLS_U32=y
CONFIG_NET_EMATCH=y
CONFIG_NET_EMATCH_U32=y
```

8.3.2.3.4.16 Interface statistics

DPAA2 Ethernet interface counters can be read via either of two standard tools, but there is a subtle difference:

- " `ifconfig ethX` " counters reflect packets received by the Ethernet driver – i.e. those frames that have passed through the Rx filters (if any are active) and have been effectively processed by the Ethernet driver on the GPP, and possibly by the kernel stack. These are software counters, maintained by the Ethernet driver and the networking stack.
- " `ethtool -S ethX` " counters reflect more detailed counters, from two categories:
  1. Statistics maintained by the DPAA2 hardware. These largely correspond in meaning to the standard "ifconfig" counters, but the values may be different from the "ifconfig" counters - i.e. they may reflect frames that have not been received on the GPP, such as those dropped by the ingress policer or due to MAC filtering. Also noteworthy is that retrieving these counters requires a series of calls into the MC firmware, which could make the operation potentially slower.
  2. Advanced counters, specific to the DPAA2 Ethernet driver. These are software-maintained driver-specific counters which do not fit into the standard " `ifconfig` " set.

     The following detailed counters are presented by the '`ethtool -S`' command:

     a. Hardware-maintained counters (prefixed by the "[hw]" tag):

        — `rx frames`: number of valid frames received from the DPNI hardware;

        — `rx bytes`: number of bytes comprised within the "rx frames" counter;

        — `rx mcast frames`: number of valid multicast frames;

        — `rx mcast bytes`: number of bytes included in "rx mcast frames";

        — `rx bcast frames`: number of valid broadcast frames;

        — `rx bcast bytes`: number of bytes included in "rx bcast frames";

        — `tx frames`: number of valid frames presented for transmission;

        — `tx bytes`: number of bytes included in "tx frames";

        — `tx mcast frames`: number of valid egress multicast frames;

        — `tx mcast bytes`: number of bytes included in "tx mcast frames";

        — `tx bcast frames`: number of valid egress broadcast frames;

        — `tx bcast bytes`: number of bytes included in "tx bcast frames";

        — `rx filtered frames`: number of valid frames but dropped because e.g. of MAC filtering;

        — `rx discarded frames`: number of frames with various physical errors;

        — `rx nobuffer discards`: number of frames discarded due to lack of buffers;

        — `tx discarded frames`: number of frames with Tx errors;

        — `tx confirmed frames`: number of Tx confirmed frames

     b. Software-maintained, driver-specific counters (prefixed by the "[sw]" tag):

        — `tx conf frames`: number of frames presented back to the Ethernet driver in the Tx confirmation queues. In an idle system, this counter should be equal to "tx frames";

        — `tx conf bytes`: number of bytes comprised by the "tx conf frames" counter;

— `tx sg frames`: number of egress frames in scatter-gather format; these are a subset of "tx frames", the difference being contiguous frames;

— `tx sg bytes` : number of bytes comprised in "tx sg frames";

— `tx realloc frames`: number of frames which had to be reallocated in the driver due to insufficient skb headroom; if a significant number of Tx frames are realloc'ed, it may be an indicator of suboptimal networking performance

— `rx sg frames`: number of frames received in scatter-gather format; typically this reflects frames larger than the largest buffer that can be used at the time of reception;

— `rx sg bytes`: number of bytes comprised in "rx sg frames";

— `enqueue portal busy`: number of times the Ethernet driver had to retry the frame enqueue command (on the egress path) due to QBMan portal being busy;

— `dequeue portal busy`: number of times the Ethernet driver had to retry the frame dequeue command (on the ingress path) due to QBMan portal being busy;

— `channel pull errors`: number of dequeue errors which are not due to the portal being busy;

— `cdan` : number of Channel Dequeue Available Notifications (CDANs) received by the Ethernet driver (Rx and Tx Conf paths). Each CDAN corresponds to one DPIO interrupt and triggers a NAPI processing cycle which can process Rx or Tx Conf frames (or both).

— `tx congestion state`: whether the Tx queues are currently congested or not; if congestion state is 1, it means one or more Tx queues have stopped and are waiting for the hardware to finish transmitting the frames already enqueued from the Ethernet driver;

8.3.2.4 Setting up Ethernet Switch Capability

8.3.2.4.1 Ethernet Switch overview

The following switch features are supported:

- Dynamic learning
- Adding/deleting static FDB entries
- Adding/deleting static multicast entries
- Configuring multicast groups
- Flooding of broadcast and multicast traffic
- Forwarding of unicast traffic that is both VLAN tagged and untagged
- Setting STP state of ports

Important notes:

- Learning is supported and enabled by default.
- Learned FDB entries cannot be displayed.
- There is no support to flush the FDB.

Acronyms and abbreviations:

- DPSW - DPAA2 object modelling an L2 Switch
- DPNI - DPAA2 object modelling a network interface

8.3.2.4.2 Switch object creation

A switch object can be created:

- Dynamically using the restool as described in Using restool for dynamic object creation on page 719 or
- Statically in a DPL file as described in Using the data path layout file (DPL) on page 720

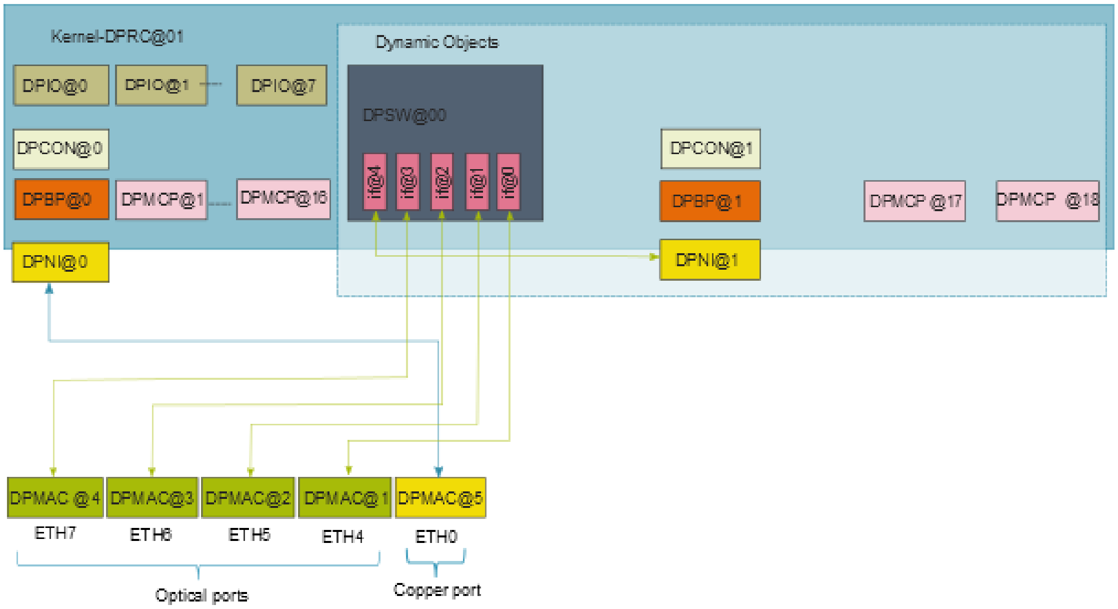
8.3.2.4.2.1 Using restool for dynamic object creation

A switch can be created at run-time, using restool. Before creating the switch, a number of DPAA2 objects (dependencies) have to be added, for which multiple restool commands are needed. Switch requires at least a DPMCP object, which is created like:

```
$ restool dpmcp create
```

The following section describes the main commands to create a switch starting from the **dpl-eth.0x2A_0x41.dtb** DPL file.

Figure 160. Dynamic DPSW demo



NOTE

When a new object is created via restool, an object with the id of the first available resource will be returned.

NOTE

Depending on the board type, DPMAC availability varies. For more details please refer to **Limitations and Known Issues**.

8.3.2.4.2.1.1 Creating a DPSW

The switch object is created by this command:

```
$ restool dpsw create --num-ifs=5 --max-vlans=16 --max-fdbs=1 --options="DPSW_OPT_CTRL_IF_DIS"
```

The command specifies some configuration options, including the number of ports in the switch, the maximum number of VLANs that can be used on the switch – including VLAN1 used implicitly, plus the number of FDBs.

For all the configuration options and parameters see the help output:

```
$ restool dpsw create -h
```

Currently VLAN private FDBs are not supported; a single shared FDB is used. Also control traffic is not supported, so option DPSW_OPT_CTRL_IF_DIS has to be specified.

8.3.2.4.2.1.2 Connecting the switch

Linking the switch ports to other objects is done with:

```
restool dprc connect "$RC" --endpoint1="$SW".0 --endpoint2=dpmac.1
restool dprc connect "$RC" --endpoint1="$SW".1 --endpoint2=dpmac.2
restool dprc connect "$RC" --endpoint1="$SW".2 --endpoint2=dpmac.3
restool dprc connect "$RC" --endpoint1="$SW".3 --endpoint2=dpmac.4
restool dprc connect "$RC" --endpoint1="$SW".4 --endpoint2="$NI"
```

In the context of the configuration script, these commands create the following layout:

- sw0p0 (dpsw@1/if@0) <-> dpmac@1 (SFP+ port, labeled ETH4 on RDB front panel)
- sw0p1 (dpsw@1/if@1) <-> dpmac@2 (SFP+ port, labeled ETH5 on RDB front panel)
- sw0p2 (dpsw@1/if@2) <-> dpmac@3 (SFP+ port, labeled ETH6 on RDB front panel)
- sw0p3 (dpsw@1/if@3) <-> dpmac@4 (SFP+ port, labeled ETH7 on RDB front panel)
- sw0p4 (dpsw@1/if@4) <-> dpni@1

For SerDes **0x2A_0x41**, DPMACs 1-4 are mapped to the optical PHYs while DPMACs 5-8 are mapped to the copper PHYs. User can choose to connect any of the optical or copper ports and any NIs to the switch.

8.3.2.4.2.1.3 Enabling the switch

This command plugs the switch object on the bus, in the Linux resource container. The switch driver probes the switch and presents the associated network interfaces in Linux.

```
$ restool dprc assign "$RC" --object="$SW" --plugged=1
```

After enabling the switch it can be configured from Linux using the commands specified in Commands supported on page 722.

8.3.2.4.2.1.4 Restool wrapper scripts

For user convenience the **ls-addsw** script is provided to assist creation of a new DPSW object.

To replicate the setup described in sectionConnecting the switch on page 719 the following commands are required:

```
$ ls-addni -n
$ ls-addsw -i=5 dpmac.3 dpmac.4 dpmac.5 dpmac.6 dpni.1
```

The first command creates a new DPNI object and the second the DPSW object. DPMACs are already defined in the DPL file. To display the DPNIs and DPMACs available one can use *ls-listni* or *ls-listmac* commands. The DPIO, DPBP, DPCON and DPMCP objects that are dependencies for the new objects are created by the script, without user intervention.

For all the script options and parameters see the help:

```
$ ls-addsw -h
```

The endpoints are connected in the specified order to switch ports. If there are less endpoints than the number of interfaces, the user can later add the rest using *ls-addni* or *restool* commands.

8.3.2.4.2.2 Using the data path layout file (DPL)

A switch object may be defined statically in the DPL, allowing it to be created automatically during platform initialization. Below is an example of switch definition in the DPL:

```
dpsw@0 {
      compatible = "fsl,dpsw";
      options = "DPSW_OPT_CTRL_IF_DIS";
      max_vlans = <0x10>;
      max_fdbs = <0x1>;
      num_fdb_entries = <0x400>;
      fdb_aging_time = <0x12c>;
      num_ifs = <0x5>;
      max_fdb_mc_groups = <32>;
};
```

This example is for a 5-port switch that includes support for up to 16 VLAN IDs, including VLAN 1 (that is internally used by the switch), up to 1024 FDB entries, and up to 32 multicast groups.

Links are defined in the DPL 'connections' section as follows:

```
connections {
       connection@1 {
             endpoint1 = "dpsw@0/if@0";
             endpoint2 = "dpmac@1";
        };
        connection@2 {
             endpoint1 = "dpsw@0/if@1";
             endpoint2 = "dpmac@2";
        };
        connection@3 {
             endpoint1 = "dpsw@0/if@2";
             endpoint2 = "dpmac@3";
        };
        connection@4 {
            endpoint1 = "dpsw@0/if@3";
            endpoint2 = "dpmac@4";
        };
        connection@5 {
             endpoint1 = "dpsw@0/if@4";
             endpoint2 = "dpni@1";
       };
};
```

The generated layout is the one described in Connecting the switch on page 719.

8.3.2.4.3 Setting up the driver

To compile the driver, you enable the **FSL_DPAA2_ETHSW** option in the kernel's config. It can be found in *menuconfig* under the following items:

```
| -> Device Drivers
|   -> Staging drivers
|     -> Freescale DPAA2 devices
|       -> Freescale DPAA2 Ethernet Switch
```

The driver is enabled in the default kernel configuration file and uses the *switchdev* kernel support, thus it depends on the *NET_SWITCHDEV* option. To have access to the kernel interfaces for configuring bridges, *BRIDGE* kernel option has to be enabled.

After deploying the driver and instantiating a DPSW, the system will present a network interface for each switch port. These are used for control, not for I/O. Any I/O through the switch must be performed using the interfaces linked to the switch.

```
                   eth<N>        eth<N+1>    eth<N+2>    eth<N+3>
                     +            +            +            +
                     |            |            |            |       Kernel abstractions
                     |            |            |            |
   +-----------------------------------------------------------------------+
                     |            |            |            |
                     |            |            |            |       MC objects
                     |            |            |            |
              +---------------------------------------------------+
              |      +            +            +            +     |
              |  dpsw.0.0     dpsw.0.1     dpsw.0.2    dpsw.0.3   |
              |      +            +            +                  |
              |      |            |            |                  |
              |      |            |            |                  |
              |      |            |   dpsw.0   |                  |
              +---------------------------------------------------+
                     |            |            |
                     |            |            |
                     |            |            |
                     |            |            |
                     |            |            |
                     +            +            +
                   dpni.1       dpmac.1      dpni.2
```

To be able to use the switch, etwork interfaces have to be configured in a bridge using standard Linux tools like those in iproute2 suite.

Configuring the switch in Linux

1. Create a bridge interface

After the DPAA2 Ethernet Switch is created, network interfaces for each switch port are available. They need to be added to a bridge interface:

```
ip link add name <brX> type bridge
ip link set <brX> up
```

2. Configuring MAC addresses

To be able to configure the ports, a unique MAC address has to be configured on each interface:

```
ip link set eth<N> down
ip link set eth<N> address xx:xx:xx:xx:xx:xx
ip link set eth<N> up
```

3. Adding switch ports to bridge interface

Switch ports that are not added to the bridge interface cannot be used to forward traffic.

```
ip link set eth<N> master <brX>
```

8.3.2.4.4 Commands supported

The switch management commands supported are:

- ifup/ifdown (using ifconfig or similar)
- setting large frame size limit (using ifconfig or similar)
- retrieving statistics (using ifconfig or similar)
- configuring FDB (using bridge fdb)
- configuring multicast groups (using bridge fdb)
- configuring VLANs (using bridge vlan)
- configuring learning (using bridge link set)

8.3.2.4.4.1 Interface control

Any of the switch ports can be enabled/disabled using any of the following commands:

```
$ ifconfig eth<N> { up | down }
$ ip link set eth<N> { up | down }
```

Disabling the bridge device will also disable the assigned switch ports.

8.3.2.4.4.2 Maximum frame size configuration

The DPAA2 hardware supports large frames. Ethernet switch driver correlates between the Layer-2 maximum frame length (MFL) and Layer-3 MTUs. The maximum MTU that a Linux user can request on a DPAA2 Ethernet switch interface is 10218 bytes and has to be set on each port individually.

```
$ ifconfig eth<N> mtu <NN>
$ ip link set { eth<N> | dev eth<N> } mtu <NN>
```

Notes:

- Frames larger than the configured MTU will be dropped, so connected Ethernet devices need to have the same setting.
- All Ethernet devices on the same LAN must have the same MTU to avoid traffic loss.

8.3.2.4.4.3 Learning control

The switch is set by default to enable learning, and the learning can be controlled using this command:

```
$ bridge link set dev eth<N> learning { on | off }
```

NOTE

The command is executed on a switch port, although it does affect the learning function at switch level. Turning off learning does not remove the learned entries.

To establish a static topology, learning should be disabled before injecting any traffic.

8.3.2.4.4.4 FDB static entries

The default switch configuration does not include any static entries; these can be added using the *bridge fdb add* command, as shown below:

```
$ bridge fdb add xx:xx:xx:xx:xx:xx dev eth<N> master
```

8.3.2.4.4.5 Multicast entries

Multicast groups can be configured via bridge `fdb add|append|delete` commands:

```
$ bridge fdb add 01:00:05:00:00:13 dev eth6
$ bridge fdb append 01:00:05:00:00:13 dev eth7
$ bridge fdb append 01:00:05:00:00:13 dev eth8
$ bridge fdb append 01:00:05:00:00:13 dev eth9

$ bridge fdb del 01:00:05:00:00:13 dev eth9
```

The commands add all external ports, one by one, to the `01:00:05:00:00:13` multicast group. The last command removes the last port from the group.

8.3.2.4.4.6 VLAN configuration

All ports added to a bridge are added to the default VLAN. All untagged traffic received on switch ports is classified to VLAN 1, and all frames classified in VLAN 1 are sent out untagged on all ports. Additional VLANs can be added on the switch using these commands:

```
$ bridge vlan add vid 2 dev eth0
$ bridge vlan add vid 2 dev eth1
$ bridge vlan add vid 2 dev eth2
$ bridge vlan add vid 2 dev eth3
$ bridge vlan add vid 2 dev eth4
```

This example includes all five ports in VLAN 2. After running these commands the switch should allow frames tagged with vid 2 to pass through the switch; not all ports have to be included in the VLAN.

To remove a port from a given VLAN use the following command:

```
$ bridge vlan del vid 2 dev eth0
```

8.3.2.4.4.7 Port statistics

Hardware counters for switch ports are available through `ethtool`:

```
$ ethtool -S eth<N>
NIC statistics:
     rx frames: 0
     rx bytes: 0
     rx filtered frames: 0
     rx discarded frames: 0
     rx b-cast frames: 0
     rx b-cast bytes: 0
     rx m-cast frames: 0
     rx m-cast bytes: 0
     tx frames: 0
     tx bytes: 0
     tx discarded frames: 0
```

No software statistics are available for switch ports.

8.3.2.5 Setting Up Edge Virtual Bridge Capability

8.3.2.5.1 EVB overview

An edge virtual bridge allows the sharing of a physical connection between multiple entities (virtual hosts). It can act as a VEB or as a VEPA.

In VEB mode, traffic is forwarded between connected virtual hosts or between virtual hosts and uplink.

In VEPA mode, all traffic from virtual hosts is forwarded to uplink, bridging functions (including 'hairpin' forwarding) being performed by an external device.

Features supported:

- VEB/VEPA mode
- Traffic steering according to MAC, VLAN (in VEPA mode only) or MAC+VLAN
- Static FDB entries management (add/delete/show)
- Static multicast FDB entries management (add/delete/show)
- Flooding of broadcast and multicast traffic

8.3.2.5.2 EVB object creation

EVB objects can be created as follows:

- Dynamically using the restool as described in Using restool for dynamic object creation on page 724
- Statically in a DPL file as described in Using the data path layout file (DPL) on page 726

8.3.2.5.2.1 Using restool for dynamic object creation

A DPDMUX can be instantiated at run-time, using restool.

The following section describes the main commands to create an EVB and its dependencies starting from the **dpl-eth.0x2A_0x41.dtb** DPL file.

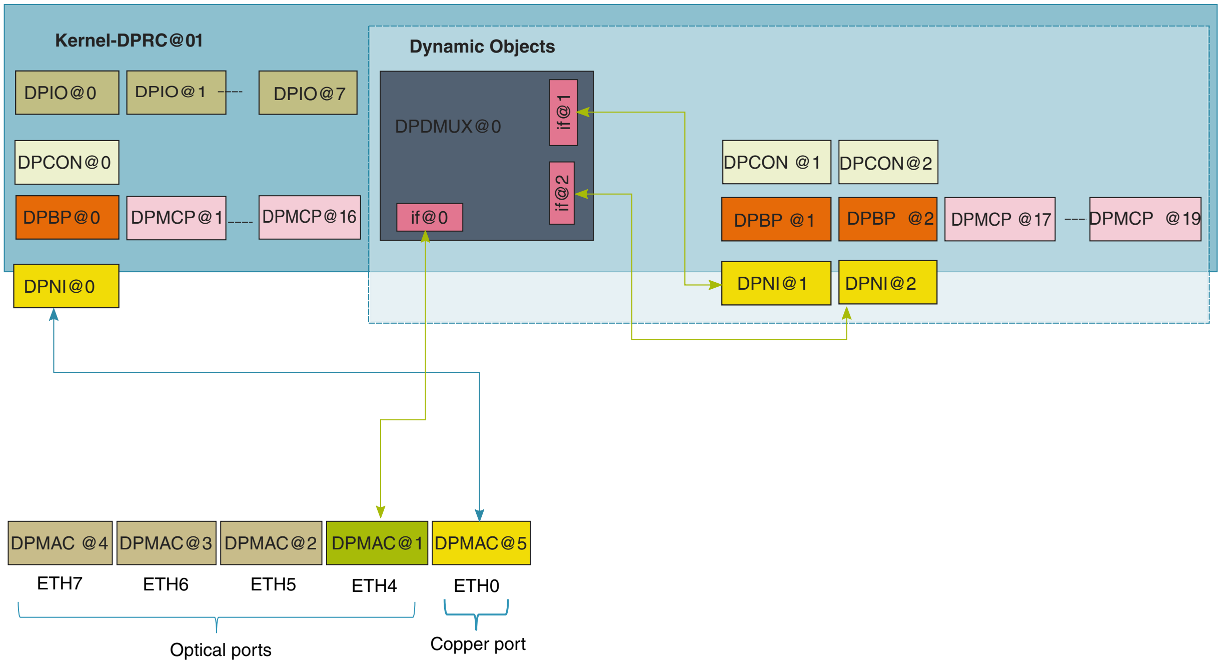


Figure 161. Dynamic DPDMUX demo

NOTE

When a new object is created using restool, an object with the ID of the first available resource is returned.

NOTE

Depending on the board type, DPMAC availability varies. For more details please refer to **Limitations and Known Issues**.

8.3.2.5.2.1.1 Creating a DPDMUX

The EVB is created by this command:

```
$ restool dpdmux create --num-ifs=2 --control-if=0                    \
          --options=DPDMUX_OPT_BRIDGE_EN --method=DPDMUX_METHOD_MAC      \
          --max-dmat-entries=8 --max-mc-groups=8 --manip=DPDMUX_MANIP_NONE
```

The command must specify the number of downlinks and the ID of the uplink (ranges for 0 to [number of downlinks -1]). The other parameters are optional. For more information about the available options see the output of the command:

```
$ restool dpdmux create -h
```

8.3.2.5.2.1.2 Connecting the EVB

Linking the EVB ports to other objects is done with:

```
$ restool dprc connect "$RC" --endpoint1="$MUX".0 --endpoint2="$MAC"
$ restool dprc connect "$RC" --endpoint1="$MUX".1 --endpoint2="$NI1"
$ restool dprc connect "$RC" --endpoint1="$MUX".2 --endpoint2="$NI2"
```

$RC represents the container for the objects, $MUX is the object identified for the EVB. The uplink is the endpoint with the id specified by *control-if* parameter at creation time.

8.3.2.5.2.1.3 Enabling the EVB

This command plugs the EVB object on the bus, in the Linux resource container. The EVB driver probes the switch and presents the associated network interfaces in Linux.

```
$ restool dprc assign "$RC" --object="$MUX" --plugged=1
```

8.3.2.5.2.1.4 Restool wrapper scripts

For user convenience the **ls-addmux** script is provided to assist creation of a new DPDMUX object.

Example to replicate setup in section Connecting the EVB on page 725 :

```
# ls-addmux -d=2 -u=0 dpmac.1
[ 4298.023745] dpaa2_evb dpdmux.0: probed evb device with 2 ports
Created EVB: evb0 (object: dpdmux.0, uplink: dpmac.1)
```

This command creates EVB evb0 (and the corresponding dpdmux.0 object) with two downlinks and the uplink connected to dpmac.1.

After creating the DPDMUX, its downlinks can be connected to DPNIs using **ls-addni** script:

```
# ls-addni dpdmux.0.1
Will allocate 8 DPCON objects for this hash size
[ 5118.645253] fsl_dpaa2_eth dpni.1: Probed interface ni1
Created interface: ni1 (object:dpni.1, endpoint: dpdmux.0.1)
# ls-addni dpdmux.0.2
Will allocate 8 DPCON objects for this hash size
[ 5122.169030] fsl_dpaa2_eth dpni.2: Probed interface ni2
Created interface: ni2 (object:dpni.2, endpoint: dpdmux.0.2)
```

8.3.2.5.2.2 Using the data path layout file (DPL)

A DPDMUX instance can statically be defined in the DPL file:

```
dpdmux@0 {
       compatible = "fsl,dpdmux";
       options = "DPDMUX_OPT_BRIDGE_EN";
       method = "DPDMUX_METHOD_MAC";
       manip = "DPDMUX_MANIP_NONE";
       control_if = <0>;
       num_ifs = <2>;
       max_dmat_entries = <8>;
       max_mc_groups = <8>;
};
```

Links are defined in the DPL 'connections' section:

```
connection@1{
        endpoint1 = "dpdmux@0/if@0";
        endpoint2 = "dpmac@1";
};
connection@2{
        endpoint1 = "dpni@1";
        endpoint2 = "dpdmux@0/if@1";
};
connection@3{
        endpoint1 = "dpni@2";
        endpoint2 = "dpdmux@0/if@2";
};
```

Based on the above configuration the DPDMUX ports are linked to:

- evb0 (dpdmux@1/if@0) <-> dpmac@1
- evb0p0 (dpdmux@1/if@1) <-> dpni@1

- evb0p1 (dpdmux@1/if@2) <-> dpni@2

NOTE

DPDMUX ports connected to a DPMAC must be configured before the others (e.g. connected to DPNIs).

8.3.2.5.3 Setting up the EVB driver

Driver compilation is enabled by default and is controlled by the FSL_DPAA2_EVB option in the kernel's config. This can be found in *menuconfig* under the following items:

```
| -> Device Drivers
|     -> Staging drivers
|         -> Freescale Management Complex (MC) bus driver
|             -> Freescale DPAA2 devices
|                 -> DPAA2 Edge Virtual Bridge
```

The kernel log will display a message when an EVB is probed as follows:

```
dpaa2_evb dpdmux.0: probed evb device with 2 ports
```

After deploying the driver and configuring an EVB (via DLP or restool), the system should present the following Linux interfaces after typing the 'ifconfig command':

```
evb0      Link encap:Ethernet HWaddr 00:00:00:00:00:00
          UP BROADCAST RUNNING MASTER 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)

evb0p0    Link encap:Ethernet HWaddr 00:00:00:00:00:00
          UP BROADCAST RUNNING SLAVE 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)

evb0p1    Link encap:Ethernet HWaddr 00:00:00:00:00:00
          UP BROADCAST RUNNING SLAVE 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)
```

Interface ***evb0*** represents the uplink and is also the handler for the EVB. Each other EVB port has its own interface. They are used for management and cannot be used for I/O. Any I/O through the EVB must be performed using the connected interfaces.

An EVB only forwards traffic to links that are enabled (peer interface is up) and only if the filtering rules on the peer interface do not lead to the frame being discarded. One way to ensure that all traffic subject to forwarding rules is actually forwarded by the EVB is to set the peer interface in promiscuous mode, as follows:

```
$ ip link set ni0 up promisc on
```

8.3.2.5.4 EVB commands supported

EVB management can be performed using the following generic Linux networking tools:

- interface up/down (using ifconfig or similar)
- setting large frame size limit (using ifconfig or similar)
- configuring FDB (using bridge fdb)
- configuring VLANs (using bridge vlan)
- configuring multicast groups (using bridge fdb)
- port statistics retrieval (ethtool or similar)

8.3.2.5.4.1 EVB interface control

Any of the EVB ports, or the EVB as a whole, can be enabled/disabled using any of the following commands:

```
$ ifconfig { evb0pX | evb0 } { up | down }
$ ip link set { evb0pX | evb0 } { up | down }
```

8.3.2.5.4.2 Maximum frame size configuration

The DPAA2 hardware supports large frames. EVB driver correlates between the Layer-2 maximum frame length (MFL) and Layer-3 MTUs. The maximum MTU that a Linux user can request on a DPAA2 EVB interface is 10222 bytes. Setting a value on a downlink port or uplink will update the value for all EVB interfaces.

```
$ ifconfig { evbX | evbXpy } mtu <NN>
$ ip link set { evbX | evbXpY | dev evbXpY } mtu <NN>
```

Notes:

- Frames larger than the configured MTU will be dropped, so connected Ethernet devices need to have the same setting.
- All Ethernet devices on the same LAN must have the same MTU to avoid traffic loss.

8.3.2.5.4.3 EVB FDB entries

The EVB method DPDMUX_METHOD_MAC allows configuration of FDB entries via a bridge utility as follows:

```
$ bridge fdb add 02:00:c0:a8:50:01 dev evb0p0
$ bridge fdb show
02:00:c0:a8:50:01 self permanent
01:00:5e:00:00:01 self permanent
```

The EVB method DPDMUX_METHOD_C_VLAN_MAC also allows configuration of FDB entries via a bridge utility as follows:

```
$ bridge fdb add 02:00:c0:a8:50:02 vlan 10 dev evb0p0 vlan 10
$ bridge fdb show dev evb0p0
02:00:c0:a8:50:02 self permanent
01:00:5e:00:00:01 self permanent
```

8.3.2.5.4.4 EVB VLAN assignment

The EVB method DPDMUX_METHOD_C_VLAN allows port VLAN assignment via a bridge utility as follows:

```
$ bridge vlan add vid 10 dev evb0p2
$ bridge vlan show dev evb0p2
port vlan ids
evb0p2 10
$ bridge vlan del vid 10 dev evb0p2
```

NOTE

This method is allowed only for VEPA mode.

8.3.2.5.4.5 EVB port statistics

EVB port statistics are available through *ip* or similar tools as follows:

```
$ ip -s link
[...]
9: evb0: <BROADCAST,MULTICAST,MASTER,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN mode
DEFAULT group default qlen 1000
       link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
       RX: bytes packets errors dropped overrun mcast
       0         0       0      0       0       0
       TX: bytes packets errors dropped carrier collsns
       384       6       0      0       0       0
10: evb0p0: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master evb0
state UNKNOWN mode DEFAULT group default qlen 1000
```

```
       link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
       RX: bytes packets errors dropped overrun mcast
       252       6       0      0       0       0
       TX: bytes packets errors dropped carrier collsns
       0         0       0      0       0       0
[...]
```

8.3.2.5.5 Forwarding methods overview

A DPAA2 DPDMUX instance can forward traffic using information from various fields in the frame headers:

- Forwarding by destination MAC address
- Forwarding by VLAN tag
- Forwarding by VLAN tag and destination MAC adress

8.3.2.5.5.1 Forwarding by destination MAC address

This method forwards frames according to the destination MAC address and the static rules added into the EVB forwarding database.

It is configured specifying *--method="DPDMUX_METHOD_MAC"* when the DPDMUX is created. It is the default value for the *ls-addmux* script.

Entries are configured in the FDB using *bridge fdb* command. See EVB FDB entries on page 728 section for more information.

Configuration example:

```
# Create a MUX with 2 downlinks and uplink connected to dpmac.1;
# forwarding method is by default DPDMUX_METHOD_MAC
$ ls-addmux -b -d=2 -u=0 dpmac.1

# Create a ni (dpni.1) and links it to evb0p0
$ ls-addni dpdmux.0.1

# Create a ni (dpni.2) and links it to evb0p1
$ ls-addni dpdmux.0.2

# Check MUX configuration
# $ restool dpdmux info dpdmux.0

# Configure ni1
$ ip netns add ns1
$ ip link set ni1 netns ns1
$ ip netns exec ns1 ifconfig ni1 192.168.10.10/24 up
$ ip netns exec ns1 ip link set ni1 promisc on

# Configure ni2
$ ip netns add ns2
$ ip link set ni2 netns ns2
$ ip netns exec ns2 ifconfig ni2 192.168.10.12/24 up
$ ip netns exec ns2 ip link set ni2 promisc on

# Connectivity checks [downlink - uplink ]
$ ip netns exec ns1 ping 192.168.10.13 -c 1
[..]
--- 192.168.10.13 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms

# Check EVB port statistics
$ ip -s link
[...]
4: evb0: <BROADCAST,MULTICAST,MASTER,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UNKNOWN mode
DEFAULT group default qlen 1000
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
    RX: bytes  packets  errors  dropped overrun mcast
    436        6        0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    460        6        0       0       0       0
5: evb0p0: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master evb0 state
UP mode DEFAULT group default qlen 1000
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
    RX: bytes  packets  errors  dropped overrun mcast
    364        6        0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    376        5        0       0       0       0
6: evb0p1: <NO-CARRIER,BROADCAST,MULTICAST,SLAVE,UP> mtu 1500 qdisc pfifo_fast master evb0
state DOWN mode DEFAULT group default qlen 1000
    link/ether 00:00:00:00:00:00 brd ff:ff:ff:ff:ff:ff
    RX: bytes  packets  errors  dropped overrun mcast
    0          0        0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    0          0        0       0       0       0
```

8.3.2.5.5.2 Forwarding by VLAN tag

This method forwards frames according to the VLAN tag of the frame, as set into the customer tag of the double-tagged frames.

It is configured specifying *--method*="*DPDMUX_METHOD_C_VLAN*" when EVB is in VEPA mode (--*options*="*DPDMUX_OPT_BRIDGE_EN*" is not set).

EVB port VLAN assignment is done with "*bridge vlan* command. See EVB VLAN assignment on page 728 section for more information.

Configuration example:

```
# Create a MUX with DPDMUX_METHOD_C_VLAN forwarding method,
# configured as a VEPA and with 2 downlinks and uplink connected
# to dpmac.1
$ ls-addmux -v -m=DPDMUX_METHOD_C_VLAN -d=2 dpmac.1

# Create a ni (dpni.1) and link it to evb0p0
$ ls-addni dpdmux.0.1

# Create a ni (dpni.2) and link it to evb0p1
$ ls-addni dpdmux.0.2

# Configure ni1
$ ip netns add ns1
$ ip link set ni1 netns ns1
$ ip netns exec ns1 ip link add link ni1 name ni1.6 type vlan id 6
$ ip netns exec ns1 ifconfig ni1.6 192.168.6.10
$ ip netns exec ns1 ip link set ni1 up
$ ip netns exec ns1 ip link set ni1 promisc on

# Configure ni2
$ ip netns add ns2
$ ip link set ni2 netns ns2
$ ip netns exec ns2 ip link add link ni2 name ni2.7 type vlan id 7
$ ip netns exec ns2 ifconfig ni2.7 192.168.7.12
$ ip netns exec ns2 ip link set ni2 up
$ ip netns exec ns2 ip link set ni2 promisc on

# For the downlinks interfaces also add the VLAN ids
$ bridge vlan add vid 6 dev evb0p0
$ bridge vlan add vid 7 dev evb0p1

# Connectivity checkings [example for downlink - uplink ]
$ ip netns exec ns1 ping -I ni1.6 192.168.6.13 -c 1

# Check VLAN assignment
$ bridge vlan show
```

8.3.2.5.5.3 Forwarding by VLAN tag and destination MAC address

This method forwards frames according to the VLAN tag and the destination MAC address of the frame .

It is configured specifying *--method="DPDMUX_METHOD_C_VLAN_MAC"* when the DPDMUX is created.

Entries are configured in the FDB using *bridge fdb* command. See EVB FDB entries on page 728 section for more information.

Configuration example:

```
# Create a MUX with DPDMUX_METHOD_C_VLAN_MAC forwarding method,
# configured as a VEB and with 2 downlinks and uplink connected
# to dpmac.1
$ ls-addmux -m=DPDMUX_METHOD_C_VLAN_MAC -d=2 dpmac.1

# Create a ni (dpni.1) and link it to evb0p0
$ ls-addni dpdmux.0.1

# Create a ni (dpni.2) and link it to evb0p1
$ ls-addni dpdmux.0.2

# Configure ni1
$ ip netns add ns1
$ ip link set ni1 netns ns1
$ ip netns exec ns1 ip link add link ni1 name ni1.6 type vlan id 6
$ ip netns exec ns1 ifconfig ni1.6 192.168.6.10
$ ip netns exec ns1 ip link set ni1 up
$ ip netns exec ns1 ip link set ni1 promisc on

# Configure ni2
$ ip netns add ns2
$ ip link set ni2 netns ns2
$ ip netns exec ns2 ip link add link ni2 name ni2.7 type vlan id 7
$ ip netns exec ns2 ifconfig ni2.7 192.168.7.12
$ ip netns exec ns2 ip link set ni2 up
$ ip netns exec ns2 ip link set ni2 promisc on

# For the downlinks interfaces, you would also need to add
# the downlinks MACs to fdb table
$ bridge fdb add 4a:64:0a:af:14:a2 dev evb0p0 vlan 6
$ bridge fdb add 62:9c:86:0f:f7:cf dev evb0p1 vlan 7

# Connectivity checkings [example for downlink - uplink ]
$ ip netns exec ns1 ping -I ni1.6 192.168.6.13 -c 1

# Check EVB FDB entries
$ bridge fdb show
```

8.3.2.6 Security Engine (SEC)

This section describes the software for the SEC hardware block that is part of the DPAA2 family of SoCs.

Introduction

Current chapter is focused on DPAA2-specific SEC details - Data Path SEC Interface (DPSECI) backend and frontend drivers. More information is provided in chapter Security Engine (SEC) on page 365, 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 v2.x, DPSECI backend can be used to submit crypto API service requests from the frontend drivers. The corresponding frontend compatible with DPSECI backend is *caamalg_qi2*, 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.

Module loading

The DPSECI backend driver (*dpseci*) is compiled built-in, while the DPSECI frontend driver (*dpaa2_caam*) is compiled, by default, as module (though it can also be compiled built-in). In this case, it has to be probed before dynamically creating `dpseci` objects with `restool`:

```
$ modprobe dpaa2_caam
```

Without any parameter, the `dpseci` object being created has 2 pairs of (rx,tx) queues.

```
$ restool dpseci create
$ restool dprc assign dprc.1 --object=dpseci.0 --plugged=1
```

To create 8 (maximum) number of queues:

```
$ restool dpseci create --num-queues=8 --priorities=1,2,3,4,5,6,7,8
$ restool dprc assign dprc.1 --object=dpseci.0 --plugged=1
```

More options can be displayed by using:

```
$ restool dpseci create --help
```

The list of algorithms registered by the *dpaa2_caam* driver is available in /proc filesystem:

```
$ grep caam-qi2 /proc/crypto
```

Enabling congestion management

Congestion management can be enabled when working with a MC that has a DPSECI object version greater or equal to 5.1. The first MC firmware version that supports the congestion management feature is 10.2. Enabling congestion management is done when creating the DPSECI object:

```
$ restool dpseci create --num-queues=8 --priorities=1,2,3,4,5,6,7,8 --
options="DPSECI_OPT_HAS_CG"
$ restool dprc assign dprc.1 --object=dpseci.0 --plugged=1
```

Source files

The driver source files are maintained in the Linux kernel source tree: `drivers/crypto/caam.`

How to test the driver

To test the driver, in the kernel configuration menu, under `"Cryptographic API -> Cryptographic algorithm manager"`, ensure that run-time self-tests are not disabled, i.e. the "`Disable run-time self tests`" entry is not set (CONFIG_CRYPTO_MANAGER_DISABLE_TESTS=n). This will run standard test vectors against the driver after the driver registers its supported algorithms with the kernel crypto API. To verify if the 'selftest' fields have 'passed', the /proc/crypto entries should be checked. An entry such as this:

```
name         : cbc(aes)
driver       : cbc-aes-caam-qi2
module       : kernel
priority     : 2000
refcnt       : 1
selftest     : passed
internal     : no
type         : givcipher
async        : yes
blocksize    : 16
```

```
min keysize  : 16
max keysize  : 32
```

means the driver has successfully registered support for the algorithm with the kernel crypto API. Note that although a test vector may not exist for a particular algorithm supported by the driver, the kernel will emit messages saying which algorithms weren't tested, and mark them as *passed* anyway. The driver's capabilities can also be tested with tcrypt testing framework available in linux kernel by selecting "`Cryptographic API -> Testing module`" (also `Disable run-time self tests` should be unchecked). A kernel module will be generated: *crypto/tcrypt.ko*. This has to be copied on the target. Then on target, after a dpseci object is registered:

```
$ insmod tcrypt.ko mode=10
```

Other ways to test with tcrypt:

- functional testing:
  - mode=3, 4, 35, 150, 152, 155, 181-191;
  - alg="algorithm_name" (see Table 84. Algorithms supported by each interface / backend on page 372 for choosing values for algorithm name)
- speed (*sec* - seconds parameter is optional):
  - mode=500 [sec=1] - xxx(aes) acipher_speed
  - mode=501 [sec=1] - xxx(3des) acipher_speed
  - mode=502 [sec=1] - xxx(des) acipher_speed etc.

There is no need to rmmod, tcrypt does not stay "resident", it exits after running the tests. That's why you'll see:

```
insmod: ERROR: could not insert module tcrypt.ko: Resource temporarily
```

For algorithms not supported, errors like below will be shown:

```
[ 2650.067737] failed to load transform for rmd128: -2
[ 2650.076480] failed to load transform for rmd160: -2
[ 2650.085099] failed to load transform for rmd256: -2
[ 2650.093739] failed to load transform for rmd320: -2
```

These are expected. Algorithm names registered by dpaa2_caam frontend driver are ending in `"-caam-qi2"`.

To verify the operation and correctness of the driver, other than noting the performance advantages due to the crypto offload, one can also ensure the h/w is doing the crypto by looking for driver messages in dmesg. The driver emits console messages at initialization time:

```
$ dmesg | grep dpaa2_caam
[ 1172.598591] dpaa2_caam dpseci.0: Opened dpseci object successfully
[ 1172.619979] dpaa2_caam dpseci.0: prio 0: rx queue 135, tx queue 119
[ 1172.626633] dpaa2_caam dpseci.0: prio 1: rx queue 136, tx queue 128
[ 1172.633278] dpaa2_caam dpseci.0: prio 2: rx queue 137, tx queue 129
[ 1172.639915] dpaa2_caam dpseci.0: prio 3: rx queue 138, tx queue 130
[ 1172.646555] dpaa2_caam dpseci.0: prio 4: rx queue 139, tx queue 131
[ 1172.653195] dpaa2_caam dpseci.0: prio 5: rx queue 140, tx queue 132
[ 1172.659831] dpaa2_caam dpseci.0: prio 6: rx queue 141, tx queue 133
[ 1172.666470] dpaa2_caam dpseci.0: prio 7: rx queue 142, tx queue 134
[ 1172.694319] dpaa2_caam dpseci.0: DPSECI version 3.0
[ 1172.700617] dpaa2_caam dpseci.0: algorithms registered in /proc/crypto
```

Given a time period when crypto requests are being made, the SEC h/w will fire completion notification interrupts:

```
$ cat /proc/interrupts | grep DPIO
```

If the number of interrupts fired increment, then the h/w is being used to do the crypto. If the numbers do not increment, then check if the algorithm being exercised is supported by the driver.

Running OpenSSL

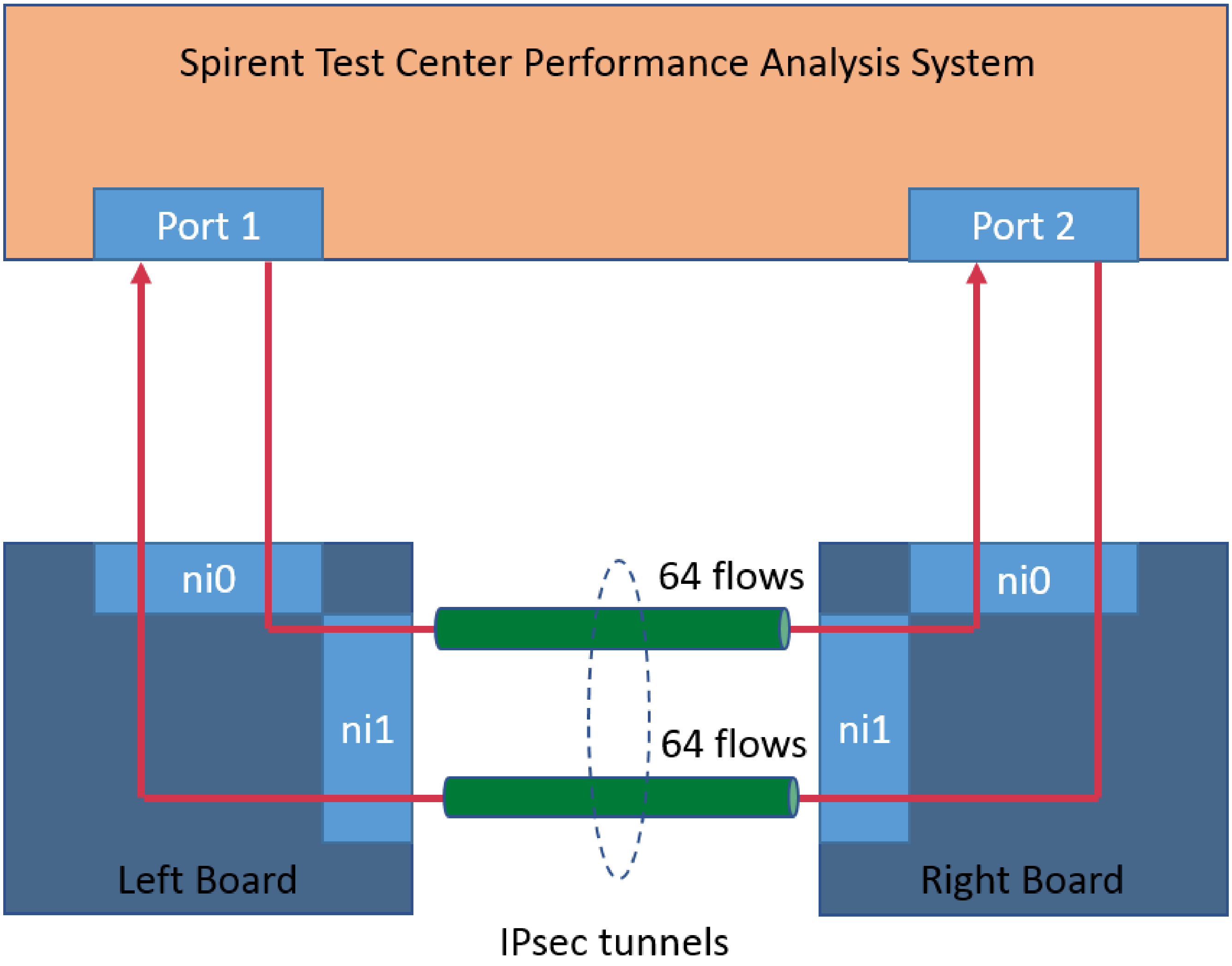
Some of the OpenSSL cryptographic operations (for e.g. TLS 1.0 record layer encryption, some non-protocol-specific crypto algorithms) can be offloaded to Linux kernel (and then further to SEC crypto engine) via cryptodev module.

Please refer to Hardware Offloading with OpenSSL on page 861 chapter for more details.

>**Running IPsec**

Setup Description

IPsec can be configured and used for NXP boards taking advantage of the cryptographic acceleration provided by the CAAM engine. Below is the description of the setup used to test IPsec traffic between two LS2088ARDB boards.



Traffic is generated from the Test Center on Port 1 as 64 flows. A flow is defined as a stream of packets that has a unique pair of values for IP source and IP destination. In our configuration the IP source ranges from 192.85.1.2 to 192.85.1.9 and the IP destination ranges from 192.86.1.2 to 192.86.1.9. The flows are received on the network interface ni0 of the left board, encapsulated and then sent over the ni1 network interface to the right board. Here the flows are decapsulated and routed to the network interface ni0 towards the Port 2 of the Test Center. A similar traffic, of 64 flows, is sent from Port 2 to Port 1 of the Test Center.

Board Bootup

Each LS2088ARDB board must be setup and configured properly. For more information about the booting process please refer to NXP SoC Booting Principles on page 135. For more details on the specifics of LS2088ARDB board boot please refer to LS2088ARDB on page 126. This paragraph only provides some custom values and configuration files used for booting the LS2088ARDB board while testing IPSec. Although using the default configuration may work we strongly encourage using the values/configuration files chosen below.

- While in the U-Boot prompt make sure that the variable "mcmemsize" is set to 0x80000000. This will ensure that enough memory was allocated to MC. See DPAA2-specific Environment Variables on page 129 for more details.

- Use the files "dpl.dts" and "dpc.dts" provided under the title "Useful Resources". They provide a minimum viable MC configuration that will enable IPSec testing. Both are in the .dts file format. To use them to configure the MC you need to compile them using the "dtc" compiler to obtain the "dtb" files:

  ```
  $ dtc -O dtb -I dts -o dpc.dtb dpc.dts
  $ dtc -O dtb -I dts -o dpl.dtb dpl.dts
  ```

  For more information about MC resource files refer to chapter Key Release Files: RCW, DPC and DPL on page 697.

Linux setup

When the Linux console prompt is presented to the user (after inserting the username and password) the following actions must be taken:

- Create DPSECI object and assign them to a DPRC. Make sure to enable congestion management for the DPSECI object.

  ```
  $ restool dpseci create --num-queues=8 --priorities=1,2,3,4,5,6,7,8 --
  options="DPSECI_OPT_HAS_CG"
  $ restool dprc assign dprc.1 --object=dpseci.0 --plugged=1
  ```

- Create the IPSec tunnels for each board (left/right) using the script "iproute_128tunnels.sh". The script takes as parameter the board position (left/write) and uses it to configure each board accordingly. The script can be found in the "Useful Resources" section. You can create your own copy on the board by copying and pasting the content to a local script file, preserving the name.

  To create the tunnels on the left board run:

  ```
  $ ./iproute_128tunnels.sh left
  ```

  To create the tunnels on the right board run:

  ```
  $ ./iproute_128tunnels.sh right
  ```

- Disable flow control on board for both ni0 and ni1.

  ```
  $ ethtool -A ni0 rx off
  $ ethtool -A ni0 tx off
  $ ethtool -A ni1 rx off
  $ ethtool -A ni1 tx off
  ```

> **NOTE**
>
> The flow control must be either on or off but must match the settings of the Test Center. The situation where the Test Control and the boards don't match causes resource in the boards to be oversubscribed which in turn will lead to memory corruption.

Running the Test

After Spirent Test Center application is configured and the testing Ethernet interfaces are connected to the traffic generator, start to generate traffic to measure IPv4 SEC & Forward throughput.

Useful resources

- **The "iproute_128tunnels.sh" script**

  ```
  #!/bin/bash
  eth0=ni0
  eth1=ni1

  make_esp_tunnel() {

  echo "add $1 $2 esp 0x$3 -m tunnel
      -E $4  0x7aeaca3f87d060a12f4a4487d5a5c3355920fae69a96c831
      -A hmac-sha1 0xe9c43acd5e8d779b6e09c87347852708ab49bdd3;" | setkey -c

  echo "add $2 $1 esp 0x`expr $3 + 100` -m tunnel
      -E $4  0xf6ddb555acfd9d77b03ea3843f2653255afe8eb5573965df
      -A hmac-sha1 0xea6856479330dc9c17b8f6c37e2a895363d83f21;" | setkey -c

  }
  ```

```
make_esp_policy() {

if [ $1 == left ]
then
    dir1=out
    dir2=in
    echo "spdadd $2 $3 any -P $dir1 ipsec
           esp/tunnel/$4-$5/require;" | setkey -c
    echo "spdadd $3 $2 any -P $dir2 ipsec
           esp/tunnel/$5-$4/require;" | setkey -c
else
    dir1=in
    dir2=out
    echo "spdadd $2 $3 any -P $dir1 ipsec
           esp/tunnel/$4-$5/require;" | setkey -c
    echo "spdadd $3 $2 any -P $dir2 ipsec
           esp/tunnel/$5-$4/require;" | setkey -c
fi

}

# Flush the SAD and SPD
setkey -F
setkey -FP

# set ip address
left_addr_ip=192.85.1.1
right_addr_ip=192.86.1.1
left_src_mac=00:10:94:00:00:01
right_src_mac=00:10:94:00:00:02
proto="aes-cbc"
base1=200
base2=200
echo 1 > /proc/sys/net/ipv4/ip_forward

case $1 in
        left)
                ifconfig $eth0 $left_addr_ip
                i=2
                for((j=2;j<10;j++))
                do
                    arp -s 192.85.1.$j $left_src_mac -i $eth0
                    for((k=2;k<10;k++))
                    do
                    if [ $base2 == 256 ]
                    then
                        base2=`expr $base2 - 256`
                        base1=`expr $base1 + 1`
                    fi
                    ip addr add 200.$base1.$base2.10/24 dev $eth1
                        make_esp_policy $1 192.85.1.$j 192.86.1.$k 200.$base1.$base2.10
200.$base1.$base2.20
                        make_esp_tunnel 200.$base1.$base2.10 200.$base1.$base2.20 `expr
200 + $i` $proto
                        ((base2++))
                        ((i++))
                    done
                done
            ;;
        right)
                ifconfig $eth0 $right_addr_ip
                i=2
                for((j=2;j<10;j++))
                do
                    arp -s 192.86.1.$j $right_src_mac -i $eth0
                    for((k=2;k<10;k++))
                    do
                    if [ $base2 == 256 ]
                    then
```

```
                        base2=`expr $base2 - 256`
                        base1=`expr $base1 + 1`
                    fi
                    ip addr add 200.$base1.$base2.20/24 dev $eth1
                        make_esp_policy $1 192.85.1.$j 192.86.1.$k 200.$base1.$base2.10 
200.$base1.$base2.20
                        make_esp_tunnel 200.$base1.$base2.10 200.$base1.$base2.20 `expr 
200 + $i` $proto
                        ((base2++))
                        ((i++))
                    done
                done
            ;;
esac

ifconfig $eth1 up
route add default dev $eth1
```

- **The "dpc.dts" configuration file**

```
/dts-v1/;

/ {

    mc_general {

        log {
            mode = "LOG_MODE_ON";
            level = "LOG_LEVEL_WARNING";
        };

        console {
            mode = "CONSOLE_MODE_ON";
            uart_id = <3>;
        };
    };

    resources {

        icid_pools {

            icid_pool@1 {
                num = <0x64>;
                base_icid = <0x0>;
            };
        };
    };

    controllers {

        qbman {
            total_bman_buffers = <0xe0000>;
            wq_ch_conversion = <32>;
        };
    };

    board_info {

        ports {
        };
    };
};
```

- **The "dpl.dts" configuration file**

```
/dts-v1/;
/ {
    dpl-version = <10>;
    /*****************************************************************
         * Containers
         *****************************************************************/
    containers {
    dprc@1 {
        parent = "none";
        options = "DPRC_CFG_OPT_SPAWN_ALLOWED" , "DPRC_CFG_OPT_ALLOC_ALLOWED",
"DPRC_CFG_OPT_IRQ_CFG_ALLOWED";
        objects {
        /* ------------- MACs --------------*/
        obj_set@dpmac {
            type = "dpmac";
            ids = <1 2 3 4 5 6 7 8>;
        };

        /* ------------ DPNIs --------------*/
        obj_set@dpni {
            type = "dpni";
            ids = <0 1>;
        };

        /* ------------ DPBPs --------------*/
        obj_set@dpbp {
            type = "dpbp";
            ids = <0 1>;
        };

        /* ------------ DPIOs --------------*/
        obj_set@dpio {
            type = "dpio";
            ids = <0 1 2 3 4 5 6 7>;
        };

        /* ------------ DPMCPs --------------*/
        obj_set@dpmcp {
            type = "dpmcp";
            ids = <1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16>;
        };

        /* ------------ DPCON --------------*/
        obj_set@dpcon {
            type = "dpcon";
            ids = <0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15>;
        };
        };
    };
    };

    /*****************************************************************
     * Objects
     *****************************************************************/
    objects {

    /* ------------ DPNI --------------*/
    dpni@0 {
        type = "DPNI_TYPE_NIC";
        options = "";
        num_queues = <8>;
        num_tcs = <1>;
        mac_filter_entries = <16>;
        vlan_filter_entries = <0>;
        fs_entries = <0>;
        qos_entries = <0>;
    };

    dpni@1 {
        type = "DPNI_TYPE_NIC";
        options = "";
        num_queues = <8>;
        num_tcs = <1>;
        mac_filter_entries = <16>;
        vlan_filter_entries = <0>;
        fs_entries = <0>;
        qos_entries = <0>;
    };

    /* ------------ DPBP --------------*/
    dpbp@0 {
    };

    dpbp@1 {
    };

    /* ------------ DPIO --------------*/
    dpio@0 {
        channel_mode = "DPIO_LOCAL_CHANNEL";
        num_priorities = <8>;
    };
    dpio@1 {
        channel_mode = "DPIO_LOCAL_CHANNEL";
        num_priorities = <8>;
    };
    dpio@2 {
        channel_mode = "DPIO_LOCAL_CHANNEL";
        num_priorities = <8>;
    };
    dpio@3 {
        channel_mode = "DPIO_LOCAL_CHANNEL";
        num_priorities = <8>;
    };
    dpio@4 {
        channel_mode = "DPIO_LOCAL_CHANNEL";
        num_priorities = <8>;
    };
    dpio@5 {
        channel_mode = "DPIO_LOCAL_CHANNEL";
        num_priorities = <8>;
    };
    dpio@6 {
        channel_mode = "DPIO_LOCAL_CHANNEL";
        num_priorities = <8>;
    };
    dpio@7 {
        channel_mode = "DPIO_LOCAL_CHANNEL";
        num_priorities = <8>;
    };

    /* ------------ DPMAC --------------*/

    dpmac@1 {
    };
    dpmac@2 {
    };
    dpmac@3 {
    };
    dpmac@4 {
    };
    dpmac@5 {
    };
    dpmac@6 {
    };
    dpmac@7 {
    };
    dpmac@8 {
    };

    /* ------------ DPMCP --------------*/
    dpmcp@1 {
    };
    dpmcp@2 {
    };
    dpmcp@3 {
    };
    dpmcp@4 {
    };
    dpmcp@5 {
    };
    dpmcp@6 {
    };
    dpmcp@7 {
    };
    dpmcp@8 {
    };
    dpmcp@9 {
    };
    dpmcp@10 {
    };
    dpmcp@11 {
    };
    dpmcp@12 {
    };
    dpmcp@13 {
    };
    dpmcp@14 {
    };
    dpmcp@15 {
    };
    dpmcp@16 {
    };

    /* ------------ DPCON --------------*/
    dpcon@0 {
        num_priorities=<2>;
    };

     dpcon@1 {
        num_priorities=<2>;
    };

    dpcon@2 {
        num_priorities=<2>;
    };

     dpcon@3 {
        num_priorities=<2>;
    };

    dpcon@4 {
        num_priorities=<2>;
    };

     dpcon@5 {
        num_priorities=<2>;
    };

    dpcon@6 {
        num_priorities=<2>;
    };

     dpcon@7 {
        num_priorities=<2>;
    };

    dpcon@8 {
        num_priorities=<2>;
    };

     dpcon@9 {
        num_priorities=<2>;
```

```
    };

    dpcon@10 {
        num_priorities=<2>;
    };

     dpcon@11 {
        num_priorities=<2>;
    };

    dpcon@12 {
        num_priorities=<2>;
    };

     dpcon@13 {
        num_priorities=<2>;
    };

    dpcon@14 {
        num_priorities=<2>;
    };

     dpcon@15 {
        num_priorities=<2>;
    };
   };

    /*****************************************************************
     * Connections
     *****************************************************************/
    connections {
    connection@0{
        /* First copper port (ETH0 on the RDB chassis) */
        endpoint1 = "dpni@0";
        endpoint2 = "dpmac@1";
    };
    connection@1{
        /* Second copper port (ETH1 on the RDB chassis) */
        endpoint1 = "dpni@1";
        endpoint2 = "dpmac@2";
    };
    };
};
```

Supporting Documentation

General SEC information, Job Ring Interface (JRI)

DPAA1-specific SEC details - Queue Interface (QI)

8.3.2.7 Decompression Compression Engine (DCE)

Introduction

This section describes the software interface to DCE (Decompression Compression Engine) accelerator available on the LS2088A SoC.The interface is designed to simplify interaction with DCE as much as possible without loss of flexibility and acceleration offered by DCE hardware.

Hardware Overview

This section gives an overview of the operation of the DCE hardware to provide fundamentals for software developers using the driver API. More detailed information is available in the hardware reference manual.

The DCE is a hardware accelerator that is part of the Datapath Acceleration Architecture version 2 (DPAA2). The DCE is one of the DPAA2 accelerators that include others like security and pattern matching engines. These accelerators are connected using a queue manager (QMan) and buffer manager (BMan) that allows data to be exchanged between software and accelerators.

Software enqueues data to a TX frame queue leading to DCE. DCE receives the data and processes it. It then enqueues a response on an RX frame queue leading back to software. The software then provides a DCE object that abstracts the details of frame queue configuration and usage, as well as other hardware details. This object is called DPDCEI.

Example Application

The LSDK contains an example application called dce-api-perf-test.c This test can be run in multiple modes that simulate many of the interesting DCE use cases. The README in the dce directory has instructions on how to run the test.

Software Details

User-space Interface

DCE Driver

Provides a set of user-space APIs that simplifies access to DCE. The driver provides accessor objects called sessions which maintain state in coordination with the hardware. Users can pull the responses from DCE . This DCE driver is meant to be entirely sufficient for a user to use DCE without having to read the related HW documents. It is documented in the dce repository included in `<PATH_TO_LSDK>/packages/apps/dce/dce.h`

Linux

There is no DCE driver in the Kernel.

Functionality

Configuration

The *DCE* configuration and setup is documented in the article DPDCEI Commands.

Build Procedure

The procedure is a standard LSDK build.

Test Procedure

Refer to https://source.codeaurora.org/external/qoriq/qoriq-components/dce/tree/README?h=github.qoriq-os/integration for detailed descriptions of sample DCE test procedure.

8.3.3 DPAA2 Standard Linux Documentation

Following is a summary of relevant documentation from standard Linux sources and formats. It provides links to these documents, provides a snapshot of the document, or both.

8.3.3.1 Kernel Documentation Directory

The Linux kernel source code contains a documentation directory, and there is some information there that is relevant to DPAA2. It is possible to see the upstream versions of these documents by going to https://www.kernel.org/ and browsing the Linux source code trees.

- Kernel Management Complex (MC) bus driver: This document is in-flight to kernel.org so a copy is provided below rather than a link to kernel.org.

```
Copyright (C) 2016 Freescale Semiconductor Inc.

DPAA2 (Data Path Acceleration Architecture Gen2)
------------------------------------------------
```

This document provides an overview of the Freescale DPAA2 architecture and how it is integrated into the Linux kernel.

```
Contents summary
   -DPAA2 overview
   -Overview of DPAA2 objects
   -DPAA2 Linux driver architecture overview
        -bus driver
        -dprc driver
        -allocator
        -dpio driver
        -Ethernet
        -mac
```

DPAA2 Overview

DPAA2 is a hardware architecture designed for high-speeed network packet processing. DPAA2 consists of sophisticated mechanisms for processing Ethernet packets, queue management, buffer management, autonomous L2 switching, virtual Ethernet bridging, and accelerator (e.g. crypto) sharing.

A DPAA2 hardware component called the Management Complex (or MC) manages the DPAA2 hardware resources. The MC provides an object-based abstraction for software drivers to use the DPAA2 hardware.

The MC uses DPAA2 hardware resources such as queues, buffer pools, and network ports to create functional objects/devices such as network interfaces, an L2 switch, or accelerator instances.

The MC provides memory-mapped I/O command interfaces (MC portals) which DPAA2 software drivers use to operate on DPAA2 objects:

```
+--------------------------------------+
|                  OS                  |
|                        DPAA2 drivers |
|                             |        |
+-----------------------------|--------+
                              |
                              | (create,discover,connect
                              |  config,use,destroy)
                              |
                DPAA2         |
+------------------------| mc portal |-+
|                             |        |
|   +- - - - - - - - - - - - -V- - -+  |
|   |                               |  |
|   |   Management Complex (MC)     |  |
|   |                               |  |
|   +- - - - - - - - - - - - - - - -+  |
|                                      |
| Hardware                  Hardware   |
| Resources                 Objects    |
| ---------                 -------    |
| -queues                   -DPRC      |
| -buffer pools             -DPMCP     |
| -Eth MACs/ports           -DPIO      |
| -network interface        -DPNI      |
|  profiles                 -DPMAC     |
| -queue portals            -DPBP      |
| -MC portals                ...       |
|  ...                                 |
|                                      |
+--------------------------------------+
```

The MC mediates operations such as create, discover, connect, configuration, and destroy. Fast-path operations on data, such as packet transmit/receive, are not mediated by the MC and are done directly using memory mapped regions in DPIO objects.

Overview of DPAA2 Objects

The section provides a brief overview of some key objects in the DPAA2 hardware. A simple scenario is described illustrating the objects involved in creating a network interfaces.

-DPRC (Datapath Resource Container)

A DPRC is an container object that holds all the other types of DPAA2 objects. In the example diagram below there are 8 objects of 5 types (DPMCP, DPIO, DPBP, DPNI, and DPMAC) in the container.

```
+---------------------------------------------------------+
| DPRC                                                    |
|                                                         |
|  +-------+  +-------+  +-------+  +-------+  +-------+  |
|  | DPMCP |  | DPIO  |  | DPBP  |  | DPNI  |  | DPMAC |  |
|  +-------+  +-------+  +-------+  +---+---+  +---+---+  |
|  | DPMCP |  | DPIO  |                                   |
|  +-------+  +-------+                                   |
|  | DPMCP |                                              |
|  +-------+                                              |
|                                                         |
+---------------------------------------------------------+
```

From the point of view of an OS, a DPRC is bus-like. Like a plug-and-play bus, such as PCI, DPRC commands can be used to enumerate the contents of the DPRC, discover the hardware objects present (including mappable regions and interrupts).

```
 dprc.1 (bus)
   |
   +--+--------+-------+-------+-------+
      |        |       |       |       |
    dpmcp.1  dpio.1  dpbp.1  dpni.1  dpmac.1
    dpmcp.2  dpio.2
    dpmcp.3
```

Hardware objects can be created and destroyed dynamically, providing the ability to hot plug/unplug objects in and out of the DPRC.

A DPRC has a mappable mmio region (an MC portal) that can be used to send MC commands. It has an interrupt for status events (like hotplug).

All objects in a container share the same hardware "isolation context". This means that with respect to an IOMMU the isolation granularity is at the DPRC (container) level, not at the individual object level.

DPRCs can be defined statically and populated with objects via a config file passed to the MC when firmware starts it. There is also a Linux user space tool called "restool" that can be used to create/destroy containers and objects dynamically.

-DPAA2 Objects for an Ethernet Network Interface

A typical Ethernet NIC is monolithic-- the NIC device contains TX/RX queuing mechanisms, configuration mechanisms, buffer management, physical ports, and interrupts. DPAA2 uses a more granular approach utilizing multiple hardware objects. Each object has specialized functions, and are used together by software to provide Ethernet network interface functionality. This approach provides efficient use of finite

```
hardware resources, flexibility, and performance advantages.

The diagram below shows the objects needed for a simple
network interface configuration on a system with 2 CPUs.



          +---+---+ +---+---+
             CPU0      CPU1
          +---+---+ +---+---+
              |         |
          +---+---+ +---+---+
             DPIO      DPIO
          +---+---+ +---+---+
                \     /
                 \   /
                  \ /
               +---+---+
                  DPNI  --- DPBP,DPMCP
               +---+---+
                   |
                   |
               +---+---+
                 DPMAC
               +---+---+
                   |
                port/PHY

Below the objects are described.  For each object a brief description
is provided along with a summary of the kinds of operations the object
supports and a summary of key resources of the object (mmio regions
and irqs).

   -DPMAC (Datapath Ethernet MAC): represents an Ethernet MAC, a
    hardware device that connects to an Ethernet PHY and allows
    physical transmission and reception of Ethernet frames.
       -mmio regions: none
       -irqs: dpni link change
       -commands: set link up/down, link config, get stats,
         irq config, enable, reset

   -DPNI (Datapath Network Interface): contains TX/RX queues,
    network interface configuration, and rx buffer pool configuration
    mechanisms.
       -mmio regions: none
       -irqs: link state
       -commands: port config, offload config, queue config,
        parse/classify config, irq config, enable, reset

   -DPIO (Datapath I/O): provides interfaces to enqueue and dequeue
    packets and do hardware buffer pool management operations.  For
    optimum performance there is typically one DPIO per CPU.  This allows
    each CPU to perform simultaneous enqueue/dequeue operations.
       -mmio regions: queue operations, buffer mgmt
       -irqs: data availability, congestion notification, buffer
              pool depletion
       -commands: irq config, enable, reset

   -DPBP (Datapath Buffer Pool): represents a hardware buffer
    pool.
       -mmio regions: none
       -irqs: none
       -commands: enable, reset

   -DPMCP (Datapath MC Portal): provides an MC command portal.
    Used by drivers to send commands to the MC to manage
    objects.
       -mmio regions: MC command portal
       -irqs: command completion
```

```
              -commands: irq config, enable, reset

        Object Connections
        ------------------
        Some objects have explicit relationships that must
        be configured:

           -DPNI <--> DPMAC
           -DPNI <--> DPNI
           -DPNI <--> L2-switch-port
              A DPNI must be connected to something such as a DPMAC,
              another DPNI, or L2 switch port.  The DPNI connection
              is made via a DPRC command.

                  +-------+  +-------+
                  | DPNI  |  | DPMAC |
                  +---+---+  +---+---+
                      |          |
                      +==========+

           -DPNI <--> DPBP
              A network interface requires a 'buffer pool' (DPBP
              object) which provides a list of pointers to memory
              where received Ethernet data is to be copied.  The
              Ethernet driver configures the DPBPs associated with
              the network interface.

        Interrupts
        ----------
        All interrupts generated by DPAA2 objects are message
        interrupts.  At the hardware level message interrupts
        generated by devices will normally have 3 components--
        1) a non-spoofable 'device-id' expressed on the hardware
        bus, 2) an address, 3) a data value.

        In the case of DPAA2 devices/objects, all objects in the
        same container/DPRC share the same 'device-id'.
        For ARM-based SoC this is the same as the stream ID.


    DPAA2 Linux Driver Overview
    ---------------------------

    This section provides an overview of the Linux kernel drivers for
    DPAA2-- 1) the bus driver and associated "DPAA2 infrastructure"
    drivers and 2) functional object drivers (such as Ethernet).

    As described previously, a DPRC is a container that holds the other
    types of DPAA2 objects.  It is functionally similar to a plug-and-play
    bus controller.

    Each object in the DPRC is a Linux "device" and is bound to a driver.
    The diagram below shows the Linux drivers involved in a networking
    scenario and the objects bound to each driver.  A brief description
    of each driver follows.



                                             +------------+
                                             | OS Network |
                                             |   Stack    |
                     +------------+          +------------+
                     | Allocator  |. . . . . . | Ethernet   |
                     |(dpmcp,dpbp)|          |   (dpni)   |
                     +-.----------+          +---+---+----+
                      .          .                ^   |
                     .            .     <data avail, |   |<enqueue,
                    .              .     tx confirm> |   | dequeue>
        +-------------+             .                |   |
    | DPRC driver |              .           +---+---V----+     +---------+
    |   (dprc)    |               . . . . . .| DPIO driver|     |   MAC   |
    +----------+--+                          |  (dpio)    |     | (dpmac) |
               |                             +------+-----+     +-----+---+
               |<dev add/remove>                    |                 |
               |                                    |                 |
          +----+--------------+                     |              +--+---+
          |   mc-bus driver   |                     |              | PHY  |
          |                   |                     |              |driver|
          | /fsl-mc@80c000000 |                     |              +--+---+
          +-------------------+                     |                 |
                                                    |                 |
 ================================ HARDWARE =========|=================|======
                                                  DPIO                |
                                                    |                 |
                                                  DPNI---DPBP         |
                                                    |                 |
                                                  DPMAC               |
                                                    |                 |
                                                   PHY ---------------+
 ===================================================|========================

A brief description of each driver is provided below.

    mc-bus driver
    -------------
    The mc-bus driver is a platform driver and is probed from an
    "/fsl-mc@xxxx" node in the device tree passed in by boot firmware.
    It is responsible for bootstrapping the DPAA2 kernel infrastructure.
    Key functions include:
       -registering a new bus type named "fsl-mc" with the kernel,
        and implementing bus call-backs (e.g. match/uevent/dev_groups)
       -implemeting APIs for DPAA2 driver registration and for device
        add/remove
       -creates an MSI irq domain
       -do a device add of the 'root' DPRC device, which is needed
        to bootstrap things

    DPRC driver
    -----------
    The dprc-driver is bound to DPRC objects and does runtime management
    of a bus instance.  It performs the initial bus scan of the DPRC
    and handles interrupts for container events such as hot plug.

    Allocator
    ----------
    Certain objects such as DPMCP and DPBP are generic and fungible,
    and are intended to be used by other drivers.  For example,
    the DPAA2 Ethernet driver needs:
       -DPMCPs to send MC commands, to configure network interfaces
       -DPBPs for network buffer pools

    The allocator driver registers for these allocatable object types
    and those objects are bound to the allocator when the bus is probed.
    The allocator maintains a pool of objects that are available for
    allocation by other DPAA2 drivers.

    DPIO driver
    -----------
    The DPIO driver is bound to DPIO objects and provides services that allow
    other drivers such as the Ethernet driver to receive and transmit data.
    Key services include:
        -data availability notifications
        -hardware queuing operations (enqueue and dequeue of data)
        -hardware buffer pool management

    There is typically one DPIO object per physical CPU for optimum
    performance, allowing each CPU to simultaneously enqueue
    and dequeue data.
```

```
The DPIO driver operates on behalf of all DPAA2 drivers
active in the kernel--  Ethernet, crypto, compression,
etc.

Ethernet
--------
The Ethernet driver is bound to a DPNI and implements the kernel
interfaces needed to connect the DPAA2 network interface to
the network stack.

Each DPNI corresponds to a Linux network interface.

MAC driver
----------
An Ethernet PHY is an off-chip, board specific component and is managed
by the appropriate PHY driver via an mdio bus.  The MAC driver
plays a role of being a proxy between the PHY driver and the
MC.  It does this proxy via the MC commands to a DPMAC object.
```

8.3.3.2 DPAA2 Resource Management Tool (restool) User Manual

Restool is a Linux user space program that allows DPAA2 objects to be created, destroyed, and manipulated. Its primary documentation is in the style of a Linux man page.

The Management Complex architecture uses a hardware object called a “container” (or DPRC) to hold I/O resources and hardware objects for use by GPP software contexts.

DPRCs can be created and populated in two different ways:

- at MC initialization during system boot in a configuration file called a “DPL file”
- dynamically at runtime

This document describes how restool can be used to do dynamic management of MC resources in the context of Linux. Key resource management operations include:

- listing containers and their contents
- creating/destroying containers
- creating/destroying new MC objects
- move object between parent container and child container
- establishing connections between MC objects

The version of restool -restool v1.4 - included in this release is compatible will all MC firmware versions and will export different options based on the firmware found on the board. In the following pages it will be described the available options found running MC10.x on the board.

8.3.3.2.1 DPRC commands

8.3.3.2.1.1 list command

The **list** command lists all containers in the system.

SYNTAX:

restool dprc list

ARGUMENTS:

none

EXAMPLE:

List all the containers in the system

```
$ restool dprc list
  dprc.1
      dprc.2
      dprc.3
```

The container hierarchy (parent-child relationships) is shown by indentation.

8.3.3.2.1.2 show command

The **show** command displays the contents (objects and resources) of a DPRC/container.

SYNTAX:

restool dprc show <*container*>

restool dprc show <*container*> **--resources**

restool dprc show <*container*> **--resource-type**=<resource-type>

ARGUMENTS:

<*container*>

A string specifying the target dprc—e.g. “dprc.2”.

The container argument value “mc.global” is special and refers to the global

container of resource pools inside the Management Complex.

--resources

Display a container’s resource count for each resource (instead of displaying

objects/resources)

--resource-type <resource-type>

Specifies the type of resource to list. The resource-type argument is a string specifying

the resource name—e.g. “mcp”.

EXAMPLE :

Show all objects in dprc 2:

```
$ restool dprc show dprc.2
dprc.2 contains 6 objects:
object label plugged state
dpni.7 xyz  plugged
dpni.8 abc  plugged
dpio.2 plugged
dpio.3 unplugged
dpcon.9 plugged
dpbp.1 plugged
```

Show all resources in dprc 2:

```
$ restool dprc show dprc.2 --resources
bpid: 16
fqid: 100
channel: 4
qpr: 2
cgid: 2
```

Show dprc with no objects in it:

```
$ restool dprc show dprc.4
(empty)
```

Show all buffer pool IDs in dprc 2:

```
$ restool dprc show dprc.2 --resource-type=bp
bp.35 – bp.36
bp.50
bp.52 - bp.63
```

Show all MC portal IDs in the global MC container:

```
$ restool dprc show mc.global --resource-type=mcp
mcp.30 – mcp.250
```

8.3.3.2.1.3 info command

The **info** command displays detailed information about a specific container.

SYNTAX:

restool dprc info <*dprc-object*> [**--verbose**]

ARGUMENTS :

<*dprc-object*>

Specifies which container to show detailed info for. The object argument is a string

specifying the container name—e.g. “dprc.2”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dprc info dprc.2
container id: 2
icid: 2
portal id: 5
version: 0.0
dprc options: 0x3
    DPRC_CFG_OPT_SPAWN_ALLOWED
    DPRC_CFG_OPT_ALLOC_ALLOWED
object label: nadk’s dprc

$ restool dprc info dprc.2 --verbose
container id: 2
icid: 2
portal id: 5
version: 0.0
dprc options: 0x3
    DPRC_CFG_OPT_SPAWN_ALLOWED
    DPRC_CFG_OPT_ALLOC_ALLOWED
object label: nadk-usage-dprc
number of mappable regions: 1
number of interrupts: 1
interrupt 0's mask: 0
interrupt 0's status: 0x1
```

8.3.3.2.1.4 create command

The **create** command creates a new child DPRC under the specified parent. The name/id of the

object created is displayed to stdout.

SYNTAX:

restool dprc create <parent-container> [**--options**=<*options-mask*>] [**--label**=<*object’s-label*>]

OPTIONS :

<parent-container>

--options=<*options-mask*>

Where <options-mask> is a comma separated list of DPRC options:

```
            DPRC_CFG_OPT_SPAWN_ALLOWED
            DPRC_CFG_OPT_ALLOC_ALLOWED
            DPRC_CFG_OPT_OBJ_CREATE_ALLOWED
            DPRC_CFG_OPT_TOPOLOGY_CHANGES_ALLOWED
            DPRC_CFG_OPT_IOMMU_BYPASS
            DPRC_CFG_OPT_AIOP
            DPRC_CFG_OPT_IRQ_CFG_ALLOWED
```

--label=<*object's-label*>

Specify a label for the newly created object. It is kind of an alias for that object.

Length of the string is 15 characters maximum.

Say --label=”nadk’s dprc”

EXAMPLE:

Create a child DPRC under parent dprc.1 with default options:

```
    $ restool dprc create dprc.1
    dprc.9 is created under dprc.1
```

Create a child DPRC under parent dprc.1 with default options, with label “nadk’s dprc”:

```
    $ restool dprc create dprc.1 --label=”nadk’s dprc”
    dprc.11 is created under dprc.1
```

8.3.3.2.1.5 create command

The **create** command creates a new child DPRC under the specified parent. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dprc create <parent-container> [**--options**=<*options-mask*>] [**--label**=<*object's-label*>]

OPTIONS :

<parent-container>

--options=<*options-mask*>

Where <options-mask> is a comma separated list of DPRC options:

```
            DPRC_CFG_OPT_SPAWN_ALLOWED
            DPRC_CFG_OPT_ALLOC_ALLOWED
            DPRC_CFG_OPT_OBJ_CREATE_ALLOWED
            DPRC_CFG_OPT_TOPOLOGY_CHANGES_ALLOWED
            DPRC_CFG_OPT_IOMMU_BYPASS
            DPRC_CFG_OPT_AIOP
            DPRC_CFG_OPT_IRQ_CFG_ALLOWED
```

--label=<*object's-label*>

Specify a label for the newly created object. It is kind of an alias for that object.

Length of the string is 15 characters maximum.

Say --label=”nadk’s dprc”

EXAMPLE:

Create a child DPRC under parent dprc.1 with default options:

```
$ restool dprc create dprc.1
dprc.9 is created under dprc.1
```

Create a child DPRC under parent dprc.1 with default options, with label “nadk’s dprc”:

```
$ restool dprc create dprc.1 --label=”nadk’s dprc”
dprc.11 is created under dprc.1
```

8.3.3.2.1.6 destroy command

The **destroy** command destroys the specified DPRC.

SYNTAX:

restool dprc destroy <container> **--help**

OPTIONS:

<container>

--help

Displays help for the command.

EXAMPLE:

Destroy a specified DPRC, say dprc.2:

```
$ restool dprc destroy dprc.2
dprc.2 is destroyed
```

8.3.3.2.1.7 assign command

The **assign** command moves an object or resource from a parent container to a child container. Object (dpni, dpbp, etc) assignment is always explicit and the exact object id to be assigned must be specified. Resources (e.g. mcp, bp, fq, etc) are assigned by type and count.

SYNTAX:

restool dprc assign <parent-*container*> [**--child**=<child-*container*>] **--object**=<*object*> **--plugged**=<*state*>

This syntax changes the plugged state. The child-container must be the same as parent-container, or omit --target option. It is not possible to change the plugged state of a dprc.

restool dprc assign <parent-*container*> [**--child**=<child-*container*>] **--object**=<*object*>

This syntax moves one object from parent-container to another, so the target-container must be different from the parent-container. Limitation: cannot move dprc from one container to another.

restool dprc assign <parent-*container*> [**--child**=<child-*container*>] --resource-type=<*type*> **-- count**=<*number*>

This syntax moves a resource from parent-container to a child-container. If the childcontainer is the same as the parent-container, the resource will be taken from the parent of parent-container and will be assigned to the parent-container.

ARGUMENTS :

<container>

Specifies the parent container from which the object will be moved.

--object=<*object*>

Specifies the object to assign— value is a string specifying object name and ID
(e.g. dpni.5)

--child=<*child-container*>

Specifies the destination container for the operation. Valid values are any child container. (The target container may be the same as the parent container, allowing "assign to self")

--plugged=<*state*>

Specifies the plugged state of the object (valid values are 0 or 1)

--resource-type=<*type*>

String specifying the resource type to assign (e.g "mcp", "fq", "cg", etc). To see valid resources that may be assigned use the "dprc show <container> --resources" command.

--count=<*number*>

Number of resources to assign.

EXAMPLE:

Set the plugged state of dpni.5. Note source and destination containers are the same.

```
$ restool dprc assign dprc.1 --object=dpni.5 --child=dprc.1
--plugged=1
$ restool dprc assign dprc.1 --object=dpni.5 --plugged=1
```

Unset the plugged state of dpni.5. Note source and destination containers are the same.

```
$ restool dprc assign dprc.1 --object=dpni.5 --child=dprc.1
--plugged=0
$ restool dprc assign dprc.1 --object=dpni.5 --plugged=0
```

Move dpni.5 from dprc.1 (parent) to dprc.3 (child):

```
$ restool dprc assign dprc.1 --object=dpni.5 --child=dprc.3
```

Move 3 mcp resources from dprc.1 (parent) to dprc.2 (child):

```
$ restool dprc assign dprc.1 --resource-type=mcp --count=3
--child=dprc.2
```

8.3.3.2.1.8 unassign command

The **unassign** command moves an object or resource from a child container to a parent container

SYNTAX:

restool dprc unassign <*container*> **--object**=<*object*> [**--child**=<child-*container*>]

restool dprc unassign <*container*> **--resource-type**=<*type*> **--count** <*number*> [**--child**=<child-*container*>]

ARGUMENTS :

<*container*>

Specifies the container to which the object will be moved.

--object=<*object*>

Specifies the object to unassign— value is a string specifying object name and ID (e.g.

dpni.5)

--child=<child-*container*>

Specifies the container from which the object/resource will be moved from.

--plugged=<*plugged-state*>

Specifies the plugged state of the object (valid values are 0 or 1)

--resource-type=<*type*>

String specifying the resource type to assign (e.g “mcp”, “fq”, “cg”, etc)

--count=<*number*>

Number of resources to unassign.

EXAMPLE:

Unassign 3 mcp resources from dprc.2 (child) to dprc.1 (parent):

```
$ restool dprc unassign dprc.1 --resource-type=mcp --count=3
--child=dprc.2
```

Unassign dpni.5 from dprc.3 (child) to dprc.1 (parent):

```
$ restool dprc unassign dprc.1 --object=dpni.5 --child=dprc.3
```

8.3.3.2.1.9 set-quota command

The **set-quota** command sets quota policies for a child container, specifying the number of resources a child may take from its parent container. But remember a parent can assign any number of resource to its child if it wants to, and if it has enough resources to assign. So the quota is effective only when the child dprc does have enough resource and it wants to borrow resource from its parent. It could only “borrow” the quota number of resources from its parent.

SYNTAX:

restool dprc set-quota <*parent-container*> **--resource-type**=<*type*> **--count**=<*number*>

--child-container=<*container*>

ARGUMENTS :

<*parent-container*>

Specifies the parent container.

--resource-type=<*type*>

String specifying the resource type to set the quota for (e.g “mcp”, “fq”, “cg”, etc)

--count=<*number*>

Max number of resources the child is able to allocate.

--child-container=<*container*>

EXAMPLE:

Set a quota of 10 mcp resource that child container dprc.5 may take from parent dprc.1:

```
$ restool dprc set-quota dprc.1 --resource-type=mcp --count=10
   --child-container=dprc.5
```

8.3.3.2.1.10 set-label command

The **set-label** command sets label for any objects excluding dprc.1

SYNTAX:

restool dprc set-label <*object*> **--label**=<*label*>

ARGUMENTS :

<object>

Specifies the object to be set.

--label=*<label>*

String specifying the label, maximum length is 15 characters.

EXAMPLE:

Set label of dprc.4 to “mountain view”:

```
$ restool dprc set-label dprc.4 --label=”mountain view”
```

8.3.3.2.1.11 connect command

The **connect** command connects 2 objects, creating a link between them.

SYNTAX:

restool dprc connect *<container>* **--endpoint1**=*<object>* **--endpoint2**=*<object>*

ARGUMENTS :

<container>

A string specifying the target dprc—e.g. “dprc.2”.

--endpoint1=*<object>*

Specifies the first endpoint object.

--endpoint2=*<object>*

Specifies the second endpoint object.

EXAMPLE:

The connect command connects a network object such as a DPNI to a peer object such as a DPMAC or DPSW port.

Connect dpni.2 to dpmac.5:

```
$ restool dprc connect dprc.2 --endpoint1=dpni.2 --endpoint2=dpmac.5
```

Connect dpni.2 to dpsw.1 interface 7:

```
$ restool dprc connect dprc.2 --endpoint1=dpni.2 --endpoint2=dpsw.1.7
```

8.3.3.2.1.12 disconnect command

The **disconnect** command removes the link between two objects. Either endpoint can be

specified as the target of the operation.

SYNTAX:

restool dprc disconnect *<container>* **--endpoint**=*<object>*

ARGUMENTS:

<container>

A string specifying the target dprc—e.g. “dprc.2”.

--endpoint=*<object>*

Specifies the first endpoint object.

EXAMPLE:

Remove the link between dpni.2 and dpmac.5

```
    $ restool dprc disconnect dprc.2 -endpoint=dpni.2
```

8.3.3.2.1.13 generate-dpl command

The **generate-dpl** command prints to the standard output a DPL syntax file describing the specified container

SYNTAX:

restool dprc generate-dpl <*container*>

ARGUMENTS:

<*container*>

A string specifying the target dprc—e.g. “dprc.2”.

EXAMPLE:

```
Generate a DPL for dprc.1
    $ restool dprc generate-dpl dprc.1
```

8.3.3.2.2 DPNI Commands

8.3.3.2.2.1 help command

The **help** command displays usage information for the DPNI object

SYNTAX:

restool dpni help

ARGUMENTS:

none

EXAMPLE:

```
    $ restool dpni help
    usage: restool dpni <command> [--help][ARGS…]
    Where <command> can be:
        info - displays detailed information about a DPNI object.
        create - creates a child DPNI under the root DPRC
        destroy - destroys a child DPNI under the root DPRC

    For command-specific help, use the --help option available for each command.
```

8.3.3.2.2.2 info command

The **info** command displays detailed information about a specific dpni object.

SYNTAX:

restool dpni info <*dpni-object*> [**--verbose**]

ARGUMENTS :

<*dpni-object*>

Specifies which dpni object to show detailed info for. The dpni-object argument is a

string specifying the object name—e.g. “dpni.7”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpni info dpni.7
dpni version: 5.0
dpni id: 7
plugged state: plugged
endpoint: dpmac.2, link is down
link status: 0 - down
mac address: 00:00:00:00:00:07
dpni_attr.options value is: 0x190
    DPNI_OPT_DIST_HASH
    DPNI_OPT_UNICAST_FILTER
    DPNI_OPT_MULTICAST_FILTER
max senders: 8
max traffic classes: 1
max distribution's size per RX traffic class:
    class 0's size: 15
max unicast filters: 16
max multicast filters: 64
max vlan filters: 0
max QoS entries: 0
max QoS key size: 0
max distribution key size: 4

$ restool dpni info dpni.7 --verbose
dpni version: 5.0
dpni id: 7
plugged state: plugged
endpoint: dpmac.2, link is down
link status: 0 - down
mac address: 00:00:00:00:00:07
dpni_attr.options value is: 0x190
    DPNI_OPT_DIST_HASH
    DPNI_OPT_UNICAST_FILTER
    DPNI_OPT_MULTICAST_FILTER
max senders: 8
max traffic classes: 1
max distribution's size per RX traffic class:
    class 0's size: 15
max unicast filters: 16
max multicast filters: 64
max vlan filters: 0
max QoS entries: 0
max QoS key size: 0
max distribution key size: 4
number of mappable regions: 0
number of interrupts: 1
interrupt 0's mask: 0
interrupt 0's status: 0
```

8.3.3.2.2.3 create command

The **create** command creates a new DPNI. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpni create --mac-addr=<*addr*> [OPTIONS]

ARGUMENTS :

--mac-addr=<*addr*>

String specifying primary MAC address (e.g., 00:00:05:00:00:05)

OPTIONS :

--max-senders=<*number*>

maximum number of different senders; will be used as the number of dedicated TX flows;

In case it isn't power-of-2 it will be ceiling to the next power-of-2 as HW demand it; 0 will

be treated as 1

--options=<*options-mask*>

Where <*options-mask*> is a comma separated list of DPNI options:

```
DPNI_OPT_ALLOW_DIST_KEY_PER_TC
DPNI_OPT_TX_CONF_DISABLED
DPNI_OPT_PRIVATE_TX_CONF_ERR_DISABLED
DPNI_OPT_DIST_HASH
DPNI_OPT_DIST_FS
DPNI_OPT_UNICAST_FILTER
DPNI_OPT_MULTICAST_FILTER
DPNI_OPT_VLAN_FILTER
DPNI_OPT_IPR
DPNI_OPT_IPF
DPNI_OPT_VLAN_MANIPULATION
DPNI_OPT_QOS_MASK_SUPPORT
DPNI_OPT_FS_MASK_SUPPORT
```

--max-tcs=<*number*>

Specifies the maximum number of traffic-classes

--max-dist-per-tc=<*dist-size*>,<*dist-size*>,…

Comma separated list of counts specifying the maximum distribution's size per RX traffic-

class

--max-unicast-filters=<*number*>

maximum number of unicast filters; 0 will be treated as 16

--max-multicast-filters=<*number*>

maximum number of multicast filters; 0 will be treated as 64

--max-vlan-filters=<*number*>

maximum number of vlan filters; '0' will be treated as '16'

--max-qos-entries=<*number*>

if max_tcs > 1, declares the maximum entries for the QoS table; '0' will be treated as '64'

--max-qos-key-size=<*number*>

maximum key size for the QoS look-up; '0' will be treated as '24' which enough for IPv4 5-tuple

--max-dist-key-size=<*number*>

maximum key size for the distribution; '0' will be treated as '24' which enough for IPv4 5-tuple

EXAMPLE:

Create a DPNI, specifying MAC address, with all default options:

```
$ restool dpni create --mac-addr=00:00:05:00:00:05
dpni.9 is crated under dprc.1
```

Create a DPNI, specifying MAC address, and some options:

```
$ restool dpni create --mac-addr=00:00:05:00:00:05
--options=DPNI_OPT_MULTICAST_FILTER,DPNI_OPT_UNICAST_FILTER
dpni.11 is created under dprc.1
```

8.3.3.2.2.4 create command

The **create** command creates a new DPNI. The name/id of the object created is displayed to stdout.

In the following part are presented the options when creating a DPNI using MC10.x firmware version. Also, restool is compatible with older MC firmware versions and will export another set of options in these other cases.

SYNTAX:

restool dpni create [OPTIONS]

OPTIONS :

--options=<*options-mask*>

Where <*options-mask*> is a comma separated list of DPNI options:

```
        DPNI_OPT_TX_FRM_RELEASE
        DPNI_OPT_NO_MAC_FILTER
        DPNI_OPT_HAS_POLICING
        DPNI_OPT_SHARED_CONGESTION
        DPNI_OPT_HAS_KEY_MASKING
        DPNI_OPT_NO_FS
```

--num-queues=<*number*>

Number of TX/RX queues use for traffic distribution. Used to distribute traffic to multiple GPP cores.
Defaults to one queue. Maximim supported value is 8

--num-tcs=<*number*>

Number of traffic classes (TCs), reserved for the DPNI. Defaults to one TC. Maximum supported value is 8

--num-entries=<*number*>
Number of entries in the MAC address filtering table. Allows both unicast and multicast entries.
By default, there are 80 entries.Maximum supported value is 80.

--vlan-entries=<*number*>
Number of entries in the VLAN address filtering table. By default, VLAN filtering is disabled. Maximum values is 16.

--qos-entries=<*number*>
Number of entries in the QoS classification table. Ignored if DPNI has a single TC. By default, set to 64.

--fs-entries=<*number*>
Number of entries in the flow steering table. Defaults to 64. Maximum value is 1024.

--container=<*container-name*>
Specifies the parent container name. e.g. dprc.2, dprc.3 etc.

EXAMPLE:

Create a DPNI with all default options:

```
$ restool dpni create
dpni.9 is created under dprc.1
```

Create a DPNI with some specific options as a child object of dprc.2 (dprc already created):

```
$ restool dpni create --options=DPNI_OPT_TX_FRM_RELEASE,DPNI_OPT_NO_FS --container=dprc.2
dpni.11 is created under dprc.2
```

8.3.3.2.2.5 destroy command

The **destroy** command destroys a DPNI.

SYNTAX:

restool dpni destroy <*dpni-object*>

ARGUMENTS :

<*dpni-object*>

Specifies which DPNI to destroy.

EXAMPLE:

```
$ restool dpni destroy dpni.9
dpni.9 is destroyed
```

8.3.3.2.3 DPIO Commands

8.3.3.2.3.1 help command

The **help** command displays usage information for the DPIO object

SYNTAX:

restool dpio help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpio help
usage: restool dpio <command> [--help][ARGS…]
Where <command> can be:
info - displays detailed information about a DPIO object.
create - creates a DPIO under the root DPRC
destroy - destroys a DPIO under the root DPRC

For command-specific help, use the --help option available for each command.
```

8.3.3.2.3.2 info command

The **info** command displays detailed information about a specific dpio object.

SYNTAX:

restool dpio info <*dpio-object*> [**--verbose**]

ARGUMENTS :

<*dpio-object*>

Specifies which dpio object to show detailed info for. The dpio-object argument is a

string specifying the object name—e.g. “dpio.7”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
# restool dpio info dpio.1
    dpio version: 3.0
    dpio id: 1
    plugged state: plugged
    offset of qbman software portal cache-enabled area: 0x20000
    offset of qbman software portal cache-inhibited area: 0x4020000
    qbman software portal id: 0x2
    dpio channel mode is: DPIO_LOCAL_CHANNEL
    number of priorities is: 0x8
    # restool dpio info dpio.1 --verbose
    dpio version: 3.0
    dpio id: 1
    plugged state: plugged
    offset of qbman software portal cache-enabled area: 0x20000
    offset of qbman software portal cache-inhibited area: 0x4020000
    qbman software portal id: 0x2
    dpio channel mode is: DPIO_LOCAL_CHANNEL
    number of priorities is: 0x8
    number of mappable regions: 2
    number of interrupts: 1
    interrupt 0's mask: 0
    interrupt 0's status: 0x8
```

8.3.3.2.3.3 create command

The **create** command creates a new DPIO. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpio create [OPTIONS]

OPTIONS :

--channel-mode=<*mode*>

Where <*mode*> is one of:

```
            DPIO_LOCAL_CHANNEL
            DPIO_NO_CHANNEL
```

Default value is DPIO_LOCAL_CHANNEL .

--num-priorities=<*number*>

Valid values for <*number*> are 1-8. Default value is 8.

EXAMPLE:

Create a DPIO with all default options:

$ restool dpio create

dpio.10 is created under dprc.1

Create a DPIO, specifying number of priorities:

```
    $ restool dpni create –num-priorities=4
    dpio.2 is created under dprc.1
```

8.3.3.2.3.4 create command

The **create** command creates a new DPIO. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpio create [OPTIONS]

OPTIONS :

--channel-mode=<*mode*>

Where <*mode*> is one of:

```
            DPIO_LOCAL_CHANNEL
            DPIO_NO_CHANNEL
```

Default value is DPIO_LOCAL_CHANNEL .

--num-priorities=<*number*>

Valid values for <*number*> are 1-8. Default value is 8.

--container=<*container_name*>

Specifies the parent container name. e.g. dprc.2, dprc.3 etc.

If it is not specified, the new object will be created under the default dprc.

EXAMPLE:

Create a DPIO with all default options:

```
    $ restool dpio create
    dpio.8 is created under dprc.1
```

Create a DPIO, specifying number of priorities:

```
    $ restool dpni create –num-priorities=4
    dpio.2 is created under dprc.1
```

8.3.3.2.3.5 destroy command

The **destroy** command destroys a DPIO.

SYNTAX:

restool dpio destroy <*dpio-object*>

ARGUMENTS :

<*dpio-object*>

Specifies which DPIO to destroy.

EXAMPLE:

```
    $ restool dpio destroy dpio.9
    dpio.9 is destroyed
```

8.3.3.2.4 DPSW Commands

text

8.3.3.2.4.1 help command

The **help** command displays usage information for the DPSW object

SYNTAX:

restool dpsw help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpsw help
    usage: restool dpsw <command> [--help][ARGS…]
    Where <command> can be:
    info - displays detailed information about a DPSW object.
    create - creates a DPSW under the root DPRC
    destroy - destroys a DPSW under the root DPRC

    For command-specific help, use the --help option available for each command.
```

8.3.3.2.4.2 info command

The **info** command displays detailed information about a specific dpsw object.

SYNTAX:

restool dpsw info <*dpsw-object*> [**--verbose**]

ARGUMENTS :

<*dpsw-object*>

Specifies which object to show detailed info for. The dpsw-object argument is a string

specifying the object name—e.g. “dpsw.2”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
    $ restool dpsw info dpsw.1
dpsw version: 6.0
dpsw id: 1
plugged state: unplugged
endpoints:
endpoint state: -1
        interface 0: No object associated
endpoint state: -1
        interface 1: No object associated
endpoint state: -1
        interface 2: No object associated
endpoint state: -1
        interface 3: No object associated
dpsw_attr.options value is: 0x1
        DPSW_OPT_FLOODING_DIS
max VLANs: 8
max FDBs: 8
DPSW frame storage memory size: 0
number of interfaces: 4
current number of VLANs: 1
current number of FDBs: 1
```

8.3.3.2.4.3 create command

The **create** command creates a new DPSW. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpsw create --num-ifs=<*number*> [OPTIONS]

ARGUMENTS :

--num-ifs=<*number*>

Number of external and internal interfaces.

OPTIONS :

--options=<*options-mask*>

Where <*options-mask*> is a comma separated list of DPSW options:

```
            DPSW_OPT_FLOODING_DIS
            DPSW_OPT_MULTICAST_DIS
            DPSW_OPT_CTRL_IF_DIS
            DPSW_OPT_FLOODING_METERING_DIS
            DPSW_OPT_METERING_EN
```

--max-vlans=<*number*>

Maximum Number of VLAN's. Default is 16.

--max-fdbs=<*number*>

Maximum Number of FDB's. Default is 16.

--num-fdb-entries=<*number*>

Number of FDB entries. Default is 1024.

--fdb-aging-time=<*number*>

Default FDB aging time in seconds. Default is 300 seconds.

--max-fdb-mc-groups=<*number*>

Number of multicast groups in each FDB table. Default is 32.

EXAMPLE:

Create a 4-port switch with all default options:

```
$ restool dpsw create --num-ifs=4
dpsw.8 is created under dprc.1
```

Create a 4-port switch with options:

```
$ restool dpsw create –num-ifs=4 –max-vlans=8 –max-fdb-mc-groups=300
--options=DPSW_OPT_TC_DIS,DPSW_OPT_FLOODING_DIS
dpsw.2 is created under dprc.1
```

8.3.3.2.4.4 create command

The **create** command creates a new DPSW. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpsw create --num-ifs=<*number*> [OPTIONS]

ARGUMENTS :

--num-ifs=<*number*>

Number of external and internal interfaces.

OPTIONS :

--options=<*options-mask*>

Where <*options-mask*> is a comma separated list of DPSW options:

```
DPSW_OPT_FLOODING_DIS
DPSW_OPT_MULTICAST_DIS
DPSW_OPT_CTRL_IF_DIS
DPSW_OPT_FLOODING_METERING_DIS
DPSW_OPT_METERING_EN
```

--max-vlans=<*number*>

Maximum Number of VLAN's. Default is 16.

--max-fdbs=<*number*>

Maximum Number of FDB's. Default is 16.

--num-fdb-entries=<*number*>

Number of FDB entries. Default is 1024.

--fdb-aging-time=<*number*>

Default FDB aging time in seconds. Default is 300 seconds.

--max-fdb-mc-groups=<*number*>

Number of multicast groups in each FDB table. Default is 32.

--container=<*container_name*>

Specifies the parent container name. e.g. dprc.2, dprc.3 etc.

If it is not specified, the new object will be created under the default dprc.

EXAMPLE:

Create a 4-port switch with all default options under dprc.2:

```
$ restool dpsw create --num-ifs=4 --container=dprc.2
dpsw.8 is created under dprc.2
```

Create a 4-port switch with options:

```
$ restool dpsw create –num-ifs=4 –max-vlans=8 –max-fdb-mc-groups=300
--options=DPSW_OPT_TC_DIS,DPSW_OPT_FLOODING_DIS
dpsw.2 is created under dprc.1
```

8.3.3.2.4.5 destroy command

The **destroy** command destroys a DPSW.

SYNTAX:

restool dpsw destroy <*dpsw-object*>

ARGUMENTS :

<*dpsw-object*>

Specifies which DPSW to destroy.

EXAMPLE:

```
$ restool dpsw destroy dpsw.8
dpsw.8 is destroyed
```

8.3.3.2.5 DPBP Commands

8.3.3.2.5.1 help command

The **help** command displays usage information for the DPBP object

SYNTAX:

restool dpbp help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpbp help
usage: restool dpbp <command> [--help][ARGS…]
Where <command> can be:
info - displays detailed information about a DPBP object.
create - creates a DPBP under the root DPRC
destroy - destroys a DPBP under the root DPRC

For command-specific help, use the --help option available for each command.
```

8.3.3.2.5.2 info command

The **info** command displays detailed information about a specific dpbp object.

SYNTAX:

restool dpbp info <*dpbp-object*> [**--verbose**]

ARGUMENTS :

<*dpbp-object*>

Specifies which dpbp object to show detailed info for. The dpbp-object argument is a

string specifying the object name—e.g. “dpbp.3”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpbp info dpbp.1
dpbp version: 2.0
dpbp id: 1
plugged state: plugged
buffer pool id: 0
```

8.3.3.2.5.3 create command

The **create** command creates a new DPBP. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpbp create [OPTIONS]

OPTIONS:

--container=<*container_name*>

Specifies the parent container name. e.g. dprc.2, dprc.3 etc.

If it is not specified, the new object will be created under the default dprc.

EXAMPLE:

Create a DPBP:

```
$ restool dpbp create
dpbp.2 is created under dprc.1
```

8.3.3.2.5.4 create command

The **create** command creates a new DPBP. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpbp create [OPTIONS]

OPTIONS:

--container=<*container_name*>

Specifies the parent container name. e.g. dprc.2, dprc.3 etc.

If it is not specified, the new object will be created under the default dprc.

EXAMPLE:

Create a DPBP under container dprc.3:

```
$ restool dpbp create --container=dprc.3
dpbp.2 is created under dprc.3
```

8.3.3.2.5.5 destroy command

The **destroy** command destroys a DPBP.

SYNTAX:

restool dpbp destroy <*dpbp-object*>

ARGUMENTS :

<*dpbp-object*>

Specifies which DPBP to destroy.

EXAMPLE:

```
$ restool dpbp destroy dpbp.2
dpbp.2 is destroyed
```

8.3.3.2.6 DPCON Commands

text

8.3.3.2.6.1 help command

The **help** command displays usage information for the DPCON object.

SYNTAX:

restool dpcon help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpcon help
usage: restool dpcon <command> [--help][ARGS…]
Where <command> can be:
info - displays detailed information about a DPCON object.
create - creates a DPCON under the root DPRC
destroy - destroys a DPCON under the root DPRC

For command-specific help, use the --help option available for each command.
```

8.3.3.2.6.2 info command

The **info** command displays detailed information about a specific dpcon object.

SYNTAX:

restool dpcon info <*dpcon-object*> [**--verbose**]

ARGUMENTS :

<*dpcon-object*>

Specifies which dpcon object to show detailed info for. The dpcon-object argument is

a string specifying the object name—e.g. “dpcon.8”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpcon info dpcon.1
dpcon version: 2.0
dpcon id: 1
plugged state: plugged
qbman channel id to be used by dequeue operation: 40
number of priorities for the DPCON channel: 8
```

8.3.3.2.6.3 create command

The **create** command creates a new DPCON. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpcon create [OPTIONS]

OPTIONS :

--num-priorities=<*number*>

Specifies the number of priorities, valid values are 1-8. Default is 1.

EXAMPLE:

Create a DPCON with 4 priorities:

```
$ restool dpcon create --num-priorites=4
dpcon.8 is created under dprc.1
```

8.3.3.2.6.4 create command

The **create** command creates a new DPCON. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpcon create [OPTIONS]

OPTIONS :

--num-priorities=<*number*>

Specifies the number of priorities, valid values are 1-8. Default is 1.

--container=<*container_name*>

Specifies the parent container name. e.g. dprc.2, dprc.3 etc.

If it is not specified, the new object will be created under the default dprc.

EXAMPLE:

Create a DPCON with 4 priorities:

```
$ restool dpcon create --num-priorites=4
dpcon.8 is created under dprc.1
```

8.3.3.2.6.5 destroy command

The **destroy** command destroys a DPCON.

SYNTAX:

restool dpcon destroy <*dpcon-object*>

ARGUMENTS :

<*dpcon-object*>

Specifies which DPCON to destroy.

EXAMPLE:

```
$ restool dpcon destroy dpcon.9
```

dpcon.9 is destroyed

8.3.3.2.7 DPCI Commands

8.3.3.2.7.1 help command

The **help** command displays usage information for the DPCI object.

SYNTAX:

restool dpci help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpci help
usage: restool dpci <command> [--help][ARGS…]
Where <command> can be:
info - displays detailed information about a DPCI object.
create - creates a DPCI under the root DPRC
destroy - destroys a DPCI under the root DPRC

For command-specific help, use the --help option available for each command.
```

8.3.3.2.7.2 info command

The **info** command displays detailed information about a specific dpci object.

SYNTAX:

restool dpci info <*dpci-object*> [**--verbose**]

ARGUMENTS :

<*dpci-object*>

Specifies which dpci object to show detailed info for. The dpci-object argument is a

string specifying the object name—e.g. “dpci.8”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpci info dpci.1
dpci version: 2.0
dpci id: 1
plugged state: plugged
num_of_priorities: 2
connected peer: dpci.4
peer's num_of_priorities: 2
link status: 0 - down
```

8.3.3.2.7.3 create command

The **create** command creates a new DPCI. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpci create [OPTIONS]

OPTIONS :

--num-priorities=<*number*>

Specifies the number of priorities, valid values are 1 or 2. Default is 1.

EXAMPLE:

Create a DPCI with 4 priorities:

```
    $ restool dpci create --num-priorites=2
    dpci.8 is created under dprc.1
```

8.3.3.2.7.4 create command

The **create** command creates a new DPCI. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpci create [OPTIONS]

OPTIONS :

--num-priorities=<*number*>

Specifies the number of priorities, valid values are 1 or 2. Default is 1.

--container=<*container_name*>

Specifies the parent container name. e.g. dprc.2, dprc.3 etc.

If it is not specified, the new object will be created under the default dprc.

EXAMPLE:

Create a DPCI with 4 priorities:

```
    $ restool dpci create --num-priorites=2
    dpci.8 is created under dprc.1
```

8.3.3.2.7.5 destroy command

The **destroy** command destroys a DPCI.

SYNTAX:

restool dpci destroy <*dpci-object*>

ARGUMENTS :

<*dpci-object*>

Specifies which DPCI to destroy.

EXAMPLE:

```
    $ restool dpci destroy dpci.9
```

dpci.9 is destroyed

8.3.3.2.8 DPSECI Commands

8.3.3.2.8.1 help command

The **help** command displays usage information for the DPSECI object.

SYNTAX:

restool dpseci help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpseci help
usage: restool dpseci <command> [--help][ARGS…]
Where <command> can be:
info - displays detailed information about a DPSECI object.
create - creates a DPSECI under the root DPRC
destroy - destroys a DPSECI under the root DPRC
For command-specific help, use the --help option available for each command.
```

8.3.3.2.8.2 info command

The **info** command displays detailed information about a specific dpseci object.

SYNTAX:

restool dpseci info *<dpseci-object>* [**--verbose**]

ARGUMENTS :

<dpseci-object>

Specifies which dpseci object to show detailed info for. The dpseci-object argument is

a string specifying the object name—e.g. “dpseci.8”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpseci info dpseci.1
dpseci version: 2.0
dpseci id: 1
plugged state: plugged
number of priorities: 1
dpci id: 1
```

8.3.3.2.8.3 create command

The **create** command creates a new DPSECI. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpseci create [OPTIONS]

OPTIONS :

--priorities=*<priority1,priority2>*

DPSEC support 2 priorities that can be individually set. Valid values for *<priority1>* and *<priority2>* are 1-8. Default is 1.

EXAMPLE:

Create a DPSECI with 4 priorities:

```
$ restool dpseci create --priorites=2,4
dpseci.9 is created under dprc.1
```

8.3.3.2.8.4 create command

The **create** command creates a new DPSECI. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpseci create [OPTIONS]

OPTIONS :

--priorities=<*priority1,priority2*>

DPSEC support 2 priorities that can be individually set. Valid values for <*priority1*> and <*priority2*> are 1-8. Default is 1.

--container=<*container_name*>
Specifies the parent container name. e.g. dprc.2, dprc.3 etc.
If it is not specified, the new object will be created under the default dprc.

EXAMPLE:

Create a DPSECI with 4 priorities:

```
$ restool dpseci create --priorites=2,4
dpseci.9 is created under dprc.1
```

8.3.3.2.8.5 destroy command

The **destroy** command destroys a DPSECI.

SYNTAX:

restool dpseci destroy <*dpseci-object*>

ARGUMENTS :

<*dpseci-object*>

Specifies which DPSECI to destroy.

EXAMPLE:

```
$ restool dpseci destroy dpseci.9
```

dpseci.9 is destroyed

8.3.3.2.9 DPDMUX Commands

8.3.3.2.9.1 help command

The **help** command displays usage information for the DPDMUX object.

SYNTAX:

restool dpdmux help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpdmux help
usage: restool dpdmux <command> [--help][ARGS…]
Where <command> can be:
info - displays detailed information about a DPDMUX object.
create - creates a DPDMUX under the root DPRC
destroy - destroys a DPDMUX under the root DPRC
```

```
    For command-specific help, use the --help option available for each command.
```

8.3.3.2.9.2 info command

The **info** command displays detailed information about a specific dpdmux object.

SYNTAX:

restool dpdmux info <*dpdmux-object*> [**--verbose**]

ARGUMENTS :

<*dpdmux-object*>

Specifies which dpdmux object to show detailed info for. The dpdmux-object

argument is a string specifying the object name—e.g. “dpdmux.2”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpdmux info dpdmux.0
dpdmux version: 4.1
dpdmux id: 0
plugged state: plugged
endpoints:
endpoint state: 0
        interface 0: dpmac.1, link is down
endpoint state: 0
        interface 1: dpni.0, link is down
endpoint state: 0
        interface 2: dpni.1, link is down
dpdmux_attr.options value is: 0x2
        DPDMUX_OPT_BRIDGE_EN
DPDMUX address table method: DPDMUX_METHOD_MAC
DPDMUX manipulation type: DPDMUX_MANIP_NONE
number of interfaces (excluding the uplink interface): 3
DPDMUX frame storage memory size: 0
control interface ID: 0
```

8.3.3.2.9.3 create command

The create command creates a new DPDMUX. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpdmux create --num-ifs=<*number*> [OPTIONS]

ARGUMENTS :

--num-ifs=<*number*>

Number of virtual interfaces (excluding the uplink interface).

OPTIONS :

--method=<*dmat_method*>

Where <*dmat_method*> defines the method of the DPDMUX address table. A valid value is one of the following:

```
            DPDMUX_METHOD_NONE
            DPDMUX_METHOD_C_VLAN_MAC
            DPDMUX_METHOD_MAC
            DPDMUX_METHOD_C_VLAN
            DPDMUX_METHOD_S_VLAN
```

Default is `DPDMUX_METHOD_C_VLAN_MAC`

--manip=<*manip*>

Where <*manip*> defines the DPDMUX required manipulation operation. A valid value is one of the following:

```
DPDMUX_MANIP_NONE

DPDMUX_MANIP_ADD_REMOVE_S_VLAN
```

Default is DPDMUX_MANIP_NONE

--options=<*options-mask*>

```
DPDMUX_OPT_BRIDGE_EN
```

Default is 0 (don’t set any options)

--max-dmat-entries=<*number*>

max entries in DPDMUX address table. Default is 64.

--max-mc-groups=<*number*>

Number of multicast groups in DPDMUX table. Default is 32 groups.

EXAMPLE:

Create a DPDMUX with all default options:

```
$ restool dpdmux create --num-ifs=4
    dpdmux.11 is created under dprc.1
```

8.3.3.2.9.4 create command

The create command creates a new DPDMUX. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpdmux create --num-ifs=<*number*> [OPTIONS]

ARGUMENTS :

--num-ifs=<*number*>

Number of virtual interfaces (excluding the uplink interface).

OPTIONS :

--method=<*dmat_method*>

Where <*dmat_method*> defines the method of the DPDMUX address table. A valid value is one of the following:

```
DPDMUX_METHOD_NONE
DPDMUX_METHOD_C_VLAN_MAC
DPDMUX_METHOD_MAC
DPDMUX_METHOD_C_VLAN
DPDMUX_METHOD_S_VLAN
```

Default is `DPDMUX_METHOD_C_VLAN_MAC`

--manip=<*manip*>

Where <*manip*> defines the DPDMUX required manipulation operation. A valid value is one of the following:

```
DPDMUX_MANIP_NONE

DPDMUX_MANIP_ADD_REMOVE_S_VLAN
```

Default is DPDMUX_MANIP_NONE

--options=<*options-mask*>

```
            DPDMUX_OPT_BRIDGE_EN
```

Default is 0 (don't set any options)

--max-dmat-entries=<*number*>

max entries in DPDMUX address table. Default is 64.

--max-mc-groups=<*number*>

Number of multicast groups in DPDMUX table. Default is 32 groups.

--container=<*container_name*>
Specifies the parent container name. e.g. dprc.2, dprc.3 etc.
If it is not specified, the new object will be created under the default dprc.

EXAMPLE:

Create a DPDMUX with all default options under dprc.2:

```
$ restool dpdmux create --num-ifs=4 --container=dprc.2
    dpdmux.11 is created under dprc.2
```

8.3.3.2.9.5 destroy command

The **destroy** command destroys a DPDMUX.

SYNTAX:

restool dpdmux destroy <*dpdmux-object*>

ARGUMENTS :

<*dpdmux-object*>

Specifies which DPDMUX to destroy.

EXAMPLE:

```
    $ restool dpdmux destr
```

8.3.3.2.10 DPMCP Commands

8.3.3.2.10.1 help command

The **help** command displays usage information for the DPMCP object.

SYNTAX:

restool dpmcp help

ARGUMENTS:

none

EXAMPLE:

```
    $ restool dpmcp help
    usage: restool dpmcp <command> [--help] [ARGS…]
    Where <command> can be:
        info - displays detailed information about a DPMCP object.
        create - creates a DPMCP under the root DPRC
        destroy - destroys a DPMCP under the root DPRC

    For command-specific help, use the --help option available for each command.
```

8.3.3.2.10.2 info command

The **info** command displays detailed information about a specific dpmcp object.

SYNTAX:

restool dpmcp info <*dpmcp-object*> [**--verbose**]

ARGUMENTS :

<*dpmcp-object*>

Specifies which dpmcp object to show detailed info for. The dpmcp-object argument

is a string specifying the object name—e.g. “dpmcp.8”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpmcp info dpmcp.5
dpmcp version: 1.0
dpmcp object id/portal id: 5
plugged state: plugged
```

8.3.3.2.10.3 create command

The **create** command creates a new DPMCP. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpmcp create

EXAMPLE:

Create a DPMCP:

```
$ restool dpmcp create
dpmcp.15 is created under dprc.1
$ restool dpmcp create
MC error: No resource (status 0x8)
// when you see this error, it usually means no free portal available at this time.
```

8.3.3.2.10.4 create command

The **create** command creates a new DPMCP. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpmcp create [OPTIONS]

--container=<*container_name*>
Specifies the parent container name. e.g. dprc.2, dprc.3 etc.
If it is not specified, the new object will be created under the default dprc.

EXAMPLE:

Create a DPMCP:

```
$ restool dpmcp create
dpmcp.15 is created under dprc.1
$ restool dpmcp create
MC error: No resource (status 0x8)
// when you see this error, it usually means no free portal available at this time.
```

8.3.3.2.10.5 destroy command

The **destroy** command destroys a DPMCP.

SYNTAX:

restool dpmcp destroy <*dpmcp-object*>

ARGUMENTS :

<*dpmcp-object*>

Specifies which DPMCP to destroy.

EXAMPLE:

```
$ restool dpmcp destroy dpmcp.9
dpmcp.9 is destroyed
```

8.3.3.2.11 DPMAC Commands

8.3.3.2.11.1 help command

The **help** command displays usage information for the DPMAC object.

SYNTAX:

restool dpmac help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpmac help
usage: restool dpmac <command> [--help] [ARGS…]
Where <command> can be:
    info - displays detailed information about a DPMAC object.
    create - creates a DPMAC under the root DPRC
    destroy - destroys a DPMAC under the root DPRC

For command-specific help, use the --help option available for each command.
```

8.3.3.2.11.2 info command

The **info** command displays detailed information about a specific dpmac object.

SYNTAX:

restool dpmac info <*dpmac-object*> [**--verbose**]

ARGUMENTS:

<*dpmac-object*>

Specifies which dpmac object to show detailed info for. The dpmac-object argument

is a string specifying the object name—e.g. “dpmac.8”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpmac info dpmac.5
dpmcp version: 2.0
dpmac object id/phy id: 5
plugged state: plugged
```

8.3.3.2.11.3 create command

The **create** command creates a new DPMAC. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpmac create --mac-id=<*number*>

--mac-id=<*number*>

Specifies the mac id.

EXAMPLE:

Create a DPMAC with valid portal id:

```
$ restool dpmac create --mac-id=15
dpmac.15 is created under dprc.1
```

8.3.3.2.11.4 create command

The **create** command creates a new DPMAC. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpmac create --mac-id=<*number*> [OPTIONS]

--mac-id=<*number*>

Specifies the mac id.

OPTIONS:
--container=<*container_name*>
Specifies the parent container name. e.g. dprc.2, dprc.3 etc.
If it is not specified, the new object will be created under the default dprc.

EXAMPLE:

Create a DPMAC with valid portal id:

```
$ restool dpmac create --mac-id=6
dpmac.6 is created under dprc.1
```

8.3.3.2.11.5 destroy command

The **destroy** command destroys a DPMAC.

SYNTAX:

restool dpmac destroy <*dpmac-object*>

ARGUMENTS :

<*dpmac-object*>

Specifies which DPMAC to destroy.

EXAMPLE:

```
$ restool dpmac destroy dpmac.9
dpmac.9 is destroyed
```

8.3.3.2.12 DPDCEI Commands

8.3.3.2.12.1 help command

The **help** command displays usage information for the DPDCEI object.

SYNTAX:

restool dpdcei help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpdcei help
usage: restool dpdcei <command> [--help][ARGS…]
Where <command> can be:
info - displays detailed information about a DPDCEI object.
create - creates a DPDCEI under the root DPRC
destroy - destroys a DPDCEI under the root DPRC

For command-specific help, use the --help option available for each command.
```

8.3.3.2.12.2 info command

The **info** command displays detailed information about a specific dpdcei object.

SYNTAX:

restool dpdcei info <*dpdcei-object*> [**--verbose**]

ARGUMENTS :

<*dpdcei-object*>

Specifies which dpdcei object to show detailed info for. The dpdcei-object argument

is a string specifying the object name, e.g. “dpdcei.2”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpdcei info dpdcei.5
dpdcei version: 0.0
dpdcei id: 5
plugged state: plugged
DPDCEI engine: DPDCEI_ENGINE_COMPRESSION
```

8.3.3.2.12.3 create command

The **create** command creates a new DPDCEI. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpdcei create --engine=<*engine*> --priority=<*number*>

--engine=<*engine*>

Compression or decompression engine to be selected.

A valid value is one of the following:

```
DPDCEI_ENGINE_COMPRESSION

DPDCEI_ENGINE_DECOMPRESSION
```

--priority=<*number*>

Priority for DCE hardware processing (valid values 1-8)

EXAMPLE:

Create a DPDCEI:

```
$ restool dpdcei create --engine=DPDCEI_ENGINE_COMPRESSION --priority=2
dpdcei.0 is created under dprc.1
$ restool dpdcei create --engine=DPDCEI_ENGINE_COMPRESSION --priority=3
dpdcei.1 is created under dprc.1
```

8.3.3.2.12.4 create command

The **create** command creates a new DPDCEI. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpdcei create --engine=<*engine*> --priority=<*number*> [OPTIONS]

--engine=<*engine*>

Compression or decompression engine to be selected.

A valid value is one of the following:

```
DPDCEI_ENGINE_COMPRESSION
DPDCEI_ENGINE_DECOMPRESSION
```

--priority=<*number*>

Priority for DCE hardware processing (valid values 1-8)

OPTIONS:
--container=<*container_name*>
Specifies the parent container name. e.g. dprc.2, dprc.3 etc.
If it is not specified, the new object will be created under the default dprc.

EXAMPLE:

Create a DPDCEI:

```
$ restool dpdcei create --engine=DPDCEI_ENGINE_COMPRESSION --priority=2
dpdcei.0 is created under dprc.1
$ restool dpdcei create --engine=DPDCEI_ENGINE_COMPRESSION --priority=3
dpdcei.1 is created under dprc.1
```

8.3.3.2.12.5 destroy command

The **destroy** command destroys a DPDCEI.

SYNTAX:

restool dpdcei destroy <*dpdcei-object*>

ARGUMENTS :

<*dpdcei-object*>

Specifies which DPDCEI to destroy.

EXAMPLE:

```
$ restool dpdcei destroy dpdcei.9
dpdcei.9 is destroyed
```

8.3.3.2.13 DPAIOP Commands

8.3.3.2.13.1 help command

The **help** command displays usage information for the DPAIOP object.

SYNTAX:

restool dpaiop help

ARGUMENTS:

none

EXAMPLE:

```
$ restool dpaiop help
usage: restool dpaiop <command> [--help][ARGS…]
Where <command> can be:
info - displays detailed information about a DPAIOP object.
create - creates a DPAIOP under the root DPRC
destroy - destroys a DPAIOP under the root DPRC

For command-specific help, use the --help option available for each command.
```

8.3.3.2.13.2 info command

The **info** command displays detailed information about a specific dpaiop object.

SYNTAX:

restool dpaiop info <*dpaiop-object*> [**--verbose**]

ARGUMENTS :

<*dpaiop-object*>

Specifies which dpaiop object to show detailed info for. The *dpaiop-object* argument

is a string specifying the object name—e.g. “dpaiop.8”.

--verbose

Shows extended/verbose information about the object

EXAMPLE:

```
$ restool dpaiop info dpaiop.5
```

dpmcp version: 1.0

dpmcp id: 5

plugged state: plugged

dpaiop server layer version: 2.1.3

DPAIOP state: DPAIOP_STATE_RUNNING

8.3.3.2.13.3 create command

The **create** command creates a new DPAIOP. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpaiop create --aiop-id=<*number*> --aiop-container=<*container-name*>

ARGUMENTS :

--aiop-container=<*container-name*>

Specifies the AIOP container name, e.g. dprc.3, dprc.4, etc.

OPTIONS :

--aiop-id=<*number*>

Specifies the AIOP ID. Currently aiop container could only hold one dpaiop. Valid

number is 0. Default number is 0.

EXAMPLE:

Create a DPAIOP:

```
$ restool dpaiop create –-aiop-id=0 --aiop-container=dprc.3
dpaiop.0 is created under dprc.3

$ restool dpaiop create --aiop-container=dprc.3
dpaiop.0 is created under dprc.3
```

8.3.3.2.13.4 create command

The **create** command creates a new DPAIOP. The name/id of the object created is displayed to stdout.

SYNTAX:

restool dpaiop create --aiop-container=<*container-name*> [OPTIONS]

ARGUMENTS :

--aiop-container=<*container-name*>

Specifies the AIOP container name, e.g. dprc.3, dprc.4, etc.

OPTIONS :

--container=<*container-name*>

Specifies the parent container name. e.g. dprc.2, dprc.3 etc.
If it is not specified, the new object will be created under the default dprc.

EXAMPLE:

Create a DPAIOP:

```
$ restool dpaiop create --aiop-container=dprc.3
dpaiop.0 is created under dprc.1
```

Create a DPAIOP as a child object of dprc.2:

```
$ restool dpaiop create --aiop-container=dprc.3 --container=dprc.2
dpaiop.0 is created under dprc.2
```

8.3.3.2.13.5 destroy command

The **destroy** command destroys a DPAIOP.

SYNTAX:

restool dpaiop destroy <*dpaiop-object*>

ARGUMENTS :

<*dpaiop-object*>

Specifies which DPAIOP to destroy.

EXAMPLE:

```
$ restool dpio destroy dpaiop.9
dpaiop.9 is destroyed
```

8.3.4 DPAA2 User Manual

DPAA2 is a hardware-level networking architecture found on some NXP SoCs. This section provides technical information on this architecture mainly for software developers.

Click here to access the DPAA2 User Manual PDF.

8.3.5 DPAA2 API Reference Manual

Click here to access the DPAA2 API Reference Manual PDF.

8.3.6 Backplane Support on Layerscape

8.3.6.1 Overview

This section describes how to enable backplane support for Layerscape devices with embedded support for this type of connection.

Ethernet operation over electrical backplanes, also referred to as “Backplane Ethernet,” combines the IEEE 802.3 Media Access Control (MAC) and MAC Control sublayers with a family of Physical Layers defined to support operation over a modular chassis backplane. Usually, there is no external PHY involved and the connection is made at the SoC’s PCS (Physical Coding Sublayer) level. Based on the link quality, a signal equalization is required. In cases where the link is realized based on passive direct attach cables, the link may need to be established with only the default (recommended) parameters for equalization. The standard states that a start-up algorithm should be in place in order to get the link up.

8.3.6.1.1 10GBase-KR Support on Layerscape Platforms

Layerscape devices come with embedded support for backplane connections at different baud rates. 10G is present in custom boards of LS1088A, LS2088A, and LS1046A devices.

The enablement of backplane support is done in two parts. One refers to support from the device tree while the other is contained in the Linux kernel driver.

In the device tree, the following value is valid `backplane-mode`:

- `10gbase-kr`

In the Linux kernel driver, the implementation is different depending on each of the four different cases. However, the following changes are common for all:

- Advertise the link partner with the correct working mode.
- Put the lane in the correct mode (10GBase-KR).
- Use the recommended (if it is the case) parameters for pre- and post-tap coefficients in the lane initialization phase. This affects the starting point of the algorithm.
- Optionally, update the constraint relation between tap coefficients if this is needed.

8.3.6.1.2 Physical Layer Signaling System

The Backplane Ethernet extends the family of 10GBASE-R Physical Layer signaling systems to include the 10GBASE-KR. This embodiment specifies 10 Gb/s operation over two differential, controlled impedance pairs of traces (one pair for transmit, one pair for receive). This system employs the 10GBASE-R PCS, the serial PMA, and the 10GBASE-KR PMD sublayers.

The 10GBASE-KR PMD's control function implements the 10GBASE-KR start-up protocol. This protocol facilitates timing recovery and equalization while also providing a mechanism through which the receiver can tune the transmit equalizer to optimize performance over the backplane interconnect. The 10GBASE-KR PHY may optionally include 10GBASE-R Forward Error Correction (FEC).

Details about the aforementioned layers can be found in Clause 49, 51 and 74 of the IEEE Std 802.3.

8.3.6.1.3 Auto-negotiation

Auto-negotiation allows the devices at both ends of a link segment to advertise abilities, acknowledge receipt, and discover the common modes of operation that both devices share. It also rejects the use of operational modes not shared by both devices. Auto-negotiation does not test link segment characteristics.

8.3.6.1.4 Link Training

Link Training occurs after auto-negotiation has determined the link to be a 10GBase-KR, but before auto-negotiation is done. It continuously exchanges messages (training frames) between the local and the remote device as part of the start-up phase. Link training also tunes the analog parameters of the remote and local SerDes transmitter to improve the link quality. Both LP (link partner/remote device) and LD (local device) perform link training in parallel. Link training stops when both sides decide that the link is passable. Then the link is considered up.

8.3.6.2 Enable Backplane Support on Layerscape

8.3.6.2.1 Setup

Hardware Setup

- Two custom boards (LS1088A, LS2088A, LS1046A) with no XFI retimers
- Passive direct attach cable (l <= 1M)

Software Setup

- Linux kernel with backplane support enabled
- Device tree for custom boards (LS1088A, LS2088A, LS1046A) with backplane PHY devices

NOTE

The custom boards (LS1088A, LS2088A, LS1046A) were used for this implementation/demonstration. You may use any custom board that enables access to the Backplane Ethernet feature.

8.3.6.2.2 Enable Backplane Connection from MC

This step is required only for devices based on DPAA2 architecture.

Use `MAC_LINK_TYPE_BACKPLANE` for all ports that will be used for backplane connections. In order to do that in the MC data path configuration file, add an entry like below for all ports used:

```
board_info {
        ports {
                        …
            mac@1 {
                link_type = "MAC_LINK_TYPE_BACKPLANE";
            };
            …
            };
        };
    };
```

Deploy this configuration file on the target board as per Data Path Configuration chapter from DPAA2 User Manual.

NOTE: This is a very important step and omitting it can lead to an unreliable backplane connection. Random link-down or link-up events can be experienced. This is due to a concurrent access to MDIO bus between MC core (MC firmware) and GPP core (Linux kernel).

8.3.6.2.3 Enable Backplane Support in Linux Kernel

8.3.6.2.3.1 Enable Backplane PHY Driver

Enable backplane support from:

```
Device Drivers->Network device support->PHY device support and infrastructure->FSL_BACKPLANE
```

.

8.3.6.2.3.2 Add Backplane PHY Devices in Device Tree

8.3.6.2.3.2.1 SerDes Device and Internal MDIO Buses

Normally the SerDes device and all internal MDIO buses should be listed in the SoC’s common device tree source file:

```
<linux_kernel_repo>/arch/arm64/boot/dts/freescale/fsl-<device>xa.dtsi file.
```

To see if the SerDes module is listed, examine a block like the one below:

```
serdes1: serdes@1ea0000 {
    reg = <0x0 0x1ea0000 0 0x00002000>;
       compatible = "fsl,serdes-10g";
};
```

If the SerDes module is listed then it means that the serdes1 (label for SerDes node) is registered and can be used. The only client of this node is the kernel backplane PHY driver which uses the node's unit address as a base address. The base address is mapped in the SOC's memory space to further access specific MDIO registers used to control the backplane connection.

In the DTS, there must be a `serdes1` node like the one represented above. If the node is not present, then it must be added. The device base address is listed in SoC’s CCSR memory map. One thing that should be considered is endianness of SerDes module, which can be different than that of the GPP. In this case, the module's registers must be accessed using the endianness. Currently, the Linux kernel backplane PHY driver does not support access dependent on target endianness. One way to do this will be to use the endianness attribute in the device tree. This way the driver can be used on different targets without changes.

Next look after internal MDIO buses listed in the device tree. See the block below as an example:

```
pcs_mdio1:  mdio@0x8c07000 {
        compatibile = “fsl, fman-memac-mdio”;
        reg = <0x0 0x8c07000 0x0 0x1000>;
        device_type = “mdio”;
        little-endian;
};
```

The block above shows that `pcs_mdio1` is listed in the device tree. The unit address of this node (`0x8c7000`) is the WRIOP internal physical port 1 base address as it is mapped in the SoC memory space. The address `0x8c0000` is the WRIOP port block base address as it is listed in SoC reference manual. The address `0x7000` is the physical port offset in the WRIOP internal memory map. All `pcs_mdio` ports have an offset of `0x4000` between them, so the next port will be located at `0xb000` and so on. The attribute `fsl, fman-memac-mdio` means that the FSL MDIO driver will be used to access this MDIO bus. It is required to use a dedicated MDIO bus driver to access internal MDIO buses, because it uses proprietary MDIO control registers block and offset. See the DPAA2 User Manual for details about MDIO registers block.

The kernel MDIO driver used is:

```
<linux_kernel_repo>/drivers/net/Ethernet/freescale/xgmac_mdio.c
```

Internal MDIO buses should be listed for all PCS ports that support backplane KR connection, in the device tree. This is because for every port used, the management registers are accessed through the MDIO bus. See DPAA2 architecture for details on how internal MDIO registers block is mapped for every physical port and MDIO registers subchapter of SerDes chapter from SoC reference manual.

If no internal MDIO bus is listed then add one internal MDIO bus for every PCS port target that will be used in a backplane connection.

8.3.6.2.3.2.2 Backplane PHY Devices

PCS ports are specific to each board. Backplane PHY devices should be added in board-specific device trees:

```
<linux_kernel_repo>/arch/arm64/boot/dts/freescale/fsl-<device>-<qds, rdb>.dts.
```

A backplane PHY device is registered on an internal MDIO bus. The block below is an example:

```
&pcs_mdio1 {
    pcs_phys1: ethernet-phy@0  {
        backplane-mode = "10gbase-kr";
        compatible = "ethernet-phy-ieee802.3-c45";
        reg = <0x0>;
        fsl, lane-handle = *lt;&serdes1>;
        fsl, lane-reg = <0x9c0 0x40>; /* lane H */
};
```

Pcs-phys1 is listed on the MDIO bus and should be discovered when this bus is probed. The kernel backplane PHY driver should also register a PHY driver using PHY hardware ID (read via MDIO bus).

The `backplane-mode` attribute tells the kernel backplane PHY driver how to configure a specific SerDes lane. Currently, a SerDes lane can be configured as: 10gbase-kr

The `fsl, lane-handle` attribute is used to identify which SerDes lane the PCS port belongs to. In this case "serdes1" is used.

The `fsl, lane-reg` attribute is used to identify the SerDes lane used to send and receive data. `0x9c0` is the lane H offset in the SerDes1 internal memory map. See each platform's SoC Reference Manual for details and to find the other lane's offsets.

For LS2088A boards backplane PHY devices are added already for use with lane from H to E. Note that for SerDes 1 lane are numbered in the reversed order compared to WRIO physical ports and MACs.

If the backplane PHY device is not registered on internal MDIO buses for a specific board, then it can be added in the DTS.

8.3.6.2.3.2.3 Connect with Backplane PHY Device Handle

Because the kernel PHY driver is instantiated by the kernel MAC driver, there should be a specified connection between the MAC and a specific PHY in the device tree. In the following example, the backplane PHY from SerDes lane H is used:

```
&dpmac1 {
        phy-handle = <&pcs_phy1>;
};
```

8.3.6.2.4 SerDes Setup

- Enable XFI protocol on SerDes lane - reset configuration word.
- Initialize the SerDes lane registers for the mode:

  ```
  - Ethernet 10GBASE-KR
  ```
- Check the link capabilities with AN - software.
- Train the link - software.

8.3.6.2.5 Board Configuration

HW Adjustments

XFI retimers soldered on boards must be removed and PCS output signals should be routed directly to SFP+ cages pins. This is a very important operation and should be carried out carefully.

On SerDes1 module, XFI/10GBase-KR protocol will be activated on all eight lanes using `0x2a` for protocol selection bits.

Connection Cables

Connect with passive direct attach cable.

Two custom boards (LS1088A, LS2088A, LS1046A) will be connected back to back with a passive direct attach copper cable, with a maximum length of 1m (for ex SFP-H10GB-CU1M).

8.3.6.3 Use Cases

ping

In order to run a backplane *ping use case*, two boards must be connected back to back with a passive direct attach copper cable. Start MC with specified DPC file. Apply DPL using `fsl_mc apply dpl` command from U-boot and then boot Linux on both boards. After booting Linux, the interfaces must be configured properly for the two ports connected together using two IP addresses from the same IP class.

For example, use:

- On first board: `ifconfig ni0 1.1.1.1`
- On second board: `ifconfig ni0 1.1.1.2`

Once the interfaces are configured, traffic can be sent between the two the boards through the backplane link:

- On first board: `ping 1.1.1.2`
- On second board: `ping 1.1.1.1`

netperf

the backplane *netperf use case* is similar to the ping usecase described above, and it is used for backplane performance benchmark. The board configuration must be done identically as described above. The difference is how traffic is sent between the two boards.

For example using UDP streams:

- On first board: `netperf -H 1.1.1.2 -l 60 -t UDP_STREAM -N &`
- On second board: `netperf -H 1.1.1.1 -l 60 -t UDP_STREAM -N &`

8.3.7 AIOP

8.3.7.1 AIOP Sample Applications

The Advanced I/O Processor (AIOP) hardware, an optional component of the DPAA2 architecture, is a C-programmable engine that enables power efficient packet-oriented processing. This section provides sample applications that can be used to exercise functionality of offloading packet processing in AIOP.

8.3.7.1.1 Creating AIOP Containers

This section describes how to dynamically create Data Path Resource Containers owned by AIOP and how to load AIOP ELF by using AIOP Tool.

cd /usr/aiop/scripts

In this folder there are two scripts based on restool available:

- `dynamic_aiop_only.sh` - script used by reference applications only with AIOP side, not interacting with other components
- `dynamic_aiop_root.sh` - script used by reference applications that interact with Linux Kernel running on GPPS

Scripts do not require any command line argument as input. Sample execution flow is shown below:

```
# ./dynamic_aiop_only.sh
Creating AIOP Container
Assigned dpbp.1 to dprc.2
Assigned dpbp.2 to dprc.2
Assigned dpbp.3 to dprc.2
Assigned dpni.1 to dprc.2
Connecting dpni.1<------->dpmac.1
Assigned dpni.2 to dprc.2
Connecting dpni.2<------->dpmac.2
AIOP Container dprc.2 created
----- Contents of AIOP Container: dprc.2 -----
dprc.2 contains 5 objects:
object          label           plugged-state
dpni.2                          plugged
dpni.1                          plugged
dpbp.3                          plugged
dpbp.2                          plugged
dpbp.1                          plugged
-----
====================================================
Creating AIOP Tool Container
Assigned dpaiop.0 to dprc.3
Assigned dpmcp.22 to dprc.3
AIOP Tool Container dprc.3 created
----- Contents of AIOP Tool Container: dprc.3 -----
dprc.3 contains 2 objects:
object          label           plugged-state
dpaiop.0                        plugged
dpmcp.22                        plugged
-----
====================================================
Performing VFIO mapping for AIOP Tool Container (dprc.3)
Performing vfio mapping for dprc.3
[  796.531485] vfio-fsl-mc dprc.3: Binding with vfio-fsl_mc driver
[  796.540756] vfio-fsl-mc dpaiop.0: Binding with vfio-fsl_mc driver
[  796.547364] vfio-fsl-mc dpmcp.22: Binding with vfio-fsl_mc driver
========== Summary =================================
 AIOP Container: dprc.2
 AIOP Tool Container: dprc.3
====================================================
```

To load the AIOP binary by using AIOP Tool, the AIOP Tool Container is necessary. In the above case is `dprc.3`. The following command should be executed:

```
# aiop_tool load -g dprc.3 -f /usr/aiop/bin/aiop_reflector.elf &
AIOP Image (aiop_reflector.elf) loaded successfully.
```

To ensure that AIOP image is indeed loaded and running, the AIOP console can be checked by executing one of the following commands:

```
# restool dpaiop info dpaiop.0
dpaiop id: 0
dpaiop version: 2.3
plugged state: plugged
dpaiop server layer version: 7.2.0
DPAIOP state: DPAIOP_STATE_RUNNING

root@ls2085ardb:~# cat /dev/fsl_aiop_console
        . . .
```

8.3.7.1.2 AIOP Packet Reflector Application

8.3.7.1.2.1 AIOP Packet Reflector Overview

This section demonstrates a simple application data path on the AIOP.

The application performs the following functions:

- Configure the fields used for the initial order scope hash generation. The fields are: the source address, the destination address, the protocol type fields from the IP header and the source port and the destination port from the L4 header. For every packet received by WRIOP a hashed Initial Ordering Scope (IOS) will be generated based on these values.
- Set Concurrent Execution (XC) as the initial packets processing mode. In the initial stage of processing, the packets are processed concurrently by many cores in their ordering scope.
- Drop non IPv4 packets.
- Switch the source and destination MAC and IP addresses of the received packets.
- Transition into Exclusive execution (XX) in order to restore packet order. This is optional and can be deactivated through define EXCLUSIVE_MODE
- Reflect back the packet on the same interface from which it was received.

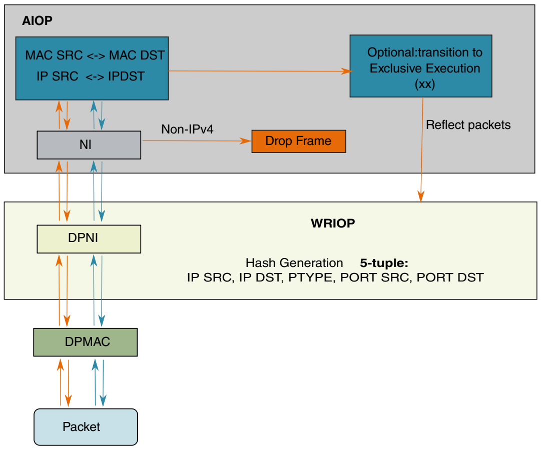


Figure 162. AIOP Packet Reflector overview

8.3.7.1.2.2 Running the Reflector Application

The ELF for AIOP reflector is `/usr/aiop/bin/aiop_reflector.elf`. This can be loaded via AIOP Tool as section Creating AIOP Containers describes. For this application the script `dynamic_aiop_only.sh` must be executed before loading the image:

```
# ./dynamic_aiop_only.sh
```

To load the AIOP Reflector Application, execute the following command:

```
# aiop_tool load -g dprc.3 -f /usr/aiop/bin/aiop_reflector.elf &
```

To check if the AIOP Reflector Application loaded successfully, execute the following command in the Linux command shell:

```
# cat /dev/fsl_aiop_console | grep REFLECTOR
```

The command output should display information about the Network Interfaces (NIs) that were successfully configured: NI instance (e.g. NI 0) together with the associated MAC address.

```
REFLECTOR : Successfully configured ni0 (dpni.2)
REFLECTOR : dpni.2 <---connected---> dpmac.2 (MAC addr: 00:00:00:00:00:07)
REFLECTOR : Successfully configured ni1 (dpni.1)
REFLECTOR : dpni.1 <---connected---> dpmac.1 (MAC addr: 00:00:00:00:00:06)
```

NOTE

Although the AIOP container contains multiple three DPNIs (DPNI6, DPNI7 and DPNI10), the AIOP Reflector will use only the DPNIs that have a DPMAC as endpoint (on which it can succesfully configure the order scope). The others will be skipped in application initialization.

During frame processing, the AIOP Logger will print the following brief information about every reflected packet:

- AIOP Core number on which the frame was processed
- Received MAC source and destination addresses
- Received IP source and destination addresses

```
# busybox tail -f /dev/fsl_aiop_console

RX on NI 0 | CORE:15
      MAC_SA: 00-10-94-00-00-02 MAC_DA: 00-00-00-00-00-06
      IP_SRC: 192.85.1.1 IP_DST: 192.0.0.1

RX on NI 0 | CORE:14
      MAC_SA: 00-10-94-00-00-02 MAC_DA: 00-00-00-00-00-06
      IP_SRC: 192.85.1.2 IP_DST: 192.0.0.1

 . . .

RX on NI 0 | CORE:9
      MAC_SA: 00-10-94-00-00-02 MAC_DA: 00-00-00-00-00-06
      IP_SRC: 192.85.1.3 IP_DST: 192.0.0.1
```

8.3.7.1.2.3 Generating Traffic to Test AIOP Reflector Application

An external traffic source is needed to pass traffic to the reflector application. This could be another RDB or some other traffic generator. Two optical 10G ports need to be connected to MAC1 and MAC2 of RDB-1.

The following graphic illustrates how traffic moves from a packet generator to RDB-1 board

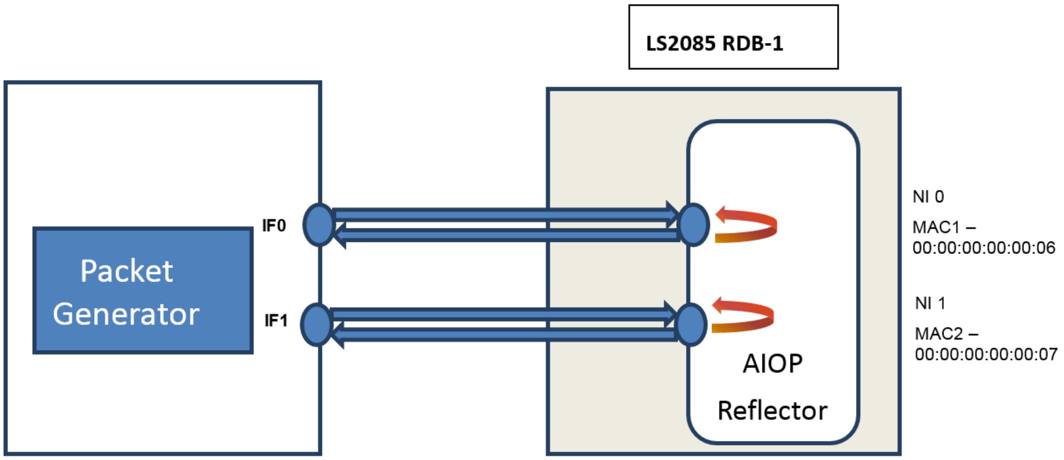


Figure 163. Generating traffic for AIOP Packet Reflector

When using a Packet Generator to inject traffic, connect one port of the packet generator to MAC1 of RDB-1 and a second port to MAC2 of RDB-1.

On RDB-1 ensure that both AIOP network interfaces are in link up state:

```
$ cat /dev/fsl_aiop_console | grep REFLECTOR
REFLECTOR : ni0 link is UP
REFLECTOR : ni1 link is UP
```

Generate an IPv4 frames for each of the two ports:

```
        MAC source:          ANY
        MAC destination:     00:00:00:00:00:06 for MAC1
                             00:00:00:00:00:07 for MAC2
        IP source:           ANY
        IP destination:      ANY
```

On RDB-1, the AIOP Logger will print a brief information about every frame that is being reflected.

8.3.7.1.3 AIOP Packet Classifier Application

8.3.7.1.3.1 AIOP Packet Classifier Overview

The purpose of this sample application is to demonstrate how to perform a simple Classification on the AIOP. The application will apply a Classification criteria for every IPv4 frame and will reflect back only accepted frames.

There are three execution modes listed below:

- Exclusive execution (XX): This mode forces atomic processing of packets. Each packet is processed in order of its arrival before the next packet is processed.
- Concurrent execution (XC): This mode processes packets within a flow concurrently on the AIOP. Packets may become misordered as part of this parallel processing. To restore packet order, the data path transitions to exclusive execution mode before transmitting the forwarded packets.
- Unordered. In this mode, packet ordering is not considered and concurrent packet processing takes place.

The application performs the following functions:

- Put network interfaces in promiscuous mode in order to allow packet reception regardless of its MAC destination address.
- Configure the fields used for the initial order scope hash generation. The fields are: the source address, the destination address, the protocol type fields from the IP header and the source port and the destination port from the L4 header. For every packet received by WRIOP a hashed Initial Ordering Scope (IOS) will be generated based on these values.
  - — Set Concurrent Execution (XC) as the initial packets processing mode. In the initial stage of processing, the packets are processed concurrently by many cores in their ordering scope.
  - — Classify the packets to enable different processing modes to be run based on the type of traffic. Classification is based on the following fields: IPv4 source and destination addresses, Protocol number, Source and Destination Layer 4 ports. The packets processing mode is selected as a result of the classification as follows:
  - — TCP packets are processed in exclusive execution mode (XX)
  - — UDP packets are processed in concurrent execution mode (XC). In order to keep the UDP packets in order, the application must transition to exclusive execution mode before sending packets to the destination port.
  - — SCTP packets are processed with no respect regarding their arrival order (Unordered).
  - — The packets not matching the classification criteria are dropped.

In order to determine the processing mode in which a packet should be processed, when a lookup hit is obtained, the lookup result will return the new processing mode to which the application should transition

- Switch the source and destination MAC and IP addresses of the received packets.

Reflect back the accepted packets on the same interface from which they were received

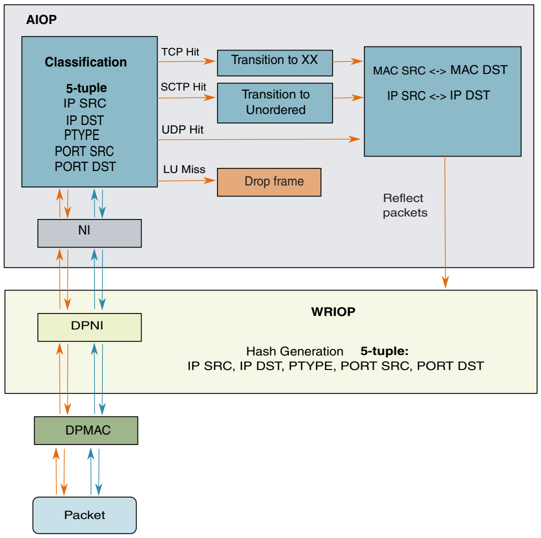


Figure 164. AIOP Packet Classifier overview

The Exact Match table in AIOP that will be used for table lookup will be populated with the following entries:

IP Source Address	IP Destination Address	Protocol Type	Source Port	Destination Port
198.20.1.1	198.19.1.0	6 (TCP)	1024	1025
198.20.1.1	198.19.1.64	17 (UDP)	1024	1025
198.20.1.1	198.19.1.128	132 (SCTP)	1024	1025

For each entry in the table a mask with value 0xE0 is applied on the last byte of the IP Destination address, allowing a number of 32 IP Destination Address to be received. The following traffic flows will generate a HIT in the Classification table:

IP Source Address	IP Destination Address	Protocol Type	Source Port	Destination Port
198.20.1.1	198.19.1.0 .. 198.19.1.31	6 (TCP)	1024	1025
198.20.1.1	198.19.1.64 .. 198.19.1.95	17 (UDP)	1024	1025
198.20.1.1	198.19.1.128 .. 198.19.1.159	132 (SCTP)	1024	1025

8.3.7.1.3.2 Running the Classifier Application

The classifier application should be run on the RDB-1 system.

The ELF for AIOP Classifier is `/usr/aiop/bin/aiop_classifier.elf`. This can be loaded via AIOP tool as section Creating AIOP Containers describes. For this application the script `dynamic_aiop_only.sh` must be executed before loading the image:

```
# ./dynamic_aiop_only.sh
```

To load the AIOP Classifier Application, execute the following command:

```
# aiop_tool load -g dprc.3 -f /usr/aiop/bin/aiop_classifier.elf &
```

To check if the AIOP Classifier Application loaded successfully, execute the following command in the Linux command shell:

```
# cat /dev/fsl_aiop_console | grep CLASSIFIER
```

The command output should display information about the Network Interfaces (NIs) that were successfully configured: NI instance (e.g NI 0) together with the associated MAC address.

```
CLASSIFIER : Successfully configured exact table match
CLASSIFIER : Successfully configured ni0 (dpni.2)
CLASSIFIER : dpni.2 <---connected---> dpmac.2 (MAC addr: 00:00:00:00:00:07)
CLASSIFIER : Successfully configured ni1 (dpni.1)
CLASSIFIER : dpni.1 <---connected---> dpmac.1 (MAC addr: 00:00:00:00:00:06)
```

NOTE

Although the AIOP container contains multiple NIs the AIOP Reflector will use only the NIs that have a DPMAC as endpoint (on which it can succesfully configure the order scope).

During frame processing, the AIOP Logger will print the following brief information about every reflected packet:

- AIOP Core number on which the frame was processed
- Received MAC source and destination addresses
- Received IP source and destination addresses

```
# busybox tail –f /dev/fsl_aiop_console

RX on NI 0 | CORE:15
   MAC_SA: 00-10-94-00-00-02 MAC_DA: 00-00-00-00-00-06
   IP_SRC: 198.20.1.1 IP_DST: 198.19.1.0

RX on NI 0 | CORE:14
   MAC_SA: 00-10-94-00-00-02 MAC_DA: 00-00-00-00-00-06
   IP_SRC: 198.20.1.1 IP_DST: 198.19.1.64

. . .

RX on NI 1 | CORE:9
   MAC_SA: 00-10-94-00-00-02 MAC_DA: 00-00-00-00-00-07
   IP_SRC: 198.20.1.1 IP_DST: 198.19.1.128
```

8.3.7.1.3.3 Generating Traffic to Test AIOP Classifier Application

An external traffic source is needed to pass traffic to the classifier application. This could be another RDB or some other traffic generator. Two optical 10G ports need to be connected to MAC1 and MAC2 of RDB-1.

The following graphic illustrates how traffic moves from a packet generator to RDB-1 board.

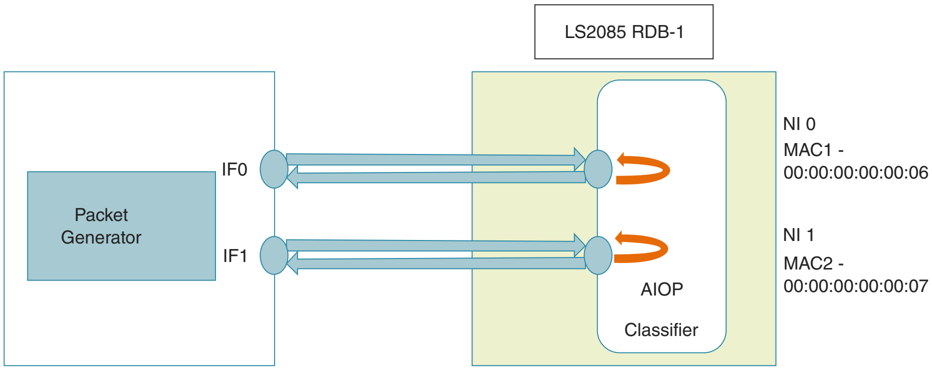


Figure 165. Generating traffic for AIOP Classifier

When using a Packet Generator to inject traffic, connect one port of the packet generator to MAC1 of RDB-1 and a second port to MAC2 of RDB-1.

On RDB-1 ensure that both AIOP network interfaces are in link up state:

```
$ cat /dev/fsl_aiop_console | grep CLASSIFIER
...
CLASSIFIER : ni0 link is UP
CLASSIFIER : ni1 link is UP
```

Generate IPv4 frames for each of the two ports:

- MAC source: ANY
- MAC destination: ANY
- For IP & Layer 4, the accepted values are show in the following table:

IP Source Address	IP Destination Address	Protocol Type	Source Port	Destination Port
198.20.1.1	198.19.1.0 .. 198.19.1.31	6 (TCP)	1024	1025
198.20.1.1	198.19.1.64 .. 198.19.1.95	17 (UDP)	1024	1025
198.20.1.1	198.19.1.128 .. 198.19.1.159	132 (SCTP)	1024	1025

The AIOP Logger will print a brief information about every frame that passed the Classification criteria.

NOTE

Traffic can be generated using frames available in the provided classifier.pcap file. The traffic available is shown in the table below.

IP Source Address	IP Destination Address	Protocol Type	Source Port	Destination Port
198.20.1.1	198.19.1.0 .. 198.19.1.63	6 (TCP)	1024	1025
198.20.1.1	198.19.1.64 .. 198.19.1.127	17 (UDP)	1024	1025
198.20.1.1	198.19.1.128 .. 198.19.1.191	132 (SCTP)	1024	1025

This file can be used to generate the traffic from a traffic generator (e.g.: a packet generator or another RDB board). Location on board for this file, after bringup, is in `/usr/aiop/traffic_files/classifier.pcap`. On RDB-1, the AIOP Reflector Classifier Application will drop half of the frames in each range as a result of classification look-up miss.

8.3.7.1.4 AIOP Control Flow Application

8.3.7.1.4.1 AIOP Control Flow Overview

The purpose of this sample application is to demonstrate a simple AIOP-GPP communication using a DPNI-DPNI connection. The application filters ICMP and ARP request frames and sends them to GPP where Linux Kernel replies with ICMP Echo Reply and ARP Response.

Application has the following required connections:

- DPNI to DPNI connection: a network interface from AIOP that provides connection to another network interface on GPP
- DPNI to DPMAC connection: a network interface from AIOP connected to external traffic (physical wired connection, e.g. XFI or SGMII as links)

The application performs the following functions:

Initialization:

All network interfaces are in promiscuous mode.

Depending on a network interface's' connected endpoint object type, there are two different frame processing callbacks.

NOTE

Failures in performing the above actions lead to dropping the packets received on that network interface or in no link between AIOP and GPP.

Runtime

For every packet received on AIOP network interface connected to external traffic:

- Drop non IPv4 packets and non ARP packets

- Detect, using Parser, if packets received are either ICMP Echo Request or ARP Request. In this case send packets to Linux Kernel using GPP connection
- For ARP or IPv4 packets other than ICMP Echo Request and ARP Request: switch the source and destination MAC and IP addresses of the received packets and reflect back the packet on the same interface from which it was received.

For every packet received from GPP:

- Drop all packets that are not ICMP Echo Reply and not ARP response
- Print brief information about the accepted packets
- Forward ICMP Echo Reply and ARP response packets to the NI havingconnected to traffic generatorDPMAC as endpoint and callback config and callback configured.

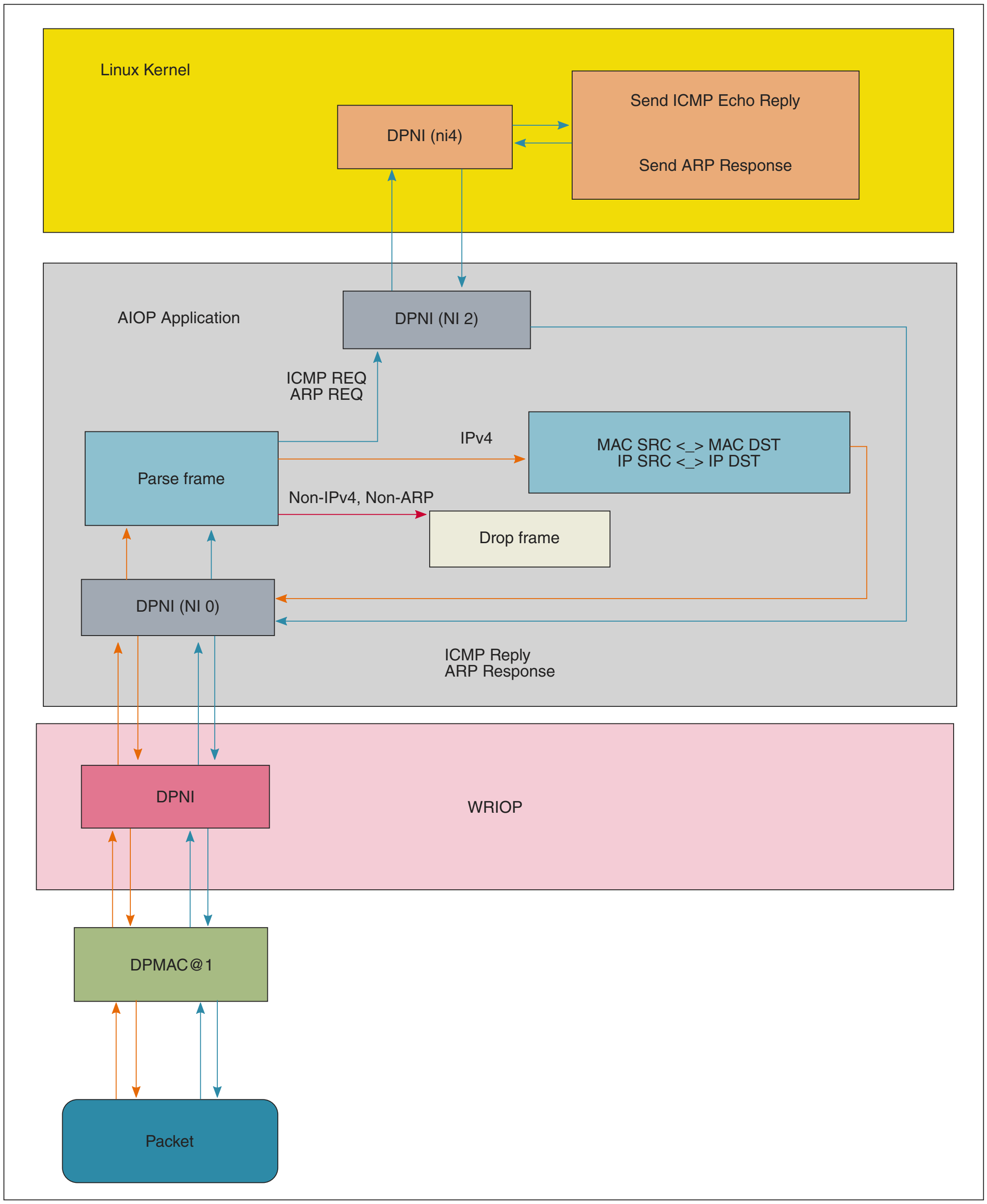


Figure 166. AIOP Control Flow overview

8.3.7.1.4.2 Running the Control Flow Application

The ELF for AIOP Classifier is `/usr/aiop/bin/aiop_control_flow.elf`. This can be loaded via AIOP tool as section Creating AIOP Containers describes. For this application the script `dynamic_aiop_root.sh` must be executed before loading the image:

```
# ./dynamic_aiop_root.sh
```

To load the AIOP Classifier Application, execute the following command:

```
# aiop_tool load -g dprc.3 -f /usr/aiop/bin/aiop_control_flow.elf &
```

To check if AIOP Control Flow Application loaded successfully execute the following command in the Linux command shell:

```
$ cat /dev/fsl_aiop_console | grep CONTROL_FLOW
```

The command output should display the number of NIs that were successfully configured together with the specific MAC addresses that are provided to the AIOP Control Flow Application:

```
CONTROL_FLOW : Successfully configured ni0 (dpni.2)
CONTROL_FLOW : dpni.2 <---connected---> dpni.0
CONTROL_FLOW : Successfully configured ni1 (dpni.1)
CONTROL_FLOW : dpni.1 <---connected---> dpmac.1 (MAC addr: 00:00:00:00:00:06)
```

On RDB board, to enable AIOP-GPP communication via DPNI-DPNI, it is required to configure the Linux network interface:

```
# ip link set dev ni0 down
# ip addr flush dev ni0
# ip addr add 6.6.6.1/8 dev ni0
# ip link set dev ni0 up
```

After executing the commands for configuring Linux network interface, in AIOP console will be displayed a message for the NI 0 (DPNI 2) link up due to a DPNI link event:

```
CONTROL_FLOW : ni0 link is UP
```

It is also required to create a static entry in the ARP table, so that Linux responds directly to AIOP:

```
# arp -s 6.6.6.10 <MAC_address_of_traffic_generator>
# arp -n
Address     HWtype     HWaddress                             Flags Mask     Iface
6.6.6.10    ether      <MAC_address_of_traffic_generator>    CM             ni4
```

During frame processing, the AIOP Logger will print the following brief information about every received packet on all interfaces:

- Received MAC source and destination addresses
- Received IP source and destination addresses
- Traffic generator IP address is 6.6.6.10
- For ARP Frames AIOP Logger will print: operation code (OPCODE), Sender and Target Protocol Address.
- For ICMP Frames AIOP Logger will print: ICMP Type and ICMP Code.

```
RX on NI 0
   MAC_SA: 00-00-00-00-00-02 MAC_DA: 02-00-c0-a8-48-01
   IP_SRC: 6.6.6.10 IP_DST: 6.6.6.1
   ICMP_TYPE: 8 ICMP_CODE: 0

RX on NI 0
   MAC_SA: 00-00-00-00-00-02 MAC_DA: 02-00-c0-a8-48-01
   ARP_OPCODE: 1 S_ADDR: 6.6.6.10 T_ADDR: 6.6.6.1

RX on NI 0
   MAC_SA: 00-00-00-00-00-06 MAC_DA: 00-10-94-00-00-10
   IP_SRC: 192.0.0.1 IP_DST: 192.85.1.2

. . .
```

Assign an IP address to the Linux interface for SSH connections to RDB-1 board, for example using the following command:

```
# ip addr add 192.168.1.20/24 dev eth0
# ip link set dev eth0 up
```

Alternatively, if the DHCP server is active on the network, run following command to get an IP address automatically, for example using the following command:

```
# udhcpc -i eth0
```

8.3.7.1.4.3 Generating Traffic to Test AIOP Control Flow Application

An external traffic source is needed to pass traffic to the control flow application. This could be another RDB or some other traffic generator. The following graphic illustrates traffic flow between Packet Generator and RDB board.

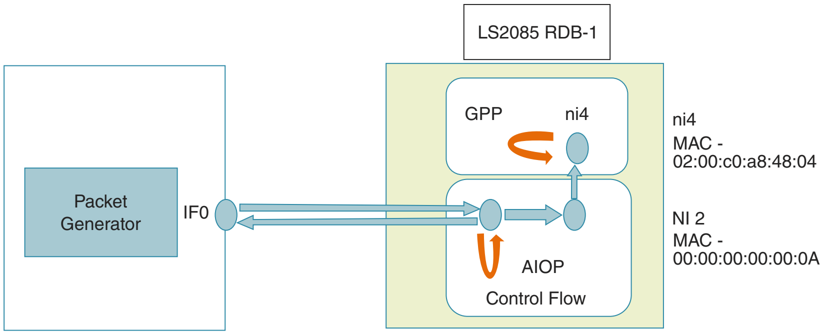


Figure 167. Generating traffic for AIOP Control Flow

When using a Packet Generator to inject traffic, connect one port of the packet generator to MAC1 of RDB-1.

Generate IPv4 frames for each of the two ports:

- MAC source: 00:00:00:00:00:02 (MAC address of the traffic generator)
- MAC destination: 02:00:C0:A8:48:00 (MAC of ni0 in Linux)
- For ARP and ICMP, the values that are sent to Linux GPP are:

IP Source Address	IP Destination Address	Protocol Type	ICMP Type	ICMP Code
6.6.6.10	6.6.6.1	01 (ICMP)	08 (Echo Request)	00

For other ICMP fields use any value:

Protocol Type	Sender Hardware Address	Sender Protocol Address	Target Hardware Address	Target Protocol Address
00 01 (ARP Request)	<<MAC Address of Traffic Generator>>	6.6.6.10	00:00:00:00:00:00	6.6.6.1

- For TCP or UDP use any data in IP source/destination and header specific.

Traffic can be injected in board only after the network interface connected to trafficgenerator (external traffic) is up due to internal event in AIOP:

```
CONTROL_FLOW : ni0 link is UP
```

Open two AIOP consoles (using SSH). On one of them show the AIOP Logger using command described below. When injecting traffic, on RDB-1, the AIOP Logger will print a brief information about every frame that is being processed.

```
$ busybox tail -f /dev/fsl_aiop_console
```

On the other console start packet capture, using the following command, to sniff communication between AIOP and GPP:

```
$ tcpdump -i ni1 -w aiop_control_flow.pcap
```

8.3.7.1.5 AIOP Header Manipulation Application

8.3.7.1.5.1 AIOP Header Manipulation Overview

The purpose of this application is to demonstrate how to perform a simple GRE tunneling using header manipulation operations available in AIOP. Application offers support only for IPv4 packets and requires one DPNI interface, acting as tunnel interface. The packet header content is used to determine the type of operation to be performed on the packet: encapsulation or decapsulation

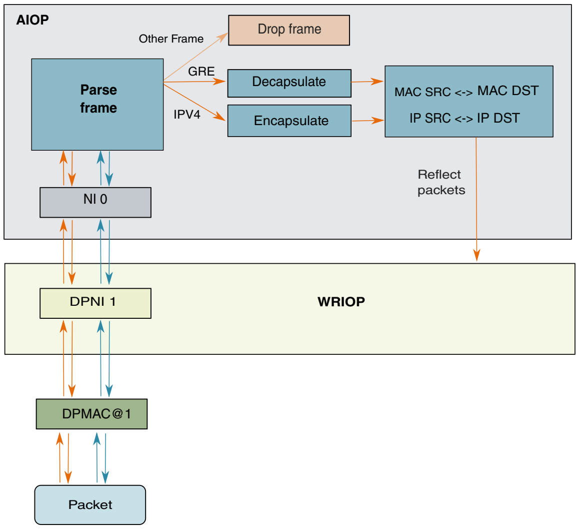


Figure 168. AIOP Header Manipulation overview

The application performs the following operations:

- For regular IPv4 packets without inner IP header: creates a basic IP header, computes outer IP checksum, creates a basic GRE header (all flags 0, GRE version 0, encapsulated protocol: IPv4), inserts these headers between ETH and IP headers and switches the source and destination MAC addresses
- For packets having an inner IP and GRE headers: deletes the outer IP and GRE headers, switches the source and destination MAC and IP addresses
- Discards non-IPv4 and non-GRE packets
- Re-runs parser and prints the packet after header manipulation occurred. This is optional and can be deactivated by defining the macro RERUN_PARSER
- Sends back packets on same network interface they were received from.

8.3.7.1.5.2 Running the Header Manipulation application

The ELF for AIOP Classifier is `/usr/aiop/bin/aiop_header_manip.elf`. This can be loaded via AIOP tool as described in Creating AIOP Containers on page 787. For this application the script `dynamic_aiop_only.sh` must be executed before loading the image:

```
# ./dynamic_aiop_only.sh
```

To load the AIOP Classifier Application, execute the following command:

```
# aiop_tool load -g dprc.3 -f /usr/aiop/bin/aiop_header_manip.elf &
```

To check if the AIOP Classifier Application loaded successfully, execute the following command in the Linux command shell:

```
# cat /dev/fsl_aiop_console | grep HEADER_MANIP
```

The command output should display information about the Network Interfaces (NIs) that were successfully configured: NI instance (e.g ni0) together with the associated MAC address.

```
HEADER_MANIP : Successfully configured ni0 (dpni.6)
HEADER_MANIP : dpni.6 <---connected---> dpmac.1 (MAC addr: 00:00:00:00:00:06)
```

NOTE

Although the AIOP container contains multiple NIs the AIOP Reflector will use only the NIs that have a DPMAC as endpoint (on which it can succesfully configure the order scope).

During packet processing, the AIOP Logger prints the following brief information about every received packet on all interfaces:

- Received IP source and destination addresses (for the inner header and for the outer IP header in case packet is encapsulated)
- GRE Flags, version and encapsulated protocol, in case the packet is encapsulated.

If parser re-running was enabled the information above is printed, in order to show the result of encapsulation/decapsulation header manipulation operations performed.

```
# busybox tail -f /dev/fsl_aiop_console

Decapsulated Frame | FD Len 124 | SEG Len 124
  OUTER IP HEADER
    PROTO: 1 IP_SRC: 192.168.1.10, IP_DST: 10.171.77.121, TOTAL LEN: 110, IHL: 20

Encapsulated Frame | FD Len 148 | SEG Len 124
  OUTER IP HEADER
    PROTO: 47 IP_SRC: 122.122.122.122, IP_DST: 138.138.138.138, TOTAL LEN: 134, IHL: 20
  Generic Routing Encapsulation
    FLAGS: 0x00000, VERSION: 0x000, PTYPTE: 0x0800
  INNER IP HEADER
    PROTO: 1 IP_SRC: 192.168.1.10, IP_DST: 10.171.77.121, TOTAL LEN: 110, IHL: 20

Encapsulated Frame | FD Len 148 | SEG Len 128
  OUTER IP HEADER
    PROTO: 47 IP_SRC: 10.8.8.8, IP_DST: 10.7.7.7, TOTAL LEN: 134, IHL: 20
  Generic Routing Encapsulation
    FLAGS: 0x00000, VERSION: 0x000, PTYPTE: 0x0800
  INNER IP HEADER
    PROTO: 6 IP_SRC: 198.168.2.21, IP_DST: 10.18.1.1, TOTAL LEN: 106, IHL: 20

Decapsulated Frame | FD Len 120 | SEG Len 120
  OUTER IP HEADER
    PROTO: 6 IP_SRC: 10.18.1.1, IP_DST: 198.168.2.21, TOTAL LEN: 106, IHL: 20
```

8.3.7.1.5.3 Generating Traffic to Test AIOP Header Manipulation Application

An external traffic source is needed to pass traffic to the classifier application. This could be another RDB or some other traffic generator. One optical 10G ports need to be connected to MAC1 or MAC2 of RDB-1.

The following diagram illustrates how traffic moves from a packet generator to RDB-1 board.

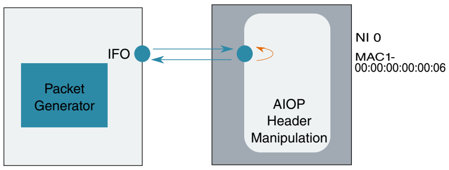


Figure 169. Generating traffic for AIOP Header Manipulation

When using a Packet Generator to inject traffic, connect one port of the packet generator to MAC1 of RDB-1.

On RDB-1 ensure that both AIOP network interface is in link up state:

```
$ cat /dev/fsl_aiop_console | grep HEADER_MANIP
HEADER_MANIP: NI 0 link is UP
```

Generate IPv4 packets:

- Common values:
  - — MAC destination: 00:00:00:00:00:06 (address of MAC1)
  - — MAC source: ANY
  - — Outer IP source: ANY
  - — Outer IP destination: ANY
- For GRE encapsulated packets:
  - — Version: 0
  - — GRE Flags and version: 0x0000
  - — GRE Encapsulated Protocol: 0x0800 (IP)
  - — Inner IP source: ANY
  - — Inner IP destination: ANY

On RDB-1, the AIOP Logger prints brief information about every packet that is being encapsulated/decapsulated and if parser re-running is enabled the AIOP Logger prints brief information about the packet after the GRE encapsulation/decapsulation header manipulations occurred.

The encapsulated packets using GRE by AIOP application have the following pre-defined values:

- Outer IP Version: 4
- Outer IP source: 122.122.122.122
- Outer IP destination: 138.138.138.138
- GRE Version: 0
- GRE Flags and version: 0x0000
- GRE Protocol Type: 0x0800 (IP)
- Inner IP source: IP source from original packet
- Inner IP destination: IP destination from original packet

The packets that were decapsulated by AIOP will have the inner IP source and destination addresses and MAC addresses swapped.

8.3.7.1.6 AIOP Statistics Application

8.3.7.1.6.1 AIOP Statistics Overview

The purpose of this application is to demonstrate the usage of the AIOP atomic operations in order to update software definedstatistics counters. There are two types of statistics counters:

- Global: total number of the received packets, total number of the received bytes , total number of the accepted packets, total number of the rejected packets
- Per-flow: total number of the recived packets and total number of the received bytes, for each packet matching the applied classification criteria

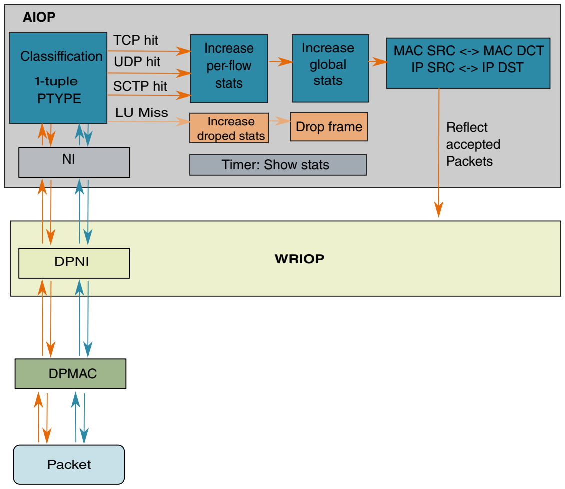


Figure 170. AIOP Statistics overview

The application performs the following operations:

- Classifies the packets to enable different counters update. Classification is based only on the Protocol Number field. Only the TCP, UDP and SCTP protocols are accepted. The packets not matching the classification criteria are dropped and number of dropped packets is increased. [GI1]
- Increases the global statistics and the statistics for each packet matching the classification criteria (per-flow statistics).
- Swap the MAC and IP addresses
- Sends back packets on same network interface they were received from.
- By default, the global and per flow statistics are displayed at every 60 seconds. In order achieve this, a timer is used. The timer time interval, measured in seconds, can be changed by setting the value of the macro definition APP_TMAN_TIMER_DURATION, at compile time.

Note: Both global and per-flow statistics are stored in the DP-DDR (Data Path DDR) memory partition. For the global statistics, a contiguous block of memory is obtained and for each flow a buffer from DP-DDR is obtained by using pool-based allocation.

8.3.7.1.6.2 Running the AIOP Statistics Application

The statistics application should be run on the RDB-1 system.

The ELF for AIOP reflector is `/usr/aiop/bin/aiop_statistics.elf`. This can be loaded via AIOP tool as section Creating AIOP Containers describes. For this application the script `dynamic_aiop_only.sh` must be executed before loading the image:

```
# ./dynamic_aiop_only.sh
```

To load the AIOP Statistics Application, execute the following command:

```
# aiop_tool load -g dprc.3 -f /usr/aiop/bin/aiop_statistics.elf &
```

To check if the AIOP Statistics Application loaded successfully, execute the following command in the Linux command shell:

```
# cat /dev/fsl_aiop_console | grep STATISTICS
```

The command output should display information about the Network Interfaces (NIs) that were successfully configured: NI instance (e.g ni0) together with the associated MAC address.

```
STATISTICS : Created TMI id=0x2
STATISTICS : Created timer for showing statistics, handle=0x102
STATISTICS : Application initialized successfully
STATISTICS : Successfully configured ni0 (dpni.2)
STATISTICS : dpni.2 <---connected---> dpmac.2 (MAC addr: 00:00:00:00:00:07)
STATISTICS : Successfully configured ni1 (dpni.1)
STATISTICS : dpni.1 <---connected---> dpmac.1 (MAC addr: 00:00:00:00:00:06)
STATISTICS : ni1 link is UP
```

Once the application started, at every 60 seconds (or at the time interval defined by APP_TMAN_TIMER_DURATION macro) the AIOP Logger prints all statistics counters values. Default value should be 0 for all counters.

```
AIOP received 47753801 packets (5921471324 bytes)
        12585359 dropped packets, 35168442 accepted packets, 35168442 transmitted packets
         * PROTO=0x6: received: 12601479 packets (1562583396 bytes)
         * PROTO=0x11: received: 12608080 packets (1563401920 bytes)
         * PROTO=0x84: received: 9958883 packets (1234901492 bytes)
```

8.3.7.1.6.3 Generating Traffic to Test AIOP Statistics Application

An external traffic source is needed to inject packets into the application. This could be another RDB-1 or some other traffic generator. One optical 10G port needs to be connected to MAC1 or MAC2 port of the RDB

The following diagram illustrates how traffic moves from a packet generator to RDB-1 board.

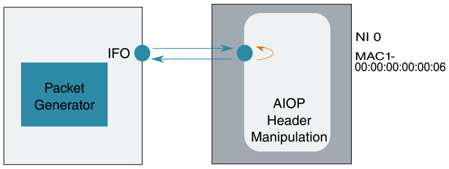


Figure 171. Generating traffic for AIOP Statistics

When using a Packet Generator to inject traffic, connect one port of the packet generator to MAC1 of RDB-1 and a second port to MAC2 of RDB-1.

On RDB-1 ensure that both AIOP network interfaces are in link up state:

```
$ cat /dev/fsl_aiop_console | grep STATISTICS
STATISTICS : NI 0 link is UP
```

Generate IPv4 frames for each of the two ports:

- MAC source: ANY
- MAC destination: 00:00:00:00:00:06 (address of MAC1)
- For IP & Layer 4, the accepted values are show in the following table:

IP Source Address	IP Destination Address	Protocol Type	Source Port	Destination Port
ANY	ANY	6 (TCP)	ANY	ANY
ANY	ANY	17 (UDP)	ANY	ANY
ANY	ANY	132 (SCTP)	ANY	ANY

On RDB-1, the AIOP Logger prints at every 60 seconds (or at the time interval configured with the APP_TMAN_TIMER_DURATION macro) the global statistics and the per-flow statistics for each matched protocol (TCP, UDP and SCTP).

8.3.7.1.7 AIOP QoS_demo Application

8.3.7.1.7.1 AIOP QoS_demo Overview

The purpose of this sample application is to demonstrate how to use the QoS features on the AIOP. The application is a basic reflector: a received frame is sent back unchanged on the same dpni.

The application performs the following functions:

- For dpni0, it doesn't enable QoS features
- Configures QoS features on dpni1 for ingress (classification, prioritization, policing) and egress (traffic shaping)
- Traffic is classified in 4 classes: TCP is TC0, UDP is TC1, SCTP is TC2, all the rest of traffic goes to TC7
- The priorities of the traffic classes are set as follows: TC0 has high priority 0, TC1 has high priority 1, TC2 has medium priority, TC7 has lowest priority
- The policer is configured to discard TC2 traffic
- The tx rate limit is set to 100 Mbps
- Prints information about the received frame (interface id, traffic class, protocols)
- Reflects back the frame on the same interface from which it was received

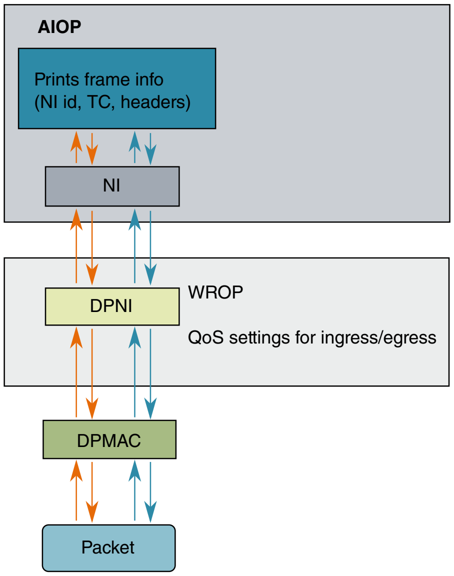


Figure 172. AIOP qos_demo Overview

8.3.7.1.7.2 Running the QoS_demo Application

The qos_demo should be run on the RDB-1 system.

The ELF for AIOP QoS_demo is `/usr/aiop/bin/ aiop_qos_demo.elf`. This can be loaded via AIOP tool as section Creating AIOP Containers describes. For this application, the script `dynamic_aiop_only.sh` must be executed before loading the image:

```
# ./dynamic_aiop_only.sh
```

To load the AIOP QoS_demo, execute the following command:

```
# aiop_tool load -g dprc.3 -f /usr/aiop/bin/ aiop_qos_demo.elf &
```

To check if the AIOP QoS_demo Application loaded successfully, execute the following command in the LS2088 Linux command shell:

```
# cat /dev/fsl_aiop_console | grep QoS_demo
```

The command output should display information about the Network Interfaces (NIs) that were successfully configured: NI instance (e.g NI 0) together with the associated MAC address.

```
QoS_demo : Successfully configured exact table match
QoS_demo : Successfully configured ni0 (dpni.2)
QoS_demo : dpni.2 <---connected---> dpmac.2 (MAC addr: 00:00:00:00:00:07)
QoS_demo : Successfully configured ni1 (dpni.1)
QoS_demo : dpni.1 <---connected---> dpmac.1 (MAC addr: 00:00:00:00:00:06)
```

During frame processing, the AIOP Logger will print the following brief information about every reflected packet:

- NI id on which the frame was processed
- AIOP Core number on which the task executes
- Traffic class for the incoming frame
- Header information

```
# busybox tail -f /dev/fsl_aiop_console

QoS_demo: RX on NI 1 | CORE:0 | TC = 7 | ARP | unknown

QoS_demo: RX on NI 1 | CORE:0 | TC = 7 | IPv4 | ICMP

QoS_demo: RX on NI 1 | CORE:0 | TC = 0 | IPv4 | TCP

QoS_demo: RX on NI 1 | CORE:0 | TC = 1 | IPv4 | UDP

QoS_demo: RX on NI 0 | CORE:0 | TC = 0 | ARP | unknown

QoS_demo: RX on NI 0 | CORE:0 | TC = 0 | IPv4 | ICMP

QoS_demo: RX on NI 0 | CORE:0 | TC = 0 | IPv4 | TCP

QoS_demo: RX on NI 0 | CORE:0 | TC = 0 | IPv4 | UDP

QoS_demo: RX on NI 0 | CORE:0 | TC = 0 | IPv4 | SCTP
```

8.3.7.1.7.3 Generating traffic to test AIOP QoS_demo Application

An external traffic source is needed to pass traffic to the qos_demo application. This could be another RDB or some other traffic generator. Two optical 10G ports need to be connected to MAC1 and MAC2 of RDB-1.The following graphic illustrates how traffic moves from a packet generator to RDB-1 board.

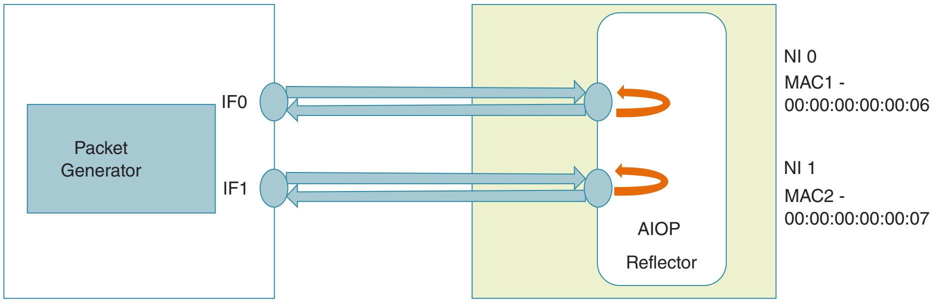


Figure 173. Generating Traffic for AIOP qos_demo

When using a Packet Generator to inject traffic, connect one port of the packet generator to MAC1 of RDB-1 and a second port to MAC2 of RDB-1.

On RDB-1 ensure that both AIOP network interfaces are in link up state:

```
$ cat /dev/fsl_aiop_console | grep QoS_demo
QoS_demo : ni0 link is UP
QoS_demo : ni1 link is UP
```

Generate different type of traffic for each of the two ports: ARP, ICMP, IP, TCP, UDP, SCTP. On RDB-1, the AIOP Logger will print a brief information about every frame that is received. On ni1, the policer discards TC2 traffic (SCTP) so there are no SCTP packets received by AIOP on ni1.

8.3.7.2 AIOP Tool User's Guide

8.3.7.2.1 Introduction

The following section contains information on compilation and execution steps for the AIOP Tool application. This section is a part of the series intended to help developers use DPAA2 software on NXP’s LS family of network processors. This section intended to get users up and running quickly.

8.3.7.2.2 DPAA2 Software

For information about the DPAA2 software, see the DPAA2 Software Overview on page 673 section. For users that are unfamiliar with DPAA2 software, it is recommended to read that content before proceeding.

8.3.7.2.3 Product Description

8.3.7.2.3.1 Overview

What is AIOP?

AIOP or Advanced I/O Processor is an optional component of the DPAA2 architecture. It is a C-programmable engine that is optimized for packet processing and can be used on a SoC in conjunction with the general purpose cores and the other elements of the DPAA2 architecture. AIOP executes a software image containing the packet processing logic.

What is AIOP Tool?

AIOP Tool is a Linux user space application which performs the following operations on an AIOP:

1. Loading an image on an AIOP
2. Starting (running) the image on an AIOP after it has been loaded
3. Extracting state information for an AIOP
4. Getting and setting time of day on an AIOP

8.3.7.2.3.2 Product features

AIOP Tool provides following broad functions:

- Provides a command line interface for executing a set of operations
    - Command line will accept an operation from user, along with the appropriate corresponding arguments.
    - The DPRC to work on is also accepted as an argument on the command line.
- Operation for Loading AIOP Image to an AIOP
    - Command line will accept an AIOP image file.
    - Whether a reset of the AIOP should be done before loading or not can be controlled using a toggle argument passed on command line. For performing reset of AIOP before load operation, an optional toggle argument can be passed to the load operation.
        - Current hardware revision (rev1) doesn’t support reset and it is expected to work in subsequent releases. Thus, in future, the same tool can be used without any modifications.

    - — Result of load API call will be provided to the caller.
- Operation for Resetting the AIOP
    - — On execution, this would put the AIOP in Reset state for loading another image.
        - ◦ In the current hardware revision (rev1), this is not supported and would result in error. In subsequent versions of LS family hardware, this is expected to be operational. Once that is done, with the combination of load and run operations, multiple images can be loaded on AIOP (serially) without a hardware reset.
    - — Response value of API execution will be provided to caller.
- Operation for getting and setting time of day on AIOP
    - — Once an AIOP is running, 2 operations for getting and setting the time of day on AIOP are available.
    - — Success or failure of the API calls would be provided to caller in case of time setting. In case of time getting operation, time since epoch would be shown.
- Operation for printing AIOP status
    - — On execution of this operation, status of the AIOP is provided to the caller. This includes the current state and version information.
- Help menu will be printed by command line for displaying usage of above operations and their corresponding arguments.

8.3.7.2.4 System Requirements

This chapter describes the environment requirements for executing the AIOP Tool.

8.3.7.2.4.1 Environment required

1. MC Firmware version 9.0.x for LS2085A (RDB and QDS) EAR-6.0
2. Uboot and Linux compatible with LS2085A (RDB and QDS) EAR-6.0
3. AIOP image (ELF) is required as input to the tool

8.3.7.2.5 AIOP Tool Usage

8.3.7.2.5.1 Run time pre-requisites

Besides the environment pre-requisites listed in Environment required on page 808, the following execution time pre-requisites exist:

1. AIOP Tool requires a DPRC which has an AIOP included in it. It should be a non-root DPRC containing at least a dpmcp and dpaiop object.
    - This DPRC can be statically defined (static DPL) or dynamically created (using `restool`).
    - Please refer Steps For Dynamic DPRC Suitable For AIOP Tool Using restool on page 814 below for steps for dynamically creating a suitable DPRC using `restool`.
    - Also, refer `dynamic_AIOP_dpl.sh` script delivered as part of the LSDK 1709 release. This is a script version of dynamic DPRC creation steps mentioned in Steps For Dynamic DPRC Suitable For AIOP Tool Using restool on page 814.
2. VFIO support in Linux kernel
    - AIOP is a hardware device which AIOP Tool attempts to access from Linux userspace. For exchanging information between AIOP Tool and AIOP, a secure memory area is required which can be accessed by both. Linux VFIO or Virtual Function I/O, provides a framework for securely exposing certain memory area which is equally accessible by AIOP Tool and AIOP hardware. VFIO would map the addressing understood by AIOP hardware to that of addressing managed by user-space application.

- This application assumes that the `dpaiop` device would be mapped into VFIO driver so as to use DMA mapping between AIOP Tool and AIOP hardware and for pushing SMMU addresses for AIOP hardware's access to user-space memory
- Refer Sample VFIO Binding Script on page 813 for a sample script to perform binding of a DP object with VFIO.

3. Valid AIOP image for loading
    - For sample, refer to `cmdif_integ_dbg.elf` image provided along with '`aiopsl`' repo.

8.3.7.2.5.2 Environment setting

For executing the AIOP Tool, a valid DPRC is required. There are three ways to define a DPRC:

1. Provide DPRC through '`-g`' command line tool. For e.g., `aiop_tool <sub-command> -g dprc.4`.
2. If not provided through command line, the binary expects the environment variable `DPRC` defined.
3. If neither an argument nor environment variable is provided, binary assumes a default value of DPRC as `dprc.5`. (Current hard-coded in the code)

8.3.7.2.5.3 Command line arguments

All variants of the execution follow a standard pattern:

```
aiop_tool <sub-command> <Arguments>
```

`<Sub-command>` includes `help`, `load`, `gettod`, `settod`, `status` and `reset`.

`<Arguments>` are either mandatory (for e.g., image file path), or optional (for e.g., DPRC name).

All variants assume that the VFIO binding has already been performed on the command line for the DPRC on which command would be executed. Further, following samples/snippets assume that DPRC would be passed on command line, though all three methods mentioned in Environment setting on page 809 are valid.

Following table describes all the variants of the AIOP Tool execution.

Argument/command pattern	Description
`aiop_tool` **`help`**	Displays help including various sub-commands and their output.

Table continued from the previous page...

<table>
<tr><th>Argument/command pattern</th><th>Description</th></tr>
<tr><td><code>aiop_tool <b>load</b></code></td><td>Loading a valid ELF image onto the AIOP and starting it.
Valid arguments are:
<table>
<tr><td>-g &lt;DPRC name&gt;</td><td>Name of the DPRC containing the AIOP Object. Th
Optionally, use <code>--container=</code>.</td></tr>
<tr><td>-f &lt;AIOP image file&gt;</td><td>[Mandatory] Name, with path, of the AIOP image fil
Optionally, use <code>--file=</code>.</td></tr>
<tr><td>-a &lt;AIOP argument file&gt;</td><td>[Mandatory] A file containing arguments to be pass
Optionally, use <code>--args-file=</code>.</td></tr>
<tr><td>-c &lt;Threads per AIOP Core&gt;</td><td>[Optional] Threads per AIOP Core to execute. Defau
Optionally, use <code>--threadpercore=</code>.</td></tr>
<tr><td>-r</td><td>[Optional] Reset toggle; If provided, AIOP Tool will p done.
Optionally, use <code>--reset</code>.</td></tr>
<tr><td>-v</td><td>Verbose Output (More informative with INFO level m
Optionally, use <code>--verbose</code>.</td></tr>
<tr><td>-d</td><td>Debug Output (DEBUG level messages).
Optionally, use <code>--debug</code>.</td></tr>
</table>
Following is the expected output of the command:
• In case of incorrect parameters (incorrect DPRC, wrong file path, or incorrect parameter), failure would be reported along with help usage.
• Success of failure, as reported by MC’s API; the AIOP Tool would convert the error into a readable string.</td></tr>
</table>

Table continues on the next page...

Table continued from the previous page...

<table>
<tr><th>Argument/command pattern</th><th>Description</th></tr>
<tr><td><code>aiop_tool <b>gettod</b></code></td><td>Obtaining the Time of day on the AIOP.
Valid arguments are:
<table>
<tr><td>-g <DPRC name></td><td>Name of the DPRC containing the AIOP Object. Th</td></tr>
<tr><td>-v</td><td>Verbose Output (More informative with INFO level m</td></tr>
<tr><td>-d</td><td>Debug Output (DEBUG level messages)</td></tr>
</table>
Following is the expected output of the command:
<ul>
<li>In case of incorrect parameters (incorrect DPRC), failure would be reported along with usage help.</li>
<li>Time of day (64Bit value) as returned by MC’s API in case of success.
<code>Time of day: 184082</code></li>
<li>Failure, as reported by MC’s API; the AIOP Tool would convert the error into a readable string.</li>
</ul></td></tr>
<tr><td><code>aiop_tool <b>settod</b></code></td><td>Set time on the AIOP.
Valid arguments are:
<table>
<tr><td>-g <DPRC name></td><td>Name of the DPRC containing the AIOP Object. Th</td></tr>
<tr><td>-t sec_since_epoch</td><td>[Mandatory] Time, in milli-seconds, since epoch.</td></tr>
<tr><td>-v</td><td>Verbose Output (More informative with INFO level m</td></tr>
<tr><td>-d</td><td>Debug Output (DEBUG level messages)</td></tr>
</table>
Following is the expected output of the command:
<ul>
<li>In case of incorrect parameters (incorrect DPRC, non-integer time), failure would be reported along with usage help.</li>
<li>Success of failure as reported by MC’s API; the AIOP Tool would convert the error into readable string.</li>
</ul></td></tr>
</table>

Table continues on the next page...

Table continued from the previous page...

<table>
<tr><th>Argument/command pattern</th><th>Description</th></tr>
<tr><td>aiop_tool <b>status</b></td><td>Valid arguments are:
<table>
<tr><td>-g &lt;DPRC name&gt;</td><td>Name of the DPRC containing the AIOP Object. Th</td></tr>
<tr><td>-v</td><td>Verbose Output (More informative with INFO level m</td></tr>
<tr><td>-d</td><td>Debug Output (DEBUG level messages)</td></tr>
</table>
Following is the expected output of the command:
• In case of incorrect parameters (incorrect DPRC), failure would be reported along with usage help.
• Status, including current state, SL version.
AIOP running status:
  Major Version: 2, Minor Version: 1
  Service Layer:- Major Version: 0,
Minor Version: 7, Revision: 0
  State: DPAIOP_STATE_RUNNING
• Success of failure as reported by MC's API; the AIOP Tool would convert the error into readable string.</td></tr>
<tr><td>aiop_tool <b>reset</b></td><td>Reset the AIOP. This is current feature support, dependent on support by hardware.
Valid arguments are:
<table>
<tr><td>-g &lt;DPRC name&gt;</td><td>Name of the DPRC containing the AIOP Object. Th</td></tr>
<tr><td>-v</td><td>Verbose Output (More informative with INFO level m</td></tr>
<tr><td>-d</td><td>Debug Output (DEBUG level messages)</td></tr>
</table>
Following is the expected output of the command:
• In case of incorrect parameters (incorrect DPRC), failure would be reported along with usage help.
• Success of failure as reported by MC's API; the AIOP Tool would convert the error into readable string.</td></tr>
</table>

8.3.7.2.5.4 Command execution samples

1. Obtaining status of AIOP

   ```
   $ aiop_tool status -g dprc.4
   AIOP running status:
     Major Version: 0, Minor Version: 0
     Service Layer:- Major Version: 0, Minor Version: 0, Revision: 0
     State: DPAIOP_STATE_RESET_DONE
   ```

   Before loading the image, for this version where hardware Reset operation is not supported, the state DPAIOP_STATE_RESET_DONE can be verified using above example.

2. Loading an ELF image on AIOP

```
$ aiop_tool load -g dprc.4 -f cmdif_integ_dbg.elf
AIOP Image (cmdif_integ_dbg.elf) loaded successfully.
```

The `load` sub-command loads as well as runs an AIOP. Once the image has been loaded, after a few seconds, the status output would look similar to:

```
$ aiop_tool status -g dprc.4
AIOP running Status:
  Major Version: 2, Minor Version: 1
  Service Layer:- Major Version: 0, Minor Version: 7, Revision: 0
  State: DPAIOP_STATE_RUNNING
```

3. Get time of day from AIOP

   ```
   $ aiop_tool gettod -g dprc.4
   Time of day: 184082
   ```

4. Set time of day on AIOP

   ```
   $ aiop_tool gettod -g dprc.4 -t 100010
   <No output, if successful>
   ```

8.3.7.2.6 Known Limitations

1. It assumed that the DPRC passed to AIOP Tool has a single AIOP (`dpaiop`) and MC portal (`dpmcp`). This application would parse the contents of DPRC and stop on first found instances of `dpaiop` and `dpmcp` – even if multiple instances have been defined. The order of finding would be dependent on how objects are parsed from the sysfs directory.
2. Due to a limitation of the MC API, AIOP Tool instance doesn't automatically exit once executed (not a run-to-completion model). As soon as the AIOP image is successfully loaded, the AIOP Tool application will keep looping without relieving the foreground shell. Following are some consequences of this limitations:
   a. Once executed, the AIOP Tool application sits in foreground without releasing the Linux Shell it ran on. To obtain the Linux shell on which AIOP Tool was executed, AIOP Tool application should be pushed into background (or started in background) through Linux shell semantics.
   b. Once executed and backgrounded, another instance of the AIOP Tool over same DPRC cannot be executed. This also implies that status requests for the AIOP cannot be done, either through same instance or through another instance of the application. To obtain the AIOP status, use other available methods, like `restool`.

8.3.7.2.7 Sample VFIO Binding Script

```
#/*
# * Sample bind script for VFIO.
# */

DPRC_4=/sys/bus/fsl-mc/devices/dprc.4

if [ -e /sys/module/vfio_iommu_type1 ];
then
    echo "#1) Enabling interrupts"
    echo 1 > /sys/module/vfio_iommu_type1/parameters/allow_unsafe_interrupts
else
    echo "No VFIO support available."
    exit
fi

if [ -e $DPRC_4 ];
then
    echo "#1.1) dprc container driver override"
    echo vfio-fsl-mc > /sys/bus/fsl-mc/devices/dprc.4/driver_override
    echo "#1.2) Bind dprc.4 to VFIO driver"
    echo dprc.4 > /sys/bus/fsl-mc/drivers/vfio-fsl-mc/bind
fi

if [ -e /dev/vfio ];
then
```

```
    ls /dev/vfio/
else
    echo "No VFIO support available."
fi
```

8.3.7.2.8 Steps For Dynamic DPRC Suitable For AIOP Tool Using restool

Assuming that a DPL is available which has AIOP container defined in it, this section provides information and sample commands for creating a DPRC which is suitable for use with the AIOP Tool.

AIOP Tool expects a DPRC which contains at least one dpmcp and a dpaiop object. Further, the DPRC should be a non-root DPRC – so that user-space application (AIOP Tool) can access its content.

All the commands would be performed using `restool` compatible with the MC version for which AIOP Tool is targeted. The sample steps below assume that there are enough resources to create a MC portal (`dpmcp`) and the AIOP object (`dpaiop`).

1. Creating a new DPRC which is child object of root DPRC, `dprc.1`.

   ```
   $ restool dprc create dprc.1 --
   options=DPRC_CFG_OPT_SPAWN_ALLOWED,DPRC_CFG_OPT_ALLOC_ALLOWED,DPRC
   _CFG_OPT_IRQ_CFG_ALLOWED
   ```

   The output would be similar to:

   ```
   dprc.2 is created under dprc.1
   ```

   Note the new DPRC name, `dprc.2`, in the above output sample. Steps hereafter assume that new DPRC is `dprc.2`; please replace DPRC name in subsequent steps where required.

   State of newly created DPRC can be confirmed using following command:

   ```
   $ restool dprc info dprc.2
   ```

   Output would be similar to:

   ```
   container id: 2
   icid: 26
   portal id: 3
   version: 5.1
   dprc options: 0x43
   DPRC_CFG_OPT_SPAWN_ALLOWED
   DPRC_CFG_OPT_ALLOC_ALLOWED
   DPRC_CFG_OPT_IRQ_CFG_ALLOWED
   ```

2. *In case dpaiop object already exists in any DPRC, skip to Step 3 below.* The steps below are for creating a `dpaiop` object in the `dprc.1`. This requires information about the AIOP container (`dprc.3` has been assumed below).

   ```
   $ restool dpaiop create --aiop-id=0 --aiop-container=dprc.3
   ```

   In the above command, it is assumed that the AIOP container is dprc.3. The output would be similar to:

   ```
   dpaiop.0 is created under dprc.3
   ```

   The `dpaiop.0` object would be available in the `dprc.1` after this command.

3. Assuming that a `dpaiop` object, `dpaiop.0`, is available in the root `dprc.1`; Unplug the `dpaiop.0` in the `dprc.1` so that it can be assigned to a new DPRC.

   ```
   $ restool dprc assign dprc.1 --child=dprc.1 --object=dpaiop.0 --
   plugged=0
   ```

   Check the status of dpaiop.0 object using

   ```
   restool dprc show dprc.1
   ```

   to confirm if it has been put in '`unplugged`' state. If `dpaiop` object is already part of another DPRC, it has to be assigned to unplugged and assigned to `dprc.1` before it can be assigned to a targeted container. Steps for that would be similar to:

   Unplugging the dpaiop object state

```
$ restool dprc assign <dprc.X> --object=dpaiop.0 --plugged=0
```

Above command sample assumes dpaiop.0 is present in dprc.X Thereafter, move the dpaiop.0 object to dprc.1

```
$ restool dprc unassign dprc.1 --child=dprc.X --object=dpaiop.0
```

Hereafter, dpaiop.0 is available in dprc.1 to be moved to a target DPRC – as described in steps below.

4. Assign the dpaiop.0 object to dprc.2 and toggle state to plugged. This would work only if dpaiop.0 is already unplugged in root DPRC.

   ```
   $ restool dprc assign dprc.1 --child=dprc.2 --object=dpaiop.0 --
   plugged=1
   ```

   Check the status of dpaiop.0 object using

   ```
   restool dprc show dprc.2
   ```

   to confirm if it has been put in ‘plugged’ state.

5. Create a new MC portal object

   ```
   $ restool dpmcp create
   ```

   This result in an output similar to:

   ```
   dpmcp.4 is created under dprc.1
   ```

   New MC portal dpmcp.4 has been created under root DPRC. This needs to be moved to dprc.2 – as shown in next step.

   NOTE

   Note: It is possible to re-use an existing MC portal using steps similar to those shown below. It is possible that dpmcp object is already bound to certain driver (for e.g. VFIO driver), in which case, it needs to be unbound before being moved out.

6. Move dpmcp.4 from dprc.1 to dprc.2 and move to ‘plugged’ state.

   ```
   $ restool dprc assign dprc.1 --child=dprc.2 --object=dpmcp.4 --
   plugged=1
   ```

   Check the status of dpmcp.4 object using

   ```
   restool dprc show dprc.2
   ```

   to confirm if it has been moved into dprc.2 and put into ‘plugged’ state.

7. Confirm that both the objects, dpmcp.4 and dpaiop.0, are part of dprc.2

   ```
   $ restool dprc show dprc.2
   ```

   Output would be similar to:

   ```
   dprc.2 contains 2 objects:
   object       label      plugged-state
   dpaiop.0                plugged
   dpmcp.4                 plugged
   ```

   Hereafter, the DPRC dprc.2 is ready to be used with AIOP Tool once VFIO binding is done. See Sample VFIO Binding Script on page 813 above for information on VFIO binding.

8.3.7.3 AIOP User Manual

Click here to access the AIOP User Manual PDF.

8.3.7.4 AIOP Program Profiling

8.3.7.4.1 Overview

This document describes techniques for performance enhancements for software developed for AIOP. The reader should understand basic AIOP architecture and service layer API.

The structure of this document is the following:

- Explain the performance capabilities and limits of different processing elements of AIOP that should be taken into account at design time and in optimization time.
- Specific methods for identifying bottlenecks for different processing elements and memory subsystem

Readers should also have access to the CodeWarrior Development Studio for Advanced Packet Processing product inside a larger software suite called CodeWarrior Development Suites for Networked Applications (CW4NET), which includes the AIOP Analysis Tool and Scenarios tool.

AIOP Analysis Tool trace examples are used throughout this document. Users should be familiar with this tool since it is regularly used to debug and profile AIOP. Meanwhile, the Scenarios Tool is crucial when measuring memory bandwidth.

8.3.7.4.2 AIOP Program Design: Budgets Per Processing Elements

AIOP developers must consider the available capacity for each hardware element and design their programs accordingly.

For example, the LS2085A AIOP has 16 cores running at 800 MHz. This means that 800 x 16 = 12,800 million core cycles per second are available.

If instructions per cycle (IPC) is about 0.75 for each core, cores can execute about 10,000 million instructions per second. In order to handle 10 million frames per second, a budget of 1,000 instructions per each frame is available.

In order to achieve 17 million packets per second (MPPS) (which is approximately line rate for 2 x 10GE for 128-byte packet), maximum 588 instructions can be used to process a packet.

The Performance Capacity per Resource table below shows performance budgets for some AIOP operations.

Table 153. Performance Capacity per Resource

	Use case and other Information	**Performance** **capacity – Rev1**
Cores	16 x 800 MHz MCPS (million cycles per second) x 0.75 IPC = 9600 Instraction per second	16x800Mhz
OSM (Ordering Scope Manager)	Enter/exit pair, transitions	80 MOPS (million operations per second)
TMan (Timers Manager)	Timer commands per second (assuming 10M timers)	1 M [Timer fires/sec]
	Timer tasks initiated per second (assuming 10M timers)	1 M [commands/sec]
STE (Stats Engine)	Number of STE commands per second	68 M [commands/sec]
CDMA/FDMA (Context/Frame DMA)	17 MPPS packet presentations/ enqueues with 3 CDMA operation combined	
Tables	Exact Match (EM) key size up to 124 B;	51 MOPS

Table continues on the next page...

Table 153. Performance Capacity per Resource (continued)

	LPM key sizes, 4 byte EM + 4 byte LPM (IPv4) or 4byte EM + 16 byte LPM (IPv6)	17 MOPS
	Management commands on a 10 K rules balanced tree on PEB	500 K [commands/sec]
	Total 5 lookups at 17 MPPS: 3 Exact Match + 1 LPM + 1 MFLU	
	17 MOPS. ACL key size up to 56 B	17 MOPS
Parser	Max parser performance capacity	34 MOPS
	Max accesses to CTLU at 17 Mpps, including parser	6
	Typical use case at 17 Mpps	5 lookups + 1 parser

8.3.7.4.3 AIOP Program Profiling and Performance Tuning

AIOP programs have tight performance requirements and developers need to profile their applications in order to improve their performance characteristics.

There are two stages in profiling AIOP applications:

1. Finding bottlenecks in the application
2. Fixing bottlenecks found for each specific application

An AIOP task is a sequence of jobs executing on different processing elements. The performance of the overall task is dictated by the performance of the weakest element, which makes the weakest element a bottleneck of the application.

Take for example a task that involves receiving a frame, doing a look up, and sending the frame to another interface.

This task uses the core, FDMA, CDMA and TLU processing elements. Assuming that the core is going to run for 1000 cycles per task and that the task has 2 FDMA jobs, 2 CDMA jobs and one LPM CTLU job, how many tasks per second can the AIOP handle?

To answer this question, calculate the number of jobs each processing element can handle. Use the following assumptions to solve the example above:

- 16 cores running at 800 MHz
- Based on table for rev1, we can see that AIOP can perform 17 million of operations of 3 FDMAs + 2 CDMAs per second
- CTLU can perform 17 million LPM lookups per second

Because there are 16 x 800 MHz = 12,800 million cycles per second, 12,800/1,000 = 12.8 million tasks per second can be processed.

FDMA/CDMA: 17 millions per second = throughput of 17 million tasks

CTLU: 17/1 = throughput of 17 million tasks

Based on this analysis, the estimated maximum performance boundary for this application will be 12.8 tasks per second and its bottleneck is core performance. We do not take into account other possible artifacts, such as synchronization constrains etc. just for simplification of initial analysis.

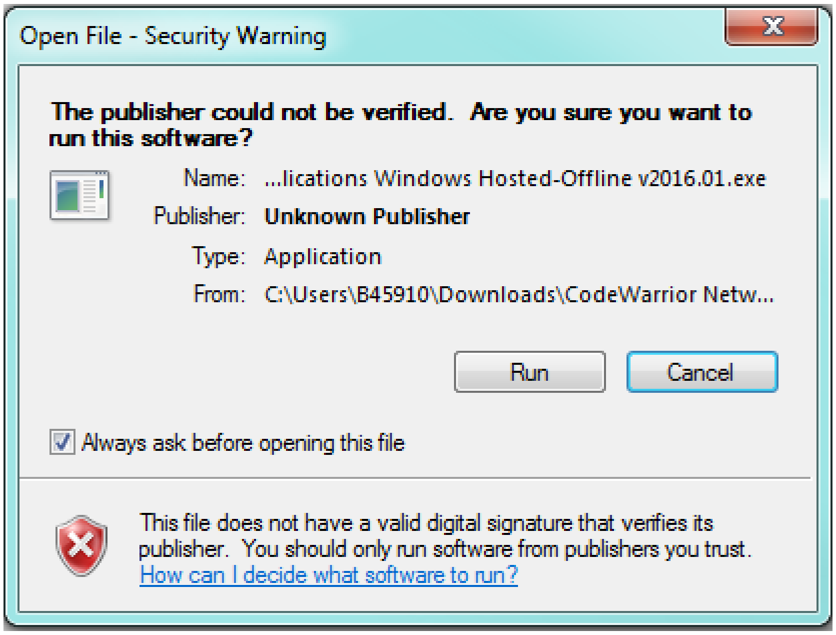
The analysis above is very useful for initial estimation during the design and implementation stages as it allows the programmer to design the program with specific performance goals in mind. For more information on performance characteristics of different processing elements see AIOP Program Design: Budgets Per Processing Elements on page 816.

Installing the AIOP Analysis Tool

In order to find bottlenecks within applications, AIOP tools are used; one in particular is the AIOP Analysis Tool.

The AIOP Analysis Tool is a component of CodeWarrior Development Studio for Advanced Packet Processing product inside a larger software suite called CodeWarrior Development Suites for Networked Applications (CW4NET). The AIOP Analysis Tool is available for download on the NXP Semiconductors website.

To download the AIOP Analysis Tool, click on the "Downloads" tab and under "CodeWarrior Development Studio for Advanced Packet Processing", select "Latest Version". Click on "CodeWarrior for Advanced Packet Processing Evaluation / Updates" and download the installer. After the download is complete, run the installer:



Open File - Security Warning

The publisher could not be verified. Are you sure you want to run this software?

Name: ...lications Windows Hosted-Offline v2016.01.exe

Publisher: **Unknown Publisher**

Type: Application

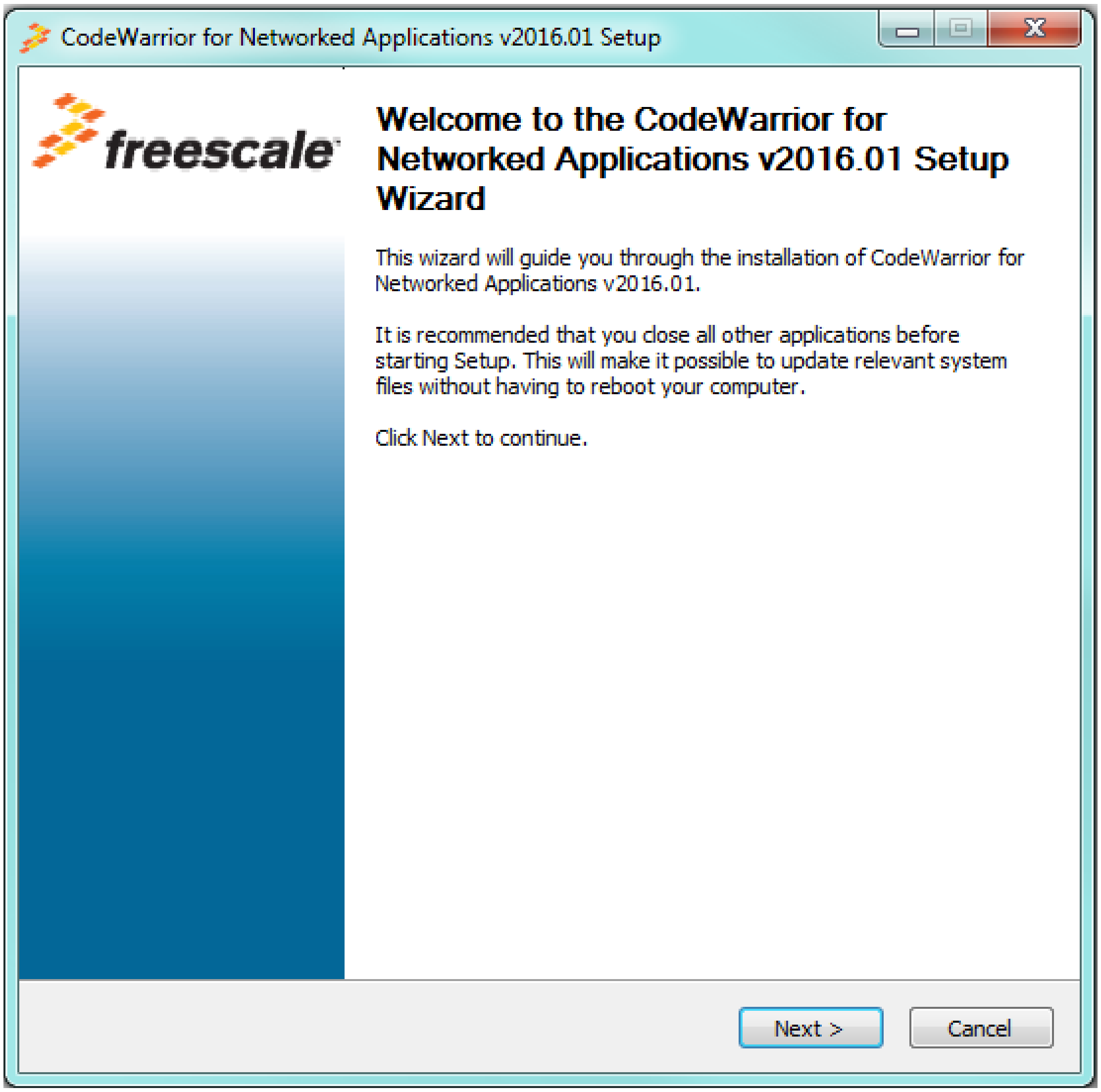
From: C:\Users\B45910\Downloads\CodeWarrior Netw...

☑ Always ask before opening this file

This file does not have a valid digital signature that verifies its publisher. You should only run software from publishers you trust. How can I decide what software to run?

Figure 174. CodeWarrior for Networked Applications Installer

After running the installer, follow the Setup Wizard and Click "Next":



Welcome to the CodeWarrior for Networked Applications v2016.01 Setup Wizard

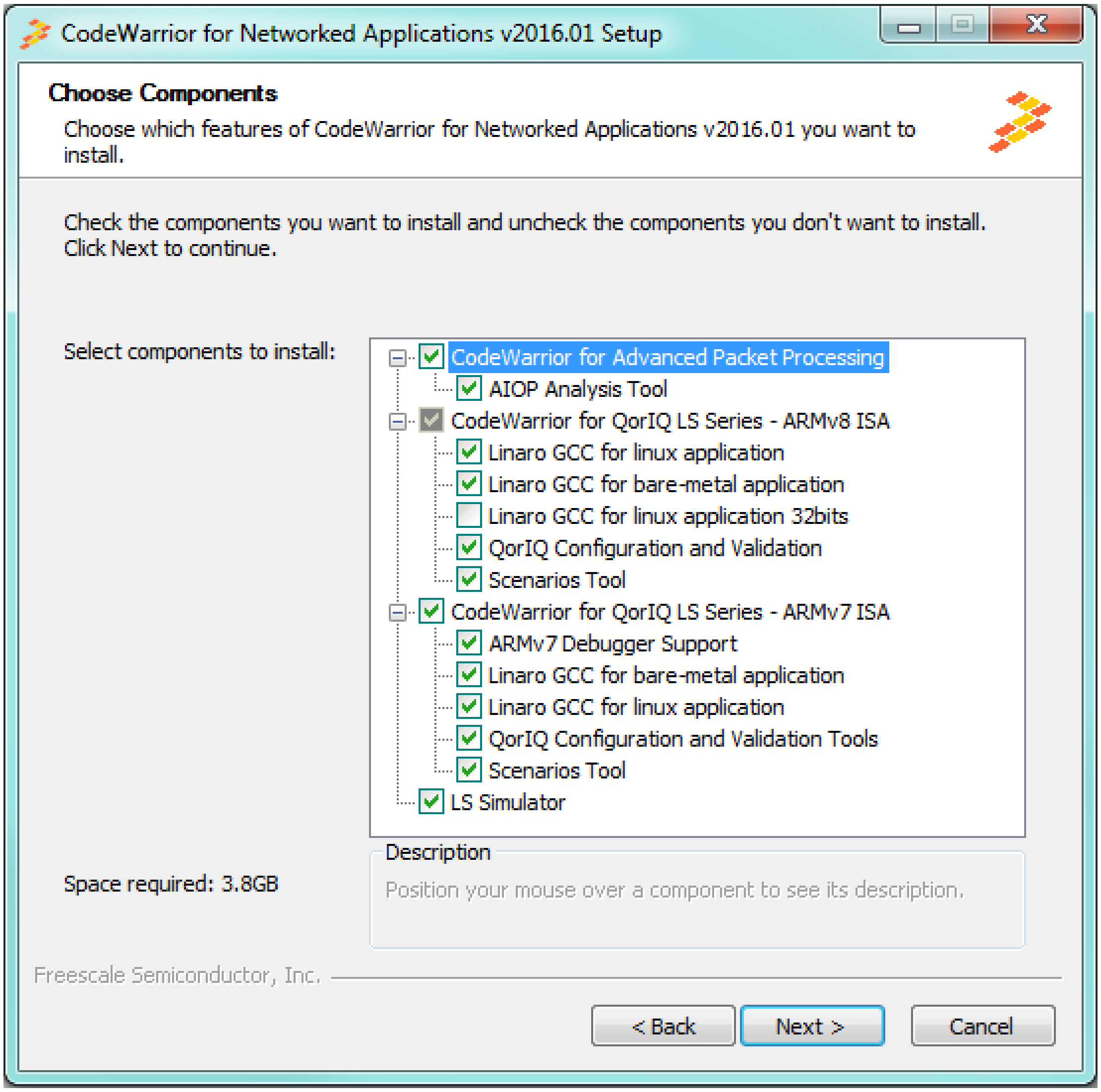
This wizard will guide you through the installation of CodeWarrior for Networked Applications v2016.01.

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

Figure 175. CodeWarrior for Networked Applictions Setup Wizard

The AIOP Analysis Tool has to be selected within the components to install. After confirming the selection, the installer will download the tools automatically according to the selection:



Choose Components

Choose which features of CodeWarrior for Networked Applications v2016.01 you want to install.

Check the components you want to install and uncheck the components you don't want to install. Click Next to continue.

Select components to install:

- ☑ CodeWarrior for Advanced Packet Processing
  - ☑ AIOP Analysis Tool
- ☑ CodeWarrior for QorIQ LS Series - ARMv8 ISA
  - ☑ Linaro GCC for linux application
  - ☑ Linaro GCC for bare-metal application
  - ☐ Linaro GCC for linux application 32bits
  - ☑ QorIQ Configuration and Validation
  - ☑ Scenarios Tool
- ☑ CodeWarrior for QorIQ LS Series - ARMv7 ISA
  - ☑ ARMv7 Debugger Support
  - ☑ Linaro GCC for bare-metal application
  - ☑ Linaro GCC for linux application
  - ☑ QorIQ Configuration and Validation Tools
  - ☑ Scenarios Tool
- ☑ LS Simulator

Space required: 3.8GB

Description

Position your mouse over a component to see its description.

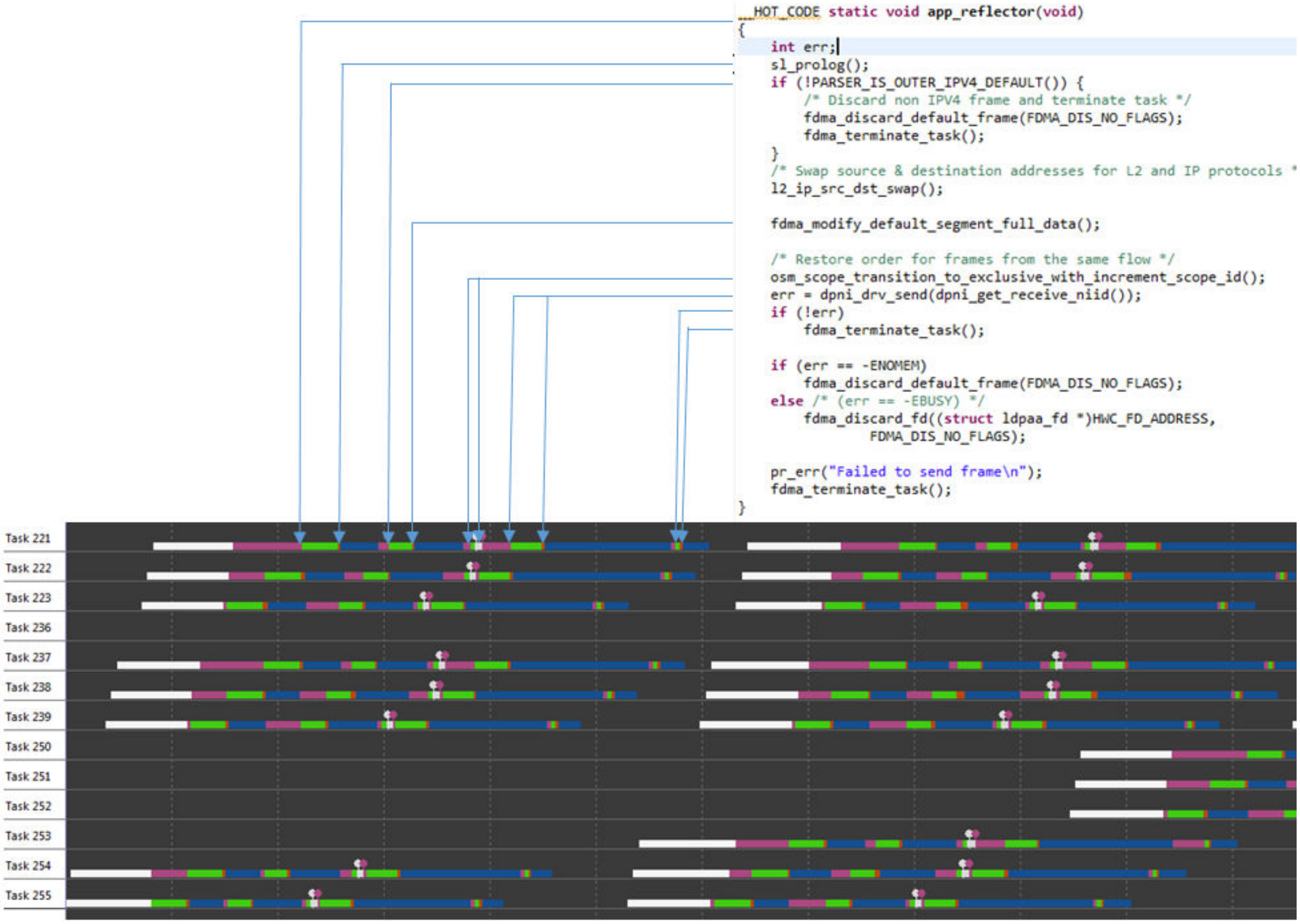
Figure 176. CodeWarrior for Networked Applictions Tool Selection

For more information, please refer to the CodeWarrior Development Studio for Advanced Packet Processing.

Bottleneck analysis of existing application

This section explains in detail how to find bottlenecks within applications using the AIOP Analysis Tool. This tool captures the scheduler trace, presents it in graphical way and provides some statistics based on this trace.

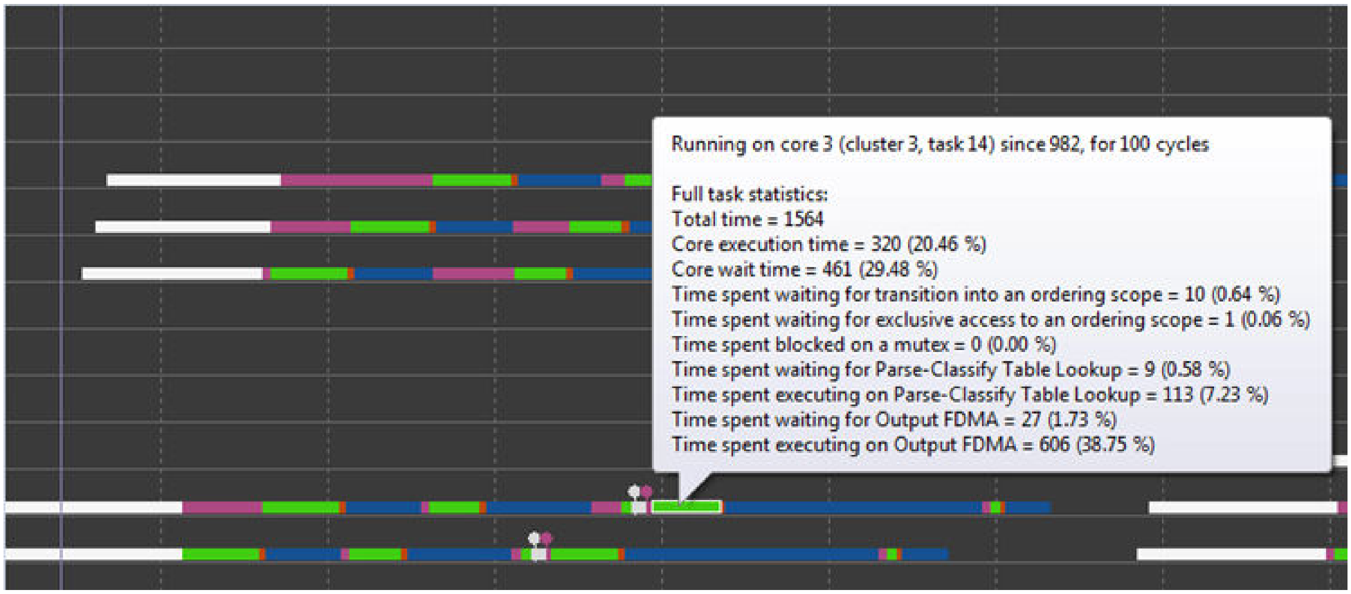
The figure below shows a code snippet and its associated trace for a simple reflector program:



```
__HOT_CODE static void app_reflector(void)
{
    int err;
    sl_prolog();
    if (!PARSER_IS_OUTER_IPV4_DEFAULT()) {
        /* Discard non IPV4 frame and terminate task */
        fdma_discard_default_frame(FDMA_DIS_NO_FLAGS);
        fdma_terminate_task();
    }
    /* Swap source & destination addresses for L2 and IP protocols *
    l2_ip_src_dst_swap();

    fdma_modify_default_segment_full_data();

    /* Restore order for frames from the same flow */
    osm_scope_transition_to_exclusive_with_increment_scope_id();
    err = dpni_drv_send(dpni_get_receive_niid());
    if (!err)
        fdma_terminate_task();

    if (err == -ENOMEM)
        fdma_discard_default_frame(FDMA_DIS_NO_FLAGS);
    else /* (err == -EBUSY) */
        fdma_discard_fd((struct ldpaa_fd *)HWC_FD_ADDRESS,
                FDMA_DIS_NO_FLAGS);

    pr_err("Failed to send frame\n");
    fdma_terminate_task();
}
```

Figure 177. Trace of Reflector Application

The trace measures the length of each job in the task and shows the utilization of resources. (hover the mouse to highlight a specific job):



Running on core 3 (cluster 3, task 14) since 982, for 100 cycles

Full task statistics:
Total time = 1564
Core execution time = 320 (20.46 %)
Core wait time = 461 (29.48 %)
Time spent waiting for transition into an ordering scope = 10 (0.64 %)
Time spent waiting for exclusive access to an ordering scope = 1 (0.06 %)
Time spent blocked on a mutex = 0 (0.00 %)
Time spent waiting for Parse-Classify Table Lookup = 9 (0.58 %)
Time spent executing on Parse-Classify Table Lookup = 113 (7.23 %)
Time spent waiting for Output FDMA = 27 (1.73 %)
Time spent executing on Output FDMA = 606 (38.75 %)

Figure 178. Task Information

What is the bottleneck of this application? In the following sections, we will show how you can find bottlenck based on this trace

8.3.7.4.4 FDMA/CDMA

The implementation of the CDMA/FDMA module is the following:

- FDMA and CDMA are actually one hardware block sharing the same resources.
- Thread = an FDMA/CDMA context which executes job on behalf of a task. 64 are present in rev1.
- Foreground slot = a HW execution resource that is loaded with a thread (and its context) in order to perform work on the thread (execute its command). 16 are present in rev1.
- Background slot = a HW resource that holds a suspended thread. This is a thread that is assigned (from a software point of view), but is blocked waiting for a high latency action to complete.
- Thread switching = moving FDMA threads between foreground (execution) and background (suspended) slots.

In the trace window we can see the number of assigned threads. This number is not precise due to how we measure it, but it can provide a good estimate.

Table 154. FDMA/CDMA Jobs

Core 13	63.12% Utilization
Core 14	63.97% Utilization
Core 15	63.23% Utilization
PC-CTLU	1.43 job average
TL-CTLU	11.86 job average
P.FDMA	3.65 job average
CDMA	20.91 job average
O.FDMA	10.49 job average
Total tasks	167
Tasks created	9,060,691.863 tasks/sec
Tasks finished	8,666,748.738 tasks/sec

We see here that the following number of threads are utilized:

1.43 + 11.86 + 3.65 + 20.91 + 10.49 = 48.34

48 is smaller than 64 so we are not running short of number of FDMA/CDMA threads.

Example of overall task analysis

Here we look at task trace of a reflector application and and analyzing it trying to find a bottlneck.

First, determine how many tasks are executing in parallel with the AIOP by looking at the following trace:

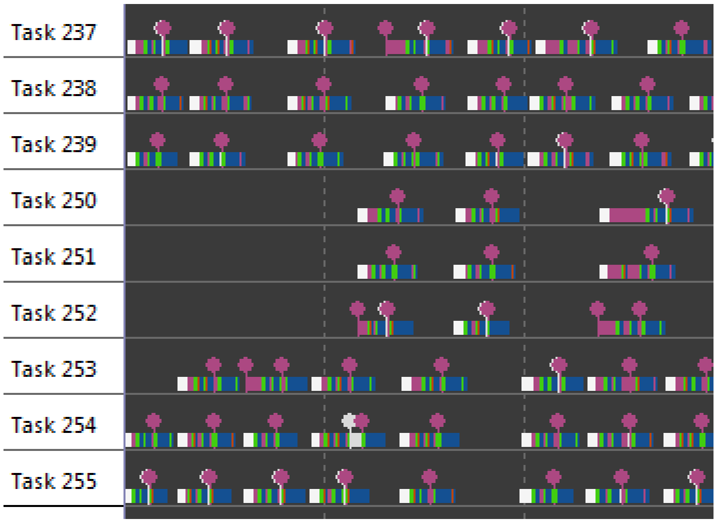


Figure 179. Trace Analysis

For simplicity we show only snapshot of tasks with higher numbers. Task scheduler begins scheduling with those tasks as well. Trace tool only shows tasks that were active at some point.

On core 15, only 6 tasks (Task 250 to Task 255) are executing, instead of 16 tasks. There are two possible reasons for this:

- The AIOP is underutilized; not enough traffic is sent to it.
- The FDMA/CDMA is a bottleneck as *only FDMA/CDMA can push back* to the work scheduler.

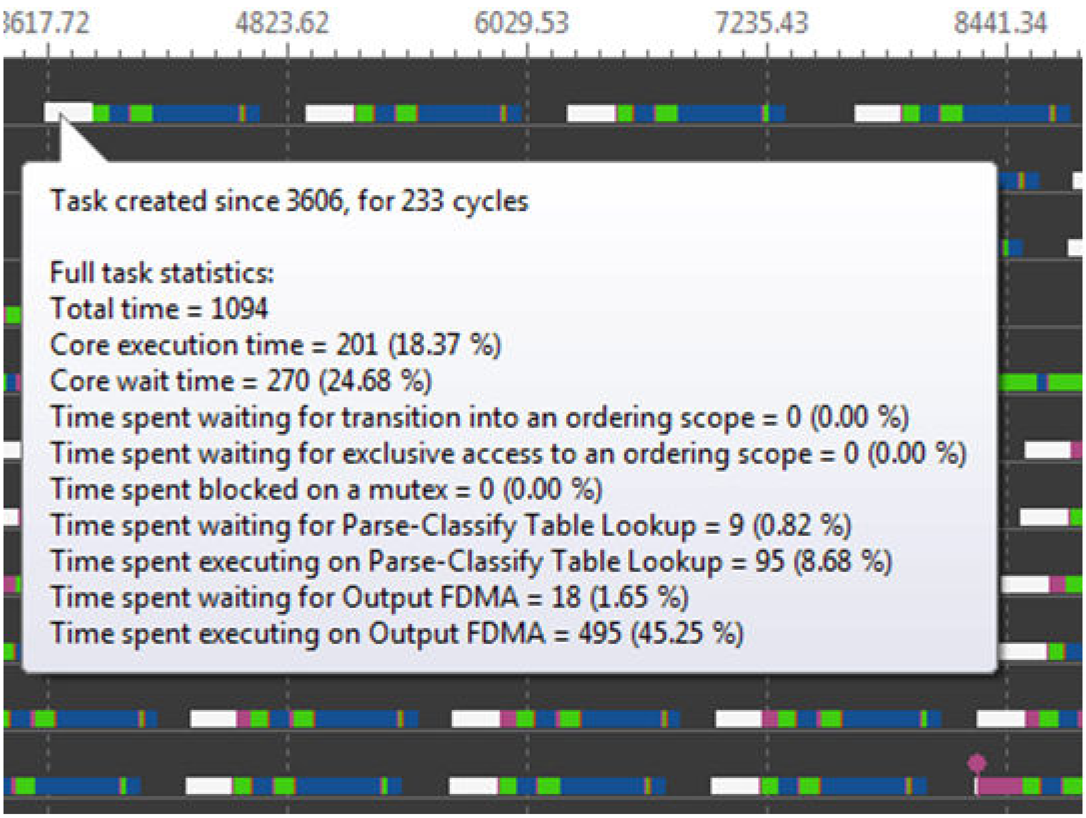
If enough traffic is sent to saturate the AIOP, but not all AIOP tasks are utilized, then the FDMA/CDMA is the bottleneck of your application.

Bottlenecks caused by CDMA/FDMA:

- Run out of Foreground slots. This may happen when memory becomes very congested and the CDMA waits for transactions for a long time. This bottleneck can be identified by looking at the average number of CDMA/FDMA jobs. If this number is close to the number of threads, then the bottleneck is within the CDMA/FDMA.
- Runs out of Threads. This happens when all CDMA/FDMA foreground slots are busy executing. This condition rarely happens as the previous condition happens first.

Strategies for reducing FDMA/CDMA bottlenecks

- Reduce the size of the initial presentation. The FDMA block is sensitive to presentation size; reducing presentation size from 128 to 64 Bytes may significantly reduce the pressure on the FDMA. For example, the figure below shows that task creation job (which is using FDMA) for 64 Byte presentation takes 233 cycles, while a 128-Byte presentation can take over 500 cycles. For IPv6 frames (or for any bigger header frames), initially present a bigger area or re-represent more data after determining that one is dealing with an IPv6 frame.



Task created since 3606, for 233 cycles

Full task statistics:
Total time = 1094
Core execution time = 201 (18.37 %)
Core wait time = 270 (24.68 %)
Time spent waiting for transition into an ordering scope = 0 (0.00 %)
Time spent waiting for exclusive access to an ordering scope = 0 (0.00 %)
Time spent blocked on a mutex = 0 (0.00 %)
Time spent waiting for Parse-Classify Table Lookup = 9 (0.82 %)
Time spent executing on Parse-Classify Table Lookup = 95 (8.68 %)
Time spent waiting for Output FDMA = 18 (1.65 %)
Time spent executing on Output FDMA = 495 (45.25 %)

Figure 180. Task Creation Job Using FDMA

- Reduce the number of CDMA calls. Look for opportunities to combine one or more reads or writes. This will reduce pressure on the CDMA block and memory. For example, when reading two integers, a and c, in struct such as:

  ```
  struct {
  int a;
  int b;
  int c;}
  ```

  It is better to read both integers in one read, even if that means also reading c.

- Mutex – Mutex on rev1 can be called 12 million times per second when multiple Mutex IDs are used and about 5 million times per second if a single ID is used.
- Try to align and pack all CDMA accesses. For example, when reading 64 bytes of data per packet, place it in DDR memory with a 64-byte alignment. In any case, try not to cross 64-byte boundaries when possible, as it will cause two accesses instead of one.
- Allow sufficient headroom for frame changes (inserts). There is an API that can override some parameters in the default storage profile, which should be called during the early initialization stage dpni_drv_register_rx_buffer_layout_requirements().
- Call once for multiple changes. For example multiple changes in a frame header can be made using a single FDMA call.
- Do not represent segments (for example frame data segments) when possible.
- While a frame is open for the AIOP task, it may be edited with FDMA commands, but the changes may not be visible to consumers outside the AIOP. There is no need to do an FDMA store command unless those updates must be visible to consumers outside the AIOP (e.g. an external memory).
- Some FDMA commands have options to encapsulate other commands, creating compound commands. Use compound commands when appropriate. See the table below.

Table 155. FDMA Compound Commands

Command	Combined Options
Present (open) Frame	Present frame data segment
Enqueue Frame	Terminate task Discard frame (when enqueue fails) Relinquish OSM exclusivity in current scope right after the enqueue to QMan is issued
Replace Segment Data	Represent Segment Data Close segment
Concatenate Frames	Close concatenated frame
Split Frame	Close new frame Present new frame data segment
Replicate Frame	Enqueue new frame, optionally Relinquish OSM exclusivity Discard source frame

8.3.7.4.5 Core Profiling

Strategies for reducing core bottlenecks

Core bottlenecks are easily identified by examining the core load in the trace tool. If the core load gets close to 100% percent, it means that the bottleneck is core cycles. Generally, it is a good problem to have after the final optimization stage as the core is the most valuable resource. If the application has a bottleneck with the core, that means the core is being used to its maximum. This table is an example of usage of trace tool for core profiling:

Table 156. Core Utilization

Range Time	1118954.00 cycles
Core 0	99.60% Utilization
Core 1	83.13% Utilization
Core 2	93.11% Utilization
Core 3	99.61% Utilization
Core 4	99.62% Utilization
Core 5	86.42% Utilization
Core 6	83.14% Utilization
Core 7	89.74% Utilization
Core 8	86.37% Utilization
Core 9	86.43% Utilization
Core 10	83.15% Utilization

Table continues on the next page...

Table 156. Core Utilization (continued)

Core 11	89.72% Utilization
Core 12	86.43% Utilization
Core 13	89.68% Utilization
Core 14	89.83% Utilization
Core 15	99.92% Utilization

Example task analysis for core utilization

What can be done to resolve this problem?

First, look at the trace, identify all core jobs and check that job lengths are reasonable and expected. If you find something not expected, examine code that is executed for this job and try to find ShRAM accesses, make sure code is running from iRAM (for performance sensitive code), no accesses to PEB, DDR or regicters are made in your code. Access to DDR can take up to 200 cycles and should be avoided.

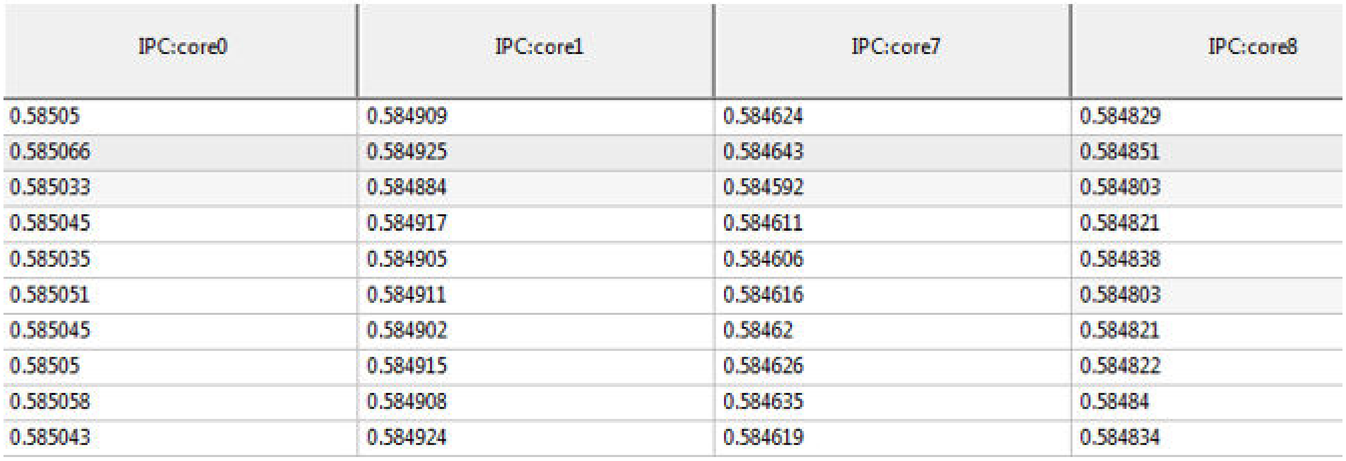
It is a good idea to measure IPC of your program so that you know that it is in a reasonable range. We use scenario tool to measure it on AIOP. In order to measure IPC directly in a running program, open the following scenario: `aiop_throughput_IPC_ssram_access` (*warning: you must disable run-time stack check in AIOP by undefining STACK_OVERFLOW_DETECTION as this feature is utilizing the same resources as this scenario)*

The IPC for each core is measured based on the following formula:

INSTR_COMPLETED / (PLATFORM_CLK - CTS_NO_TASK_CYCLES)

Based on the formula, divide the *number of instructions completed by each specific core* by the *number of cycles that core was executing tasks* (was not idle).

After measurement, the result is similar to the figure below:

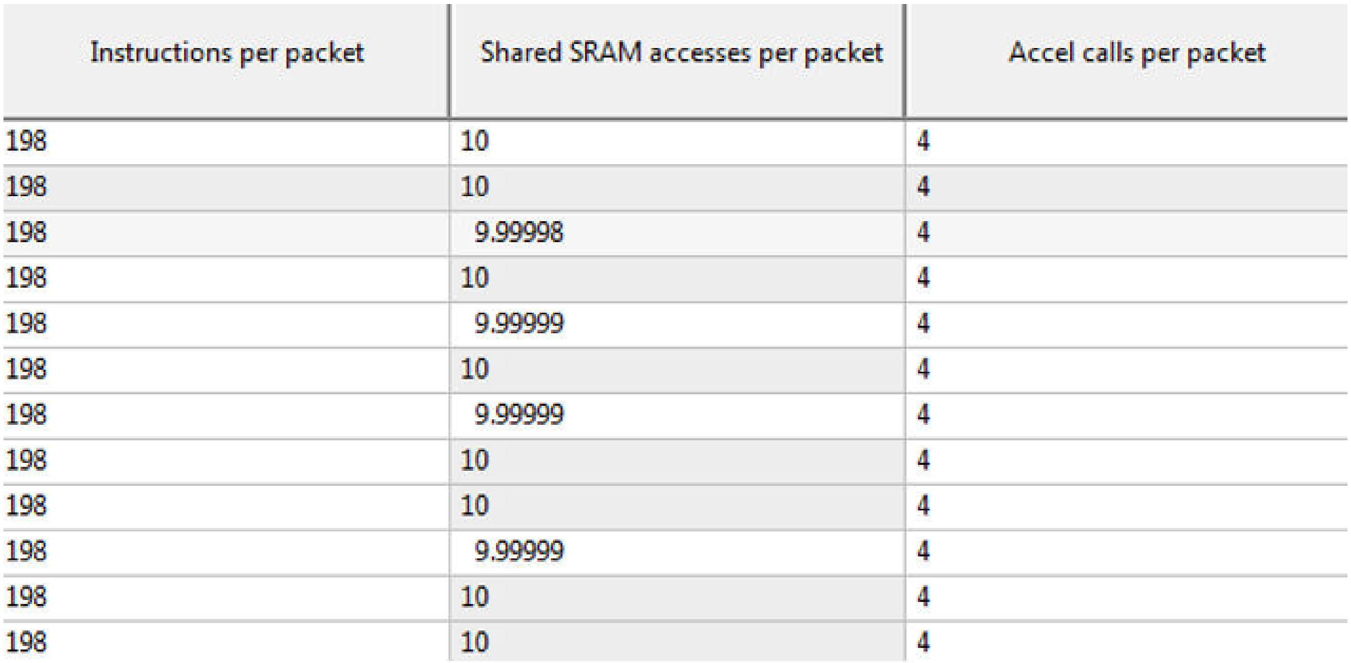


IPC:core0	IPC:core1	IPC:core7	IPC:core8
0.58505	0.584909	0.584624	0.584829
0.585066	0.584925	0.584643	0.584851
0.585033	0.584884	0.584592	0.584803
0.585045	0.584917	0.584611	0.584821
0.585035	0.584905	0.584606	0.584838
0.585051	0.584911	0.584616	0.584803
0.585045	0.584902	0.58462	0.584821
0.58505	0.584915	0.584626	0.584822
0.585058	0.584908	0.584635	0.58484
0.585043	0.584924	0.584619	0.584834

Figure 181. Example IPC for Each Core

Generally, if the IPC is lower than 0.70, we need to investigate why. One of the most common reasons for low IPC would be accessing ShRAM (Shared SRAM) in the program. However if the resulting IPC is a really low number, then the code must not be running from IRAM.

Here is how one can calculate the approximate IPC based on number of ShRAM accesses and accelerator calls. In the previous scenario, we can look at following data:



Instructions per packet	Shared SRAM accesses per packet	Accel calls per packet
198	10	4
198	10	4
198	9.99998	4
198	10	4
198	9.99999	4
198	10	4
198	9.99999	4
198	10	4
198	10	4
198	9.99999	4
198	10	4
198	10	4

Figure 182. Packet Information: ShRAM Accesses and Accelerator Calls

The formula below was used to approximate the number of cycles spent by the core as a function of the number of ShRAM accesses, accelerator calls, and instructions executed.

N_Core_Cycles = (N_Instr * CPI) + (N_ShRAM * 12) + N_Accel_Calls * 4

In our case we get:

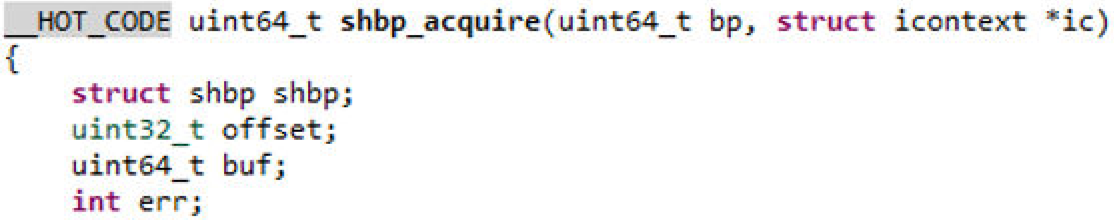
N_Core_Cycles = 198 * 1 + 10 *12 + 4*4 = 332

IPC ~ 198/332 = 0.596

For the example above, this low IPC is due to a high number of ShRAM accesses.

Improving core performance

- Inline the code where possible, especially for functions that are part of a “hot path”. Most of SL APIs are already in-lined.
- Reduce the number of accesses to shared memory. Shared memory has 12 cycles latency and 10 accesses will incur at least 120 cycles.
- Place all per-packet code in IRAM. For that, qualify such code with `__HOT_CODE`. For example:



```
__HOT_CODE uint64_t shbp_acquire(uint64_t bp, struct icontext *ic)
{
    struct shbp shbp;
    uint32_t offset;
    uint64_t buf;
    int err;
```

Figure 183. Example of `__HOT_CODE`

- Never place any performance sensitive code in DDR.
- Avoid floating-point operations and operands. The e200 core emulates floating-point operations/operands, which causes performance to be very low. Use fixed-point operations instead, when required.

- Use all available cores and maximize possible tasks per core. Typically, it will not be an issue to use all the cores, but using all tasks could be challenging sometimes because of stack size. If the stack requires more than ~1500 available entries to maximize the number of tasks, do the following:
    1. Run the stack static analysis tool and check where you use most of the stack
    2. Restructure your code, so that:

       `a() -> b() ->c()` chain that will require a lot of stack

       is replaced with the following:

       ```
       a();
       b();
       c();
       ```

8.3.7.4.6 Memory profiling

AIOP programs may use different types of memories:

- external (DDR, PEB)
- internal (ShRAM, Workspace)

DP-DDR (DDR3 controller)

DDR memories are the first suspect to be oversubscribed. During the design stage of the system, developers should keep DDR bandwidth in mind.

The bandwidth of DP-DDR is 4 Bytes x 1.6 GHz = 6.4 GB/s

In a perfect situation where everything is aligned, accesses are all 32 bytes in length. However, this is not usually the case, but this number can provide a good starting point.

Example task analysis for DDR usage

So, let's assume we have context data in DP-DDR. It has 32 byte size and it accesses each frame with 17 MFPS.

We will get 32 x 17 MFPS = 544 MB/s, which is 544 MB/s / 6.4 GB/s = 8.5% of bandwidth

What happens when tables are placed into this memory?

The following is how look-up hardware accesses memories:

- LPM IPv4 takes 4*(1+alpha/2) memory accesses
- LPM IPv6 takes 6*(1+alpha/2) memory accesses
- EM takes (1+alpha/2) memory accesses

Alpha is the fill factor which goes from 0 to 1. The worst case condition is when alpha is 1. This means the tables fully utilize the CTLU memory.

Each access is 64 bytes.

Assume that the EM table has been placed in DP-DDR. Calculate the load on DP-DDR when running at 17 MFPS load.

The load will be:

1 x 64 x 17 MFPS / 6.4 GB/s = 0.17 = 17%

DP-DDR can safely be loaded up to 60%. However at 60% load, expect a spike in latency of DP-DDR. For example, it is impossible to place the LPM table there at 17 MFPS:

4 x 64 x 17 MFPS / 6.4 GB/s = 0.61 = 61%

Improving memory bottlenecks

Recommendations for table placement

The decision for each table placement should be made based on the following parameters:

1. Frequency—how often the table is accessed.
2. Size of the table—big tables will not fit in PEB.
3. Availability of bandwidth or place in specific memory.

Some guidelines:

- If DP-DDR is populated, it should be utilized as much as possible up to 60% of bandwidth, in order to reduce pressure on system DDR

For small tables that are used relatively frequently PEB could be the best candidate.

For System DDR, the L3 cache must be enabled in rev1.

For example:

Small (several hundred entries) ACL tables that are accessed per packet at a very high rate (millions times per second) should go to PEB memory.

Big LPM tables that are accessed infrequently should be placed in DP-DDR. The same table that is accessed higher than 60% DP-DDR utilization should be placed in System-DDR.

As a practical approach, we suggest not to use DP-DDR at the first stage of development and leave this optimization for later stages of development.

Recommendations for data placement

Data can be placed in several types of memory:

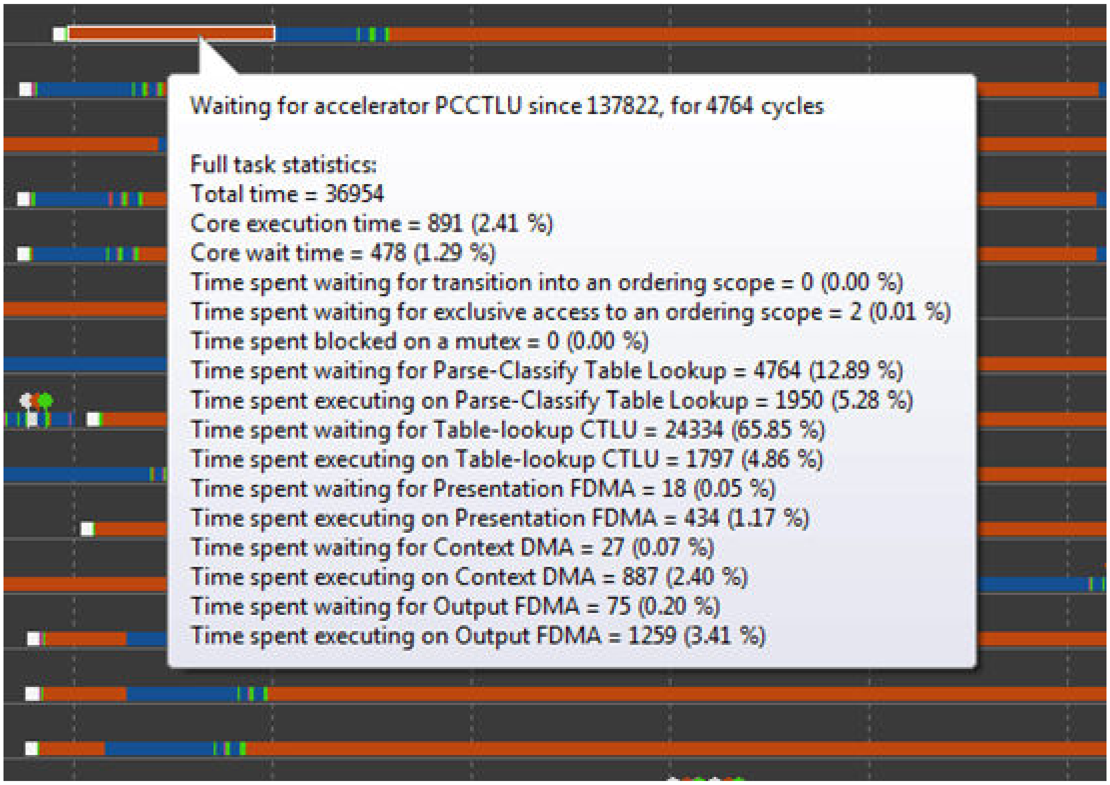
- System DDR
  - — Direct access by core (not recommended) is ~200 cycles
- DP-DDR
  - — Direct access by core (not recommnded) is ~200 cycles
- Shared RAM
  - — ~10-15 cycles access
- PEB
  - — ~40 cycles access by core (not recommnded)

When deciding about placement of different types of data, first consider how this data scales:

- Data scaling with number of flows
  - — As number of flows is typically high (more than thousands), it should be placed in DDR
- Frequently accessed data that is scaling as number of interfaces or application instances
  - — Such data should be placed in ShRAM (as statically defined data). Exceptions could be made for data that is accessed in bulk. In that case, place the data in PEB, so that it could be brought from DMA to local memory as a whole structure and then several data fields can be accessed with zero latency

8.3.7.4.7 CTLU - Parser

The CTLU or parser becomes a bottleneck of the application when tasks wait for the CTLU or parser for a long time. For example, the trace below shows this situation:



Waiting for accelerator PCCTLU since 137822, for 4764 cycles

Full task statistics:
Total time = 36954
Core execution time = 891 (2.41 %)
Core wait time = 478 (1.29 %)
Time spent waiting for transition into an ordering scope = 0 (0.00 %)
Time spent waiting for exclusive access to an ordering scope = 2 (0.01 %)
Time spent blocked on a mutex = 0 (0.00 %)
Time spent waiting for Parse-Classify Table Lookup = 4764 (12.89 %)
Time spent executing on Parse-Classify Table Lookup = 1950 (5.28 %)
Time spent waiting for Table-lookup CTLU = 24334 (65.85 %)
Time spent executing on Table-lookup CTLU = 1797 (4.86 %)
Time spent waiting for Presentation FDMA = 18 (0.05 %)
Time spent executing on Presentation FDMA = 434 (1.17 %)
Time spent waiting for Context DMA = 27 (0.07 %)
Time spent executing on Context DMA = 887 (2.40 %)
Time spent waiting for Output FDMA = 75 (0.20 %)
Time spent executing on Output FDMA = 1259 (3.41 %)

Figure 184. CTLU - Parser Bottleneck Example

Based on the trace above, it is evident that the wait time for the parser (PCCTLU - "Parse Classify CTLU") is very significant. The following are reasons for the long wait time:

- Too many calls to the parser/TLU
- The memory that the TLU works with experiences high load, which results in high latency. For memory issues, refer to the Memory profiling on page 828 section.

Example task analysis for CTLU usage

Improving CTLU bottlenecks

Listed below are some ideas on how to alleviate load on the parser:

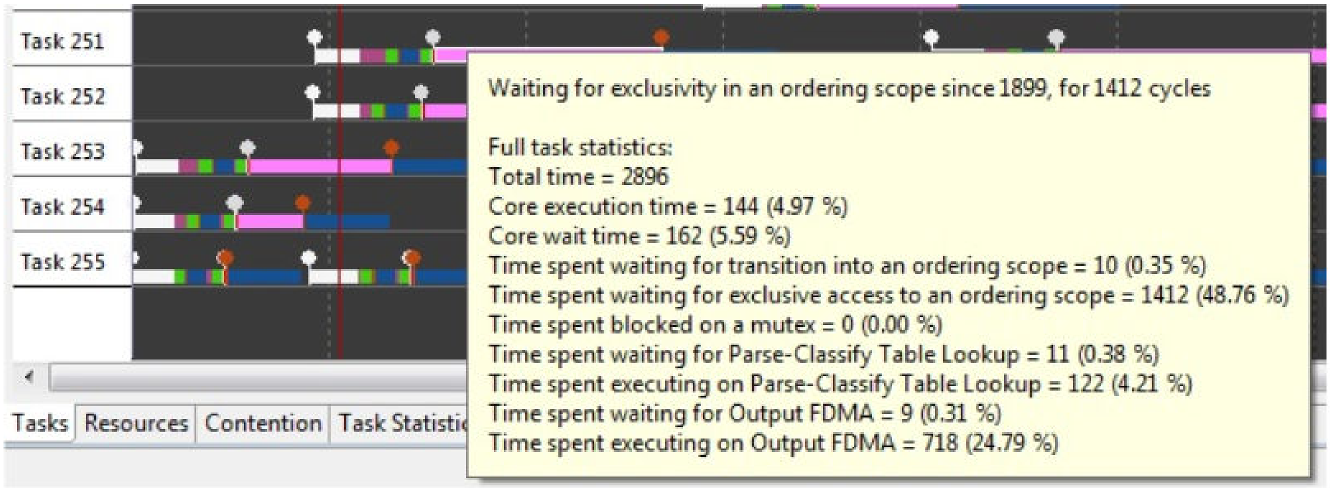
- One of the features of DPNI is to calculate gross running checksum for ingress frames. So, when frame is received form by AIOP, it has associated valid gross running checksum. It is a checksum of the entire frame. Once we change frame data, the checksum becomes invalid unless we update it. If the parser is called with validation flags enabled, and the gross running sum was set to 0, it will first recalculate the gross running sum of the entire frame. When appropriate, do not invalidate (set to 0) the gross running sum field if the parser will later be called with the validation flags enabled.
- In case a VLAN header was added or removed, it is possible to call the software functions `parser_push_vlan_update()` or `parser_pop_vlan_update()` to update the workspace parse results instead of calling the hardware parser routines. This method should be used when the ratio of calls to the hardware parser routine is beyond the performance capacity. See the AIOP Program Design: Budgets Per Processing Elements on page 816 section.

8.3.7.4.8 OSM

It is assumed the reader is already familiar with the details of OSM operation and use. Here we are only concerned with refining the use of OSM to enable best performance. It is also assumed the reader has a basic familiarity with the AIOP analysis tool to view what is going on with task/job scheduling within the AIOP.

OSM is used to add order constraints on the scheduling of jobs of a task within the AIOP based on network ordering requirements of the packet being processed. Therefore the goal in optimizing performance when using OSM is to minimize how often the task scheduler will block a task's progress based on order constraints. In other words OSM *inhibits* jobs of a task from being scheduled and optimization minimizes the time it does so.

The image below demonstrates using the AIOP analysis tool to show how contention in an ordering scope looks like. The mouse when hovered over the pink bar in Task 251 shows additional information about this phase of a task execution. It shows in this example that the task was blocked for 1412 cycles because it needs exclusive phase of a particular ordering scope but it is blocked because other tasks are ahead of it.



Waiting for exclusivity in an ordering scope since 1899, for 1412 cycles

Full task statistics:
Total time = 2896
Core execution time = 144 (4.97 %)
Core wait time = 162 (5.59 %)
Time spent waiting for transition into an ordering scope = 10 (0.35 %)
Time spent waiting for exclusive access to an ordering scope = 1412 (48.76 %)
Time spent blocked on a mutex = 0 (0.00 %)
Time spent waiting for Parse-Classify Table Lookup = 11 (0.38 %)
Time spent executing on Parse-Classify Table Lookup = 122 (4.21 %)
Time spent waiting for Output FDMA = 9 (0.31 %)
Time spent executing on Output FDMA = 718 (24.79 %)

Figure 185. Task Waiting on Ordering

The obvious optimization here is to reduce the number of tasks in the same ordering scope at a given time and to minimize the time each of these tasks keeps the exclusive phase of any ordering scope. An optimized software design will do both as much as possible while remaining correct. Also in a perfect optimization, all tasks would be in different ordering scopes or all tasks would use zero cycles in the exclusive phases of an ordering scope but this is never possible. The AIOP analysis tool will help the software designer to gain experience in the practical use of ordering scopes.

Example task analysis for OSM

Consider a simple example of a task. This task extracts the destination IP address as a key and does a TLU lookup based on that key. The lookup result is a reference to system memory that contains an ARP table entry. The entry is copied from system memory to local workspace. A hit counter in the entry is incremented and the system copy updated to reflect that. The entry also contains a MAC address which is used to replace the destination MAC address in the packet received and then forwarded.

```
key.dst = _presentation_with_vlan.ipv4hdr.dst_addr;
key_desc.em_key = (void*)&key;
table_lookup_by_key(TABLE_ACCEL_ID_CTLU, tid_arp, key_desc, sizeof(struct simple_key),
&lookup_result);
cdma_read( (void*)&arp_entry, lookup_result.opaque0_or_reference, 16);
arp_entry.hits++;
cdma_write(lookup_result.opaque0_or_reference + offsetof(struct arp_entry, hits),
(void*)&arp_entry.hits, 8);
memcpy(&_initial_presentation.ethhdr.da, &arp_entry.mac, 6);
fdma_modify_default_segment_full_data();
fdma_store_and_enqueue_default_frame_fqid( DESTINATION_FQ, FDMA_EN_TC_TERM_BITS);
```

This is a made up example which does no error checking. Assume from this simple example that forwarded frame is required to be in the same order as the frame arrived on a network interface for any flow. This is not a very realistic example but it will suit the purpose of observing the performance when OSM operations are included.

The simple method to meet the order requirement in this application is to run all packets as if they are in a single flow (single common initial scope ID) and in exclusive mode for the duration.

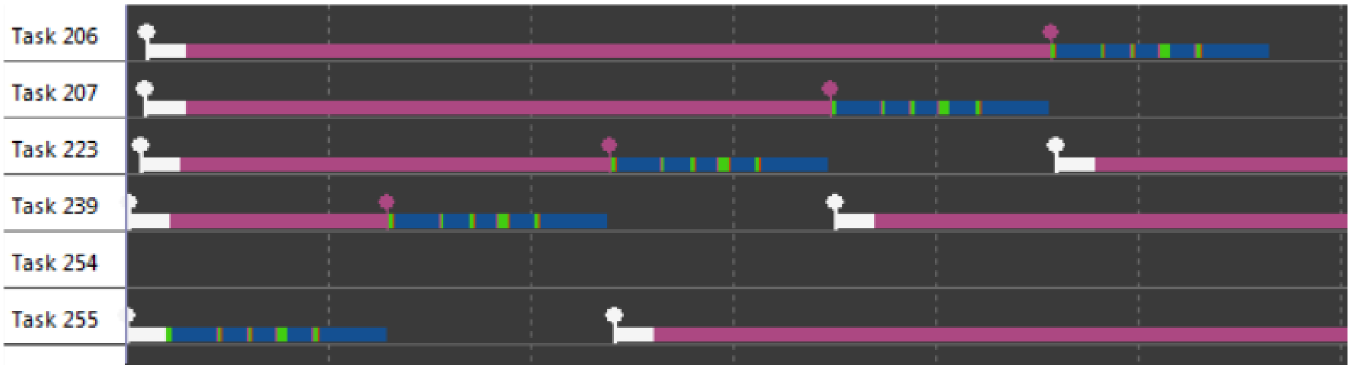


Figure 186. Tasks Executing Sequentially

A snapshot from the AIOP analysis tool shows the result of running all packets as if they are from a single flow in the exclusive phase of a common ordering scope. In this image, blue bars are hardware jobs, green are software jobs, and red is blocked. It is immediately obvious each packet is processed one at a time until done before a subsequent packet starts processing. This is the baseline to improve upon.

Spread based on flow

AIOP tasks will run in parallel more effectively if tasks of different flows begin in different initial scope IDs. In this way they, by definition, do not compete. To demonstrate this, the example of the previous section is repeated, all tasks start as exclusive, but now the packets come from 32 different flows and each flow defines an initial scope ID.

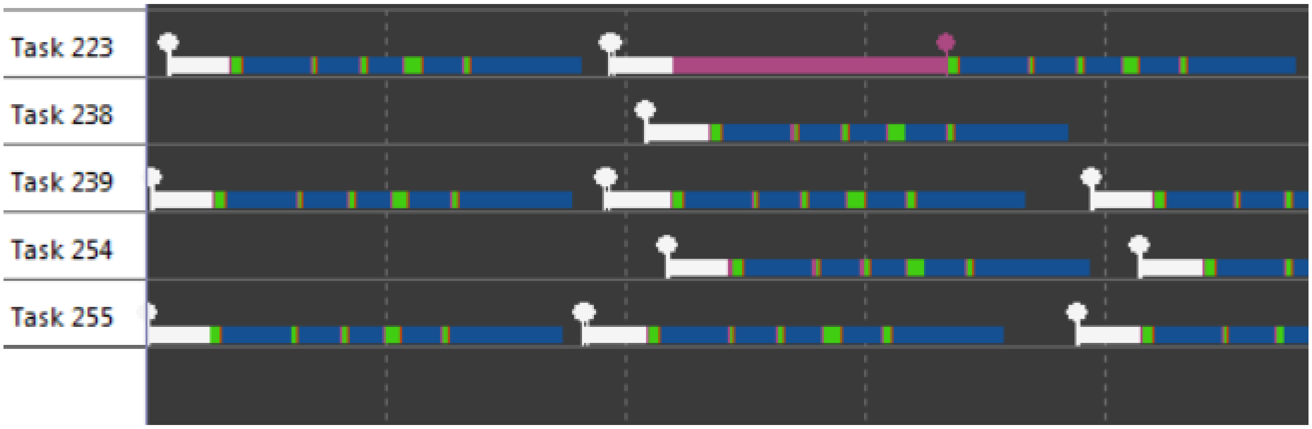


Figure 187. Tasks in Different Flows Running Parallel

As seen from the AIOP analysis tool trace, many operations are occurring in parallel. Both hardware jobs (blue) and software jobs (green) are happening in parallel. In this snippet only one task is observed as blocked. This is the most simple and effective way to improve parallel operation in the AIOP. However, it is never possible to know that ingress traffic will be in a large number of flows at any instant in time.

The DPNI may or may not allow sufficient distribution (spread) of flows. However for every application there is a unique key comprised of header fields to uniquely identify a flow. This unique key is reduced to a hash and used as a new scope ID when classification must be refined within the AIOP. This would be step one of a task and it can be performed in concurrent mode of the initial scope ID to maintain parallel operation.

Either way, if DPNI does or does not provide sufficient spread of flow, it is the application responsibility to pick what header fields uniquely identifies a flow. Use dpni_drv_set_order_scope() to specify initial order scope construction from header fields so that tasks are created with the best possible initial order scope ID. This is the preferred and best performing method. If that is not possible then the user may create a key composition ID (KeyID) from the header fields required during application initialization and use that KeyID to generate a hash suitable for order scope ID after tasks are launched. See key_composition_rule_create().

Run concurrent where possible

In general an AIOP task begins execution in the concurrent phase of the initial ordering scope and delays transition to a subsequent scope in the exclusive phase as long as it is practical. In our example task it is possible to run concurrently up to the point where the ARP table is read and updated. Consider a single flow where our example application runs concurrently and then moves to exclusive after the TLU lookup.

```
osm_scope_transition_to_exclusive_with_increment_scope_id();
```

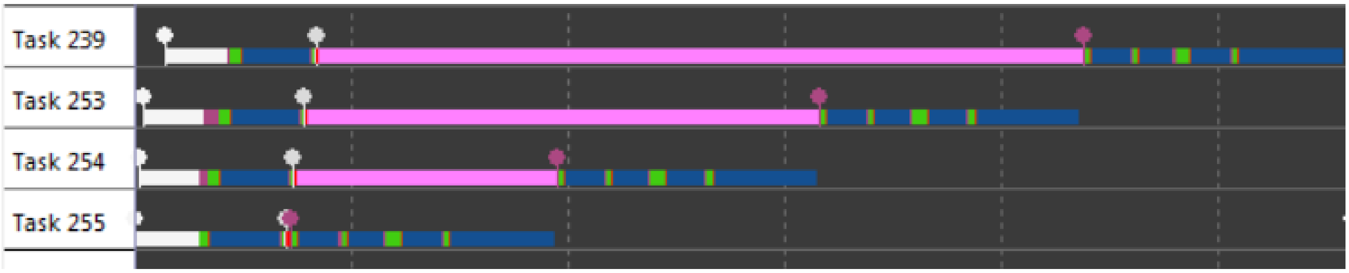


Figure 188. Tasks in a Flow Blocking for Exclusivity

The TLU will perform lookup operations in an atomic fashion. Take advantage of this characteristic of the AIOP accelerators by using them from a concurrent phase of an ordering scope where ever possible. As can be seen in the AIOP analysis tool snippet, the first hardware job, in this case a TLU lookup, is performed in parallel with other tasks. The remainder of the task is run exclusively and parallel operation ceases.

Relinquish exclusivity quickly

In our example application the read-modify-write of the ARP table entry must be done exclusively. However the MAC address update does not because the data and frame are private to the task. A relinquish exclusivity is inserted following the ARP update and another transition to exclusive is inserted prior to forwarding to remain ordered.

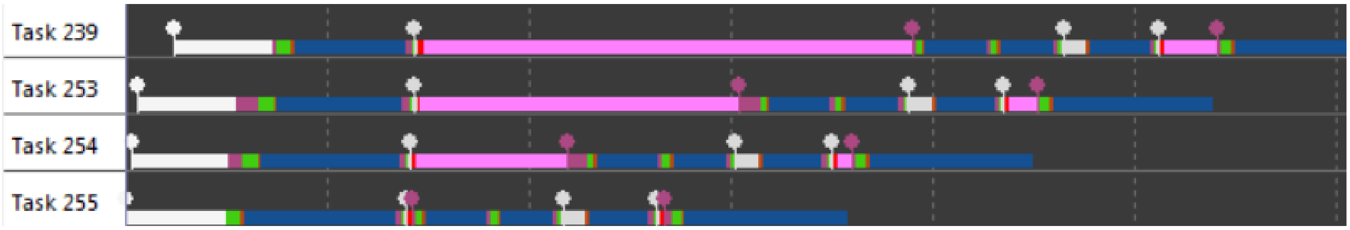


Figure 189. Tasks of a Flow Relinquishing Exclusivity Affect

Consider alternatives

Our example takes advantage of OSM to create an exclusive phase to perform a critical operation, namely to read-modify-write the ARP table entry. Note this is an *ordered* excusive operation. That is tasks will update their ARP entry in the order of packet arrival. In this example the update is just a hit counter and does not require update in order.

It is possible to take advantage of two properties of the hardware accelerators here. First, the sequential update of the ARP entry hit counter can be performed by the STE (statistics engine). Second, the atomic operation of CDMA commands can assure consistent values regardless of other readers or writers. Our example can take advantage of these properties to rewrite our ARP entry handling.

NOTE

The atomic operation of CDMA read and write operations is a specific enhancement over the first revision of the silicon. This atomic behavior allows a reader to be certain a writer will not corrupt an entry; the value read will be valid but not ordered with respect to the writer.

Rewriting this section will look like the following:

```
cdma_read( (void*)&arp_entry, lookup_result.opaque0_or_reference, 16);
ste_inc_counter( (lookup_result.opaque0_or_reference + offsetof(struct arp_entry,
hits)),STE_MODE_64_BIT_CNTR_SIZE);
```

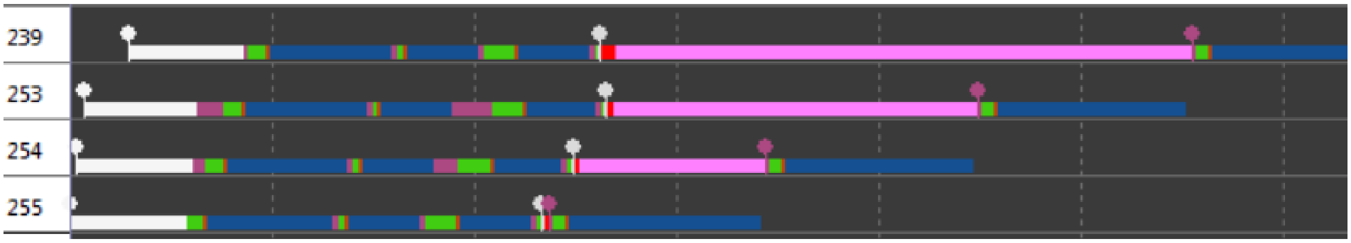


Figure 190. Tasks of a Flow Making Use of Atomic Accelerator Operations

In our alternative method of using STE, all hardware accelerations with the exception of forwarding can now operate in parallel even though all packets are of a single flow. Cycle count for the first four packets forwarding is down to about 3,000 cycles.

OSM Capacity

OSM itself is a resource of limited capacity. To maintain maximum task throughput the number of transitions and enter/exit scope pairs should be limited to a total of about five. Beyond that the maximum task rate will start to decline. It is rare that a task of a complexity to require more than five OSM operations will be limited by OSM before it is limited by other hardware accelerator throughput. However the AIOP analysis tool will be the designer's main insight into where bottlenecks are occurring within the AIOP.

Use osm_scope_enter_to_exclusive_with_new_scope_id() and osm_scope_enter_to_exclusive_with_increment_scope_id() instead which pick the best hardware options by default and have little software overhead

Additional Guidelines

Experience with insight provided by using the AIOP analysis tool will guide the user to best practices for improving parallel operation and relieving bottlenecks.

- Primarily use transition increment forms of OSM commands as they are the best performers and naturally partition an application's overall design into steps from ingress to egress.
- Reduce cycles in exclusive phases as much as possible.
- Use accelerators in concurrent phases as much as possible.
- Create new scope IDs to refine (distribute) flows.
- Take advantage of atomic accelerator characteristics in concurrent phases.
- Avoid using the osm_scope_enter() function as its many options and overhead take many cycles.
    - Use osm_scope_enter_to_exclusive_with_new_scope_id() and osm_scope_enter_to_exclusive_with_increment_scope_id() instead which pick the best hardware options by default and have little software overhead
- Consider splitting a long operation requiring the exclusive phase into multiple exclusive phases.

These are only guidelines to consider while experience will guide the designer. The AIOP analysis tool gives insight into the behavior of tasks. Often times the interaction between decisions is not obvious and some trial and error is required for best results.

8.3.7.4.9 Statistics Engine

The statistics engine will stall the core when it is overloaded. It is relatively easy to see this scenario on trace – core jobs will become much longer around statistics engine (STE) calls and the IPC will go down. In order to isolate an STE bottleneck, remove some STE calls and measure the IPC of the application. If the IPC grows significantly, then an STE bottleneck is being experienced.

Below are important rules that developers need to be aware of when implementing statistics counters on AIOP during the design stage of an application. The most important factor that these rules address is DDR bandwidth limitation.

1. Though seemingly simple, it is important to use as few counters as possible. Reduce counters if possible, and make counters optional where possible.
2. Frequently used counters should not be placed in DDR but rather in internal memory (PEB or ShRAM in some cases).
3. If the context is not “read-only” and a lock of some sort (mutex or OSM based) is taken, it is good practice to put counters in that context and update them by the core without using STE (statistics engine). Similarly, if the number of counters is big, it is good practice to update them using the CDMA engine.
4. Use compound STE operations, which allows two counters to be updated in one operation.

There are restrictions on alignments of counters that the STE API has to follow. It can be found at STE section of AIOP Service Layer API Reference Manual.

8.3.7.4.10 IP Fragmentation (IPF)

For best performance it is recommended to work concurrently, and move to Exclusive Mode (XX) ordering only before enqueuing the last fragment. From this point (moving to exclusive before enqueue of the last fragment) transition to concurrent is not allowed. This way fragments of different frames will be interleaved but ordering will be kept between the last fragments of different frames.

8.3.7.4.11 IP Reassembly (IPR)

- Ordering scope
  - Call the IPR in Concurrent mode (XC)
  - Do a per-frame flow distribution, according to the IP identification field
  - Have at least two available OSM scope levels when calling IPR
- Place the lookup tables on the PEB memory
- The input frame (fragment) should be stored in a single buffer
- The frame buffer size should be larger or equal to (16 * <max number of fragments>) + any offset, headroom and annotation
- API configuration parameters
  - Do not enable external statistics
  - Use timeout mode to be per reassembled frame
- Send in-order fragments (relevant to the sender side, usually in closed systems)

Table 157. IPR configuration options for best performance

Flags and options	Best Performance
IPR_MODE_TABLE_LOCATION_PEB	Set
IPR_MODE_EXTENDED_STATS_EN	Cleared
IPR_MODE_IPV4_TO_TYPE	Set
IPR_MODE_IPV6_TO_TYPE	Set

8.3.7.4.12 IPSec

IPSec module supports IPSec encaspulation and decapsulation as a part of service layer. There are several ideas on how to achive the best performance for IPSec:

- Use tunnel mode.
- Use IPv4 frames and outer header.
- Do not enable transport mode pad check.
- Do not use the DSCP set option for tunnel mode.
- Use the system DDR as the IPsec FM context memory (currently not programmable).
- Do not enable UDP encapsulation in transport mode
- Enable reuse buffer mode

The following table describes the IPsec functional module configuration parameters value for achieving the best performance.

Table 158. IPsec Configuration Options for Best Performance

Flags & options	Best performance
IPSEC_FLG_TUNNEL_MODE	Set
IPSEC_FLG_TRANSPORT_PAD_CHECK	Cleared
IPSEC_FLG_BUFFER_REUSE	Set
IPSEC_ENC_OPTS_NAT_EN	Cleared
IPSEC_ENC_OPTS_NUC_EN	Cleared
IPSEC_FLG_ENC_DSCP_SET	Cleared
IPSEC_FLG_LIFETIME_KB_CNTR_EN	Cleared
IPSEC_FLG_LIFETIME_PKT_CNTR_EN	Cleared
IPSEC_FLG_LIFETIME_SEC_CNTR_EN	Cleared
IPSEC_OPTS_ESP_ESN	Cleared
IPSEC_OPTS_ESP_IPVSN	Cleared
IPSEC_DEC_OPTS_ARSNONE	Set
IPSEC_DEC_OPTS_ARS32	Cleared
IPSEC_DEC_OPTS_ARS128	Cleared
IPSEC_DEC_OPTS_ARS64	Cleared

8.3.7.4.13 Appendix A

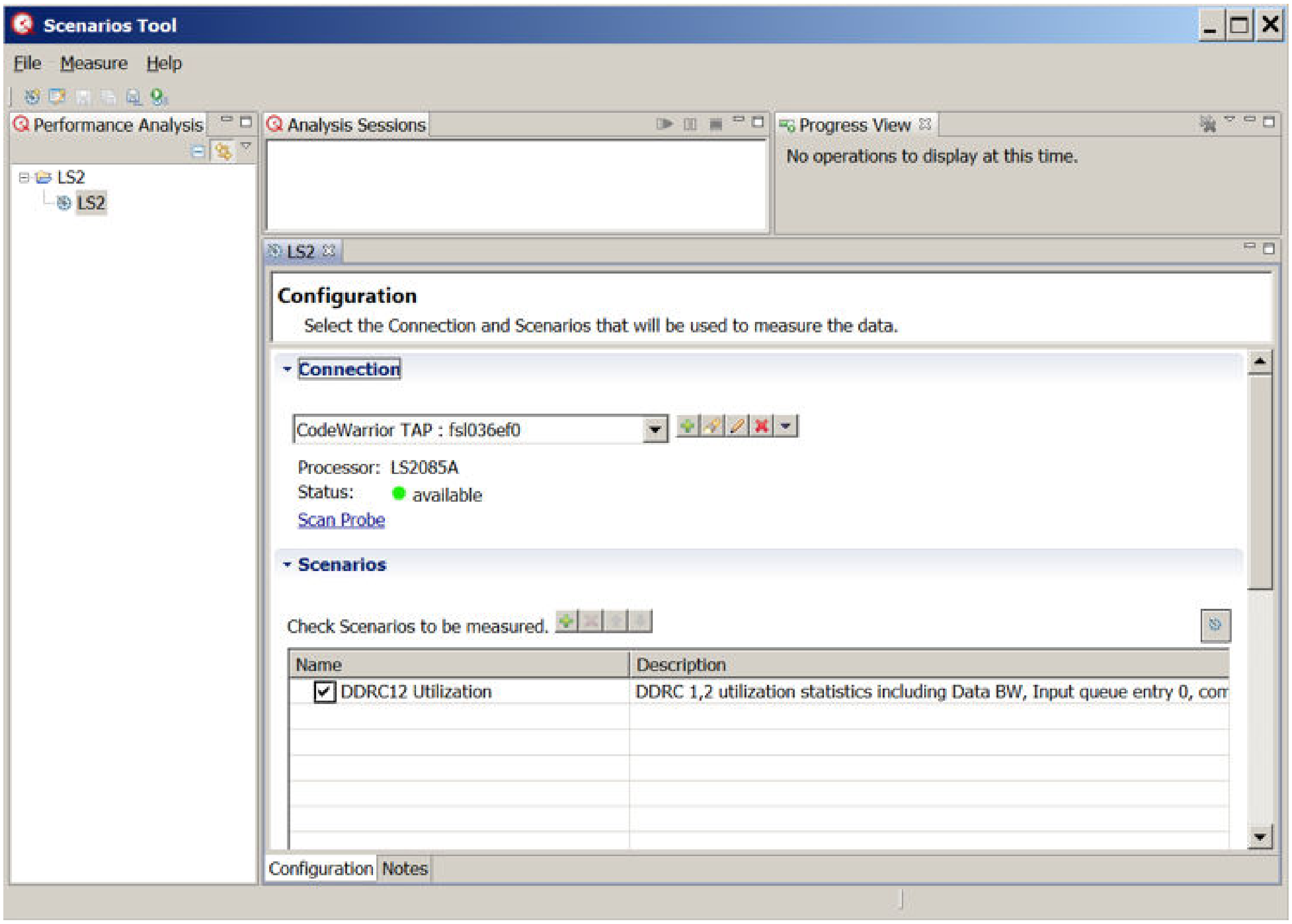
DDR bandwidth measurement

The Scenarios Tools allows a real time measurement of DDR bandwidth for all the controllers. Use this tool as a help guide for the configuration procedure.

As an example, in the LS2085A, there are three DDR controllers. In ***Scenarios Tools*** those are called DDRC1, DDRC2 and DDRC3.

DDRC1 and DDRC2 are for system memory and DDRC3 is for DP-DDR.

Add the utilization measurement as shown below, then press the green “Launch” button in the toolbar.



No operations to display at this time.

Configuration

Select the Connection and Scenarios that will be used to measure the data.

Connection

CodeWarrior TAP : fsl036ef0

Processor: LS2085A

Status: ● available

Scan Probe

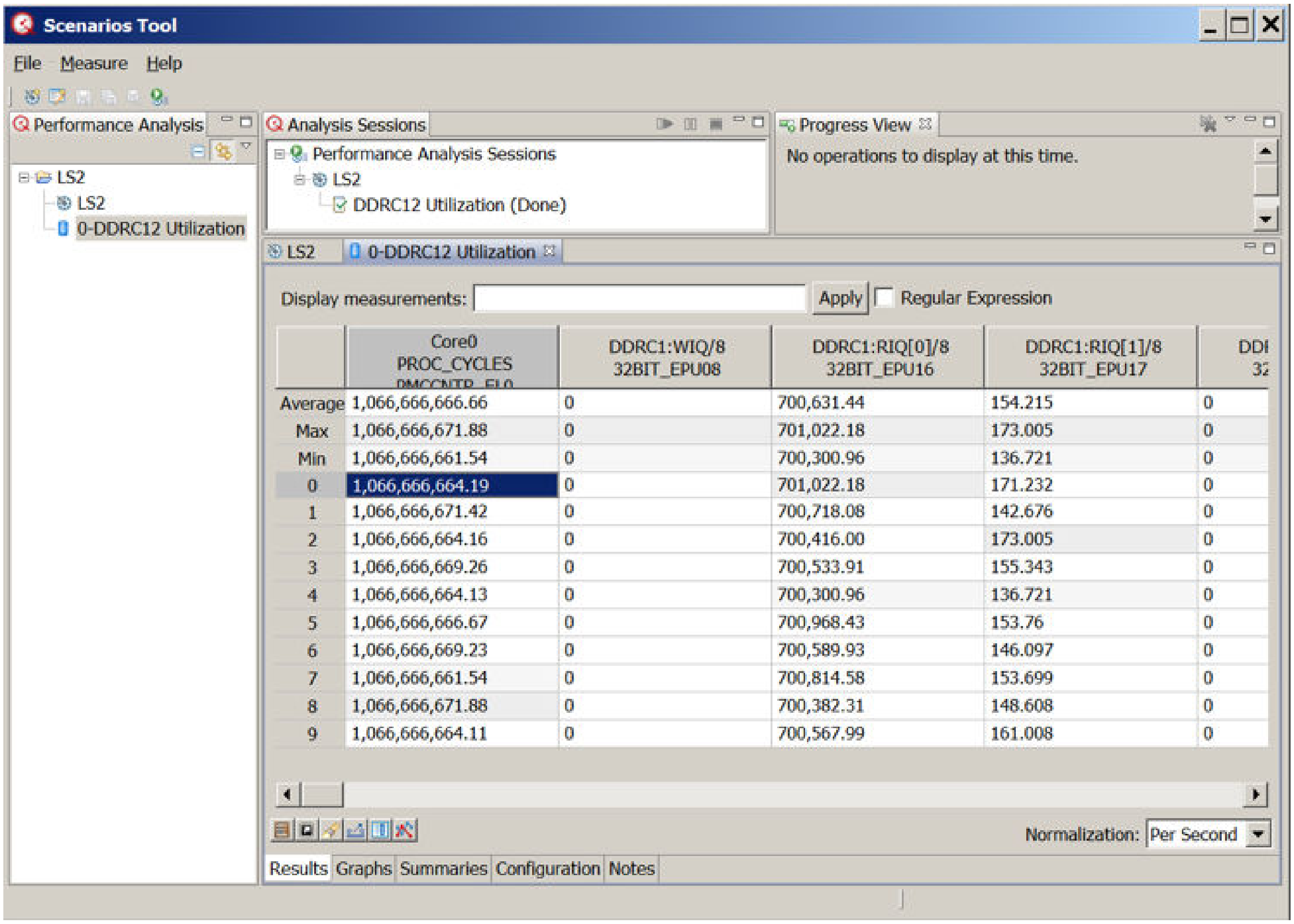
Scenarios

Check Scenarios to be measured.

Name	Description
☑ DDRC12 Utilization	DDRC 1,2 utilization statistics including Data BW, Input queue entry 0, com

Figure 191. Utilization Measurement in Scenarios Tool

The measurement will complete in a few seconds, and the table of results will be displayed, as shown below.



No operations to display at this time.

Display measurements:

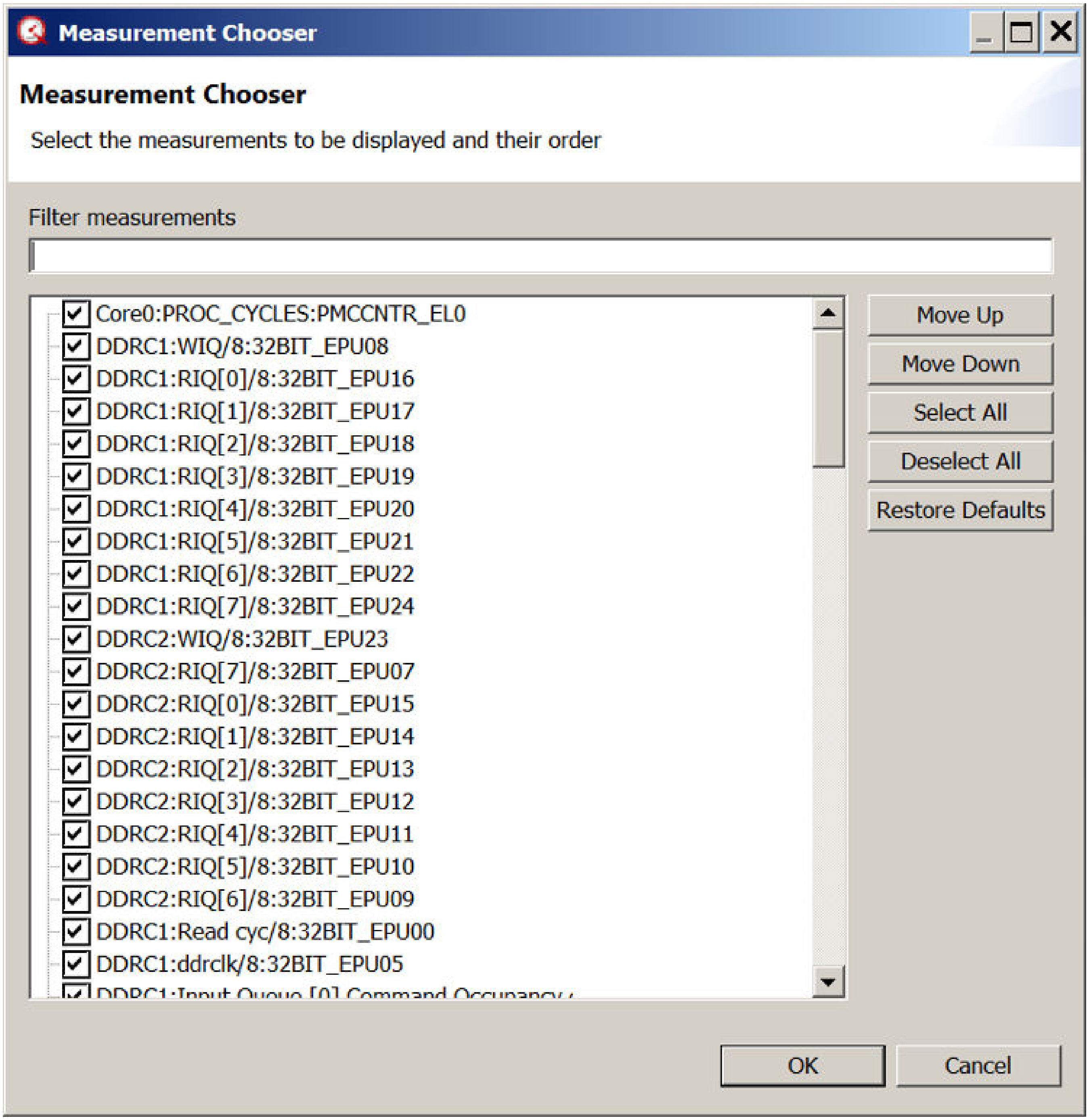
□ Regular Expression

	Core0 PROC_CYCLES PMCCNTR_EL0	DDRC1:WIQ/8 32BIT_EPU08	DDRC1:RIQ[0]/8 32BIT_EPU16	DDRC1:RIQ[1]/8 32BIT_EPU17	DDI 32
Average	1,066,666,666.66	0	700,631.44	154.215	0
Max	1,066,666,671.88	0	701,022.18	173.005	0
Min	1,066,666,661.54	0	700,300.96	136.721	0
0	1,066,666,664.19	0	701,022.18	171.232	0
1	1,066,666,671.42	0	700,718.08	142.676	0
2	1,066,666,664.16	0	700,416.00	173.005	0
3	1,066,666,669.26	0	700,533.91	155.343	0
4	1,066,666,664.13	0	700,300.96	136.721	0
5	1,066,666,666.67	0	700,968.43	153.76	0
6	1,066,666,669.23	0	700,589.93	146.097	0
7	1,066,666,661.54	0	700,814.58	153.699	0
8	1,066,666,671.88	0	700,382.31	148.608	0
9	1,066,666,664.11	0	700,567.99	161.008	0

Normalization: Per Second

Figure 192. Utilization Measurement Table of Results

The initial table display will show all measured events and metrics. Use the Measurement Chooser dialog to select the measurements of interest, in this case the utilization metrics.



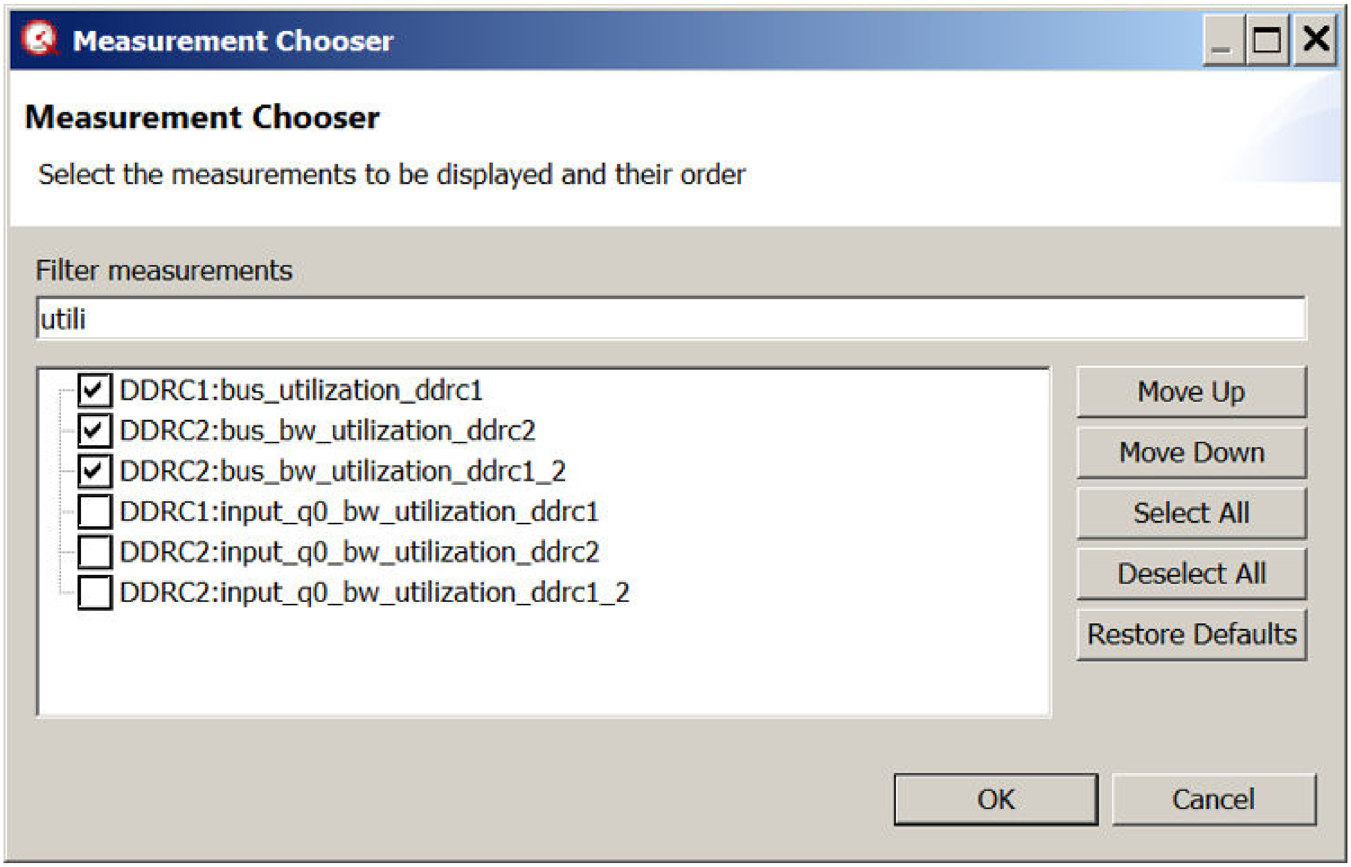
Measurement Chooser

Select the measurements to be displayed and their order

Filter measurements

☑ Core0:PROC_CYCLES:PMCCNTR_EL0
☑ DDRC1:WIQ/8:32BIT_EPU08
☑ DDRC1:RIQ[0]/8:32BIT_EPU16
☑ DDRC1:RIQ[1]/8:32BIT_EPU17
☑ DDRC1:RIQ[2]/8:32BIT_EPU18
☑ DDRC1:RIQ[3]/8:32BIT_EPU19
☑ DDRC1:RIQ[4]/8:32BIT_EPU20
☑ DDRC1:RIQ[5]/8:32BIT_EPU21
☑ DDRC1:RIQ[6]/8:32BIT_EPU22
☑ DDRC1:RIQ[7]/8:32BIT_EPU24
☑ DDRC2:WIQ/8:32BIT_EPU23
☑ DDRC2:RIQ[7]/8:32BIT_EPU07
☑ DDRC2:RIQ[0]/8:32BIT_EPU15
☑ DDRC2:RIQ[1]/8:32BIT_EPU14
☑ DDRC2:RIQ[2]/8:32BIT_EPU13
☑ DDRC2:RIQ[3]/8:32BIT_EPU12
☑ DDRC2:RIQ[4]/8:32BIT_EPU11
☑ DDRC2:RIQ[5]/8:32BIT_EPU10
☑ DDRC2:RIQ[6]/8:32BIT_EPU09
☑ DDRC1:Read cyc/8:32BIT_EPU00
☑ DDRC1:ddrclk/8:32BIT_EPU05
☑ DDRC1:Input Queue [0] Command Occupancy (

Figure 193. Measurement Chooser Dialog



Measurement Chooser

Measurement Chooser

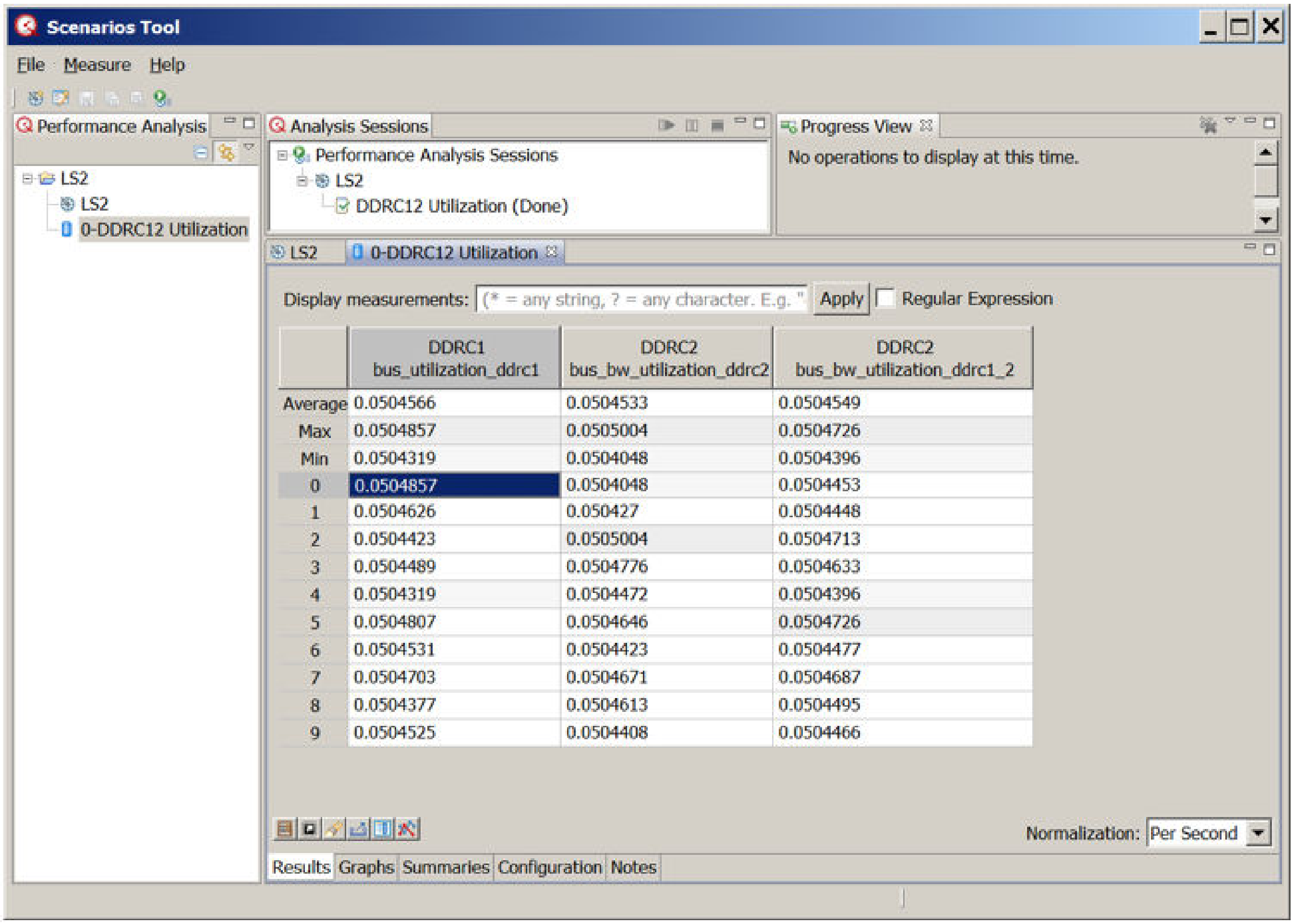
Select the measurements to be displayed and their order

Filter measurements

utili

☑ DDRC1:bus_utilization_ddrc1
☑ DDRC2:bus_bw_utilization_ddrc2
☑ DDRC2:bus_bw_utilization_ddrc1_2
□ DDRC1:input_q0_bw_utilization_ddrc1
□ DDRC2:input_q0_bw_utilization_ddrc2
□ DDRC2:input_q0_bw_utilization_ddrc1_2

Figure 194. Selecting Utilization Metrics



No operations to display at this time.

Display measurements: (* = any string, ? = any character. E.g. " Apply □ Regular Expression

	DDRC1 bus_utilization_ddrc1	DDRC2 bus_bw_utilization_ddrc2	DDRC2 bus_bw_utilization_ddrc1_2
Average	0.0504566	0.0504533	0.0504549
Max	0.0504857	0.0505004	0.0504726
Min	0.0504319	0.0504048	0.0504396
0	0.0504857	0.0504048	0.0504453
1	0.0504626	0.050427	0.0504448
2	0.0504423	0.0505004	0.0504713
3	0.0504489	0.0504776	0.0504633
4	0.0504319	0.0504472	0.0504396
5	0.0504807	0.0504646	0.0504726
6	0.0504531	0.0504423	0.0504477
7	0.0504703	0.0504671	0.0504687
8	0.0504377	0.0504613	0.0504495
9	0.0504525	0.0504408	0.0504466

Normalization: Per Second

Figure 195. Example of DDR Utilization

In last figure, observe that DDR1 and DDR2 are only 5% utilized. Generally, utilization of lower than 65% is considered to be low and not affecting latency.

8.3.7.5 AIOP Service Layer API Reference Manual

Click here to access the AIOP Service Layer API Reference Manual PDF.

8.3.7.6 AIOP SDK Applications Debug

Click here to access the AIOP SDK Applications Debug PDF.

8.4 Packet Forward Engine (PFE) Network Driver

8.4.1 Introduction

8.4.1.1 Overview

This section describes the Linux driver which enables support for Ethernet on Packet Forward Engine (PFE) hardware. EMACs are part of PFE IP, to receive/transmit packets through EMAC interface it should be accessed through PFE interface by programing it.

8.4.1.2 Purpose

The purpose of this section is to provide a user guide and configuration details for the PFE driver, and a high-level view of the driver's structure, as well as to describe its major functionalities with a focus on the features provided by the PFE IP.

8.4.1.3 Features

This section provides an overview of the major PFE features:

- MAC Layer.
- MAC Address Filter.
- Interrupt for Tx/Rx packets.
- Scatter/Gather support.
- Interrupt coalescing.
- TCP/UDP checksum verification and generation.

8.4.2 High level decomposition and data flow

A system level block view, from a network device perspective, may be depicted as follows:

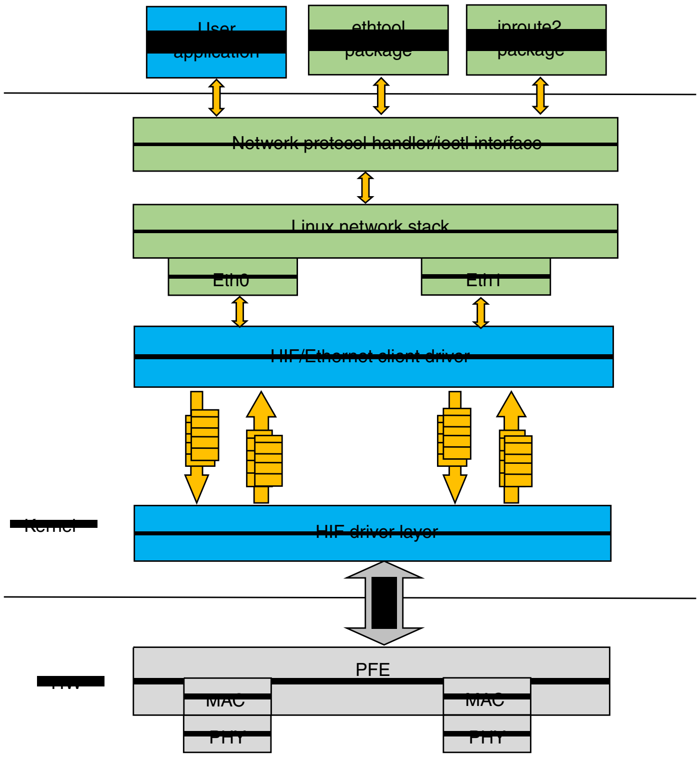


Figure 196. High level decomposition and data flow block level view

The PFE, MAC and PHY are the hardware blocks, the kernel networking stack along with the network driver are running in the Kernel space, and finally ethtool and iproute2 are examples of user space tools used for configuring the network devices.

The PFE hardware supports one HIF RX and TX descriptor queues to send and receive packets through PFE. Both network interface traffic is multiplexed and send over Host Interface (HIF) queue.

User space packages like ethtool and iproute are used to configure the network device parameters. The ethtool interface is extended to provide support for filer programming. The kernel space module for the network driver is the most important block as it communicates with both the user space and the H/W IP to control the processing of packets.

The basic functionality of any Ethernet driver is to handle the reception of packets from an ingress port (might include checksum calculation, header verification, etc), as well as the transmission of packets on the egress port (might include checksum re-calculation, header manipulation, etc). There are also the device configuration and control functionalities, and device status reporting. When the Ethernet driver is actually implementing these functionalities, it needs to interact with the core (Kernel) as well as the hardware IP (the Ethernet controller).

The PFE Linux kernel module has following two main parts:

- **HIF driver layer:**This part of the driver talks with HIF hardware interface and send and receive the packets from it. It receives packets from HIF interface and identifies from which MAC interface it received and send the packet to corresponding client driver queue. Similarly, if there is any pending packet from client queue to transmit packet it takes and inserts the HIF header and put it into the HIF queue. It uses the NAPI to receive packets and send it to corresponding client queues and triggers client to process packets from the queue.
- **HIF/Ethernet client driver:** Ethernet client driver is a hardware independent driver and registers with the HIF driver to transmit and receive packet through HIF interface. For each interface one instance of client driver should be register with the HIF driver layer, other side it registers with Linux kernel stack as network interface. Each client driver will have software queues to communicate with HIF driver layer. Each client driver registers with NAPI and indicate packets to the stack through the NAPI poll.

8.4.3 NAPI support

PFE HIF driver layer uses NAPI handling for Rx path processing, the Linux polling mechanism being triggered by frame receive interrupts. The driver registers irqs for receive and the NAPI (polling) handlers are provided to the Kernel. Similarly, HIF Ethernet client driver also uses NAPI handling to processes software queues and pass them to the Kernel Network stack.

On the receive path:

- When the receive interrupt gets triggered, a softirq for the polling function on Rx is scheduled.
- The RX_SOFTIRQ thread is raised by the Kernel, and the HIF Rx queues will be processed by the driver's polling function and the incoming packets are being passed to client Rx queues and triggers the client NAPI handling.
- HIF/Ethernet client NAPI poll receives packets from client Rx queues and passes to the Network stack.

8.4.4 Interrupt coalescing

On a high speed network interface the rate of packet reception and transmission can be as high as the CPUs would be spending most of the time servicing these interrupts. With the interrupt coalescing feature, packets are collected and one single interrupt is generated for multiple packets to avoid flooding the system with interrupts from the Ethernet device.

PFE hardware supports hardware coalescing for receive interrupts, complemented by timer-based thresholds. PFE driver provides basic support for setting the coalescing parameters via ethtool -C by implementing the “rx-usec” option.

8.4.5 Checksum offloading

For large frames, offload of checksum verification saves a significant fraction of the CPU cycles that would otherwise be spent by the TCP/IP stack. IP packet fragmentation and re-assembly, and TCP stream establishment and tear-down are not performed in hardware.

On Tx side, PFE hardware provides IPv4/IPv6 and TCP/UDP header checksum generation. On the Rx side, PFE driver lets the Kernel know that checksum verification is not required if valid IP headers or TCP/UDP headers were found and valid sums were verified, by setting the CHECKSUM_UNNECESSARY flag. On Tx side, the checksum is generated (offloaded) for TCP/UDP packets over IPv4 based on the pseudo-header checksum (phcs) provided by the Linux networking stack. PFE Linux driver instructs the stack about its ability to provide partial checksumming, based on the phcs for TCP/UDP packets, by setting the NETIF_F_IP_CSUM device capability flag. PFE hardware doesn't support per packet based checksum calculation control, it should be enabled or disabled for all packets.

8.4.6 Scatter gather support

Scatter-Gather I/O is a method by which a single procedure call sequentially writes data from multiple buffers to a single data stream or reads data from a data stream to multiple buffers. The buffers are given in a vector of buffers. Scatter/gather refers to the process of gathering data from, or scattering data into, the given set of buffers. The I/O can be performed synchronously or asynchronously to this procedure.

On the Tx side, PFE HIF interface supports "gathering" big packets from multiple buffers. This ability is signaled by the driver to the Linux network stack by setting the NETIF_F_SG device hardware feature flag. The driver takes into account the number of fragments composing the packet that is going to be transmitted, and places each fragment into consecutive BD ring buffers before issuing the command to start sending the frame.

On the Rx side, the PFE HIF interface is capable of "scattering" big packets into multiple fixed size buffers having consecutive buffer descriptors (BDs).

8.4.7 Ethtool support

Non-exhaustive list of the most notable ethtool commands implemented by PFE Linux driver:

```
-C | --coalesce DEVNAME [rx-usecs N]
```

Sets Rx interrupt coalescing in microsecs('usecs').

```
-K | --offload DEVNAME
```

Sets UDP/TCP checksum offloading enabled or disabled.

- `rx on|off` - Specifies whether RX checksum is enabled or disabled.
- `tx on|off` - Specifies whether TX checksum is enabled or disabled.

```
-S | --statistics DEVNAME
```

Queries the specified network device for NIC- and driver-specific statistics.

```
-s DEVNAME
```

Allows changing some or all settings of the specified network device. All following options only apply if -s was specified.

- `wol g` - Sets Wake-on-LAN options. The argument to this option is a string of characters specifying which options to enable.

```
-A|--pause devname
```

[tx on|off] Specifies whether TX pause should be enabled.

8.5 Linux Ethernet Driver for eTSEC

8.5.1 Linux Ethernet Driver for eTSEC

8.5.1.1 Introduction

8.5.1.1.1 Overview

Gianfar is the Linux driver that enables Ethernet support for the SoCs featuring eTSEC (Enhanced Three-Speed Ethernet Controllers). Though the driver is designed to support the latest eTSEC2.0 features present on the low-power QorIQ platforms, it also maintains backward compatibility with older IPs from the same family, like eTSEC (eTSEC 1.x) and TSEC (present on the PowerQUICC III platforms) and FEC (Fast Ethernet Controller).

8.5.1.1.2 Purpose

The purpose of this document is to provide a user guide and configuration details for the Gianfar driver, and a high-level view of the driver's structure, as well as to describe its major functionalities with a focus on the features provided by the eTSEC2.0 IP.

8.5.1.1.3 Features

This section provides an overview of the major eTSEC2.0 ("virtualized" eTSEC) features:

- o MAC Layer
- *o Interrupt grouping mechanism*
- *o Virtualized register space*
- o Rx Subsystem:
  - — MAC Address Filter
  - — L2/L3/L4 Parser
  - — Filer Engine
  - — *Hash or RR Distribution*
  - — *Multiple Rx Interrupt*
- o Tx Subsystem:
  - — Tx Scheduler
  - — L3/L4 Offload
  - — *Multiple Tx Interrupt*

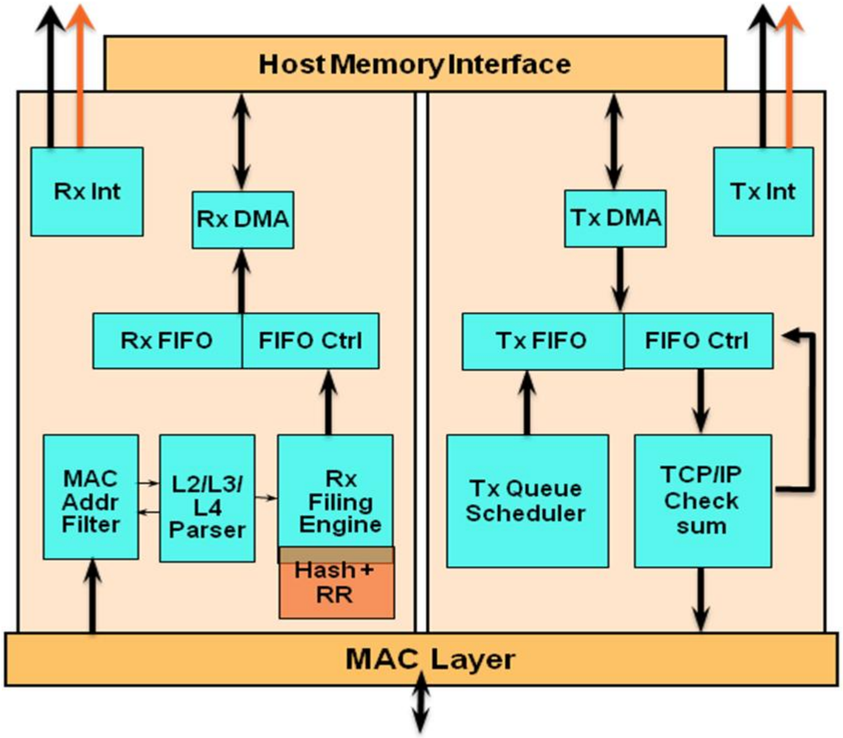


Figure 197. “Virtualized” eTSEC Block Diagram

- *o Interrupt virtualization:*
  - — Each ring maps to one of two separate groups for interrupt and BD management; each group associated by software with a CPU.
  - — Separate address spaces per group and for MDIO.
  - — Interrupt coalescing controls per ring in multi-group mode, packet-count based and timer based thresholds, for both Rx and Tx.
- *o TCP/IP Offload Engine (TOE):*
  - — IP v4 and IP v6 header recognition on receive
  - — IP v4 header checksum verification and generation
  - — TCP and UDP checksum verification and generation
  - — Per-packet configurable offload
  - — Recognition of VLAN, stacked-VLAN, 802.2, PPPoE session, MPLS stacks, and ESP/AH IP-Security headers
- *o Quality of service (QoS) support:*
  - — Transmission from up to eight queues: priority-based queue selection or modified weighted round-robin (MWRR) queue selection with fair bandwidth allocation
  - — Reception to up to eight physical queues:

    - Table-oriented queue filing strategy based on 16 header fields or flags

    - Frame rejection support for filtering applications

    - Filing based on Ethernet, IP, and TCP/UDP properties, including VLAN fields, Ether-type, IP protocol type, IP TOS or differentiated services, IP source and destination addresses, TCP/UDP port number

8.5.1.1.4 Notes on high level decomposition and data flow

A system level block view, from a network device perspective, may be depicted as follows:

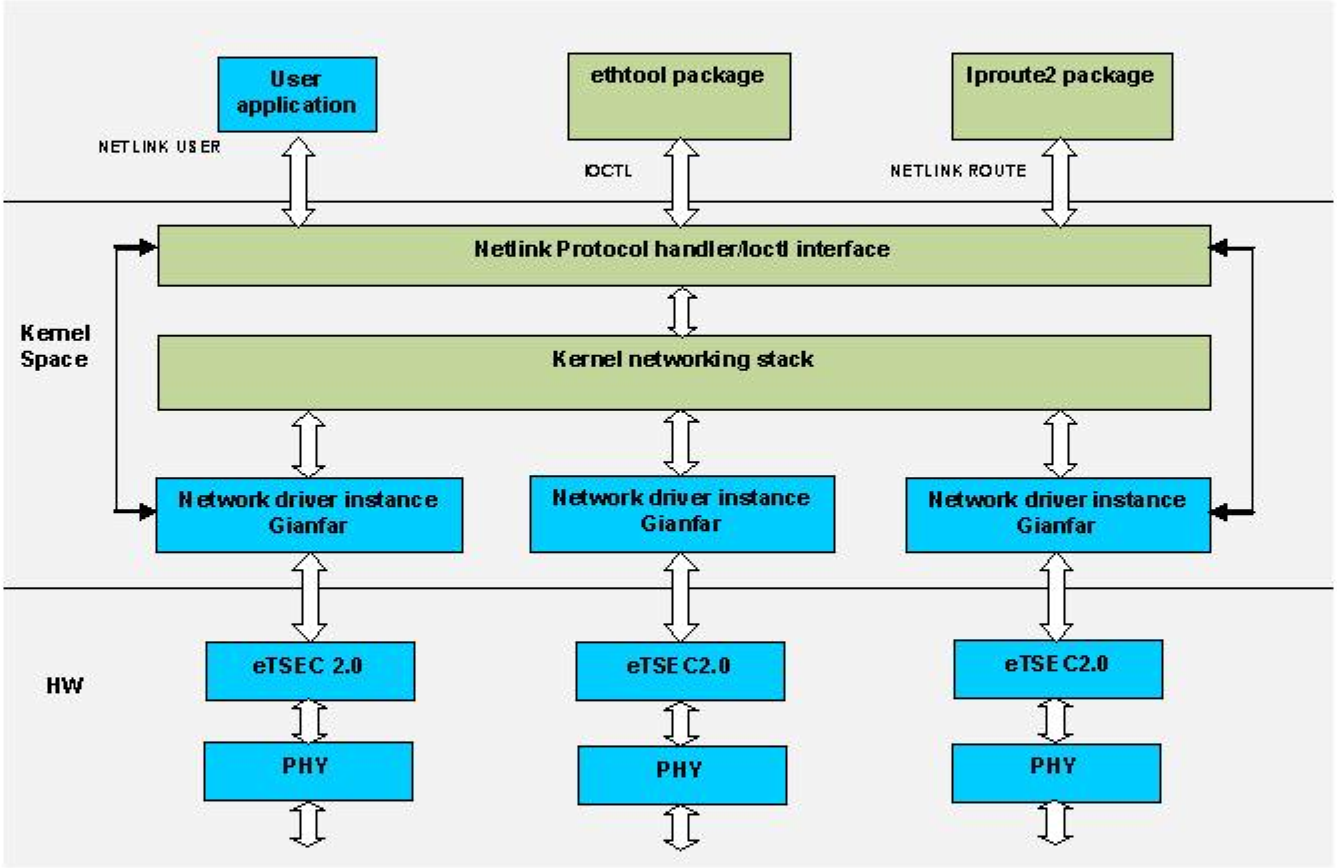


Figure 198. Gianfar High level decomposition

The eTSEC2.0 and PHY are the hardware blocks, the kernel networking stack along with the network driver are running in the Kernel space, and finally ethtool and iproute2 are examples of user space tools used for configuring the network devices.

The eTSEC2.0 includes some additional support compared with the previous versions:

- it has support for interrupt virtualization
- on the TX side, it can distribute packets to the multiple queues based on simple hashing or round robin mechanisms

The eTSEC2.0 has support for multiple RX and TX queues. On the receive side, an incoming packet will be filed to one of the queues based on the rules programmed into the filer. By default, all the packets will be filed to queue 0. On the transmit side, either a simple hash based implementation or a round robin algorithm distributes the packets to the available number of queues. User space packages like ethtool and iproute are used to configure the network device parameters. The ethtool interface is extended to provide support for filer programming.

The kernel space module for the network driver is the most important block as it communicates with both the user space and the H/W IP to control the processing of packets. The eTSEC network device driver will be referred to as Gianfar in the rest of the document.

The Gianfar driver may be divided into sub-blocks based on the number of independent threads that Linux will run in order to completely transfer a packet from ingress to egress side. The basic functionality of any Ethernet driver is to handle the reception of packets from an ingress port (might include checksum calculation, header verification, etc), as well as the transmission of packets on the egress port (might include checksum re-calculation, header manipulation, etc). There are also the device configuration and control functionalities, and device status reporting. When the Ethernet driver is actually implementing these functionalities, it needs to interact with the core (Kernel) as well as the hardware IP (the Ethernet controller).

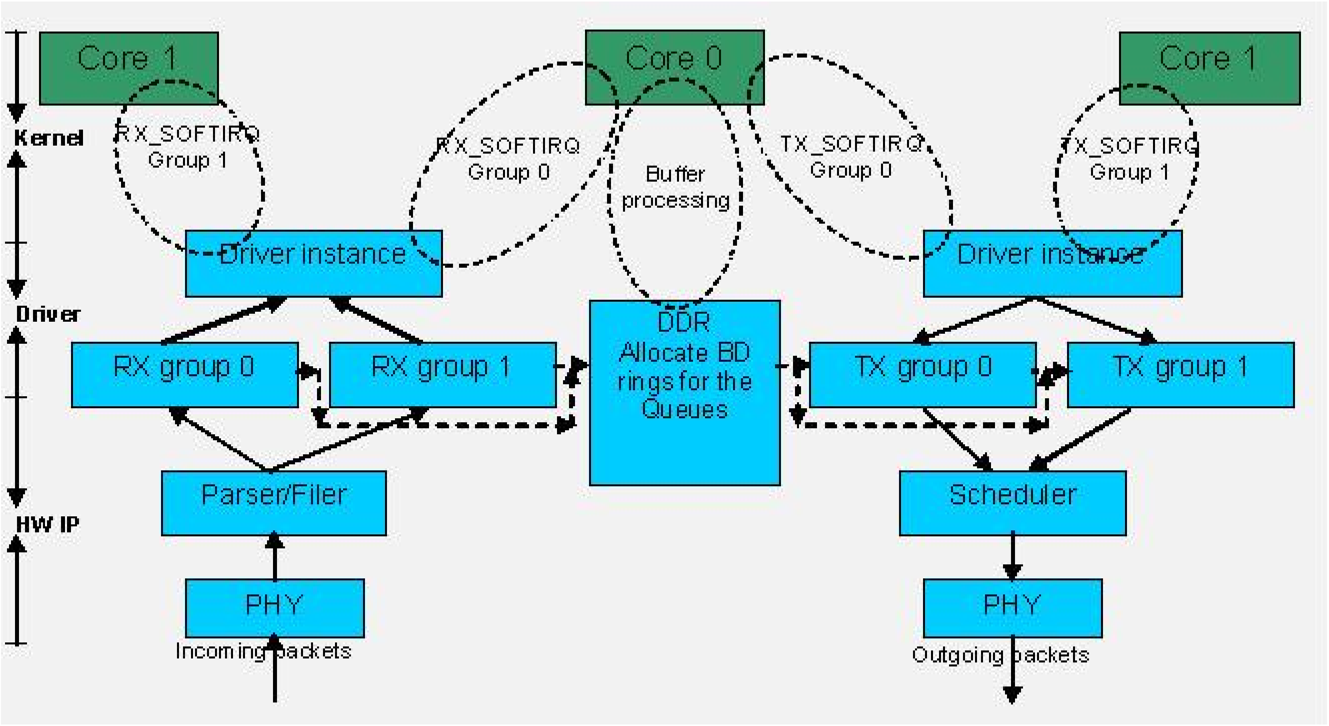


Figure 199. Gianfar Packet Flow

In the above Figure it can be noted that the receive side includes parsing/ filing before a packet is "put" into a buffer descriptor. The transmit side includes a H/W queue scheduler for transmission of packets.

As already mentioned, eTSEC2.0 has support for multiple hardware queues for Rx and Tx in hardware. These queues are basically divided into two groups; let’s say all odd numbered queues correspond to one group and even numbered queues correspond to other group. In a multi core environment (e.g. a dual core system), each group of queues can be programmed to be handled by one of the two cores, which will result in an increased performance.

For simplicity, we always assume that:

- All the even numbered queues are mapped to Group 0 and odd numbered queues are mapped to Group1.
- Group 0 interrupts can be assigned to be processed by Core 0 and Group 1 interrupts to be processed by Core 1 (except for error the interrupts, which are always destined to Core 0, which is the master core).

From the above figure, it can be noticed that there will be a receive, and a transmit thread running on each core, for processing the packets corresponding to the group assigned to that core. The receive thread processes the received packets - handles the RX buffer descriptor (BD) rings, and passes the received packets to the networking stack for further processing; the transmit thread schedules the packets passed down by the stack to be transmitted out of the device. There's also a transmission cleanup thread, triggered by the TX confirmation interrupts, to handle the TX BD rings and congestion.

Gianfar may be broadly decomposed into the following sub-blocks:

1. Initialization block
2. Receive block
3. Transmit block
4. Control block

So, the Receive and Transmit blocks handle processing of ingress and egress packets.

Before processing packets the driver needs to perform some initialization steps like:

- extracting the device tree parameters
- initialization of the multiple queues
- registering the driver with the kernel
- allocating buffer descriptors
- registering the interrupts (etc.)

All these functionalities are implemented by the Initialization block.

Each of these sub-modules implements various functionalities, as detailed in the coming section.

8.5.1.2 Functionality

8.5.1.2.1 Multi-Queue support

eTSEC features multiple physical queues or BD rings. The multi-queue support (MQ) in the driver is enabled by default for eTSEC2.0 IPs.

Hardware queue events are mapped to one of the two available CPUs via eTSEC *Interrupt Groups*. For eTSEC2.0, each Rx/Tx hardware queue or BD ring is mapped to one of the two available Interrupt Groups, and each group in turn has its Rx/Tx interrupt lines assigned to a given CPU. By default, the driver enables 1 Rx and 1 Tx queue per Interrupt Group.

eTSEC2.0 supports 2 Interrupt Groups, this is also known as the Multi-Group (MG) mode in Gianfar. Each group has its own Rx, Tx and Err interrupt lines which can be individually affined to any of the 2 CPUs, as a measure to balance the processing load. Also, each interrupt group has its own block of registers, most notably `ievent`, `imask`, `tstat`, and `rstat`, so queue events are handled at the interrupt group level. Having more than 1 Rx and 1 Tx queue assigned to a single interrupt group would thus incur a software processing overhead that would not be justifiable for the majority of use cases. This is why the driver enables by default only 1 set of Rx and Tx queues per Interrupt Group.

eTSEC1.x and other older eTSEC IPs support only one interrupt group (g0), meaning that they are working in Single Group (SG) mode.

The mapping Rx/Tx queues to interrupt groups is by default: Rx Q0 and Tx Q0 assigned to Group0 (`g0`), and Rx Q1 and Tx Q1 assigned to Group1 (`g1`).

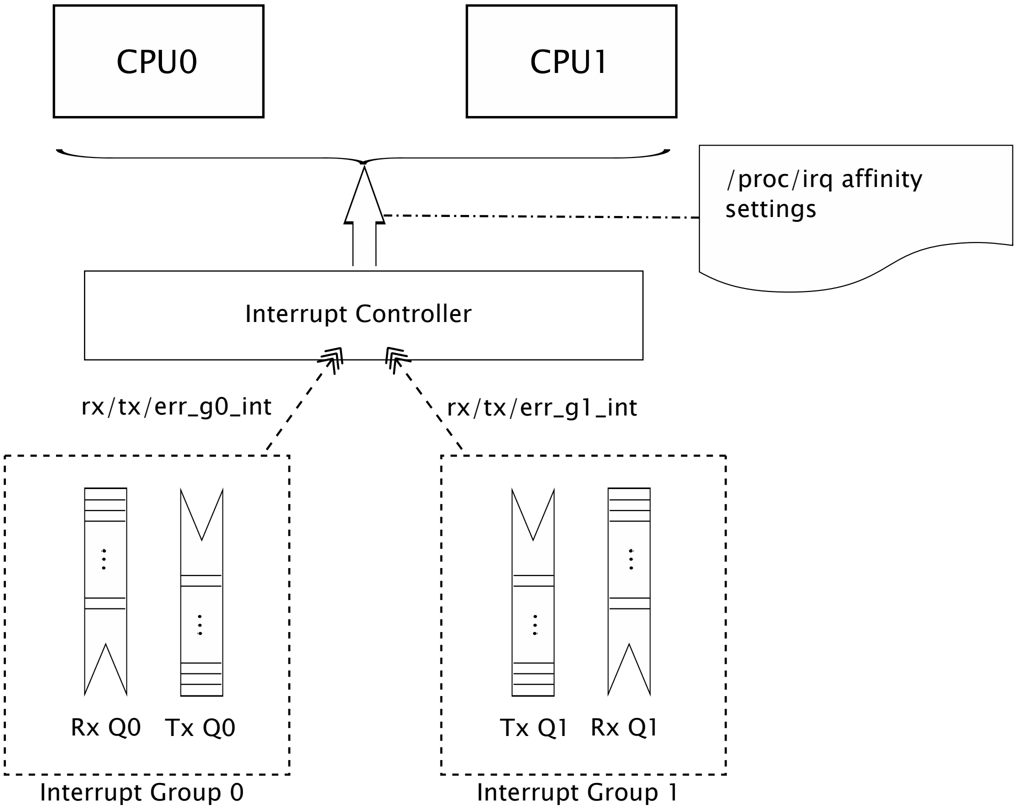


Figure 200. Multi-Queue Multi-Group

NOTE

Supporting more than one Rx/Tx queue per interrupt group has been obsoleted (see above). As a result, the following device tree properties are *obsolete*: `fsl,num-rx-queues`, `fsl,num-tx-queues`, `fsl,rx-bit-map`, and `fsl,tx-bit-map`.

8.5.1.2.2 Receive Side Scaling support

eTSEC supports multiple Rx and Tx descriptor queues (see multi-queue support). On reception, eTSEC can send different packets to different queues to distribute processing among CPUs. This mechanism is generally known as “Receive-side Scaling” (RSS).

In Gianfar, packets are distributed by applying "n-tuple" filters configured from `ethtool -N` (`--config-ntuple` option). These filters are converted by Gianfar to eTSEC Filer H/W rules. Based on these programmable filters, each packet is assigned to one of a small number of logical flows. Packets for each flow are steered to separate receive queues, which in turn can be processed by separate CPUs.

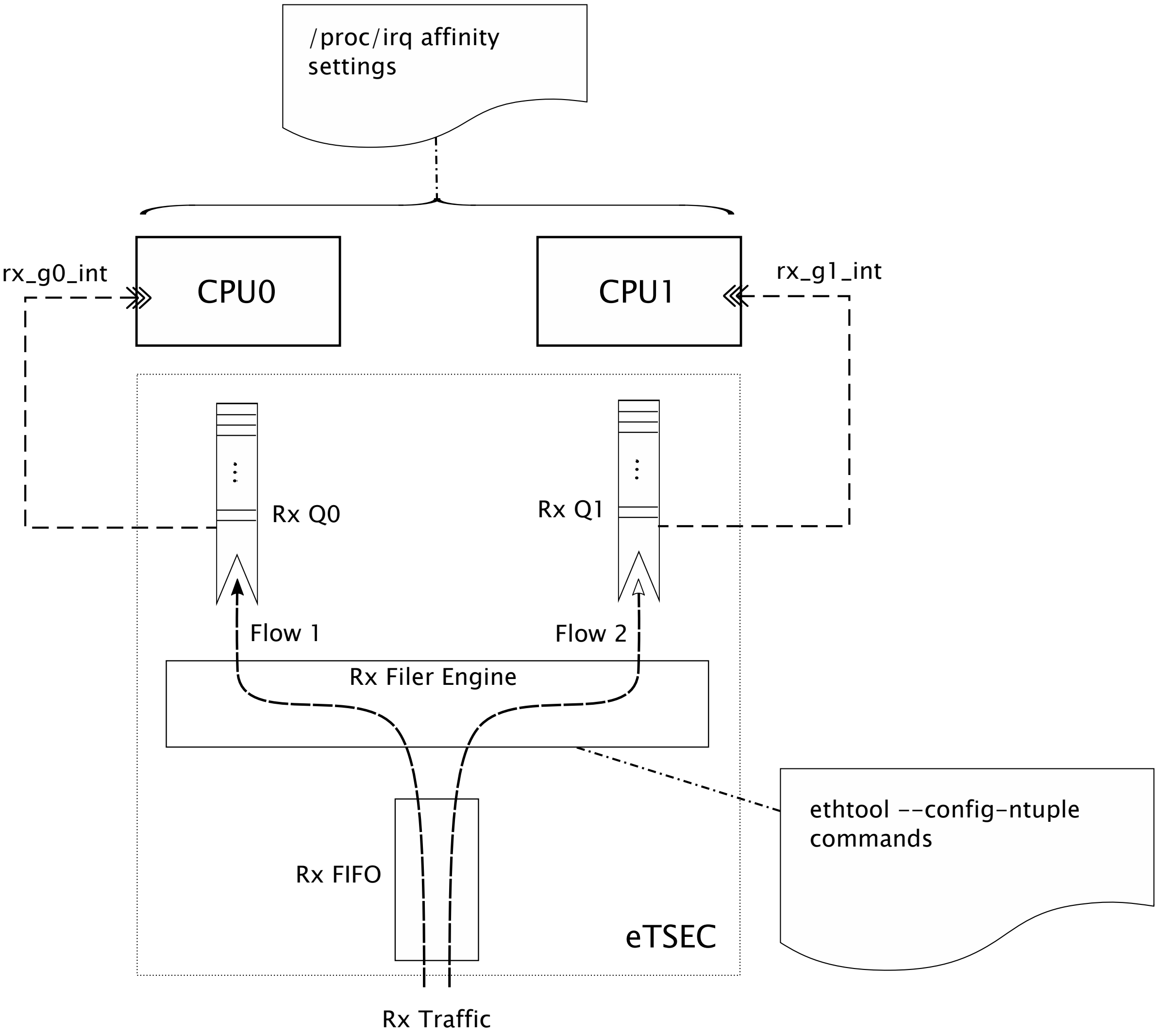


Figure 201. eTSEC RSS support

In Gianfar, Rx flows may be classified either by hashing various protocol header fields, see `ethtool -N rx-flow-hash` option, or by specifying flow type classification rules, see `ethtool -N flow-type` option. Refer to ethtool Linux man-pages for `ethtool -N` option details. A simple usage example is shown below.

```
root@ls1021aqds:~# ethtool -N eth0 flow-type udp4 src-ip 172.16.1.4 dst-port 5000 action 0
fsl-gianfar ethernet.4 eth0: Receive Queue Filtering enabled
Added rule with ID 254
root@ls1021aqds:~# ethtool -N eth0 flow-type udp4 src-ip 172.16.1.4 dst-port 5001 action 1
Added rule with ID 253
root@ls1021aqds:~# ethtool -n eth0
2 RX rings available
Total 2 rules

Filter: 253
        Rule Type: UDP over IPv4
        Src IP addr: 172.16.1.4 mask: 0.0.0.0
        Dest IP addr: 0.0.0.0 mask: 255.255.255.255
        TOS: 0x0 mask: 0xff
        Src port: 0 mask: 0xffff
        Dest port: 5001 mask: 0x0
        Action: Direct to queue 1

Filter: 254
        Rule Type: UDP over IPv4
        Src IP addr: 172.16.1.4 mask: 0.0.0.0
        Dest IP addr: 0.0.0.0 mask: 255.255.255.255
        TOS: 0x0 mask: 0xff
        Src port: 0 mask: 0xffff
        Dest port: 5000 mask: 0x0
        Action: Direct to queue 0

root@ls1021aqds:~# iperf -s -u -p 5000 &
[1] 1017
------------------------------------------------------------
Server listening on UDP port 5000
Receiving 1470 byte datagrams
UDP buffer size:  160 KByte (default)
------------------------------------------------------------
root@ls1021aqds:~# iperf -s -u -p 5001 &
[2] 1020
------------------------------------------------------------
Server listening on UDP port 5001
Receiving 1470 byte datagrams
UDP buffer size:  160 KByte (default)
------------------------------------------------------------
root@ls1021aqds:~# cat /proc/interrupts | grep eth
176:          7          0       GIC 176  eth0_g0_tx
177:          6          0       GIC 177  eth0_g0_rx
178:          0          0       GIC 178  eth0_g0_er
179:          0          0       GIC 179  eth0_g1_tx
180:          0          0       GIC 180  eth0_g1_rx
181:          0          0       GIC 181  eth0_g1_er

[  3] local 172.16.1.100 port 5000 connected with 172.16.1.4 port 52163
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.003 ms    0/  893 (0%)

root@ls1021aqds:~# cat /proc/interrupts | grep eth
176:          9          0       GIC 176  eth0_g0_tx
177:        902          0       GIC 177  eth0_g0_rx
178:          0          0       GIC 178  eth0_g0_er
179:          1          0       GIC 179  eth0_g1_tx
180:          0          0       GIC 180  eth0_g1_rx
181:          0          0       GIC 181  eth0_g1_er

[  3] local 172.16.1.100 port 5001 connected with 172.16.1.4 port 46257
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.004 ms    0/  893 (0%)

root@ls1021aqds:~# cat /proc/interrupts | grep eth
176:         10          0       GIC 176  eth0_g0_tx
177:        902          0       GIC 177  eth0_g0_rx
178:          0          0       GIC 178  eth0_g0_er
179:          1          0       GIC 179  eth0_g1_tx
180:        894          0       GIC 180  eth0_g1_rx
181:          0          0       GIC 181  eth0_g1_er
root@ls1021aqds:~# echo 1 > /proc/irq/177/smp_affinity
root@ls1021aqds:~# echo 2 > /proc/irq/180/smp_affinity
root@ls1021aqds:~# iperf -s -u -p 1000 &
[3] 1031
------------------------------------------------------------
Server listening on UDP port 1000
Receiving 1470 byte datagrams
UDP buffer size:  160 KByte (default)
------------------------------------------------------------
[  3] local 172.16.1.100 port 1000 connected with 172.16.1.4 port 58669
[ ID] Interval       Transfer     Bandwidth        Jitter   Lost/Total Datagrams
[  3]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.002 ms    0/  893 (0%)

root@ls1021aqds:~# cat /proc/interrupts | grep eth
176:         13          0       GIC 176  eth0_g0_tx
177:       1798          0       GIC 177  eth0_g0_rx
178:          0          0       GIC 178  eth0_g0_er
```

```
179:          1          0       GIC 179  eth0_g1_tx
180:        894          0       GIC 180  eth0_g1_rx
181:          0          0       GIC 181  eth0_g1_er

[  4] local 172.16.1.100 port 5001 connected with 172.16.1.4 port 58876
[  4]  0.0-10.0 sec  1.25 MBytes  1.05 Mbits/sec   0.004 ms    0/  893 (0%)

root@ls1021aqds:~# cat /proc/interrupts | grep eth
176:         16          0       GIC 176  eth0_g0_tx
177:       1800          0       GIC 177  eth0_g0_rx
178:          0          0       GIC 178  eth0_g0_er
179:          1          0       GIC 179  eth0_g1_tx
180:        894        894       GIC 180  eth0_g1_rx
181:          0          0       GIC 181  eth0_g1_er

root@ls1021aqds:~# ethtool -n eth0
2 RX rings available
Total 2 rules

Filter: 253
        Rule Type: UDP over IPv4
        Src IP addr: 172.16.1.4 mask: 0.0.0.0
        Dest IP addr: 0.0.0.0 mask: 255.255.255.255
        TOS: 0x0 mask: 0xff
        Src port: 0 mask: 0xffff
        Dest port: 5001 mask: 0x0
        Action: Direct to queue 1

Filter: 254
        Rule Type: UDP over IPv4
        Src IP addr: 172.16.1.4 mask: 0.0.0.0
        Dest IP addr: 0.0.0.0 mask: 255.255.255.255
        TOS: 0x0 mask: 0xff
        Src port: 0 mask: 0xffff
        Dest port: 5000 mask: 0x0
        Action: Direct to queue 0

root@ls1021aqds:~# ethtool -N eth0 delete 254
root@ls1021aqds:~# ethtool -N eth0 delete 253
root@ls1021aqds:~# ethtool -n eth0
2 RX rings available
Total 0 rules

root@ls1021aqds:~#
```

8.5.1.2.3 NAPI support

Gianfar uses NAPI handling on both Rx and Tx paths, the Linux polling mechanism being triggered by frame receive interrupts and, respectively, frame transmit confirmation interrupts. The driver registers irqs for both Rx and Tx, and the NAPI (polling) handlers are provided to the Kernel.

On the receive path:

- When the receive interrupt gets triggered on a given CPU, a softirq for the polling function on Rx is scheduled.
- The RX_SOFTIRQ thread is raised by the Kernel, on the CPU on which it was triggered (and scheduled), and the Rx queues mapped to the corresponding interrupt group will be processed by the driver's polling function and the incoming packets are being passed to the networking stack.

Similarly on the transmit part:

- A frame transmit confirmation interrupt triggers the scheduling of a softirq under whose context the driver's polling routine for cleaning the Tx rings is invoked.
- The Tx polling routine is also associated with a given interrupt group and it will handle only the transmit queues that are affiliated to that interrupt group.

For packet forwarding, for instance, by mapping the per flow Rx and Tx queues to interrupt groups that are associated to the same CPU, makes it possible to maintain per CPU buffer pools used for reclaiming buffers on a per flow basis, improving cache locality at the same time.

8.5.1.2.4 Interrupt Coalescing

On a high speed network interface the rate of packet reception and transmission can be as high as the CPUs would be spending most of the time servicing these interrupts. With the interrupt coalescing feature, packets are collected and one single interrupt is generated for multiple packets to avoid flooding the system with interrupts from the Ethernet device.

eTSEC supports hardware coalescing of interrupts for both receive and transmit, using packet-count-based thresholds, complemented by timer-based thresholds. Gianfar provides basic support for setting the coalescing parameters via `ethtool -C`, for each device instance, by implementing the following "set coalesce" options:

Table 159. `ethtool -C` options:

rx-frames	packet count threshold for receive (Rx)
rx-usecs	time threshold in microseconds, for receive (Rx)
tx-frames	packet count threshold for transmit confirmation (Tx)
tx-usesc	time threshold in microseconds, for transmit confirmation (Tx)

8.5.1.2.5 Header Recognition and Csum Offload

Header recognition on receive (feature provided by eTSEC), combined with parsing functions and/or hashing of extracted property fields (in case of eTSEC2.0), is used to implement advanced TCP/IP offloading functionality and QoS provisions by programming queue filing strategies into hardware.

Gianfar provides an API to program eTSEC's filer hardware block with packet filtering rules.

On Rx, the TCP/IP Offload Engine (TOE):

- can parse frames:
  - at layer 2 of the stack only (Ethernet headers and switching headers)
  - layers 2 to 3 (including IPv4 or IPv6)
  - layers 2 to 4 (including TCP and UDP)
- provides protocol header recognition
- provides header verification (IPv4 header checksum verification)
- provides TCP/UDP payload checksum verification including verification of associated pseudo-header checksums

For large frames, offload of checksum verification saves a significant fraction of the CPU cycles that would otherwise be spent by the TCP/IP stack. IP packet fragmentation and re-assembly, and TCP stream establishment and tear-down are not performed in hardware.

On Tx side, TOE provides IPv4 and TCP/UDP header checksum generation. The eTSEC does not checksum transmitted packets with IPv6 routing headers or calculate TCP/UDP checksums from IP fragments. If a transmitted TCP segment requires checksum generation but IPv6 extension headers would prevent eTSEC from calculating the pseudoheader checksum, software can calculate just the pseudoheader checksum in advance and supply it to the eTSEC as part of per-frame TOE configuration.

On the Rx side, Gianfar lets the Kernel know that checksum verification is not required if valid IP headers or TCP/UDP headers were found and valid sums were verified, by setting the `CHECKSUM_UNNECESSARY` flag. On Tx side, the checksum is generated (offloaded) for TCP/UDP packets over IPv4 based on the pseudo-header checksum *(phcs)* provided by the Linux networking stack. Gianfar instructs the stack about its ability to provide partial checksumming, based on the *phcs* for TCP/UDP packets, by setting the `NETIF_F_IP_CSUM` device capability flag.

The Frame Control Blocks (FCBs) are 8-byte blocks of TOE control and/or status data that are passed between the driver and each eTSEC. A FCB always precedes the frame it applies to, and is present only when TOE functions are being used.

The first BD of each frame points to the initial data buffer and the FCB. Custom or received Ethernet preamble sequences also follow the FCB if preambles are visible.

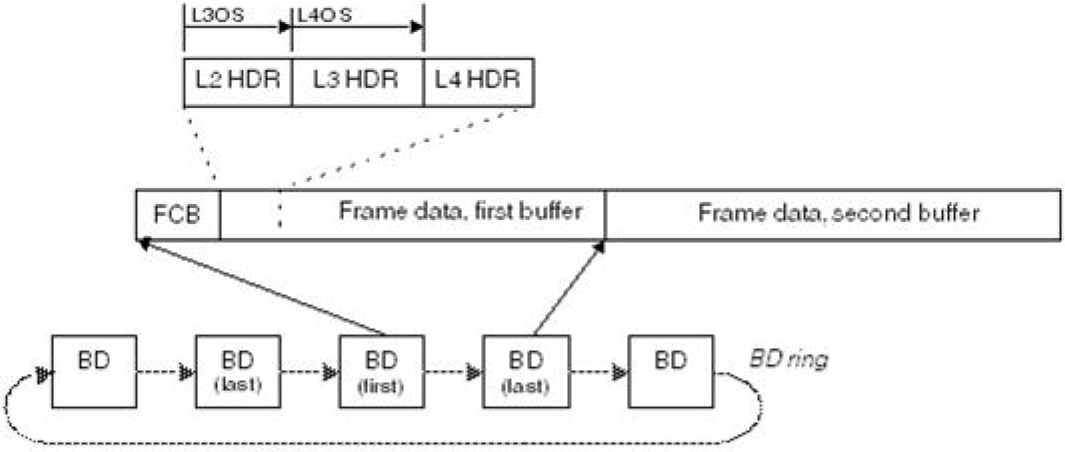


Figure 202. Location of Frame Control Blocks for TOE Parameters

For Tx, FCBs are inserted by Gianfar and TOE acceleration may be applied on a frame-by-frame basis. In the case of RxBD rings, the FCBs are inserted by eTSEC and TOE acceleration is enabled for receive for all frames in this case.

8.5.1.2.6 Scatter Gather Support

Scatter-Gather I/O is a method by which a single procedure call sequentially writes data from multiple buffers to a single data stream or reads data from a data stream to multiple buffers. The buffers are given in a vector of buffers. Scatter/gather refers to the process of gathering data from, or scattering data into, the given set of buffers. The I/O can be performed synchronously or asynchronously to this procedure.

On the Tx side, Gianfar supports "gathering" big packets from multiple buffers. This ability is signaled by the driver to the Linux network stack by setting the `NETIF_F_SG` device hardware feature flag. The driver takes into account the number of fragments composing the packet that is going to be transmitted, and places each fragment into consecutive BD ring buffers before issuing the command to start sending the frame.

On the Rx side, the eTSEC controller is capable of "scattering" big packets into multiple fixed size buffers having consecutive buffer descriptors (BDs). Gianfar supports this feature by implementing paged allocation, so that jumbo frames exceeding a fixed buffer size of 2048 bytes can be automatically received into multiple such buffers. Instead of pre-allocating huge memory buffers to be able to support jumbo frame reception, the paged allocation scheme implemented in Gianfar uses multiple half-page sized buffers thus reducing memory allocation pressure. The driver is also managing a local cache of memory pages, re-using free pages from the cache for future receptions, further improving Rx allocation overhead.

8.5.1.3 Configuration & Control

8.5.1.3.1 Device Tree initialization

Gianfar complies with the device tree (DTS) based open firmware support requirements, and supports multiple Ethernet device instances. The default configuration parameters that are passed via DTS for an Ethernet device instance (node) include:

1. `compatible` and `model` fields defining the driver compatibility across multiple controller H/W IP generations:

Table 160. Gianfar compatibility

Device type (IP):	**.compatible**	**.model**
eTSEC2.0 (veTSEC)	“fsl,etsec2”	"eTSEC”
eTSEC (eTSEC1.x)	“gianfar”	“eTSEC”
TSEC	“gianfar”	“TSEC”
FEC	“gianfar”	“FEC”

2. Interrupt grouping of multiple queues, for eTSEC2.0: `queue-group` subnode, including:
    - Interrupt numbers assignment for the Rx, Tx and Error lines, `interrupts` property;
    - Interrupt group register block address and size, `reg` property;
3. Power management capability properties:
    - `fsl,magic-packet`: If present, indicates that the hardware supports waking up via magic packet;
    - `fsl,wake-on-filer`: If present, indicates that the hardware supports waking up by Filer General Purpose Interrupt (FGPI) asserted on the Rx int line. This is an advanced power management capability allowing certain packet types (user) defined by filer rules to wake up the system.
4. For older DTs, number of supported TX and RX queues: `fsl,num-rx-queues` and `fsl,num-tx-queues`; *[obsolete]*
5. Various link management properties.

Typical eTSEC2.0 device tree node (LS1021a example):

```
enet0: ethernet@2d10000 {
    compatible = "fsl,etsec2";
    device_type = "network";
    #address-cells = <2>;
    #size-cells = <2>;
    interrupt-parent = <&gic>;
    model = "eTSEC";
    fsl,magic-packet;
    fsl,wake-on-filer;

    queue-group@2d10000 {
        #address-cells = <2>;
        #size-cells = <2>;
        reg = <0x0 0x2d10000 0x0 0x1000>;
        interrupts = <GIC_SPI 144 IRQ_TYPE_LEVEL_HIGH>,
            <GIC_SPI 145 IRQ_TYPE_LEVEL_HIGH>,
            <GIC_SPI 146 IRQ_TYPE_LEVEL_HIGH>;
    };

    queue-group@2d14000  {
        #address-cells = <2>;
        #size-cells = <2>;
        reg = <0x0 0x2d14000 0x0 0x1000>;
        interrupts = <GIC_SPI 147 IRQ_TYPE_LEVEL_HIGH>,
            <GIC_SPI 148 IRQ_TYPE_LEVEL_HIGH>,
            <GIC_SPI 149 IRQ_TYPE_LEVEL_HIGH>;
    };
};
```

8.5.1.3.2 Ethtool support

Table 161. Non-exhaustive list of the most notable *ethtool* commands implemented by Gianfar:

Commands	Description

Table continues on the next page...

Table 161. Non-exhaustive list of the most notable *ethtool* commands implemented by Gianfar: (continued)

`ethtool -C`	`rx-usecs N rx-frames N tx-usecs N tx-frames N`	Set interrupt coalescing for a given device, packet count ('frames') and time in microsecs ('usecs') thresholds, for Rx and resp. Tx.
`ethtool -G`	`rx N  tx N`	Set RxBD ring, resp. TxBD ring sizes for a given device.
`ethtool -K`	`rxvlan on|off txvlan on|off`	Turn on/off H/W VLAN tag extraction(rx) / insertion(tx).
`ethtool -S`		Show interface statistics, Linux specific counters and various eTSEC H/W counters supporting RMON MIB group 1, group 2 (ifTable counters), group 3, group 9, RMON MIB 2, and the 802.3 Ethernet MIB statistics.
`ethtool -N`	`rx-flow-hash tcp4|udp4|tcp6|udp6 v|t|s|d|f|n`	Configure Rx network flow classification options. The classified flows may be tcp/udp over ipv4/v6, and the hashing may be performed on various header fields, according to the 3rd parameter: • s,d: src/dest IP addresses; • v: VLAN id; • t: L3 PROTO field, • f,n: source and dest TCP/UDP ports.
`ethtool -N`	`flow-type ether|ip4|tcp4|udp4|sctp4`	Inserts or updates a classification rule for the specified flow type. Most IPv4 flow types are supported: raw IPv4, TCP, UDP, SCTP, as well as L2 flow specifications (ether). For a detailed description of the command sub-options refer to ethtool Linux man-pages.

NOTE

For detailed description of `ethtool` command options refer to ethtool Linux man-pages.

Chapter 9
Linux user space

9.1 Libraries

9.1.1 OpenSSL

9.1.1.1 Overview

The Secure Socket Layer (SSL) protocol is the most widely deployed application protocol to protect data during transmission by encrypting the data using popular cipher algorithms such as AES, DES and 3DES.

Apart from encryption it also provides message authentication services using popular hash/digest algorithms such as SHA1 and MD5. SSL is widely used in application web servers (HTTP) and other applications such as SMTP POP3, IMAP, Proxy servers etc., where protection of data in transit is essential.

There are various version of SSL protocol such as SSLv3, TLSv1.0, TLSv1.1, TLSv1.2, TLSv1.3 and DTLS (Datagram TLS). Of all the SSL protocol versions, TLSv1.0 and SSLv3 are in commonly in use, with other versions seeing more adoption as well.

This document introduces NXP SSL acceleration solution on QorIQ platforms using OpenSSL.

OpenSSL Software architecture

The OpenSSL library has several sub-components such as:

1. SSL protocol library
2. Crypto library (Symmetric and Asymmetric cipher support, digest support etc.)
3. Certificate Management

The following figure presents the general interconnect architecture for OpenSSL and the interfaces with hardware acceleration drivers:

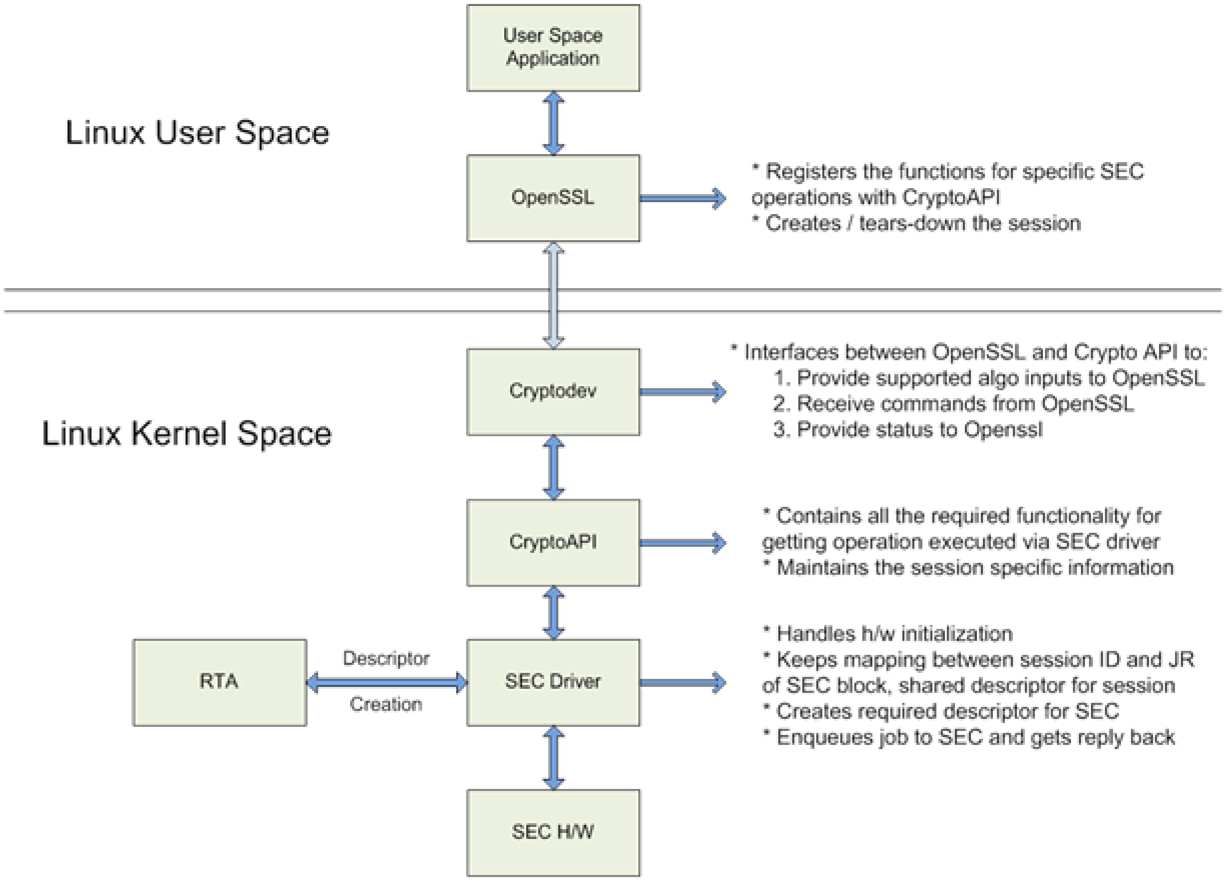

User Space Application
Linux User Space
OpenSSL
* Registers the functions for specific SEC operations with CryptoAPI
* Creates / tears-down the session
Cryptodev
* Interfaces between OpenSSL and Crypto API to:
1. Provide supported algo inputs to OpenSSL
2. Receive commands from OpenSSL
3. Provide status to Openssl
Linux Kernel Space
CryptoAPI
* Contains all the required functionality for getting operation executed via SEC driver
* Maintains the session specific information
RTA
Descriptor
Creation
SEC Driver
* Handles h/w initialization
* Keeps mapping between session ID and JR of SEC block, shared descriptor for session
* Creates required descriptor for SEC
* Enqueues job to SEC and gets reply back
SEC H/W

Figure 203. OpenSSL interface with Linux kernel

OpenSSL’s ENGINE Interface

OpenSSL Crypto library provides Symmetric and Asymmetric (PKI) cipher support that is used in a variety of applications such as OpenSSH, OpenVPN, PGP, IKE, XML-SEC etc. The OpenSSL Crypto library provides software support for:

1. Cipher algorithms
2. Digest algorithms
3. Random number generation
4. Public Key Infrastructure

Apart from the software support, OpenSSL can offload these functions to hardware accelerators via the ENGINE interface. The ENGINE interface provides callback hooks that integrate hardware accelerators with the crypto library. The callback hooks provide the glue logic to interface with the hardware accelerators. Generic offloading of cipher and digests algorithms through Linux kernel is possible with cryptodev engine.

NXP solution for OpenSSL hardware offloading

The following layers can be observed in NXP's solution for OpenSSL hardware offloading:

- OpenSSL (user space) - implements the SSL protocol
- cryptodev-engine (user space) - implements the OpenSSL ENGINE interface; talks to cryptodev-linux (/dev/crypto) via ioctls, offloading cryptographic operations in kernel
- cryptodev-linux (kernel space) - Linux module that translates ioctl requests from cryptodev-engine into calls to Linux Crypto API

- Linux Crypto API (kernel space) - Linux kernel crypto abstraction layer
- CAAM driver (kernel space) - Linux device driver for the CAAM (Cryptographic Acceleration and Assurance Module) crypto engine

The following can be offloaded in hardware in current SDK (not a complete list):

- SSL: TLS v1.0 with one-shot cipher modes (a single ioctl for both encryption and authentication):
    - AES128-SHA
    - AES256-SHA
- Crypto algorithms:
    - AES-CBC
    - 3DES
- Digest algorithms:
    - MD5
    - SHA1
    - SHA256
- Public key algorithms:
    - RSA

9.1.1.2 Manual Build of OpenSSL with Cryptodev Engine Support

This chapter is optional since the root filesystem can be configured to include both OpenSSL and cryptodev automatically.

```
$ cd flexbuild
$ source setup.env

Build cryptodev-linux:
$ flex-builder -c cryptodev-linux -a arm64   # automatically setup cross-toolchain and fetch
cryptodev-linux repository to build

Build OpenSSL:
$ flex-builder -c openssl -a arm64

Merge OpenSSL and cryptodev-linux components into target rootfs:
$ flex-builder -i merge-component -a arm64

Generate bootpartition tarball:
$ flex-builder -i mkbootpartition -a arm64
```

Follow flexbuild documentation to finalize the building of the root filesystem and kernel on the host system.

Manual build

Manual build or rebuild may be necessary for a number of reasons. This section describes how to build and install OpenSSL natively on the target system. Cross-build procedure requires appropriate toolchains and is not described.

Both OpenSSL and cryptodev must be installed since they depend on each-other:

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

Build cryptodev and optionally run self tests:

```
$ cd cryptodev-linux
$ make
$ sudo make install
```

```
$ sudo modprobe cryptodev
$ make check
```

Build openssl with cryptodev support:

```
$ cd openssl
$ ./Configure -DHAVE_CRYPTODEV --prefix=/usr/local/ --openssldir=/usr/local/openssl linux-
aarch64 shared
$ make
$ sudo make install
```

After installation verify that the binary is linking with the correct share library from /usr/local/lib:

```
$ ldd /usr/local/bin/openssl
```

If the binary is linking with the original library from /usr/lib then it may be necessary to adjust the linker paths. Put /usr/local/lib in a line before /usr/lib inside /etc/ld.so.conf and then update the linker cache:

File: /etc/ld.so.conf

```
...
/usr/local/lib
...
/usr/lib
...
```

```
$ sudo ldconfig
$ ldd /usr/local/bin/openssl
```

9.1.1.3 Hardware Offloading with OpenSSL

Overview

OpenSSL can delegate execution of crypto operations to a variety of hardware devices through the engine interface. On top of this interface is implemented the engine cryptodev which is used to offload crypto operations to hardware devices under the control of the operating system kernel. Cryptodev engine was originally developed for OpenBSD and later the same API was ported to GNU/Linux operating system by several drivers like OCF and cryptodev-linux.

Cryptodev-linux is a Linux kernel driver that exposes the internal crypto API to user-space via the device file /dev/crypto. User-space applications use ioctl system calls to ask the Linux kernel to perform crypto operations on their behalf. The Linux kernel supports a multitude of crypto algorithms with software implementations running on CPU. Drivers for hardware accelerators are installed with higher priority and override software implementations with no further configuration.

From the point of view of any application, the fastest implementation of an algorithm is used transparently. This behavior is transferred also to cryptodev interface which is oblivious to the fact that an algorithm may run on CPU or on a hardware accelerator. For this reason, it is the job of the operator of the application to ensure that hardware kernel drivers are available before running the application.

This translates simply to running modprobe if the NXP SEC driver is not built-in the kernel. In our case:

```
# modprobe caamalg
# modprobe caamhash
# modprobe caam_pkc
```

NXP platforms have several SEC frontends exposed with different device drivers: JRI (job ring), QI (queue interface), DPSECI. Refer to your platform and SEC guide for available Linux kernel drivers. Usually, at least the JR driver should be available for any platform

For QI frontend the symmetric cyphers has a kernel module called caamalg_qi. This driver installs algorithms with lower priority than caamalg so they will be shadowed by the latter. To use the QI frontend load this driver instead of caamalg:

```
# modprobe caamalg_qi
```

DPSECI frontend has yet another driver caamalg_qi2 which currently is always built-in the kernel.

Verify setup

There are a few simple steps to confirm if crypto hardware drivers are available. Running these steps contribute to a smooth experience and easier debugging if things go wrong.

Linux kernel can check and report ciphers availability with the help of tcrypt module. After probing tcrypt, crypto algorithms will be listed in /proc/crypto. Tcrypt module is not always available in default kernels but it is a simple way to run tests and list all available crypto algorithms:

```
$ modprobe tcrypt
$ grep aes /proc/crypto
<...>
```

Load CAAM device drivers if they are not built-in and check their interrupt count:

```
# modprobe caamalg (or caamalg_qi)
# modprobe caamhash
# modprobe caam_pkc
<...>
# grep tls /proc/crypto
name         : tls10(hmac(sha1),cbc(aes))
driver       : tls10-hmac-sha1-cbc-aes-caam-qi
# grep rsa /proc/crypto
<...>
```

Hardware operations can be monitored with the interrupt counters for CAAM JR and QI (DPAA and DPAA2) interfaces:

```
# cat /proc/interrupts | grep jr
 88: 0 0 0 0 26 0 0 0   OpenPIC    88 Level     ffe301000.jr
 89: 0 0 0 0 0 1117204 0 0   OpenPIC    89 Level     ffe302000.jr
 90: 0 0 0 0 0 0 24 0   OpenPIC    90 Level     ffe303000.jr
 91: 0 0 0 0 0 0 0 24   OpenPIC    91 Level     ffe304000.jr
```

```
# cat /proc/interrupts | grep -i qman
108: 0 0 0 0 0 0 0 7508   OpenPIC   108 Level     QMan portal 7
110: 0 0 0 0 0 0 7524 0   OpenPIC   110 Level     QMan portal 6
112: 0 0 0 0 0 7542 0 0   OpenPIC   112 Level     QMan portal 5
114: 0 0 0 0 7565 0 0 0   OpenPIC   114 Level     QMan portal 4
116: 0 0 0 7576 0 0 0 0   OpenPIC   116 Level     QMan portal 3
118: 0 0 7524 0 0 0 0 0   OpenPIC   118 Level     QMan portal 2
120: 0 7535 0 0 0 0 0 0   OpenPIC   120 Level     QMan portal 1
122: 7521 0 0 0 0 0 0 0   OpenPIC   122 Level     QMan portal 0
470: 0 0 0 0 0 0 0 0      OpenPIC   2006 Edge      qman-err
```

```
# cat /proc/interrupts | grep DPIO
<...>
```

The interrupt counters may also be incremented during networking operations unrelated to crypto. Further analysis is required to understand the source of their modification.

Load cryptodev driver and check if OpenSSL communicates with it. If cryptodev driver is not loaded, openssl will report only dynamic engine support and all operations will be done in software by OpenSSL itself.

```
# openssl engine
(dynamic) Dynamic engine loading support

# modprobe cryptodev
# ls /dev/crypto
<...>
# openssl engine
(cryptodev) BSD cryptodev engine
(dynamic) Dynamic engine loading support
```

Offloading Symmetric and Public Key Algorithms

With cryptodev and SEC drivers loaded there is no other configuration necessary for OpenSSL to run crypto operations through hardware accelerator. OpenSSL will automatically use cryptodev engine if available. Some applications that link with OpenSSL like OpenSSH will authomatically use the available accelerator. Others, like nginx web server may need explicit activation in their configuration file.

```
# modprobe cryptodev
# openssl speed -evp AES128-SHA -elapsed
<...>
# openssl speed rsa1024
<...>
```

TLS 1.0 Offloading in Nginx Server

Nginx does not use any openssl engines by default. If an engine is to be used, including cryptodev, it must be explicitely listed in nginx configuration file. Here is a fragment of nginx configuration file that activates cryptodev and allows hardware offloading of TLS1.0 record layer protocol:

```
/etc/nginx/nginx.conf:
```

```
ssl_engine cryptodev;
worker_processes 4;
worker_cpu_affinity 0001 0010 0100 1000; #for 4 Core CPU; For 2 Core CPU worker_cpu_affinity
01 10;
...
    # HTTPS server
    #
    server {
        listen       443;
        server_name  localhost;

        ssl                  on;
        ssl_certificate      server.crt;
        ssl_certificate_key  server.key;
        ssl_session_timeout  5m;
        ssl_protocols  TLSv1;
        ssl_ciphers AES128-SHA:AES256-SHA;
        ssl_prefer_server_ciphers   on;

        location / {
            root   /var/www/localhost/html;
            index  index.html index.htm;
        }
    }
...
```

Worker processes and affinity should be set according to the number of CPU cores available on the platform. Refer to nginx documentation for more details.

TLS1.0 Record Layer Testing

We will use only OpenSSL functionality to verify the offloading of TLS record layer.

First create the RSA public and private keys to be used by the server:

```
$ openssl req -x509 -newkey rsa:2048 -keyout key.pem -out cert.pem -days 365 -nodes
```

Start https webserver:

```
# modprobe cryptodev
$ openssl s_server -key key.pem -cert cert.pem -accept 44330 -www -cipher AES128-SHA -tls1
```

And connect to it with the client from another console:

```
$ openssl s_client -connect localhost:44330
(or)
$ echo "GET /" | openssl s_client -connect localhost:44330 -quiet
```

Just like with other tests, hardware offloading can be verified by listing the interrupt counters of the SEC driver.

9.1.1.4 TLS Ciphersuites and TLS Protocol Versions

Please refer to the official RFC documents for an up-to-date list of supported algorithms:

Transport Layer Security (TLS) Parameters

Table 162. OpenSSL CipherSuite Compatibility

CipherSuite	TLS Protocol Version
SSL_RSA_WITH_NULL_MD5	SSL3.0
SSL_RSA_WITH_NULL_SHA	SSL3.0
SSL_RSA_EXPORT_WITH_RC4_40_MD5	SSL3.0
SSL_RSA_WITH_RC4_128_MD5	SSL3.0
SSL_RSA_WITH_RC4_128_SHA	SSL3.0
SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5	SSL3.0
SSL_RSA_WITH_IDEA_CBC_SHA	SSL3.0
SSL_RSA_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_RSA_WITH_DES_CBC_SHA	SSL3.0
SSL_RSA_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_DH_DSS_WITH_DES_CBC_SHA	SSL3.0
SSL_DH_DSS_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_DH_RSA_WITH_DES_CBC_SHA	SSL3.0
SSL_DH_RSA_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_DHE_DSS_WITH_DES_CBC_SHA	SSL3.0
SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_DHE_RSA_WITH_DES_CBC_SHA	SSL3.0
SSL_DHE_RSA_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_DH_anon_EXPORT_WITH_RC4_40_MD5	SSL3.0
SSL_DH_anon_WITH_RC4_128_MD5	SSL3.0
SSL_DH_anon_EXPORT_WITH_DES40_CBC_SHA	SSL3.0
SSL_DH_anon_WITH_DES_CBC_SHA	SSL3.0
SSL_DH_anon_WITH_3DES_EDE_CBC_SHA	SSL3.0
SSL_FORTEZZA_KEA_WITH_NULL_SHA	SSL3.0
SSL_FORTEZZA_KEA_WITH_FORTEZZA_CBC_SHA	SSL3.0
SSL_FORTEZZA_KEA_WITH_RC4_128_SHA	SSL3.0
TLS_RSA_WITH_NULL_MD5	TLS1.0
TLS_RSA_WITH_NULL_SHA	TLS1.0

Table continues on the next page...

Table 162. OpenSSL CipherSuite Compatibility (continued)

CipherSuite	TLS Protocol Version
TLS_RSA_EXPORT_WITH_RC4_40_MD5	TLS1.0
TLS_RSA_WITH_RC4_128_MD5	TLS1.0
TLS_RSA_WITH_RC4_128_SHA	TLS1.0
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5	TLS1.0
TLS_RSA_WITH_IDEA_CBC_SHA	TLS1.0
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_RSA_WITH_DES_CBC_SHA	TLS1.0
TLS_RSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_DES_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_DES_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_DES_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_DES_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_DH_anon_EXPORT_WITH_RC4_40_MD5	TLS1.0
TLS_DH_anon_WITH_RC4_128_MD5	TLS1.0
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_DES_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_RSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_RSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_AES_128_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_AES_256_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_AES_128_CBC_SHA	TLS1.0

Table continues on the next page...

Table 162. OpenSSL CipherSuite Compatibility (continued)

CipherSuite	TLS Protocol Version
TLS_DHE_DSS_WITH_AES_256_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_AES_128_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_AES_256_CBC_SHA	TLS1.0
TLS_RSA_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_RSA_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_CAMELLIA_128_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_CAMELLIA_256_CBC_SHA	TLS1.0
TLS_RSA_WITH_SEED_CBC_SHA	TLS1.0
TLS_DH_DSS_WITH_SEED_CBC_SHA	TLS1.0
TLS_DH_RSA_WITH_SEED_CBC_SHA	TLS1.0
TLS_DHE_DSS_WITH_SEED_CBC_SHA	TLS1.0
TLS_DHE_RSA_WITH_SEED_CBC_SHA	TLS1.0
TLS_DH_anon_WITH_SEED_CBC_SHA	TLS1.0
TLS_ECDH_RSA_WITH_NULL_SHA	TLS1.0
TLS_ECDH_RSA_WITH_RC4_128_SHA	TLS1.0
TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_ECDH_ECDSA_WITH_NULL_SHA	TLS1.0
TLS_ECDH_ECDSA_WITH_RC4_128_SHA	TLS1.0
TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA	TLS1.0

Table continues on the next page...

Table 162. OpenSSL CipherSuite Compatibility (continued)

CipherSuite	TLS Protocol Version
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_ECDHE_RSA_WITH_NULL_SHA	TLS1.0
TLS_ECDHE_RSA_WITH_RC4_128_SHA	TLS1.0
TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_ECDHE_ECDSA_WITH_NULL_SHA	TLS1.0
TLS_ECDHE_ECDSA_WITH_RC4_128_SHA	TLS1.0
TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA	TLS1.0
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA	TLS1.0
TLS_ECDH_anon_WITH_NULL_SHA	TLS1.0
TLS_ECDH_anon_WITH_RC4_128_SHA	TLS1.0
TLS_ECDH_anon_WITH_3DES_EDE_CBC_SHA	TLS1.0
TLS_ECDH_anon_WITH_AES_128_CBC_SHA	TLS1.0
TLS_ECDH_anon_WITH_AES_256_CBC_SHA	TLS1.0
TLS_RSA_WITH_NULL_SHA256	TLS1.2
TLS_RSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_RSA_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_RSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_RSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_DH_RSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_DH_RSA_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_DH_RSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_DH_RSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_DH_DSS_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_DH_DSS_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_DH_DSS_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_DH_DSS_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	TLS1.2

Table continues on the next page...

Table 162. OpenSSL CipherSuite Compatibility (continued)

CipherSuite	TLS Protocol Version
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_DHE_DSS_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_DHE_DSS_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_DHE_DSS_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_DHE_DSS_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_ECDH_RSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_ECDH_RSA_WITH_AES_256_CBC_SHA384	TLS1.2
TLS_ECDH_RSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_ECDH_RSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA384	TLS1.2
TLS_ECDH_ECDSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_ECDH_ECDSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	TLS1.2
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA384	TLS1.2
TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_ECDHE_ECDSA_WITH_AES_256_GCM_SHA384	TLS1.2
TLS_DH_anon_WITH_AES_128_CBC_SHA256	TLS1.2
TLS_DH_anon_WITH_AES_256_CBC_SHA256	TLS1.2
TLS_DH_anon_WITH_AES_128_GCM_SHA256	TLS1.2
TLS_DH_anon_WITH_AES_256_GCM_SHA384	TLS1.2

9.1.2 Runtime Assembler Library Reference

Use the Runtime Assembler Library to write SEC descriptors.

9.1.2.1 Runtime Assembler Library Reference

Use the Runtime Assembler Library to write SEC descriptors. This reference describes the structure, concept, functionality, and high level API.

Click here to access the Writing descriptors for NXP CAAM using RTA library PDF.

9.2 Data Plane Development Kit (DPDK)

9.2.1 Introduction

DPDK is an user space packet processing framework.

This guide contains instructions for installing and configuring the user space Data Plane Development Kit (DPDK) v17.11 software. Besides highlighting the applicable platforms, this guide describes steps for compiling and executing sample DPDK applications in a Linux application (*linuxapp*) environment over Layerscape boards.

OVS-DPDK is a popular software switching package which uses DPDK as the underlying platform. The guide also detail methods to execute *ovs-dpdk* in conjuction with DPDK over Layerscape boards.

9.2.1.1 Supported Platforms and Platform-specific Details

DPDK supports LS1012A, LS1043A, LS1046A, LS1088A and LS2088A family of SoCs. This section details the architectural and port layout of their Reference Design Boards. Port layout information is especially relevant while executing DPDK applications - to map DPDK port number to physical ports..

Refer to the following for board specific information:

9.2.1.1.1 LS1012A Reference Design Board (RDB)

LS1012A is a PPFE-based platform. For more information on LS1012ARDB, see www.nxp.com/LS1012ARDB

Hardware Specification of LS1012ARDB

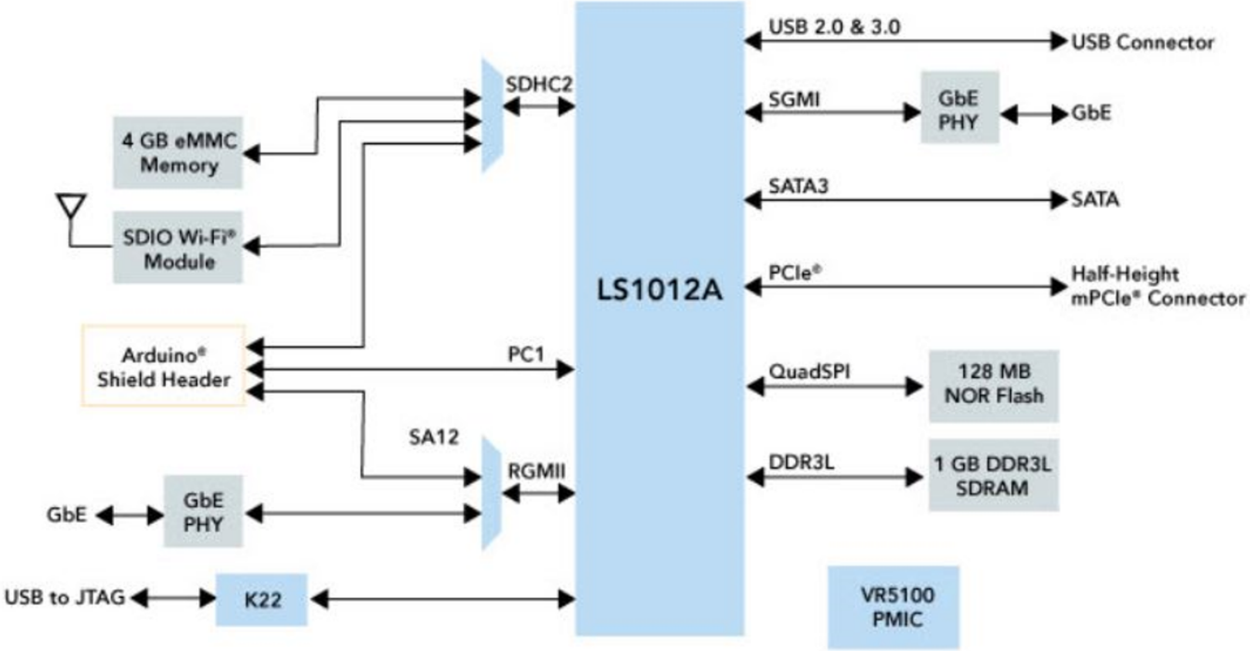


Figure 204. QorIQ LS1012A Reference Design

LS1012ARDB Port Layout

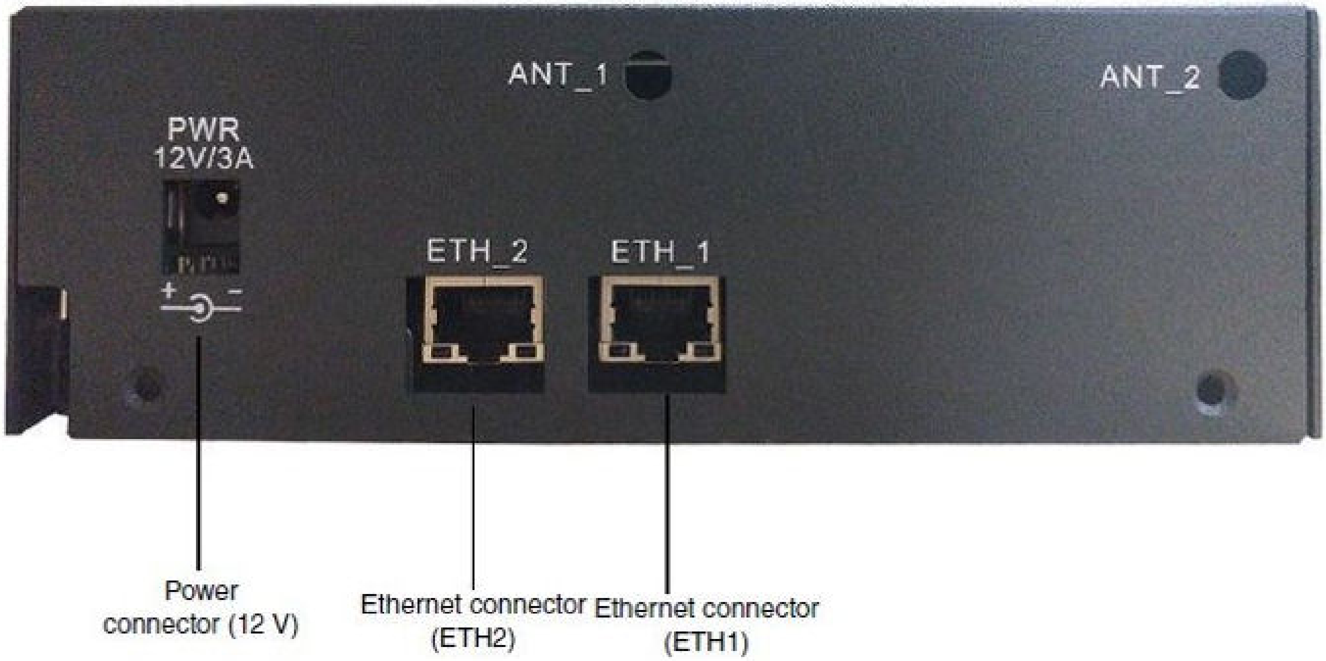


Figure 205. LS1012A Port Layout

Label on Case	DPDK vdev Port Names
ETH0	eth_pfe0
ETH1	eth_pfe1

9.2.1.1.2 LS1043A Reference Design Board (RDB)

LS1043A is a DPAA-based platform. For more information on LS1043ARDB, see www.nxp.com/LS1043ARDB

Hardware Specification of LS1043ARDB

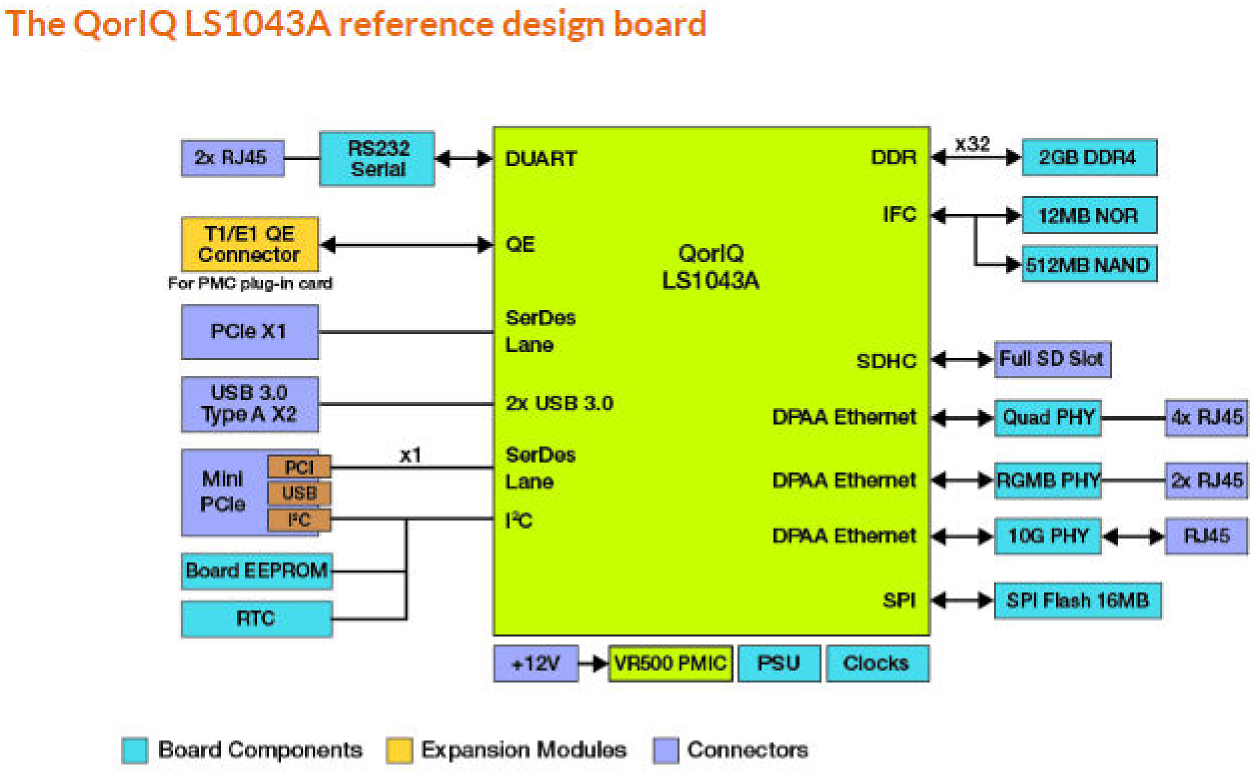


Figure 206. QorIQ LS1043A Reference Design

LS1043ARDB Port Layout

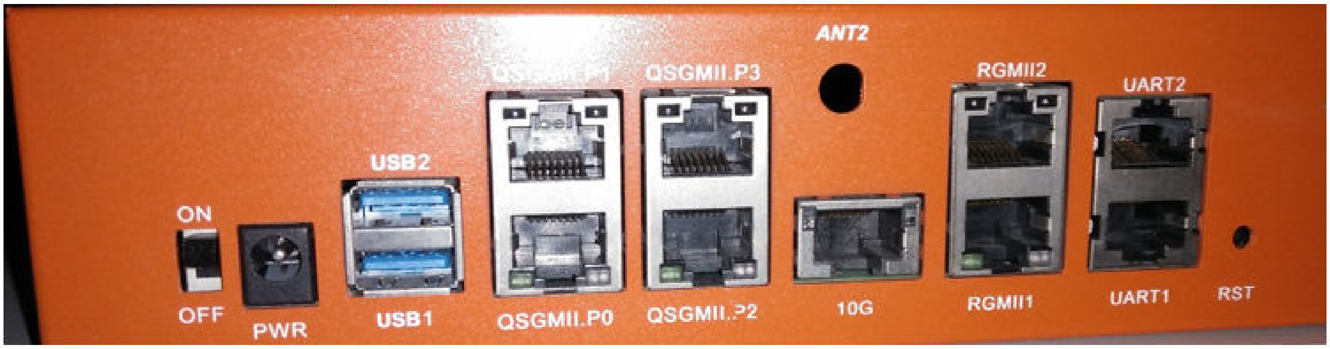


Figure 207. LS1043A Port Layout

Label on Case	FMAN Port Names	Userspace Ports	Comment
QSGMII.P0	FM0-MAC1	0	1G Port
QSGMII.P1	FM0-MAC2	1	1G Port
QSGMII.P2	FM0-MAC4	4	1G Port

Table continues on the next page...

Table continued from the previous page...

QSGMII.P3	FM0-MAC5	5	1G Port
RGMII1	FM0-MAC2	2	1G Port
RGMII2	FM0-MAC3	3	1G Port
10G	FM0-MAC9	6	10G - Copper Port

NOTE

Information provided in the "Userspace Ports" column above is conditional to default Device tree (DTB) provided as part of Board Support Package. The ordering can change for a custom DTB.

9.2.1.1.3 LS1046A Reference Design Board (RDB)

LS1046A is a DPAA based platform. For more information on QorIQ LS1046A, see www.nxp.com/LS1046ARDB.

Hardware Specification of LS1046ARDB

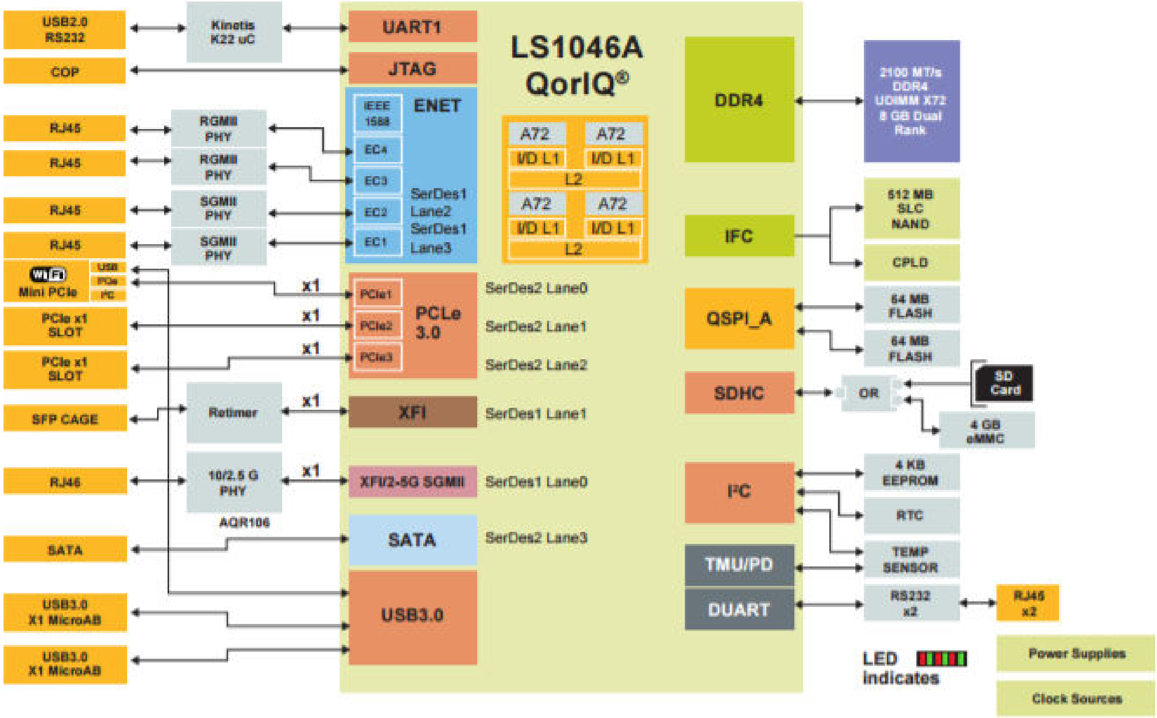


Figure 208. QorIQ LS1046A Reference Design

LS1046ARDB Port Layout

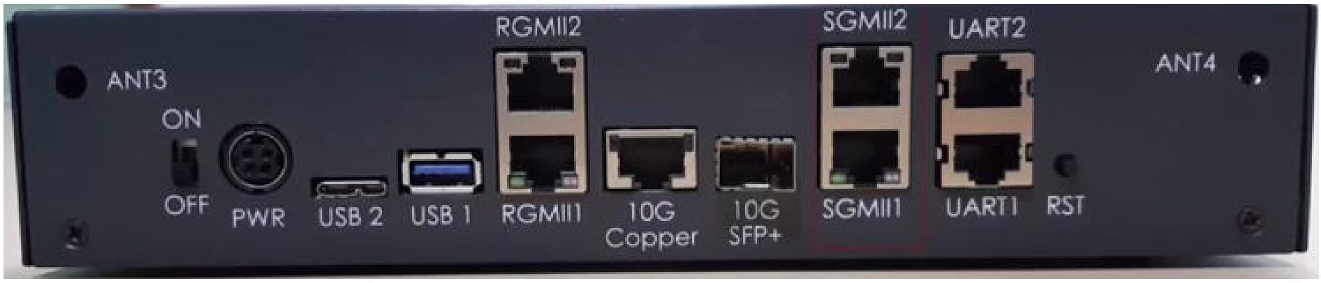


Figure 209. LS1046ARDB Port Layout

Label on Case	FMAN Port Names	Userspace Ports	Comment
RGMII1	FM0-MAC3	0	1G Port
RGMII2	FM0-MAC4	1	1G Port
SGMII1	FM0-MAC5	2	1G Port
SGMII2	FM0-MAC6	3	1G Port
10G-Copper	FM0-MAC9	4	10G – Copper Port
10G-SFP+	FM0-MAC10	5	10G – SFP+ Optical Port

NOTE

Information provided in the "Userspace Ports" column above is conditional to default Device tree (DTB) provided as part of Board Support Package. The ordering can change for a custom DTB.

9.2.1.1.4 LS1088A Reference Design Board (RDB)

LS1088A is a DPAA2 based platform. For more information on QorIQ LS1088A, see www.nxp.com/LS1088ARDB.

Hardware Specifications of LS1088ARDB

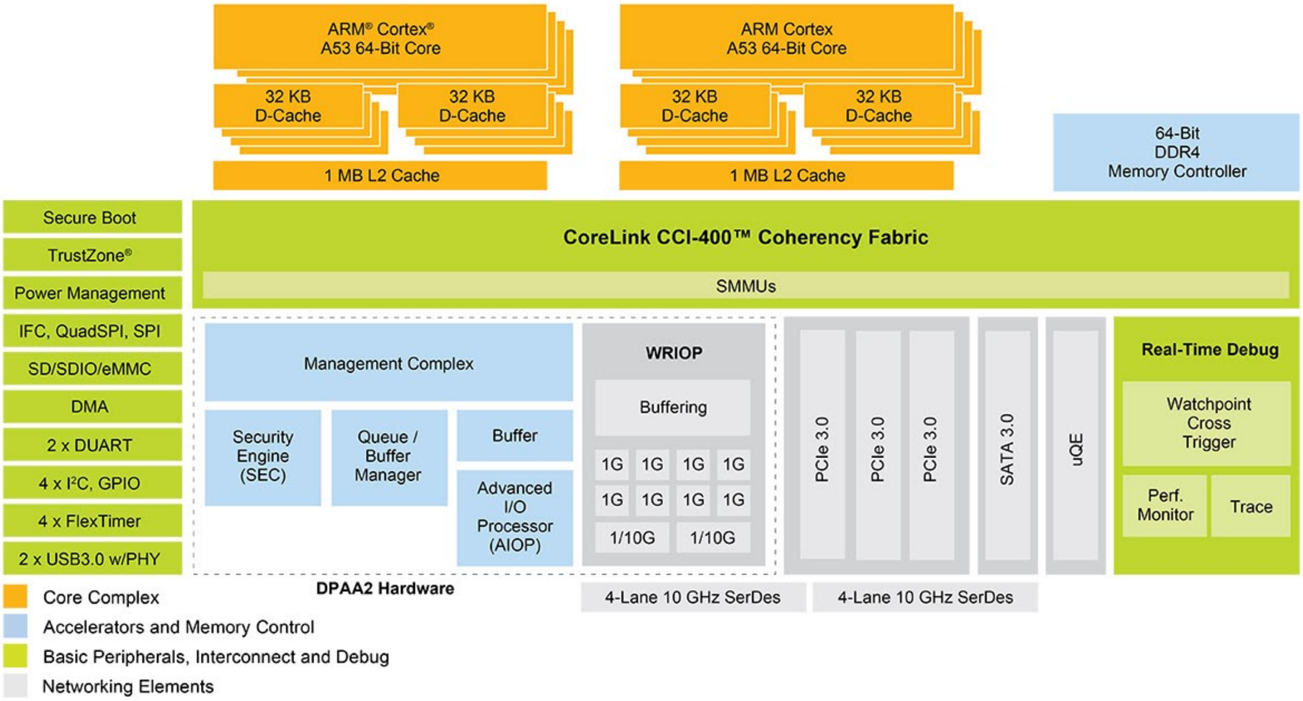


Figure 210. QorIQ LS1088A Architecture

LS1088ARDB Port Layout

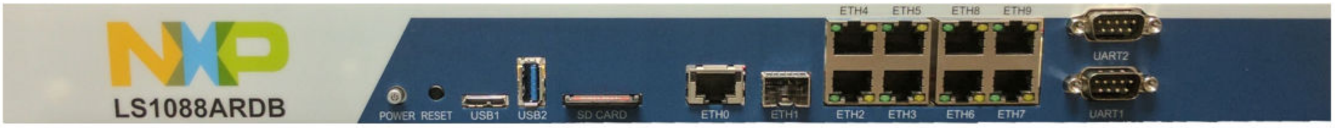


Figure 211. LS1088ARDB Port Layout

Label on Case	Physical Ports	Comment
ETH0	DPMAC.1	10G - Copper port
ETH1	DPMAC.2	10G – SFP+ (Optical port)
ETH2	DPMAC.7	QSGMII port (1G)
ETH3	DPMAC.8	QSGMII port (1G)
ETH4	DPMAC.9	QSGMII port (1G)
ETH5	DPMAC.10	QSGMII port (1G)
ETH6	DPMAC.3	QSGMII port (1G)
ETH7	DPMAC.4	QSGMII port (1G)
ETH8	DPMAC.5	QSGMII port (1G)
ETH9	DPMAC.6	QSGMII port (1G)

9.2.1.1.5 LS2088A Reference Design Board (RDB)

LS2088A is a DPAA2 based platform. For more information on QorIQ LS2088A, see www.nxp.com/LS2088ARDB.

Hardware specifications

LS2088A Reference Design Board

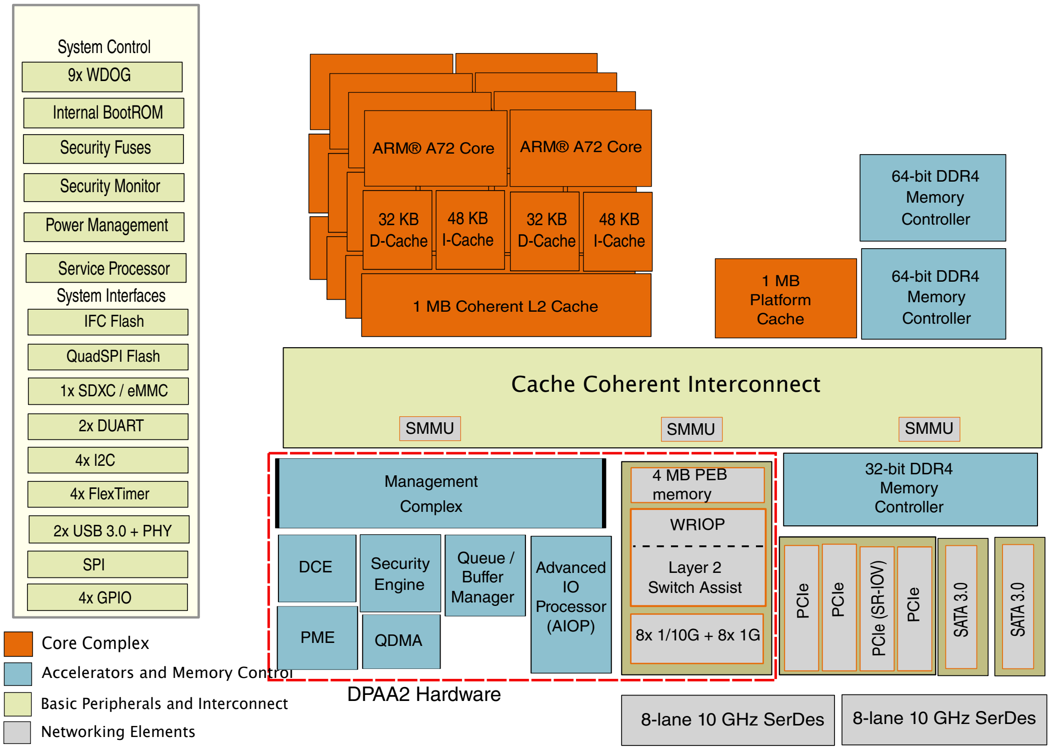


Figure 212. QorIQ LS2088A Architecture

LS1088ARDB Port Layout

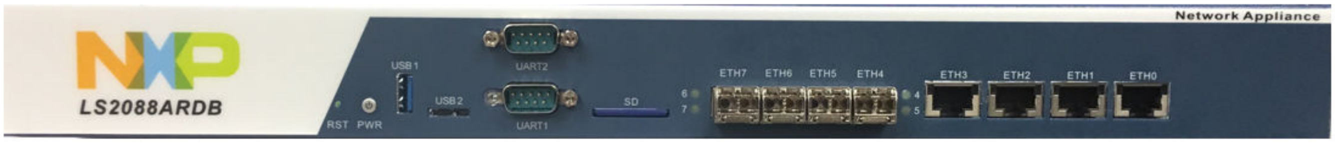


Figure 213. LS2088ARDB Port Layout

Label on Case	Physical Ports	Comment
ETH0	DPMAC.5	10G - Copper port
ETH1	DPMAC.6	10G - Copper port
ETH2	DPMAC.7	10G - Copper port
ETH3	DPMAC.8	10G - Copper port
ETH4	DPMAC.1	10G – SFP+ (Optical port)

Table continues on the next page...

Table continued from the previous page...

ETH5	DPMAC.2	10G – SFP+ (Optical port)
ETH6	DPMAC.3	10G – SFP+ (Optical port)
ETH7	DPMAC.4	10G – SFP+ (Optical port)

9.2.1.2 References

Table 163. DPDK Application References

Sample Applications	DPDK Web Manual Link	Description
Layer-2 Forwarding (`l2fwd`)	l2fwd usage	Layer 2 Forwarding sample application setup and usage guide.
Layer-2 Forwarding with Crypto (`l2fwd-crypto`)	l2fwd-crypto	Layer 2 Forwarding with Crypto sample application setup and usage guide.
Layer-3 Forwarding (`l3fwd`)	l3fwd usage	Layer 3 Forwarding sample application setup and usage guide.
IPSec Gateway (`ipsec-secgw`)	ipsec-secgw usage	IPSec Security Gateway sample application setup and usage guide.
PMD Test Application (`testpmd`)	testpmd usage	Guide for test application which can be used to test all PMD supported features.
DPDK Web Guide	DPDK Documentation	Link to DPDK Web Manual containing information about all supported PMD and Applications.

Table 164. Release References

Component	Release Version
DPDK	17.11
OVS	2.8.90

9.2.2 DPDK Overview

Key goal of the DPDK is to provide a simple, complete framework for fast packet processing in data plane applications. Using the APIs provided as part of the framework, applications can leverage the capabilities of underlying network infrastructure.

The framework creates a set of libraries for target environments, layered through an Environment Abstraction Layer (EAL) which hides all the device glue logic beneath a set of consistent APIs. These environments are created through the use of configuration files. Once the EAL library is created, the user may link with the library to create their own applications. Various other libraries, outside of EAL, including the Hash, Longest Prefix Match (LPM) and rings libraries are also available for performing specific operations. Sample applications are also provided to help understand various features and uses of DPDK framework.

DPDK implements a run-to-completion model for packet processing where all resources must be allocated prior to calling data plane applications, running as execution units on logical processing cores. In addition, a pipeline model may also be used by passing packets or messages between cores via rings. This allows work to be performed in stages, resulting in more efficient use of code on cores.

More information on general working of DPDK can be found through DPDK website.

9.2.2.1 DPDK Platform Support

This section describes the NXP Data Path Acceleration Architecture, see the diagram below:

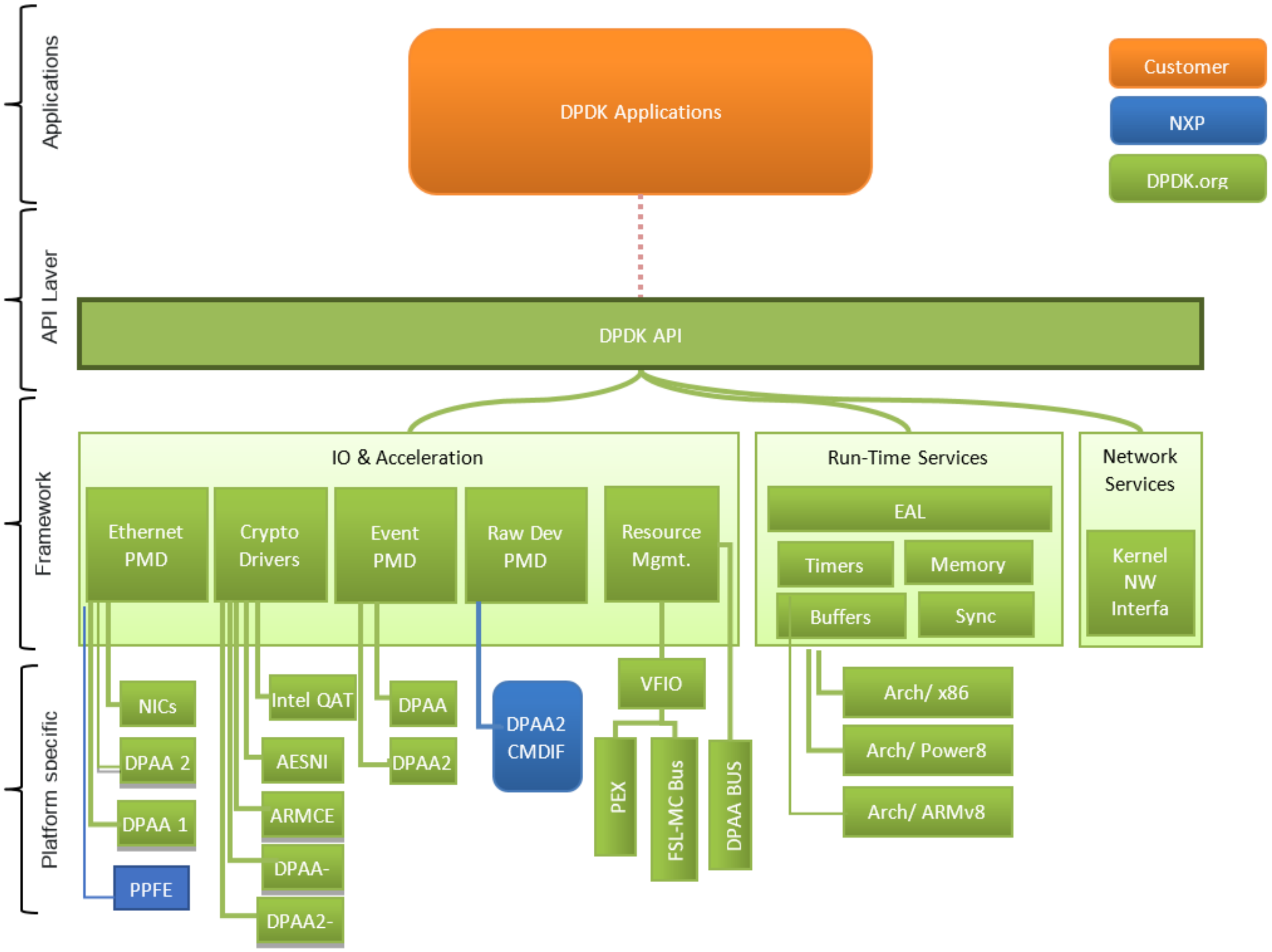


Figure 214. DPDK Architecture with NXP Components

The NXP Data Path Acceleration Architecture comprises a set of hardware components which are integrated via a hardware queue manager and use a common hardware buffer manager. Software accesses the DPAA via hardware components called "*Software Portals*". These directly provide queue and buffer manager operations such as enqueues, dequeues, buffer allocations, and buffer releases and indirectly provide access to all of the other DPAA hardware components via the queue manager.

NXP DPAA architecture based *PMD (Poll Mode Drivers)* has been added to DPDK infrastructure to support seamless working on NXP platform. With the addition of these drivers, DPDK framework on NXP platforms permits Linux user space applications to be build using standard DPDK APIs in a portable fashion. The drivers directly access the DPAA queue and buffer manager software portals in a high performance manner and the internal details remains hidden from higher level DPDK framework. Besides drivers for network interfaces, drivers (PMDs) for interfacing with Crypto (CAAM) block have also been included in the DPDK source code.

NOTE

Since this guide contains support for PPFE, DPAA2 and DPAA platforms, the following markers are used throughout the guide:

- DPAA2 – This marker marks the steps/text applicable only for DPAA2 platforms, for example, LS2088
- DPAA – This marker marks the steps/text applicable only for DPAA platforms, for example, LS1043
- PPFE - This marker marks the steps/text applicable only for PPFE platforms, for example, LS1012

All other steps which don't have any marker are applicable for both the platforms.

NOTE

See DPDK Performance Reproducibility Guide on page 926 to tune the system for best DPDK performance on NXP platforms.

NOTE

Multi-thread environment

DPDK was originally designed for Intel architectures, however efforts are underway to make it multiple architecture friendly. There are still some restrictions which should be taken care when used on NXP platforms.

1. Multiple pthreads

   DPDK usually pins one pthread per core to avoid the overhead of task switching. This allows for significant performance gains, but lacks flexibility and is not always efficient. DPDK is comprised of several libraries - some of the functions in these libraries can be safely called from multiple threads simultaneously, while others cannot.

   The run-time environment of the DPDK is typically a single thread per logical core. It is best to avoid sharing data structures between threads and/or processes where possible. Where this is not possible, the execution blocks must access the data in a thread-safe manner. Mechanisms such as atomic variables or locking can be used to allow execution blocks to operate serially. However, this can effect the performance of the application.

2. Fast-path APIs

   Applications operating in the data plane are performance sensitive but certain functions within those libraries may not be safe to call from multiple threads simultaneously.

   The Hash, LPM, Mempool libraries and RX/TX in the PMD are examples of such multi-thread unsafe functions. The RX/TX of the *PMD* are the most critical aspects of a DPDK application and it is recommended that no locking be used with these paths as it will impact performance. However, these functions can be safely used from multiple threads when each thread is performing I/O on a different NIC queue. If multiple threads are to use the same hardware queue on the same NIC port, then locking or some other form of mutual exclusion is necessary. In the NXP implementation, each thread has to use a software portal (DPIO) instance to access the underlying DPAA hardware. Thus, it is recommended that only one thread per logical core should be created for RX/TX and other I/O access to DPAA hardware.

9.2.2.2 DPAA: Supported DPDK Features

Following is the list of DPDK NIC features which DPAA driver support:

- Allmulticast mode
- Basic stats
- Extended stats
- Flow control
- Firmware Version information
- Jumbo frame

- L3 checksum offload
- L4 checksum offload
- Link status
- MTU update
- Promiscuous mode
- Queue start/stop
- Speed Capabilities
- Scattered RX
- Unicast MAC filter
- RSS Hash
- Packet type parsing
- ARMv8

9.2.2.3 DPAA2: Supported DPDK Features

Following is the list of DPDK NIC features which DPAA2 driver support:

- Allmulticast mode
- Basic stats
- Firmware Version information
- Flow control
- Jumbo frame
- L3 checksum offload
- L4 checksum offload
- Link Status
- Link Status Events
- MTU update
- Packet type parsing
- Promiscuous mode
- Queue start/stop
- RSS hash
- Unicast MAC filter
- VLAN offload
- VLAN filter
- Speed capabilities
- ARMv8
- Linux VFIO
- Extended stats

9.2.2.4 PPFE supported DPDK features

Following is the list of DPDK NIC features which PPFE driver support:

- ALLmulticast mode
- Basic Stats
- MTU update
- Promiscuous mode
- Packet type parsing
- ARMv8

9.2.3 Build DPDK

This section includes three sub-sections which detail:

1. Building DPDK binaries (libraries and sample applications) using the Flexbuild build system.
2. Building DPDK binaries as standalone package, through DPDK's own build system.
3. Building Pktgen application which can be used as a software packet generator using DPDK as underlying layer.

9.2.3.1 Build DPDK using Flexbuild

DPDK is one of the application packages of the Flexbuild system. This section details method to build DPDK as a standalone package within the Flexbuild environment. It is assumed that the Flexbuild environment has already been configured before executing the commands below.

Refer to LSDK Quick Start on page 46 for complete details of using the Flexbuild build system.

Once the Flexbuild environment has been setup, following commands can be used to build DPDK applications and libraries. Generated files (libraries and binaries) would be available in the `<Flexbuild>/build/apps/components_arm64` folder. Once the *rootfs* (root filesystem) is generated, the *components_arm64* folder would be merged in it.

```
flex-builder -c openssl -a arm64     # to resolve the dependency on OpenSSL package
flex-builder -c linux -a arm64     # to resolve the dependency of KNI module

flex-builder -c dpdk -a arm64      # build dpdk application


flex-builder -c pktgen-dpdk -a arm64  # to generate dpdk pktgen application
```

NOTE

DPDK is dependent on OpenSSL package for software crypto and OpenSSL PMD. It is necessary to build OpenSSL before DPDK in Flexbuild environment to suffice this dependency. If building DPDK on target platform, it is possible that OpenSSL libraries are already available in library path. In this case, building OpenSSL library would not be required.

Refer How to build LSDK with Flexbuild on page 60 for packing these binaries into the target *rootfs* using the Flexbuild build system.

Layout of DPDK Binaries

Single image of DPDK binary supports both the DPAA and DPAA2, and PPFE platforms. Once the DPDK package has been installed, binaries would be available in `/usr/local/bin` folder in the rootfs. Flexbuild system generates a single rootfs for all NXP platforms it supports.

```
/usr/local/bin             # Contains the sample applications listed in Table 163. DPDK
Application References on page 876
/usr/local/include/dpdk    # All DPDK header necessary for external application development
/usr/local/lib            # Various static DPDK libraries for external application development
```

NOTE

DPDK binaries have been placed in the /usr/local/bin folder to take advantage of the binary search path set in the PATH variable. In case the PATH variable doesn't contain the /usr/local/bin by default, it can be added to it to enable BASH command completion.

At various places in this document, above binaries would be referred for representing execution as well as other information. It is assumed that execution is being done either using the PATH variable set, as explained above, or with absolute path to the binaries.

Besides the above folders, another set of files are also available in *rootfs* to support DPDK application execution. These files are available in the /usr/local/dpdk folder in the *rootfs*.

Table below depicts various DPDK artifacts which are available in the Flexbuild generated rootfs:

S/No	File/Image name related to **/usr/local/**	Description
1	./lib/lib*.a	DPAA, DPAA2, and PPFE Static Libraries for compiling external applications.
2	./include/dpdk/*.h	DPAA, DPAA2, and PPFE Headers for compiling external applications.
2	./bin/l2fwd ./bin/l3fwd ./bin/l2fwd-crypto ./bin/ipsec-secgw ./bin/testpmd	DPAA, DPAA2, and PPFE DPDK Example applications and PMD test application.
3	./dpaa/usdpaa_config_ls<PLAT>.xml ./dpaa/usdpaa_policy_hash_ipv4_1queue.xml ./dpaa/usdpaa_policy_hash_ipv4_2queue.xml ./dpaa/usdpaa_policy_hash_ipv4_4queue.xml	DPAA Only. FMC Configurations and Policy files. <PLAT> is platform name for DPAA platform, for example ls1043 or ls1046. Each Policy file for defining the number of queues per port as mentioned in its name.
4	./dpaa2/dynamic_dpl.sh ./dpaa2/destroy_dynamic_dpl.sh	DPAA2 Only. Dynamic DPL container creation and teardown script.
5	./share/dpdk/usertools/dpdk-setup.sh ./share/dpdk/usertools/dpdk_devbind.py	DPAA, DPAA2, and PPFE DPDK NIC binding utility. This is only applicable for executing DPDK applications in VM.

Table continues on the next page...

Table continued from the previous page...

6	`./disable_services.sh`	When executing a Ubuntu OS over Layerscape board, performance on core 0 can become non-deterministic because of OS services and threads. This script disables the extra services on a Ubuntu OS so that all cores of a board can be used without major impact to the performance.

7.	`./examples/ipsec_secgw/ep0.cfg`
	`./examples/ipsec_secgw/ep1.cfg`
	`./ipsec/ep0_64X64.cfg`
	`./ipsec/ep1_64X64.cfg`
	`./ipsec/ep0_64X64_proto.cfg`
	`./ipsec/ep0_64X64_sha256.cfg`
	`./ipsec/ep1_64X64_proto.cfg`
	`./ipsec/ep1_64X64_sha256.cfg`
		Configuration files for `ipsec-gw` example application.
		The `ep0` and `ep1` files are standard configurations for 2 tunnels for encryption and decryption, each. The `ep0_64X64` and `ep1_64X64` are for 64 tunnels for encryption and decryption, each.

8.	`/usr/bin/pktgen`	Packetgeneration application
9.	./debug_dump.sh	Dumping the debug data for further analysis.

9.2.3.2 Standalone build of DPDK Libraries and Applications

This section details steps required to build DPDK binaries (libraries and example applications) in a standalone environment. This environment can either be on a host enabled for cross building for Layerscape boards or directly on the Layerscape target board.

NOTE

This section primarily focuses on standalone building of DPDK on a host machine using cross compilation for Layerscape boards as target. Though, necessary notes have been added to enable compilation directly on target boards. Refer How to build LSDK with Flexbuild on page 60 for creating an environment suitable for building DPDK on Layerscape boards.

For steps detailing building DPDK using Flexbuild system, refer How to build LSDK with Flexbuild on page 60 and Build DPDK using Flexbuild on page 880.

Obtain the DPDK source code

The DPDK source code contains all the necessary libraries for build example applications as well as test applications. The source code also includes various configuration and scripts for supporting build and execution. Obtain the DPDK source code using the link below:

-
  ```
  git clone https://source.codeaurora.org/external/qoriq/qoriq-components/dpdk -b
  github.qoriq-os/integration
  ```

  Once the above repository has been cloned, DPDK source code is available for compilation. This source is common for both, DPAA, DPAA2, and PPFE platforms.

Prerequisites before compiling DPDK

Before compiling DPDK as a standalone build, following dependencies need to be resolved independently:

- Platform compliant and compiled Linux Kernel source code so that KNI modules can be built.

- — This is optional and if KNI module support is not required, this can be ignored.
- — For details of compiling platform compliant Linux Kernel, refer How to build LSDK with Flexbuild on page 60.
- — For disabling KNI module, see notes below.

- OpenSSL libraries required for building software crypto driver (OpenSSL PMD).
  - — OpenSSL package needs to be separately compiled and libraries installed at a known path before DPDK build can be done.
  - — This is optional and if software crypto driver support is not required, this dependency can be ignored.

NOTE

Refer to How to build LSDK with Flexbuild on page 60 for more information on how to build OpenSSL as part of Flexbuild system. If using Flexbuild and referring to this link for building OpenSSL package, commands specified below can be skipped.

Following steps are for building OpenSSL as a standalone package, outside the Flexbuild system. This is not a preferred way and should be used only if Flexbuild system is not available. Follow the steps given below to build OpenSSL package.

```
git clone git://git.openssl.org/openssl.git  # Clone the OpenSSL source code
cd openssl                                   # Change into cloned directory
```

```
git checkout OpenSSL_1_0_2l                  # Checkout the specific branch supported
by DPDK
```

Export the Cross Compilation tool chain for building OpenSSL for target. The following step for exporting cross compilation toolchain is required only when compiling on Host. On a target board, it is assumed default build toolchain would be used.

```
export CROSS_COMPILE=<path to uncompressed toolchain archive>/bin/aarch64-linux-gnu-
```

Configure the OpenSSL build system with following command. The `--prefix` argument specifies a path where OpenSSL libraries would be deployed after build completes. This is also a path which would be provided to DPDK build system for accessing the compiled OpenSSL libraries.

```
./Configure linux-aarch64  --prefix=<OpenSSL library path> shared
```

```
make depend
make
make install
export OPENSSL_PATH=<OpenSSL library path>
```

NOTE

When building DPDK on target board, it is possible that OpenSSL libraries required by DPDK are already available as part of the *rootfs*, in which case external compilation of OpenSSL package would not be required.

  - — For disabling OpenSSL PMD support, see notes below.

Compiling DPDK

Follow the below steps to compile DPDK once the above prerequisites are resolved. These steps are common for DPAA and DPAA2 targets and are needed only when cross compiling on a host for Layerscape boards as target. In case of direct compilation on target boards, it is assumed that prerequisites would be satisfied using the root filesystem. In case root filesystem doesn't contain necessary prerequisites, below steps would be required once prerequisites have been built/obtained independently.

1. Setup the environment for compilation
   a. Setup Linux Kernel path. This is optional and required only for KNI and ixgb_uio module compilation. Skip it, if ixgb_uio or KNI module or KNI example application is not required.

      ```
      export RTE_KERNELDIR=<Path to compiled Linux kernel to compile KNI kernel module>
      ```

   b. Setup cross compilation toolchain.

This step is required only on the host environment where default toolchain is not for target boards. When compiling on a target board, this step can be skipped.

```
export CROSS=<path to cross-compile toolchain>
```

c. Setup OpenSSL path for software crypto drivers (OpenSSL PMD). This is optional and can be skipped in case software crypto driver (OpenSSL PMD) support is not required.

```
export OPENSSL_PATH=<path to installed OpenSSL>
```

2. Use DPDK build system for compiling DPDK.

NOTE

DPDK binaries generated using below steps are compatible for DPAA, DPAA2, and PPFE platforms.. This is also valid when DPDK is build through Flexbuild build system. Refer How to build LSDK with Flexbuild on page 60 for steps to build DPDK using Flexbuild build system.

a. Execute the following command:

```
make T=arm64-dpaa-linuxapp-gcc install DESTDIR=<location to install DPDK>
```

Where `DESTDIR=<location to install DPDK>` is an optional parameter to deploy all the DPDK binaries (libraries and example applications) to a standard Linux package specific layout within a directory represented by this parameter. Alternatively, a directory named `arm64-dpaa-linuxapp-gcc` is also created and binaries and libraries are also available in it.

b. **Disabling KNI and other kernel module compilation**: In case DPDK kernel modules is not required (`RTE_KERNELDIR` variable is not set), use the following command. `DESTDIR` can be added, as explained above, if required.

```
make T=arm64-dpaa-linuxapp-gcc CONFIG_RTE_KNI_KMOD=n CONFIG_RTE_EAL_IGB_UIO=n

CONFIG_RTE_EAL_IGB_UIO=n

 install
```

c. **Enabling software crypto driver support**: Software crypto driver (OpenSSL PMD) is disabled by default. If it is required set `OPENSSL_PATH` variable, use the following command. `DESTDIR` can be added, as explained above, if required.

```
make T=arm64-dpaa-linuxapp-gcc CONFIG_RTE_LIBRTE_PMD_OPENSSL=y EXTRA_CFLAGS="-I$
{OPENSSL_PATH}/include/" EXTRA_LDFLAGS="-L${OPENSSL_PATH}/lib/" install
```

d. In case KNI is not required and software crypto support is required, use the following command. `DESTDIR` can be added, as explained above, if required..

```
make T=arm64-dpaa-linuxapp-gcc CONFIG_RTE_LIBRTE_PMD_OPENSSL=y EXTRA_CFLAGS="-I$
{OPENSSL_PATH}/include/" EXTRA_LDFLAGS="-L${OPENSSL_PATH}/lib/"
CONFIG_RTE_KNI_KMOD=n CONFIG_RTE_EAL_IGB_UIO=n install
```

NOTE

For more information about the DPDK build system, refer DPDK Documentation.

NOTE

DPDK `arm64-dpaa-linuxapp-gcc` folder contains `.config` file for storing the build configuration. Another way of disabling or enabling support features, like KNI and software crypto drivers, is to edit this file before executing the `make` command. If this method is adopted, parameters to command line for disabling the feature are not required.

NOTE

If KNI or software crypto driver support is disabled using the `make` command line parameters, it would *not* modify the configuration file for DPDK in the `<target>` folder. Every subsequent compilation of DPDK or example application would need to include the same command line arguments to avoid failure because of missing features which were not compiled. Or, edit the `.config` folder in the `arm64-dpaa-linuxapp-gcc` build folder.

Compiling DPDK Example applications

Once the DPDK source code has been compiled, the DPDK example applications can be built independently as required.

1. Before the example applications can be built, the path to DPDK SDK needs to be set which includes the DPDK source code. This would be used by build system to look for compiled libraries and headers.

   ```
   export RTE_SDK=<path to DPDK source code, where compilation was done>
   ```

2. Target should be set to same value as done for compilation of DPDK.

   ```
   export RTE_TARGET=arm64-dpaa-linuxapp-gcc
   ```

3. Once the above variables are set, example applications can be compiled using the following commands:

   ```
   make -C examples/l3fwd            # for the L3 forwarding application

   make -C examples/l2fwd            # for the L2 forwarding application

   make -C examples/ip_fragmentation # for the IP fragmentation application

   make -C examples/ip_reassembly    # for the IP reassembly application

   make -C examples/ipsec-secgw      # for the IPSec gateway application
   ```

   ```
   make -C examples/ipsec-secgw CONFIG_RTE_LIBRTE_PMD_OPENSSL=y EXTRA_CFLAGS="-I$
   {OPENSSL_PATH}/include/" EXTRA_LDFLAGS="-L${OPENSSL_PATH}/lib/"             # for IPSec
   application with openssl PMD
   ```

   ```
   make -C examples/l2fwd-crypto     # for the L2 forwarding with crypto support application
   ```

   ```
   make -C examples/l2fwd-crypto CONFIG_RTE_LIBRTE_PMD_OPENSSL=y EXTRA_CFLAGS="-I$
   {OPENSSL_PATH}/include/" EXTRA_LDFLAGS="-L${OPENSSL_PATH}/lib/"            # for L2
   forwarding crypto operations with openssl PMD
   ```

   Above are sample commands for a limited set of DPDK example applications. Other applications too be compiled using similar command pattern.

   NOTE

   - All the example applications currently supported by DPDK are available as part of the DPDK source code in the `./examples/` folder. Other examples can also be compiled using the pattern stated above.
   - It should be noted though that verification of all example applications has not been done for DPAA, DPAA2 and PPFE platforms
   - `testpmd` is not supported on PPFE platform . Only l2fwd, l3fwd and ipsec-secgw examples are supported by PPFE.

4. Once the example application are compiled, the binaries would be available in the following folder within the DPDK source code folder:

   ```
   examples/l3fwd/build/app/*        # for L3 forwarding application binary and map file
   ```

   Similar to the pattern above, binaries for the example applications compiled would be available in their respective `build/app` folder.

   Besides the above example application, DPDK also provides a `testpmd` binary which can be used for comprehensive verification of DPDK driver (PMD) features for available and compatible devices. This binary is compiled by default during DPDK source compilation explained in Compiling DPDK section.

   NOTE

   Only a small set of DPDK example applications are currently deployed to root filesystem (`/usr/local/bin`) when compiling DPDK through Flexbuild build system. These are: `l2fwd`, `l3fwd`, `l2fwd-crypto`, `ipsec-gw` and `testpmd`.

9.2.3.3 DPDK based Packet Generator

Pktgen is a packet generator powered by DPDK. It requires DPDK environment for compilation and DPDK compliant infrastructure for execution. DPAA and DPAA2 DPDK PMD (Poll Mode Drivers) can be used by Pktgen for building a packet generator using the DPAA infrastructure.

Prerequisities for compiling Pktgen

For compiling Pktgen, **libpcap** library is required. If **Pktgen** is being built as a cross compiled target, the `libpcap` too should be compiled against the same compiled. If using the Flexbuild system, libpcap can be obtained as an external package from Ubuntu repository. Refer How to build LSDK with Flexbuild on page 60 for more information.

NOTE

For libpcap library compilation and deployment, refer Tcpdump and libpcap project pages. Libpcap current and past releases can be obtained from this link. Documentation for libpcap is included in its source code. Also note that libpcap should be compiled for target board if working in a cross compilation environment.

Obtaining the Pktgen source code

Fetch the **Pktgen** source code using the following clone command:

```
git clone http://dpdk.org/git/apps/pktgen-dpdk
git checkout pktgen-3.4.3
```

Compiling Pktgen

Compilation steps below assume that compiled DPDK binaries (libraries and headers) are available in build directory generated by DPDK. Refer DPDK Build Steps for compiling DPDK and creating the build (`arm64-dpaa-linuxapp-gcc`) directory. Further, it is expected that `libpcap` libraries and headers are also present in this build folder.

Export the path to DPDK build environment and build folder defined by the compilation target:

```
export RTE_SDK=<path to compiled DPDK source code containing build folder>
```

```
export RTE_TARGET=<arm64-dpaa-linuxapp-gcc or arm64-dpaa-linuxapp-gcc>  # Select the build
folder based on required DPAA or DPAA2 target>
```

Build the source code:

```
make
```

Before executing the Pktgen application

For executing the Pktgen application, *Pktgen.lua* file and *pktgen* binary are needed on the execution environment.

If build was done using a cross compiled environment, transfer these binaries to the target environment from the build host. If the compilation was done on the target board, skip this step.

```
cd <Pktgen compiled source code>
cp Pktgen.lua <target board>
cp app/app/arm64-dpaa-linuxapp-gcc/pktgen <target board>
```

9.2.3.4 Build OVS-DPDK using Flexbuild

OVS is a popular multilayer virtual switch for enabling massive network automation through programmatic extensions.

OVS-DPDK is one of the application packages of the Flexbuild system which used DPDK as underlying framework. This section details method to build OVS-DPDK as a standalone package within the Flexbuild environment. It is assumed that the Flexbuild environment has already been configured before executing the commands below.

Refer to LSDK Quick Start on page 46 for complete details of using the Flexbuild build system.

NOTE

In the Flexbuild configurations, OVS-DPDK needs to be configured to '`y`' for enabling packaging of OVS-DPDK in Flexbuild generated root filesystem, if not already enabled. For more information, refer How to build LSDK with Flexbuild.

Once the Flexbuild environment has been setup, following commands can be used to build OVS-DPDK package. Generated files (libraries and binaries) would be available in the `<Flexbuild>/build/apps/components_arm64` folder. Once the *rootfs* (root filesystem) is generated, the *components_arm64* folder would be merged in it.

```
$ flex-builder -c ovs-dpdk
```

NOTE

OVS-DPDK is dependent on DPDK package as it is used as its underlying framework. Flexbuild is designed to compile DPDK before OVS-DPDK if not already built.

Layout of OVS-DPDK Binaries

A OVS-DPDK binary image supports both the DPAA and DPAA2 platforms. Once the OVS-DPDK package has been installed, binaries would be available in `/usr/local/` folder in the rootfs. Flexbuild system generates a single rootfs for all NXP platforms it supports.

NOTE

OVS-DPDK binaries are deployed into the root filesystem as per the default layout of installation target for OVS-DPDK build system.

Table below depicts various OVS-DPDK artifacts which are available in the Flexbuild generated rootfs:

S/No	File/Image name related to `/usr/local/`	Description

1	`./bin/ovs-dpctl`
	`./bin/ovs-vsctl`
	`./bin/ovsdb-client`
	`./sbin/ovsdb-server`
	`./sbin/ovs-vswitchd`
	And various other binaries installed by OVS package as default.
		For both, DPAA and DPAA2, platforms.
		Various OVS binaries.

2	`./share/man/man7/ovs-*`	Various OVS man-pages.

9.2.3.5 Virtual machine (VM or guest) images

This section describes steps for deploying a Virtual Machine and executing DPDK applications in it. Additionally, OVS-DPDK package is used for deploying a software switch on the host machine through which virtual machines communicate with other virtual machine or external network.

NOTE

For obtaining necessary artifacts (kernel image, rootfs) for booting up a virtual machine on Layerscape board, refer Configuring and Building on page 954 KVM/Qemu.

9.2.4 Executing DPDK Applications on Host

This section describes how to execute DPDK and related applications in both Host and VM environments.

NOTE

`IP_ADDR_BRD`, `IP_ADDR_IMAGE_SERVER`, and `TFTP_BASE_DIR` are not U-Boot or Linux environment variables. They are used in this document to represent:

1. **IP_ADDR_BRD**: IP address of target board in test setup.
2. **IP_ADDR_IMAGE_SERVER**: IP address of the machine where all the software images are kept. These images are transferred to the board using either tftp or scp.
3. **TFTP_BASE_DIR**: TFTP base directory of TFTP server running on the machine where images are kept.

9.2.4.1 Prerequisites for running DPDK Applications

This section describes the procedures once the target platform is booted up and logged into the Linux shell. This section is applicable to DPAA, DPAA2 and PPFE platforms and is organized as follows:

- Generic Setup contain common steps to be executed before executing any of DPDK sample application or external DPDK applications. One of these sections would be relevant depending on the platform DPAA, DPAA2 or PPFE being used.
- Application specific sections contain steps on how to execute the DPDK example and related applications.

  For more details refer the following topics:

  — Test Environment Setup on page 890

  — Generic Setup - DPAA on page 891

  — Generic Setup - DPAA2 on page 892

  — Generic Setup - PPFE on page 893

9.2.4.2 Booting up the Target board

Follow the instructions mentioned in LSDK Quick Start on page 46 to get the target board up and working.

NOTE

While bringing up a DPAA2 platform specific board (LS2088A or LS1088A), use the following boot arguments to obtain best performance. This can be done by appending the following string to the `bootargs` environment variable in *uboot*.

```
default_hugepagesz=1024m hugepagesz=1024m hugepages=8 isolcpus=1-7
```

Above setting insures that at least 8GB of hugepages are available with the application. `isolcpus` insures that Linux Kernel doesn't use these CPUs for scheduling its tasks - that prevents context switching of any application running on these cores. If the installed memory is lesser, lower number of hugepages can be used.

NOTE

DPAA1 platorms (LS1043ARDB, LS1046A) may have lower memory e.g (2GB) and lower number of cores. In that case, following string can be appended to `bootargs`:

```
default_hugepagesz=2MB hugepagesz=2MB hugepages=448 isolcpus=1-3
bportals=s0 qportals=s0
```

Above setting insures that at least 448 hugepages are available with the application. `isolcpus` insures that Linux Kernel doesn't use these CPUs for scheduling its tasks - that prevents context switching of any application running on these cores. If the installed memory is lower, you may use lower number of hugepages. The bportals and qportals ensures that only 1 portal is available for kernel use (since only one core is for kernel), rest are available for user space.

NOTE

PPFE platform (LS1012ARDB) have lower memory e.g (1GB) and single core. In that case, following string can be appended to `bootargs`:

```
default_hugepagesz=2MB hugepagesz=2MB hugepages=256
```

NOTE

For UEFI, to update the boot arguments please refer to UEFI section in the user manual.

Update grub.cfg file for hugepage and isolcpus related changes.

On DPAA2 platforms: `"rootwait=20 default_hugepagesz=1024m hugepagesz=1024m hugepages=8 isolcpus=1-7"`

On DPAA1 platforms: `"rootwait=20 default_hugepagesz=2MB hugepagesz=2MB hugepages=448 isolcpus=1-3 bportals=s0 qportals=s0"`

NOTE

For the DPAA platform, DPDK specific Device Tree file (for example, `fsl-ls1046a-rdb-usdpaa.dtb` for LS1046A and `fsl-ls1043a-rdb-usdpaa.dtb` for LS1043A) should be used for booting up the board. This Device tree file is configured to provide userspace applications with network interfaces.

Also note that once the above mentioned Device Tree configuration is used, all FMAN ports would be available in the userspace only. Changes to the Device Tree file would be required to assign some of the FMAN ports to Linux Kernel.

One can use the following method to replace default `fsl-ls104xa-rdb-sdk.dtb` with `fsl-ls104xa-rdb-usdpaa.dtb` to support DPDK on LS104XRDB platforms.

Example 1: After enterring Ubuntu on the board, run following instructions for LS1046ARDB:

```
cd /boot
mv fsl-ls1046a-rdb-sdk.dtb fsl-ls1046a-rdb-ori.dtb
ln -s fsl-ls1046a-rdb-usdpaa.dtb fsl-ls1046a-rdb-sdk.dtb
```

then reboot the board

As an alternative, the following method can be used.

Example 2: On the host computer, run the following instructions for LS1046ARDB:

```
cd flexbuild
source setup.env
sed -i 's/fsl-ls1046a-rdb-sdk.dtb/fsl-ls1046a-rdb-usdpaa.dtb/g' configs/
board/ls1046ardb/manifest
flex-builder -i mkdistroscr -a arm64
```

The new auto boot script will be in build/firmware/u-boot/ls1046ardb/ls1046ardb_boot.scr.

Then replace the old non-DPAA boot script in SD cards' boot partition with the one you just generated.

NOTE

Optionally follow the below instructions to assign one of the FMAN ports on LS104x (DPAA) RDB boards to Linux.

With standard flexbuild generated dtb all interfaces will be assinged to either Linux or Userspace. When using `fsl-ls1043a-rdb-sdk.dtb` or `fsl-ls1046a-rdb-sdk.dtb` all network interfaces will be assigned to Linux. When using `fsl-ls1046a-rdb-usdpaa.dtb` or `fsl-ls1046a-rdb-usdpaa.dtb` all network interfaces will be assigned to user space.
The example below shows the changes that are required to assign one network interface to Linux and configure FMAN to support DPDK applications.

Example: Modify `fsl-ls1046a-rdb-usdpaa.dts` file to assign FMAN ports to Linux by removing the following ethernet node that corresponds to FM0-MAC3 (RGMII-1).

```
ethernet@2 {
        compatible = "fsl,dpa-ethernet-init";
        fsl,bman-buffer-pools = <&bp7 &bp8 &bp9>;
        fsl,qman-frame-queues-rx = <0x54 1 0x55 1>;
        fsl,qman-frame-queues-tx = <0x74 1 0x75 1>;
};
```

Then modify the file `usdpaa_config_ls1046.xml` (located in `/usr/local/dpdk/dpaa`) by removing the corresponding port entry. For example the below entry needs to be removed for FM0-MAC3(RGMII-1):

```
<port type="MAC" number="3" policy="hash_ipsec_src_dst_spi_policy_mac3"/
>
```

9.2.4.3 Prerequisities for running DPDK Applications

This section describes the procedures once the target platform is booted up and logged into the Linux shell. This section is applicable to DPAA, DPAA2 and PPFE platforms and is organized as follows:

- Generic Setup - DPAA on page 891 , Generic Setup - DPAA2 on page 892 and Generic Setup - PPFE contain common steps to be executed before executing any of DPDK sample application or external DPDK applications. One of these sections would be relevant depending on the platform, DPAA, DPAA2 or PPFE, being used.
- Application specific sections contain steps on how to execute the DPDK example and related applications.

9.2.4.3.1 Test Environment Setup

Test Environment Setup

Various sample application execution steps are detailed in the following sections. Figure below describes the setup containing the DUT (Device Under Test) and the Packet Generator (Spirent, Ixia or any other software/hardware packet generator). This is applicable for the commands provided in following section.

The setup includes a one-to-one link between DUT and Packet generator unit. DPDK application running on the DUT is expected to forward the traffic from one port to another. The setup below and commands described in following sections can be scaled for more number of ports.

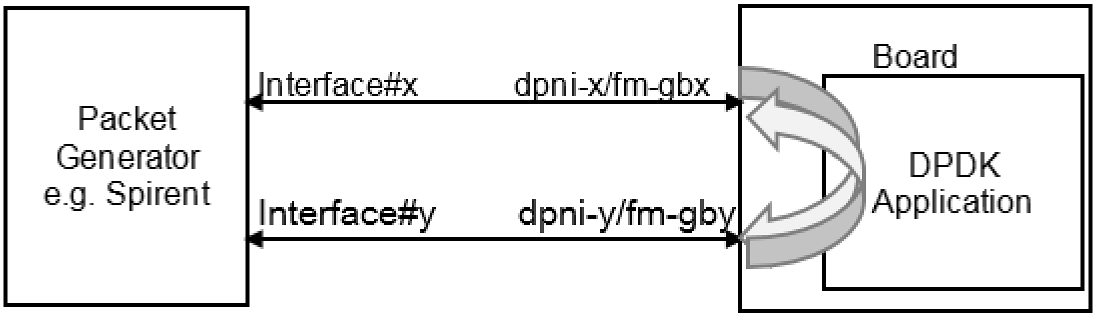


Figure 215. Test Setup

9.2.4.3.2 Generic Setup - DPAA

This section details steps required to setup necessary environment for execution of DPDK applications on DPAA platform. This section is applicable for sample as well as any external DPDK applications. For further details about the applicable configuration file for DPAA platform, refer to Build OVS-DPDK using Flexbuild on page 886. For DPAA2 platform specific setup, refer to Generic Setup - DPAA2 on page 892.

DPAA Hardware Configuration files

DPAA platforms supports hardware acceleration of packet queues. These queues need to be configured in the *FMAN (Frame Manager)* prior to being used. This can be done by choosing the appropriate policy configuration file packaged along with Flexbuild rootfs or DPDK source code.

Either of 1, 2, or 4 queue based policy files can be selected before application is executed. For example, 1 queue policy file would define single queue per physical interface of DPAA. Similarly, 2 and 4 queue are for defining 2 or 4 queues for each defined interface, respectively.

NOTE

Once selected, it is not possible to change the configuration for number of max queues, without rebooting the board. Configuration file should be selected based on requirement and this constraint.

Following are the available platform specific configuration files:

- `usdpaa_config_ls1043.xml` for LS1043A board
- `usdpaa_config_ls1046.xml` for LS1046A board

Following are the available policy files:

- `usdpaa_policy_hash_ipv4_1queue.xml` for 1 queue per port
- `usdpaa_policy_hash_ipv4_2queue.xml` for 2 queues per port
- `usdpaa_policy_hash_ipv4_4queue.xml` for 4 queues per port

NOTE

It is important to execute the applications using the same queue configuration as per the policy file used. This is because once the queue configuration is done, DPAA hardware would distribute packets across configured number of queues. Not consuming packets from any queue would lead to queue buildup eventually stopping the I/O.

Setting up the DPAA Environment

Configure number of queues using environment variable:

```
export DPAA_NUM_RX_QUEUES=<Number of queues>
```

Based on the number of queues defined in the above parameter, select the policy configuration file and execute the `fmc` binary:

```
fmc -c <Configuration file> -p <Policy File> -a
```

For example, in case of LS1043 platform, using 1 queue, following would be the command to execute:

```
export DPAA_NUM_RX_QUEUES=1
fmc -c ./usdpaa_config_ls1043.xml -p ./usdpaa_policy_hash_ipv4_1queue.xml -a
```

NOTE

It is important that value of `DPAA_NUM_RX_QUEUES` matches to the policy file being used. In case of mismatch, DPDK application may show unexpected behavior.

NOTE

LSDK 18.03 (or dpdk release 18.02) onwards DPAA platforms enables the push mode by default. However, if you are using eventdev or have limited number of portal resources, set

```
#export DPAA_PUSH_QUEUES_NUMBER=0
```

Setup hugepages for DPDK application to use for packet and general buffers. This step can be ignored if hugepages are already mounted. Use command `mount | grep hugetlbfs` to check if hugepages are already setup.

```
mkdir /mnt/hugepages
```

```
mount -t hugetlbfs none /mnt/hugepages
```

Hereafter, DPDK sample applications are ready to be executed on the DPAA platform.

Cleanup of the DPAA Environment

To remove the configuration done using the `fmc` tool, use the `-x` parameter. It is a good practice to cleanup the configuration before setting up a new configuration. Even in cases where change of configuration is required, for example, increasing the number of queues supported, following command can be used for cleaning up the previous configuration.

```
fmc -x
```

9.2.4.3.3 Generic Setup - DPAA2

This section details steps required to setup necessary environment for execution of DPDK applications over DPAA2 platform. This section is applicable for sample as well as any external DPDK applications. For further details about the applicable configuration file for DPAA2 platform, refer to Build OVS-DPDK using Flexbuild on page 886. For DPAA platform specific setup, refer to Generic Setup - DPAA on page 891.

These steps must be performed before running any of the DPDK application on host.

Setting up the DPAA2 Environment

For executing DPDK application on DPAA2 platform, a resource container needs to be created which contains all necessary interfaces to the DPAA2 hardware blocks. Necessary configuration scripts are provided with DPDK package for creating and destroying containers.

1. Configure the DPAA2 resource container with `dynamic_dpl.sh` script. This script is available under `/usr/local/dpdk/dpaa2/dpdk-extras` folder in the rootfs.

   ```
   cd /usr/local/dpdk/dpaa2/     # Or, any other folder if custom installation of DPDK is done
   ./dynamic_dpl.sh <DPMAC1.id> <DPMAC2.id> ... <DPMACn.id>
   ```

   In the above command, `<DPMAC1.id>` refers to the DPAA2 MAC resource, for example, `dpmac.1` or `dpmac.2`. Modify the above command as per the number of physical MAC ports required by the application (constrained by availability and connectivity on the DUT).

   Output of `dynamic_dpl.sh` command shows the name of the container created. This name is passed to DPDK applications using the `DPRC` environment variable. Following block shows sample output of the `dynamic_dpl.sh` command:

   ```
   #################### Container  dprc.2  is created ####################
   Container dprc.2 have following resources :=>
   ```

```
  * 16 DPBP
  * 5 DPCON
  * 4 DPSECI
  * 3 DPNI
  * 10 DPIO
  * 10 DPCI

######################### Configured Interfaces #########################

Interface Name        Endpoint              Mac Address
==============        ========              ==================
dpni.1                dpmac.1               -Dynamic-
dpni.2                dpmac.2               -Dynamic-
```

The MAC addresses are auto-assigned by the DPDK applications after fetching information from the firmware. These would be same as the one programmed by u-boot. For creating flows, see the application output or note the MAC addresses during board bootup. Testpmd application can also be used to find the MAC address assigned.

NOTE

It is possible to modify the number of interfaces (`DPBP`, `DPCON`, `DPNI`, etc) in a container. This can be done by defining environment variable `COMPONENT_COUNT=<number>` before executing the script. For example, to set number of DPBP to 4, use `export DPBP_COUNT=4`.

Though the flexibility has been provided to modify the interfaces in the container, note that resources need to be balanced and changing any count will require corresponding changes to other interfaces. Incorrect changes can render the DPDK application unable to execute.

2. Setup the environment variable using the container name reported by `dynamic_dpl.sh` command:

   ```
   export DPRC=dprc.2
   ```

Once the above setup is complete, DPDK application can be executed on the DPAA2 platform.

Teardown of DPAA2 Environment

It might be required to change the configuration of the resource contain to modify the components included in it. As the number of resources in the system are limited, number of containers which can be created as also limited. It is possible to remove an existing container and create another.

Execute the following command to teardown a container:

```
cd /usr/local/dpdk/dpaa2                   # Or, any other folder if custom installation of
DPDK is done
./destroy_dynamic_dpl.sh <Container Name>  # for example, "dprc.2"
```

9.2.4.3.4 Generic Setup - PPFE

This section provides steps required to setup necessary environment for execution of DPDK applications over PPFE platform.

These steps must be performed before running any of the DPDK application on host.

Setting up the PPFE Environment

For executing DPDK application on PPFE platform, a kernel module `pfe.ko` must be loaded in user space mode which will do the necessary initialization to run the DPDK applications. By default, `pfe.ko` will be loaded automatically during kernel bootup. User must ensure the value of `/sys/module/pfe/parameters/us` is 1 to check pfe.ko module is loaded in user space mode. If `/sys/module/pfe/parameters/us` is not `1`, then user shall unload the module and then load again with module argument as `us=1`.

```
rmmod pfe.ko
insmod pfe.ko us=1
```

Additionally, user must run the below commands to fulfill DPDK applications huge pages requirements.

```
mkdir /mnt/hugepages
mount -t hugetlbfs none /mnt/hugepages
```

9.2.4.3.5 DPAA2: Multiple parallel DPDK Applications

This section describes steps for executing multiple parallel DPDK application on DPAA2 platform.

For executing multiple DPDK applications, each application instance should run with its own resource container (*DPRC*). This constraint is because of the way DPDK framework is designed to use a given container for exclusive use, irrespective of resources within, and bind it using VFIO layer. This design prevents parallel access to single resource container from multiple parallel instances of a single DPDK application, multiple parallel execution of different DPDK applications.

Creating Multiple DPRC instances

Using the resouce container script documented in this section, create multiple resource container instances on host. Following command creates a resource container with 2 network interfaces (and all other resources necessary to run a DPDK application).

First DPRC: (assuming name as `dprc.2` through rest of the document)

```
cd /usr/local/dpdk/dpaa2                      # Or, any other folder if custom installation of
DPDK is done
./dynamic_dpl.sh <DPMAC1.id> <DPMAC2.id>    # For example, execute ./dynamic_dpl.sh dpmac.1
dpmac.2
```

Second DPRC: (assuming name as `dprc.3` through rest of the document)

```
cd /usr/local/dpdk/dpaa2                      # Or, any other folder if custom installation of
DPDK is done
./dynamic_dpl.sh <DPMAC3.id> <DPMAC4.id>    # For example, execute ./dynamic_dpl.sh dpmac.3
dpmac.4
```

Executing multiple DPDK Applications

Once the resource containers are created, on two separate terminals, execute the following commands to run **l2fwd** application, bridging traffic between both interfaces available in the container:

```
export DPRC=dprc.2
cd /usr/local/bin
./l2fwd -c 0x3 -n 1 --file-prefix=p1 --socket-mem=1024 -- -p 0x3 -q 1
```

Some of the arguments, which are deviations from general **l2fwd** command, are explained below:

`--file-prefix`: Each DPDK Application attempts to allocate some hugepages for DMA'd area. This allocation is done in the hugepages through the use of *hugepage* mount, by creating and mapping a file. This arguments instructs the EAL to append a string to the file name. This way, multiple instances, having different such arguments, wouldn't attempt to open same hugepage mapping file.

`--socket-mem`: Passed to EAL, this instructs the EAL to allocate only specified amount of memory from the hugepages. By default, if this is not provided, a DPDK application would acquire all possible hugepages (all free pages) available on the Linux system.

For the second instance, command like following can be executed:

```
export DPRC=dprc.3
cd /usr/local/bin
./l2fwd -c 0xc -n 1 --file-prefix=p2 --socket-mem=1024 -- -p 0x3 -q 1
```

Note the difference of values for **`-c`** and **`--file-prefix`** between the first and second command.

9.2.4.4 DPDK example applications

DPDK example application binaries are available in the `/usr/local/bin` folder in the Flexbuild generated rootfs.

NOTE

Command snippets below assume that commands are executed while being present in `/usr/local/bin` or appropriate `PATH` variable has been set. Also, a DPDK binary can be executed on both, DPAA and DPAA2, platform without any modifications.

NOTE

Only a selected few DPDK example applications have been deployed in the root filesystem by default. For non-deployed example application, compilation needs to be done using DPDK source code. Refer Standalone build of DPDK Libraries and Applications on page 882 for more details.

NOTE

For PPFE platform, since LS1012ARDB has only 1 core, so `-c` with 0x1 is only acceptable core mask for all DPDK applications. Additionally, user must provide the `—vdev` argument with value `eth_pfe` to enable ethernet device for DPDK applications.

L2fwd – Layer 2 Forwarding Application

Sample application to show forwarding between multiple ports based on the Layer 2 information (switching).

```
l2fwd -c 0x2 -n 1 -- -p 0x1 -q 1 -T 0
```

In the above command: `-c` refers to the core mask for cores to be assigned to DPDK; `-p` is the port mask for ports to be used by application; `-q` defines the number of queues to serve on each port. Other command line parameters may also be provided - for a complete list, refer L2 Forwarding Sample Application (in Real and Virtualized Environments).

NOTE

- `isolcpus` provided as boot argument to u-boot assures that isolated cores are not scheduled by Linux kernel. Using Core 0 for DPDK application can lead to non-deterministic behavior, including drop in performance. It is recommended that DPDK applicaton core mask values avoid using Core 0.
- L2fwd application periodically prints the I/O stats. To avoid CPU core to be interrupted because of these scheduled prints, '-t 0' option can be appended at the end of command line.
- Command to run `l2fwd` on LS1012ARDB:

  ```
  ./l2fwd -c 0x1 -n 1 --vdev 'eth_pfe0' --vdev='eth_pfe1' -- -p 0x3 -q 3
  ```

NOTE

For best performance on LS1046ARDB, use the following command. This includes an option `-b 7` which sets optimal I/O burst size:

```
l2fwd -c 0x2 -n 1 -- -p 0x1 -q 1 -T 0 -b 7
```

L2fwd–qdma - Layer 2 Forwarding using QDMA

Sample application to show forwarding between multiple ports based on the Layer 2 information (switching) using QDMA.

```
l2fwd-qdma -c 0x2 -n 1 -- -p 0x1 -q 1 -m 1 -T 0
```

In the above command: `-c` refers to the core mask for cores to be assigned to DPDK; `-p` is the port mask for ports to be used by application; `-q` defines the number of queues to serve on each port. `-m` mode specifies HW (-m is 0) or Virtual (-m is 1) mode for QDMA queues. Apart from `-m` parameter other parameters are similar to DPDK l2fwd application -refer L2 Forwarding Sample Application (in Real and Virtualized Environments).

L3fwd – Layer 3 Forwarding Application

Sample application to show forwarding between multiple ports based on the Layer 3 information (routing).

```
l3fwd -c 0x6 -n 1 -- -p 0x3 --config="(0,0,1),(1,0,2)"
```

In the above command: `-c` refers to the core mask for cores to be assigned to DPDK; `-p` is the port mask for ports to be used by application; `--config` is *(Port, Queue, Core)* configuration used by application for attaching cores to queues on each port. Other command line parameters may also be provided - for a complete list, refer L3 Forwarding Sample Application.

Other variations of the above command described below change the configuration of ports, queue and cores services them.

1. 4 core - 2 Port, 2 queues per port:

```
l3fwd -c 0xF -n 1 -- -p 0x3 -P --config="(0,0,0),(0,1,1),(1,0,2),(1,1,3)"
```

2. 4 core - 2 Port with destination MAC address:

```
l3fwd -c 0xF -n 1 -- -p 0x3 -P --config="(0,0,0),(0,1,1),(1,0,2),(1,1,3)" --eth-
dest=0,11:11:11:11:11:11 --eth-dest=1,00:00:00:11:11:11
```

3. 8 core - 2 Port with 4 queues per port:

```
l3fwd -c 0xFF -n 1 -- -p 0x3 --config="(0,0,0),(0,1,1),(0,2,2),(0,3,3),(1,0,4),(1,1,5),
(1,2,6),(1,3,7)"
```

NOTE

Although, above command snippets use the Core 0 for DPDK application, for best performance Core 0 use is not recommended as Linux OS schedules its tasks on it. It is also recommended that `isolcpus` be used in Linux boot argument to prevent Linux from scheduling tasks on other Cores.

NOTE

Example command to run l3fwd on LS1012ARDB:

```
./l3fwd -c 0x1 --vdev='eth_pfe0' --vdev='eth_pfe1' -n 1 -- -p 0x3 --
config="(0,0,0),(1,0,0)" -P
```

NOTE

For best performance on LS1046ARDB, use the following command. This includes an option `-b 7` which sets optimal I/O burst size:

```
l3fwd -c 0xF -n 1 -- -p 0x3 -P -b 7 --config="(0,0,0),(0,1,1),(1,0,2),
(1,1,3)"
```

This is valid for any configuration of cores, queues and ports (i.e., `--config` option).

L2fwd-Crypto – Layer 2 Forwarding using DPAA/DPAA2 CAAM Hardware

This variation of Layer 2 forwarding application uses DPAA/DPAA2 CAAM block for encryption of packets.

- Layer 2 forwarding with Cipher only support:

```
l2fwd-crypto -c 0x2 -n 1 -- -p 0x1 -q 1  --chain CIPHER_ONLY  --cipher_algo aes-cbc --
cipher_op ENCRYPT --cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10
```

- Layer 2 forwarding with Cypher-Hash support:

```
l2fwd-crypto -c 0x2 -n 1 -- -p 0x1 -q 1 --chain CIPHER_HASH  --cipher_algo aes-cbc --
cipher_op ENCRYPT --cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --
auth_algo sha1-hmac --auth_op GENERATE --auth_key_random_size 64
```

- Layer 2 forwarding with Hash only support:

```
l2fwd-crypto -c 0x2 -n 1 -- -p 0x1 -q 1 --chain HASH_ONLY --auth_algo sha1-hmac --auth_op
GENERATE --auth_key_random_size 64
```

L2fwd-Crypto – Layer 2 Forwarding using OpenSSL Software Instructions

This variation of Layer 2 forwarding application uses OpenSSL library for performing software crypto operations. Internally, the OpenSSL library would use the ARMCE instructions specific for ARM CPUs. For DPDK, this application uses the OpenSSL PMD as its underlying driver.

NOTE

This command requires support of OpenSSL package while building the DPDK applications. Refer this section of this document, for details about toggling compilation of software crypto support, which includes the OpenSSL driver.

- Cipher_only
    - For DPAA Platform

- 1 core: Depending on the platform being executed on, append the above blacklisting parameter to the end of this command:

  ```
  l2fwd-crypto --vdev "crypto_openssl"  -c 0x2 -n 1 -- -p 0x1 -q 1 --chain CIPHER_ONLY
  --cipher_algo aes-cbc --cipher_op ENCRYPT --cipher_key
  01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --cryptodev_mask 0x10
  ```

- 2 core: Depending on the platform being executed on, append the above blacklisting parameter to the end of this command:

  ```
  l2fwd-crypto --vdev "crypto_openssl0" --vdev "crypto_openssl1" -c 0x6 -n 1 -- -p
  0x3 -q 1  --chain CIPHER_ONLY  --cipher_algo aes-cbc --cipher_op ENCRYPT --
  cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --cryptodev_mask 0x30
  ```

- 4 core: Depending on the platform being executed on, append the above blacklisting parameter to the end of this command:

  ```
  l2fwd-crypto --vdev "crypto_openssl0" --vdev "crypto_openssl1" --vdev
  "crypto_openssl2" --vdev "crypto_openssl3" -c 0xf -n 1 -- -p 0xf -q 1  --chain
  CIPHER_ONLY  --cipher_algo aes-cbc --cipher_op ENCRYPT --cipher_key
  01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --cryptodev_mask 0xF0
  ```

— For DPAA2 Platform

- 1 core: Depending on the platform being executed on, append the above blacklisting parameter to the end of this command:

  ```
  l2fwd-crypto --vdev "crypto_openssl" -c 0x2 -n 1 -- -p 0x1 -q 1 --chain CIPHER_ONLY
  --cipher_algo aes-cbc --cipher_op ENCRYPT --cipher_key
  01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --cryptodev_mask 0x100
  ```

- 2 core: Depending on the platform being executed on, append the above blacklisting parameter to the end of this command:

  ```
  l2fwd-crypto --vdev "crypto_openssl0" --vdev "crypto_openssl1" -c 0x6 -n 1 -- -p
  0x3 -q 1  --chain CIPHER_ONLY  --cipher_algo aes-cbc --cipher_op ENCRYPT --
  cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --cryptodev_mask 0x300
  ```

- 4 core: Depending on the platform being executed on, append the above blacklisting parameter to the end of this command:

  ```
  l2fwd-crypto --vdev "crypto_openssl0" --vdev "crypto_openssl1" --vdev
  "crypto_openssl2" --vdev "crypto_openssl3" -c 0xf -n 1 -- -p 0xf -q 1  --chain
  CIPHER_ONLY  --cipher_algo aes-cbc --cipher_op ENCRYPT --cipher_key
  01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --cryptodev_mask 0xF00
  ```

- Cipher_hash

  — For DPAA Platform:

  - 1 core: Append the above blacklisting parameter to the end of this command:

    ```
    l2fwd-crypto --vdev "crypto_openssl" -c 0x2 -n 1 -- -p 0x1 -q 1 --chain CIPHER_HASH
    --cipher_algo aes-cbc --cipher_op ENCRYPT --cipher_key
    01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --auth_algo sha1-hmac --auth_op
    GENERATE --cryptodev_mask 0x10 --auth_key_random_size 64
    ```

  — For DPAA2 Platform:

  - 1 core: Append the above blacklisting parameter to the end of this command:

    ```
    l2fwd-crypto --vdev "crypto_openssl" -c 0x2 -n 1 -- -p 0x1 -q 1 --chain CIPHER_HASH
    --cipher_algo aes-cbc --cipher_op ENCRYPT --cipher_key
    01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --auth_algo sha1-hmac --auth_op
    GENERATE --cryptodev_mask 0x100 --auth_key_random_size 64
    ```

  In the above ccommands, for scaling to multiple cores or ports, toggle the `-c` and `-p` arguments as described above.

- Hash_cipher

  — For DPAA Platform:

  - 1 core: Append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl" -c 0x2 -n 1 -- -p 0x1 -q 1 --chain HASH_CIPHER
--auth_algo sha1-hmac --auth_op GENERATE --cipher_algo aes-cbc --cipher_op ENCRYPT
--cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --cryptodev_mask 0x10
--auth_key_random_size 64
```

— For DPAA2 Platform:

◦ 1 core: Append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl" -c 0x2 -n 1 -- -p 0x1 -q 1 --chain HASH_CIPHER
--auth_algo sha1-hmac --auth_op GENERATE --cipher_algo aes-cbc --cipher_op ENCRYPT
--cipher_key 01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:0f:10 --cryptodev_mask
0x100 --auth_key_random_size 64
```

In the above commands, for scaling to multiple cores or ports, toggle the `-c` and `-p` arguments.

- Hash_only

— For DPAA Platform:

◦ 1 core: Append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl" -c 0x2 -n 1 -- -p 0x1 -q 1 --chain HASH_ONLY
--auth_algo sha1-hmac --auth_op GENERATE --cryptodev_mask 0x10 --
auth_key_random_size 64
```

— For DPAA2 Platform:

◦ 1 core: Append the above blacklisting parameter to the end of this command:

```
l2fwd-crypto --vdev "crypto_openssl" -c 0x2 -n 1 -- -p 0x1 -q 1 --chain HASH_ONLY
--auth_algo sha1-hmac --auth_op GENERATE --cryptodev_mask 0x100 --
auth_key_random_size 64
```

— For scaling to multiple cores or ports, toggle the `-c` and `-p` arguments as described above.

For more information on L2fwd-crypto application, refer to L2 Forwarding with Crypto Sample Application.

NOTE

Example command to run l2fwd-crypto with openssl on LS1012ARDB (cipher only):

```
./l2fwd-crypto -c 0x1 --vdev='eth_pfe0' --vdev='crypto_openssl' -n 1 -- -p 0x1 -q 1 --chain
CIPHER_ONLY --cipher_algo aes-cbc --cipher_key 00:01:02:03:04:05:06:07:08:09:0a:0b:0c:0d:0e:
0f --cipher_op ENCRYPT
```

IPSec-secgw – IPSec Gateway using DPAA/DPAA2 CAAM Hardware

For IPsec application, two DUTs need to be configured as `endpoint 0 (ep0)` and `endpoint 1 (ep1)`. Assuming that `endpoint` have **4 ports** each:

- Connect Port 1 and Port 3 of the `ep0` and `ep1` to each other (back-to-back).
- Connect Port 0 and Port 2 of the `ep0` and `ep1` to packet generator (for example, Spirent).

The Stream generated by packet generator needs to have IP addresses in following pattern:

```
EP0:
   port 0: 32 flows with destination IP: 192.168.1.XXX, 192.168.2.XXX, ..... ,192.168.31.XXX,
192.168.32.XXX
    port 2: 32 flows with destination IP: 192.168.33.XXX, 192.168.34.XXX, ..... ,
192.168.63.XXX,192.168.64.XXX
EP1:
    port 0: 32 flows with destination IP: 192.168.101.XXX, 192.168.102.XXX, ..... ,
192.168.131.XXX,192.168.132.XXX
    port 2: 32 flows with destination IP: 192.168.133.XXX, 192.168.134.XXX, ..... ,
192.168.163.XXX,192.168.164.XXX
```

Above represents default configurations for the endpoints in `ep0_64X64.cfg` and `ep1_64X64.cfg`. Custom port mappings, SA/SP and the routes can be configured in the corresponding configuration file named as `ep0.cfg` and `ep1.cfg` for respective endpoint. These files are available in Flexbuild generated rootfs; for further details, refer this table.

For more information, refer to IPsec Security Gateway Sample Application .

Endpoint 0 (`ep0`) configuration:

```
ipsec-secgw -c 0xf  -- -p 0xf -P -u 0xa --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3)" -f
ep0_64X64.cfg
```

Endpoint 1 (`ep1`) configuration:

```
ipsec-secgw -c 0xf  -- -p 0xf -P -u 0xa --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3)" -f
ep1_64X64.cfg
```

Running IPSec Gateway application with protocol offload

The DPAA/DPAA2 CAAM hardware also support IPSec protocol offload. The command and configurations are exactly same except the cfg files. For protocol offload, the cfg files are `ep0_64X64_proto.cfg` and `ep1_64X64_proto.cfg`. Performance with protocol offload would be much better than the standard case. *In case of platforms which have 8 cores, the command for 8 core will also be exactly same as non-offload case, except the name of the cfg files.*

Endpoint 0 (`ep0`) configuration:

```
ipsec-secgw -c 0xf  -- -p 0xf -P -u 0xa --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3)" -f
ep0_64X64_proto.cfg
```

Endpoint 1 (`ep1`) configuration:

```
ipsec-secgw -c 0xf  -- -p 0xf -P -u 0xa --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3)" -f
ep1_64X64_proto.cfg
```

Running IPSec Gateway Application with 8 Cores

NOTE

For running this 8 core `ipsec-secgw` use-case, it is required that the default configuration of the DPRC container be modified to have at least 16 DPIO objects. Default configuration for DPRC created using `dynamic_dpl.sh` has max 10 DPIO objects.

For increasing the number of DPIO objects, following steps can be performed:

1. Destroy the existing DPRC container, if any, using "`destroy_dynamic_dpl.sh <container name>`" command.
2. Set number of DPIO to 16 using "`export DPIO_COUNT=16`". It is important to note that DPIO resources are limited in the system. Based on currently used objects (for example, another DPRC container existing, or being used by Linux Kernel), it is possible that higher number of DPIO objects maynot be available.
3. Create DPRC container by using "`dynamic_dpl.sh <list of DPMACS to add>`" command.

For running IPsec application with multiple queues using 64X64 tunnels and with 8 cores, following command and configuration needs to be done:

Endpoint 0 (`ep0`) configuration: Sample configuration for this is available in `ep0_64X64.cfg` file available in `/usr/local/dpdk/dpaa2/` folder in root filesystem.

```
ipsec-secgw -c 0xFF -- -p 0xf -P -u 0xa --config="(0,0,0),(0,1,1),(1,0,2),(1,1,3),(2,0,4),
(2,1,5),(3,0,6),(3,1,7)" -f ep0_64X64.cfg
```

Endpoint 1 (`ep1`) configuration: Sample configuration for this is available in `ep1_64X64.cfg` file available in `/usr/local/dpdk/dpaa2/` folder in root filesystem.

```
ipsec-secgw -c 0xFF -- -p 0xf -P -u 0xa --config="(0,0,0),(0,1,1),(1,0,2),(1,1,3),(2,0,4),
(2,1,5),(3,0,6),(3,1,7)" -f ep1_64X64.cfg
```

IPSec-secgw – IPSec Gateway using OpenSSL PMD

The command, flow stream and port configuration is similar to the IPSec-secgw – IPSec Gateway using DPAA/DPAA2 CAAM Hardware on page 898 command, flow stream and port configuration, except that it uses OpenSSL PMD for crypto operations. Internally, the OpenSSL PMD uses the ARMCE instructions for the ARM CPUs for performaning crypto operations.

- For DPAA Platform:

- Endpoint 0 configuration

  ```
  ipsec-secgw -c 0xf --vdev "crypto_openssl" -- -p 0xf -P -u 0xa --config="(0,0,0),
  (1,0,1),(2,0,2),(3,0,3)" --cryptodev_mask 0x10 -f ep0_64X64.cfg
  ```

  Endpoint 1 configuration

  ```
  ipsec-secgw -c 0xf --vdev "crypto_openssl" -- -p 0xf -P -u 0xa --config="(0,0,0),
  (1,0,1),(2,0,2),(3,0,3)" --cryptodev_mask 0x10 -f ep1_64X64.cfg
  ```

- For DPAA2 Platform:

  - Endpoint 0 configuration

    ```
    ipsec-secgw -c 0xf --vdev "crypto_openssl" -- -p 0xf -P -u 0xa --config="(0,0,0),
    (1,0,1),(2,0,2),(3,0,3)" --cryptodev_mask 0x100 -f ep0_64X64.cfg
    ```

    Endpoint 1 configuration

    ```
    ipsec-secgw -c 0xf --vdev "crypto_openssl" -- -p 0xf -P -u 0xa --config="(0,0,0),
    (1,0,1),(2,0,2),(3,0,3)" --cryptodev_mask 0x100 -f ep1_64X64.cfg
    ```

NOTE

Example command to run ipsec-secgw with openssl on LS1012ARDB:

```
./ipsec-secgw -c 0x1 --vdev='eth_pfe0' --vdev='eth_pfe1' --
vdev='crypto_openssl'  -- -p 0x3 -P -u 0x2 --config="(0,0,0),(1,0,0)" -f
ep0_64X64.cfg
```

KNI - Using Kernel Network Interface Module

The Kernel NIC Interface (KNI) is a DPDK control plane solution that allows userspace applications to exchange packets with the kernel networking stack. For details please refer: http://dpdk.org/doc/guides/sample_app_ug/kernel_nic_interface.html

Loading the KNI kernel module without any parameter. By default only one kernel thread is created for all KNI devices for packet receiving in kernel side:

```
#insmod rte_kni.ko
```

Affine the kni task to a single core e.g core number #1

```
#taskset -pc 1 `pgrep -fl kni_single | awk '{print $1}'`
```

Run the kni application

```
kni [EAL options] -- -P -p PORTMASK --config="(port,lcore_rx,lcore_tx[,lcore_kthread,..])
[,port,lcore_rx,lcore_tx[,lcore_kthread,..]]"
#./kni -c 0xf -n 1 -- -p 0x3 -P --config="(0,0,1),(1,0,1)"
where config is : (PORT, kni lcore Rx core, kni lcore tx core )
```

On another console check the interfaces with:

```
#ifconfig -a
```

Enable the given interface and assign IP address (if any)

Pktgen – DPDK based Software Packet Generator

Pktgen is a software packet generator based on DPDK. Refer DPDK based Packet Generator on page 886 for steps required for building *Pktgen*.

All the commands below assume that *Pktgen* application is either executed from current folder or appropriate path environment variable has been set.

1. 3 Port, 1 Core each

   ```
   pktgen -l 0-3 -n 1 --proc-type auto --file-prefix pg --log-level 8 -- -T -P -m "[1].0,
   [2].1, [3].2"
   ```

2. 1 Port, 2 Core

   ```
   pktgen -l 0-3 -n 1 --proc-type auto --file-prefix pg --log-level 8 -- -T -P -m "[1:2].0"
   ```

3. To start or stop traffic on a specific port:

```
start 0 # start <port number>
stop 0  # stop <port number>
```

4. To start or stop traffic on all ports:

```
str
stp
```

9.2.4.5 Command interface (CMDIF) demo application

DPDK based Command interface (CMDIF) demo application demonstrates the communication between GPP and AIOP using DPDK API's and Command Interface library. Command Interface library is provided as a lib module within `examples/cmdif/` (`examples/cmdif/lib/librte_cmdif.a`).

This application requires a corresponding process running on AIOP core/s, which will read and respond to CMDIF application. CMDIF application is only supported on DPAA2 which will have AIOP.

.

NOTE

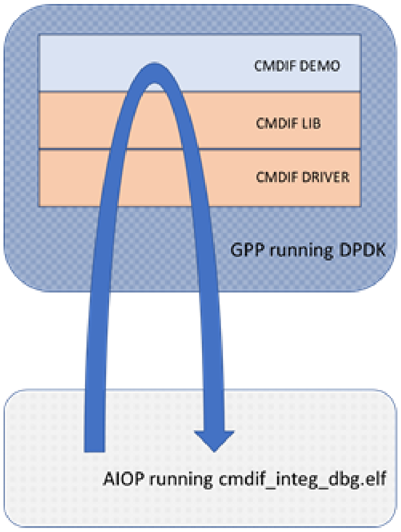
Include the library `librte_cmdif.a`, when you are writing an application over DPAA2 CMDIF based raw device.

The application verifies the following:

1. CMDIF client (where GPP is the client and AIOP is the server)
2. CMDIF server (where GPP is the server and AIOP is the client)

CMDIF Client (GPP is client)

In the CMDIF client, the GPP is the client and the AIOP is the server. Requests are initiated by the GPP and are sent to the AIOP core. The AIOP responds back with the response.

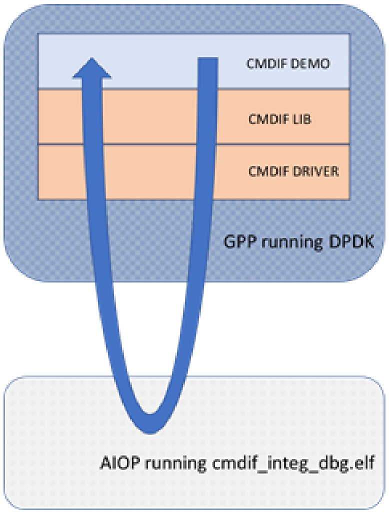


The CMDIF client (demo) is responsible for the following:

- Opens a CI communication channel using a single DPCI device, defined in container used by application
- Sends multiple messages from GPP to AIOP using synchronous commands
- Sends and receive response for multiple messages from GPP to AIOP using asynchronous commands
- Application Validates the response received from the AIOP Server application and prints the result on console
- Closes the opened CI communication channels

CMDIF Server (GPP is server)

In the CMDIF server, the GPP is the server and the AIOP is the client. Requests are initiated by the AIOP and are sent to the GPP core. The GPP responds back to the AIOP with success or error.



The CMDIF server (demo) is responsible for the following:

- Registers the server module
- Opens the Sever session
- Initiates the client open on the AIOP client
- Receives requests/commands from the AIOP
- Closes the server session
- Unregisters the module

Running demo application

The demo application showcases only a single thread or core use-case, thus supporting the coremask with single core.

NOTE

`dynamic_dpl.sh` is not required to run along with `cmdif_demo`.

Executing demo application example also requires the following:

- Running `dynamic_AIOP_dpl.sh` (*instead of dynamic_dpl.sh*)
- Loading the `cmdif_integ_dbg.elf` (provided in AIOPSL - https://github.com/qoriq-open-source/aiopsl/tree/integration/demos/images) using the `aiop_tool` which needs to run in background.

  For example:

```
./dynamic_AIOP_dpl.sh
export DPRC = > dprc container created for GPP<
aiop_tool load -g dprc.3 -f cmdif_integ_dbg.elf &
cmdif_demo -c 0x2
```

Description about the command:

- `dynamic_AIOP_dpl.sh` – creates three containers
  - — First one for the AIOP
  - — Second one for the aiop_tool which loads the AIOP FW
  - — Third one for DPDK's use **(Use this container name as `$DPRC` export variable)**
- The -c option enables cores 2

Expected output

The application should print below logs on console in case of CMDIF client:

- PASSED open commands
- PASSED synchronous send commands
- PASSED asynchronous send/receive commands
- PASSED: close commands

Also, verify that application prints below logs in console in case of CMDIF server:

- PASSED cmdif session open
- PASSED sync command
- PASSED Async commands
- PASSED Isolation context test
- PASSED cmdif session close

9.2.5 OVS-DPDK and DPDK in VM with VIRTIO Interfaces

DPDK example and DPDK-based applications can also run inside the virtual machine. This section describes steps to run these applications inside the virtual machine on both DPAA and DPAA2 platforms.

The virtual machine runs inside the host Linux system and is launched by an application called QEMU.

NOTE

While using the virtual machine, the console logs for the guest Linux do not appear on the host Linux console (i.e UART). The guest logs are exposed through `telnet`, and they can be accessed by doing `telnet` on the host board's IP Address (`IP_ADDR_BRD`) and `GUEST_CONSOLE_TELNET_PORT`. Each Virtual machine that is run on a single host is allocated a different `GUEST_CONSOLE_TELNET_PORT`, and this port number is specified by user running virtual machine through the QEMU command line.

Following is the layout of the sub-sections of this chapter:

- Generic steps on page 903 describing steps required for QEMU setup for both, DPAA and DPAA2 platforms.
- Configuring OVS on page 904 describing steps necessary to launch OVS-DPDK on the host machine for switching traffic between VMs and external network.
- Various sections for lauching a virtual machine and executing a DPDK application:
  - Launch Virtual Machine on page 906 for launching a virtual machine.
  - Accessing virtual machine console on page 907 for accessing a virtual machine console from a network connected machine over `telnet`.
  - Launching two virtual machines on page 908 for launching more than one virtual machine.
  - Running DPDK applications in VM on page 908 for running DPDK applications in the virtual machine.
- Multi Queue VIRTIO support on page 910 describes steps for DPDK application using multiple queues.

9.2.5.1 Generic steps

Refer to Configuring and Building on page 954 KVM/Qemu for detailed information about deploying virtual machines using KVM/QEMU using Layerscape boards.

The reference above serves as base for deploying Virtual Machines and DPDK application in them. All following sections assume that kernel image and virtual machine rootfs is available with DPDK sample application images in it.

NOTE

Give IP Address to the board so that virtual machine consoile can be accessed using telnet.

```
ifconfig eth<x> <IP_ADDR_BRD>
```

9.2.5.2 Configuring OVS

OVS-DPDK application binary and configuration files are available in the `/usr/local` folder in the Flexbuild generated rootfs.

It is assumed that before executing command snippets in this section, necessary steps mentioned in Generic steps on page 903 have already been executed.

NOTE

Command snippets below assume that commands are executed while being present in `/usr/local/` folder. Or, appropriate `PATH` variable has been set. As the OVS commands are spread across multiple folder, each command snippet also shows the location of these binaries relative to above folder.

Command snippets below assume that commands are executed while being present in this folder or appropriate PATH variable has been set.

OVS is used as a back-end for VHOST USER ports. The physical ports on the target platform and the vhost user ports (virtio devices) are added to ovs-vswitch and the flows in OVS are programmed so as to establish traffic switching between physical ports and vhost devices as follows:

- Incoming traffic Physical port1 `=>` output to vhost-user port 1
- Incoming traffic on vhost-user port1 `=>` output on physical port 1
- Incoming traffic on physical port 2 `=>` output on vhost-user port 2
- Incoming traffic on vhost-user port 2 `=>` output on physical port 2

The following steps must be followed to setup OVS as vhost switching back-end:

1. **Reset the OVS environment.**

   ```
   pkill -9 ovs
   rm /usr/local/etc/openvswitch/conf.db
   rm -rf /usr/local/var/run/openvswitch/vhost-user-1
   rm -rf /usr/local/var/run/openvswitch/vhost-user-2
   ```

2. **Specify the initial Open vSwitch (OVS) database to use:**

   ```
   mkdir -p /usr/local/etc/openvswitch  # If the folder doesn't already exist
   mkdir -p /var/log/openvswitch        # to ensure that OVS logging can be done
   mkdir -p /usr/local/var/run/openvswitch
   cd /usr/bin/ovs-dpdk/
   ./ovsdb-tool create /usr/local/etc/openvswitch/conf.db ./vswitch.ovsschema
   ./ovsdb-server --remote=punix:/usr/local/var/run/openvswitch/db.sock --
   remote=db:Open_vSwitch,Open_vSwitch,manager_options --pidfile=/tmp/ovsdb-server.pid --
   detach --log-file=/var/log/openvswitch/ovs-vswitchd.log
   export DB_SOCK=/usr/local/var/run/openvswitch/db.sock
   ```

3. Configure the OVS to support DPDK ports:

   ```
   ./ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-init=true
   ```

4. Configure OVS to work with 1G Hugepages

   ```
   export SOCK_MEM=1024
   ./ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk-socket-mem="$SOCK_MEM"
   ```

5. **Define Cores for OVS Operations**

```
export OVS_SERVICE_MASK=0x1
export OVS_CORE_MASK=0x6
```

```
./ovs-vsctl  --no-wait set Open_vSwitch . other_config:dpdk-lcore-mask=$OVS_SERVICE_MASK
```

```
./ovs-vsctl  --no-wait set Open_vSwitch . other_config:pmd-cpu-mask=$OVS_CORE_MASK
```

NOTE

`OVS_CORE_MASK` should be chosen such as to not include Core 0. `OVS_SERVICE_MASK` should be any core which is not already assigned to `OVS_CORE_MASK`. This way, OVS services threads (defined by `OVS_SERVICE_MASK`) will not compete for CPU scheduling with OVS I/O threads (`OVS_CORE_MASK`). `OVS_SERVICE_MASK` can be set to Core 0 as defined in example abov

6. **Start the ovs-vswitchd daemon:**

```
./ovs-vswitchd unix:$DB_SOCK --pidfile --detach -c $OVS_CORE_MASK
```

NOTE

–`detach` option makes the daemon run in background. If this option is given same shell can be used to run further commands, otherwise ssh to the target board and run further commands. Each time you reboot or there is an OVS termination, you need to rebuild the OVS environment and repeat steps 1-6 of this section

7. **Create an OVS bridge.**

```
./ovs-vsctl add-br br0 -- set bridge br0 datapath_type=netdev
```

8. **Create DPDK port**

For creating DPDK ports with OVS, platform specific port information needs to be provided to OVS.

-
  ```
  ./ovs-vsctl add-port br0 dpdk0 -- set Interface dpdk0 type=dpdk options:dpdk-
  devargs=dpni.1
  ```

  ```
  ./ovs-vsctl add-port br0 dpdk1 -- set Interface dpdk1 type=dpdk options:dpdk-
  devargs=dpni.2
  ```

  Above commands attach the DPAA2 ports `dpni.1` and `dpni.2` with OVS. In case different ports are required, above command should be modified accordingly.

9. **Create vhost-user port**

```
./ovs-vsctl add-port br0 vhost-user1 -- set Interface vhost-user1 type=dpdkvhostuser
```

```
./ovs-vsctl add-port br0 vhost-user2 -- set Interface vhost-user2 type=dpdkvhostuser
```

10. **Commands to Configure Multi Queues**

```
./ovs-vsctl set Interface dpdk0 options:n_rxq=4
./ovs-vsctl set Interface dpdk1 options:n_rxq=4
./ovs-vsctl set Interface dpdk0 options:n_txq=4
./ovs-vsctl set Interface dpdk1 options:n_txq=4
```

NOTE

The above commands are required only in case of multi-queue use case(Four queues been used in above reference commands). For single queue mode no commands needed as OVS by default configures single queue.

11. **Delete OVS flow**

```
./ovs-ofctl del-flows br0
```

12. **Set OVS flow rules for external-to-external path:**

NOTE

The commands below configure a hard-coded bi-directional data path between Port 1 and Port 2. Use this step only for OVS external-to-external testing. For OVS Host-to-VM configuration, skip and continue with next step.

```
./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=1,actions=output:2
```

```
./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=2,actions=output:1
```

13. **Set OVS flow rules between Host to VM:**

   NOTE

   The steps below configure OVS such that `Port 1 <=> Port 3` and `Port 2 <=> Port 4` are connected to each other. If a different configuration is required, the commands below should be altered as well as VM configurations.

   ```
   ./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=1,actions=output:3
   ./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=3,actions=output:1
   ./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=2,actions=output:4
   ./ovs-ofctl add-flow br0 -O OpenFlow13 table=0,in_port=4,actions=output:2
   ```

   NOTE

   OVS Switch (`ovs-vswitchd`) must be run before launching the virtual machine using QEMU, otherwise the virtual machine launch will fail.

14. Run the following command to enable emc-cache lookups in OVS. This helps in enhancing the lookup speed to ensure better performance.

   ```
   ./ovs-vsctl --no-wait set Open_vSwitch . other_config:emc-insert-inv-prob=1
   ```

15. **Verify the Flows inserted:**

   ```
   ./ovs-ofctl dump-flows br0
   ```

9.2.5.3 Launch Virtual Machine

This section describes necessary environment setup and commands for launching a Virtual Machine (VM).

It is assumed that before executing command snippets in this section, necessary steps mentioned in Generic steps on page 903 and Configuring OVS on page 904 have already been executed.

Setup the environment

For accessing the VM, *telnet* is used. This environment variable defines the *telnet* port to be used.

```
export GUEST_CONSOLE_TELNET_PORT=4446    # Telnet port to be used for accessing the virtual
machine
```

```
export ROOTFS_IMG=<VM_ROOTFS_IMG>
```

Define other environment variables which are used by the QEMU command to configure the virtual machine environment:

```
export VM_MEM=2048M
export VM_CORES=2
export NUM_QUEUES=1
```

NOTE

- VM_CORES are the number of cores to reserve for the virtual machine operation.

Export the following paths:

```
export VHOST1_PATH=/usr/local/var/run/openvswitch/vhost-user1
export VHOST2_PATH=/usr/local/var/run/openvswitch/vhost-user2
```

Launch QEMU and virtual machine

Launch the QEMU emulator using the following command.

```
qemu-system-aarch64 -nographic -object memory-backend-file,id=mem,size=$VM_MEM,mem-path=/mnt/
hugepages,share=on -cpu host -machine type=virt -kernel /boot/Image -enable-kvm  -serial tcp::
$GUEST_CONSOLE_TELNET_PORT,server,telnet -append 'root=/dev/vda rw console=ttyAMA0,115200
rootwait earlyprintk' -m $VM_MEM -numa node,memdev=mem -chardev socket,id=char1,path=
$VHOST1_PATH -netdev type=vhost-user,id=hostnet1,chardev=char1,vhostforce,queues=$NUM_QUEUES
-device virtio-net-pci,disable-modern=false,addr=0x3,netdev=hostnet1,id=net1,mrg_rxbuf=off -
chardev socket,id=char2,path=$VHOST2_PATH -netdev type=vhost-
user,id=hostnet2,chardev=char2,vhostforce,queues=$NUM_QUEUES -device virtio-net-pci,disable-
```

```
modern=false,addr=0x4,netdev=hostnet2,id=net2,mrg_rxbuf=off -smp $VM_CORES -S -drive
if=none,file=$ROOTFS_IMG,id=foo,format=raw -device virtio-blk-device,drive=foo
```

NOTE

For best performance, Core 0 in the VM should not be used for DPDK I/O threads.

Also, to avoid system services from using GPUs scheduled for DPDK I/O threads, it is recommended that `isolcpus` be used for isolating cores from Linux Kernel scheduling in VM. The exact configuration is dependent on number of CPU assigned by QEMU to VM using the `VM_CORES` environment variable.

Append `isolcpus=1-$VM_CORES` to the '`root=/dev/vda rw console=ttyAMA0,115200 rootwait earlyprintk`' string in the `qemu-system-aarch64` command given above.

NOTE

Extra care should be taken for value assigned to `mem-path` variable. It should point to a valid mounted hugepage filesystem. In case the value assigned to `mem-path` is not a valid hugepage filesystem, Qemu would create a mmap'd file for its work which might negatively impact performance.

Following logs will appear on the host UART console:

```
QEMU 2.5.0 monitor - type 'help' for more information
(qemu) QEMU waiting for connection on: disconnected:telnet::4446,server
```

NOTE

Complete QEMU logs are visible only when `telnet` is used for logging into the guest machine, as described in Accessing virtual machine console on page 907.

The `-S` option mentioned in the qemu command stops the virtual machine bootup after initial setup. Run the `info cpus` command on QEMU CLI interface to see the QEMU threads.

```
(qemu) info cpus
* CPU #0: thread_id=2559
  CPU #1: (halted) thread_id=2560
```

SSH on the board (`telnet` to `IP address IP_ADDR_BRD`) from other console and affine the threads to the cores using the `taskset` command:

```
taskset -p 0x4 <tid1>
taskset -p 0x8 <tid1>
```

NOTE

It is recommended to affine the VCPUs to the cores on which OVS threads are not running. For better performance VCPU threads should be given one physical CPU each if possible.

Run the `c` command from the QEMU CLI to continue the VM boot-up:

```
(qemu) c
```

9.2.5.4 Accessing virtual machine console

Telnet to the `IP_ADDR_BRD` at port `GUEST_CONSOLE_PORT` from any machine, which can reach `IP_ADDR_BRD` over network:

```
telnet  192.168.1.141 4446
Trying 192.168.1.141...
Connected to 192.168.1.141.
Escape character is '^]'.
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 4.4.65 (root@dash1) (gcc version 5.4.0 20160609 (Ubuntu/Linaro
5.4.0-6ubuntu1~16.04.4) ) #1 SMP PREEMPT Fri Jun 23 07:34:43 IST 2017
```

Only a partial terminal output has been shown above.

9.2.5.5 Launching two virtual machines

This section describes steps for launching 2 virtual machine simultaenously for multiple VM use case.

NOTE

- Memory assigned to each virtual machine should not exceed the total number of huge pages assigned on system. In following example, 2048Mb to each virtual machine has been specified and verified to be working correctly.
- Console `telnet` port of both virtual machine must be different. In the below example, VM1 has port 4446 and VM2 has port 4447 configured for `telnet`. Modify the command accordingly if different values are required.

Launch VM1:

```
qemu-system-aarch64 -nographic -object memory-backend-file,id=mem,size=$VM_MEM,mem-path=/mnt/
hugepages,share=on -cpu host -machine type=virt -kernel /boot/Image -enable-kvm  -serial tcp::
4446,server,telnet -append 'root=/dev/vda rw console=ttyAMA0,115200 rootwait earlyprintk' -m
$VM_MEM -numa node,memdev=mem -chardev socket,id=char1,path=$VHOST1_PATH -netdev type=vhost-
user,id=hostnet1,chardev=char1,vhostforce,queues=$NUM_QUEUES -device virtio-net-pci,disable-
modern=false,addr=0x3,netdev=hostnet1,id=net1,mrg_rxbuf=off -chardev socket,id=char2,path=
$VHOST2_PATH -netdev type=vhost-user,id=hostnet2,chardev=char2,vhostforce,queues=$NUM_QUEUES
-device virtio-net-pci,disable-modern=false,addr=0x4,netdev=hostnet2,id=net2,mrg_rxbuf=off -
smp $VM_CORES -S -drive if=none,file=$ROOTFS_IMG,id=foo,format=raw -device virtio-blk-
device,drive=foo
```

Launch VM2:

```
qemu-system-aarch64 -nographic -object memory-backend-file,id=mem,size=$VM_MEM,mem-path=/mnt/
hugepages,share=on -cpu host -machine type=virt -kernel /boot/Image -enable-kvm  -serial tcp::
4447,server,telnet -append 'root=/dev/vda rw console=ttyAMA0,115200 rootwait earlyprintk' -m
$VM_MEM -numa node,memdev=mem -chardev socket,id=char1,path=$VHOST1_PATH -netdev type=vhost-
user,id=hostnet1,chardev=char1,vhostforce,queues=$NUM_QUEUES -device virtio-net-pci,disable-
modern=false,addr=0x3,netdev=hostnet1,id=net1,mrg_rxbuf=off -chardev socket,id=char2,path=
$VHOST2_PATH -netdev type=vhost-user,id=hostnet2,chardev=char2,vhostforce,queues=$NUM_QUEUES
-device virtio-net-pci,disable-modern=false,addr=0x4,netdev=hostnet2,id=net2,mrg_rxbuf=off -
smp $VM_CORES -S -drive if=none,file=$ROOTFS_IMG,id=foo,format=raw -device virtio-blk-
device,drive=foo
```

9.2.5.6 Running DPDK applications in VM

All the DPDK applications mentioned in this section have been tested in following configuration:

- Two Physical network interfaces.
- Two virtio-net devices in the virtual machine.

Following figure illustrates the test setup:

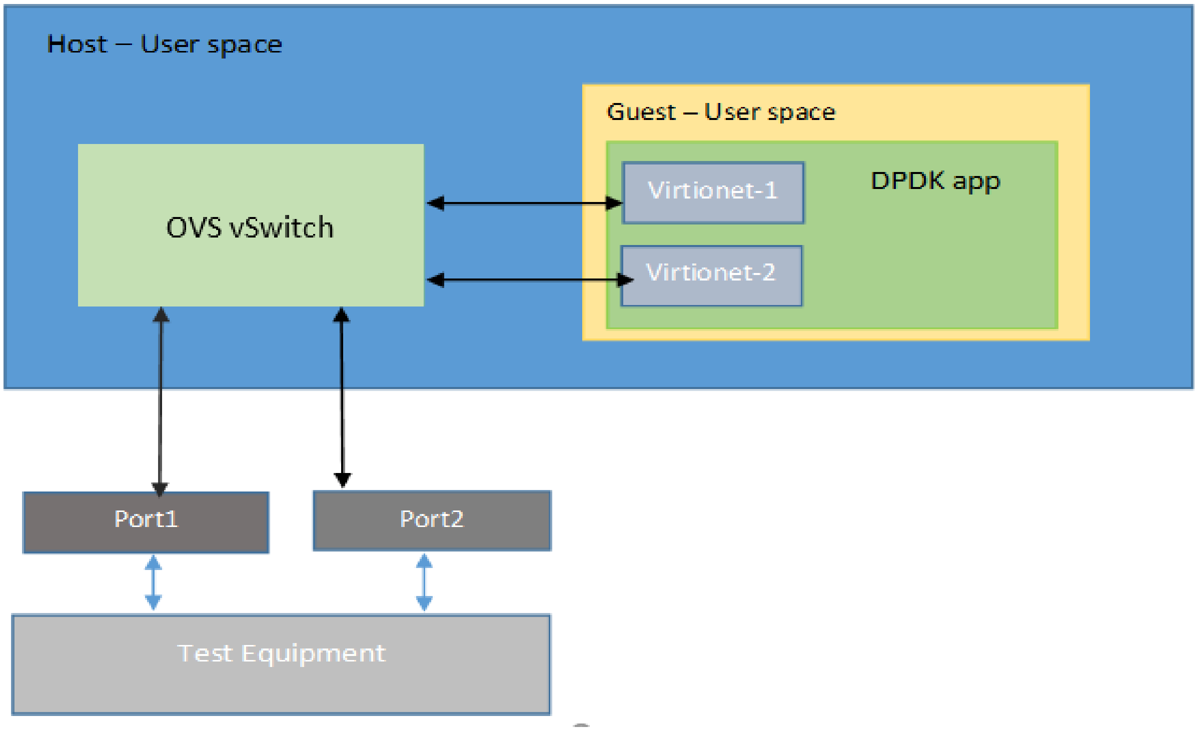


Figure 218. DPDK virtionet test setup

Generic Setup

DPDK example application binaries are available in the `/usr/local/bin` folder in the Flexbuild generated rootfs.

- Setup Hugepages

```
mkdir /mnt/hugepages
mount -t hugetlbfs none /mnt/hugepages
echo 512 > /proc/sys/vm/nr_hugepages ; for dpaa1 change change size as 512
```

> **NOTE**
>
> For the below commands, it is assumed that they are executed from `/usr/local` folder. Modify the commands for different path or `PATH` variable configuration.

- Setup the devices using DPDK Scripts

```
/usr/share/usertools/dpdk-devbind.py --status
/usr/share/usertools/dpdk-devbind.py -b uio_pci_generic 0000:00:03.0
/usr/share/usertools/dpdk-devbind.py -b uio_pci_generic 0000:00:04.0
```

Run DPDK Applications

> **NOTE**
>
> Using Core 0 for DPDK application can lead to non-deterministic behavior, including drop in performance. It is recommended that DPDK applicaton core mask values avoid using Core 0.

Executing `l2fwd` application:

```
bin/l2fwd -c 0x2 -n 1 -- -p 0x1 -q 1 -T 0
```

Executing `l3fwd` application:

```
bin/l3fwd -c 0x2 -n 1 -- -p 0x1 --config="(0,0,1)" -P --parse-ptype
```

Testpmd

- For TX only:

  ```
  ./bin/testpmd -c 0x3 -n 1 -- -i --nb-cores=1  --portmask=0x1 --nb-ports=1 --forward-
  mode=txonly --disable-hw-vlan --port-topology=chained
  ```

- For RX only:

  ```
  ./bin/testpmd -c 0x3 -n 1 -- -i --nb-cores=1  --portmask=0x1 --nb-ports=1 --forward-
  mode=rxonly --disable-hw-vlan --port-topology=chained
  ```

9.2.5.7 Multi Queue VIRTIO support

To scale the performance against the number of VM cores, the VIRTIO devices need to be configured with multiple queues. This section explains the steps required for setup multi queue VIRTIO devices.

See **Generic Setup** of DPAA platform including configuration necessary for defining multiple queues before DPDK application is executed. No special setup is required for DPAA2 before DPDK application start. Further, refer Configuring OVS on page 904 for setting OVS-DPDK on the host. Steps defined below build upon the configurations and steps provided in these sections for multiqueue support.

QEMU commands for multiqueue vhost devices are different and are shown later in the section.

Additional steps for setup of OVS

Besides the steps mentioned in Configuring OVS on page 904, following changes are required to modify the number of supported queues in the virtual machine.

Run following commands after adding DPDK and vhost-user ports to the bridge:

```
./ovs-vsctl set Interface dpdk0 options:n_rxq=2
./ovs-vsctl set Interface dpdk1 options:n_rxq=2
./ovs-vsctl set Interface dpdk0 options:n_txq=2
./ovs-vsctl set Interface dpdk1 options:n_txq=2
./ovs-vsctl set Interface vhost-user1 options:n_rxq=2
./ovs-vsctl set Interface vhost-user2 options:n_rxq=2
./ovs-vsctl set Interface vhost-user1 options:n_txq=2
./ovs-vsctl set Interface vhost-user2 options:n_txq=2
```

Launch VM with multiqueue VHOST devices

Similar to the steps mentioned in Launch Virtual Machine on page 906, following steps are required to start the virtual machine. Changes are highlighted with **bold**:

NOTE

Command snippets shown below are valid for DPAA2 platform. Replace **dpaa2** with **dpaa** for equivalent command on DPAA platform.

```
export GUEST_CONSOLE_TELNET_PORT=4446
export VM_MEM=2048M     # For DPAA1 use VM_MEM=650M
export VM_CORES=2
```

```
export NUM_QUEUES=2
```

```
export ROOTFS_IMG=<VM_ROOTFS_IMG>
```

```
export VHOST1_PATH=/usr/local/var/run/openvswitch/vhost-user1
export VHOST2_PATH=/usr/local/var/run/openvswitch/vhost-user2
```

```
qemu-system-aarch64 -nographic -object memory-backend-file,id=mem,size=$VM_MEM,mem-path=/mnt/
hugepages,share=on -cpu host -machine type=virt -kernel /boot/Image -enable-kvm  -serial tcp::
$GUEST_CONSOLE_TELNET_PORT,server,telnet -append 'root=/dev/vda rw console=ttyAMA0,115200
rootwait earlyprintk' -m $VM_MEM -numa node,memdev=mem -chardev socket,id=char1,path=
$VHOST1_PATH -netdev type=vhost-user,id=hostnet1,chardev=char1,vhostforce,queues=$NUM_QUEUES
-device virtio-net-pci,disable-
modern=false,addr=0x3,netdev=hostnet1,mq=on,id=net1,mrg_rxbuf=off,vectors=6 -chardev
socket,id=char2,path=$VHOST2_PATH -netdev type=vhost-
user,id=hostnet2,chardev=char2,vhostforce,queues=$NUM_QUEUES -device virtio-net-pci,disable-
modern=false,addr=0x4,netdev=hostnet2,mq=on,id=net2,mrg_rxbuf=off,vectors=6 -smp $VM_CORES -S
-drive if=none,file=$ROOTFS_IMG,id=foo,format=raw -device virtio-blk-device,drive=foo
```

DPDK applications in VM

Connect to VM terminal as explained in Accessing virtual machine console on page 907. Once logged-in as Guest, DPDK applications using multiple queues can be run in VM.

NOTE

If the number of queues defined for DPDK application in VM is *not* equal to number of queues (`NUM_QUEUES`) defined in QEMU command, the application may fail to start.

NOTE

For the below commands, it is assumed that they are executed from `/usr/local` folder. Modify the commands for different path or `PATH` variable configuration.

Besides the above steps, all steps are same as described in single queue VM usecase.

Setup the devices using DPDK scripts:

```
/usr/share/usertools/dpdk-devbind.py --status
/usr/share/usertools/dpdk-devbind.py -b uio_pci_generic 0000:00:03.0
/usr/share/usertools/dpdk-devbind.py -b uio_pci_generic 0000:00:04.0
```

Execute `l3fwd` application:

```
./examples/l3fwd  -c 0x3 -n 1 --  -p 0x3 --config="(0,0,0),(0,1,0),(1,0,1),(1,1,1)"  -P --parse-ptype
```

Execute `testpmd` application:

```
./bin/testpmd -c 3 -n 1 -- -i --nb-cores=1 --nb-ports=1 --total-num-mbufs=1025 --forward-mode=txonly --disable-hw-vlan --rxq=2 --txq=2 --port-topology=chained
```

9.2.6 Enabling DPAA2 direct assignment for DPDK

The DPAA2 architecture supports the assignment of direct dpaa2 resource access from the QEMU guest VM (Kernel or userspace app). See **Direct assigned devices**.

This section describes necessary environment setup and commands for launching a Virtual Machine (VM) with VFIO device passthrough or direct device assignment support.

NOTE

ARM-V8 currently support VM to work in NO-IOMMU mode only. Which means that all HW access will use physical address mode only. The default sdk code is build with virtual addressing mode only. You will need to rebuild the the dpdk for `arm64-dpaa2-linuxapp-gcc`, using CONFIG_RTE_LIBRTE_DPAA2_USE_PHYS_IOVA=y to use the dynamic IOVA mode.

To rebuild the DPDK binaries in dynamic IOVA mode: Change CONFIG_RTE_LIBRTE_DPAA2_USE_PHYS_IOVA=y in config/defconfig_arm64-dpaa-linxuapp-gcc or, config/defconfig_arm64-dpaa2-linxuapp-gcc. (whichever config you are using to build the DPDK).

Than follow the instructions in Standalone build of DPDK Libraries and Applications on page 882 section to build DPDK applications.

You can transfer the applications manually to the virtual machine using the host-vm connections as suggested to configure in next section.

9.2.6.1 Launch Virtual Machine

1. Make sure that Kernel is enabled for direct assignment mode. See, Host kernel: Enabling DPAA2 direct assignment on page 955.

2. The default QEMU present in filesystem may not support the direct assignment feature. See, Building QEMU on page 957

   Execute the following commands to build the QEMU 2.9 with VFIO passthrough support locally:

   ```
   git clone https://source.codeaurora.org/external/qoriq/qoriq-components/qemu
   cd qemu
   git checkout qemu-2.9
   git submodule update --init dtc
   ```

   Make sure your m/c has the required packages to build QEMU. Refer the example below:

   ```
   #update your ubuntu m/c with required packages.
   apt-get install pkg-config
   apt-get install libglib2.0-dev
   apt-get install libpixman-1-dev
   apt-get install libaio-dev
   apt-get install libusb-1.0-0-dev
   ```

   Now, build the QEMU

   ```
   ./configure --prefix=/root/qemu-2.9 --target-list=aarch64-softmmu --enable-fdt --enable-
   kvm --with-system-pixman
   make
   make install
   ```

   The new QEMU will be installed in /root/qemu-2.9 folder.

3. Create DPAA2 resources for the VM guest kernel and VM guest userspace (dpdk) on the board.

   The dynamic scripts to support the dpaa2 resource creation are available in (/usr/local/dpdk/dpaa2) for ubuntu rootfs. It is also part of the DPDK source code in `nxp` folder.

   ```
   export DPDK_SCRIPTS=/usr/local/dpdk/dpaa2
   ```

   Create a dpni based interface for file transfer and communication between host and VM guess kernel.

   ```
   ls-addni -n
   Output:---Created interface: eth0 (object:dpni.1, endpoint:)
   ```

   Next create the VM guest kernel container. See *How to use DPAA2 direct assignment* chapters for further information.

   A sample vm_linux conf file is provided in scripts to create the vm guest kernel container. In this, the number of resources are good for 2 core VM. The previously created dpni object is also passed to connect it with guest kernel container.

   ```
   source $DPDK_SCRIPTS/dynamic_dpl.sh -c $DPDK_SCRIPTS/vm_linux.conf dpni.1
   ```

   Next step is to create the container for VM guest userspace for DPDK

   ```
   source $DPDK_SCRIPTS/dynamic_dpl.sh -c $DPDK_SCRIPTS/vm_dpdk.conf <dpmac.1> <dpmac.2>
   ```

   NOTE

   Make sure to enter the created parent DPRC into vm_dpdk.conf

   For the rest of the chapter, it is assumed that VM guest kernel container is dprc.2 and VM guest userspace child container is dprc.3 (nested).

   Create an ethernet connect between host and VM for communication/transfer. This was already created and passed during vm-linux container.

   ```
   #assign IP to host interface created to communicate with VM (dprc.2, eth0)
   ifconfig eth0 192.168.2.2
   ```

4. Create hugepages mount

   ```
   echo hugetlbfs /mnt/hugetlbfs hugetlbfs defaults,mode=0777 0 0 >> /etc/fstab
   mkdir /mnt/hugetlbfs
   mount /mnt/hugetlbfs
   ```

5. Launch qemu (Version: 2.9.0) using following command:

   For generating a root filesystem image, refer Creating a guest Linux root filesystem. Assign the `ROOTFS_IMG` in below command with the absolute path to the generated image.

```
export ROOTFS_IMG=/ubuntu_bionic_arm64_rootfs.ext4.img # Telnet port to be used for
accessing this instance of virtual machine export GUEST_CONSOLE_TELNET_PORT=4446
export KERNEL_IMG=/root/Image-4.14 export KERNEL_IMG=/root/Image-4.14
```

Define other environment variables which are used by the QEMU command to configure the virtual machine environment:

```
export VM_MEM=4096M(2048M for LS1088ARDB)
```

1. Add the device command below (for the GUEST KERNEL DPRC to be assigned) to the QEMU command line:

   ```
   -device vfio-fsl-mc,host=dprc.2
   ```

   Also, make sure to specify the appropriate number of cores for the guest VM. It should match the number of dpio objects created in the child container. In our case, 1 core.

   ```
   -smp $VM_CORES
   ```

2. Start QEMU with -S option (the vcpu threads are not yet started). We need this in order for the Ethernet drivers in the guest to correctly bind the objects to the cores.

   ```
   # single core VM launch /root/qemu-2.9/bin/qemu-system-aarch64 -smp $VM_CORES -m $VM_MEM
   -mem-path /mnt/hugetlbfs -cpu host -machine type=virt,gic-version=3 -kernel $KERNEL_IMG -
   enable-kvm -display none -serial tcp::$GUEST_CONSOLE_TELNET_PORT,server,telnet -drive
   if=none,file=$ROOTFS_IMG,id=foo,format=raw -device virtio-blk-device,drive=foo -append
   'root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk' -monitor stdio -device vfio-fsl-
   mc,host=dprc.2 -S
   ```

   # Two core VM launch (check the isolcpus for core #1 in bootargs).

   NOTE

   - For best performance, Core 0 in the VM should not be used for DPDK I/O threads.
   - To avoid system services from using GPUs scheduled for DPDK I/O threads, it is recommended that isolcpus be used for isolating cores from Linux Kernel scheduling in VM. The exact configuration is dependent on number of CPU assigned by QEMU to VM using the VM_CORES environment variable.

   Append isolcpus=1-$VM_CORES to the 'root=/dev/vda rw console=ttyAMA0,115200 rootwait earlyprintk' string in the qemu-system-aarch64 command given above.

   ```
   /root/qemu-2.9/bin/qemu-system-aarch64 -smp $VM_CORES -m $VM_MEM -mem-path /mnt/hugetlbfs
   -cpu host -machine type=virt,gic-version=3 -kernel $KERNEL_IMG -enable-kvm -display none
   -serial tcp::$GUEST_CONSOLE_TELNET_PORT,server,telnet -drive
   if=none,file=$ROOTFS_IMG,id=foo,format=raw -device virtio-blk-device,drive=foo -append
   'root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk isolcpus=1' -monitor stdio
   -device vfio-fsl-mc,host=dprc.2 -S
   ```

   NOTE

   Make sure to specify the appropriate number of cores for the guest VM. It should match the number of dpio objects created in. Also, make sure that the /mnt/hugetlbfs folder exists and is mounted when starting QEMU.

   Following logs will appear on the host UART console:

   QEMU 2.9.0 monitor - type 'help' for more information

   (qemu) qemu-system-aarch64: -serial tcp::4446,server,telnet: QEMU waiting for connection on: disconnected:telnet::4446,server

3. Launch VM using : telnet <Board ip addr> <GUEST_CONSOLE_TELNET_PORT> For example, telnet localhost 4446

   Make sure to assign each vcpu thread to one physical CPU only.

   Get the VM thread IDs entering QEMU shell.

   ```
   (qemu) info cpus
   * CPU #0: thread_id=7211
   CPU #1: (halted) thread_id=7212
   ```

   Assign one vcpu thread to one core only.

```
$ taskset -p 0x1 7211
pid 7211's current affinity mask: ff
pid 7211's new affinity mask: 1
$ taskset -p 0x2 7212
pid 7212's current affinity mask: ff
pid 7212's new affinity mask: 2
```

start the vcpu threads:

```
(qemu) c
```

9.2.6.2 Accessing the virtual machine console

```
root@ls2088ardb:~# telnet localhost 4446
Trying ::1...
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.14.16-00004-ga5a4b5d (b10814@bf-netperf1.idc) (gcc version
7.2.1 20171011 (Linaro GCC 7.2-2017.11)) #2 SMP PREEMPT Tue Apr 3 12:24:09 IST 2018
[    0.000000] Boot CPU: AArch64 Processor [410fd082]
[    0.000000] Machine model: linux,dummy-virt
[    0.000000] efi: Getting EFI parameters from FDT:
[    0.000000] efi: UEFI not found.
[    0.000000] cma: Reserved 16 MiB at 0x00000000ff000000
[    0.000000] NUMA: No NUMA configuration found
----\
Ubuntu 16.04.3 LTS localhost ttyAMA0
localhost login: root
Password:
Last login: Wed May  2 20:08:32 UTC 2018 on ttyAMA0
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.14.16-00004-ga5a4b5d aarch64)
```

Only a partial terminal output has been shown above.

Use "root" as login & password

Execute the following commands:

```
echo 1000 > /proc/sys/vm/nr_hugepages
# child DPRC container for VM guest userspace
export DPRC=dprc.3
echo 1 > /sys/module/vfio/parameters/enable_unsafe_noiommu_mode
echo vfio-fsl-mc > /sys/bus/fsl-mc/devices/$DPRC/driver_override
echo $DPRC > /sys/bus/fsl-mc/drivers/vfio-fsl-mc/bind
```

Set the Start CPU core as per taskset (w.r.t host) to run DPDK app. This should be the first physical core which is assigned to VM. E.g if you are running two core VM and Core #2 and #3 are given to VM.

```
export HOST_START_CPU=2
```

Setup Hugepages

```
echo hugetlbfs /mnt/hugetlbfs hugetlbfs defaults,mode=0777 0 0 >> /etc/fstab
mkdir /mnt/hugetlbfs
mount /mnt/hugetlbfs
```

configure the host connection for SCP, ssh and file transfer

```
ifconfig eth1 192.168.2.1
```

9.2.6.3 Running DPDK applications with direct device assignments

All the DPAA2 based dpdk application will work in VM similar to the host.

If the dpdk example applications are not present, you can bring them via scp/tftp using eth1 interface.

NOTE

Using Core 0 for DPDK application can lead to non-deterministic behavior, including drop in performance. It is recommended that DPDK applicaton core mask values avoid using Core 0.

Refer some example test commands below:

```
#one core VM (core #0 for dpdk)
./l3fwd -c 0x1 -n 1 --log-level=bus.fslmc,8 -- -p 0x1 -P --config="(0,0,0)"
./l2fwd-crypto -c 0x1 -n 1 --log-level=bus.fslmc,8 -- -p 0x1 -q 1 --chain HASH_ONLY
--auth_algo sha2-256-hmac --auth_op GENERATE --auth_key_random_size 64
```

#two core VM (core #1 for DPDK)

```
./l3fwd -c 0x2 -n 1 -- -p 0x1 -P --config="(0,0,1)"
./l3fwd -c 0x2 -n 1 -- -p 0x3 -P --config="(0,0,1),(1,0,1)"
./testpmd -c 0x3 -n 1 -- -i --portmask=0x3 --nb-cores=1 --forward-mode=txonly
```

9.2.7 DPDK on Docker

9.2.7.1 Docker Overview

Docker provides an environment for a given image, over which any user space application can be executed. An image must contain/expose all the tools which are required to run any application.

For more information on Docker, see https://docs.docker.com/engine/userguide/.

9.2.7.2 DPAA1-Platform

9.2.7.2.1 Running the Docker Container

To execute Docker, make sure you have completed the following prerequisites:

1. The Docker daemon must be running. If not, follow the instructions given at the link below to execute the daemon.

   https://docs.docker.com/engine/docker-overview/

2. The Docker tool must be installed, which will be working as the client to run the Docker container.

Download the required image, which should be run as an environment. Use the command below to get generic prebuilt images:

```
docker pull <distribution>:<tag>
```

```
e.g. docker pull Ubuntu:latest /*for standard latest Ubuntu version*/
```

All downloaded images can be verified using the command below:

```
docker images
```

Once images are downloaded, the Docker container can be started using the steps below:

```
/*Below commands will execute a docker container named as docker0*/

/*Running docker container*/
docker run
--privileged /*It provides privilege to docker container to
                 access host completely*/
--interactive /*Docker container will be running state */

/*Exporting host environment variable to docker container*/
--env LD_LIBRARY_PATH=/usr/local/lib

/*User defined name to docker container */
--name=docker0 --hostname=docker0

/*container will be detached once it is launched and host prompt will
be available to use*/
```

```
--detach

/*Exporting host partitions to docker container*/
--volume=/usr:/usr
--volume=/sys:/sys
--volume=/dev:/dev ubuntu: latest

/*Attaching docker console*/
docker exec -it docker0 bash
```

9.2.7.2.2 Running the DPDK Application

Once docker is launched and connected, then execute the DPDK application by running the respective command. The command below is a sample to run DPDK L3FWD:

- export DPAA_DYNAMIC_DIST=1
- l3fwd -c 0x0C -n 1 – -p 0x30 --config="(4,0,2),(5,0,3)" -P

9.2.7.3 DPAA2-Platform

9.2.7.3.1 Traffic Multiplexer/De-Multiplexer

On the DPAA2 architecture, the MC provides various methods by which incoming traffic can be split of over the multiple DPNIs. The sections below provide more information.

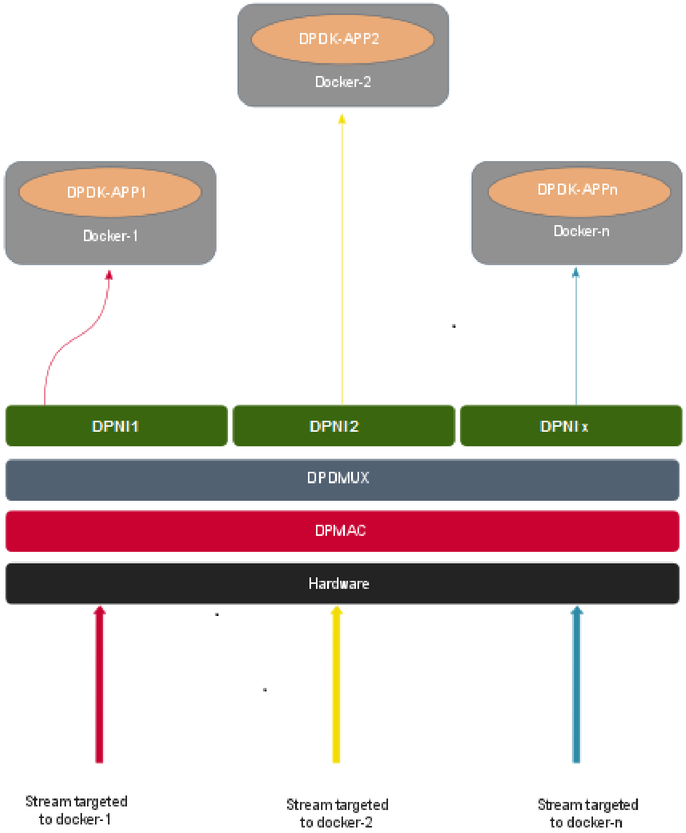
Using DPDMUX

MC provides an object (DPDMUX) which splits incoming traffic over the multiple DPNIs based on following parameters:

1. MAC based classification
2. VLAN based classification
3. MAC + VLAN base classification
4. User defined key based classification.

DPDMUX has its own filter table which consists of default filtering rules. Default filtering rules are a combination of MAC address configured on DPNI and port information as a destination. Once the DPDMUX object is connected to a given DPNI, then the entry for a particular DPNI will be added to the filtering table. All incoming default traffic will be distributed based on the destination MAC address in the packet. The user may add more entries to the filtering table as per his/her requirement.

The diagram below shows a sample use case for DPDMUX and associated links for a single DPMAC object. It can be extended up-to a maximum number of DPMACs, each having its own DPDMUX object.



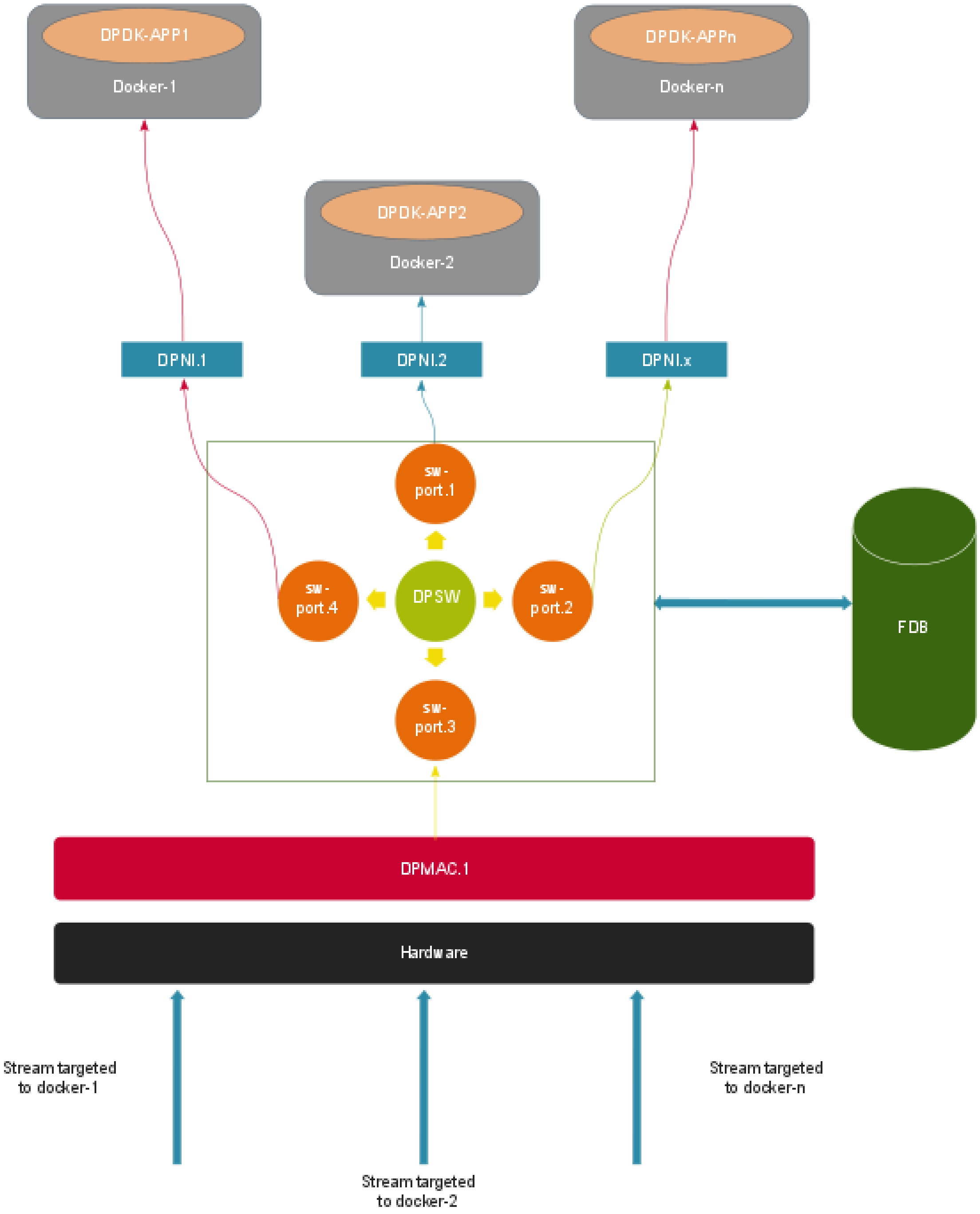
Using DPSW

MC also provides another object(DPSW) which internally implements DPAA2 H/W Switch. This Switch instance can also be used for traffic forwarding to multiple hosts. On LS2088 there is only once instance of DPSW that can be created and required ports will be connected to the same DPSW instance.

DPSW has its own filter table which populates dynamically with source MAC address and port which packet is received on. Default incoming traffic will be flooded to all ports except ingress port and filtering rules will be learnt into filtering table. After learning, same packet will be forwarded to the destined port only.

Below diagram shows a sample use case for DPSW and associated links for single DPMAC object. It can be extended up-to maximum number of DPMACs with same DPSW instance.

Linux user space



9.2.7.3.2 Docker's Resource Setup

9.2.7.3.2.1 Application Container Configuration

For each Docker instance, a DPRC needs to be created containing DPAA2 hardware blocks necessary for the Docker container.

A helper script `dynamic_dpl.sh`, part of the LSDK rootfs, can be used for creating such DPRC. For example, following command snippet creates a DPRC containing 8 DPNI objects (logical network interfaces) which are not backed by any physical link (DPMAC) and have MAC addresses starting from `00:00:00:00:05:00`. For more details about creating DPRC, refer Creating DPRCs on page 696

```
/usr/local/dpdk/dpaa2/dynamic_dpl.sh dpni dpni dpni dpni dpni dpni dpni dpni -b
00:00:00:00:05:00
```

The output of the above command would be similar to:

```
##################### Container  dprc.2  is created ####################

Container dprc.2 have following resources :=>

 * 16 DPBP
 * 8 DPCON
 * 8 DPSECI
 * 8 DPNI
 * 10 DPIO
 * 2 DPCI

 ######################### Configured Interfaces #########################

Interface Name        Endpoint              Mac Address
==============        ========              ==================
dpni.1                UNCONNECTED           00:00:00:00:05:01
dpni.2                UNCONNECTED           00:00:00:00:05:02
dpni.3                UNCONNECTED           00:00:00:00:05:03
dpni.4                UNCONNECTED           00:00:00:00:05:04
dpni.5                UNCONNECTED           00:00:00:00:05:05
dpni.6                UNCONNECTED           00:00:00:00:05:06
dpni.7                UNCONNECTED           00:00:00:00:05:07
dpni.8                UNCONNECTED           00:00:00:00:05:08
```

Each such DPRC would be assigned to a Docker container. Thus, multiple such DPRC would have to be created as per the use-case and Docker instances required for it.

NOTE

Resouces available on a DPAA2 system are limited and assigning them to DPRC can result error if requested resources are not available. For the above script output, if the script doesn't return any error and all the DPNIs have different MAC addresses, result can be considered success. In case of error or failure to assign MAC addresses, resource assignment to the DPRCs need to be restructured.

9.2.7.3.2.2 Kernel Container Configuration

The kernel has its defined root container (dprc.1). The user will be adding required traffic multiplexer/de-multiplexer objects into the kernel's container and connecting the application container's network interfaces for traffic distribution.

Configuration using DPDMUX

```
/*Create DPDMUX objects with total number of required links i.e. downlinks and uplinks both.
Here dpdmux.0 object is created*/

restool dpdmux create --num-ifs=3 --method DPDMUX_METHOD_MAC --max-dmat-entries=8 --max-mc-
groups=8 --manip=DPDMUX_MANIP_NONE

/*Connecting downlinks and uplinks with above created DPDMUX */
restool dprc connect dprc.1 --endpoint1=dpmac.x --endpoint2=dpdmux.0.0
restool dprc connect dprc.1 --endpoint1=dpni.y --endpoint2=dpdmux.0.1
restool dprc connect dprc.1 --endpoint1=dpni.z --endpoint2=dpdmux.0.2


/*Where x, y and z are object indices created in resource containers*/
```

Configuration using DPSW

```
/*Create DPSW object with total number of required links i.e. downlinks and uplinks both. Here
dpsw.0 object is created */
```

```
    restool dpsw create --num-ifs=3

/*Connecting downlinks and uplinks with above created DPSW */
    restool dprc connect dprc.1 --endpoint1=dpmac.x --endpoint2=dpsw.0.0
    restool dprc connect dprc.1 --endpoint1=dpni.y --endpoint2=dpsw.0.1
    restool dprc connect dprc.1 --endpoint1=dpni.z --endpoint2=dpsw.0.2

/*Where x, y and z are object indices which are created in resource containers*/
```

9.2.7.3.3 Running the Docker Container

To execute Docker, make sure you have completed the following prerequisites:

1. The Docker daemon must be running. If not, follow the instructions given at the link below to execute the daemon.

   https://docs.docker.com/engine/docker-overview/

2. The Docker tool must be installed, which will be working as the client to run the Docker container.

Download the required image, which should be run as an environment. Use the command below to get generic prebuilt images:

```
docker pull <distribution>:<tag>

e.g. docker pull Ubuntu:16.04 for standard Ubuntu version 16.04
```

All downloaded images can be verified using the command below:

```
docker images
```

Once images are downloaded, the Docker container can be started using the steps below:

```
/*Below commands will execute a docker container named as docker0*/

/*Exporting application resource container to docker tool*/
export DPRC="dprc.<index>"
export VFIO_NO=`readlink /sys/bus/fsl-mc/devices/$DPRC/iommu_group | xargs basename`

/*Running docker container*/
docker run
--privileged /*It provides privilege to docker container to
                 access host completely*/
--interactive /*Docker container will be running state */

/*Exporting host environment variable to docker container*/
--env DPRC=$DPRC --env LD_LIBRARY_PATH=/usr/local/lib

/*Exporting host devices to docker container*/
--device=/dev/vfio/vfio:/dev/vfio/vfio
--device=/dev/vfio/$VFIO_NO:/dev/vfio/$VFIO_NO

/*User defined name to docker container */
--name=docker0 --hostname=docker0

/*container will be detached once it is launched and host prompt will
be available to use*/
--detach

/*Exporting host partitions to docker container*/
--volume=/usr:/usr
--volume=/sys:/sys
--volume=/dev:/dev ubuntu:16.04

/*Attaching docker console*/
docker exec -it docker0 bash
```

9.2.7.3.4 Running the DPDK Application

Once docker is launched and connected, then execute the DPDK application by running the respective command. The command below is a sample to run DPDK L3FWD:

```
./l3fwd -c 0xFF -n 4 -- -p 0xFF -P --config="(0,0,0),(1,0,1),(2,0,2),(3,0,3),(4,0,4),(5,0,5),
(6,0,6),(7,0,7)" -P
```

9.2.7.3.5 Example Configuration: Using DPDMUX

On Host:

```
export MAX_QOS=16
export NXP_CHRT_PERF_MODE=1
export DPNI_NORMAL_BUF=1
```

Create container for docker0:

```
./dynamic_dpl.sh dpni dpni dpni dpni dpni dpni dpni dpni -b 00:00:00:00:05:00

##################### Container  dprc.2  is created ####################
Container dprc.2 have following resources :=>
 * 16 DPBP
 * 8 DPCON
 * 8 DPSECI
 * 8 DPNI
 * 10 DPIO
 * 2 DPCI
######################### Configured Interfaces #########################

Interface Name        Endpoint              Mac Address
==============        ========              ==================
dpni.1                UNCONNECTED           00:00:00:00:05:01
dpni.2                UNCONNECTED           00:00:00:00:05:02
dpni.3                UNCONNECTED           00:00:00:00:05:03
dpni.4                UNCONNECTED           00:00:00:00:05:04
dpni.5                UNCONNECTED           00:00:00:00:05:05
dpni.6                UNCONNECTED           00:00:00:00:05:06
dpni.7                UNCONNECTED           00:00:00:00:05:07
dpni.8                UNCONNECTED           00:00:00:00:05:08
```

Create container for docker1:

```
./dynamic_dpl.sh dpni dpni dpni dpni dpni dpni dpni dpni -b 00:00:00:00:05:08

##################### Container  dprc.3  is created ####################
Container dprc.3 have following resources :=>
 * 16 DPBP
 * 8 DPCON
 * 8 DPSECI
 * 8 DPNI
 * 10 DPIO
 * 2 DPCI
######################### Configured Interfaces #########################
Interface Name        Endpoint              Mac Address
==============        ========              ==================
dpni.9                UNCONNECTED           00:00:00:00:05:09
dpni.10               UNCONNECTED           00:00:00:00:05:0a
dpni.11               UNCONNECTED           00:00:00:00:05:0b
dpni.12               UNCONNECTED           00:00:00:00:05:0c
dpni.13               UNCONNECTED           00:00:00:00:05:0d
dpni.14               UNCONNECTED           00:00:00:00:05:0e
dpni.15               UNCONNECTED           00:00:00:00:05:0f
dpni.16               UNCONNECTED           00:00:00:00:05:10
```

Create DPDMUX objects with downlinks and uplinks

```
restool dpdmux create --num-ifs=2 --method DPDMUX_METHOD_MAC --max-dmat-entries=8 --max-mc-
groups=8 --manip=DPDMUX_MANIP_NONE
restool dpdmux create --num-ifs=2 --method DPDMUX_METHOD_MAC --max-dmat-entries=8 --max-mc-
groups=8 --manip=DPDMUX_MANIP_NONE
restool dpdmux create --num-ifs=2 --method DPDMUX_METHOD_MAC --max-dmat-entries=8 --max-mc-
groups=8 --manip=DPDMUX_MANIP_NONE
restool dpdmux create --num-ifs=2 --method DPDMUX_METHOD_MAC --max-dmat-entries=8 --max-mc-
groups=8 --manip=DPDMUX_MANIP_NONE
```

Create uplink connections

```
restool dprc connect dprc.1 --endpoint1=dpdmux.0.0 --endpoint2=dpmac.1
restool dprc connect dprc.1 --endpoint1=dpdmux.1.0 --endpoint2=dpmac.2
restool dprc connect dprc.1 --endpoint1=dpdmux.2.0 --endpoint2=dpmac.3
restool dprc connect dprc.1 --endpoint1=dpdmux.3.0 --endpoint2=dpmac.4
```

Create downlink connections for docker0

```
restool dprc connect dprc.1 --endpoint1=dpni.1 --endpoint2=dpdmux.0.1
restool dprc connect dprc.1 --endpoint1=dpni.2 --endpoint2=dpdmux.1.1
restool dprc connect dprc.1 --endpoint1=dpni.3 --endpoint2=dpdmux.2.1
restool dprc connect dprc.1 --endpoint1=dpni.4 --endpoint2=dpdmux.3.1
```

Create downlink connections for docker1

```
restool dprc connect dprc.1 --endpoint1=dpni.5 --endpoint2=dpdmux.0.2
restool dprc connect dprc.1 --endpoint1=dpni.6 --endpoint2=dpdmux.1.2
restool dprc connect dprc.1 --endpoint1=dpni.7 --endpoint2=dpdmux.2.2
restool dprc connect dprc.1 --endpoint1=dpni.8 --endpoint2=dpdmux.3.2
```

```
NOTE: The above commands are for 1G test. In case 10G port is to be used append the above
commands to create uplink and downlink
connections to append "--committed-rate=10000  --max-rate=10000"
```

Running DPDK L2fwd on docker0

```
export DPRC="dprc.2"
export VFIO_NO=`readlink /sys/bus/fsl-mc/devices/$DPRC/iommu_group | xargs basename`


docker run --privileged --interactive --env DPRC=$DPRC --env LD_LIBRARY_PATH=/usr/local/lib --
device=/dev/vfio/vfio:/dev/vfio/vfio --device=/dev/vfio/$VFIO_NO:/dev/vfio/$VFIO_NO --
name=docker0 --hostname=docker0 --detach --volume=/usr:/usr --volume=/sys:/sys --volume=/dev:/
dev ubuntu:18.04

docker exec -it docker0 bash

cd /usr/local/bin

./l2fwd -c 0xF0 -n 1 --file-prefix=docker0 --socket-mem=2048 -- -p 0x0F -q 1
```

Running DPDK L2fwd on docker1

```
export DPRC="dprc.3"
export VFIO_NO=`readlink /sys/bus/fsl-mc/devices/$DPRC/iommu_group | xargs basename`

docker run --privileged --interactive --env DPRC=$DPRC --env LD_LIBRARY_PATH=/usr/local/lib --
device=/dev/vfio/vfio:/dev/vfio/vfio --device=/dev/vfio/$VFIO_NO:/dev/vfio/$VFIO_NO --
name=docker1 --hostname=docker1 --detach --volume=/usr:/usr --volume=/sys:/sys --volume=/dev:/
dev ubuntu:18.04

docker exec -it docker1 bash

cd /usr/local/bin
./l2fwd -c 0xF0 -n 1 --file-prefix=docker1 --socket-mem=2048 -- -p 0x0F -q 1
```

NOTE

The above set of commands are for reference on LS2088A. On LS1088 DPDMUX object supports upto 4 downlinks (dpni's). These can be assigned to a docker instance as per requirement. For example, one usecase would assign two dpni's in each of the two docker container instances however other usecase would be to assign one dpni to each of four docker instances.

9.2.7.3.6 Example Configuration: Using DPSW

On Host

Common Container settings:

```
export MAX_QOS=16
export NXP_CHRT_PERF_MODE=1
export DPNI_NORMAL_BUF=1
```

Create container for docker0:

```
./dynamic_dpl.sh dpni -b 00:00:00:00:05:00

##################### Container  dprc.2  is created ####################

Container dprc.2 have following resources :=>
 * 16 DPBP
 * 8 DPCON
 * 8 DPSECI
 * 1 DPNI
 * 10 DPIO
 * 2 DPCI


######################### Configured Interfaces #########################

Interface Name        Endpoint              Mac Address
==============        ========              ==================
dpni.1                UNCONNECTED           00:00:00:00:05:01
```

Create container for docker1:

```
./dynamic_dpl.sh dpni -b 00:00:00:00:05:01

##################### Container  dprc.3  is created ####################

Container dprc.3 have following resources :=>

 * 16 DPBP
 * 8 DPCON
 * 8 DPSECI
 * 1 DPNI
 * 10 DPIO
 * 2 DPCI


######################### Configured Interfaces #########################

Interface Name        Endpoint              Mac Address
==============        ========              ==================
dpni.2                UNCONNECTED           00:00:00:00:05:02
```

Create DPSW objects

```
restool dpsw create --num-ifs=3
    restool dprc connect dprc.1 --endpoint1=dpsw.0.0 --endpoint2=dpmac.1
```

Create downlink connections for docker0

```
restool dprc connect dprc.1 --endpoint1=dpni.1 --endpoint2=dpsw.0.1
```

Create downlink connections for docker1

```
restool dprc connect dprc.1 --endpoint1=dpni.2 --endpoint2=dpsw.0.2
```

Running DPDK L2fwd on docker0

```
export DPRC="dprc.2"
export VFIO_NO=`readlink /sys/bus/fsl-mc/devices/$DPRC/iommu_group | xargs basename`

docker run --privileged --interactive --env DPRC=$DPRC --env LD_LIBRARY_PATH=/usr/local/lib --
device=/dev/vfio/vfio:/dev/vfio/vfio --device=/dev/vfio/$VFIO_NO:/dev/vfio/$VFIO_NO --
name=docker0 --hostname=docker0 --detach --volume=/usr:/usr --volume=/sys:/sys --volume=/dev:/
dev ubuntu:18.04

docker exec -it docker0 bash
```

```
cd /usr/local/bin
./l2fwd -c 0x04 -n 1 --file-prefix=docker0 --socket-mem=2048 -- -p 0x01 -q 1
```

Running DPDK L2fwd on docker1

```
export DPRC="dprc.3"
export VFIO_NO=`readlink /sys/bus/fsl-mc/devices/$DPRC/iommu_group | xargs basename`


docker run --privileged --interactive --env DPRC=$DPRC --env LD_LIBRARY_PATH=/usr/local/lib --
device=/dev/vfio/vfio:/dev/vfio/vfio --device=/dev/vfio/$VFIO_NO:/dev/vfio/$VFIO_NO --
name=docker1 --hostname=docker1 --detach --volume=/usr:/usr --volume=/sys:/sys --volume=/dev:/
dev ubuntu:18.04

docker exec -it docker1 bash

cd /usr/local/bin
./l2fwd -c 0x08 -n 1 --file-prefix=docker1 --socket-mem=2048 -- -p 0x01 -q 1
```

NOTE

The above commands are for LS2088A only as LS1088A doesn't support DPSW object.

9.2.8 Known Limitations and Future Work

Generic Limitations:

1. The hardware internally uses the hardware access portals for each thread doing packet I/O, which limits the number of I/O threads thereby impacting the performance. The number of available portals is different for DPAA and DPAA2.
2. Not all functionalities supported by DPDK framework have been implemented by PPFE, DPAA and DPAA2 drivers (PMDs). For list of supported features, refer PPFE: Supported DPDK Features, DPAA: Supported DPDK Features on page 878 and DPAA2: Supported DPDK Features on page 879.
3. Using Core 0 for I/O related work is known to impact performance - whether on host or in VM. Disabling services can result in normal performance but the results are non-deterministic. Affining Core 0 to I/O should be avoided as much as possible.
4. It has been observed that PCI NIC card events can lead to performance drop on certain platforms. The behavior is non-deterministic across platforms. For peak performance numbers, PCI NIC cards should be disabled.
5. DPDK docker support is currently only available for DPAA2 and DPAA platforms.

PPFE Specific Limitations:

1. While using PPFE in user space, if the kernel mode pfe module is loaded before using the user space mode, the HIF rings do not get cleaned sometimes and user need to restart the application again till the rings are cleaned (DPDK-1373).
2. SG is not supported. It will be supported in next release.
3. Link update for PPFE interfaces is not supported. It will be supported in next release.
4. Multiple buffer pools are not supported.
5. Some of the features are listed in DPDK-1375 which are currently not supported.

DPAA Specific Limitations:

1. Ports assigned to user space cannot be assigned dynamically to kernel space or vice-versa.
2. The FMAN configuration script cannot be re-run without system restart. i.e. number of queues configured cannot be changed.
3. If the system is configured with FMAN configuration script once, it will not work with non-FMC mode.
4. When running the default config as (no DPAA_FMC_MODE), the system take upto 5 second in clean up the resources.

9.2.9 Troubleshooting

Following are some common steps and suggestions outlined for best performance from DPDK Applications:

1. To obtain best performance, please ensure that the boot-up time command line arguments are similar to below:

   For DPAA2:

   ```
   console=ttyS1,115200 root=/dev/mmcblk0p3 earlycon=uart8250,mmio,0x21c0600
   default_hugepagesz=1024m hugepagesz=1024m hugepages=8 isolcpus=1-7
   ```

   For DPAA:

   ```
   console=ttyS0,115200 root=/dev/mmcblk0p3 earlycon=uart8250,mmio,0x21c0500
   default_hugepagesz=2m hugepagesz=2m hugepages=448 isolcpus=1-3 bportals=s0 qportals=s0
   ```

   `isolcpus` in the above ensures that only Linux Kernel schedules its threads on Core 0 only. Core 1-x would be used for DPDK application threads.

   Hugepage count defined by `hugepages` should also be modified to maximum possible so as to allow DPDK applications to have larger buffers.

   NOTE

   The value of hugepages is dependent on the size of RAM available on the board. Value should be selected based on specific use-case as any memory allocated for hugepage is not usable for Linux Kernel OS operations.

2. If there is issue with reception of transmission of packets, verify the following points:

   a. Ensure that no error has been reported by DPDK application at startup. Generally the output is descriptive enough for cause of problem.

   b. Check the mapping of ports against the physical ports:

      - In case of DPAA platform, ensure that the mapping of physical interfaces with DPDK ports is correct. Refer LS1043ARDB Port Layout or LS1046ARDB Port Layout.
      - In case of DPAA2 platform, ensure that correct `dpni.X` has been used in the `dynamic_dpl.sh` script while creating the `dprc` containers. A common pitfall is to use an incorrect `dpni` as against the physical port being used for IO.

   c. Ensure that traffic generator to board connectivity is proper. You may run `testpmd` in `tx_only` mode to validate if the packets are going out on specific interfaces. For information about `testpmd` application and its supported arguments, refer the web documentation.

   d. Ensure that the traffic generator stream settings are correct and enough streams are being generated for proper distribution between DPDK application cores.

   e. Ensure that the MAC address of stream generated by traffic generator matches that of the `dpni` port, or the interface is in promiscous mode.

3. If the performance is not as expected:

   a. Ensure that the stream configuration of the traffic generator is approrpriate and that it can generate multiple streams. In case the streams have all same IP destination and/or source, the distribution of traffic across multiple cores wouldn't happen.

      NOTE

      For obtaining best performance, it is important to configure the number of streams from packet generator adequately. If the number of streams generated by packet generator are not adequate, it would lead to improper distribution across the queues defined (especially in case of multiple queue setup) and eventually lack of performance.

   b. Using standard process tools in Linux, for example `ps`, `top`, verify that all the DPDK application threads have been started (as per application configuration on command line) and busy looping.

4. DPAA2 - the default mode for packet rx is prefetch mode, where it is expected that the rx API is receiving equal number of buffers in each call. However, if you want different number of buffers in each call, please do "export DPAA2_NO_PREFETCH_RX=1"
5. For DPAA2 Platform certain tuning parameters are available. User can enable them according to the requirements.
   - To offload the RX error packet drop (parsing error) handling in hardware. Set,

     ```
     #export DPAA2_PARSE_ERR_DROP=1
     ```
   - To disable the TX congestion control - i.e. infinite size of TX queues, set:

     ```
     #export DPAA2_TX_CGR_OFF=1
     ```
   - To configure the TQ queue congestion control - taildrop size in byte (default is 64K bytes), set:

     ```
     #export DPAA2_TX_TAILDROP_SIZE=<size>
     ```
6. System tuning parameters can be checked with "debug_dump.sh" script located in "/usr/local/dpdk" directory. You can share the output with support team for further analysis.
7. DPAA2 status can be checked from restool commands. (e.g. restool dpni info dpni.1)
8. DPAA2 - Application hangs during initialization when number of buffers are very large (more than 1 Million) Problem occurs due to maximum limit of number of buffers in QBMAN. It is configurable in dpc.0x2A_0x41.dts. To configure number of buffers, following node needs to be added if node is not present otherwise add only entry marked.controllers.

   ```
   {
   qbman  {
    ....
   total_bman_buffers=<0x1B7790>;
    };
    };
   ```

9.2.10 DPDK Performance Reproducibility Guide

This chapter describes various cases and points which are important for obtaining best performance from DPDK software on the NXP platforms. This is a suggestive list of best practices and optimal configurations which can help extract maximum performance of the NXP DPAA hardware.

NOTE

The practices mentioned in this chapter are based on tests in controlled environment. These are not intended for production or deployment without adequate analysis of the impact on use-cases.

This document is divided into two broad sections: Steps required before booting up the Linux Kernel and steps required before DPDK application execution.

Before booting up Linux

1. **Choosing Optimal Board Support Packages (BSP)**
   - Choosing a compatible board support package is critical for functionality as well as performance of DPDK application.

     For DPAA and DPAA2 platforms, select the top frequency RCW/PBL binaries stably supported by boards. For example, for LS2088ARDB DPAA2, Rev 1.1 boards with frequency of **2100x800x2133** is known to perform best. Other frequency, though stable, would result in slower performance. Below table describes an indicative set of known BSP files for DPDK supported SoC.
2. **Disabling hardware prefetching through u-boot**
   - For LS2088A DPAA2 platform, it is possible to disable hardware prefetching through u-boot. This can enhance performance in multicore scenario.
   - For disabling hardware prefetching, following command should be used on u-boot prompt:

     ```
     setenv hwconfig 'fsl_ddr:bank_intlv=auto;core_prefetch:disable=0xFE'
     ```

After executing the above command, board bank needs to be reset for the setting to take place. In the above command, field `disable=0xFE` defines the mask for disabling prefetching on specific cores. For example, for disabling prefetching on 3rd and 4th core, use `disable=0x0C`.

Also, it should be noted that disabling prefetching on Core 0 is not supported.

NOTE

This setting doesn't have impact on single core case. Maximum performance gain is observed when all 8 cores of LS2088 board are being used (of which 7 cores have prefetching disabled as Core 0 doesn't support this feature).

3. **Linux Boot Arguments**

   - For DPAA platform, if the on-board memory is limited (e.g. LS1043 RDB), following configuration should be appended to default boot arguments:

     ```
     default_hugepagesz=2m hugepagesz=2m hugepages=448 isolcpus=1-3 bportals=s0 qportals=s0
     ```

     Through the above boot arguments, 896 Mb of hugepages have been assigned for all DPDK applications (448 pages of 2M size each).

     `isolcpus` isolates the CPUs 1, 2, 3 from Linux Kernel process schedulers' scheduling algorithm. All System Service would be scheduled on Core 0 and that should be avoided in application configuration for I/O threads.

   - For DPAA2 platform, following configuration should be appended to default boot arguments:

     ```
     default_hugepagesz=1024m hugepagesz=1024m hugepages=8 isolcpus=1-7
     ```

     It is recommended to use 1G huge page size for DPAA2 platform.

   - In case UEFI based booting is used, the boot agruments are changed from grub.cfg. Please refer to UEFI section on how to update the arguments.

   NOTE

   It should be noted that CPU isolation configuration cannot be changed in a running Linux Kernel. Whereas, huge page configuration can be changed from Linux prompt by writing to `/proc/sys/vm/nr_hugepages` file. Thus, CPU isolation should be carefully decided before booting up Linux Kernel.

   NOTE

   `nousb` can be appended to boot arguments to disable USB in Linux Kernel. This prevents any interrupts from USB devices to be serviced by CPU cores. This is especially important when Core 0 is being used for DPDK I/O performance. But, this option should only be used if there is no dependency of USB devices for system execution, for example, a USB mass storage which contains either the root filesystem or extra filesystem containing data necessary for execution.

4. For Best performance, use the data cores as isolated cpus and operate them in tickless mode on kernel version 4.4 above. For this:

   a. Compile the Kernel with `CONFIG_NO_HZ_FULL=y`

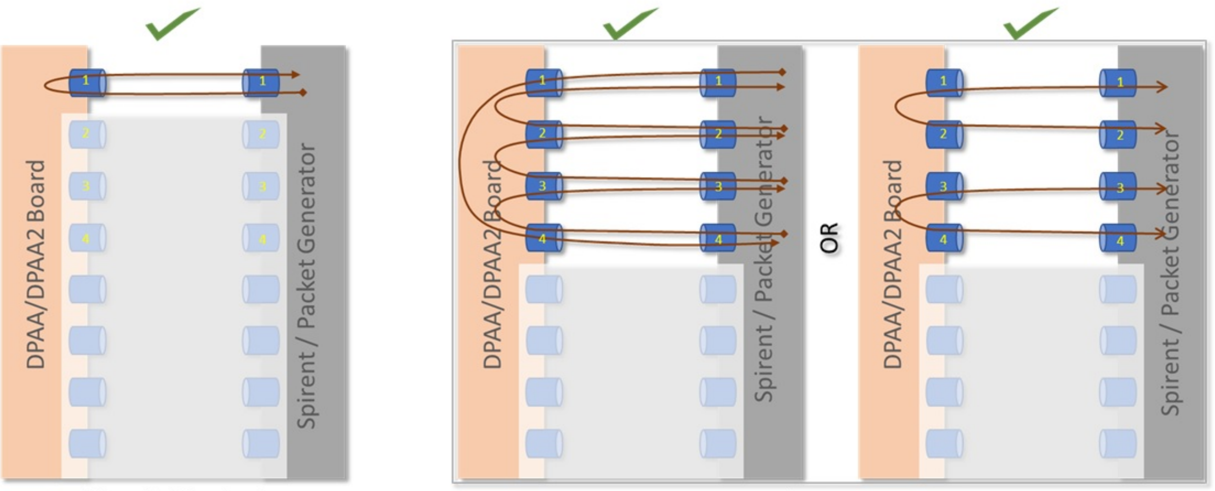
   b. Add bootargs with `'isolcpus=1-7 rcu_nocbs=1-7 nohz_full=1-7'` for 8 core platform and `'isolcpus=1-3 rcu_nocbs=1-3 nohz_full=1-3'` for 4 core platform

   NOTE

   The CONFIG_NO_HZ_FULL linux kernel build option is used to configure a tickless kernel. The idea is to configure certain processor cores to operate in tickless mode and these cores do not receive any periodic interrupts. These cores will run dedicated tasks (and no other tasks will be schedules on such cores obviating the need to send a scheduling tick). A CONFIG_HZ based timer interrupt will invalidate L1 cache on the core and this can degrade dataplane performance by a few % points (to be quantified, but estimated to be 1-3%). Running tickless typically means getting 1 timer interrupt/sec instead of 1000/sec

5. **Setup of the Performance Validation Environment**

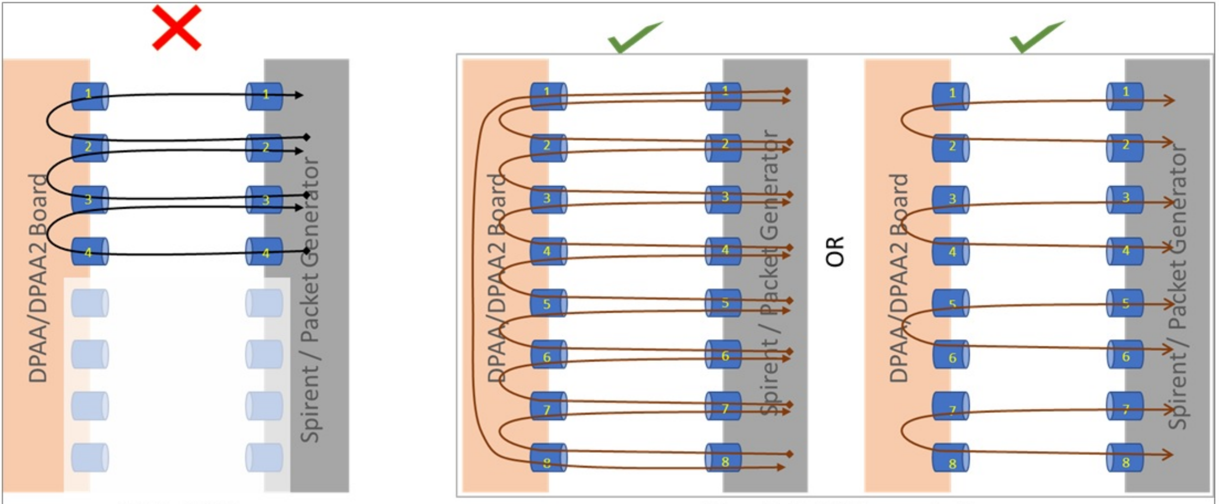
- It is important that the environment for performance verification uses a balanced core loading approach. Each core should be loaded with equal number of Rx/Tx queues, irrespective of their count. Images below describe some of the I/O scenario using an example setup containing a target board and a packet generator. In all the cases shown, it is assumed that each port has a single queue being serviced by a CPU core. Also, even though below images show 8 ports, it is a generic representation. DPAA boards may not have 8 equal ports (1G/10G) - this representation is assuming traffic is always distributed across equal capacity ports.



1 Port; Unidirectional

4 Ports; Each GPU Core serving 2 queues (1 RX, 1 TX);

Image above describes 2 cases: One for single port and another for 4 ports. It can be noted that all the cores are equally loaded (equal number of cores, irrespective numbers of ports being serviced). Further, the 4 port case shows that there is more than one way to move stream of packets. (Note the direction of arrows in each case.)



4 Core, 4 Port;
Core 2 & 3 serving 2 queues each;
Core 1 & 4 serving 1 queue each.

8 Ports; Each GPU Core serving 2 queues (1 RX, 1 TX);

Image above describes a case with 4 ports where the CPU cores are not equally loaded. This is not a recommended combination as this would mean some streams being served (packet per second) slower than others. 8 port combination shown in the image above extends the mapping of 4 ports shown in image before. Once again, it should be noted that there are multiple ways to create a balanced set of streams. A performance setup should choose one baseline and all performance reports should be based on that baseline.

NOTE

For Performance measurement, performing I/O across non-equal capacity ports (1G=>10G, vice-verse) is not a valid case. This would lead to build up of queues on higher capacity links eventually stopping traffic when hardware is unable to obtain buffers for storing new incoming packets - eventually stopping traffic.

6. **Uninstalling PCI Ethernet (e1000) NIC Cards**
   - It has been observed that when PCI Ethernet card (for example, on DPAA/DPAA2 RDB boards Intel e1000) are installed, they have a tendency to poll frequently the CPU cores (Core 0, in case of isolation). This has adverse impact on the application performance if DPDK I/O threads are scheduled on same cores which services these interrupts.
   - For best performance, such PCI Ethernet cards should be uninstalled from the hardware. If un-installation is not possible, see the comments mentioned in section below to disable the interface by unlinking it from the Linux Kernel.

NOTE

`nopci` can be appended to boot arguments to completely disable PCI devices from being detected by Linux Kernel. This prevents PCI interrupts from being serviced by CPU. But, this option should not be used if there is dependency on any PCI device for system execution.

Before and during DPDK Application start

1. **Setting real-time priority for DPDK Application**
   - In full fledged distributions, like Ubuntu, the root filesystem contains various system services by default. These services are targeted towards a generic environment. Many of these services require periodic CPU cycles. DPDK I/O threads execute as a run-to-completion process, infinitely looping over CPUs they are affined to. Services which require periodic CPU cycles can interrupt the DPDK I/O threads causing loss of packets and/or latency. Ideally, such services should be disabled or a rootfs without such services should be used for optimal performance. But, in case this cannot be done, real-time priority of application can also achieve desired results.
   - Execute the script `/usr/local/dpdk/enable_performance_mode.sh`. Care should be taken to run the DPDK application from same shell as the one on which script was executed. This is because the script sets some environment variables which are used by DPDK application to define real-time priorities for its threads. This script is also designed to set to "performance" mode the CPU scaling governor. This prevents the CPU from putting itself into lower power state when not busy. This causes loss of traffic in initial I/O streams when the CPU is expected to spin upto its maximum frequency.

NOTE

This script sets the real-time priorities for any DPDK application which is run after the script has been executed. This also applies to application configured to run on Core 0. Thus, it is important to consider the implication. If the application is run on Core 0 and it is busy in I/O, it can lead to CPU stall causing complete lock-up. DPDK sample applications like `l2fwd, l3fwd, ipsec-secgw` are designed to relinquish the CPU when no I/O is being done. That way, using sample application, all core performance can be calculated. Similar care should be taken while developing custom DPDK applications.

NOTE

Though this script doesn't necessarily require core isoluation and tickless kernel, it is still recommended that I/O cores be isolated and tickless kernel be used to get the best performance environment. Also, this script assumes that it is a Ubuntu environment with power governor support and that no other process is running in priority higher than DPDK application.

NOTE

An opposite script, `/usr/local/dpdk/disable_performance_mode.sh`, is also available. This puts the processor back in the "on-demand" scaling governor configuration and also removed the environment variables. It is important to run this script once performance verification of a DPDK sample application has been completed. This would avoid issues with inadvertently executing DPDK application on Core 0 and causing a lock-up.

2. **Using High Performance (PEB) Buffer (Only for DPAA2)**

- In DPAA2 platform, while creating the resource container using the `dynamic_dpl.sh` script, it is possible to toggle between high performance PEB buffers and normal buffers (DDR). By default, the high performance buffers are enabled for LS2088A; for LS1088A, default configuration is normal buffers.

  NOTE

  For LS2088A, it is recommended to use high performance buffers which is enabled by default. Though, there is caveat to this as described below.

  PEB buffers are limited resources. Overusage of buffers, either through large number of queues or deep taildrop settings, can cause the PEB buffers to overflow causing a interruption of I/O. The hardware might also enter a state from which it will not recover until board is restarted.

  Exact limitations of number of queues is based on various parameters and cannot be stated objectively without defining the use-case. As a thumb-rule, refrain from using PEB buffers if configuration requires more than 1 queue per CPU core to be used, assuming all ports and CPU cores are being employed.

  For toggling between normal and high performance buffers, use the following environment variable **before** executing the `dynamic_dpl.sh` script:

  ```
  export DPNI_NORMAL_BUF=1
  # disables high performance buffers; enables normal buffers
  ```

3. **Disabling PCI Ethernet (e1000) NICs**

   - As mentioned in the above section, it is preferable if no PCI Ethernet hardware (like e1000 on DPAA/DPAA2 boards) is installed. But, if it is not possible to uninstall a hardware device, following command can be used to unlink the Ethernet card from PCI driver in the Linux Kernel thereby preventing the CPU cores from being interrupted with periodic interrupts. This is specially important when all core performance is to be recorded.

     ```
     echo 1 > /sys/bus/pci/devices/<PCI device BDF address>/remove
     ```

     In the above command, replace `<PCI device BDF address>` with appropriate BDF format bus address of the PCI device, for example `0000:01:00.0`, after properly bypassing the `:` character in the name to avoid failure reported by Linux Bash prompt. For example, `echo 1 > /sys/bus/pci/devices/0000\:01\:00.0/remove`.

     This command would unlink the PCI device with BDF address `0000:01:00.0` from its PCI driver's control, thereby disabling it from Linux Kernel.

     NOTE

     Once the device is unlined from the PCI driver, it would not be usable through the Linux Kernel interface until bound to same or another PCI driver. It is out of scope for this document to record steps necessary for linking a PCI device to a PCI driver to bring it under Linux Kernel control.

4. **DPDK Optimal Example Application Configuration**

   - Avoiding Core 0
     - As mentioned above, distributions like Ubuntu have large number of system services. Though some of these services can be disabled, there would always be cases of interrupts or un-interruptible services which would require Core 0 cycles. Isolating the cores through Linux Kernel can be done using Linux boot arguments. This would allow isolated cores to be used exclusively for DPDK I/O threads.
     - Once a configuration of isolated cores is set, similar configuration should be done in DPDK application using the `-c` or `--coremask` command line option.
     - If 4 core (in LS1043A or LS1046A) or 8 core (LS1088A or LS2088A) performance is required, system services should be disabled. Though, it should be noted that performance number using Core 0 show un-deterministic behavior of latency and packet losses. For example, LS2088A has been observed to perform fairly stable on 8 core configuration with services disabled, but same cannot be stated for LS1088A boards.
   - Avoiding Core 0 in case of Virtual Machine

  - Core 0 impact on the DPDK I/O performance is valid for host as well as for Virtual Machine (VM). While configuring DPDK application in VM, Core 0 should be avoided. The Qemu configuration should be such as to avoid using the Host's Core 0 for any VM logical core which is running DPDK I/O threads.
  - For a VM environment, OVS or similar switching stack maybe used on the host. Qemu configuration should be such as to avoid mapping the logical cores (VCPU) assigned to VM with any of the CPU cores which run the switching stack threads. `taskset` command is recommended for affining the Qemu threads (serving VM VCPUs) to a particular core. Refer Launch QEMU and virtual machine for more details.
- Using Multi-queue configuration to spread load across multiple CPUs
  - DPDK applications can utilize RSS based spreading of incoming frames across multiple queues servicing a particular port. This is especially helpful in obtaining better performance by utilizing 1:N mapping of ports to CPU cores. That is, more than 1 CPU core serves a single port.

    This requires adequate configuration of Port-Queue-Core combination through DPDK application command line. For example, `l3fwd` application can be configured to use 8 ports on a LS2088A board for serving 2 ports using the following command:

    ```
    l3fwd -c 0xFF -n 1 -- -p 0x3 --config="(0,0,0),(0,1,1),(0,2,2),(0,3,3),(1,0,4),
    (1,1,5),(1,2,6),(1,3,7)"
    ```

    In the above command, the `--config` argument takes multiple tuples of *(port, queue, core)*. Note that Port number 0 is being served by Core 0, 1, 2 and 3 using separate queue numbers.

    Using similar configuration described for `l2fwd` application above, optimal utilization of Cores can be achieved. The command line options vary with DPDK application and DPDK online web manual should be referred for specific example applications.

    Though the above command snippet utilizes Core 0, necessary care should be taken as described in text above.
  - As mentioned above, DPDK uses RSS (Receive Side Scaling) to spread the incoming frames across multiple queues. Multi-queue setup needs to be supported by varying flows from the Packet Generator. The flows created should be such as to have varying Layer-2 or Layer-3 field values.
    - As flow distribution is based on hash over Layer-2 and Layer-3 fields, it is possible that lower number of flows would distribute unevenly across queues. Number of flows created should be large enough to spread equally across all the configured queues.
- Consideration for CPU clusters
  - SoC have multiple clusters housing one or more CPUs. Each cluster shares a L2 cache. In general, this allows threads sharing data over CPUs from same cluster to perform better than threads sharing data across CPUs from different clusters.
  - For best performance, it is recommended that DPDK application configuration for selecting CPU cores should be such to either use all CPUs from same cluster or spread queues equally across clusters. When this is combined with Core 0 issue, it implies that using Cluster having Core 0 might perform slightly worse than using cluster which doesn't use Core 0.
- Using limited number of I/O buffers
  - DPDK allows an application to change the number of maximum in-flight buffers. This is especially useful when there is memory constraint and DPDK application has limited resources.
  - Each buffer, for processing, has to be fetched into the system caches (L2/L1). Larger the number of buffers in-flight simultaneously, more would be the flushing of buffer addresses. To avoid excessive pressure on the L2 caches (eviction, hit, miss cycle), lower number of buffers should be used. Exact numbers would depend on the use-case and resources available.

    For example, in case of `l3fwd` application, `--socket-mem=1025` like EAL argument can be provided to the application as shown in command snippet below. Note that the argument has been provided before the `--` - these are passed to DPDK framework rather than the application itself.

```
./l3fwd -c 0xFF -n 1 --socket-mem=1025 -- -p 0x1 --config="(0,0,0),(0,1,1),
(0,2,2),(0,3,3),(0,4,4),(0,5,5),(0,6,6),(0,7,7)"
```

- Degradation of OVS performance with increase in flows
  - It has been observed that OVS doesn't perform well when the number of flows are large. This is because of OVS's inherent design to use a flow matching table of size 8000. If larger than 8000 flows are used, the overall performance degrades because of hash collisions. If more than 8000 flows are required, use the following command **after** OVS bridge has been created:

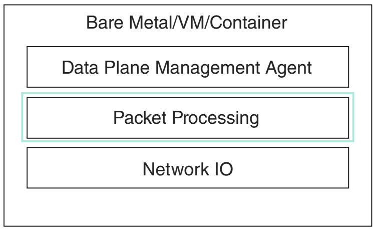
    ```
    ovs-vsctl set bridge br0 other-config:flow-eviction-threshold=65535
    ```

    This command would set the size of OVS internal flow table to 65535.
- Use `-n 1` as argument passed to DPDK EAL
  - `-n` argument for DPDK application is for defining number of DDR channels for the system - which is typically valid for NUMA architectures. This parameter is used for mempool memory alignments. For NXP SoCs, this should be set to "`1`". NXP SoCs supported by DPDK are non-NUMA.

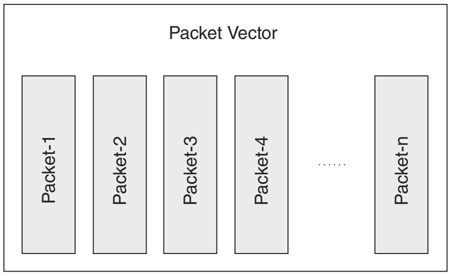
9.3 Vector Packet Processing (VPP)

9.3.1 Introduction

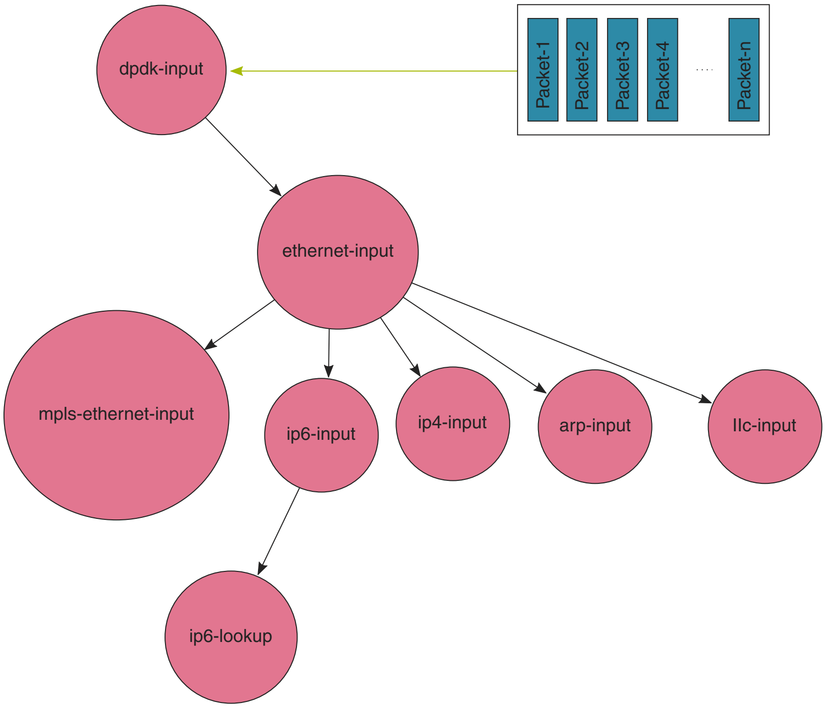
The Vector Packet Processing (VPP) platform is an extensible framework that provides out-of-the-box production quality switch/ router functionality. It is the open source version of VPP technology: a high performance, packet-processing stack that can run on commodity CPUs. The benefits of this implementation of VPP are its high performance, proven technology, its modularity and flexibility, and rich feature set. It is a modular design. The framework allows anyone to "plug in" new graph nodes without the need to change core/kernel code.



VPP reads the largest available vector of packets from the network IO layer.



VPP then processes the vector of packets through a Packet Processing graph.

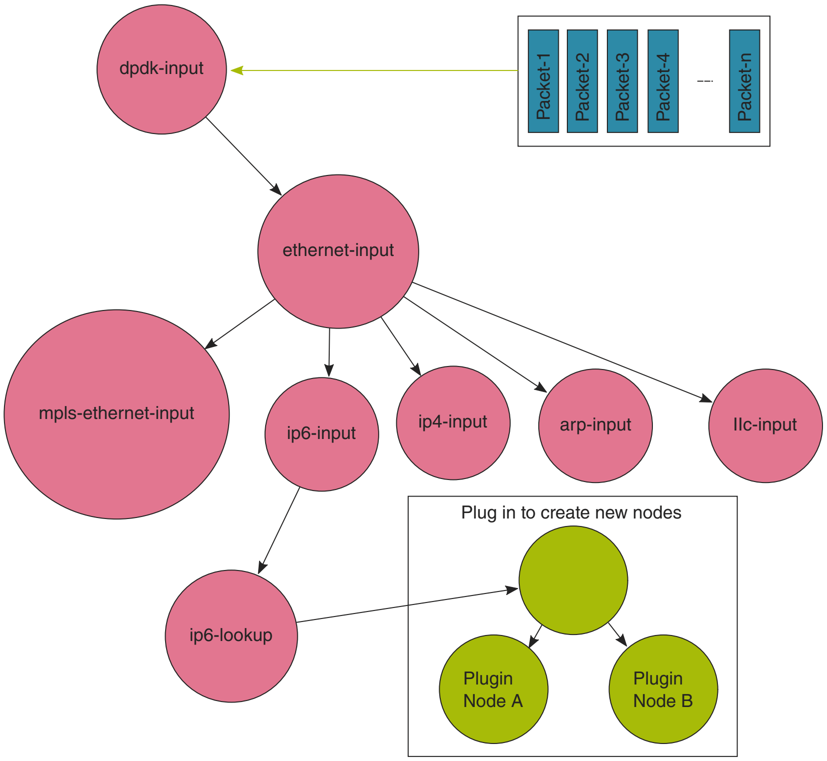


Rather than processing the first packet through the whole graph, and then the second packet through the whole graph, VPP instead processes the entire vector of packets through a graph node before moving on to the next graph node.

Because the first packet in the vector warms up the instruction cache, the remaining packets tend to be processed at extreme performance. The fixed costs of processing the vector of packets are amortized across the entire vector. This leads not only to very high performance, but also statistically reliable performance. If VPP falls a little behind, the next vector contains more packets, and thus the fixed costs are amortized over a larger number of packets, bringing down the average processing cost per packet, causing the system to catch up. As a result, throughput and latency are very stable. If multiple cores are available, the graph scheduler can schedule (vector, graph node) pairs to different cores.

The graph node architecture of VPP also makes for easy extensibility. You can build an independent binary plugin for VPP from a separate source code base (you need only the headers). Plugins are loaded from the plugin directory. A plugin for VPP can rearrange the packet graph and introduce new graph nodes. This allows new features to be introduced via the plugin, without needing to change the core infrastructure code.

Linux user space



For more details see, https://wiki.fd.io/view/VPP.

9.3.2 Supported platform

VPP supports LS1043A, LS1046A, and LS2088A family of SoCs. This section details the architectural and port layout of their Reference Design Boards.

Refer to the following for board specific information:

- LS1043A Reference Design Board
- LS1046A Reference Design Board
- LS2088A Reference Design Board

9.3.3 Supported usecases

- vRouter: VPP as virtual router
- vSwitch: VPP as virtual switch
- IPsec: VPP can perform the IPsec in *protocol offload* and *non-protocol offload* modes. DPDK supports 2 crypto drivers that VPP can use for IPsec:
    - — OpenSSL Crypto driver
    - — DPAA Sec driver

NOTE

Only **DPAA sec** driver supports the protocol offload feature.

9.3.4 Build VPP

Standalone Build steps:

This section details steps required to build VPP in a standalone environment.

Prerequisites before compiling VPP:

Before compiling VPP as a standalone build, following dependencies need to be resolved independently:

1. DPDK libraries required for packets processing. Please refer section “ Standalone build of DPDK Libraries and Applications” for DPDK compilation. Please note that It is required to add “EXTRA_CFLAGS='-fPIC'” in DPDK compilation command to make DPDK libraries compatible with VPP.
2. OpenSSL libraries.

VPP Compilation:

- git clone https://source.codeaurora.org/external/qoriq/qoriq-components/vpp
- git checkout -b 18.09-LSDK 18.09-LSDK
- export DPDK_PATH=<DPDK installed path>
- export CROSS_PREFIX=aarch64-linux-gnu
- export ARCH=arm64
- export PATH=<toolchain path>/bin:<toolchain path>/aarch64-linux-gnu/bin:$PATH
- export OPENSSL_PATH=<openssl path>
- cd vpp
- make install-dep
- cd build-root
- make distclean
- ./bootstrap
- make V=0 PLATFORM=dpaa TAG=dpaa install-deb -j8

After compilation, there will be some "deb" packages generated in vpp/build-root directory. please copy all ".deb" packages to the "/usr/local/vpp" directory in rootfs.

9.3.5 Executing VPP

Installing VPP

Following are the steps for installing VPP:

- cd /usr/local/vpp
- dpkg -i *.deb
- export LD_LIBRARY_PATH=/usr/lib64

Running VPP

Following are steps for running VPP:

- **LS2088 Board setup:**
  - cd /usr/local/dpdk/dpaa2
  - . ./dynamic_dpl.sh dpmac.1 dpmac.2

  - mkdir /mnt/hugepages
  - mount -t hugetlbfs none /mnt/hugepages
  - echo 256 > /proc/sys/vm/nr_hugepages
- **LS1046 and LS1043 Board setup:**
  - mkdir /mnt/hugepages
  - mount -t hugetlbfs none /mnt/hugepages
  - echo 256 > /proc/sys/vm/nr_hugepages

NOTE: <int0> and <int1> in the following commands are the interfaces names and must be replaced with the actual interfaces names. Please run the command “vppctl show int” after running “vpp -c /etc/vpp/startup.conf &” to check the interfaces names. To check the associated MAC address of an interface please run the command “vppctl show hard”

VPP Commands to execute on LS1043, LS1046 and LS2088 boards:

VPP cross-connect:

- vpp -c /etc/vpp/startup.conf &
- vppctl set interface state <int0> up
- vppctl set interface state <int1> up
- vppctl set interface l2 xconnect <int0> <int1>
- vppctl set interface l2 xconnect <int1> <int0>

VPP v-router:

- vpp -c /etc/vpp/startup.conf &
- vppctl
- set interface ip address <int0> 1.1.1.2/24
- set interface ip address <int1> 2.1.1.2/24
- set interface state <int0> up
- set interface state <int1> up
- set interface mtu 1500 <int0>
- set interface mtu 1500 <int1>

VPP IPsec:

VPP can perform the IPsec using 2 crypto drivers named “OpenSSL” and “DPAA Sec (QI based)”. By default, VPP will use the “DPAA Sec”.

To run the VPP with “OpenSSL”, please follow the below steps:

- Edit the config file “/etc/vpp/startup.conf” and add the below lines under “’DPDK” section:
  - vdev crypto_openssl0
  - vdev crypto_openssl1
- For LS1046/LS1043 Board**,** please run the below command before running VPP:
  - export DPAA_SEC_DISABLE=1
- For LS2088 board, please run the below command before running the dynamic_dpl.sh:
  - export DPSECI_COUNT=0

Commands to run on Board1:

- vpp -c /etc/vpp/startup.conf &

- vppctl
- set interface ip address <int0> 1.1.1.2/24
- set interface ip address <int1> 192.168.100.2/24
- set interface state <int0> up
- set interface state <int1> up
- set ip arp static <int1> 192.168.100.3 00:22:22:22:22:23
- ipsec sa add 10 spi 1001 esp crypto-alg aes-cbc-128 crypto-key 4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key 4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.2 tunnel-dst 192.168.100.3
- ipsec sa add 11 spi 1002 esp crypto-alg aes-cbc-128 crypto-key 4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key 4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.3 tunnel-dst 192.168.100.2
- ipsec spd add 1
- set interface ipsec spd <int1> 1
- set interface promiscuous on <int1>
- ipsec policy add spd 1 priority 10 outbound action protect sa 10 local-ip-range 1.1.1.3 - 1.1.1.3 remote-ip-range 2.1.1.3 - 2.1.1.3
- ipsec policy add spd 1 priority 10 inbound action protect sa 11 local-ip-range 1.1.1.3 - 1.1.1.3 remote-ip-range 2.1.1.3 - 2.1.1.3
- ip route add count 1 2.1.1.3/32 via 192.168.100.3 <int1>
- set ip arp static <int0> 1.1.1.3 00:22:22:22:22:28
- ipsec policy add spd 1 priority 100 inbound action bypass protocol 50
- ipsec policy add spd 1 priority 100 outbound action bypass protocol 50

Commands to run on board2:

- vpp -c /etc/vpp/startup.conf &
- vppctl
- set interface ip address <int0> 2.1.1.2/24
- set interface ip address <int1> 192.168.100.3/24
- set interface state <int0> up
- set interface state <int1> up
- set ip arp static <int1> 192.168.100.2 00:22:22:22:22:24
- ipsec sa add 20 spi 1001 esp crypto-alg aes-cbc-128 crypto-key 4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key 4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.2 tunnel-dst 192.168.100.3
- ipsec sa add 21 spi 1002 esp crypto-alg aes-cbc-128 crypto-key 4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key 4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.3 tunnel-dst 192.168.100.2
- ipsec spd add 1
- set interface ipsec spd <int1> 1
- set interface promiscuous on <int1>
- ipsec policy add spd 1 priority 10 inbound action protect sa 20 local-ip-range 2.1.1.3 - 2.1.1.3 remote-ip-range 1.1.1.3 - 1.1.1.3
- ipsec policy add spd 1 priority 10 outbound action protect sa 21 local-ip-range 2.1.1.3 - 2.1.1.3 remote-ip-range 1.1.1.3 - 1.1.1.3
- ip route add count 1 1.1.1.3/32 via 192.168.100.2 <int1>
- set ip arp static <int0> 2.1.1.3 00:22:22:22:22:25
- ipsec policy add spd 1 priority 100 inbound action bypass protocol 50

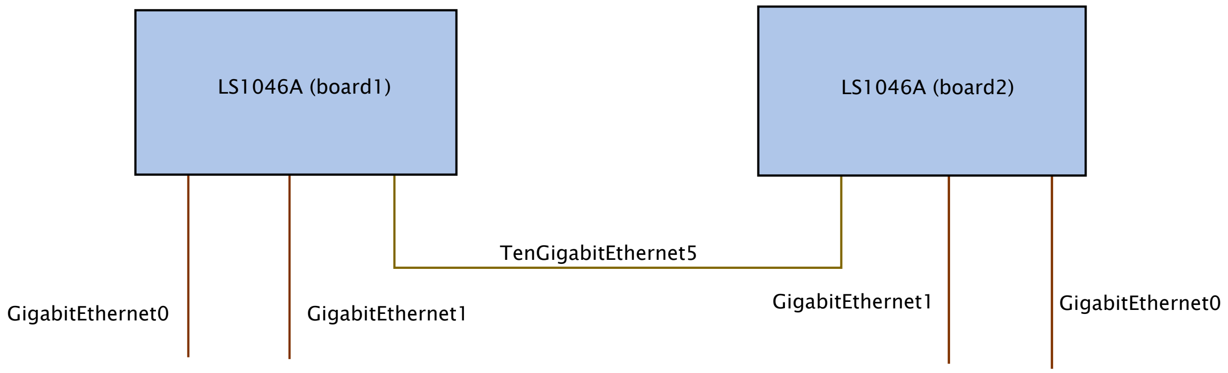
- ipsec policy add spd 1 priority 100 outbound action bypass protocol 50

9.3.6 Known limitations

- In DPAA1 platform, number of queues per ethernet interface can only be in multiple of 2.
- Sometimes changing IP address of an interface causes error and new IP address don't get assigned. (DPDK-1408)

9.3.7 VPP performance reproducibility guide

IPsec Performance reproducibility on LS1046A/LS1043A boards: This setup will use two 1g ports and one 10g port of both boards. 1g ports should be connected to the Spirent whereas 10g ports of both boards should be connected back to back.



Follow the below instructions and execute commands to reproduce the IPsec numbers on LS1046A/LS1043A board.

1. Edit the /etc/vpp/startup.conf file and update the below arguments:
    a. Under cpu section: “workers 3”
    b. Under dpdk section:
        - “num-rx-queues 2”
        - uncomment the “rss { ipv4 }”
2. Execute the commands below on LS1046A and LS1043A:

```
vpp -c /etc/vpp/startup.conf &
vppctl
```

3. Run the below set of commands on Board1 (LS1046A):

```
set interface ip address GigabitEthernet0 1.1.1.2/24
set interface ip address GigabitEthernet1 10.1.1.2/24
set interface ip address TenGigabitEthernet5 192.168.100.2/24
set interface state GigabitEthernet0 up
set interface state GigabitEthernet1 up
set interface state TenGigabitEthernet5 up
ipsec sa add 10 spi 1001 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.2 tunnel-dst
192.168.100.3
ipsec sa add 11 spi 1002 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.3 tunnel-dst
192.168.100.2
```

```
ipsec spd add 1
set interface ipsec spd TenGigabitEthernet5 1
set interface promiscuous on TenGigabitEthernet5
ipsec policy add spd 1 priority 10 outbound action protect sa 10 local-ip-range 1.1.1.3
- 1.1.1.3 remote-ip-range 2.1.1.3 - 2.1.1.3
ipsec policy add spd 1 priority 10 inbound action protect sa 11 local-ip-range 1.1.1.3
- 1.1.1.3 remote-ip-range 2.1.1.3 - 2.1.1.3
ip route add count 1 2.1.1.3/32 via 192.168.100.3 TenGigabitEthernet5
set ip arp static TenGigabitEthernet5 192.168.100.3 00:22:22:22:22:23
set ip arp static GigabitEthernet0 1.1.1.3 00:22:22:22:22:28
ipsec policy add spd 1 priority 100 inbound action bypass protocol 50
ipsec policy add spd 1 priority 100 outbound action bypass protocol 50
ipsec sa add 40 spi 2001 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.4 tunnel-dst
192.168.100.5
ipsec sa add 41 spi 2002 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.5 tunnel-dst
192.168.100.4
ipsec policy add spd 1 priority 10 outbound action protect sa 40 local-ip-range 1.1.1.3
- 1.1.1.3 remote-ip-range 2.1.1.4 - 2.1.1.4
ipsec policy add spd 1 priority 10 inbound action protect sa 41 local-ip-range 1.1.1.3
- 1.1.1.3 remote-ip-range 2.1.1.4 - 2.1.1.4
ip route add count 1 2.1.1.4/32 via 192.168.100.3 TenGigabitEthernet5
set ip arp static TenGigabitEthernet5 192.168.100.5 00:22:22:22:22:33
set ip arp static GigabitEthernet0 1.1.1.4 00:22:22:22:22:38
ipsec sa add 50 spi 3001 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.2 tunnel-dst
192.168.100.3
ipsec sa add 51 spi 3002 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.3 tunnel-dst
192.168.100.2
ipsec policy add spd 1 priority 10 outbound action protect sa 50 local-ip-range
10.1.1.3 - 10.1.1.3 remote-ip-range 20.1.1.3 - 20.1.1.3
ipsec policy add spd 1 priority 10 inbound action protect sa 51 local-ip-range 10.1.1.3
- 10.1.1.3 remote-ip-range 20.1.1.3 - 20.1.1.3
ip route add count 1 20.1.1.3/32 via 192.168.100.3 TenGigabitEthernet5
set ip arp static GigabitEthernet1 10.1.1.3 00:22:22:22:22:38
ipsec sa add 70 spi 4001 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.4 tunnel-dst
192.168.100.5
ipsec sa add 71 spi 4002 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.5 tunnel-dst
192.168.100.4
ipsec policy add spd 1 priority 10 outbound action protect sa 70 local-ip-range
10.1.1.3 - 10.1.1.3 remote-ip-range 20.1.1.4 - 20.1.1.4
ipsec policy add spd 1 priority 10 inbound action protect sa 71 local-ip-range 10.1.1.3
- 10.1.1.3 remote-ip-range 20.1.1.4 - 20.1.1.4
ip route add count 1 20.1.1.4/32 via 192.168.100.3 TenGigabitEthernet5
set ip arp static GigabitEthernet1 10.1.1.4 00:22:22:22:22:38
```

4. Run the below set of commands on board2

```
set interface ip address GigabitEthernet0 2.1.1.2/24
set interface ip address GigabitEthernet1 20.1.1.2/24
set interface ip address TenGigabitEthernet5 192.168.100.3/24
set interface state GigabitEthernet0 up
set interface state GigabitEthernet1 up
set interface state TenGigabitEthernet5 up
ipsec sa add 20 spi 1001 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.2 tunnel-dst
192.168.100.3
ipsec sa add 21 spi 1002 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
```

```
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.3 tunnel-dst
192.168.100.2
ipsec spd add 1
set interface ipsec spd TenGigabitEthernet5 1
set interface promiscuous on TenGigabitEthernet5
ipsec policy add spd 1 priority 10 inbound action protect sa 20 local-ip-range 2.1.1.3
- 2.1.1.3 remote-ip-range 1.1.1.3 - 1.1.1.3
ipsec policy add spd 1 priority 10 outbound action protect sa 21 local-ip-range 2.1.1.3
- 2.1.1.3 remote-ip-range 1.1.1.3 - 1.1.1.3
ip route add count 1 1.1.1.3/32 via 192.168.100.2 TenGigabitEthernet5
set ip arp static TenGigabitEthernet5 192.168.100.2 00:22:22:22:22:24
set ip arp static GigabitEthernet0 2.1.1.3 00:22:22:22:22:25
ipsec policy add spd 1 priority 100 inbound action bypass protocol 50
ipsec policy add spd 1 priority 100 outbound action bypass protocol 50
ipsec sa add 30 spi 2001 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.4 tunnel-dst
192.168.100.5
ipsec sa add 31 spi 2002 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.5 tunnel-dst
192.168.100.4
ipsec policy add spd 1 priority 10 inbound action protect sa 30 local-ip-range 2.1.1.3
- 2.1.1.3 remote-ip-range 1.1.1.4 - 1.1.1.4
ipsec policy add spd 1 priority 10 outbound action protect sa 31 local-ip-range 2.1.1.3
- 2.1.1.3 remote-ip-range 1.1.1.4 - 1.1.1.4
ip route add count 1 1.1.1.4/32 via 192.168.100.2 TenGigabitEthernet5
set ip arp static TenGigabitEthernet5 192.168.100.4 00:22:22:22:22:44
set ip arp static GigabitEthernet0 2.1.1.4 00:22:22:22:22:55
ipsec sa add 60 spi 3001 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.2 tunnel-dst
192.168.100.3
ipsec sa add 61 spi 3002 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.3 tunnel-dst
192.168.100.2
ipsec policy add spd 1 priority 10 inbound action protect sa 60 local-ip-range 20.1.1.3
- 20.1.1.3 remote-ip-range 10.1.1.3 - 10.1.1.3
ipsec policy add spd 1 priority 10 outbound action protect sa 61 local-ip-range
20.1.1.3 - 20.1.1.3 remote-ip-range 10.1.1.3 - 10.1.1.3
ip route add count 1 10.1.1.3/32 via 192.168.100.2 TenGigabitEthernet5
set ip arp static GigabitEthernet1 20.1.1.3 00:22:22:22:22:55
ipsec sa add 80 spi 4001 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.4 tunnel-dst
192.168.100.5
ipsec sa add 81 spi 4002 esp crypto-alg aes-cbc-128 crypto-key
4a506a794f574265564551694d653768 integ-alg sha1-96 integ-key
4339314b55523947594d6d3547666b45764e6a58 tunnel-src 192.168.100.5 tunnel-dst
192.168.100.4
ipsec policy add spd 1 priority 10 inbound action protect sa 80 local-ip-range 20.1.1.3
- 20.1.1.3 remote-ip-range 10.1.1.4 - 10.1.1.4
ipsec policy add spd 1 priority 10 outbound action protect sa 81 local-ip-range
20.1.1.3 - 20.1.1.3 remote-ip-range 10.1.1.4 - 10.1.1.4
ip route add count 1 10.1.1.4/32 via 192.168.100.2 TenGigabitEthernet5
set ip arp static GigabitEthernet1 20.1.1.4 00:22:22:22:22:55
```

5. Configure traffic from the Spirent ports as specified below:

Board1				**Board2**			
Port	**Flow**	**SRC-IP**	**DEST-IP**	**Port**	**Flow**	**SRC-IP**	**DEST-IP**
Gigabit Eth0	1	1.1.1.3	2.1.1.3	Gigabit Eth0	1	2.1.1.3	1.1.1.3

Table continues on the next page...

Table continued from the previous page...

Board1				Board2			
Port	**Flow**	**SRC-IP**	**DEST-IP**	**Port**	**Flow**	**SRC-IP**	**DEST-IP**
	2	1.1.1.3	2.1.1.4		2	2.1.1.3	1.1.1.4
Gigabit Eth1	1	10.1.1.3	20.1.1.3	Gigabit Eth1	1	20.1.1.3	10.1.1.3
	2	10.1.1.3	20.1.1.4		2	20.1.1.3	10.1.1.4

9.4 USDPAA

USDPAA is no longer supported as an API for direct customer use. All non-NXP software should use one of the standard APIs, DPDK instead of USDPAA. Some of the USDPAA software components may still exist as a layer below other software components such as DPDK, but do not assume that this will continue in future software releases.

Chapter 10 Virtualization

10.1 KVM/QEMU

10.1.1 KVM/QEMU Overview

This document is a guide and tutorial to building and using KVM (Kernel-based Virtual Machine) on NXP QorIQ SoCs.

Virtualization provides an environment that enables running multiple operating systems on a single computer system. Virtualization uses hardware and software technologies together to enable this by providing an abstraction layer between system hardware and the OS. The isolated environment in which OSes run is known as a *virtual machine* (or VM). The abstraction layer that manages all this is referred to as a *hypervisor or virtual machine manager*. The hypervisor layer operates at a privilege level higher than that of the operating systems, thus enabling it to enforce system security, ensure that virtual machines cannot interfere with each other, and transparently provide other services such as I/O sharing to the VM.

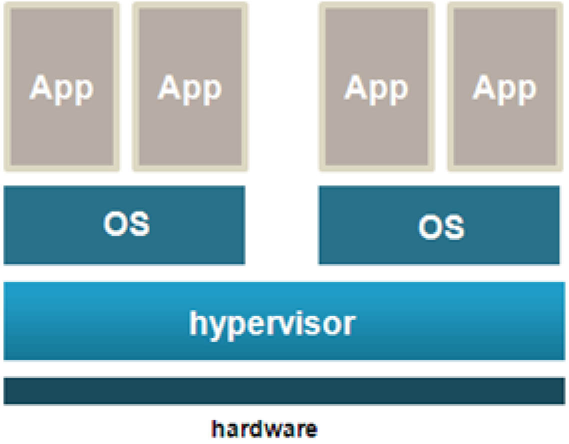


Figure 221.

KVM is a Linux kernel driver that together with QEMU, an open source machine emulator, provides an open source virtualization platfiorm based on Linux. KVM and QEMU together act as a virtual machine manager that can boot and run operating systems in virtual machines. See Figure below.

In this document the term *host* kernel refers to the underlying instance of Linux with the KVM driver that acts as the hypervisor. The term *guest* refers to the operating system, such as Linux, that runs in a virtual machine. A virtual machine will be referred to as a "VM".

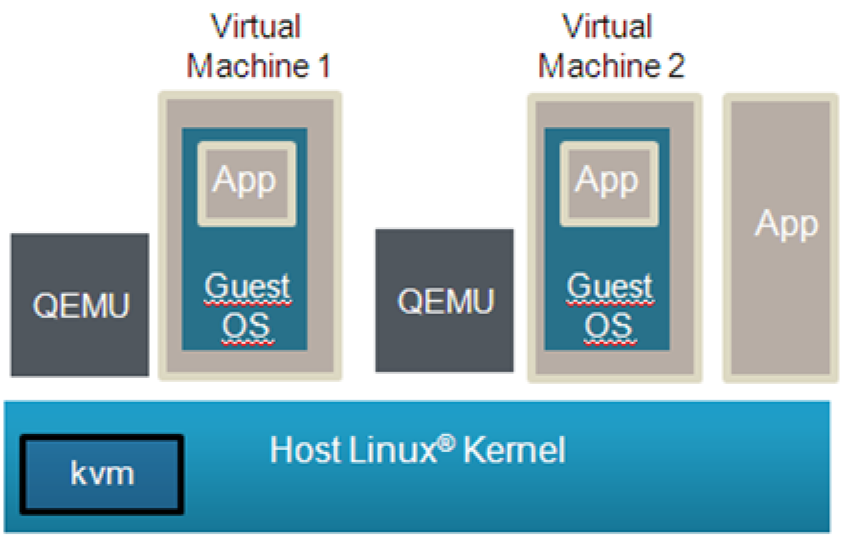


Figure 222.

NXP QorIQ SoCs based on ARM v7 and ARM v8 CPUs are supported.

10.1.1.1 Using QEMU and KVM

10.1.1.1.1 Overview of Using QEMU

QEMU is used to start virtual machines. The QEMU application is named **qemu-system-arm** (for 32 bit platforms) or **qemu-system-aarch64** (for 64 bit platforms).

In addition to the QEMU executable itself, the following is a list of the minimum components that must be available on the target system to launch a virtual machine using QEMU:

- The host Linux kernel on the target must be built with virtualization support for KVM enabled .
- A guest OS kernel image (e.g. zImage or Image for Linux)
- A guest root filesystem (If needed by the guest OS. For example, a Linux guest requires a rootfs.)
- Recommended: A working network interface (to interface to the guest's console and the QEMU monitor)

The QEMU Emulator User Documentation [1] (see References on page 951) contains complete documentation for all QEMU command line arguments. The Table below summarizes some of the flags and arguments for basic operation.

Table 165.

Argument	Descriptions
-enable-kvm	Specifies that the Linux KVM should be used for the virtual machine's CPUs
-nographic	Disables graphical output-console will be on emulated serial port.
-M *machine*	Specifies the type of virtual machine. One value is supported: • virt

Table continues on the next page...

Table 165. (continued)

Argument	Descriptions
-smp *cpu_count*	Specifies the number of CPUs for the virtual machine. The number of virtual CPUs allowed is the same as the value of the CONFIG_NR_CPUS config option in the host Linux kernel. To see this value issue the following command from Linux on the target board: `zcat /proc/config.gz | grep NR_CPUS`
-kernel *file*	Specifies the guest OS image. The supported image types are in *Image* format (the generic Linux kernel binary image file) and *zImage* (a compressed version of the Linux kernel image)
-initrd *file*	Specifies a root filesystem image
-append *cmdline*	Use *cmdline* as the guest OS kernel command line (passed in the bootargs property of the /chosen node in the guest device tree)
-serial *dev*	Redirects the virtual serial port to the host device *dev*. QEMU supports many possible host devices. Please refer to the QEMU User Documentation [1] (see References on page 951) for complete details. Note: if using a tcp device with the **server** option QEMU will wait for a connection to the device before continuing unless the **nowait** option is used.
-m *megs*	Specifies the size of the VM's RAM in megabytes. This option is ignored if using direct mapped memory. See Virtual Machine Memory on page 945 for further details on options for allocating memory.
-mem-path *path*	Specifies the path to a file from which to allocate memory for the virtual machine. This option should be used to allocate memory from hugetlbfs. See Virtual Machine Memory on page 945 for further details on options for allocating memory.
-monitor *dev*	Redirects the QEMU monitor to the host device *dev*. QEMU supports many possible host devices. Please refer to the QEMU User Documentation [1] (see References on page 951) for complete details. Note: if using a tcp device with the **server** option QEMU will wait for a connection to the device before continuing unless the **nowait** option is used.
-S	Do not start CPU at startup (you must type 'c' in the monitor). This can be useful if debugging.
-gdb *dev*	Wait for gdb connection on device dev
-drive *[args]*	Used to create a virtual disk in a virtual machine. See Virtual block devices on page 946 for additional information.
-netdev *[args]* -device virtio-net-device *[args]*	The -netdev and -device virtio-net-device arguments specify the network backend and front end for createing virtual network devices in virtual machines. See Virtual network interfaces on page 945 for additional information.

Table continues on the next page...

Table 165. (continued)

Argument	Descriptions
-cpu *model*	Select CPU model. Only one model is supported: • host

Below is an example command line a user would run from the host Linux to start virt virtual machine booting a Linux guest:

```
qemu-system-aarch64 -smp 8 -m 1024 -cpu host -machine type=virt,gic-version=3 -kernel /boot/
Image -enable-kvm -display none -serial tcp::4446,server,telnet -drive
if=none,file=ubuntu_bionic_arm64_rootfs.ext4.img,id=foo,format=raw -device virtio-blk-
device,drive=foo -append 'root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk' -monitor
stdio
```

10.1.1.1.2 Virtual Machine Memory

QEMU allocates and loads images into a VM's memory prior to starting the VM. The amount of memory needed for a virtual machine will be dependent on the workload to be run in the VM. There are two ways to allocate memory:

1. Allocation via hugetlbfs

   Hugetlbfs is a Linux mechanism that allows applications to allocate memory backed large physically contiguous regions of memory. QEMU can take advantage of hugetlbfs for allocation of memory for virtual machines, which can provide a significant performance improvement over malloc allocated memory. Hugetlbfs allocated memory provides the flexibility of memory that can be allocated and freed with performance comparable to direct mapped memory.

   The **-mem-path** argument to QEMU specifies the path to the hugetlbfs mount point where the huge pages should be allocated from.

   The **-m** argument to QEMU specifies the amount of memory to allocate to the virtual machine. There are no constraints on the size passed to this argument other than that the amount of memory must fit within the constraints of the system and be enough for the workload in the VM.

   See the how-to article Quick-start Steps to Run KVM Using Hugetlbfs on page 959 for an example of how to use hugetlbfs.

2. Allocation via malloc

   The default for QEMU is to allocate guest memory by the standard malloc facility available to user space applications in Linux. The amount of memory is specified with the -m command line argument. Malloc'ed memory has the flexibility of being allocated and freed by QEMU as needed. However, malloc'ed memory is backed by 4KB physical pages that are not contiguous and emulation is required by KVM to present a contiguous guest physical memory region to the VM. This approach is discouraged since the emulation can result in a substantial performance penalty for certain workloads.

The guest device tree generated by QEMU will contain a memory node that specifies the total amount of memory.

NOTE

A virtual machine's memory is part of the address space of the QEMU process. This means that the amount of memory allocated to a VM is limited by the standard limits that exist for Linux processes. A 32-bit host kernel has a 2GiB virtual address space used for stack, text, and other data, and this limits the amount of memory that can be allocated to a VM.

10.1.1.1.3 Virtual network interfaces

QEMU provides a number of options for creating virtual network interfaces in virtual machines. Virtual network interfaces are specified using the QEMU command line and guest software sees them as memory mapped devices.

There are two aspects of virtual network interfaces with QEMU:

1. The network “front-end”, which is the network card as seen by the guest. This is specified with the **-device** QEMU argument. The argument to specify a virtio network front end would look like: **-device virtio-net-pci**

2. The network "backend", which connects the network card to some network. Network backend options include user mode networking, a host TAP interface, sockets, or virtual distributed Ethernet. The network backend is specified using the **-netdev** command line argument of QEMU. Note: It is possible to connect two virtual machines using virtual network interfaces. Normally QEMU userspace process emulates I/O accesses from the guest. However, there is an in-kernel implementation: *vhost-net* which puts the data plane emulation code into the kernel.

For example, to use a virtio NIC card with a TAP interface back-end the QEMU command line argument would look like:

```
-netdev tap,id=tap0,script=/root/qemu-ifup -device virtio-net-pci,netdev=tap0
```

The script “/root/qemu-ifup” is a script that QEMU invokes and passes the TAP interface name as an argument. For example, the script could add the TAP interface to an Ethernet bridge.

See the QEMU Users Manual [1] (see References on page 951) for detailed information about command line options and the types of network interfaces and backends. For best performance, the virtio front-end is recommended.

For additional information about QEMU networking see the references in For More Information on page 952.

For a detailed example, see the how-to article How to Use Virtual Network Interfaces Using Virtio on page 960 .

10.1.1.1.4 Virtual block devices

There are a number of approaches to provide a virtual disk to a KVM/QEMU virtual machine. A guest disk image can be a single raw file on the host filesystem, a file in a virtual disk format such as qcow2 and vdi, or a block device on the host Linux system. The virtual disk is assigned on the QEMU command line. In the example below, the file **my_guest_disk** is a disk image and is assigned to the VM when QEMU is launched: `-drive file=my_guest_disk,cache=none,if=virtio`

Refer to the QEMU Users manual [1] (see References on page 951) for details on the types of virtual disk images that may be created and the related arguments to QEMU.

QEMU allows for various storing caching attributes to be set for the guest. The cache option is specified with `cache=` property. The following options are supported:

- `writethrough`: The host page cache is used, but the data is written to the physical device. This mode ensures data integrety.
- `writeback`: This is the default mode (when the cache property is missing). The host page cache is used, the normal page cache management will handle the write to the storage device.
- `none`: The host page cache is bypassed, the guests writes go directly to the storage device. The storage device may have a write cache.
- `directsync`: The host page cache is bypassed and the data is written to the physical device.
- `unsafe`: The flush commands to ensure the data integrity are ignored.

For a detailed example, see How to use **Virtual Disks Using Virtio**.

10.1.1.1.5 Direct assigned devices

VFIO - Virtual Function I/O

The VFIO is a Linux userspace driver infrastructure, an IOMMU/device agnostic framework for exposing direct device access from userspace. For the highest possible I/O performance, virtual machines make use of direct device access, called also *device assignment.* From a host and device perspective, the VFIO framework turns the virtual machines - QEMU - into a userspace driver, with the benefits of significantly reduced latency and direct use of device drivers.

The VFIO framework provides:

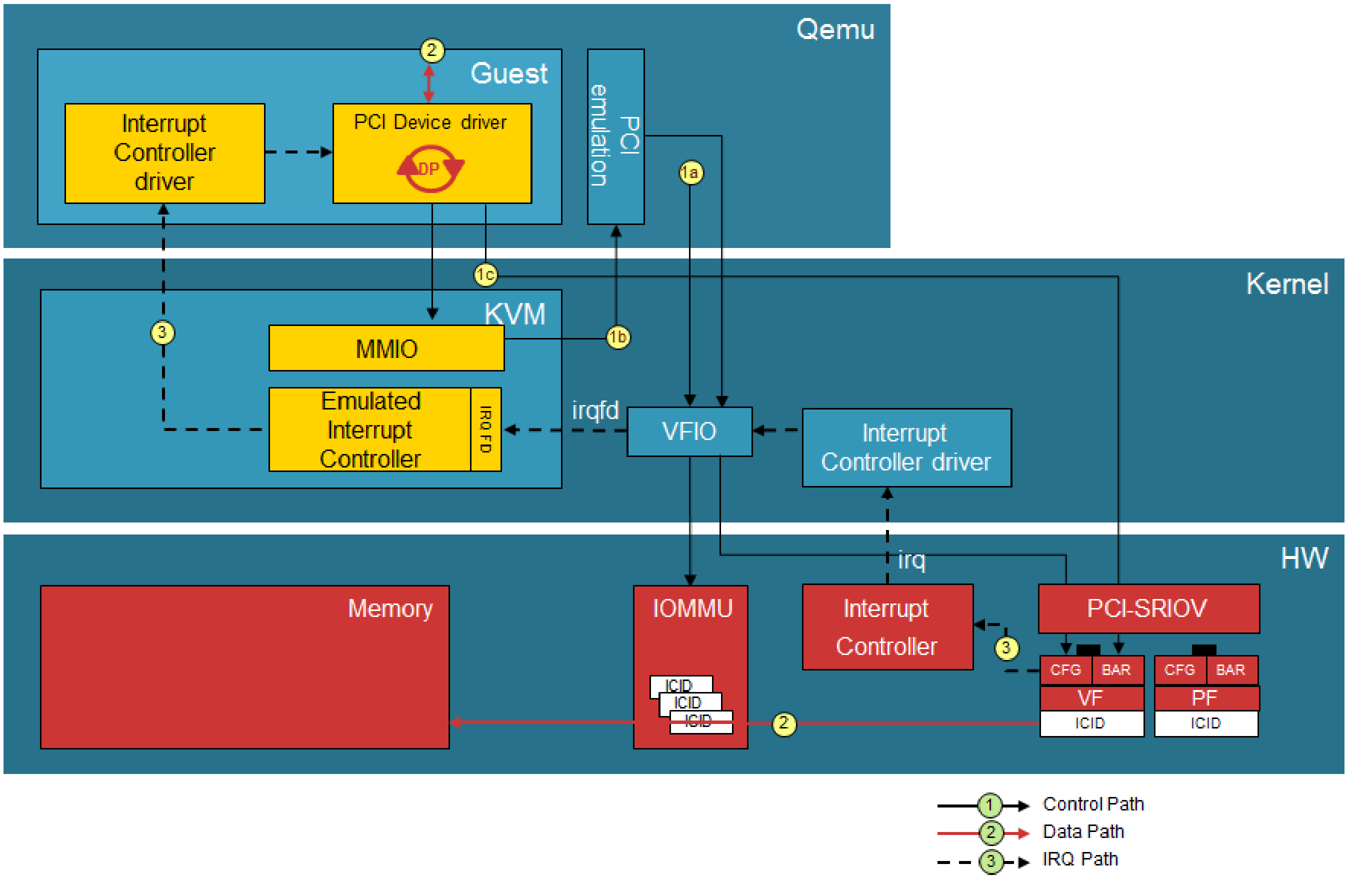
- device access
- IOMMU programming interface
- high performance interrupt support

Furthermore, the VFIO framework supports several bus infrastructures, such as PCI, platform devices and also the LS2 MC bus. In the following paragraphs both PCI and LS2 MC bus infrastructure support will be presented.

VFIO PCI

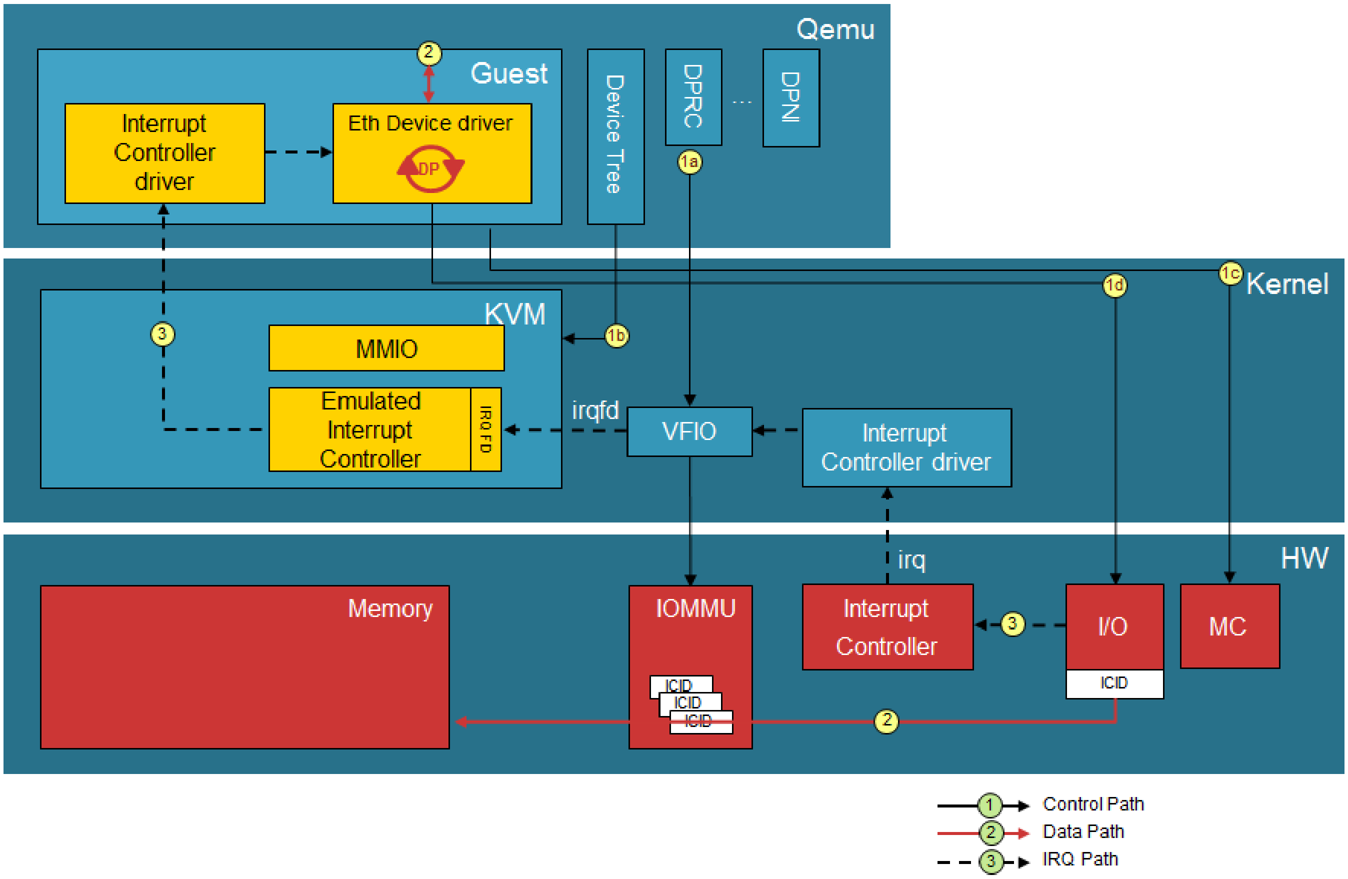
The VFIO driver abstracts PCI devices as regions and IRQs. The *regions* component includes the PCI configuration space, MMIO and I/O port BAR spaces and MMIO PCI ROM access, while the *IRQs* include INTx, legacy interrupts, but also Message Signaled Interrupts.

One can follow the Control path, Data path and IRQ path through a VFIO PCI infrastructure in the below image. Also, more information on how to use the PCI Direct Assignment feature can be found in the How to use PCIE direct assignment on page 971 chapter.



VFIO for LS2 MC Bus

The DPAA2 architecture works with the concept of MC containers - DPRCs. From the point of view of the OS, a DPRC behaves similar to a plug and play bus, like PCI. DPRC commands can be used to enumerate the contents of the DPRC and discover the hardware objects present (includin mappable regions and interrupts). The VFIO infrastructure for the FSL MC Bus can be found below.



The root container always belongs to the Linux host, while any child container can be assigned to user-space applications such as DPDK or virtual machines - QEMU. In the context of *direct device assignment*, this means that any DPAA2 object that needs to be made available to a guest VM should be places in a child container and, furthermore, the child container should be bound to the VFIO FSL MC driver. One can find more on how to use this feature in the How to use DPAA2 direct assignment without scripts on page 964 chapter.

10.1.1.1.6 VMs and the Linux Scheduler

Each virtual machine appears to the host Linux as a process with each virtual CPU in the VM implemented as a thread. A VM appears as an instance of QEMU when looking at Linux processes as can be seen in the example below:

```
$ ps -ef
                o
                o
root      1333     1  0 Oct01 ttyS0 00:00:00       -sh
root      1336     2  0 08:24 ?        00:00:00    [kworker/u4:2]
root      1372  1333 18 08:27 ttyS0    00:00:17    qemu-system-arm   -enable-kvm -m
root      1361  1304  0 08:28 ?        00:00:00    sshd: root@pts/0
root      1363  1361  0 08:28 pts/0    00:00:00    -sh
                o
                o
```

CPUs appear as threads. To see thread IDs use the `info cpus` command in the QEMU monitor. Example of a VM with 8 virtual CPUs:

```
(qemu) info cpus
* CPU #0: thread_id=1984
  CPU #1: (halted) thread_id=1985
  CPU #2: (halted) thread_id=1986
  CPU #3: (halted) thread_id=1987
  CPU #4: (halted) thread_id=1988
  CPU #5: (halted) thread_id=1989
  CPU #6: (halted) thread_id=1990
  CPU #7: (halted) thread_id=1991
```

To see the QEMU threads using the ps command:

```
root@ls_machine:~# ps -eL | grep qemu
 1981  1981 ttyS1    00:00:00 qemu-system-aar
 1981  1982 ttyS1    00:00:00 qemu-system-aar
 1981  1983 ttyS1    00:00:00 qemu-system-aar
 1981  1984 ttyS1    00:00:00 qemu-system-aar
 1981  1985 ttyS1    00:00:00 qemu-system-aar
 1981  1986 ttyS1    00:00:00 qemu-system-aar
 1981  1987 ttyS1    00:00:00 qemu-system-aar
 1981  1988 ttyS1    00:00:00 qemu-system-aar
 1981  1989 ttyS1    00:00:00 qemu-system-aar
 1981  1990 ttyS1    00:00:00 qemu-system-aar
 1981  1991 ttyS1    00:00:00 qemu-system-aar
```

Being a Linux thread means that standard Linux mechanisms can be used to control aspects of how the threads are scheduled relative to other threads/processes. These mechanisms include:

- process priority
- CPU affinity
- isolcpus
- cgroups

10.1.1.2 Virtual Machine Overview

A guest OS running in a KVM/QEMU virtual machine "sees" a hardware environment similar to running on a physical board. The guest sees CPUs, memory, and a number of I/O devices. Some aspects of this environment are virtualized (emulated in software by KVM/QEMU) but this virtualization is mostly transparent to the guest, and changes to the guest are typically not required to run in a virtual machine.

The number of virtual machines that can be run simultaneously is only limited by the amount of available resources (like any other application on Linux).

KVM/QEMU implements a generic virt machine which is described completely by the device tree. The virtual machine contains the following resources:

- one or more ARMv7/ARMv8 virtual CPUs
- memory
- virtual console based on an emulated PL011
- virtio over PCI (used for virtual devices such as block and network devices)
- ARM Virtual Generic Interrupt Controller
- ARM virtual timer and counter

10.1.1.3 Introduction to KVM and QEMU

QEMU (pronounced KYOO-em-yoo) is a software-based machine emulator that emulates a variety of CPUs and hardware systems. KVM is a Linux kernel device driver that provides virtual CPU services to QEMU. The two software components work together as a virtual machine manager.

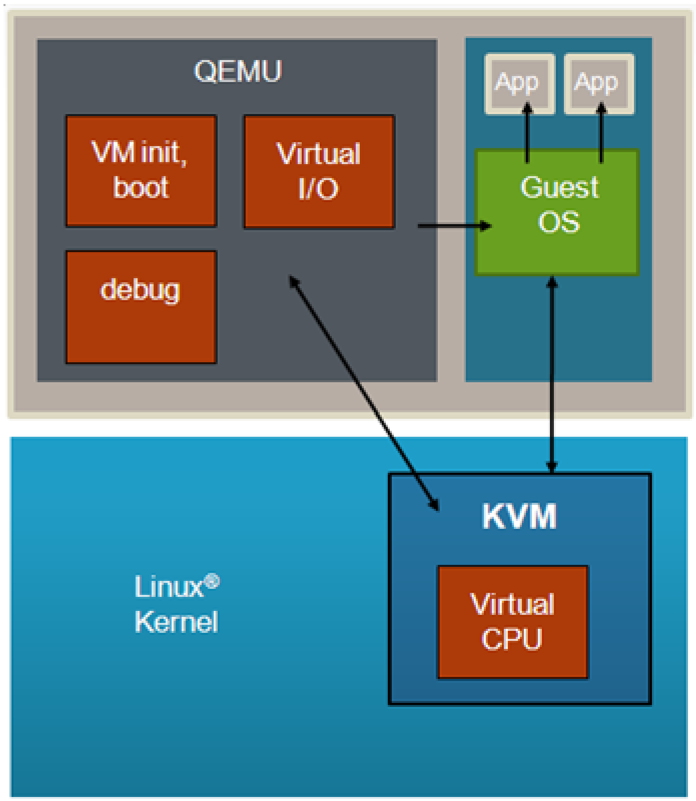


Figure 223.

QEMU is a Linux user-space application that runs on the host Linux instance and is used to start and manage a virtual machine. QEMU provides the following:

- A command line interface that provides extensive customization and configuration of a virtual machine when it is started--e.g. type of VM, which images to load, and how virtual devices are configured
- Loading of all images needed by the guest-- e.g kernel images, root filesystem, guest device tree
- Setting the initial state of the VM and booting the guest
- Virtual I/O services, such as virtual network interfaces and virtual disks
- Debug services-which provide the capability to debug a guest OS using GDB (similar to a virtual JTAG)

KVM is a device driver in the Linux kernel whose key role in the VM architecture is to provide virtual CPU services. These services involve two aspects:

1. First, KVM provides an API set that QEMU uses to set and get the state of virtual CPUs and run them. For example, QEMU sets the initial values of the CPU's registers before starting the VM.
2. Second, after KVM starts a guest OS, certain operations (such as privileged instructions) performed by the OS cause an exception (or exit) into the host Linux kernel that must be handled and processed by KVM. This handling of traps is referred to as "emulation". These traps are transparent to the guest.

The KVM API is documented in the Linux kernel-- Documentation/virtual/kvm/api.txt.

KVM/QEMU supports virtual I/O which allows sharing of physical I/O devices by multiple VMs. Virtual network and block I/O are supported. See For More Information on page 952 for references that provide additional information on virtio.

10.1.1.4 Device Tree Overview

A device tree is a data structure that describes hardware resources such as CPUs, memory, and I/O devices. An device tree aware OS is passed a device tree which it reads to determine what hardware resources are available.

The host Linux kernel is booted first by a bootloader, for example u-boot (an open source bootloader). U-boot passes the kernel a **hardware** device tree that lists and describes all system hardware resources available to the host kernel (CPUs/cores, memory, interrupt controller and I/O).

Similarly, when a guest OS is booted in a KVM/QEMU virtual machine, QEMU passes it a **guest** device tree that describes all the hardware resources in the VM. See Figure below.

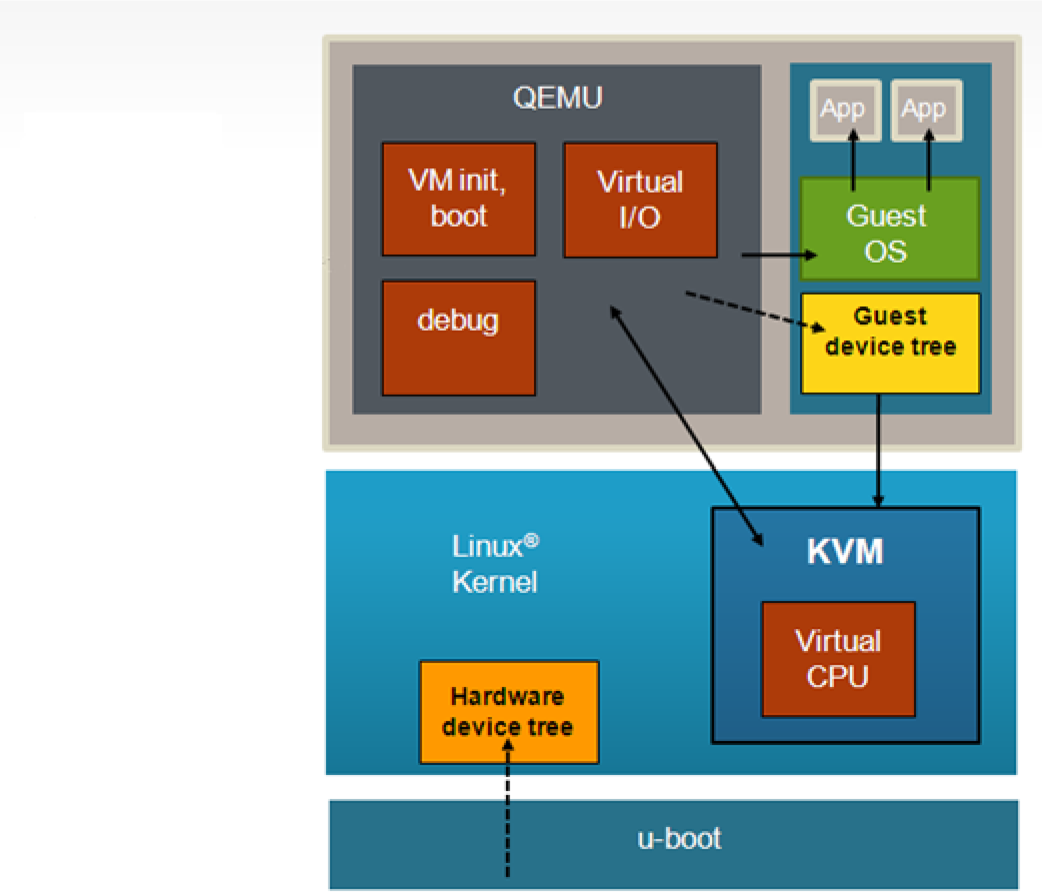


Figure 224.

The guest device tree is generated by QEMU and is used to define the resources a virtual machine will see. The guest device tree defines CPUs, memory, and I/O devices. QEMU places the guest device tree in the virtual machine's memory prior to starting the virtual machine.

10.1.1.5 References

[1] QEMU Emulator User Documentation: http://qemu.weilnetz.de/qemu-doc.html

[2] The Linux usage model for device tree data: https://www.kernel.org/doc/Documentation/devicetree/usage-model.txt

[3] Specification for virtio devices: http://docs.oasis-open.org/virtio/virtio/v1.0/csprd01/virtio-v1.0-csprd01.pdf

10.1.1.6 For More Information

KVM

- KVM website: http://www.linux-kvm.org
- ARM VM specification: http://lwn.net/Articles/589122/
- Supporting KVM on ARM architecture: http://lwn.net/Articles/557132/

QEMU

- QEMU website: http://www.qemu.org/

Device Trees

- devicetree.org website: http://devicetree.org
- DTC, the device tree compiler is available at: https://git.kernel.org/pub/scm/utils/dtc/dtc.git . DTC also includes a library called libfdt which can be used by software to parse device trees.

Virtio-- a framework for doing virtual I/O using KVM/QEMU

- http://www.ibm.com/developerworks/linux/library/l-virtio/
- http://ozlabs.org/~rusty/virtio-spec/virtio-paper.pdf
- http://docs.oasis-open.org/virtio/virtio/v1.0/csprd01/virtio-v1.0-csprd01.pdf

Virtual Networking with QEMU

- http://wiki.qemu.org/Documentation/Networking
- http://www.linux-kvm.org/page/Networking

10.1.1.7 Virtual machine reference

10.1.1.7.1 VM Overview

In general the architecture of KVM/QEMU is such that few changes should be needed to guest software to run in a VM-- i.e. a full virtualization approach is used, which means that virtual CPUs and virtual I/O devices behave like the physical hardware they are emulating.

However, there are some differences between virtual machines and native hardware that should be considered when targeting an OS to a KVM virtual machine. These differences can be divided into 2 general categories that will be discussed in further detail in this section:

1. Initial state and boot
2. CPUs

10.1.1.7.2 Memory Map of Virtual I/O Devices

The virt virtual machine contains a small subset of the devices found on a SoC. The available devices will be represented in the device tree passed to the guest at boot (e.g. virtual interrupt controller, virtual PCIE controller).

10.1.1.7.3 Virtual machine state at initialization

10.1.1.7.3.1 Initial State and Boot

When booting the Host, kernel is entered into the HYP mode for ARMv7 respectively EL2 privillege level for ARMv8. After the boot the kernel uses a stub to install KVM and switches back to SVC, respectively EL1. The virtual machine has no virtualization extensions available, so the guest kernel will be entered in SVC mode (ARMv7) respectively EL1 (ARMv8).

In case of a real hardware the boot program will provide some services before giving control to the OS. The necessary steps needed to be done by the bootloader are described in the kernel documentation: *Documentation/arm/Booting/* (ARMv7), *Documentation/arm64/booting.txt* (ARMv8). In case of virtualization, KVM/QEMU makes the necessary actions to put hardware into the initial state (as seen by the guest) and also will take the role of the bootloader and makes the necessary settings.

It is recommended that a guest OS be minimally device tree aware. The libfdt library (available with the DTC tool) provides a full range of APIs to parse and manipulate device trees and will make the process of adding device tree awareness to an OS straightforward.

10.1.1.7.3.2 Initial State of Virtual CPUs

In a VM with multiple virtual CPUs, CPU #0 is the boot CPU and all other vcpus in the partition are considered secondary. The boot method for the secondary CPUs is PSCI.

The virtual CPU entry conditions comply with the entry conditions specified by *linux/Documentation/arm/Booting* (ARMv7) or *Documentation/ arm64/booting.txt* (ARMv8)

10.1.1.7.4 Virtual CPUs

10.1.1.7.4.1 Virtual CPU Specification

Software running in a virtual machine sees a virtual CPU that emulates an ARMv7/ARMv8 core without virtualization extensions.

The virtual CPU type will match that of the host hardware platform.

10.1.1.7.4.2 Time in the Virtual CPU

ARM architecture has an optional extension, the generic timers, which provides:

- a counter (*physical counter*) that measures passing of time in real time
- a timer (*physical timer*) for each CPU. The timer is programmed to raise an interrupt to the CPU after a certain amount of time has passed.

The generic timers include virtualization support by introducing:

- a new counter, the *virtual counter*
- a new timer, the *virtual timer*.

This allows the virtual machine to have direct access to reading (virtual) counters and programming (virtual) timers without trapping.

KVM uses the physical timers in the host, the virtual machine access to the physical timers being disabled.

The virtual machine accesses the virtual timer and can, in this way, directly access the timer hardware without trapping to the hypervisor. However, the virtual timers do not raise virtual interrupts, but hardware interrupts which trap to the hypervisor. KVM injects a corresponding virtual interrupt into the VM when it detects that the virtual timer expired.

10.1.1.7.5 VGIC

The ARM Generic Interrupt Controller (GIC) provides hardware support for virtualization. The guest is able to mask, acknowledge and EOI interrupts without trapping to the hypervisor. However, there is a central part of the GIC called distributor which is responsible for interrupt prioritization and distribution to each CPU which does not provide virtualization extensions and for this part KVM provides an in-kernel emulation. Also, all the physical interrupts cannot be directly received by the guest. Instead, the KVM will program a virtual interrupt which will be raised in the guest. But, with the virtualization support in the GIC controller, when the guest is ACK-ing and EOI-ing the virtual interrupt, there is no need to trap into KVM.

QEMU/KVM provides 2 flavours of an emulated GIC:

- a GICv2 emulation which is the default option. Example command line:`-machine type=virt`

- a GICv3 emulation selected by the `gic-version` property. Example command line: `-machine type=virt,gic-version=3`. The GICv3 emulated interrupt controller is available only for platforms that have a physical GICv3 interrupt controller.

10.1.2 Configuring and Building

10.1.2.1 Overview

Linux with KVM enabled and QEMU can be built as part of the standard build process used to build the NXP LSDK.

The build instructions in the sections that follow assume a succesfull build/installation of the host. Please refer to the LSDK documentation for the host installation steps.

By default, the QEMU package installed on the target board will be the one retrieved from the Ubuntu 16.04 sources. In order to use features such as DPAA2 Direct Assignment or PCIE Direct Assignment, please refer to the Building QEMU on page 957 chapter in order to compile and build the necessary QEMU 2.9.0 version.

10.1.2.2 Quick Start - Recommended Configuration Options

The steps below show all the recommended configuration options to enable in order to build a kernel with virtual I/O enabled with the same kernel image serving as both host and guest. The sections that follow explain these options in further detail.

Note: The configuration options to run virtual machines are enabled by default in the LSDK. However they are listed here for reference.

1. From the main menuconfig window enable virtualization:

```
[*] Virtualization
```

2. In the virtualization menu enable the following options:

```
[*] Kernel-based Virtual Machine (KVM) support
```

3. Enable network bridging

```
Networking support  --->
    Networking options  --->
        <*> 802.1d Ethernet Bridging
```

4. Enable virtio PCI

```
Device Drivers  --->
    Virtio drivers  --->
      <*> PCI driver for virtio devices
```

5. Enable virtio for block devices

```
Device Drivers  --->
    [*] Block devices  --->
        <*>   Virtio block driver
```

6. Enable virtio for network devices

```
Device Drivers --->
  [*] Network device support
    [*] Network core driver support
          <*>   Universal TUN/TAP device driver support
          <*>   Virtio network driver
```

7. Enable vhost for virtio network devices

```
[*] Virtualization
    <*>   Host kernel accelerator for virtio net
```

8. Enable Huge TLB file support

```
File Systems --->
    Pseudo filesystems --->
        [*] Huge TLB file system support
```

9. Enable guest serial support

   ```
   Device Drivers --->
       Character devices --->
           Serial drivers --->
                <*> ARM AMBA PL011 serial port support
                [*]   Support for console on AMBA serial port
   ```

10. Enable VFIO support

    ```
    Device Drivers --->
        <*> VFIO Non-Privileged userspace driver framework
    ```

11. Enable VFIO support for QorIQ DPAA2 fsl-mc (Management Complex) devices

    ```
    Device Drivers --->
        <*> VFIO Non-Privileged userspace driver framework (VFIO [=y]) --->
            [*]   VFIO No-IOMMU support  ----
            <*>   VFIO support for QorIQ DPAA2 fsl-mc bus devices
    ```

12. Enable support for PCI VFIO

    ```
    Device Drivers --->
        <*> VFIO Non-Privileged userspace driver framework (VFIO [=y]) --->
            [*]   VFIO No-IOMMU support  ----
            <*>   VFIO support for PCI devices
    ```

10.1.2.3 Host Kernel: Enabling KVM

This section describes the core, basic options needed to enable KVM in the host kernel. KVM is enabled in the host kernel under the virtualization menu of the main kernel menuconfig window.

```
[*] Virtualization
```

Core KVM support is enabled as follows:

```
[*] Kernel-based Virtual Machine (KVM) support
```

10.1.2.4 Host Kernel: Enabling Virtual Networking

Virtual network interfaces on page 945 describes how virtual networking can be used to give each VMs a virtual network interface which share physical network interfaces in Linux.

One common approach to configuring virtual networking is for QEMU to use a tun/tap interface bridged to a physical network interface. To do this Ethernet bridging and the kernel's tun/tap features must be enabled in the host kernel:

```
Networking support  --->
          Networking options  --->
                    <*> 802.1d Ethernet Bridging
Device Drivers --->
          [*] Network device support
            [*] Network core driver support
                    <*>   Universal TUN/TAP device driver support
```

In order to enable vhost-net, the following config option should ne enabled:

```
[*] Virtualization
    <*>   Host kernel accelerator for virtio net
```

10.1.2.5 Host kernel: Enabling DPAA2 direct assignment

Direct assigned devices on page 946 chapter describes the mechanism used to passthrough fsl-mc bus devices to guest VMs using the VFIO framework. This section lists the Kconfig options that should be enabled in the Linux host kernel in order to support *DPAA2 Direct Assignment.*

Enable VFIO framework support

```
Device Drivers --->
    <*> VFIO Non-Privileged userspace driver framework
```

Enable VFIO support for QorIQ DPAA2 fsl-mc (Management Complex) devices

```
Device Drivers --->
    <*> VFIO Non-Privileged userspace driver framework (VFIO [=y]) --->
        [*]   VFIO No-IOMMU support  ----
        <*>   VFIO support for QorIQ DPAA2 fsl-mc bus devices
```

NOTE

"VFIO No-IOMMU support" option is needed (only) for VFIO support in guest (e.g. DPDK in guest userspace)

10.1.2.6 Host kernel: Enabling PCIE direct assignment

Direct assigned devices on page 946 chapter describes the mechanism used to passthough PCI devices using the VFIO framework.

This section lists the required Kconfig options in the host Linux kernel in order to use the aforementioned feature.

Enable VFIO framework support

```
Device Drivers --->
    <*> VFIO Non-Privileged userspace driver framework
```

Enable support for PCI VFIO

```
Device Drivers --->
    <*> VFIO Non-Privileged userspace driver framework (VFIO [=y]) --->
        <*>   VFIO support for PCI devices
```

10.1.2.7 Guest kernel: Enabling console

QEMU emulates an AMBA/PL011 console.

Below the kernel configuration options are shown to enable console:

```
Device Drivers --->
    Character devices --->
        Serial drivers --->
             <*> ARM AMBA PL011 serial port support
             [*]   Support for console on AMBA serial port
```

10.1.2.8 Guest Kernel: Enabling Network and Block Virtual I/O

Virtio is a framework for doing paravirtualized I/O using QEMU/KVM. In order to support communication between guest and hypervisor virtio uses a PCI transport protocol.

Below the kernel configuration options are shown to enable virtio-pci:

```
Device Drivers  --->
    Virtio drivers  --->
      <*> PCI driver for virtio devices
```

Below the kernel configuration options are shown to enable virtio drivers in the Linux kernel to support networking I/O and block (disk) I/O.

```
Device Drivers --->
          [*] Network device support
                  [*] Network core driver support
                          <*>   Virtio network driver
Device Drivers  --->
          [*] Block devices  --->
                    <*>   Virtio block driver
```

10.1.2.9 Building kernel with KVM support using flexbuild

NOTE

The steps presented here assume an understanding of using the `flex-builder` script to build LSDK. For details refer to LSDK building instructions.

The kernel can be built using the `flex-builder` script (For more information please refer to the LSDK building instructions):

```
flex-builder -c linux -a arm64 // build linux for arm64
flex-builder -c linux -a arm32 -m ls1021atwr  //build for LS1021 platform
```

If the kernel configuration needs to be changed, the menuconfig should be invoked and the necessary changes performed:

```
flex-builder -c linux -a arm64 -B menuconfig // build linux for arm64
flex-builder -c linux -a arm32 -m ls1021atwr -B menuconfig  //build for LS1021 platform
```

The same kernel image will be used by both guest and host.

10.1.2.10 Building QEMU

`flex-builder` script is used to generate the host root file system (For more details please see the LSDK building instructions). The generated host root filesystem already contains the QEMU installed. In the case the user wants to use a different QEMU version, this new version should be manually compiled and installed on the target (*Note:*The provided steps are targetting 64 bit platforms).

In order to use the DPAA2 Direct Assignment feature, a user should use the following commands in order to compile and install the proper QEMU version on the target.

1. Clone QEMU:

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

2. Get the right branch:

   ```
   $ git checkout qemu-2.9
   ```

3. Install the fdt library. The fdt library is a dependency, QEMU 2.9 uses a newer library than the one from the Ubuntu 16.04 userland, so compile it locally.

   ```
   $ git submodule update --init dtc
   ```

4. Install the dependencies. These are the minimum dependencies required:

   ```
   $ apt-get install pkg-config
   $ apt-get install libglib2.0-dev
   $ apt-get install libpixman-1-dev
   $ apt-get install libaio-dev
   $ apt-get install libusb-1.0-0-dev
   ```

   If the last two dependencies are not present, the config step will not complain but will not build the required support.

5. Configure and compile QEMU

   ```
   $ ./configure --prefix=<folder_where_the_qemu_will_be_installed> --target-list=aarch64-
   softmmu --enable-fdt --enable-kvm --with-system-pixman
   $ make
   ```

6. Install QEMU

   ```
   $ make install
   ```

7. Include the folder containing the qemu executable in the system path.

   ```
   $ export PATH=<folder_where_the_qemu_will_be_installed>/bin:$PATH
   ```

8. Make sure that the minimum required libraries are linked:

   ```
   $ ldd qemu-system-aarch64
       linux-vdso.so.1 =>  (0x0000ffffbad94000)
       libz.so.1 => /lib/aarch64-linux-gnu/libz.so.1 (0x0000ffffbad35000)
   ```

```
        libaio.so.1 => /lib/aarch64-linux-gnu/libaio.so.1 (0x0000ffffbad23000)
        libpixman-1.so.0 => /usr/lib/aarch64-linux-gnu/libpixman-1.so.0 (0x0000ffffbacbe000)
        libutil.so.1 => /lib/aarch64-linux-gnu/libutil.so.1 (0x0000ffffbacab000)
        libnuma.so.1 => /usr/lib/aarch64-linux-gnu/libnuma.so.1 (0x0000ffffbac8d000)
        libusb-1.0.so.0 => /lib/aarch64-linux-gnu/libusb-1.0.so.0 (0x0000ffffbac67000)
        libglib-2.0.so.0 => /lib/aarch64-linux-gnu/libglib-2.0.so.0 (0x0000ffffbab60000)
        libstdc++.so.6 => /usr/lib/aarch64-linux-gnu/libstdc++.so.6 (0x0000ffffba9d1000)
        libm.so.6 => /lib/aarch64-linux-gnu/libm.so.6 (0x0000ffffba924000)
        libgcc_s.so.1 => /lib/aarch64-linux-gnu/libgcc_s.so.1 (0x0000ffffba903000)
        libpthread.so.0 => /lib/aarch64-linux-gnu/libpthread.so.0 (0x0000ffffba8d7000)
        libc.so.6 => /lib/aarch64-linux-gnu/libc.so.6 (0x0000ffffba790000)
        /lib/ld-linux-aarch64.so.1 (0x0000ffffbad69000)
        libudev.so.1 => /lib/aarch64-linux-gnu/libudev.so.1 (0x0000ffffba75f000)
        libpcre.so.3 => /lib/aarch64-linux-gnu/libpcre.so.3 (0x0000ffffba6ee000)
```

9. Make sure that the qemu version is the expected one.

   ```
   $ qemu-system-aarch64 --version
   QEMU emulator version 2.9.0 ...
   ```

10.1.2.11 Creating a host Linux root filesystem

Creating a Linux root filesystem is out of the scope of this document. Please reference the NXP LSDK documentation on how to create root filesystems with `flex-builder` installer script. This section describes the software components needed on the host root filesystem to use KVM/QEMU.

The host root filesystem is the filesystem booted by the host kernel. The host rootfs is distinct from a guest root filesystem which may be needed by certain guest such as Linux.

A host root filesystem capable of running Linux as a guest needs the following components:

- Guest Linux kernel image (e.g. Image, zImage)
- QEMU executable (qemu-system-aarch64 or qemu-system-arm)
- Guest root fileystem

Example host root filesystem layout with the required components to boot a Linux guest:

```
/root/zImage                                    # guest Linux kernel
/root/ubuntu_bionic_arm64_rootfs.ext4.img       # guest virtual disk image
/usr/bin/qemu-system-arm                        # QEMU for ARMv7 platforms
/usr/bin/qemu-system-aarch64                    # QEMU for ARMv8 platforms
```

10.1.2.12 Creating a guest Linux root filesystem

In order to run a virtual machine, a guest Linux root filesystem is needed. There are various possibilities to host a guest root filesystem: a ramdisk, a virtual disk image, a block device on the host Linux system.

Also there are multiple virtual disk formats. `qemu-img` command can be used to generate, alter and convert between various virtual disk image formats.

A raw virtual disk can be created with the `flex-builder` script (the command is actually a wrapper over `qemu-img`):

```
flex-builder -i mkguestrfs -a <arch> -B 2G
```

The command will generate a 2GB raw disk image:

```
build/images/ubuntu_bionic_arm64_rootfs.ext4.img
```

10.1.3 KVM/QEMU How-to's

10.1.3.1 Quick-start Steps to Build and Deploy KVM

The following steps show how to build and deploy the necessay components in order to run virtual machines:

1. Build and install the LSDK on the board. (for details see #unique_113)
2. Build the guest virtual disk (for details see Creating a guest Linux root filesystem on page 958)
3. Transfer the guest virtual image and the guest image on the host. The guest image (Image or zImage) is already in the /boot partition on the host system.

10.1.3.2 Quick-start Steps to Run KVM Using Hugetlbfs

This example assumes that the host Linux kernel is booted, has a working network interface, and the following images are present in the host root filesystem:

- Guest kernel image (*/boot/zImage* or */boot/Image*)
- Guest virtual disk image (*/root/ubuntu_bionic_arm64_rootfs.ext4.img*)
- QEMU (*/usr/bin/qemu-system-arm* or */usr/bin/qemu-system-aarch64*)

Mount the HugeTLB filesystem on the host:

```
echo 512 >  /proc/sys/vm/nr_hugepages
mkdir /mnt/hugetlbfs     #any mount point can be used
mount -t hugetlbfs none /mnt/hugetlbfs/
```

This example will use 512 2M pages (2M is the defaul huge page size)

Start QEMU specifying the 2MB huge page pool as the file from which to allocate memory. In this example 512MB of memory is allocated to the VM:

64 bit ARMv8:

```
qemu-system-aarch64 -smp 8 -m 1024 -mem-path /mnt/hugetlbfs -cpu host -machine type=virt,gic-
version=3 -kernel /boot/Image -enable-kvm -display none -serial tcp::4446,server,telnet -
drive if=none,file=ubuntu_bionic_arm64_rootfs.ext4.img,id=foo,format=raw -device virtio-blk-
device,drive=foo -append 'root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk' -monitor
stdio
```

NOTE

On the GICv3 capable platforms the following emulated GIC controllers can be used:

- an emulated GICv3 interrupt controller can be used: `-machine type=virt,`**`gic-version=3`**
- an emulated GICv2 interrupt controller can be used: `-machine type=virt`

The ITS emulation is supported only with a GICv3 emulated interrupt controller.

On the GICv2 capable platforms only an emulated GICv2 interrupt comtroller can be used: `-machine type=virt`

32bit ARMv7:

```
qemu-system-arm -smp 2 -m 512 -mem-path /mnt/hugetlbfs -cpu host -machine type=virt -kernel /
boot/zImage -enable-kvm -display none -serial tcp::4446,server,telnet -drive
if=none,file=ubuntu_bionic_arm32_rootfs.ext4.img,id=foo,format=raw -device virtio-blk-
device,drive=foo -append 'root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk' -monitor
stdio
```

NOTE

Make sure that the /mnt/hugetlbfs folder exists and is mounted when starting QEMU.

Explanation of the command line options:

- `-smp 2`: specifies the number of virtual CPUs.
- `-m 512`: the amount of memory for the VM
- `-mem-path /mnt/hugetlbfs`: allocates from hugetlbfs based memory
- `-cpu host`: the type of the CPU. In this case it is the same as the host CPU

- `-machine type=virt,gic-version=3`: the type of the virtual machine: virt machine + an GICv3 emulated interrupt controller
- `-machine type=virt`: the type of the virtual machine: virt machine + an GICv2 emulated interrupt controller
- `-kernel /boot/Image` : name of guest Linux kernel
- `-enable-kvm`: specifies that KVM should be used
- `-serial tcp::4446,server,telnet` : provide an emulated serial port (telnet server) on port 4446 on the host Linux system. Default behavior will be for QEMU to wait until the user connects to this port before booting the VM.
- `-drive if=none,file=ubuntu_bionic_arm64_rootfs.ext4.img,id=foo,format=raw -device virtio-blk-device,drive=foo`: creates a virtio based virtual disk (for details see Virtual block devices on page 946)
- `-append 'root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk'`: guest Linux boot args
- `-display none`: do not display video output
- `-monitor stdio`: start QEMU monitor

At this point QEMU is waiting for a telnet connection to the virtual machine's console (port 4446 of the target board) prior to starting the virtual machine.

Connect to QEMU via telnet to start the virtual machine booting. In this example the target board has IP address 192.168.4.100.

```
-bash-3.2$ telnet 192.168.4.100 4446
Trying 192.168.4.100...
Connected to 192.168.4.100 (192.168.4.100).
Escape character is '^]'.
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Initializing cgroup subsys cpuacct
.........................
[  OK  ] Reached target Multi-User System.
[  OK  ] Reached target Graphical Interface.
         Starting Update UTMP about System Runlevel Changes...
[  OK  ] Started Update UTMP about System Runlevel Changes.

Ubuntu 16.04.2 LTS localhost ttyAMA0

localhost login: root
Password:
Last login: Mon Jun  5 22:07:09 UTC 2017 on ttyAMA0
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.65-00001-g6fed54f aarch64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage
root@localhost:~#
```

10.1.3.3 How to Use Virtual Network Interfaces Using Virtio

As discussed in Virtual network interfaces on page 945, there are two aspects of virtual network interfaces-- 1) the "front end" (the device as seen by the guest OS) and 2) the "backend" (the means by the virtual device is connected to the network).

This example uses a "virtio" model NIC card and a tap network backend. The virtual network interface is bridged via a TAP interface to the physical network. The guest OS is Linux.

When starting QEMU we will add the following arguments to create the virtual network interface:

```
-netdev tap,id=tap0,script=/home/root/qemu-ifup,downscript=no,ifname="tap0" -device virtio-net-pci,netdev=tap0
```

Perform the following steps:

1. Enable virtio networking in the host and guest Linux kernels.

2. On the host Linux create a bridge to the physical network interface to be used by the virtual network interface in the virtual machine using the **brctl** command. In this example the physical interface being used is eth2:

```
brctl addbr br0
ifconfig br0 192.168.3.30 netmask 255.255.248.0
ifconfig eth2 0.0.0.0
brctl addif br0 eth2
```

3. Create a qemu-ifup script on the host Linux system. For the TAP backend type, when QEMU creates the virtual network interface it invokes a user-created script that allows customization of how the TAP interface is to be handled. The name of the TAP interface created by QEMU is passed as an argument. In this example we will bridge the the TAP inteface to the bridge created in step #2. See the example qemu-ifup script below:

```
#!/bin/sh
# TAP interface will be passed in $1
bridge=br0
guest_device=$1
ifconfig $guest_device 0.0.0.0 up
brctl addif $bridge $guest_device
```

4. When starting QEMU specify that the network device type is "virtio" and specify the path to the script created in step #3:

```
qemu-system-aarch64 -smp 8 -m 1024 -cpu host -machine type=virt,gic-version=3 -kernel /
boot/Image -enable-kvm -display none -serial tcp::4446,server,telnet -drive
if=none,file=ubuntu_bionic_arm64_rootfs.ext4.img,id=foo,format=raw -device virtio-blk-
device,drive=foo  -netdev tap,id=tap0,script=qemu-ifup,downscript=no,ifname="tap0" -
device virtio-net-pci,netdev=tap0 -append 'root=/dev/vda rw console=ttyAMA0 rootwait
earlyprintk' -monitor stdio
```

5. In the guest OS the virtual network interface will appear and can be brought up and assigned an IP address in the normal way. In the example below (the commands are run from the guest command shell) the virtio interface is eth0.

```
root@localhost:~# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether 52:54:00:12:34:56 brd ff:ff:ff:ff:ff:ff
3: sit0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN group default qlen 1
    link/sit 0.0.0.0 brd 0.0.0.0
6: docker0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default
    link/ether 02:42:a5:57:0b:85 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever

root@localhost:~# ethtool -i enp0s1
driver: virtio_net
version: 1.0.0
firmware-version:
expansion-rom-version:
bus-info: 0000:00:01.0
supports-statistics: no
supports-test: no
supports-eeprom-access: no
supports-register-dump: no
supports-priv-flags: no

$ ifconfig enp0s1 192.168.3.31 netmask 255.255.248.0
```

10.1.3.4 How to use vhost-net with virtio

vhost-net is a character device that can be used to reduce the number of system calls involved in virtio networking. vhost-net moves network packets between the guest and the host system using the Linux kernel, bypassing QEMU.

In order to use vhost-net perform the following steps:

1. Enable virtio networking and vhost-net in the host and guest Linux kernels.
2. On the host Linux create a bridge to the physical network interface to be used by the virtual network interface in the virtual machine using the **brctl** command. In this example the physical interface being used is eth2:

   ```
   brctl addbr br0
   ifconfig br0 192.168.3.30 netmask 255.255.248.0
   ifconfig eth2 0.0.0.0
   brctl addif br0 eth2
   ```

3. Create a qemu-ifup script on the host Linux system. For the TAP backend type, when QEMU creates the virtual network interface it invokes a user-created script that allows customization of how the TAP interface is to be handled. The name of the TAP interface created by QEMU is passed as an argument. In this example we will bridge the the TAP inteface to the bridge created in step #2. See the example qemu-ifup script below:

   ```
   #!/bin/sh
   # TAP interface will be passed in $1
   bridge=br0
   guest_device=$1
   ifconfig $guest_device 0.0.0.0 up
   brctl addif $bridge $guest_device
   ```

4. When starting QEMU specify that the network device type is "virtio" and that vhost-net (**vhost=on** parameter) is used:

   ```
   qemu-system-aarch64 -smp 8 -m 1024 -cpu host -machine type=virt,gic-version=3 -kernel /
   boot/Image -enable-kvm -display none -serial tcp::4446,server,telnet -drive
   if=none,file=ubuntu_bionic_arm64_rootfs.ext4.img,id=foo,format=raw -device virtio-blk-
   device,drive=foo  -netdev tap,id=tap0,script=qemu-
   ifup,downscript=no,ifname="tap0",vhost=on -device virtio-net-pci,netdev=tap0 -append
   'root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk' -monitor stdio
   ```

5. In the guest the virtual interface will come up as described in How to Use Virtual Network Interfaces Using Virtio on page 960. In the Host kernel the vhost thread can be seen consuming CPU:

   ```
    2928 root      20   0 3258364 458340  19956 S 109.3  3.1   1:59.36 qemu-system-
   aar
    2944 root      20   0       0      0      0 R  99.7  0.0   1:43.52
   vhost-2928

    3020 root      20   0  225660   1224   1068 S  88.7  0.0   0:05.75 iperf
   ```

10.1.3.5 How to Use Virtual Disks Using Virtio

As discussed in Virtual block devices on page 946, there are a number of formats available for virtual disk images.

The example below uses a raw file. The steps below go through the process of creating a virtual disk image, assigning it to a VM, partitioning the disk, creating a filesystem on it, and mounting it.

1. On the host Linux, create a binary image to represent the guest disk. For example to create a 16MB disk:

   ```
   $ dd if=/dev/zero of=my_guest_disk bs=4K count=4K
   ```

2. Start QEMU, specifying the name of the virtual disk file for the -drive argument:

   ```
   qemu-system-aarch64 -smp 8 -m 1024 -cpu host -machine type=virt,gic-version=3 -kernel /
   boot/Image -enable-kvm -display none -serial tcp::4446,server,telnet -drive
   if=none,file=ubuntu_bionic_arm64_rootfs.ext4.img,id=foo,format=raw -device virtio-blk-
   device,drive=foo -drive if=none,file=my_guest_disk,cache=none,id=user,format=raw -
   device virtio-blk-pci,drive=user -append 'root=/dev/vda rw console=ttyAMA0 rootwait
   earlyprintk' -monitor stdio
   ```

3. After the guest has booted the virtual disk is visible as a block device in /dev with the name vda, vdb, etc. In this example we have actually two virtual disks: one for the guest rootfs (`vda`) and one for `my_guest_disk`.

   ```
   $ ls -l /dev/vdb
   brw-rw---- 1 root disk 254, 0 Jan  1  1970 /dev/vdb
   ```

   A virtual block device can be treated like any other hard disk. It can be partitioned, formatted, and mounted.

4. Configure a partition on the disk with fdisk:

```
root@localhost:~# fdisk /dev/vdb

Welcome to fdisk (util-linux 2.27.1).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Device does not contain a recognized partition table.
Created a new DOS disklabel with disk identifier 0xc9820d64.

Command (m for help):
```

Display the partition table:

```
Command (m for help): p
Disk /dev/vdb: 16 MiB, 16777216 bytes, 32768 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xc9820d64

Command (m for help):
```

Create a new partition:

```
Command (m for help): n
Partition type
   p   primary (0 primary, 0 extended, 4 free)
   e   extended (container for logical partitions)
Select (default p): p
Partition number (1-4, default 1):
First sector (2048-32767, default 2048):
Last sector, +sectors or +size{K,M,G,T,P} (2048-32767, default 32767):

Created a new partition 1 of type 'Linux' and of size 15 MiB.

Command (m for help):
```

Display the new partition:

```
Command (m for help): p
Disk /dev/vdb: 16 MiB, 16777216 bytes, 32768 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xc9820d64

Device     Boot Start   End Sectors Size Id Type
/dev/vdb1        2048 32767   30720  15M 83 Linux
```

Write the partition table to disk and exit:

```
Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.
```

5. Create a filesystem on the new partition:

```
root@localhost:~# mkfs.ext4 /dev/vdb1
mke2fs 1.42.13 (17-May-2015)
Creating filesystem with 15360 1k blocks and 3840 inodes
Filesystem UUID: 8f0c49e4-2737-498e-a984-c5f05ba59b99
Superblock backups stored on blocks:
    8193
```

```
Allocating group tables: done
Writing inode tables: done
Creating journal (1024 blocks): done
Writing superblocks and filesystem accounting information: done
```

6. Mount the filesystem:

```
root@localhost:~# mount /dev/vdb1 /boot/
root@localhost:~# echo "A virtual disk" > /boot/test.txt
root@localhost:~# cat /boot/test.txt
A virtual disk
```

10.1.3.6 How to use virtual disks using virtio-blk-dataplane

Virtio-blk-dataplane was developed for high performance disk I/O, especially for high IOPS devices. The QEMU performs the disk I/O in a dedicated thread that is optimized for I/O performance.

In this example an sdcard is used, a block device on the Linux host.

1. Start QEMU:

```
qemu-system-aarch64 -smp 8 -m 1024 -cpu host -machine type=virt,gic-version=3 -kernel /
boot/Image -enable-kvm -display none -serial tcp::4446,server,telnet -drive
if=none,file=ubuntu_bionic_arm64_rootfs.ext4.img,id=foo,format=raw -device virtio-blk-
device,drive=foo -object iothread,id=iothread0 -drive if=none,file=/dev/
mmcblk0,cache=none,id=drive0,format=raw,aio=native -device virtio-blk-
pci,drive=drive0,scsi=off,iothread=iothread0 -append 'root=/dev/vda rw console=ttyAMA0
rootwait earlyprintk' -monitor stdio
```

2. After the guest boots, the virtual disk is visible as a block device with the name vda, vdb, etc.

```
root@localhost:~# fdisk /dev/vdb

Welcome to fdisk (util-linux 2.27.1).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.


Command (m for help): p
Disk /dev/vdb: 16 MiB, 16777216 bytes, 32768 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xc9820d64

Device     Boot Start   End Sectors Size Id Type
/dev/vdb1        2048 32767   30720  15M 83 Linux
```

In this case the disk has 1 partition. The partition can be mounted and used.

10.1.3.7 How to use DPAA2 direct assignment without scripts

As presented in the introductory chapter Direct assigned devices on page 946, the DPAA2 architecture has the concept of MC containers which are arranged in a *tree structure*. While the root container always belongs to the host Linux, the child containers can be *directly assigned* to a user-space application such as DPDK or, as in our case, to a QEMU guest VM.

In the pursuit of creating a guest VM with one DPAA2 network interface directly assigned, we first need to create the child container and all the necessary MC objects.

In order to determine the number of DPAA2 objects needed to create a network interfaceDPAA2 objects dependencies on page 701. For our example the following rule applies:

- the DPIO number should be equal to the number of cores for the guest VM to be deployed (for better performance)
- the DPCON number is equal to the number of cores multiplied by the number of interfaces
- one DPBP and DPMCP object for each network interface

The following section describes the steps to be followed in order to create a *single core VM with one DPAA2 network interface assigned.* The objects are created using the restool userspace program. For more details about the restool usage see DPRCs and restool on page 699chapter.

1. Create and populate the child container
   - Create the necessary MC objects
     — create the child container (this container will be assigned to the guest)

     ```
     $ restool dprc create dprc.1
     dprc.2 is created under dprc.1
     ```

     — create the necessary objects in the child container

     ```
     $ restool dpio create --container=dprc.2
     dpio.11 is created under dprc.2
     $ restool dpcon create --num-priorities=2 --container=dprc.2
     dpcon.3 is created under dprc.2
     $ restool dpmcp create --container=dprc.2
     dpmcp.25 is created under dprc.2
     $ restool dpbp create --container=dprc.2
     dpbp.4 is created under dprc.2
     $ restool dpni create --container=dprc.2
     dpni.3 is created under dprc.2
     ```

   - Change the plugged state of the newly created objects to *plugged.*

     ```
     $ restool dprc assign dprc.2 --object=dpio.11 --plugged=1
     $ restool dprc assign dprc.2 --object=dpcon.3 --plugged=1
     $ restool dprc assign dprc.2 --object=dpmcp.25 --plugged=1
     $ restool dprc assign dprc.2 --object=dpbp.4 --plugged=1
     $ restool dprc assign dprc.2 --object=dpni.3 --plugged=1
     ```

   - Check if objects were created properly by listing the contents of the child container:

     ```
     $ restool dprc show dprc.2
     dprc.2 contains 4 objects:
     object          label          plugged-state
     dpni.3                         plugged
     dpbp.4                         plugged
     dpmcp.25                       plugged
     dpio.11                        plugged
     dpcon.3                        plugged
     ```

   - Connect the dpni object to the required dpmac in your scenario:

     ```
     $ restool dprc connect dprc.1 --endpoint1=dpni.3 --endpoint2=dpmac.3
     ```

2. Bind the newly created DPRC device to the vfio-fsl-mc driver

   ```
   $ echo vfio-fsl-mc > /sys/bus/fsl-mc/devices/dprc.2/driver_override
   $ echo dprc.2 > /sys/bus/fsl-mc/drivers/vfio-fsl-mc/bind
   ```

3. Add the device command below (for the DPRC to be assigned) to the QEMU command line:

   ```
   -device vfio-fsl-mc,host=dprc.2
   ```

   Also, make sure to specify the appropiate number of cores for the guest VM. It should match the number of dpio objects created in the child container. In our case, 1 core.

   ```
   -smp 1
   ```

4. Make sure to assign each vcpu thread to one physical CPU only
   - Start QEMU with -S option (the vcpu threads are not yet started). We need this in order for the Ethernet drivers in the guest to correctly bind the objects to the cores.

     ```
     qemu-system-aarch64 -smp 1 -m 1024 -cpu host -machine type=virt,gic-version=3 -kernel /
     boot/Image -enable-kvm -display none -serial tcp::4446,server,telnet -drive
     if=none,file=ubuntu_bionic_arm64_rootfs.ext4.img,id=foo,format=raw -device virtio-
     blk-device,drive=foo   -append 'root=/dev/vda rw console=ttyAMA0 rootwait
     earlyprintk' -monitor stdio -device vfio-fsl-mc,host=dprc.2 -S
     ```

Get the VM thread IDs entering QEMU shell

```
(qemu) info cpus
* CPU #0: thread_id=4952
```

- Assign one vcpu thread to one core only.

```
$ taskset -p 0x1 4952
pid 4952's current affinity mask: ff
pid 4952's new affinity mask: 1
```

- Start the vcpu threads.

```
(qemu) c
```

10.1.3.8 How to use DPAA2 direct assignment with scripts

While in the previous chapter, How to use DPAA2 direct assignment without scripts on page 964, we saw how to use the DPAA2 Direct Assignment feature manually, by creating each individual DPAA2 object needed in the child DPRC, in this chapter we will present a second method to create the desired configuration for a child container that will be *assigned* to the guest VM.

In order to describe the DPAA2 object configuration for a guest VM, thus a child DPRC, we employ the *DPL - Data Path Layout* syntax. The *restool* package has a new helper script, *ls-append-dpl*, that can parse DPL files which describe a child DPRC configuration and create that scenario using the *restool* tool.

One can check if the aforementioned script is available:

```
$ which ls-append-dpl

$ ls-append-dpl --help
Usage: /usr/local/bin/ls-append-dpl [options] <dpl-file>

Options:
  -h, --help
          Print this help and exit
          root@localhost:~#
```

The next section will describe how to use the *ls-append-dpl* script in order to create the child container that will be assigned to the guest VM. The next section will cover only the DPRC creation process, step #1 from the previous chapter, while the remaining steps are still the same.

Single core guest with one network interface

Applying the rule presented before, we already know that in order to assign a network interface to a *single core guest* the child container should contain: 1 DPNI, 1 DPBP, 1 DPMCP, 1 DPIO and 1 DPCON.

- Create the DPL file

  The file *vm_1_core.dts* is a text file that uses the DPL syntax and describes the required configuration for a child container that will be used for a *single core, one network interface* guest.

  It has the exact same syntax as a DPL file used to describe the static host configuration. In the *vm_1_core.dts* file we can see that a *dprc* object is described:

```
        dprc@2 {
                compatible = "fsl,dprc";
                parent = "dprc.1";
```

  The *parent* property is mandatory and it should describe the parent container for the new one.

  In this simple configuration, the single *dpni* created is connected to the *dpmac@1* in the *connections* section as follows:

```
        connection@1{
                endpoint1 = "dpni@1";
                endpoint2 = "dpmac@1";
        };
```

  If you want to connect the *dpni@1* with any other object just change the value of *endpoint2*. For example, for a connection to be established with dpmac@2 change the fragment to:

```
            endpoint2 = "dpmac@2";
```

- Deploy the DPL configuration

```
$ ls-append-dpl vm_1_core.dts
Created the following objects:
        dpmcp.50
        dpni.1
        dpio.8
        dpcon.1
        dprc.2
        dpbp.1
```

Multi core guest with one network interface

In order to tranzition from *1 core* guest to a *multi core* one, only the number of *dpio* and *dpcon* objects described in the DPL file need to be changed. Thus, in the case of a guest VM with 8 cores and one DPAA2 network interface, the DPL files should list and describe: *8 dpio, 8 dpcon, 1 dpmcp, 1 dpbp, 1 dpni.*

The vm_8_core.dts describes the configuration required for a 8 core guest VM with one DPAA2 interface. You can use it in a similar fashion:

```
$ ls-append-dpl vm_8_core.dts
```

ANNEX 1 - vm_1_core.dts

```
/dts-v1/;
/ {
    dpl-version = <10>;
    /*****************************************************************
     * Containers
     *****************************************************************/
    containers {

        dprc@2 {
            compatible = "fsl,dprc";
            parent = "dprc.1";
            options = "DPRC_CFG_OPT_SPAWN_ALLOWED", "DPRC_CFG_OPT_ALLOC_ALLOWED",
"DPRC_CFG_OPT_OBJ_CREATE_ALLOWED", "DPRC_CFG_OPT_IRQ_CFG_ALLOWED";

            objects {

                /* -------------- DPBPs --------------*/
                obj_set@dpbp {
                    type = "dpbp";
                    ids = <1>;
                };

                /* -------------- DPCONs --------------*/
                obj_set@dpcon {
                    type = "dpcon";
                    ids = <1>;
                };

                /* -------------- DPIOs --------------*/
                obj_set@dpio {
                    type = "dpio";
                    ids = <1>;
                };

                /* -------------- DPMCPs --------------*/
                obj_set@dpmcp {
                    type = "dpmcp";
                    ids = <1>;
                };

                /* -------------- DPNIs --------------*/
                obj_set@dpni {
                    type = "dpni";
                    ids = <1>;
                };
            };
        };
    };

    /*****************************************************************
     * Objects
     *****************************************************************/
    objects {

        dpbp@1 {
            compatible = "fsl,dpbp";
        };

        dpcon@1 {
            compatible = "fsl,dpcon";
            num_priorities = <0x2>;
        };

        dpio@1 {
            compatible = "fsl,dpio";
            channel_mode = "DPIO_LOCAL_CHANNEL";
            num_priorities = <0x8>;
        };

        dpmcp@1 {
            compatible = "fsl,dpmcp";
        };

        dpni@1 {
            compatible = "fsl,dpni";
            type = "DPNI_TYPE_NIC";
            options = "DPNI_OPT_NO_FS";
            num_queues = <8>;
            num_tcs = <1>;
            mac_filter_entries = <16>;
            vlan_filter_entries = <0>;
            fs_entries = <0>;
            qos_entries = <0>;
        };
    };

    /*****************************************************************
     * Connections
     *****************************************************************/
    connections {

        connection@1{
            endpoint1 = "dpni@1";
            endpoint2 = "dpmac@1";
        };
    };
};
```

ANNEX 2 - vm_8_core.dts

```
/dts-v1/;
/ {
    dpl-version = <10>;
    /*****************************************************************
     * Containers
     *****************************************************************/
    containers {

        dprc@2 {
            compatible = "fsl,dprc";
            parent = "dprc.1";
            options = "DPRC_CFG_OPT_SPAWN_ALLOWED", "DPRC_CFG_OPT_ALLOC_ALLOWED",
"DPRC_CFG_OPT_OBJ_CREATE_ALLOWED", "DPRC_CFG_OPT_IRQ_CFG_ALLOWED";

            objects {

                /* ------------- DPBPs --------------*/
                obj_set@dpbp {
                    type = "dpbp";
                    ids = <1>;
                };

                /* ------------- DPCONs --------------*/
                obj_set@dpcon {
                    type = "dpcon";
                    ids = <1 2 3 4 5 6 7 8>;
                };

                /* ------------- DPIOs --------------*/
                obj_set@dpio {
                    type = "dpio";
                    ids = <1 2 3 4 5 6 7 8>;
                };

                /* ------------- DPMCPs --------------*/
                obj_set@dpmcp {
                    type = "dpmcp";
                    ids = <1>;
                };

                /* ------------- DPNIs --------------*/
                obj_set@dpni {
                    type = "dpni";
                    ids = <1>;
                };
            };
        };
    };

    /****************************************************************
     * Objects
     *****************************************************************/
    objects {

        dpbp@1 {
            compatible = "fsl,dpbp";
        };

        dpcon@1 {
            compatible = "fsl,dpcon";
            num_priorities = <0x2>;
        };

        dpcon@2 {
            compatible = "fsl,dpcon";
            num_priorities = <0x2>;
        };

        dpcon@3 {
            compatible = "fsl,dpcon";
            num_priorities = <0x2>;
        };

        dpcon@4 {
            compatible = "fsl,dpcon";
            num_priorities = <0x2>;
        };

        dpcon@5 {
            compatible = "fsl,dpcon";
            num_priorities = <0x2>;
        };

        dpcon@6 {
            compatible = "fsl,dpcon";
            num_priorities = <0x2>;
        };

        dpcon@7 {
            compatible = "fsl,dpcon";
            num_priorities = <0x2>;
        };

        dpcon@8 {
            compatible = "fsl,dpcon";
            num_priorities = <0x2>;
        };

        dpio@1 {
            compatible = "fsl,dpio";
            channel_mode = "DPIO_LOCAL_CHANNEL";
            num_priorities = <0x8>;
        };

        dpio@2 {
            compatible = "fsl,dpio";
            channel_mode = "DPIO_LOCAL_CHANNEL";
            num_priorities = <0x8>;
        };

        dpio@3 {
            compatible = "fsl,dpio";
            channel_mode = "DPIO_LOCAL_CHANNEL";
            num_priorities = <0x8>;
        };

        dpio@4 {
            compatible = "fsl,dpio";
            channel_mode = "DPIO_LOCAL_CHANNEL";
            num_priorities = <0x8>;
        };

        dpio@5 {
            compatible = "fsl,dpio";
            channel_mode = "DPIO_LOCAL_CHANNEL";
            num_priorities = <0x8>;
        };

        dpio@6 {
            compatible = "fsl,dpio";
            channel_mode = "DPIO_LOCAL_CHANNEL";
            num_priorities = <0x8>;
        };

        dpio@7 {
            compatible = "fsl,dpio";
            channel_mode = "DPIO_LOCAL_CHANNEL";
            num_priorities = <0x8>;
        };

        dpio@8 {
            compatible = "fsl,dpio";
            channel_mode = "DPIO_LOCAL_CHANNEL";
            num_priorities = <0x8>;
        };

        dpmcp@1 {
            compatible = "fsl,dpmcp";
        };
        dpni@1 {
            compatible = "fsl,dpni";
            type = "DPNI_TYPE_NIC";
            options = "DPNI_OPT_NO_FS";
            num_queues = <8>;
            num_tcs = <1>;
            mac_filter_entries = <16>;
            vlan_filter_entries = <0>;
            fs_entries = <0>;
            qos_entries = <0>;
        };
    };

    /*****************************************************************
     * Connections
     *****************************************************************/
    connections {

        connection@1{
            endpoint1 = "dpni@1";
            endpoint2 = "dpmac@1";
        };
    };
};
```

10.1.3.9 How to use PCIE direct assignment

Select the PCIe device that will be assigned to Virtual Machine. For example, it is e1000e PCI network device (0000.01.00.0).

1. Bind the PCI device to the VFIO driver:
   - Assume e1000e device with identity 0000.01.00.0

   ```
   echo vfio-pci > /sys/bus/pci/devices/0000\:01\:00.0/driver_override
   echo 0000:01:00.0 > /sys/bus/pci/drivers/e1000e/unbind
   echo 0000:01:00.0 > /sys/bus/pci/drivers/vfio-pci/bind
   ```
2. All device in the `iommu-group` must be assigned to same virtual machine.
   - The command below will list all devices in the same `iommu-group`:

   ```
   ls -l /sys/bus/pci/devices/0000:06:0d.0/iommu_group/devices
   ```
   - All devices must be bound to VFIO using step (1) above.
3. Add the device command below to the QEMU command line for all devices in the `iommu-group`:

   ```
   -device vfio-pci,host=0000:01:00.0
   ```
4. Device will be available in Virtual Machine.

This feature is enabled for LS1088 and LS2088 devices only.

10.1.3.10 Debugging: How to Examine Initial Virtual Machine State with QEMU

It can be helpful when debugging to examine the state of the virtual machine prior to executing the first instruction of the guest OS.

To do this, start QEMU with the -S option.

Example:

```
qemu-system-aarch64 -smp 8 -m 1024 -cpu host -machine type=virt,gic-version=3 -kernel /boot/
Image -enable-kvm -display none -serial tcp::4446,server,telnet -drive
if=none,file=ubuntu_bionic_arm64_rootfs.ext4.img,id=foo,format=raw -device virtio-blk-
```

```
device,drive=foo -append 'root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk' -monitor
stdio -S
```

The console was started with the "-serial tcp::4446,server,telnet" option so QEMU waits for a connection prior to starting initialization. Use telnet to connect to port 4446 of the target.

At this point QEMU initializes the VM, but does not execute the entry point to the guest OS. The monitor prompt can now be used to examine initial state:

```
QEMU 2.5.0 monitor - type 'help' for more information
(qemu) QEMU waiting for connection on: disconnected:telnet::4446,server

(qemu)
```

To see where boot images are loaded and placed by QEMU use the **info roms** command:

```
(qemu) info roms
addr=0000000000000000 size=0x000038 mem=ram name="smpboot"
addr=0000000040000000 size=0x000028 mem=ram name="bootloader"
addr=0000000040080000 size=0xf0aa00 mem=ram name="/boot/Image"
addr=0000000048000000 size=0x010000 mem=ram name="dtb"
/rom@etc/acpi/tables size=0x200000 name="etc/acpi/tables"
/rom@etc/table-loader size=0x000880 name="etc/table-loader"
/rom@etc/acpi/rsdp size=0x000024 name="etc/acpi/rsdp"
(qemu)
```

A trivial bootloader is loaded at the start of guest memory at 0x40000000

The kernel image (zImage) is loaded at 0x40080000.

To examine the initial state of registers use the info registers command:

```
(qemu) info registers
PC=0000000040000000  SP=0000000000000000
X00=0000000000000000 X01=0000000000000000 X02=0000000000000000 X03=0000000000000000
X04=0000000000000000 X05=0000000000000000 X06=0000000000000000 X07=0000000000000000
X08=0000000000000000 X09=0000000000000000 X10=0000000000000000 X11=0000000000000000
X12=0000000000000000 X13=0000000000000000 X14=0000000000000000 X15=0000000000000000
X16=0000000000000000 X17=0000000000000000 X18=0000000000000000 X19=0000000000000000
X20=0000000000000000 X21=0000000000000000 X22=0000000000000000 X23=0000000000000000
X24=0000000000000000 X25=0000000000000000 X26=0000000000000000 X27=0000000000000000
X28=0000000000000000 X29=0000000000000000 X30=0000000000000000
PSTATE=400003c5 -Z-- EL1h
q00=0000000000000000:0000000000000000 q01=0000000000000000:0000000000000000
q02=0000000000000000:0000000000000000 q03=0000000000000000:0000000000000000
q04=0000000000000000:0000000000000000 q05=0000000000000000:0000000000000000
q06=0000000000000000:0000000000000000 q07=0000000000000000:0000000000000000
q08=0000000000000000:0000000000000000 q09=0000000000000000:0000000000000000
q10=0000000000000000:0000000000000000 q11=0000000000000000:0000000000000000
q12=0000000000000000:0000000000000000 q13=0000000000000000:0000000000000000
q14=0000000000000000:0000000000000000 q15=0000000000000000:0000000000000000
q16=0000000000000000:0000000000000000 q17=0000000000000000:0000000000000000
q18=0000000000000000:0000000000000000 q19=0000000000000000:0000000000000000
q20=0000000000000000:0000000000000000 q21=0000000000000000:0000000000000000
q22=0000000000000000:0000000000000000 q23=0000000000000000:0000000000000000
q24=0000000000000000:0000000000000000 q25=0000000000000000:0000000000000000
q26=0000000000000000:0000000000000000 q27=0000000000000000:0000000000000000
q28=0000000000000000:0000000000000000 q29=0000000000000000:0000000000000000
q30=0000000000000000:0000000000000000 q31=0000000000000000:0000000000000000
FPCR: 00000000  FPSR: 00000000
(qemu)
```

The program counter is set to 0x40000000 which is the effective address of the entry point of the kernel.

10.1.3.11 Debugging: How to Profile Virtualization Overhead with KVM

Running software in a virtual machine can cause additional overhead that affects performance. The virtualization overhead is directly related to the number of times the hypervisor (KVM) is invoked to handle exception conditions that may occur in the virtual machine. These exception handling events are referred to as 'exits', because guest context is exited.

Examples of exits include things such the guest executing a privileged instruction, access a privileged CPU register, accessing a virtual I/O device, or a hardware interrupt such as a decrementer interrupt.

The type and number of exits that occur is workload dependent.

KVM implements a mechanism in which different events are logged. These events are actually tracepoint events, and perf nicely integrates with them. You have to compile the host kernel with the following options:

```
Kernel hacking  --->
     [*] Tracers  --->
         [*]   Trace process context switches and events
```

Counting Events

A count of a subset of KVM events that occur can be seen under debugfs. To see this first mount debugfs:

```
mount -t debugfs none /sys/kernel/debug
```

The statistics can be seen using perf tool:

```
# perf stat -e "kvm:*" -p 1395
^C
 Performance counter stats for process id '1395':

              5678 kvm:kvm_entry
              5678 kvm:kvm_exit
              3121 kvm:kvm_guest_fault
              2278 kvm:kvm_irq_line
                 0 kvm:kvm_mmio_emulate
                 0 kvm:kvm_emulate_cp15_imp
              2438 kvm:kvm_wfi
                 0 kvm:kvm_unmap_hva
                 2 kvm:kvm_unmap_hva_range
                 0 kvm:kvm_set_spte_hva
                 0 kvm:kvm_hvc
              3119 kvm:kvm_userspace_exit
                 0 kvm:kvm_set_irq
                 0 kvm:kvm_ack_irq
              4068 kvm:kvm_mmio
                 0 kvm:kvm_fpu
                 0 kvm:kvm_age_page

      59.316709040 seconds time elapsed
```

Tracing events

Detailed traced can be generated using ftrace:

```
[enable ftrace in kernel: events and system calls]
$echo 1 > /sys/kernel/debug/tracing/events/kvm/enable
$cat /sys/kernel/debug/tracing/trace_pipe

qemu-system-arm-1366  [000] ....   716.115891: kvm_guest_fault: ipa 0x9000000, hsr 0x93430046,
hxfar 0xa084c030, pc 0x80266a9c
qemu-system-arm-1366  [000] ....   716.115892: kvm_mmio: mmio write len 2 gpa 0x9000030 val
0xf01
qemu-system-arm-1366  [000] ....   716.115895: kvm_userspace_exit: reason KVM_EXIT_MMIO (6)
qemu-system-arm-1366  [000] d...   716.115907: kvm_entry: PC: 0x80266aa0
qemu-system-arm-1366  [000] d...   716.116234: kvm_exit: PC: 0x800cf508
qemu-system-arm-1366  [000] d...   716.118274: kvm_entry: PC: 0x800cf508
qemu-system-arm-1366  [000] d...   716.118704: kvm_exit: PC: 0x0000981c
qemu-system-arm-1366  [000] d...   716.120737: kvm_entry: PC: 0x0000981c
qemu-system-arm-1366  [000] d...   716.121159: kvm_exit: PC: 0x800bb104
qemu-system-arm-1366  [000] d...   716.123197: kvm_entry: PC: 0x800bb104
```

```
qemu-system-arm-1366  [000] d...   716.123620: kvm_exit: PC: 0x8009cae0
qemu-system-arm-1366  [000] d...   716.125696: kvm_entry: PC: 0x8009cae0
qemu-system-arm-1366  [000] d...   716.126091: kvm_exit: PC: 0x800c90f4
qemu-system-arm-1366  [000] d...   716.128130: kvm_entry: PC: 0x800c90f4
qemu-system-arm-1366  [000] d...   716.128561: kvm_exit: PC: 0x801f37f4
qemu-system-arm-1366  [000] d...   716.130594: kvm_entry: PC: 0x801f37f4
qemu-system-arm-1366  [000] d...   716.130623: kvm_exit: PC: 0x8020576c
qemu-system-arm-1366  [000] d...   716.130635: kvm_entry: PC: 0x8020576c
qemu-system-arm-1366  [000] d...   716.131018: kvm_exit: PC: 0x43014750
qemu-system-arm-1366  [000] d...   716.133053: kvm_entry: PC: 0x43014750
qemu-system-arm-1366  [000] d...   716.133478: kvm_exit: PC: 0x80205778
qemu-system-arm-1366  [000] d...   716.135555: kvm_entry: PC: 0x80205778
```

10.1.3.12 Debugging virtual machines

10.1.3.12.1 QEMU Monitor

When starting QEMU, a monitor shell is available that can be used to control and see the state of VM. By default this monitor is started in the Linux shell where QEMU is invoked.

See example below of the output when starting QEMU. The user can interact with the monitor at the (qemu) prompt.

```
QEMU 2.5.0 monitor - type 'help' for more information
(qemu) QEMU waiting for connection on: disconnected:telnet::4446,server
```

The monitor can also be exposed over a network port by using the `-monitor dev` command line option. See Overview of Using QEMU on page 943 and the QEMU user's manual [1] (see References on page 951).

Refer to the QEMU user's manual [1] for a complete listing of the monitor commands available. Below is a list of some useful commands supported in the NXP SDK implementation of QEMU:

- **help** - lists all the available commands with usage information
- **info cpus** - displays the state and thread ID of all virtual CPUs
- **info** *registers* - displays the contents of the default vcpu's registers
- **cpu cpu_number** - sets the default vcpu number
- **system_reset** - resets the VM
- **x**/*fmt addr* -- virtual memory dump starting at 'addr'
- **xp**/*fmt addr* -- physical memory dump starting at 'addr'

10.1.3.12.2 QEMU GDB Stub

QEMU supports debugging of a VM using gdb. QEMU contains a gdb stub that can be attached to from a host system and allows standard source level debugging capabilities to examine the state of the VM and do run control.

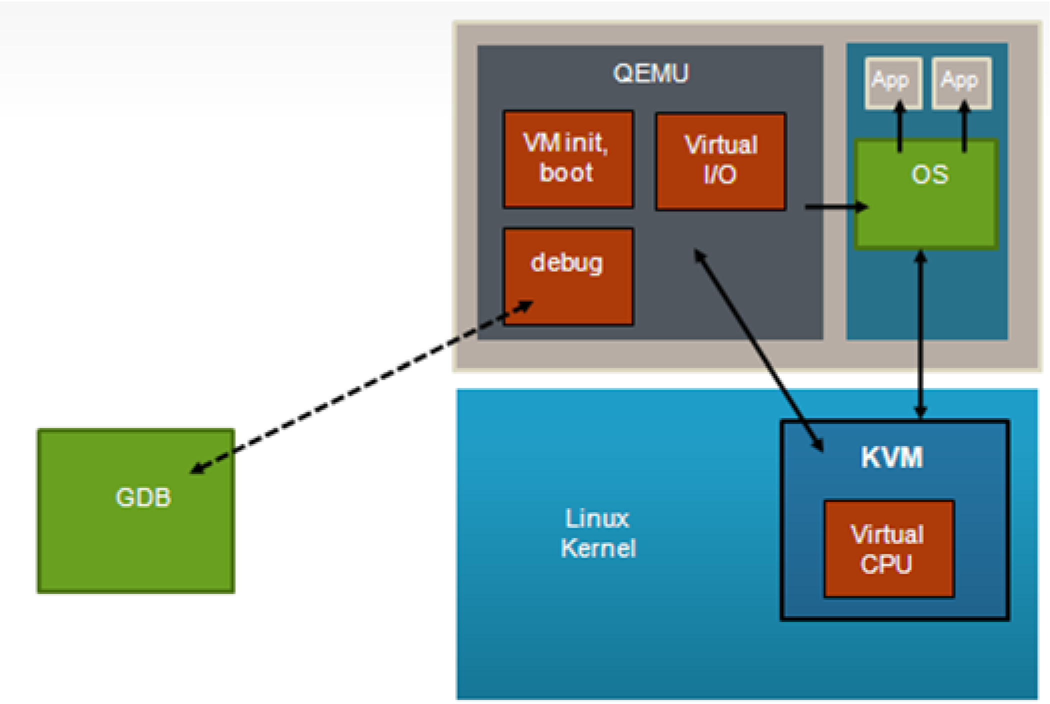


Figure 225.

To use the gdb stub, start QEMU with the -gdb *dev* option where dev specifies the type of connection to be used. See the QEMU user's manual [1] (see References on page 951) for details.

One useful option when debugging is the -S argument to QEMU which causes QEMU to wait to start the first instruction of the guest until told to start using the monitor (**continue** command).

In the example below the tcp device type is used. A gdb stub will be active on port 4445 of the host Linux kernel when starting QEMU:

```
qemu-system-aarch64 -smp 8 -m 1024 -cpu host -machine type=virt,gic-version=3 -kernel /boot/
Image -enable-kvm -display none -serial tcp::4446,server,telnet -drive
if=none,file=ubuntu_bionic_arm64_rootfs.ext4.img,id=foo,format=raw -device virtio-blk-
device,drive=foo -append 'root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk' -monitor
stdio -gdb tcp::4444
```

After the guest has been started normally, gdb can be used to connect to the VM (in this example the host kernel has an ip address of 192.168.3.30):

```
(gdb) target remote 192.168.4.100:4444
Remote debugging using 192.168.4.100:4444
0xffff000008096258 in ?? ()
```

Debugging with gdb can then proceed normally:

```
(gdb) p/x $pc
$4 = 0xffff000008096258
```

10.2 Linux Containers (LXC) for NXP QorIQ User's Guide

10.2.1 Introduction to Linux Containers

10.2.1.1 NXP LXC Release Notes

This document describes current limitations in the release of LXC for NXP SoCs.

```
Copyright (C) 2017 NXP Semiconductors, Inc.

NXP LXC Release Notes
04/27/2016

Overview
--------
This document describes new features, current limitations, and
working demos in Linux Containers (LXC) for QorIQ Layerscape SDK.


Fixes
-----
  o Seccomp support on ARMv8 platforms.

  o Unprivileged containers support on ARMv8 platforms.


SDK Demo List
-----
  o Basic container usage flow and management commands
  o Container networking setups
       o Shared networking
       o Private NICs
       o Ethernet bridge
       o MACVLAN
       o VLAN
  o Adjusting container capabilities
  o Tuning container resource usage
  o Running application containers
  o Isolating USDPAA applications in LXC containers. This has been
    tested using the USDPAA reflector app in a Multiple Instance Scenario
    on a DPAA board. After parititioning the board resources in order to
    support multiple reflector instances, these have been further isolated
    in container environments.
  o Running an unprivileged container linked to a host bridge.
  o Running containers with Seccomp protection.
```

10.2.1.2 Overview

This document is a guide and tutorial to using Linux Containers on NXP ARMv7 and ARMv8-based SoCs.

Linux Containers is a lightweight virtualization technology that allows the creation of environments in Linux called "**containers**" in which Linux applications can be run in isolation from the rest of the system and with fine grained control over resources allocated to the container (e.g. CPU, memory, network).

There are 2 implementations of containers in the LSDK:

- LXC. LXC is a user space package that provides a set of commands to create and manage containers and uses existing Linux kernel features to accomplish the desired isolation and control.
- Libvirt. The libvirt package is a virtualization toolkit that provides a set of management tools for managing virtual machines and Linux containers. The libvirt driver for containers is called "lxc", but the libvirt "lxc" driver is distinct from the user space LXC package.

Applications in a container run in a "sandbox" and can be restricted in what they can do and what visibility they have. In a container:

- An application "sees" only other processes that are in the container.
- An application has access only to network resources granted to the container.

- If configured as such, an application "sees" only a container-specific root filesystem. In addition to limiting access to data in the system's host rootfs, by limiting the */dev* entries that exist in the containers rootfs this limits the devices that the container can access.
- The file POSIX capabilities available to programs are controlled and configured by the system administrator.
- The container's processes run in what is known as a "control group" which the system administrator can use to monitor and control the container's resources.

Why are containers useful? Below are a few examples of container use cases:

- **Application partitioning** -- control CPU utilization between high priority and low priority applications, control what resources applications can access.
- **Virtual private server** -- boot multiple instances of user space, each which effectively looks like a private instance of a server. This approach is commonly used in website infrastructure.
- **Software upgrade** -- run Linux user space in a container, when it becomes necessary to upgrade applications in the system, create and test upgraded software in a new container. The old container can be stopped and the new container can be started as desired.
- **Terminal servers** -- user accesses the system with a thin client, with containers on the server providing applications. Each user gets a private, sandboxed workspace.

There are two general usage models for containers:

- **application containers**: Running a single application in a container. In this scenario, a single executable program is started in the container.
- **system containers**: Booting an instance of user space in a container. Booting multiple system containers allows multiple isolated instances of user space to run at the same time.

Containers are conceptually different than virtual machine technologies such as QEMU/KVM. Virtual machines emulate a hardware platform and are capable of booting an operating system kernel. A container is a mechanism to isolate Linux applications. In a system using containers there is only one Linux kernel running -- the host Linux kernel.

10.2.1.3 Comparing LXC and Libvirt

LXC and the lxc driver in libvirt provide similar capabilities and use the same kernel mechanisms to create containers. This section highlights some of the differences between the two tools.

LXC

- Container management is done with local LXC package commands. No remote support.
- Container creation done with **lxc-create**. LXC config file and template govern the creation of the template and the container's rootfs.

libvirt

- libvirt abstracts the container and thus a variety of tools can be used to manage containers.
- Remote management is supported.
- Container configuration defined in libvirt XML file.
- No tools to facilitate container creation.
- Same tools can be used to manage containers and KVM/QEMU virtual machines.

10.2.1.4 For Further Information

Linux containers is an approach to virtualization similar to OS virtualization solutions such as Linux VServer and OpenVZ that are widely used for virtual private servers. Documentation for these projects has helpful and relevant information:

- http://linux-vserver.org/Overview
- http://wiki.openvz.org/Main_Page

The LXC package is an open source project and much information is available online.

General Information

- libvirt LXC driver: http://libvirt.org/drvlxc.html
- Getting started with LXC using libvirt : https://www.berrange.com/posts/2011/09/27/getting-started-with-lxc-using-libvirt/
- LXC: Official web page for the LXC project: https://linuxcontainers.org/
- LXC: Overview article on LXC on IBM developerWorks (2009): http://www.ibm.com/developerworks/linux/library/l-lxc-containers/
- LXC manpages: https://linuxcontainers.org/lxc/manpages/
- Article on POSIX file capabilities: http://www.friedhoff.org/posixfilecaps.html
- SUSE LXC tutorial: https://www.suse.com/documentation/sles11/singlehtml/lxc_quickstart/lxc_quickstart.html
- LXC Linux Containers, presentation: http://www.slideshare.net/samof76/lxc-17456998
- Stephane Graber's LXC 1.0 blog posts: https://www.stgraber.org/2013/12/20/lxc-1-0-blog-post-series/
- Linux Plumbers 2013 videos: https://www.youtube.com/channel/UCIxsmRWj3-795FMlrsikd3A/videos
- Control Groups: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt

Containers and Security

If using containers to sandbox untrusted applications, a thorough understanding is needed of the capabilities granted to a container and the security vulnerabilities they may imply. The following references are helpful for understanding container security:

- Ubuntu's security issues and mitigations with LXC, https://wiki.ubuntu.com/LxcSecurity
- Emeric Nasi, Exploiting capabilities, http://packetstorm.foofus.com/papers/attack/exploiting_capabilities_the_dark_side.pdf
- Secure containers with SELinux and Smack, http://www.ibm.com/developerworks/linux/library/l-lxc-security/index.html
- Seccomp and sandboxing, http://lwn.net/Articles/332974/

Mailing Lists

For LXC, there are two mailing lists available which can be subscribed to. Archives of the lists are also available.

```
https://lists.linuxcontainers.org/listinfo/lxc-devel
https://lists.linuxcontainers.org/listinfo/lxc-users
```

10.2.2 More Details

10.2.2.1 LXC: Command Reference

This section contains links to available open source documentation for the commands in the LXC user space package.

Table 166.

LXC man page	Description	Man Page Link
lxc	lxc overview	click here
lxc-attach	start a process inside a running container	click here
lxc-autostart	start/stop/kill auto-started containers	click here

Table continues on the next page...

Table 166. (continued)

LXC man page	Description	Man Page Link
lxc-cgroup	manage the control group associated with a container	click here
lxc-checkconfig	check the current kernel for lxc support	click here
lxc-clone	clone a new container from an existing one	click here
lxc-config	query LXC system configuration	click here
lxc.conf	a description of all configuration options available	click here
lxc-console	launch a console for the specified container	click here
lxc-create	creates a container	click here
lxc-destroy	destroy a container previously created with lxc-create	click here
lxc-execute	run the specified command inside a container	click here
lxc-freeze	freeze (suspend) all the container's processes	click here
lxc-info	query information about a container	click here
lxc-ls	list the containers existing on the system	click here
lxc-monitor	monitor the container state	click here
lxc-snapshot	snapshot an existing container	click here
lxc-start	starts a container previously created with lxc-create	click here
lxc-stop	stop a container	click here
lxc-unfreeze	resumes a containers processes suspended previously with lxc-freeze	click here
lxc-unshare	run a task in a new set of namespaces	click here
lxc-usernsexec	run task as root in a new user namespace	click here
lxc-wait	wait for a specific container state	click here

The following LXC commands are not supported:

- lxc-usernsexec

10.2.2.2 LXC: Configuration Files

NOTE

This section is applicable to LXC only, not to libvirt.

For LXC, configuration files are used to configure aspects of a container at the time it is created. The configuration file defines what resources are private to the container and what is shared. By default the following resources are private to a container:

- process IDs
- sysv ipc mechanisms
- mount points

This means for example, that by default the container will share network resources and the filesystem with the host system, but will have it's own private process IDs.

The container configuration file allows additional isolation to be specified through configuration in the following areas:

- network
- console
- mount points and the backing store for the root filesystem
- control groups (cgroups)
- POSIX capabilities

See the http://man7.org/linux/man-pages/man5/lxc.conf.5.html for details on each configuration option.

When a container is created a new directory with the container's name is created in /var/lib/lxc. The configuration file for the container is stored in:

```
/var/lib/lxc/[container-name]/config
```

Below is an example of the contents of a minimal configuration file for a container named "foo", which has no networking:

```
$ cat /var/lib/lxc/foo/config
# Container with non-virtualized network
lxc.utsname = foo
lxc.tty = 1
lxc.pts = 1
lxc.rootfs = /var/lib/lxc/foo/rootfs
lxc.mount.entry=/lib /var/lib/lxc/foo/rootfs/lib none ro,bind 0 0
lxc.mount.entry=/usr/lib /var/lib/lxc/foo/rootfs/usr/lib none ro,bind 0 0
```

See the LXC: Getting Started (with a Busybox System Container) on page 984 how-to article for an introduction to the container lifecycle and how configuration files are used when creating containers.

Several example configuration files are provided with LXC:

```
/usr/share/doc/lxc-common/examples/lxc-complex.conf
/usr/share/doc/lxc-common/examples/lxc-empty-netns.conf
/usr/share/doc/lxc-common/examples/lxc-macvlan.conf
/usr/share/doc/lxc-common/examples/lxc-no-netns.conf
/usr/share/doc/lxc-common/examples/lxc-phys.conf
/usr/share/doc/lxc-common/examples/lxc-veth.conf
/usr/share/doc/lxc-common/examples/lxc-vlan.conf
/usr/share/doc/lxc-common/examples/seccomp-v1.conf
/usr/share/doc/lxc-common/examples/seccomp-v2-blacklist.conf
/usr/share/doc/lxc-common/examples/seccomp-v2.conf
```

10.2.2.3 LXC: Templates

NOTE

This section is applicable to LXC only, not to libvirt.

For LXC, When a container is "created" a directory for the container (which has the same name as the container) is created under /var/lib/lxc. This is where the container's configuration file is stored and can be edited.

For system containers (containers created with **lxc-create**), the default is for the root filesystem structure of the container to be stored here as well.

Creating containers is simplified by the use of example "templates" provided with the LXC. Template examples are provided for a number of different Linux distributions. A template is a script invoked by **lxc-create** that creates the root filesystem structure and sets up the container's config file.

The following example templates are provided with LXC and can be referred to for the expected template structure:

```
/usr/share/lxc/templates/lxc-alpine
/usr/share/lxc/templates/lxc-altlinux
/usr/share/lxc/templates/lxc-archlinux
/usr/share/lxc/templates/lxc-busybox
/usr/share/lxc/templates/lxc-centos
/usr/share/lxc/templates/lxc-cirros
/usr/share/lxc/templates/lxc-debian
/usr/share/lxc/templates/lxc-download
```

```
/usr/share/lxc/templates/lxc-fedora
/usr/share/lxc/templates/lxc-gentoo
/usr/share/lxc/templates/lxc-openmandriva
/usr/share/lxc/templates/lxc-opensuse
/usr/share/lxc/templates/lxc-oracle
/usr/share/lxc/templates/lxc-plamo
/usr/share/lxc/templates/lxc-sshd
/usr/share/lxc/templates/lxc-ubuntu
/usr/share/lxc/templates/lxc-ubuntu-cloud
```

For the NXP LSDK the busybox template is recommended and has been tested with flex-builder created root filesystems.

The how-to examples provided in this user guide that create system containers use the busybox template.

10.2.2.4 Containers with Libvirt

This section provides an overview to using libvirt-based containers.

For an general introduction to libvirt, please see the container information available on the libvirt website: http://libvirt.org/drvlxc.html.

With libvirt, a container "domain" is specified in an XML file. The XML is used to "define" the container, which then allows the container to be managed with the standard libvirt domain lifecycle.

Libvirt XML

The XML for the simplest functional container would look like the example below:

```
<domain type='lxc'>
  <name>container1</name>
  <memory>500000</memory>
  <os>
    <type>exe</type>
    <init>/bin/sh</init>
  </os>
  <devices>
    <console type='pty'/>
  </devices>
</domain>
```

Refer to the XML reference information available on the libvirt website for detailed reference information: http://libvirt.org/formatdomain.html

The <domain> element must specify a type attribute of "lxc" for a container/lxc domain. There are 4 additional sub-nodes required:

- <name> - specifies the name of the container
- <memory> - specifies the maximum memory the container may use
- <os> - identifies the initial program to run. In the example this is /bin/sh. For an application based container this is the name of the application. If booting an instance of Linux user space this would typically by /sbin/init.
- <devices> - specifies any devices, in the above example there is just a console

Filesystem mounts (from http://libvirt.org/drvlxc.html)

In the absence of any explicit configuration, the container will inherit the host OS filesystem mounts. A number of mount points will be made read only, or re-mounted with new instances to provide container specific data. The following special mounts are setup by libvirt:

- /dev a new "tmpfs" pre-populated with authorized device nodes
- /dev/pts a new private "devpts" instance for console devices
- /sys the host "sysfs" instance remounted read-only
- /proc a new instance of the "proc" filesystem

- /proc/sys the host "/proc/sys" bind-mounted read-only
- /sys/fs/selinux the host "selinux" instance remounted read-only
- /sys/fs/cgroup/NNNN the host cgroups controllers bind-mounted to only expose the sub-tree associated with the container
- /proc/meminfo a FUSE backed file reflecting memory limits of the container

Additional filesystem mounts can be created using the <filesystem> node under the <devices> node. See the libvirt.org documentation referenced above for further details.

Device nodes from http://libvirt.org/drvlxc.html

The container init process will be started with CAP_MKNOD capability removed and blocked from re-acquiring it. As such it will not be able to create any device nodes in /dev or anywhere else in its filesystems. Libvirt itself will take care of pre-populating the /dev filesystem with any devices that the container is authorized to use. The current devices that will be made available to all containers are:

- /dev/zero
- /dev/null
- /dev/full
- /dev/random
- /dev/urandom
- /dev/stdin symlinked to /proc/self/fd/0
- /dev/stdout symlinked to /proc/self/fd/1
- /dev/stderr symlinked to /proc/self/fd/2
- /dev/fd symlinked to /proc/self/fd
- /dev/ptmx symlinked to /dev/pts/ptmx
- /dev/console symlinked to /dev/pts/0

10.2.2.5 Linux Control Groups (cgroups)

Linux control groups (or cgroups) is a feature of the Linux kernel that allows the allocation, prioritization,control, and monitoring of resources such as CPU time, memory, network bandwidth among groups of Linux processes.

Cgroups is one of the underlying Linux kernel features that LXC is built upon. LXC automatically creates a cgroup for each container when it is started. A pre-requisite for using LXC is mounting the cgroup virtual filesystem.

Cgroups encompass a number of different subsystems or "controllers" that are used for managing and controlling different resources. The following subsystems/controllers are supported:

- cpu - controls CPU allocation for tasks in a cgroup;
- cpuset - assigns individual CPUs and memory nodes to tasks in a cgroup;
- cpuacct - generates automatic reports on CPU resources used by the tasks in a cgroup;
- memory - isolates the memory behavior of a group of tasks from the rest of the system;
- devices - allows or denies access to devices by tasks in a crgroup;
- freezer - suspends or resumes tasks in a cgroup;
- net_cls - tags packets with a class identifier that allows the Linux traffic controller to identify packets originating from a particular cgroup;
- net_prio - provides a way to dynamically set the priority of network traffic per each network interface for applications within various cgroups;
- blkio - controls and monitors access to I/O on block devices by tasks in cgroups.

Check out the Red Hat documentation on cgroups here: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch-Subsystems_and_Tunable_Parameters.html.

Cgroup subsystems can be configured within the configuration file used when creating a container. The configuration file accepts cgroup configuration in the following form:

```
lxc.cgroup.[subsystem name] = <value>
```

See the http://man7.org/linux/man-pages/man5/lxc.conf.5.html for further details.

Cgroup subsystems can also be displayed or updated while a container is running using the **lxc-cgroup** command:

```
lxc-cgroup -n [container-name] [cgroup-subsystem] [value]
```

For some examples of how to use cgroups to control container configuration, see the article: LXC: How to use cgroups to manage and control a containers resources on page 994.

10.2.2.6 Linux Namespaces

Linux namespaces is a feature in the Linux kernel that allows one to unshare and isolate a processes' resources like UTS, PID, IPC, file system mount and network from their parent. To achieve this the kernel places the resources in different namespaces.

When LXC spawns the container's main process it unshares all these resources except the network. The network is controlled from the configuration file and is shared by default.

A network namespace provides an isolated view of the networking stack (network device interfaces, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, the /proc/net and /sys/class/net directory trees, sockets, etc.). A physical network device can live in exactly one network namespace. A virtual network device ("veth") pair provides a pipe-like abstraction that can be used to create tunnels between network namespaces, and can be used to create a bridge to a physical network device in another namespace. When a network namespace is freed (i.e., when the last process in the namespace terminates), its physical network devices are moved back to the initial network namespace (not to the parent of the process).

Each namespace is documented in the Linux **clone** man page. See: clone (2)

10.2.2.7 POSIX Capabilities

Linux supports the concept of file "capabilities" which provides fine grained control over what executable programs are permitted to do. Instead of the "all or nothing" paradigm where a super-user or "root" has the power to perform all operations, capabilities provide a mechanism to grant a specific program specific capabilities.

LXC uses this feature of the kernel to implement containers. By default processes running in a container will have **all** capabilities, but this can be configured. Capabilities can be dropped in the container's configuration file. See LXC: Configuration Files on page 979.

For example, to drop the CAP_SYS_MODULE, CAP_MKNOD, CAP_SETUID, and CAP_NET_RAW capabilities, the following configuration file options would be specified:

```
            lxc.cap.drop = sys_module mknod setuid net_raw
```

Each capability is documented in the Linux **capabilities** man page. See: capabilities (7)

In order to fully isolate a container, the capabilities to be dropped must be carefully considered. The Linux Vserver project considers only the following capabilities as **safe** for virtual private servers:

```
CAP_CHOWN
CAP_DAC_OVERRIDE
CAP_DAC_READ_SEARCH
CAP_FOWNER
CAP_FSETID
CAP_KILL
CAP_SETGID
CAP_SETUID
CAP_NET_BIND_SERVICE
CAP_SYS_CHROOT
```

```
CAP_SYS_PTRACE
CAP_SYS_BOOT
CAP_SYS_TTY_CONFIG
CAP_LEASE
```

(see: http://linux-vserver.org/Paper#Secure_Capabilities)

10.2.3 LXC: How To's

10.2.3.1 LXC: Getting Started (with a Busybox System Container)

The following article describes steps to run a simple container example. All the command below are issued from a host Linux command prompt.

1. Confirm that your kernel environment is configured correctly using **lxc-checkconfig**. All options should show as 'enabled.

   ```
   # lxc-checkconfig
   --- Namespaces ---
   Namespaces: enabled
   Utsname namespace: enabled
   Ipc namespace: enabled
   Pid namespace: enabled
   User namespace: enabled
   Network namespace: enabled

   --- Control groups ---
   Cgroup: enabled
   Cgroup clone_children flag: enabled
   Cgroup device: enabled
   Cgroup sched: enabled
   Cgroup cpu account: enabled
   Cgroup memory controller: enabled
   Cgroup cpuset: enabled

   --- Misc ---
   Veth pair device: enabled
   Macvlan: enabled
   Vlan: enabled
   Bridges: enabled
   Advanced netfilter: enabled
   CONFIG_NF_NAT_IPV4: enabled
   CONFIG_NF_NAT_IPV6: enabled
   CONFIG_IP_NF_TARGET_MASQUERADE: enabled
   CONFIG_IP6_NF_TARGET_MASQUERADE: enabled
   CONFIG_NETFILTER_XT_TARGET_CHECKSUM: enabled
   FUSE (for use with lxcfs): enabled

   --- Checkpoint/Restore ---
   checkpoint restore: missing
   CONFIG_FHANDLE: enabled
   CONFIG_EVENTFD: enabled
   CONFIG_EPOLL: enabled
   CONFIG_UNIX_DIAG: enabled
   CONFIG_INET_DIAG: enabled
   CONFIG_PACKET_DIAG: enabled
   CONFIG_NETLINK_DIAG: enabled
   File capabilities: enabled
   ```

   Note: Before booting a new kernel, you can check its configuration

   Usage : `CONFIG=/path/to/config /usr/bin/lxc-checkconfig`

   If the cgroup namespace option shows as required:

   ```
   Cgroup namespace: required
   ```

   The `/cgroup` directory most likely needs to be created and or mounted.

2. Create a container

Create a system container using lxc-create and specify the busybox template and lxc-empty-netns.conf config file. lxc-empty-netns.conf is a simple config file with no networking:

```
lxc-create -n foo -t busybox -f /usr/share/doc/lxc-common/examples/lxc-empty-netns.conf
setting root password to "root"
Password for 'root' changed
#
```

By default, LXC will try to install the dropbear ssh utility, if it's available on the host system. The Busybox template also has support for installing OpenSSH (assuming it's installed on the host Linux) in the container. This needs to be passed explicitly using a command line parameter:

```
# lxc-create -n foo -t busybox -f /usr/share/doc/lxc-common/examples/lxc-empty-
netns.conf -- -s openssh
setting root password to "root"
Password for 'root' changed
'OpenSSH' ssh utility installed
#
```

3. List containers that exist

```
# lxc-ls -f
NAME STATE   AUTOSTART GROUPS IPV4 IPV6
foo  STOPPED 0         -      -    -
```

4. From a shell on the host Linux, start the container. When prompted, press 'Enter'.

```
# lxc-start -n foo -F

Please press Enter to activate this console.

/ #

/ #
```

Note that the shell is now running within the container. Normal Linux commands can be executed.

Important notice: while this mode starts the container and directly connects to one of its terminals, there is a minor caveat: the terminal will be stuck in this container console until the container is halted (either from here, by running **halt**, or from another terminal by running **lxc-stop**). In order to avoid this, there is also the possibility to start the container as a daemon and connect to it using **lxc-console** (this is the default mode). This provides better terminal capabilities and the user is not forced to stop the container from another terminal. On the other hand, there is no indication that after running **lxc-start** the container has actually started - no errors are reported. You must check if the container is running yourself, using **lxc-info** - see below.

```
# lxc-start -n foo
# lxc-console -n foo

Type <Ctrl+a q> to exit the console, <Ctrl+a Ctrl+a> to enter Ctrl+a itself

foo login: root
Password: (root)
~ #
~ #
~ #
~ # (Ctrl+a q)
#
```

This will be the preferred mode of starting and connecting to containers.

5. List processes in the container.

From in the container shell use the ps command to list processes:

```
~ # ps
  PID USER       VSZ STAT COMMAND
    1 root      2384 S    init
    4 root      2384 S    /bin/syslogd
    6 root      2388 S    -sh
    7 root      2384 S    init
    8 root      2388 R    ps
```

Note process IDs have a number-space unique to the container.

6. Show the status of the foo container (from a host shell):

```
# lxc-info -n foo
Name:           foo
State:          RUNNING
PID:            4544
CPU use:        0.01 seconds
Memory use:     472.00 KiB
KMem use:       0 bytes
```

7. Look at the files/directories in /var/lib/lxc related to the container

```
# ls -l /var/lib/lxc/foo
total 2
-rw-r--r--  1 root root  675 May 30 15:37 config
drwxr-xr-x 16 root root 1024 May 30 15:44 rootfs
```

This shows the containers config file and rootfs backing store.

Look at the contents of the config file:

```
# cat /var/lib/lxc/foo/config
# Template used to create this container: /usr/share/lxc/templates/lxc-busybox
# Parameters passed to the template:
# For additional config options, please look at lxc.conf(5)
lxc.utsname = omega
lxc.network.type = empty
lxc.network.flags = up
lxc.rootfs = /var/lib/lxc/foo/rootfs
lxc.haltsignal = SIGUSR1
lxc.utsname = foo
lxc.tty = 1
lxc.pts = 1
lxc.cap.drop = sys_module mac_admin mac_override sys_time

# When using LXC with apparmor, uncomment the next line to run unconfined:
#lxc.aa_profile = unconfined
lxc.mount.entry = /lib lib none ro,bind 0 0
lxc.mount.entry = /usr/lib usr/lib none ro,bind 0 0
lxc.mount.entry = /sys/kernel/security sys/kernel/security none ro,bind,optional 0 0
lxc.mount.auto = proc:mixed sys
```

8. Start a process inside the container using lxc-attach. This command will run the process inside the system container's isolated environment. The container has to be running already.

```
# lxc-attach -n foo -- /bin/sh
root@foo:/# ps
PID   USER     TIME   COMMAND
    1 root       0:00 init
    6 root       0:00 /bin/syslogd
    8 root       0:00 /bin/getty -L tty1 115200 vt100
    9 root       0:00 init
   10 root       0:00 /bin/sh
   11 root       0:00 ps
root@foo:/# ls -l /dev
total 0
crw-rw-rw-    1 root     5         136,   1 May 26 13:13 console
lrwxrwxrwx    1 root     root            13 May 26 13:12 fd -> /proc/self/fd
lrwxrwxrwx    1 root     root             7 May 26 13:13 kmsg -> console
srw-rw-rw-    1 root     root             0 May 26 13:13 log
crw-rw-rw-    1 root     root        1,   3 May 26 13:10 null
lrwxrwxrwx    1 root     root            13 May 26 13:12 ptmx -> /dev/pts/ptmx
drwxr-xr-x    2 root     root             0 May 26 13:13 pts
brw-------    1 root     root        1,   0 May 26 13:10 ram0
drwxrwxrwt    2 root     root            40 May 26 13:13 shm
lrwxrwxrwx    1 root     root            15 May 26 13:12 stderr -> /proc/self/fd/2
lrwxrwxrwx    1 root     root            15 May 26 13:12 stdin -> /proc/self/fd/0
lrwxrwxrwx    1 root     root            15 May 26 13:12 stdout -> /proc/self/fd/1
crw-rw-rw-    1 root     root         5,   0 May 26 13:10 tty
crw-rw-rw-    1 root     root         4,   0 May 26 13:10 tty0
crw--w----    1 root     root       136,   0 May 26 13:13 tty1
crw-rw-rw-    1 root     root         4,   0 May 26 13:10 tty5
crw-rw-rw-    1 root     root         1,   9 May 26 13:10 urandom
crw-rw-rw-    1 root     root         1,   5 May 26 13:10 zero
root@foo:/#
```

9. Stop the container (from a host shell)

```
# lxc-stop -n foo
#
# lxc-info -n foo
Name:           foo
State:          STOPPED
```

10. Destroy the container. This removes the containers config file and backing store.

```
# lxc-destroy -n foo
#
```

10.2.3.2 LXC: How to configure non-virtualized networking (lxc-no-netns.conf)

One approach to providing networking capability to a container is to simply allow the container to use existing host network interfaces. To accomplish this, a configuration file is created with no networking setup (i.e. the **lxc.network.type** property is not set) and the default will be to allow the container to access the host's networking interfaces.

With this approach no network namespace is created for the container.

An example config is provided:

```
/usr/share/doc/lxc-common/examples/lxc-no-netns.conf
```

The contents of lxc-no-netns.conf look like this:

```
# Container with non-virtualized network
lxc.network.type = none
lxc.utsname = delta
```

The example below shows starting an application container (running bash) with this config file and shows that the host network interface `enp1s0` is inherited and accessible by the container:

```
# lxc-execute -n mytest -f /usr/share/doc/lxc-common/examples/lxc-no-netns.conf -- /bin/bash
root@delta:/root# ifconfig
docker0   Link encap:Ethernet  HWaddr 02:42:b0:95:11:e0
          inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:b0ff:fe95:11e0/64 Scope:Link
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:18 errors:0 dropped:0 overruns:0 frame:0
          TX packets:15 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:4568 (4.5 KB)  TX bytes:1236 (1.2 KB)

enp1s0    Link encap:Ethernet  HWaddr 68:05:ca:36:9d:75
          inet addr:192.168.1.20  Bcast:0.0.0.0  Mask:255.255.248.0
          inet6 addr: fe80::6a05:caff:fe36:9d75/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:94306 errors:0 dropped:0 overruns:0 frame:0
          TX packets:40146 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:138650604 (138.6 MB)  TX bytes:2922616 (2.9 MB)
          Interrupt:100 Memory:30460c0000-30460e0000

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:1018 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1018 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:78782 (78.7 KB)  TX bytes:78782 (78.7 KB)

lxcbr0    Link encap:Ethernet  HWaddr 00:16:3e:00:00:00
          inet addr:10.0.3.1  Bcast:0.0.0.0  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

10.2.3.3 LXC: How to assign a physical network interface to a container (lxc-phys.conf)

One approach to providing networking capability to a container is to directly assign an available, unused network interface to the container. The interface is not shared, it becomes the private resource of the container.

An example LXC configuration file is provided to configure this type of networking:

```
/usr/share/doc/lxc-common/examples/lxc-phys.conf
```

The contents of the default lxc-phys.conf example are show below:

```
# Container with network virtualized using a physical network device with name
# 'eth0'
lxc.utsname = gamma
lxc.network.type = phys
lxc.network.flags = up
lxc.network.link = eth0
lxc.network.hwaddr = 4a:49:43:49:79:ff
lxc.network.ipv4 = 10.2.3.6/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3297
```

Note: The network type is set to: **phys**. Make a copy of the example config file and update it with the name of the Ethernet interface to be assigned, an appropriate IP address, and any other appropriate changes (e.g. mac address). For example, the change (in universal diff format) to set the interface `enp1s0` and IP address 192.168.10.3 would look like:

```
/usr/share/doc/lxc-common/examples/lxc-phys.conf
+++ lxc-phys.conf
@@ -3,7 +3,6 @@
 lxc.utsname = gamma
 lxc.network.type = phys
 lxc.network.flags = up
-lxc.network.link = eth0
-lxc.network.hwaddr = 4a:49:43:49:79:ff
-lxc.network.ipv4 = 10.2.3.6/24
-lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3297
+lxc.network.link = enp1s0
+lxc.network.hwaddr = 00:e0:0c:00:93:05
+lxc.network.ipv4 = 192.168.10.3/24
```

A simple way to test the new config file and the network interface is to run /bin/bash as a command with lxc-execute, which will provide a shell running in the container:

```
# lxc-execute -n mytest -f lxc-phys.conf -- /bin/bash
bash-4.2#
```

In the container, use the fm1-gb4 interface normally:

```
bash-4.3# ifconfig
enp1s0    Link encap:Ethernet  HWaddr 00:e0:0c:00:93:05
          inet addr:192.168.10.3  Bcast:192.168.10.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:508 (508.0 B)
          Memory:fe5e8000-fe5e8fff

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

bash-4.2# ping -c 3 192.168.10.1
PING 192.168.10.1 (192.168.10.1) 56(84) bytes of data.
64 bytes from 192.168.10.1: icmp_req=1 ttl=64 time=0.385 ms
64 bytes from 192.168.10.1: icmp_req=2 ttl=64 time=0.207 ms
64 bytes from 192.168.10.1: icmp_req=3 ttl=64 time=0.187 ms

--- 192.168.10.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.187/0.259/0.385/0.090 ms
```

10.2.3.4 LXC: How to configure networking with virtual Ethernet pairs (lxc-veth.conf)

One approach to providing a virtual network interface to a container is using the "Virtual ethernet pair device" feature of the Linux kernel in conjunction with a network bridge.

See the veth description in http://man7.org/linux/man-pages/man5/lxc.conf.5.html for additional details on this approach to networking.

With this approach LXC creates a new network namespace for the container.

The example configuration file lxc-veth.conf demonstrates this approach:

```
/usr/share/doc/lxc-common/examples/lxc-veth.conf
```

The contents of the default lxc-veth.conf example are show below:

```
# Container with network virtualized using a pre-configured bridge named br0 and
# veth pair virtual network devices
lxc.utsname = beta
lxc.network.type = veth
lxc.network.flags = up
lxc.network.link = br0
lxc.network.hwaddr = 4a:49:43:49:79:bf
lxc.network.ipv4 = 10.2.3.5/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3597
```

Note, the network type value is: **veth** and the link property value is **br0**.

First, create a network bridge which is attached to a physical network interface and assign the bridge an IP address. The bridge becomes one endpoint In the example below the bridge br0 is created, interface `enp1s0` is added to it, and the bridge is assigned an IP address of 192.168.20.2.

```
# brctl addbr br0
# ifconfig br0 192.168.20.2 up
# ifconfig enp1s0 up
# brctl addif br0 enp1s0
```

Make a copy of the example config file and update it with an appropriate IP address and any other appropriate changes (e.g. mac address). For example, the change (in universal diff format) to update the IP address to 192.168.20.3 would look like:

```
--- /usr/share/doc/lxc-common/examples/lxc-veth.conf
+++ lxc-veth.conf
@@ -5,5 +5,5 @@
```

```
 lxc.network.flags = up
 lxc.network.link = br0
 lxc.network.hwaddr = 4a:49:43:49:79:bf
-lxc.network.ipv4 = 10.2.3.5/24
+lxc.network.ipv4 = 192.168.20.3/24
 lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3597
```

A simple way to test the new config file and the network interface is to run /bin/bash as a command with lxc-execute, which will provide a shell running in the container:

```
# lxc-execute -n mytest -f lxc-veth.conf -- /bin/bash
bash-4.2#
```

In the container, use the virtual network interface (eth0 in this example) normally:

```
bash-4.2# ifconfig
eth0      Link encap:Ethernet  HWaddr 4a:49:43:49:79:bf
          inet addr:192.168.20.3  Bcast:192.168.20.255  Mask:255.255.255.0
          inet6 addr: fe80::4849:43ff:fe49:79bf/64 Scope:Link
          inet6 addr: 2003:db8:1:0:214:1234:fe0b:3597/64 Scope:Global
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:468 (468.0 B)  TX bytes:586 (586.0 B)

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

bash-4.2# ping -c 3 192.168.20.1
PING 192.168.20.1 (192.168.20.1) 56(84) bytes of data.
64 bytes from 192.168.20.1: icmp_req=1 ttl=64 time=0.433 ms
64 bytes from 192.168.20.1: icmp_req=2 ttl=64 time=0.221 ms
64 bytes from 192.168.20.1: icmp_req=3 ttl=64 time=0.228 ms

--- 192.168.20.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.221/0.294/0.433/0.098 ms
```

10.2.3.5 LXC: How to configure networking with macvlan (lxc-macvlan.conf)

An LXC container can be provided with a virtual network interface using the "MAC-VLAN" feature of the Linux kernel (see kernel config option CONFIG_MACVLAN). MAC-VLAN allows virtual interfaces to be created that route packets to or from a MAC address to a physical network interface.

See the macvlan description in http://man7.org/linux/man-pages/man5/lxc.conf.5.html for some additional details on this approach to networking.

The example configuration file lxc-veth.conf demonstrates this approach:

```
/usr/share/doc/lxc-common/examples/lxc-macvlan.conf
```

The contents of the provided lxc-phys.conf example configuration file are show below:

```
# Container with network virtualized using the macvlan device driver
lxc.utsname = alpha
lxc.network.type = macvlan
lxc.network.flags = up
lxc.network.link = eth0
lxc.network.hwaddr = 4a:49:43:49:79:bd
lxc.network.ipv4 = 10.2.3.4/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
```

Make a copy of the example config file and update it with the physical network interface to be used, an appropriate IP address, and any other appropriate changes (e.g. mac address). For example, the change (in universal diff format) to specify the enp1s0 interface and update the IP address to 192.168.1.24 would look like:

```
--- /usr/share/doc/lxc-common/examples/lxc-macvlan.conf
+++ lxc-macvlan.conf
@@ -2,7 +2,7 @@
 lxc.utsname = alpha
 lxc.network.type = macvlan
 lxc.network.flags = up
-lxc.network.link = eth0
+lxc.network.link = enp1s0
 lxc.network.hwaddr = 4a:49:43:49:79:bd
-lxc.network.ipv4 = 10.2.3.4/24
+lxc.network.ipv4 = 192.168.10.3/24
 lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
```

Put the network interface in promiscuous mode:

```
# ifconfig enp1s0 promisc
# ifconfig enp1s0
enp1s0    Link encap:Ethernet  HWaddr 00:e0:0c:00:93:05
          inet addr:192.168.10.2  Bcast:192.168.10.255  Mask:255.255.255.0
          inet6 addr: fe80::2e0:cff:fe00:9305/64 Scope:Link
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:5 errors:0 dropped:0 overruns:0 frame:0
          TX packets:17 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:344 (344.0 B)  TX bytes:1314 (1.2 KiB)
          Memory:fe5e0000-fe5e0fff
```

Test the MAC-VLAN interface by starting an application container running /bin/bash:

```
# lxc-execute -n mytest -f lxc-macvlan.conf -- /bin/bash
bash-4.2#
```

Note: the shell prompt above ("bash-4.2") is in the container.

Test the interface in the now running container:

```
bash-4.2# ifconfig
eth0      Link encap:Ethernet  HWaddr 4a:49:43:49:79:bd
          inet addr:192.168.10.3  Bcast:192.168.10.255  Mask:255.255.255.0
          inet6 addr: fe80::4849:43ff:fe49:79bd/64 Scope:Link
          inet6 addr: 2003:db8:1:0:214:1234:fe0b:3596/64 Scope:Global
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:586 (586.0 B)

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

bash-4.2# ping -c 3 192.168.10.1
PING 192.168.10.1 (192.168.10.1) 56(84) bytes of data.
64 bytes from 192.168.10.1: icmp_req=1 ttl=64 time=0.380 ms
64 bytes from 192.168.10.1: icmp_req=2 ttl=64 time=0.204 ms
64 bytes from 192.168.10.1: icmp_req=3 ttl=64 time=0.201 ms

--- 192.168.10.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.201/0.261/0.380/0.085 ms
```

10.2.3.6 LXC: How to configure networking using a VLAN (lxc-vlan.conf)

A container can be provided with a virtual network interface using VLANs.

See the vlan description in http://man7.org/linux/man-pages/man5/lxc.conf.5.html for some additional details on this approach to networking.

The example configuration file lxc-veth.conf demonstrates this approach:

```
/usr/share/doc/lxc-common/examples/lxc-vlan.conf
```

The contents of the provided lxc-vlan.conf example configuration file are show below:

```
 # Container with network virtualized using the vlan device driver
lxc.utsname = alpha
lxc.network.type = vlan
lxc.network.vlan.id = 1234
lxc.network.flags = up
lxc.network.link = eth0
lxc.network.hwaddr = 4a:49:43:49:79:bd
lxc.network.ipv4 = 10.2.3.4/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
```

Make a copy of the example config file and update it with the physical network interface to be used and the vlan ID, an appropriate IP address, and any other appropriate changes. For example, the change (in universal diff format) to specify the enp1s0 interface, a VLAN id of 2, and an IP address of 192.168.30.2 would look like:

```
--- /usr/share/doc/lxc/examples/lxc-vlan.conf   2013-05-30 14:22:14.980406375 +0300
+++ lxc-vlan.conf       2013-06-03 13:26:38.477580000 +0300
@@ -1,9 +1,9 @@
 # Container with network virtualized using the vlan device driver
 lxc.utsname = alpha
 lxc.network.type = vlan
-lxc.network.vlan.id = 1234
+lxc.network.vlan.id = 2
 lxc.network.flags = up
-lxc.network.link = eth0
+lxc.network.link = enp1s0
 lxc.network.hwaddr = 4a:49:43:49:79:bd
-lxc.network.ipv4 = 10.2.3.4/24
+lxc.network.ipv4 = 192.168.30.2/24
 lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
```

In this setup, the host is connected to a test machine through physical interface enp1s0. On the test machine, the following commands have been issued (interface p7p1 on this machine has physical link to enp1s0):

```
[root@everest][~]# modprobe 8021q
[root@everest][~]# lsmod | grep 8021q
8021q                  23476  0
garp                   13763  1 8021q
[root@everest][~]# vconfig add p7p1 2
Added VLAN with VID == 2 to IF -:p7p1:-
[root@everest][~]# ifconfig p7p1.2 192.168.30.1 up
```

Test the VLAN interface by starting an application container running /bin/bash:

```
# lxc-execute -n mytest -f lxc-vlan.conf -- /bin/bash
bash-4.2#
```

Test the interface in the now running container:

```
bash-4.2# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.30.2  netmask 255.255.255.0  broadcast 192.168.30.255
        inet6 fe80::21e:c9ff:fe49:bb93  prefixlen 64  scopeid 0x20<link>
        ether 00:1e:c9:49:bb:93  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 6  bytes 468 (468.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 16436
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 0  (Local Loopback)
        RX packets 4  bytes 200 (200.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4  bytes 200 (200.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

bash-4.2# ping -c 3 192.168.30.1
PING 192.168.30.1 (192.168.30.1) 56(84) bytes of data.
64 bytes from 192.168.30.1: icmp_req=1 ttl=64 time=0.338 ms
64 bytes from 192.168.30.1: icmp_req=2 ttl=64 time=0.372 ms
64 bytes from 192.168.30.1: icmp_req=3 ttl=64 time=0.355 ms

--- 192.168.30.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.338/0.355/0.372/0.013 ms
```

10.2.3.7 LXC: How to monitor containers

Containers transition through a set of well defined states. After a container is created it is in the "stopped" state.

```
 ---------
| STOPPED |<---------------
 ---------                 |
     |                     |
   start                   |
     |                     |
     V                     |
 ----------                |
| STARTING |--error-       |
 ----------         |      |
     |              |      |
     V              V      |
 ---------    ----------   |
| RUNNING |  | ABORTING |  |
 ---------    ----------   |
     |              |      |
no process          |      |
     |              |      |
     V              |      |
 ----------         |      |
| STOPPING |<-------       |
 ----------                |
     |                     |
      ---------------------
```

A number of commands are available in LXC to monitor the state of a container. The following examples provide an introduction and demonstrate the capabilities of these commands.

1. lxc-info

   The lxc-info command shows the current state of the container.

   In the example below, a container called "foo" has already been created but not started and the container is stopped:

   ```
   # lxc-info -n foo
   Name:           foo
   State:          STOPPED
   After the container is started lxc-info shows the container in the running state:
   ```

   ```
   # lxc-start -n foo
   # lxc-info -n foo
   Name:           foo
   State:          RUNNING
   PID:            5075
   ```

```
CPU use:         0.01 seconds
Memory use:      508.00 KiB
KMem use:        0 bytes
```

2. lxc-monitor

   The **lxc-monitor** command can monitor the state of one or more containers, the command continues to run until it is killed.

   In this example **lxc-monitor** monitors the state of a container named "foo":

   ```
   # lxc-monitor -n foo
   ```

   In a separate shell, start and then stop the container foo:

   ```
   # lxc-start -n foo
   # lxc-stop -n foo
   ```

   The running **lxc-monitor** command displays the state changes as they occur:

   ```
   'foo' changed state to [STARTING]
   'foo' changed state to [RUNNING]
   'foo' changed state to [STOPPING]
   'foo' changed state to [STOPPED]
   ```

3. lxc-wait

   The lxc-wait command will wait for a container state change and then exit. This can be useful for scripting and synchonizing the start or exit of a container.

   For example, to wait until the container named "foo" stops:

   ```
   # lxc-wait -n foo -s STOPPED
   ```

10.2.3.8 LXC: How to modify the capabilities of a container to provide additional isolation

As described in POSIX Capabilities on page 983 ,by default processes running in a container will have all capabilities. And the configuration for a container can further restrict these capabilities.

This example shows how to remove the ability for a container to issue the mknod command.

By default a container can issue the mknod command:

```
~ # mknod zero c 1 5
~ # ls -l zero
crw-r--r--    1 root     root         1,   5 Jun  3 17:08 zero
```

In this example we modify the config file of a container named "foo" (/var/lib/lxc/foo/config) and specify in the **lxc.cap.drop** property that the mknod capability (CAP_MKNOD) should be removed:

```
@@ -5,6 +5,7 @@
 lxc.utsname = foo
 lxc.tty = 1
 lxc.pts = 1
+lxc.cap.drop = mknod
 lxc.rootfs = /var/lib/lxc/foo/rootfs
 lxc.mount.entry=/lib /var/lib/lxc/foo/rootfs/lib none ro,bind 0 0
 lxc.mount.entry=/usr/lib /var/lib/lxc/foo/rootfs/usr/lib none ro,bind 0 0
```

Now restart the container and the mknod operation is no longer permitted:

```
~ # mknod zero c 1 5
mknod: zero: Operation not permitted
```

10.2.3.9 LXC: How to use cgroups to manage and control a containers resources

This example demonstrates how to use control croups to control which CPU's a container is scheduled on and the percentage of CPU time allocated to a container.

In this example we'll examine and change:

- the **cpuset** subsystem's **cpus** parameter which controls which physical CPUs the container's processes will run on
- the **cpu** subsystem's **shares** parameter which controls the percentage of the CPU to be allocated to the container

1. Start two application containers each running /bin/bash:

   First container:

   ```
   # lxc-execute -n foo1 -f lxc-no-netns.conf -- /bin/bash
   bash-4.2#
   ```

   Second container:

   ```
   # lxc-execute -n foo2 -f lxc-no-netns.conf -- /bin/bash
   bash-4.2#
   ```

2. In both containers start a process that will put a 100% load on the CPUs:

   ```
   (while true; do true; done) &
   ```

3. The **cpuset.cpus** subsystem/value specifies which physical CPUs the container's processes run on. From a host shell, examine this with the **lxc-cgroup** command:

   ```
   # lxc-cgroup -n foo1 cpuset.cpus
   0-7
   ```

   In this example the host system has 4 CPUs.

   This can also be seen directly through the /cgroup filesystem:

   ```
   # cat /sys/fs/cgroup/cpuset/lxc/foo1/cpuset.cpus
   0-7
   ```

4. Change both containers to run only on CPU 2:

   ```
   # lxc-cgroup -n foo1 cpuset.cpus 2
   # lxc-cgroup -n foo2 cpuset.cpus 2
   #
   ```

   The top command now shows CPU 2 with 100% utilization. The bash commands running in each container, each have about 50% of the CPU:

   ```
   top - 17:14:41 up 10 min,  4 users,  load average: 1.64, 0.61, 0.23
   Tasks: 100 total,   3 running,  97 sleeping,   0 stopped,   0 zombie
   Cpu0  :  0.0%us,  0.3%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
   Cpu1  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
   Cpu2 :100.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
   Cpu3  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
   Cpu4  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
   Cpu5  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
   Cpu6  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
   Cpu7  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
   Mem:   3996400k total,   189836k used,  3806564k free,     1652k buffers
   Swap:         0k total,        0k used,        0k free,    26180k cached

     PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
    2875 root      20   0  3624  416  164 R   50  0.0   1:28.12 bash
    2874 root      20   0  3624  424  168 R   50  0.0   1:31.06 bash
   ```

5. The cpu.shares subsystem/value specifies the percentage of the CPU allocated to the cgroup/container. By default each container has a shares value of 1024:

   ```
   # lxc-cgroup -n foo1 cpu.shares
   1024
   # lxc-cgroup -n foo2 cpu.shares
   1024
   ```

6. Change container "foo2" to have about 10% of the CPU:

   ```
   # lxc-cgroup -n foo2 cpu.shares 100
   # lxc-cgroup -n foo1 cpu.shares 900
   ```

Now the top command output reflects the new CPU allocation:

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2874 root      20   0  3624  424  168 R   90  0.0   2:53.63 bash
 2875 root      20   0  3624  416  164 R   10  0.0   2:11.36 bash
```

7. Stop the containers

```
# lxc-stop -n foo1 -k
# lxc-stop -n foo2 -k
#
```

10.2.3.10 LXC: How to run an application in a container with lxc-execute

The **lxc-execute** command allows a single application to be run in a container (as contrasted with a system container which boots an instance of Linux user space starting with System V style init).

In the example below an instance of a QEMU/KVM virtual machine is started in a container called foo.

Note, it is not required to explicitly create (and destroy) a container when running application containers with lxc-execute. The containers will automatically created and destroyed.

1. Start QEMU in the container with lxc-execute:

```
# lxc-execute -n foo -f lxc-no-netns.conf -- qemu-system-ppc -enable-kvm -smp 2 -m 256M -
nographic -M ppce500 -kernel uImage -initrd rootfs.ext2.gz -append "root=/dev/ram rw
console=ttyS0,115200" -serial tcp::4445,server,telnet
```

NOTE: For 64bit platforms, please replace qemu-system-ppc with qemu-system-ppc64.

Some notes:

- The QEMU command line follows the double dash ("--") specfied on the lxc-execute command line and distinguishes argument to lxc-execute from arguments to qemu-system-ppc.
- Using the specified configuration file, QEMU will run in the network namespace of the host system, meaning the TCP ports for serial and the monitor (ports 4445 and 4446) can be accessed from the host. However, lxc-execute will accept a configuration file as an argument allowing customization of the degree of isolation of the container.
- In this example there are 2 virtual cpus specified, which results in a total of 3 QEMU processes/threads. So we expect to see 3 QEMU processes in the container.

2. Examine the state of the container with lxc-ls and lxc-info:

```
# lxc-ls --active
foo

# lxc-info -n foo
Name:           foo
State:          RUNNING
PID:            3205
IP:             192.168.2.80
CPU use:        3.96 seconds
Memory use:     140.46 MiB
KMem use:       0 bytes
```

3. In the QEMU console look at the CPU status which shows the process IDs for the two virtual CPUs in in the virtual machine:

```
# (qemu) info cpus
* CPU #0: nip=0x00000000c001450c thread_id=4
  CPU #1: nip=0x00000000c001450c thread_id=5
(qemu)
```

Note that the process/thread IDs as viewed from within the container (thread IDs 4 and 5) are different than from the host, since they are in a different namespace.

5. Using the container's cgroup restrict the physical CPUs on which the virtual machine is allowed to run.

By default all 4 CPUs can be used by the container :

```
# by default all 4 CPUs can be used by the container
# cat /cgroup/lxc/foo/cpuset.cpus
0-3
```

Restrict the containers processes to CPUs 2 and 3:

```
# echo 2-3 > /cgroup/lxc/foo/cpuset.cpus
# cat /cgroup/lxc/foo/cpuset.cpus
2-3
```

10.2.3.11 LXC: How to run an unprivileged container

With the addition of the user namespace in the Linux kernel, a normal user on a Linux host can create and run container instances. This feature has been integrated in the LXC package, starting from version 1.0.

The steps below detail the necessary steps required in order to configure and manage an unprivileged container.

NOTE: Before running these steps, make sure that the host is properly configured for container use, by running **lxc-checkconfig** (cgroups, namespaces, etc.).

1. Create the **/etc/subuid** and **/etc/subgid** file on the Linux host. These will be used to store the unprivileged user's subordinate UIDs and GIDs. The unprivileged user has the ability to manage users on his own in his user namespace, and their IDs will be mapped to corresponding ranges on IDs on the host system. The subordinate IDs will correspond to the ranges defined in these files.

   ```
   for file in '/etc/subuid' '/etc/subgid'; do
       touch $file
       chown root:root $file
       chmod 644 $file
   done
   ```

2. Add a user in the system - **lxc-user**.

   ```
   useradd lxc-user -p $(echo test | openssl passwd -1 -stdin)
   ```

3. Check the contents of **/etc/subuid** and **/etc/subgid**. If they contain the following entries, the user has been automatically assigned a default set of subordinate IDs.

   ```
   root@t4240qds:~# cat /etc/sub*
   lxc-user:100000:65536
   lxc-user:100000:65536
   root@t4240qds:~#
   ```

   If the files are empty, you need to manually assign a set of subordinate IDs to the user.

   ```
   usermod --add-subuids 100000-165536 lxc-user
   usermod --add-subgids 100000-165536 lxc-user
   ```

4. The container will have a virtual interface linked to a bridge on the host. Use the following command to create the bridge.

   ```
   brctl addbr br0 && ifconfig br0 10.0.0.1
   ```

5. You must create and edit the **/etc/lxc/lxc-usernet** file. This file specifies how many interfaces the lxc-user will be allowed to have linked in this bridge.

   ```
   echo "lxc-user veth br0 10" > /etc/lxc/lxc-usernet
   ```

6. Create the **/home/lxc-user/.config/lxc** directory on the host. This will hold the default configuration for unprivileged containers belonging to the lxc-user.

   ```
   mkdir -p /home/lxc-user/.config/lxc
   ```

7. **Create** the default container configuration file, **/home/lxc-user/.config/lxc/default.conf**, and paste the following lines.

   ```
   lxc.network.type = veth
   lxc.network.link = br0
   lxc.network.flags = up
   lxc.id_map = u 0 100000 65536
   lxc.id_map = g 0 100000 65536
   ```

8. Change the ownership of the newly created files and folders to lxc-user.

   ```
   chown -R lxc-user:lxc-user /home/lxc-user/.config
   ```

9. For each of the mounted cgroup controllers, created a directory in the top called **lxc-user**, and change its ownership to lxc-user. Be sure to enable the cgroup.clone_children and memory.use_hierarchy flags.

   ```
   echo 1 > /sys/fs/cgroup/memory/memory.use_hierarchy

   for c in `ls /sys/fs/cgroup/`; do
       echo 1 > /sys/fs/cgroup/$c/cgroup.clone_children
       mkdir /sys/fs/cgroup/$c/lxc-user
       chown -R lxc-user:lxc-user /sys/fs/cgroup/$c/lxc-user
   done
   ```

10. **Login as the new user in a new console.**

    ```
    t4240qds login: lxc-user
    Password:
    t4240qds:~$
    ```

11. Copy the shell PID in the lxc-user cgroups.

    ```
    for c in `ls /sys/fs/cgroup/`; do
        echo $$ > /sys/fs/cgroup/$c/lxc-user/tasks
    done
    ```

12. From the same shell as before, create a Busybox container. You can pass it a custom config file using the **-f** cmdline parameter. Otherwise, it will pick the default config from **/home/lxc-user/.config/default.conf**.

    ```
    t4240qds:~$ lxc-create -n foo -t busybox
    setting root password to "root"
    Password for 'root' changed
    t4240qds:~$
    ```

13. Start the container.

    ```
    t4240qds:~$ lxc-start -n foo -F

    Please press Enter to activate this console.
    / #
    / #
    / # whoami
    root
    / #
    ```

    Now you can interact with the container as you would with one created by root. **Make sure that all container commands are run as lxc-user.**

10.2.3.12 LXC: How to run containers with Seccomp protection

A large number of system calls are exposed to every userland process with many of them going unused for the entire lifetime of the process. As system calls change and mature, bugs are found and eradicated. A certain subset of userland applications benefit by having a reduced set of available system calls. The resulting set reduces the total kernel surface exposed to the application. System call filtering is meant for use with those applications.

Seccomp (short for *secure compute*) is a system call filtering mechanism present in the kernel. Initially it has been thought to be a sandboxing mechanism that would allow userspace processes to issue a very limited set of system calls - read(), write(), exit() and sigreturn(). This has been further known to be **seccomp mode 1**, and while it is strong on the security side, it doesn't leave much room for flexibility.

The next addition to seccomp was to allow filtering (or **seccomp mode 2**) based on the kernel BPF (Berkeley Packet Filter) infrastructure. This allows the system administrator to define complex and granular policies, per system call and its arguments. This is an extension to the BPF mechanism, that allows filtering to apply to system call numbers and their arguments, besides its original purpose (socket packets). The defined filter results in a seccomp policy which is attached to the userspace process in the form of a BPF program. Each time the process issues a system call, it is checked against this policy in order to determine how it will be handled:

- **SECCOMP_RET_KILL** - the task exits immediately without executing the system call.
- **SECCOMP_RET_TRAP** - the kernel sends a SIGSYS to the triggering task without executing the system call.
- **SECCOMP_RET_ERRNO** - a custom errno is returned to userspace without executing the system call.
- **SECCOMP_RET_TRACE** - causes the kernel to attempt to notify a ptrace-based tracer prior to executing the system call. The tracer can skip the system call or change it to a valid syscall number.
- **SECCOMP_RET_ALLOW** - results in the system call being executed.

In order to make the secure computing facility more userspace-friendly, the libseccomp library has been developed, which is meant to make it easier for applications to take advantage of the packet-filter-based seccomp mode. Prior to this, userspace applications had to define the BPF filter themselves. libseccomp restructures this approach into a simple and straightforward API which userspace applications can use. The latest version of libseccomp adds support for Python bindings as well, and is designed to work on multiple architectures (ARM, MIPS). PowerPC support has also been merged on a separate branch, and is expected to be included in future releases.

Using seccomp with LXC containers

Refer toLinux Containers (LXC) for NXP QorIQ User's Guide on page 975 for information on how to build LXC with seccomp support in the SDK.

Note: Currently LXC seccomp support is not available for ARM64 architectures.

Seccomp filtering integrates well with processes sandboxed as containers, as they can be assigned to untrusted users and exposed with a limited set of allowed system calls. This is a portable and granular low-level security mechanism which can be used to increase container security. The seccomp policy file needs to be applied only to the init process in the container, and will be inherited by all its children.

The seccomp policy for the container is specified using the container configuration file, in the form of a single line containing:

```
lxc.seccomp = /var/lib/lxc/lxc_seccomp.conf
```

An example lxc_seccomp policy file can look as follows:

```
2
blacklist
[ppc64]
mknod errno 120
sched_setscheduler trap
fchmodat kill
[ppc]
mknod
```

The elements in the policy file represent the following:

1. Version number (1/2) - a single integer containing a single number, 1 or 2. Version 1 only allows to define a set of system calls which are allowed (whitelisted) in the container, specified by syscall number. This version is limited in configurability and portability, since it's only used to specify allowed syscall numbers, which may differ from arch to arch. Version 2 allows the policies to be either a whitelist (default deny, except mentioned syscalls) or a blacklist (default allow, except mentioned syscalls), and the syscalls can be expressed by name.
2. Policy type (whitelist/blacklist) - with an option of a default policy action (errno #, trap, kill, allow). The policy type is per seccomp context, and can be either whitelist or blacklist, not both.
3. Architecture tag [optional] - mentions that the following set of system calls will only be applied to a specific architecture. There can be multiple architecture tags and associated syscalls. These tags allow the same seccomp policy file to be used on multiple platforms, treating each one differently with respect to the set of system calls.
4. System calls - which can be expressed by number (in version 1) or name (in version 2). Optionally, an action can be expressed after the system call (errno #, trap, kill, allow), specifying the desired seccomp behavior. If this is omitted, the dafault rule action of the policy will be applied (allow for whitelist policies, kill for blacklist policies).

When running a container with the previous policy file on a PowerPC 64-bit platform, the mknod, sched_setscheduler (chrt) and fchmodat (chmod) system calls will be denied, with mentioned behaviors: mknod will return errno 120 without executing, chrt will trap and chmod will result in the process executing it being killed. On PowerPC platforms, only mknod will be denied, resulting in the process being killed. All other system calls will be allowed.

Notes:

- Containers can still be started without loading a seccomp policy file, simply by omitting the lxc.seccomp line in the config file. No seccomp policy is loaded by default.
- If a container process has a seccomp policy loaded, this can be seen in /proc/PID/status, on the seccomp line. This line will contain "Seccomp: 2" when using seccomp filter (mode 2). "Seccomp: 0" means there is no seccomp policy in effect.
- Seccomp policies of a process are automatically inherited by its children.
- Currently LXC supports only system call based filtering, with no support for system call arguments.
- The performance degradation of the processes running with a seccomp policy applied is directly proportional with the policy file size: normally, the system calls are listed as rules in the BPF filter program, and they all need to be parsed and matched at each system call. The longer the list, the more time this will take.
- The LXC package comes shipped with a set of example policy files which can be found at /share/doc/lxc/examples/seccomp-*. There's also a policy file, common.seccomp, which denies common security syscall threats in the container, such as kernel module manipulation, kexec and open_by_handle_at (the vector for the Shocker exploit).

10.2.4 Libvirt

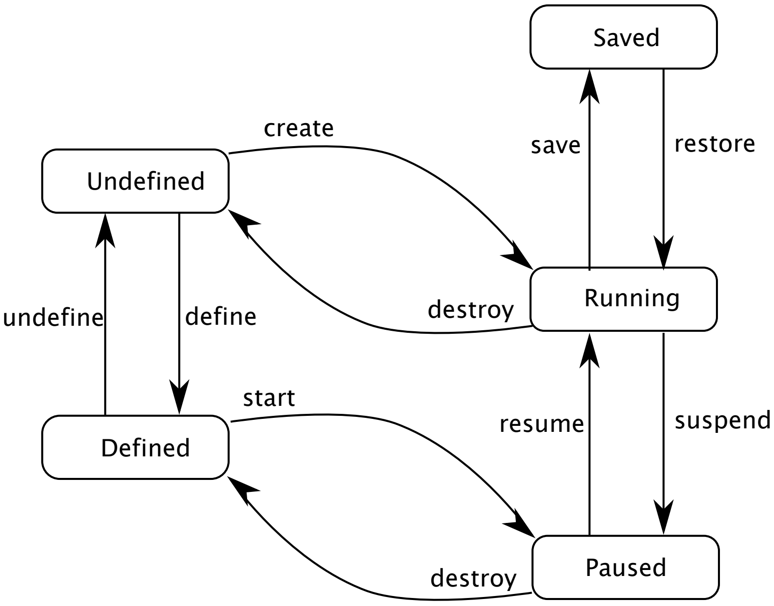
This document is a guide and tutorial to using libvirt on NXP SoCs. Libvirt is an open source toolkit that enables the management of Linux-based virtualization technologies such as KVM/QEMU virtual machines and Linux containers. The goal of the libvirt project (see https://libvirt.org) is to provide a stable, standard, hypervisor-agnostic interface for managing *virtualization domains* such as virtual machines and containers. Domains can be remote and libvirt provides full security for managing remote domains over a network. Libvirt is a layer intended to be used as a building block for higher level management tools and applications.

Libvirt provides:

- An interface to remotely manage the lifecycle of virtualization domains – provisioning, start/stop, monitoring
- Support for a variety of hypervisors – KVM/QEMU and Linux Containers are supported in the NXP SDK
- libvirtd – a Linux daemon that runs on a target node/system and allows a libvirt management tool to manage virtualization domains on the node
- virsh – a basic command shell for managing libvirt domains
- A standard XML format for defining domains

Libvirt Domain Lifecycle

Two types of libvirt domains are supported – KVM/QEMU virtual machines and Linux containers. The following state diagram illustrates the lifecycle of a domain, the states that domains can be in and the virsh commands that move the domain between states.



Domain States

- **Undefined**. There are two types of domains – persistent and transient domains. All domains begin in the *undefined* state where they are defined in XML definition file, and libvirt is unaware of them.
- **Defined**. Persistent domains begin with being *defined*. This adds the domain to libvirt, but it is not running. This state can also be conceptually thought of as *stopped*. The output of virsh list –all shows the domain as being *shut off*.
- **Running**. The *running* state is the normal state of an active domain after it has been started. The start command is used to move persistent domains into this state. Transient domains go from being undefined to *running* through the create command.
- **Paused**. The domain execution has been suspended. The domain is unaware of being in this state.
- **Saved**. The domain state has been saved and could be restored again.

Libvirt URIs

Because libvirt supports managing multiple types of virtualization domains (possibly remote) it uses uniform resource identifiers (URIs) to describes the target *node* to manage and the type of domain being managed.

An URI is specified when tools such as *virsh* makes a connection to a target node running libvirtd. Two types of URIs are supported – QEMU/KVM and LXC.

QEMU/KVM URIs are in the form:

- for a local node: *qemu:///system*
- for a remote node: *qemu[+transport]://[hostname]/system*

Linux containers URIs:

- for a local node: *lxc:///*
- for a remote node: *lxc[+transport]://[hostname]/*

A default URI can be specified using the environment variable *LIBVIRT_DEFAULT_URI* or in the */etc/libvirt/libvirtd.conf* config file.

For further information on URIs:

- https://libvirt.org/uri.html
- https://libvirt.org/remote.html#Remote_URI_reference

Virsh

The virsh command is a command line tool provided with the libvirt package for managing libvirt domains. It can be used to create, start, pause, shutdown domains. The general command format is:

```
virsh [OPTION]... <command> <domain> [ARG]...
```

Libvirt XML

The libvirt XML format is defined at http://libvirt.org/format.html.

Running libvirtd

The libvirtd daemon is installed as part of a libvirt packages installation. By default the target system init scripts should start libvirtd. Running libvirtd on the target system is a pre-requisite to running any management tools such as virsh. The libvirtd daemon can be manually started like this:

```
$ systemctl start libvirtd
```

In some circumstances the daemon may need to be restarted, such as after mounting cgroups or hugetlbfs. Daemon restart can be done like this:

```
$ systemctl restart libvirtd
```

The libvirtd daemon can be configured in */etc/libvirt/libvirtd.conf*. The file is self-documented and has detailed comments on the configuration options available.

The libvirt daemon logs data to */var/log/libvirt/*:

- General libvirtd log messages are in: */var/log/libvirt/libvirtd.log*
- QEMU/KVM domain logs are in: */var/log/libvirt/qemu/[domain-name].log*
- LXC domains logs are in: */var/log/libvirt/lxc/[domain-name].log*

The verbosity of logging can be controlled in */etc/libvirt/libvirtd.conf*.

In order to be able to start virtual machines the user used to manage virtual machines need to be added to the `libvirt` group:

```
sudo adduser <USER> libvirt
```

Examples

Libvirt KVM/QEMU Examples

Virtio Block scenario

1. We begin with a simple QEMU command line in a text file named kvm_virtio_blk.args:

   ```
   $ echo "/usr/bin/qemu-system-aarch64 -name kvm_virtio_blk -smp 2 -enable-kvm -m 1024 -
   nographic -cpu host -machine type=virt -kernel /boot/Image -serial pty -drive
   if=virtio,index=0,file=/root/ubuntu_bionic_arm64_rootfs.ext4.img,id=foo,format=raw -
   append 'root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk'" > kvm_virtio_blk.args
   ```

   Note: The serial console is a tty, not a telnet server. The *-name* option is required and specifies the name of the virtual machine.

2. Before defining the domain, the QEMU command line must be converted to libvirt XML format:

   ```
   $ virsh domxml-from-native qemu-argv kvm_virtio_blk.args > kvm_virtio_blk.xml
   ```

3. Now the domain can be defined:

```
$ virsh define kvm_virtio_blk.xml
Domain kvm_virtio_blk defined from kvm_virtio_blk.xml

$ virsh list --all
 Id    Name                           State
 ----------------------------------------------------
  - kvm_virtio_blk                 shut off
```

4. Start the domain. This starts the VM and boots the Linux Guest from the *ubuntu_bionic_arm64_rootfs.ext4.img* image.

```
$ virsh start kvm_virtio_blk
Domain kvm_virtio_blk started

$ virsh list
 Id    Name                           State
 ----------------------------------------------------
  16    kvm_virtio_blk                 running
```

5. The *virsh console* command can be used to connect to the console of the running Linux domain.

```
$ virsh console kvm_virtio_blk
Connected to domain kvm_virtio_blk
Escape character is ^]

Ubuntu 16.04.3 LTS localhost ttyAMA0

localhost login: root
Password:
Welcome to Ubuntu 16.04.3 LTS (GNU/Linux 4.9.62 aarch64)

 *Documentation:  https://help.ubuntu.com
 *Management:     https://landscape.canonical.com
 *Support:        https://ubuntu.com/advantage
```

6. To stop the domain use the *destroy* command:

```
$ virsh destroy kvm_virtio_blk
Domain kvm_virtio_blk destroyed

$ virsh list --all
  Id    Name                           State
  ----------------------------------------------------
   - kvm_virtio_blk                 shut off
```

7. To remove the domain from libvirt, use the *undefine* command:

```
$ virsh undefine kvm_virtio_blk
Domain kvm_virtio_blk has been undefined

$ virsh list --all
 Id    Name                           State
 ----------------------------------------------------
```

Note: One can find the full XML for this configuration in *Annex 1*.

Virtio Net scenario

This example uses a *virtio* model NIC card and a tap network backend. The virtual network interface is bridged via a TAP interface to the physical network.

Perform the following steps:

1. Enable virtio networking in the host and guest Linux kernels.
2. On the host, create a bridge to the physical network interface to be used by the virtual network interface in the virtual machine using the *brctl* command. In this example the physical interface being used is enp1s0:

```
$ brctl addbr br0
$ ifconfig br0 192.168.1.10 netmask 255.255.248.0
$ ifconfig enp1s0 0.0.0.0
$ brctl addif br0 enp1s0
```

3. Create a *qemu-ifup* script on the host Linux system:

   ```
   #!/bin/sh

   #TAP interface will be passed in $1
   bridge=br0
   guest_device=$1
   ifconfig $guest_device 0.0.0.0 up
   brctl addif $bridge $guest_device
   ```

4. Create a *args* file and convert it to the libvirt xml:

   ```
   $ echo "/usr/bin/qemu-system-aarch64 -name kvm_virtio_net -smp 2 -enable-kvm -m 1024 -
   nographic -cpu host -machine type=virt -kernel /boot/Image -serial pty -drive
   if=virtio,index=0,file=/root/ubuntu_bionic_arm64_rootfs.ext4.img,id=foo,format=raw -
   netdev tap,id=tap0,script=/root/qemu-ifup,downscript=no,ifname=tap0 -device virtio-net-
   pci,netdev=tap0 -append 'root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk'" >
   kvm_virtio_net.args

   $ virsh domxml-from-native qemu-argv kvm_virtio_net.args > kvm_virtio_net.xml
   ```

5. Define and start the domain. Check if the virtual network interface is created.

```
$ virsh define kvm_virtio_net.xml
Domain kvm_virtio_net defined from kvm_virtio_net.xml

$ virsh start kvm_virtio_net
Domain kvm_virtio_net started

$ virsh console kvm_virtio_net
Connected to domain kvm_virtio_net
Escape character is ^]

Ubuntu 16.04.3 LTS localhost ttyAMA0

localhost login: root
Password:

$  dmesg | grep virtio_net
[    4.121280] virtio_net virtio1 enp0s2: renamed from eth0

$  ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: enp0s2: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default qlen 1000
    link/ether 52:54:00:12:34:56 brd ff:ff:ff:ff:ff:ff
3: sit0@NONE: <NOARP> mtu 1480 qdisc noop state DOWN group default qlen 1
    link/sit 0.0.0.0 brd 0.0.0.0
4: docker0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN group default
    link/ether 02:42:81:50:d5:f5 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 scope global docker0
       valid_lft forever preferred_lft forever
```

The libvirt XML generated and used in this scenario differs from the previous one by the following lines:

```
<qemu:commandline>
  <qemu:arg value='-netdev'/>
  <qemu:arg value='tap,id=tap0,script=/root/qemu-ifup,downscript=no,ifname=tap0'/>
  <qemu:arg value='-device'/>
  <qemu:arg value='virtio-net-pci,netdev=tap0'/>
</qemu:commandline>
```

Note: Currently libvirt has no support for PCI transport, but it can be used using passthrough QEMU command line arguments (as seen in the previous xml).

Note: If you get the following error when starting the domain please use the steps from this thread to fix it.

```
could not open /dev/net/tun: Operation not permitted
```

Note: One can find the full XML for this configuration in *Annex 2*.

Virtio Block Dataplane

Virtio-blk-dataplane was developed for high performance disk I/O, especially for high IOPS devices. QEMU performs the disk I/O in a dedicated thread that is optimized for I/O performance.

Even though the scenario can use also a block device on the Linux host, the next steps will show how to implement this using a raw disk file.

Note: A direct translation between the qemu args are not possible using virsh that is why in this example we will start from the XML used in the previous scenario and build on it.

1. Create the raw disk file:

   ```
   $ dd if=/dev/zero of=/root/fake-dev0-backstore.img bs=1M count=300
   ```

2. Copy the libvirt XML file from the previous example:

   ```
   $ cp kvm_virtio_net.xml kvm_virtio_blk_dataplane.xml
   ```

3. Change the *name* and *uuid* of the new domain. Define the number of *IOThreads* to be assigned to the domain and used by the new storage device. Add the storage disk and assign it to the *iothread='1'*.

   ```
   $ diff kvm_virtio_blk_dataplane.xml kvm_virtio_net.xml
   ```

   ```
   2,3c2,3
   <   <name>kvm_virtio_blk_dataplane</name>
   <   <uuid>5c30747a-a2c9-485e-b814-2a503fef8657</uuid>
   ---
   >   <name>kvm_virtio_net</name>
   >   <uuid>5c30747a-a2c9-485e-b814-2a503fef8653</uuid>
   22d21
   <   <iothreads>1</iothreads>
   29,34d27
   <     </disk>
   <     <disk type='file' device='disk'>
   <       <driver name='qemu' type='raw' cache='none' io='native' iothread='1'/>
   <       <source file='/root/fake-dev0-backstore.img'/>
   <       <address type='pci' domain='0x0000' bus='0x00' slot='0x06' function='0x0'/>
   <       <target dev='vdb' bus='virtio'/>
   ```

4. Start the new domain and check if virtio-blk-dataplane works properly.

```
$ virsh define kvm_virtio_blk_dataplane.xml
Domain kvm_virtio_blk_dataplane defined from kvm_virtio_blk_dataplane.xml

$ virsh start kvm_virtio_blk_dataplane
Domain kvm_virtio_blk_dataplane started

# After the guest boots, the virtual disk is visible as a block device with the name vdb.
$ virsh console kvm_virtio_blk_dataplane

   root@localhost:~# ls -la /dev/vd*
   brw-rw---- 1 root disk 254,  0 Aug 23 12:00 /dev/vda
   brw-rw---- 1 root disk 254, 16 Aug 23 12:00 /dev/vdb

# We can also check if the IOThread is correctly assigned to the domain.
$ virsh iothreadinfo kvm_virtio_blk_dataplane
 IOThread ID     CPU Affinity
 ---------------------------------------------------
  1               0-7
```

Note: One can find the full XML for this configuration in *Annex 3*.

Libvirt KVM/QEMU FAQ

1. What if I get "error: XML error: No PCI buses available" error when trying to convert QEMU arguments tu XML?

If you are using a 32 bit QEMU and you are trying a command like:

```
echo -e '/usr/bin/qemu-system-arm -enable-kvm -name demo1 -enable-kvm -m 512 -nographic
-cpu host -machine type=virt -mem-path /dev/hugepages/libvirt/qemu -kernel /media/ram/
zImage  -initrd /media/ram/fsl-image-core-ls1021atwr.ext2.gz -append "root=/dev/ram rw
console=ttyAMA0,115200" -serial pty' >> demo.args
root@localhost:~# virsh domxml-from-native qemu-argv demo.args > demo1.xml
error: XML error: No PCI buses available
```

NOTE

The above QEMU command is just an example, your command can be completely different.

The reason the above mentioned command does not work is that `domxml-from-native` expects that the suffix of the `qemu-system-` to be a canonical architecture name and `arm` is not. For ARM32 bit, little endian, the canonical name is `armv7l`. The solution could be either to manually create the xml file or to create a symbolic link `qemu-system-armv7l` to point to `qemu-system-arm` and then use the symbolic link in demo.args:

```
echo -e '/usr/bin/qemu-system-armv7l -enable-kvm -name demo1 -enable-kvm -m 512 -nographic  -
cpu host -machine type=virt -mem-path /dev/hugepages/libvirt/qemu -kernel /media/ram/zImage
-initrd /media/ram/fsl-image-core-ls1021atwr.ext2.gz -append "root=/dev/ram rw
console=ttyAMA0,115200" -serial pty' >> demo.args
```

Libvirt LXC Examples

Basic Example

The following example shows the lifecycle of a simple LXC libvirt domain called *lxc_basic*.

1. Confirm the host Linux configuration. Begin by confirming that the host kernel is configured correctly and that rootfs setup such as mounting cgroups has been done. This can be done with the *lxc-checkconfig* command.
2. Create a libvirt XML file defining the container. The example below shows a very simple container defined in *lxc_basic.xml* that runs the command /bin/sh and has a console:

```
$ cat lxc_basic.xml
<domain type='lxc'>
  <name>lxc_basic</name>
  <memory>500000</memory>
  <os>
    <type>exe</type>
    <init>/bin/sh</init>
  </os>
  <devices>
    <console type='pty'/>
  </devices>
</domain>

$ virsh -c lxc:/// define lxc_basic.xml
Domain lxc_basic defined from lxc_basic.xml

$ virsh -c lxc:/// list --all
 Id    Name                           State
 ----------------------------------------------------
  - lxc_basic                      shut off

$ virsh -c lxc:/// start lxc_basic
Domain lxc_basic started

$ virsh -c lxc:/// console  lxc_basic
Connected to domain lxc_basic
Escape character is ^]
#ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 13:14 ?        00:00:00 /bin/sh
root         3     1  0 13:14 ?        00:00:00 ps -ef
```

Note: The processes inside the container are running in a separate namespace, hence the different process hierarchysince no network configuration for the domain is explicitly specified, all networking interfaces are shared with the host (all the other interfaces are present too - br0 is mentioned as an example)since no filesystem configuration is specified for the domain, the filesystem is shared with the host– all host mounts are present in the container as well.

Further Information

Libvirt is an open source project and a great deal of technical and usage information is available on the libvirt.org website:

Additional references:

- Architecture: http://libvirt.org/intro.html
- Deployment: http://libvirt.org/deployment.htmlXML
- Format: http://libvirt.org/format.html
- Virsh command reference: http://linux.die.net/man/1/virsh
- User generated content: http://wiki.libvirt.org/page/Main_Page

Mailing Lists. There are three libvirt mailing lists available which can be subscribed to. Archives of the lists are also available:

- https://www.redhat.com/archives/libvir-list
- https://www.redhat.com/archives/libvirt-users
- https://www.redhat.com/archives/libvirt-announce

Annex 1: kvm_virtio_blk.xml

```
<domain type='kvm'>
  <name>kvm_virtio_blk</name>
  <uuid>b8ec80c1-4fd6-4e08-aec7-02150fab316d</uuid>
  <memory unit='KiB'>1048576</memory>
  <currentMemory unit='KiB'>1048576</currentMemory>
  <vcpu placement='static'>2</vcpu>
  <os>
    <type arch='aarch64' machine='virt'>hvm</type>
    <kernel>/boot/Image</kernel>
    <cmdline>root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk</cmdline>
  </os>
  <features>
    <gic version='3'/>
  </features>
  <cpu mode='custom' match='exact'>
    <model fallback='allow'>host</model>
  </cpu>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-aarch64</emulator>
    <disk type='file' device='disk'>
      <driver name='qemu' type='raw'/>
      <source file='/root/ubuntu_bionic_arm64_rootfs.ext4.img'/>
      <target dev='vda' bus='virtio'/>
    </disk>
    <controller type='pci' index='0' model='pcie-root'/>
    <controller type='pci' index='1' model='dmi-to-pci-bridge'/>
    <controller type='pci' index='2' model='pci-bridge'/>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <memballoon model='none'/>
  </devices>
</domain>
```

Annex 2: kvm_virtio_net.xml

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  <name>kvm_virtio_net</name>
  <uuid>5c30747a-a2c9-485e-b814-2a503fef8653</uuid>
  <memory unit='KiB'>1048576</memory>
  <currentMemory unit='KiB'>1048576</currentMemory>
  <vcpu placement='static'>2</vcpu>
  <os>
    <type arch='aarch64' machine='virt'>hvm</type>
    <kernel>/boot/Image</kernel>
    <cmdline>root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk</cmdline>
  </os>
  <features>
    <gic version='3'/>
  </features>
  <cpu mode='custom' match='exact'>
    <model fallback='allow'>host</model>
  </cpu>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-aarch64</emulator>
    <disk type='file' device='disk'>
      <driver name='qemu' type='raw'/>
      <source file='/root/ubuntu_bionic_arm64_rootfs.ext4.img'/>
      <target dev='vda' bus='virtio'/>
    </disk>
    <controller type='pci' index='0' model='pcie-root'/>
    <controller type='pci' index='1' model='dmi-to-pci-bridge'/>
    <controller type='pci' index='2' model='pci-bridge'/>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <memballoon model='none'/>
  </devices>
  <qemu:commandline>
    <qemu:arg value='-netdev'/>
    <qemu:arg value='tap,id=tap0,script=/root/qemu-ifup,downscript=no,ifname=tap0'/>
    <qemu:arg value='-device'/>
    <qemu:arg value='virtio-net-pci,netdev=tap0'/>
  </qemu:commandline>
</domain>
```

Annex 3: kvm_virtio_blk_dataplane.xml

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  <name>kvm_virtio_blk_dataplane</name>
  <uuid>5c30747a-a2c9-485e-b814-2a503fef8657</uuid>
  <memory unit='KiB'>1048576</memory>
  <currentMemory unit='KiB'>1048576</currentMemory>
  <vcpu placement='static'>2</vcpu>
  <os>
    <type arch='aarch64' machine='virt'>hvm</type>
    <kernel>/boot/Image</kernel>
    <cmdline>root=/dev/vda rw console=ttyAMA0 rootwait earlyprintk</cmdline>
  </os>
  <features>
    <gic version='3'/>
  </features>
  <cpu mode='custom' match='exact'>
    <model fallback='allow'>host</model>
  </cpu>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <iothreads>1</iothreads>
  <devices>
    <emulator>/usr/bin/qemu-system-aarch64</emulator>
    <disk type='file' device='disk'>
      <driver name='qemu' type='raw'/>
      <source file='/root/ubuntu_bionic_arm64_rootfs.ext4.img'/>
      <target dev='vda' bus='virtio'/>
    </disk>
    <disk type='file' device='disk'>
      <driver name='qemu' type='raw' cache='none' io='native' iothread='1'/>
      <source file='/root/fake-dev0-backstore.img'/>
      <address type='pci' domain='0x0000' bus='0x00' slot='0x06' function='0x0'/>
      <target dev='vdb' bus='virtio'/>
    </disk>
    <controller type='pci' index='0' model='pcie-root'/>
    <controller type='pci' index='1' model='dmi-to-pci-bridge'/>
    <controller type='pci' index='2' model='pci-bridge'/>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <memballoon model='none'/>
  </devices>
  <qemu:commandline>
    <qemu:arg value='-netdev'/>
    <qemu:arg value='tap,id=tap0,script=/root/qemu-ifup,downscript=no,ifname=tap0'/>
    <qemu:arg value='-device'/>
    <qemu:arg value='virtio-net-pci,netdev=tap0'/>
  </qemu:commandline>
</domain>
```

10.3 Docker Containers

10.3.1 Introduction to Docker Containers

10.3.1.1 Overview

This section is a guide and tutorial to building and using Docker Containers. Docker Containers are only available on ARM64 platforms, with the exception of LS1043 Big Endian.

Docker is a different set of userspace tools implementing Linux containers and focusing on a different set of use cases. The highlights of this open source project are ease of use, shared contributions and fast deployment. In the Docker ecosystem, containers are application environment packages, which can be easily distributed and developed collaboratively, and are guaranteed to be reproducible on any supporting platform, from the development stage to production. Currently, Docker containers are mainly targeting cloud environments.

Docker can be viewed as a set of separate components:

- **Images** - the "build" component of Docker. These are read-only copies of container root filesystems, consisting of the designed application and it's userspace dependencies. For example, an image can contain an Ubuntu application, an Apache server and a user web app. This image can be used to get a webserver running.

- **Registries** - the "distribution" component of Docker. These are public or private stores where users can upload / download images. The images are versioned, and are built from layers. When sharing images, the layers are first downloaded separately, and the image is assembled at runtime. Each layer corresponds to a specific user commit. Images can also be built using buildfiles. The most representative registry example is the Docker Hub. The current Docker installation does not support registry configuration.
- **Containers** - the "run" component of Docker. These are very similar to the containers provided by the LXC package. The main difference is that Docker containers use an overlay filesystem as container support. The layers are taken as is from the image and marked read-only, with a topmost read-write layer on top. This means that no container makes any persistent changes to the image by default - these need to be explicitly committed by the user when the environment is in the desired state. Docker containers are designed to work as application containers by default.

Docker uses a client-server architecture. The client takes the user commands and talks to a daemon, which does the entire container management work. A Linux host running the daemon is called a Docker Host. The client and daemon can run on the same machine, or on different ones, communicating through sockets or a RESTful API.

The Docker official page advertises a set of use cases, mostly relevant in cloud environments: continuous integration, continuous delivery, devops, big data and infrastructure optimization. These can be easily adapted to embedded distributions as well. As for the containers themselves, the Linux Containers chapter use cases apply, with a focus on ease of use, fast deployment and distributed usage.

10.3.2 Docker How To's

10.3.2.1 Running a webserver container

The following article describes the necessary steps to deploy a web server service using a Docker container. This is based on downloading a prepared image from the Docker hub and using it to start a container.

1. Verify if the docker daemon is running. Make sure that the board has Internet access - this will be required to download the image from the Docker Hub. The daemon will configure a Linux bridge for the containers with a private network and NAT. One can verify if the docker daemon is running by using one of the following commands:

   ```
   $ docker info
   $ docker version
   ```

   In this case, the docker daemon is configured to start at boot time, but if for any reason the daemon is not running just issue the following command:

   ```
   root@localhost:~# dockerd
   ```

2. You can search the registry for available **arm64** images, or using any other keyword.

   ```
   root@localhost:~# docker search arm64
   NAME                       DESCRIPTION                     STARS      OFFICIAL
   AUTOMATED

   ericvh/arm64-ubuntu                 Base image for arm64 (armv8 aka aarch64) U...  6
   owlab/alpine-arm64              This is Alpine Linux for arm64 (or aarch64)    3
   necrose99/gentoo-arm64              Arm64 with qemu-arm64 static AMD64 host h...
   1            [OK]
   mickaelguene/arm64-debian           Arm64 debian base with umeq install so you...
   1            [OK]
   markusk/arm64-crosscompile          A debian image with the necessary tools in...
   1            [OK]
   snapcraft/zesty-arm64               Docker image for building Ubuntu snaps
   0            [OK]
   mickaelguene/arm64-debian-jenkins-slave   arm64 with java and sshd with umeq so
   you ...  0            [OK]
   containerstack/alpine-arm64            Alpine Linux (arm64/aarch64) Docker image
   0            [OK]
   arm64el/helloworld-arm64el          hello world for arm64 el platform
   0            [OK]
   arm64el/busybox-arm64el             busybox image for arm64               0
   ```

```
[OK]
eqw3rty/minecraft-server-arm64         Dockerized Minecraft server for arm64
0            [OK]
arm64el/unshare-arm64el             unshare image for arm64el platform
0            [OK]
mickaelguene/arm64-debian-dev arm64       debian images with development tool ...
0            [OK]
necrose99/gentoo-arm64-chroot          base Gentoo AMD64 + ARM64 CHROOT volume. ...
0            [OK]
marcust/jessie-arm64-rust           Debian Jessie (arm64) image containing a R...  0
ip4368/node-arm64                Node.js is a JavaScript-based platform for...  0
marcust/bionic-arm64-rust           Ubuntu bionic (arm64) image containing a R...  0
snapcraft/bionic-arm64              Docker image for building Ubuntu snaps
0            [OK]
jefby/arm64                   arm64 develop                      0
dil001/nginx-arm64               These are the arm64 version of the officia...  0
knjcode/arm64-node               arm64-compatible Docker base image with No...  0
parity/rust-arm64                RUST for GitLab CI runner (ARM64 architect...
0            [OK]
thenatureofsoftware/mc-arm64           Minio client for arm64                0
thenatureofsoftware/ubuntu-arm64       Ubuntu slim images for arm64             0
dil001/fluentd-arm64             arm64 fork of the offical docker images     0
```

3. In this example, `qoriq/arm64-ubuntu` is used. It is a standard Ubuntu compiled for ARM64, with a `lighttpd` webserver installed and with a home page configured to display some information on the board, processes and networking in the container. First download the image.

```
root@localhost:~# docker pull qoriq/arm64-ubuntu
Using default tag: latest
latest: Pulling from qoriq/arm64-ubuntu
a3ed95caeb02: Pull complete
9025035f8d16: Pull complete
d54663dfcaf9: Pull complete
b940f6a4f33c: Pull complete
688957367bc4: Pull complete
88ca67eab938: Pull complete
f5f1c1a40562: Pull complete
688957367bc4: Pull complete
88ca67eab938: Pull complete
f5f1c1a40562: Pull complete
357cdf8f1a01: Pull complete
de8e5d34ebd8: Pull complete
811aa6d4eba3: Pull complete
0dc75b6c54d0: Pull complete
654cadd8a53b: Pull complete
40d300e17719: Pull complete
ce42abd87d1e: Pull complete
Digest: sha256:eaef3a08336f59155e6cfb61bf55688711214561ddf00817b5c848211ac66b00
Status: Downloaded newer image for qoriq/arm64-ubuntu:latest
```

You can check the image is available using `docker images`:

```
root@localhost:~# docker images
REPOSITORY        TAG     IMAGE ID     CREATED     SIZE
qoriq/arm64-ubuntu      latest     903eaef3b724    12 months ago  326.4 MB
root@localhost:~#
```

4. Start a container using the following command:

```
root@localhost:~# docker run -d -p 30081:80 --name=sandbox1 \
-h sandbox1 qoriq/arm64-ubuntu \
bash -c "lighttpd -f /etc/lighttpd/lighttpd.conf -D"
```

- `run` - create and start the container. Optionally, download the image if not available on the host.
- `-d` - start the container as a daemon.
- `-p 30081:80` - forward port 80 in the container to port 30081 on the board.
- `--name=sandbox1` - the name of the container (as visible to Docker).

- `-h sandbox1` - the hostname inside the container.
- `qoriq/arm64-ubuntu` - the base image for the container.
- `bash -c "lighttpd -f /etc/lighttpd/lighttpd.conf -D"` - the command to execute as PID 1 in the container.

The command will return a unique SHA for the container. You can check that the webserver is up and running by accessing `http://BOARD_IP:30081/` from a browser. You can also check the container is running using docker:

```
root@localhost:~# docker ps -a

CONTAINERID     IMAGE         COMMAND             CREATED     STATUS
PORTS        NAMES
b5b8a45db81c    qoriq/arm64-ubuntu  "bash -c 'lighttpd -f"    16 hours ago   Exited
(0)16 hours ago                sandbox1
```

5. Stopping and deleting the container are easy operations:

```
root@localhost:~# docker stop sandbox1
sandbox1
root@localhost:~# docker rm sandbox1
sandbox1
```

6. A similar command can be used to delete the image from the board.

```
root@localhost:~# docker rmi qoriq/arm64-ubuntu
Untagged: qoriq/arm64-ubuntu:latest
Untagged: qoriq/arm64-
ubuntu@sha256:eaef3a08336f59155e6cfb61bf55688711214561ddf00817b5c848211ac66b00
Deleted: sha256:903eaef3b7240612111df4308f4d598ae1dee14b696a4b01654175b6771520f1
Deleted: sha256:48e73c491543279a59d202470394f0f91acd9b3a8a6f5f9befa933bc4cf4776a
Deleted: sha256:e21b9d6aa0007e242abb10948b13c93e4471694695a91a47d639f45927f25eb6
Deleted: sha256:7ec2184e81ef396a206e965e6dae42a122c4348dd7cfee1b731aa59931a5ec82
Deleted: sha256:0b081c8c711c2d14522ea1b5763e5ead19ab2975e4c28864a0ee2c0942ebae43
Deleted: sha256:b256d9ce72b40a1dc9dfdb13003a44976ba81e4fb31e774e913ed57241424231
Deleted: sha256:e07c8e0adb08295db7e3f2e13f41be622d5b8590575f87813922dd4ef0914e8f
Deleted: sha256:09ec9672e9e6d30855f1274415edf6a023b86764261b6cd88fc2b692f997977d
Deleted: sha256:d29d57006e3df9a03fb3d430183166c9337378404c1ad66db391251ea24592fd
Deleted: sha256:84be8839209cbbecd3b3f064b9593e16d30468d71c788fc3ab8f3125990002bf
Deleted: sha256:09be261c306e6c01756d16c31e2a9d4b638e8d205a068b767cb0a078480633a9
Deleted: sha256:47d9e04c91309d23f8135f579a302c2309b206cb392c42c55ec13b2c26fb317f
Deleted: sha256:8495eed3352e7d2a237f179e3a3a6e449a56821a77e2efd943bc9ccf8d6d964c
Deleted: sha256:423a2c50f96dad2f267bbbe11a8a9efc21e776419fbd618ec1a9a21e918c918b
Deleted: sha256:67629909bfc67e60ba87451caf1f98b375e8b81f21a87bab5f5e2740a78c025b
Deleted: sha256:f821f1edfff4c38033e84024e844e503d5e0e470155c4bd69ec3f0af04f01b6b
Deleted: sha256:837a3e2cff861610e7672192dac0342041c30b2548a3a63a47b92d964a862c8a
Deleted: sha256:129149fe5b4dc97f940c38cd37cfa3fc06bbdc12a8d9d22e4aa3b3e4ff709346
```

Chapter 11
Power Management

11.1 Power Management User Manual

Linux SDK for QorIQ Processors

Description

QorIQ Processors have features to minimize power consumption at several different levels. All processors support a sleep mode (LPM20). Some processors, such as T1040, LS1021, LS1046, also support a deep sleep mode (LPM35).

The following power management features are supported on various QorIQ processors:

- Dynamic power management
- Shutting down unused IP blocks
- Cores support low power modes (such as PW15)
- Processors enter low power state (LPM20, LPM35)
  - — LPM20 mode: most part of processor clocks are shut down
  - — LPM35 mode: power is removed to cores, cache and IP blocks of the processor such as DIU, eLBC, PEX, eTSEC, USB, SATA, eSDHC etc.
- CPU hotplug: If cores are down at runtime, they will enter low power state.

The wake-up event sources caused quitting from low power mode are listed as below:

- Wake on LAN (WoL) using magic packet
- Wake by MPIC timer or FlexTimer
- Wake by Internal and external interrupts

For more information on a specific processor, refer to processor Reference Manual.

Kernel Configure Tree View Options

For ARM platforms

Kernel Configure Tree View Options	Description

`Power management options  -->`
`  [*] Suspend to RAM and standby`
	Enable sleep feature

`Device Drivers  --->`
`    SOC (System On Chip) specific Drivers  --->`
`    [*]   Layerscape Soc Drivers`
`    [*]     FTM alarm driver`
	Enable the FTM alarm (FlexTimer module) driver

`CPU Power Management  --->`
`CPU Idle  --->`
	Enable the CPU Idle driver

Table continues on the next page...

Table continued from the previous page...

Kernel Configure Tree View Options	Description

`[*] CPU idle PM support`
`[*]   Ladder governor (for periodic timer tick)`
`-*-   Menu governor (for tickless system)`
`      ARM CPU Idle Drivers  --->`
`        [*] Generic ARM/ARM64 CPU idle Driver`	

Table continues on the next page...

Compile-time Configuration Options

Linux Framework	Hardware Feature	Platform	Kernel Config
Suspend	LPM20	LS1012, LS1021, LS1046	CONFIG_SUSPEND
	wake by Flextimer	LS1012, LS1021, LS1046	CONFIG_FTM_ALARM
CPU idle	PW15	LS1012, LS1021, LS1046	CONFIG_ARM_CPUIDLE

Device Tree Binding

Property	Type	Description
fsl,#rcpm-wakeup-cells	unsigned int	the number of cells in "rcpm-wakeup" except the pointer to "rcpm"
rcpm-wakeup	unsigned int	require if the IP block can work as a wakeup source

For processors integrated RCPM

```
    rcpm: rcpm@1ee2000 {
        compatible = "fsl,ls1046a-rcpm", "fsl,qoriq-rcpm-2.1";
        reg = <0x0 0x1ee2000 0x0 0x1000>;
        fsl,#rcpm-wakeup-cells = <1>;
    };

    ftm0: ftm0@29d0000 {
        compatible = "fsl,ftm-alarm";
        reg = <0x0 0x29d0000 0x0 0x10000>;
        interrupts = <0 86 0x4>;
        big-endian;
        rcpm-wakeup = <&rcpm 0x0 0x20000000>;
        status = "okay";
    };
```

Refer to the Linux document: Documentation/devicetree/bindings/soc/fsl/rcpm.txt

Source Files

The source files are maintained in the Linux kernel source tree.

Source File	Description
drivers/soc/fsl/layerscape/rcpm.c	the RCPM driver needed by the sleep feature
drivers/soc/fsl/layerscape/ftm_alarm.c	the FTM timer driver worked as a wakeup source
drivers/cpuidle/cpuidle-arm.c	the cpuidle driver for ARM core

Verification in Linux

- Cpuidle Driver

  The cpuidle driver can switch CPU state according to the idle policy (governor). For more information, please see "Documentation/cpuidle/sysfs.txt" in kernel source code.

  ```
  /* Check the cpuidle driver which is currently used. */
  # cat /sys/devices/system/cpu/cpuidle/current_driver

  /* Check the following directory to see the detailed statistic information of each state
  on each CPU. */
  /sys/devices/system/cpu/cpu0/cpuidle/state0/
  /sys/devices/system/cpu/cpu0/cpuidle/state1/
  ```

Supporting Documentation

- QorIQ processor reference manuals

11.2 CPU Frequency Switching User Manual

Linux SDK for QorIQ Processors

Abbreviations and Acronyms

DFS: Dynamic Frequency Scaling

Description

QorIQ Processors support DFS (Dynamic Frequency Switching) feature, also known as CPU Frequency Switch, which can change the frequency of cores dynamically.

For more information on a specific processor, refer to processor Reference Manual.

<table>
<tr><th>Kernel Configure Tree View Options</th><th>Description</th></tr>
<tr><td><pre>
CPU Power Management -->
  CPU Frequency scaling -->
    [*] CPU Frequency scaling
    <*> CPU frequency translation statistics
        Default CPUFreq governor (userspace) -->
    -*- 'userspace' governor for userspace frequency scaling
        ARM CPU frequency scaling drivers -->
           <*> CPU frequency scaling driver for Freescale QorIQ
SoCs
</pre></td><td>Enable the CPU frequency driver</td></tr>
</table>

Compile-time Configuration Options

Linux Framework	Hardware Feature	Platform	Kernel Config
cpufreq	DFS	ALL	CONFIG_CPU_FREQ, CONFIG_CPU_FREQ_DEFAULT_GOV_USERSPACE
cpufreq	DFS	Layerscape	CONFIG_QORIQ_CPUFREQ

User Space Application

Simply using command "cat" and "echo" can verify this feature.

Device Tree Binding

Property	Type	Status	Description
#clock-cells	unsigned int	Required	The number of cells in a clock-specifier
clocks	handle	Required	Clock source handle
compatible	String	Required	Compatible strings
reg	unsigned int	Required	register address range

Table continues on the next page...

```
clockgen: clocking@1ee1000 {
    compatible = "fsl,ls1012a-clockgen";
    reg = <0x0 0x1ee1000 0x0 0x1000>;
    #clock-cells = <2>;
    clocks = <&sysclk>;
};
```

Source Files

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

Table continued from the previous page...

Source File	Description
drivers/cpufreq/qoriq_cpufreq.c	CPU frequency scaling driver for qoriq chips

Verification in Linux

- CPU frequency mode

```
In order to test the CPU frequency scaling feature, we need to enable the CPU frequency 
feature on the
menuconfig and choose the USERSPACE governor.
You can learn more about CPU frequency scaling feature by referring to the kernel 
documents.
They all are put under Documentation/cpu-freq/ directory.
For example: all the information about governors is put in Documentation/cpu-freq/
governors.txt.
```

```
Test step:
1. list all the frequencies a core can support (take cpu 0 for example) :
 # cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_frequencies
1199999 599999 299999 799999 399999 199999 1066666 533333 266666

2.  check the CPU's current frequency
# cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq
1199999

3. change the CPU's frequency we expect:
# echo 799999 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed

You can check the CPU's current frequency again to confirm if the frequency transition is
successful.

Please note that if the frequency you want to change to doesn't support by current CPU,
kernel will
round up or down to one CPU supports.
```

11.3 Thermal Management User Manual

Description

The thermal management function is based on TMU (Thermal Monitoring Unit).

The driver sets two thresholds for management function. If the CPU temperature crosses the first one (75 C for LS2080. 85 C for other platforms), the driver will trigger CPU frequency limitation auto-scaling according to the temperature trend; If the CPU temperature crosses the second one (85 C for LS2080, 95 C for other platforms, critical for core) the driver will shut down the system.

User could also get current temperature through sysfs interface.

Specifications

Target boards:	T1040RDB, T1042RDB, T1023RDB, T1024RDB, LS1021ATWR, LS1043ARDB, LS2080ARDB.
Operating system:	Linux 3.12+

Kernel Configure Tree View Options (For PowerPC platform)

Kernel Configure Tree View Options	Description

`Platform support  --->`
`    CPU Frequency scaling  --->`
`        PowerPC CPU frequency scaling drivers  --->`
`            <*> CPU frequency scaling driver for NXP QorIQ SoCs`
	Enable CPUfreq driver.

`Device Drivers  --->`
`    [*] Generic Thermal sysfs driver  --->`
`        [*] generic cpu cooling support`
`        [*] Freescale QorIQ Thermal Monitoring Unit`
	Enable thermal management framework, cpu cooling device support and QorIQ thermal driver.

Kernel Configure Tree View Options (For ARM platform)

Kernel Configure Tree View Options	Description

`CPU Power Management  --->`
`    CPU Frequency scaling  --->`
`        ARM CPU frequency scaling drivers  --->`
`            <*> CPU frequency scaling driver for NXP QorIQ SoCs`
	Enable CPUfreq driver.

`Device Drivers  --->`
`    [*] Generic Thermal sysfs driver  --->`
`        [*] generic cpu cooling support`
`        [*] Freescale QorIQ Thermal Monitoring Unit`
	Enable thermal management framework, cpu cooling device support and QorIQ thermal driver.

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_QORIQ_CPUFREQ	y/n	n	Enable QorIQ CPUfreq driver
CONFIG_THERMAL	y/m/n	n	Enable thermal management support
CONFIG_CPU_THERMAL	y/m/n	n	Enable cpu cooling device support
CONFIG_QORIQ_THERMAL	y/m/n	n	Enable QorIQ thermal driver

Device Tree Binding

```
tmu: tmu@f0000 {
        compatible = "fsl,qoriq-tmu";
        reg = <0xf0000 0x1000>;
        interrupts = <18 2 0 0>;
        fsl,tmu-range = <0x000a0000 0x00090026 0x0008004a 0x0001006a>;
        fsl,tmu-calibration = <0x00000000 0x00000025
                0x00000001 0x00000028
                0x00000002 0x0000002d
                0x00000003 0x00000031
                0x00000004 0x00000036
                0x00000005 0x0000003a
                0x00000006 0x00000040
                0x00000007 0x00000044
                0x00000008 0x0000004a
                0x00000009 0x0000004f
                0x0000000a 0x00000054

                0x00010000 0x0000000d
                0x00010001 0x00000013
                0x00010002 0x00000019
                0x00010003 0x0000001f
                0x00010004 0x00000025
                0x00010005 0x0000002d
                0x00010006 0x00000033
                0x00010007 0x00000043
                0x00010008 0x0000004b
                0x00010009 0x00000053

                0x00020000 0x00000010
                0x00020001 0x00000017
                0x00020002 0x0000001f
                0x00020003 0x00000029
                0x00020004 0x00000031
                0x00020005 0x0000003c
                0x00020006 0x00000042
                0x00020007 0x0000004d
                0x00020008 0x00000056

                0x00030000 0x00000012
                0x00030001 0x0000001d>;
};
```

Source Files

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

Source File	Description
drivers/thermal/qoriq_thermal.c	QorIQ thermal driver.

Verification in Linux

There are two parts for verification: management and monitor.

[Management:]

1. When CPU temperature cross the first threshold, CPU frequency may be reduced by changing frequency limitation, use the following command to check the current frequency:

```
~$ cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_cur_freq
```

2. When CPU temperature cross the first threshold, system will shutdown.

[Monitor:]

```
You can manually read the thermal interfaces in sysfs:
~$ cat /sys/class/thermal/thermal_zone0/temp
35000
# This means the current temperature is 35 C.
```

11.4 System Monitor

11.4.1 Power Monitor User Manual

Description

There are two methods currently we can use to measure the power consumption which are called online and offline power monitoring respectively. The difference between them is that offline power monitoring support measuring power consumption during sleep or deep sleep.

The Power Monitor can be supported on P4080DS/P5020DS/P5040DS/T4240QDS/LS1043QDS/LS1046QDS/LS1088QDS/LS2088QDS board.

This User guide uses the LS240QDS board as an example.

Online Power Monitoring

The Lm-sensors tool (download from http://dl.lm-sensors.org/lm-sensors/releases) will be used to read the power/temperature from on-boards sensors. The drivers vary from sensor to sensor. Basically they would be INA220, ZL6100 and ADT7461 etc.

The device driver support either a built-in kernel or module loading.

Kernel Configure Tree View Options

Option	Description

`Device Drivers --->`
` <*> Hardware Monitoring support  --->`
`    <*>   Texas Instruments INA219 and compatibles`
	Enables INA220

`Device Drivers --->`
`    [*]   Enable compatibility bits for old user-space`
`    <*>   I2C device interface`
`    [*]   Autoselect pertinent helper modules`
`          I2C Hardware Bus support  --->`
`              <*> MPC107/824x/85xx/512x/52xx/83xx/86xx`
	Enables I2C block device driver support

`Device Drivers --->`
`      <*>   I2C bus multiplexing support`
`          Multiplexer I2C Chip support  --->`
`              <*> Philips PCA954x I2C Mux/switches`
	Enables I2C bus multiplexing PCA9547

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_I2C_MPC	y/n	y	Enable I2C bus protocol
SENSORS_INA2XX	y/n	y	Enables INA220
CONFIG_I2C_MUX_PCA954x	y/n	y	Enables I2C multiplexing PCA9547

Device Tree Binding

Property	Type	Status	Description
compatible	String	Required	"Philips,pca9547" for pca9547
reg	integer	Required	reg = <0x77>
compatible	String	Required	"ti,ina220" for ina220
reg	integer	Required	reg = <the i2c address of ina220>

```
Default node:
       i2c@118000 {
            pca9547@77 {
                compatible = "philips,pca9547";
                reg = <0x77>;
                #address-cells = <1>;
                #size-cells = <0>;

                channel@2 {
                    #address-cells = <1>;
                    #size-cells = <0>;
                    reg = <0x2>;

                    ina220@40 {
                        compatible = "ti,ina220";
                        reg = <0x40>;
                        shunt-resistor = <1000>;
                    };

                    ina220@41 {
                        compatible = "ti,ina220";
                        reg = <0x41>;
                        shunt-resistor = <1000>;
                    };

                    ina220@44 {
                        compatible = "ti,ina220";
                        reg = <0x44>;
                        shunt-resistor = <1000>;
                    };

                    ina220@45 {
                        compatible = "ti,ina220";
                        reg = <0x45>;
                        shunt-resistor = <1000>;
                    };

                    ina220@46 {
                        compatible = "ti,ina220";
                        reg = <0x46>;
                        shunt-resistor = <1000>;
                    };

                    ina220@47 {
                        compatible = "ti,ina220";
                        reg = <0x47>;
                        shunt-resistor = <1000>;
                    };
                };
            };
```

Source Files

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

Source File	Description
drivers/i2c/muxes/i2c-mux-pca954x.c	PCA9547 driver
drivers/hwmon/ina2xx.c	ina220 driver

Test Procedure

Do the following to validate under the kernel

1. The bootup information is displayed:

```
......
i2c /dev entries driver
mpc-i2c ffe118000.i2c: timeout 1000000 us
mpc-i2c ffe118100.i2c: timeout 1000000 us
mpc-i2c ffe119000.i2c: timeout 1000000 us
mpc-i2c ffe119100.i2c: timeout 1000000 us
i2c i2c-0: Added multiplexed i2c bus 6
i2c i2c-0: Added multiplexed i2c bus 7
i2c i2c-0: Added multiplexed i2c bus 8
i2c i2c-0: Added multiplexed i2c bus 9
i2c i2c-0: Added multiplexed i2c bus 10
i2c i2c-0: Added multiplexed i2c bus 11
i2c i2c-0: Added multiplexed i2c bus 12
i2c i2c-0: Added multiplexed i2c bus 13
pca954x 0-0077: registered 8 multiplexed busses for I2C mux pca9547
ina2xx 8-0040: power monitor ina220 (Rshunt = 1000 uOhm)
ina2xx 8-0041: power monitor ina220 (Rshunt = 1000 uOhm)
ina2xx 8-0045: power monitor ina220 (Rshunt = 1000 uOhm)
ina2xx 8-0046: power monitor ina220 (Rshunt = 1000 uOhm)
ina2xx 8-0047: power monitor ina220 (Rshunt = 1000 uOhm)
ina2xx 8-0044: power monitor ina220 (Rshunt = 1000 uOhm)
......
```

```
root@LS1046ARDB:~# sensors
ina220-i2c-0-40
```

```
Adapter: 2180000.i2c
in0:          +0.01 V
in1:          +1.04 V
power1:        6.82 W
curr1:        +6.48 A

adt7461-i2c-0-4c
Adapter: 2180000.i2c
temp1:        +29.0°C  (low  =  +0.0°C, high = +85.0°C)
                       (crit = +85.0°C, hyst = +75.0°C)
temp2:        +47.8°C  (low  =  +0.0°C, high = +85.0°C)
                       (crit = +85.0°C, hyst = +75.0°C)
```

NOTE

Please make sure to include the "sensors" command in your rootfs

11.4.2 Thermal Monitor User Manual

Description

The Temperature Monitoring function is provided by the chip ADT7461.

This driver exports the values of Temperature to SYSFS. The user space lm-sensors tools can get and display these values.

Kernel Configure Tree View Options

Kernel Configure Tree View Options	Description

`Device Drivers  --->`
`    [*] Hardware Monitoring support  --->`
`        [*] National Semiconductor LM90 and compatibles`
	Enable thermal monitor chip driver like ADT7461.

`Device Drivers  --->`
`    <*> I2C bus multiplexing support  --->`
`        Multiplexer I2C Chip support  --->`
`            <*> Philips PCA954x I2C Mux/switches`
	Enable I2C PCA954x multiplexer support

Compile-time Configuration Options

Option	Values	Default Value	Description
CONFIG_HWMON	y/m/n	n	Enable Hardware Monitor
CONFIG_SENSORS_LM90	y/m/n	n	Enable ATD7461 driver
CONFIG_I2C_MUX	y/m/n	n	Enable I2C bus multiplexing support
CONFIG_I2C_MUX_PCA954x	y/m/n	n	Enable PCA954x driver

Device Tree Binding

```
adt7461@4c {
    compatible = "adi,adt7461";
    reg = <0x4c>;
};

pca9547@77 {
    compatible = "philips,pca9547";
    reg = <0x77>;
};
```

Source Files

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

Source File	Description
drivers/hwmon/hwmon.c	Linux hwmon subsystem support
drivers/hwmon/lm90.c	ADT7461 chip driver
drivers/i2c/i2c-mux.c	I2C bus multiplexing support
drivers/i2c/muxes/pca954x.c	PCA954x chip driver

Verification in Linux

There are two ways to get temperature results.

```
1. You can manually read the thermal interfaces in sysfs:
~$ ls /sys/class/hwmon/hwmon1/devices
alarms            temp1_crit        temp1_min_alarm   temp2_max_alarm
driver            temp1_crit_alarm  temp2_crit        temp2_min
hwmon             temp1_crit_hyst   temp2_crit_alarm  temp2_min_alarm
modalias          temp1_input       temp2_crit_hyst   temp2_offset
name              temp1_max         temp2_fault       uevent
power             temp1_max_alarm   temp2_input       update_interval
subsystem         temp1_min         temp2_max

~$ cat /sys/class/hwmon/hwmon1/devices/temp1_input
29000

2. You can use lm_sensors tools as follows.
~ # sensors

adt7461-i2c-1-4c
Adapter: MPC adapter
temp1:       +34.0 C  (low  =  +0.0 C, high = +85.0 C)
                      (crit = +85.0 C, hyst = +75.0 C)
temp2:       +48.5 C  (low  =  +0.0 C, high = +85.0 C)
                      (crit = +85.0 C, hyst = +75.0 C)
```

"lm_sensors is integrated into rootfs file system by default. If there is no "sensors" command in your rootfs just add lmsensors-sensors package and build your own rootfs."

Chapter 12 Benchmarking guidelines

12.1 Coremark

12.1.1 Test Environment

Objectives

The Coremark benchmaring guideline aims to do the following:

- Baseline the Coremark performance on QorIQ Layerscape platforms
- Identify any optimizations and ensure they are implemented on the QorIQ Layerscape platforms.
- Investigate other changes that may improve performance

Hardware Platform Identification

Board	Silicon Revision	Default Freqeuncy(Core/CCB/DDR) in MHz	Core Type
LS1021ATWR	Rev2.0	1000/300/1600	cortex A7
LS1043ARDB	Rev1.1	1600/400/1600	cortex A53
LS1046ARDB	Rev1.0	1800/700/2100	cortex A72
LS1088ARDB	Rev1.0	1600/700/2100	cortex A53
LS2088ARDB	Rev1.0	2000/800/2133	cortex A72

For more information on each boards switch settings, refer to the boards's Reference Manual or Getting Started Guide on http://www.nxp.com/

Software Platforma Identification

All software was built from Layerscape SDK.

Boot Loader

U-boot 2017.03 with NXP-specific patches on top.

Coremark Application

- **Source Code Download:**

  Coremark Source code can be downloaded from http://www.eembc.org/coremark/index.php

  Toolchain version is gcc-5.4 with glibc-2.23

- **Build Coremark**

  1. If you are compiling Coremark on a 64-bit Linux machine (machine on which you have the intended compiler), go to `coremark_v1.0/linux64` directory, else go to `coremark_v1.0/linux directory`.

2. Give the complete compiler path under "CC" flag in `core_portme.mak` file.
3. For a 32-bit ARM platform, the toolchain path is `/usr/bin/arm-linux-gnueabihf-gcc` by default. Change the “CC” flag in `linux/core_portme.mak` as the following:

   ```
   CC = /usr/bin/arm-linux-gnueabihf-gcc
   ```
4. For a 64-bit ARM platform, the toolchain path is `/usr/bin/aarch64-linux-gnu-gcc`. Change the “CC” flag in `linux64/core_portme.mak` as the following:

   ```
   CC = /usr/bin/aarch64-linux-gnu-gcc
   ```
5. Go back to the `coremark_v1.0` directory. Perform the following commands on the command line:

   — For a 64-bit ARM platform (LS1043A/LS1046A/LS1088A/LS2088A):

   Single Thread:

   ```
   make PORT_CFLAGS="-O3 -funroll-all-loops --param max-inline-insns-auto=550"
   ```

   Multithread:

   ```
   make PORT_CFLAGS="-O3 -funroll-all-loops --param max-inline-insns-auto=550 -
   DMULTITHREAD=<Thread_number> -DUSE_FORK=1"
   ```

   — For a 32-bit ARM platform (LS1021A and LS1043A/LS1046A 32-bit)

   Single Thread:

   ```
   make PORT_CFLAGS="-O3 -march=armv7-a -mfloat-abi=hard -mfpu=neon -mtune=cortex-a7
   -funroll-all-loops --param max-inline-insns-auto=300 -static"
   ```

   Multithread:

   ```
   make PORT_CFLAGS="-O3 -march=armv7-a -mfloat-abi=hard -mfpu=neon -mtune=cortex-a7
   -funroll-all-loops --param max-inline-insns-auto=300 -static -
   DMULTITHREAD=<Thread_number> -DUSE_FORK=1"
   ```
6. The command will first compile, generate the executable file (`coremark.exe`) and try to run the benchmark. Transfer the executable file to the target.

12.1.2 Test Procedure

Running test and result collection

1. Deploy the target board with corresponding software mentioned in hte previous section.
2. Put coremark binary compiled with optimized flags mentioned in section test environment on target board
3. Run the benchmark:

   ```
   coremark.exe
   ```

Check the log below for the results:

```
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 16663
Total time (secs): 16.663000
Iterations/Sec   : 6601.452320
Iterations       : 110000
Compiler version : GCC5.4.0 20160609
Compiler flags   : -mcpu=cortex-a53 -O3 -funroll-all-loops --param max-inline-insns-auto=550
-DPERFORMANCE_RUN=1  -lrt
Memory location  : Please put data memory location here
                        (e.g. code in flash, data on heap etc)
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0x33ff
Correct operation validated. See readme.txt for run and reporting rules.
```

```
CoreMark 1.0 : 6601.452320 / GCC5.4.0 20160609 -mcpu=cortex-a53 -O3 -funroll-all-loops --param
max-inline-insns-auto=550 -DPERFORMANCE_RUN=1  -lrt / Heap
```

This test measures the variation of the benchmark results, so an average across 5 runs was taken for every result.

12.2 Dhrystone

12.2.1 Test Environment

Objectives

The Dhrystone benchmaring guideline aims to do the following:

- Baseline the Dhrystone performance on QorIQ Layerscape platforms.
- Identify any optimizations and ensure they are implemented on the QorIQ Layerscape platforms.
- Investigate other changes that may improve performance.

Hardware Platform Identification

Board	Silicon Revision	Default Freqeuncy(Core/CCB/DDR) in MHz	Core Type
LS1021ATWR	Rev2.0	1000/300/1600	cortex A7
LS1043ARDB	Rev1.1	1600/400/1600	cortex A53
LS1046ARDB	Rev1.0	1800/700/2100	cortex A72
LS1088ARDB	Rev1.0	1600/700/2100	cortex A53
LS2088ARDB	Rev1.0	2000/800/2133	cortex A72

For more information on each boards switch settings, refer to the boards's Reference Manual or Getting Started Guide on http://www.nxp.com/

Software Platforma Identification

All software was built from Layerscape SDK.

Boot Loader

U-boot 2017.03 with NXP-specific patches on top.

Dhrystone Application

Dhrystone is a synthetic computing benchmark program intended to be representative of system (integer) programming. It is a simple program that is carefully designed to statistically mimic the processor usage of some common set of programs. It also has some pitfalls, for the performance will be affected by many factors such as the compiler, libraries etc.

- **Source Code Download:**

  Dhrystone 1.0 Source code can be downloaded from: http://www.xanthos.se/~joachim/vaxmips.html (`Dhrystone-src.tar.gz`)

  Toolchain version is `gcc-5.4 with glibc-2.23.`

- **Build Dhrystone**
  1. Download the source code from http://www.xanthos.se/~joachim/vaxmips.html `Dhrystone-src.tar.gz`
  2. Unpack the package
  3. Go back to the `dhrystone_v1.0` directory and build dhrystone binary
     - — For a 64-bit ARM platform:

       ```
       #/usr/bin/aarch64-linux-gnu-gcc -O3 -funroll-all-loops --param max-inline-insns-
       auto=550 -static  dhry21a.c dhry21b.c timers.c -o dhrystone
       ```
     - — For 32-bit ARM platform:

       ```
       # /usr/bin/arm-linux-gnueabihf-gcc -O3 -funroll-all-loops --param max-inline-
       insns-auto=550 -static  dhry21a.c dhry21b.c timers.c -o dhrystone
       ```

12.2.2 Test Procedure

Running test and result collection

1. Deploy the target board with corresponding software mentioned in hte previous section.
2. Put Dhrystone binary compiled with optimized flags mentioned in section 2.2.2.3 on target board
3. Run the benchmark:

   ```
   echo 50000000 | ./dhrystone
   ```

Check the log below for the results:

```
Dhrystone Benchmark, Version 2.1 (Language: C)

Please give the number of runs through the benchmark:
Execution starts, 50000000 runs through Dhrystone
Execution ends

Final values of the variables used in the benchmark:

Int_Glob:            5
        should be:   5
Bool_Glob:           1
        should be:   1
Ch_1_Glob:           A
        should be:   A
Ch_2_Glob:           B
        should be:   B
Arr_1_Glob[8]:       7
        should be:   7
Arr_2_Glob[8][7]:    50000010
        should be:   Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:          855702608
        should be:   (implementation-dependent)
  Discr:             0
        should be:   0
  Enum_Comp:         2
        should be:   2
  Int_Comp:          17
        should be:   17
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:          855702608
        should be:   (implementation-dependent), same as above
  Discr:             0
        should be:   0
  Enum_Comp:         1
        should be:   1
```

```
  Int_Comp:          18
        should be:   18
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:           5
        should be:   5
Int_2_Loc:           13
        should be:   13
Int_3_Loc:           7
        should be:   7
Enum_Loc:            1
        should be:   1
Str_1_Loc:           DHRYSTONE PROGRAM, 1'ST STRING
        should be:   DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:           DHRYSTONE PROGRAM, 2'ND STRING
        should be:   DHRYSTONE PROGRAM, 2'ND STRING

Register option selected?  YES
Microseconds for one run through Dhrystone:      0.1
Dhrystones per Second:                       8519121.2
VAX MIPS rating =   4848.675

[root@ls1043agw ~]$ echo 50000000|./dhry21

Dhrystone Benchmark, Version 2.1 (Language: C)

Please give the number of runs through the benchmark:
Execution starts, 50000000 runs through Dhrystone
Execution ends

Final values of the variables used in the benchmark:

Int_Glob:            5
        should be:   5
Bool_Glob:           1
        should be:   1
Ch_1_Glob:           A
        should be:   A
Ch_2_Glob:           B
        should be:   B
Arr_1_Glob[8]:       7
        should be:   7
Arr_2_Glob[8][7]:    50000010
        should be:   Number_Of_Runs + 10
Ptr_Glob->
  Ptr_Comp:          1046248528
        should be:   (implementation-dependent)
  Discr:             0
        should be:   0
  Enum_Comp:         2
        should be:   2
  Int_Comp:          17
        should be:   17
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Next_Ptr_Glob->
  Ptr_Comp:          1046248528
        should be:   (implementation-dependent), same as above
  Discr:             0
        should be:   0
  Enum_Comp:         1
        should be:   1
  Int_Comp:          18
        should be:   18
  Str_Comp:          DHRYSTONE PROGRAM, SOME STRING
        should be:   DHRYSTONE PROGRAM, SOME STRING
Int_1_Loc:           5
        should be:   5
Int_2_Loc:           13
```

```
        should be:   13
Int_3_Loc:           7
        should be:   7
Enum_Loc:            1
        should be:   1
Str_1_Loc:           DHRYSTONE PROGRAM, 1'ST STRING
        should be:   DHRYSTONE PROGRAM, 1'ST STRING
Str_2_Loc:           DHRYSTONE PROGRAM, 2'ND STRING
        should be:   DHRYSTONE PROGRAM, 2'ND STRING

Register option selected?  YES
Microseconds for one run through Dhrystone:     0.1
Dhrystones per Second:                       8498703.9
VAX MIPS rating =   4837.054
```

12.3 EEMBC

12.3.1 Test Environment

Objectives

The EEMBC benchmaring guideline aims to do the following:

- Baseline the EEMBC performance on QorIQ Layerscape platforms.
- Identify any optimizations and ensure they are implemented on the QorIQ Layerscape platforms.
- Investigate other changes that may improve performance.

Hardware Platform Identification

Board	Silicon Revision	Default Freqeuncy(Core/CCB/DDR) in MHz	Core Type
LS1021ATWR	Rev2.0	1000/300/1600	cortex A7
LS1043ARDB	Rev1.1	1600/400/1600	cortex A53
LS1046ARDB	Rev1.0	1800/700/2100	cortex A72
LS1088ARDB	Rev1.0	1600/700/2100	cortex A53
LS2088ARDB	Rev1.0	2000/800/2133	cortex A72

For more information on each boards switch settings, refer to the boards's Reference Manual or Getting Started Guide on http://www.nxp.com/

Software Platforma Identification

All software was built from Layerscape SDK.

Boot Loader

U-boot 2017.03 with NXP-specific patches on top.

Endianness

For ARM architecture:

Define correct Endianness by either modifying `th_lite/<platform>/al/thcfg.h` to:

```
#if !defined( EE_BIG_ENDIAN ) && !defined( EE_LITTLE_ENDIAN)
#define EE_BIG_ENDIAN    (FALSE)
#define EE_LITTLE_ENDIAN (TRUE)
#endif
```

Or by modifying `/util/make/gcc.mak` to:

```
COMPILER_DEFINES += -DEE_BIG_ENDIAN=1 -DEE_LITTLE_ENDIAN=1
```

Data Types

`th_lite/<platform>/al/eembc_dt.h` has various data types definitions If not already done, do:

```
#define HAVE_64    (1)
```

The default definition for data types is:

```
typedef unsigned long          e_u24;
typedef signed   long       e_s24;

typedef unsigned long         e_u32;
typedef signed   long       e_s32;
```

However, as ppc follows I32LP64, in case of 64 bit execution long will be considered 64 bit. So the data type definitions should be changed to:

```
typedef unsigned int          e_u24;
typedef signed   int       e_s24;

typedef unsigned int        e_u32;
typedef signed   int       e_s32;
```

Build the benchmark with following compiler flags:

```
32bit ARM(LS1021A/LS1043A/LS1046A 32ibt):

TOOLS = /usr/bin
CC = $(TOOLS)/ arm-linux-gnueabihf-gcc
AS = $(TOOLS)/arm-linux-gnueabihf-as
LD = $(TOOLS)/ arm-linux-gnueabihf-gcc
AR = $(TOOLS)/ arm-linux-gnueabihf-ar
SIZE = $(TOOLS)/ arm-linux-gnueabihf-size

#Both TCPMark and IPMark, use following compiler flags:
COMPILER_FLAGS = -mcpu=cortex-a7 -mtune=cortex-a7 -O3 -funroll-all-loops -ftree-vectorize -
flto -fwhole-program -fgcse-las
LINKER_FLAGS   =  -lm -static --sysroot=/opt/fsl-qoriq/1.9/sysroots/ppce500v2-fsl-linux-gnuspe

64bit ARM(LS1043A/LS1046A/LS1088A/LS2088A):
TOOLS = /usr/bin

CC = $(TOOLS)/ aarch64-linux-gnu-gcc
AS = $(TOOLS)/ aarch64-linux-gnu-as
LD = $(TOOLS)/ aarch64-linux-gnu-gcc
AR = $(TOOLS)/ aarch64-linux-gnu-ar

COMPILER_FLAGS = -O3 -funroll-all-loops -ftree-vectorize
LINKER_FLAGS   =  -lm -static
SIZE = $(TOOLS)/ aarch64-linux-gnu-size
```

Generate EEMBC Binary for Target Board

1. Create a working directory.
2. Retrieve EEMBC v2.0 from http://www.eembc.org/

3. Extract the EEMBC v2.0 source code.
4. Edit `util/make/gcc.mak` so that the CC variable points to the correct location of your compiler. See the table above for detailed compiler flags and configuration.
5. Build binary using the make command:

   ```
   make VER=v2 TOOLCHAIN=gcc THLITE=_lite all-lite
   ```

6. After the build is complete, copy binary files under `<EEMBC_2.0_INSTALL_DIR>/networking/gcc/bin_lite` to target board.

12.3.2 Test Procedure

Running test and result collection

1. Deploy the target board with corresponding software mentioned in the previous section.
2. Put EEMBC Netmark binary compiled with optimized flags mentioned in section test environment on target board. EEMBC Netmark binary file list are as follows:

   ```
   networking/tcpbulk_lite.exe
   networking/tcpjumbo_lite.exe
   networking/tcpmixed_lite.exe
   networking/ip_pktcheckb1m_lite.exe
   networking/ip_pktcheckb2m_lite.exe
   networking/ip_pktcheckb4m_lite.exe
   networking/ip_pktcheckb512k_lite.exe
   networking/ip_reassembly_lite.exe
   networking/ip_reassembly_lite.exe
   networking/nat_lite.exe -INITTIME
   networking/nat_lite.exe
   networking/ospfv2_lite.exe
   networking/qos_lite.exe
   networking/routelookup_lite.exe
   ```

3. Run the benchmark:

   ```
   networking/tcpbulk_lite.exe -i 200000
   networking/tcpjumbo_lite.exe -i 300000
   networking/tcpmixed_lite.exe -i 100000
   networking/ip_pktcheckb1m_lite.exe -i 50000
   networking/ip_pktcheckb2m_lite.exe -i 30000
   networking/ip_pktcheckb4m_lite.exe -i 10000
   networking/ip_pktcheckb512k_lite.exe -i 100000
   networking/ip_reassembly_lite.exe -INITTIME -i 5000
   networking/ip_reassembly_lite.exe -i 5000
   networking/nat_lite.exe -INITTIME -i 10000
   networking/nat_lite.exe -i 10000
   networking/ospfv2_lite.exe -i 10000
   networking/qos_lite.exe -i 300
   networking/routelookup_lite.exe -i 20000
   ```

Check the log below for the results:

```
root@localhost# ./tcpbulk_lite -i 167000
 Configure benchmark for bulk data transfer test
Initialize network buffer pools
INFO: Initializing client and server NIF
>>------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 167000
>> Bench Mark               : TCP-BM bulk V2.0R1
```

```
--  Non-Intrusive CRC =    0
--  Iterations        = 167000
--  Target Duration   = 1600000
--  Target Timer Rate = 1000000
--  v1                = -4280
--  v2                = 0
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =   104375.000
--  Total Run Time    =        1.600sec
--  Time / Iter       =        0.000009581sec
>> DONE!
>> BM: TCP-BM bulk V2.0R1
>> ID: NTW tcp

root@localhost# ./tcpjumbo_lite -i 250500
 Configure benchmark for jumbo packet transfer test
Initialize network buffer pools
INFO: Initializing client and server NIF
>>------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 250500
>> Bench Mark               : TCP-BM jumbo V2.0R1
--  Non-Intrusive CRC =    0
--  Iterations        = 250500
--  Target Duration   = 1540000
--  Target Timer Rate = 1000000
--  v1                = -24000
--  v2                = 0
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =   162662.338
--  Total Run Time    =        1.540sec
--  Time / Iter       =        0.000006148sec
>> DONE!
>> BM: TCP-BM jumbo V2.0R1
>> ID: NTW tcp

root@localhost# ./tcpmixed_lite -i 83500
 Configure benchmark for mixed packet size test
Initialize network buffer pools
INFO: Initializing client and server NIF
>>------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 83500
>> Bench Mark               : TCP-BM mixed V2.0R1
--  Non-Intrusive CRC =    0
--  Iterations        = 83500
--  Target Duration   = 1940000
--  Target Timer Rate = 1000000
--  v1                = -2736
--  v2                = 0
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =    43041.237
--  Total Run Time    =        1.940sec
--  Time / Iter       =        0.000023234sec
>> DONE!
>> BM: TCP-BM mixed V2.0R1
>> ID: NTW tcp


root@localhost# ./ip_pktcheckb1m_lite -i 41750
>> Datagram buffer size          : 0x0100000
>> Datagram alignment            :         4
>> Descriptor padd size          :         8
>> Number of Datagrams allocated :       720
>>------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 41750
>> Bench Mark               : Networking: IP Packet Check Benchmark, 1.0M V2.0R1
--  Non-Intrusive CRC = e3b5
--  Iterations        = 41750
--  Target Duration   = 2260000
--  Target Timer Rate = 1000000
--  v1                = 30060000
--  v2                = 1670000
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =    18473.451
--  Total Run Time    =        2.260sec
--  Time / Iter       =        0.000054132sec
>> DONE!
>> BM: Networking: IP Packet Check Benchmark, 1.0M V2.0R1
>> ID: NTW ip_pktchec

root@localhost# ./ip_pktcheckb2m_lite -i 25050
>> Datagram buffer size          : 0x0200000
>> Datagram alignment            :         4
>> Descriptor padd size          :         8
>> Number of Datagrams allocated :      1412
>>------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 25050
>> Bench Mark               : Networking: IP Packet Check Benchmark, 2.0M V2.0R1
--  Non-Intrusive CRC =  48b
--  Iterations        = 25050
--  Target Duration   = 2720000
--  Target Timer Rate = 1000000
--  v1                = 35370600
--  v2                = 1828650
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =     9209.559
--  Total Run Time    =        2.720sec
--  Time / Iter       =        0.000108583sec
>> DONE!
>> BM: Networking: IP Packet Check Benchmark, 2.0M V2.0R1
>> ID: NTW ip_pktchec
```

```
root@localhost# ./ip_pktcheckb4m_lite -i 8350
>> Datagram buffer size          : 0x0400000
>> Datagram alignment            :         4
>> Descriptor padd size          :         8
>> Number of Datagrams allocated :      2824
>>------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 8350
>> Bench Mark               : Networking: IP Packet Check Benchmark, 4.0M V2.0R1
--  Non-Intrusive CRC = d86c
--  Iterations        = 8350
--  Target Duration   = 1800000
--  Target Timer Rate = 1000000
--  v1                = 23580400
--  v2                = 1244150
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =     4638.889
--  Total Run Time    =        1.800sec
--  Time / Iter       =        0.000215569sec
>> DONE!
>> BM: Networking: IP Packet Check Benchmark, 4.0M V2.0R1
>> ID: NTW ip_pktchec

root@localhost# ./ip_pktcheckb512k_lite -i 83500
>> Datagram buffer size          : 0x0080000
>> Datagram alignment            :         4
>> Descriptor padd size          :         8
>> Number of Datagrams allocated :       374
>>------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 83500
>> Bench Mark               : Networking: IP Packet Check Benchmark, 0.5M V2.0R1
--  Non-Intrusive CRC = 3e1d
--  Iterations        = 83500
--  Target Duration   = 2160000
--  Target Timer Rate = 1000000
--  v1                = 31229000
--  v2                = 1753500
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =    38657.407
--  Total Run Time    =        2.160sec
--  Time / Iter       =        0.000025868sec
>> DONE!
>> BM: Networking: IP Packet Check Benchmark, 0.5M V2.0R1
>> ID: NTW ip_pktchec


root@localhost# ./ip_reassembly_lite -INITTIME -i 4175
*** Initialization Timing Run, Subtract from normal run for score ***
>>------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 4175
>> Bench Mark               : INITIALIZATION Networking: IP Reassembly Benchmark V2.0R1
--  Non-Intrusive CRC =    0
--  Iterations        = 4175
--  Target Duration   = 640000
--  Target Timer Rate = 1000000
--  v1                = 200
--  v2                = 0
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =     6523.438
--  Total Run Time    =        0.640sec
--  Time / Iter       =        0.000153293sec
>> DONE!
>> BM: INITIALIZATION Networking: IP Reassembly Benchmark V2.0R1
>> ID: NTW ip_reasmIT

root@localhost# ./ip_reassembly_lite -i 4175
>>------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 4175
>> Bench Mark               : Networking: IP Reassembly Benchmark V2.0R1
--  Non-Intrusive CRC = 678e
--  Iterations        = 4175
--  Target Duration   = 1940000
--  Target Timer Rate = 1000000
--  v1                = 200
--  v2                = 4939025
--  v3                = 835000
--  v4                = 0
--  Iterations/Sec    =     2152.062
--  Total Run Time    =        1.940sec
--  Time / Iter       =        0.000464671sec
>> DONE!
>> BM: Networking: IP Reassembly Benchmark V2.0R1
>> ID: NTW ip_reasm


root@localhost# ./nat_lite -INITTIME -i 8350
*** Initialization Timing Run, Subtract from normal run for score ***
>>------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 8350
>> Bench Mark               : INITIALIZATION Network Address Translation V2.0R1
--  Non-Intrusive CRC =    0
--  Iterations        = 8350
--  Target Duration   = 960000
--  Target Timer Rate = 1000000
--  v1                = 1000
--  v2                 = 0
--  v3                 = 0
--  v4                 = 0
--  Iterations/Sec     =     8697.917
--  Total Run Time     =        0.960sec
--  Time / Iter        =        0.000114970sec
>> DONE!
>> BM: INITIALIZATION Network Address Translation V2.0R1
>> ID: NTW NATIT

root@localhost# ./nat_lite -i 8350
>>------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 8350
>> Bench Mark               : Network Address Translation V2.0R1
--  Non-Intrusive CRC = 9d46
--  Iterations        = 8350
--  Target Duration   = 2320000
--  Target Timer Rate = 1000000
--  v1                = 1000
--  v2                = 0
--  v3                = 0
--  v4                = 0
--  Iterations/Sec     =     3599.138
--  Total Run Time     =        2.320sec
--  Time / Iter        =        0.000277844sec
>> DONE!
>> BM: Network Address Translation V2.0R1
>> ID: NTW NAT


root@localhost# ./ospfv2_lite -i 8350
>>------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 8350
>> Bench Mark               : Networking: OSPF Benchmark V2.0R1
--  Non-Intrusive CRC = 7f12
--  Iterations        = 8350
--  Target Duration   = 1480000
--  Target Timer Rate = 1000000
--  v1                = 400
--  v2                = 4
--  v3                = 8
--  v4                = 32000
--  Iterations/Sec     =     5641.892
--  Total Run Time     =        1.480sec
--  Time / Iter        =        0.000177246sec
>> DONE!
>> BM: Networking: OSPF Benchmark V2.0R1
>> ID: NTW ospf


root@localhost# ./qos_lite -i 250
>>------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 250
>> Bench Mark               : Networking: QoS V2.0R1
--  Non-Intrusive CRC = fa81
--  Iterations        = 250
--  Target Duration   = 1000000
--  Target Timer Rate = 1000000
--  v1                = 100
--  v2                = 100
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =      250.000
--  Total Run Time    =        1.000sec
--  Time / Iter       =        0.004000000sec
>> DONE!
>> BM: Networking: QoS V2.0R1
>> ID: NTW QoS


root@localhost# ./routelookup_lite -i 16700
>> self-check completed ok.
>>------------------------------------------------------------
>> EEMBC Component          : EEMBC Portable Test Harness V4.100
>> EEMBC Member Company     : EEMBC
>> Target Processor         : HOST EXAMPLE
>> Target Platform          : 32 Bit
>> Target Timer Available   : YES
>> Target Timer Intrusive   : YES
>> Target Timer Rate        : 1000000
>> Target Timer Granularity : 10
>> Recommended Iterations   : 16700
>> Bench Mark               : Networking: Route Lookup Benchmark V2.0R1
--  Non-Intrusive CRC = 407d
--  Iterations        = 16700
--  Target Duration   = 2100000
--  Target Timer Rate = 1000000
--  v1                = 0
--  v2                = 0
--  v3                = 0
--  v4                = 0
--  Iterations/Sec    =     7952.381
--  Total Run Time    =        2.100sec
--  Time / Iter       =        0.000125749sec
>> DONE!
>> BM: Networking: Route Lookup Benchmark V2.0R1
>> ID: NTW routelookup EEMBC
```

NOTE

For other platforms, use the ratio (CPU_Freq_target_platform/2000) to get the corresponding parameter.

There is a run to run variation, so an average across 5 runs was taken for every result.

Networking Version 2.0 Calculation

Networking Version 2.0 produces two aggregate "mark" scores: the TCPmark™ and the IPmark™. The IPmark is intended for developers of infrastructure equipment, while the TCPmark, which includes the TCP benchmark, focuses on client- and server-based network hardware.

The IPmark is the geometric mean of the scores for QoS, Route Lookup, OSPF, IP Reassembly, Network Address Translation, and the geometric mean of the individual scores for IP Packet check, all divided by 10:

```
IPmark = Geomean ((Geomean (IP Packet Check [0.5MB], IP Packet Check [1MB], IP Packet Check
[2MB], IP Packet Check [4MB]), QoS, Route Lookup, OSPF, IP Reassembly, NAT))/10
```

The TCPmark is the geometric mean of the scores for TCP Jumbo, TCP Bulk, and TCP Mixed divided by 100:

```
TCPmark = Geomean (TCP jumbo, TCP bulk, TCP Mixed)/100
```

To calculate a geometric mean, multiply all the results of the tests together and take the nth root of the product, where n equals the number of tests.

NAT and IP Reassembly Benchmarks

To calculate the iterations per second for the NAT and IP Reassembly benchmarks, it's necessary to run the benchmarks twice:

1. Run the benchmark with the flag -INITTIME supplied on the command line.
2. Run the benchmark without the flag -INITTIME supplied on the command line. The same executable must be run both times and the number of iterations must be identical.
3. Subtract the time of the first run from the time of the second run and calculate the iterations per second based on the calculated time.

The score reported for each device is a single-number figure of merit calculated by taking the geometric mean of the individual Networking scores and dividing by 395.184. This normalization factor is derived from the lowest score in this category on December 5, 2000. Scores for each of the individual benchmarks within this suite allow designers to weight and aggregate the benchmarks to suit specific application requirements.

To calculate a geometric mean, multiply all the results (*) of the tests together and take the nth root of the product, where n equals the number of tests. (*) Scores included in geometric mean:

```
•OSPF
•Route Lookup
•Packet Flow - 512 kbytes
•Packet Flow - 1 Mbyte
•Packet Flow - 2 Mbytes
```

NOTE

This calculation can also be found on EEMBC website: http://eembc.org/benchmark/reports/mark.php?suite=NT2

12.4 LMBench

12.4.1 Test Environment

Objectives

The LMBench benchmaring guideline aims to do the following:

- Baseline the LMBench performance on QorIQ Layerscape platforms.
- Identify any optimizations and ensure they are implemented on the QorIQ Layerscape platforms.
- Investigate other changes that may improve performance.

Hardware Platform Identification

Board	Silicon Revision	Default Freqeuncy(Core/CCB/DDR) in MHz	Core Type
LS1021ATWR	Rev2.0	1000/300/1600	cortex A7
LS1043ARDB	Rev1.1	1600/400/1600	cortex A53
LS1046ARDB	Rev1.0	1800/700/2100	cortex A72
LS1088ARDB	Rev1.0	1600/700/2100	cortex A53
LS2088ARDB	Rev1.0	2000/800/2133	cortex A72

For more information on each board's switch settings, refer to the boards's Reference Manual or Getting Started Guide on http://www.nxp.com/

Software Platforma Identification

All software was built from Layerscape SDK.

Boot Loader

U-boot 2017.03 with NXP-specific patches on top.

If the target platform support chip select interleaving and address hash, please enable chip select interleaving and address hash, disabling ecc by hwconfig with syntax:

```
“hwconfig=fsl_ddr:bank_intlv=cs0_cs1,addr_hash=true,ecc=off”
```

LMBench Application

The rootfs includes LMBench binaries which were built without optimization. For general latency performance, the default LMBench binary file in the root file system will be OK. To get better bandwidth performance result, modify the compiler flags to enable "O3" optimization.

1. Download lmbench source code from following link: https://sourceforge.net/projects/lmbench/files/latest/download
2. Change “CC” and “CFLAGS” in file `src/Makefile` as the following:

   ```
   CC = /usr/bin/aarch64-linux-gnu-gcc
   CFLAGS = -O3
   ```

3. Build code with new optimized compiler flags:

   ```
   #make
   ```

4. Transfer the `bw_mem` binary file to target board.

12.4.2 Test Procedure

Running Test and Result Collection

Separate U-Boot image was flashed in alternate U-Boot flash bank and the DUT is booted out of that bank. Binaries compiled with different flags were run separately and then the data was collected for the binary showing the best results. To get full LMBench test result, run `lmbench-run` to get the full test result.

There is a run to run variation in, so an average across 5 runs was taken for every result

Scripts for LMBench Test

Execution latency test script:

```
for i in `seq 1 5`
do
echo “Integer, integer64,float,double float execution latency”
lat_ops
done
```

Memory read latency test script:

```
for i in `seq 1 5`
do
echo “L1, L2, L3 and DDR read latency”
lat_mem_rd 100M
done
```

Memory bandwidth test script:

```
#!/bin/sh

for opt in rd wr rdwr cp frd fwr fcp bzero bcopy
do
   echo "L1 cache bandwidth $opt test with #$proc process"
   #8k is fit for all platform

   for idx in `seq 1 5`
   do
       bw_mem -P 1 8k $opt
   done
   echo "L2 cache bandwidth $opt test"
   # For Layerscape platform, each platform has more than 256K L2 cache, so chose 128k as L2
cache size.
   for idx in `seq 1 5`
   do
       bw_mem -P 1 128k $opt
   done

   echo "Main mem bandwidth $opt test"
   for idx in `seq 1 5`
   do
       bw_mem -P 1 100m $opt
   done
done
```

Chapter 13
Connect to cloud: EdgeScale

13.1 What is EdgeScale

EdgeScale is a unified, scalable, and secure device management solution for Edge Computing applications. It enables OEMs and developers to leverage cloud compute frameworks like AWS Greengrass, Azure IoT, and Aliyun on Layerscape devices.

EdgeScale provides the missing piece of device security and management needed for customers to securely deploy and manage many Edge computing devices from the cloud. End-users and developers can use the EdgeScale cloud dashboard to securely enroll Edge devices, monitor their health, attest, and deploy container applications and firmware updates.

EdgeScale can also be used as a development environment to build containers and generate firmware.

Supported features

- EdgeScale dashboard for users
- Secure device enrolment
- Secure key/certificate provisioning
- OTA: firmware update (LS1012A, LS1043 or LS1046)
- Device status monitoring on the cloud
- Dynamic deployment of container-based applications

The above specified features are currently supported in LSDK. For more details, please visit: EDGESCALE: EdgeScale for Secure Edge Computing

13.2 Building EdgeScale client

To build EdigeScale client with Flexbuild, follow the steps below:

```
$ cd flexbuild
$ source setup.env
$ set CONFIG_BUILD_QORIQ_EDGESCALE to y from default n in configs/build_lsdk.cfg

If necessary, you can specify secure key pair in configs/build_lsdk.cfg to override the
default one as below:
SECURE_PRI_KEY=/home/xx/path/srk.pri
SECURE_PUB_KEY=/home/xx/path/srk.pub

$ flex-builder clean
$ flex-builder -m <machine>                (for specified machine)
or
$ flex-builder -i autobuild -a <arch>      (for all <arch> machines)

Install images into SD card:
$ flex-installer -b build/images/bootpartition_<arch>_lts_<version>  -r build/rfs/
rootfs_ubuntu_bionic_<arch>  -d /dev/sdx  (or /dev/mmcblk0)
or
$ flex-installer -b build/images/bootpartition_<arch>_lts_<version>  -r build/rfs/
rootfs_ubuntu_bionic_<arch>  -f firmware_<machine>_uboot_sdboot.img  -d /dev/sdx
or
$ flex-installer -b bootpartition_<arch>_lts_<version>.tgz -r rootfs_ubuntu_<version>.tgz -
d /dev/sdx
```

13.3 Procedure to start EdgeScale

For complete details on how to start EdgeScale, see https://doc.edgescale.org/

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.