Freescale Semiconductor, Inc.
Reference Manual

Document Number: QORIQSDKLS1
Rev A, 12/2015

# Freescale Linux LS1043A SDK v0.5

For QorIQ Processors

freescale™

# Contents


Chapter 1 SDK Overview.................................................................. 23
1.1 What's New..................................................................23
1.2 Components.................................................................. 24
1.3 Known Issues................................................................. 27

Chapter 2 Getting Started with Yocto Project...........................................29
2.1 Essential Build Instructions..................................................29
2.1.1 Install the SDK.............................................................29
2.1.2 Prepare the Host Environment.............................................. 29
2.1.3 Set Up Poky................................................................31
2.1.4 Builds.....................................................................32
2.2 Additional Instructions for Developers........................................33
2.2.1 U-Boot.....................................................................33
2.2.2 Linux Kernel...............................................................34
2.2.3 Packages...................................................................35
2.2.4 Customize Root Filesystem..................................................36
2.2.5 Native Packages............................................................37
2.2.6 Extract Source Code........................................................37
2.2.7 Standalone toolchain...................................................... 37
2.2.8 Shared State (sstate) Cache............................................... 38
2.2.9 How to use hob ............................................................38
2.2.10 FAQ...................................................................... 50
2.2.11 BitBake User Manual.......................................................53

Chapter 3 Deployment Guides........................................................... 55
3.1 Introduction..................................................................55
3.2 Basic Host Set-up.............................................................55
3.3 Target Board Set-up.......................................................... 57
3.4 Boards....................................................................... 58
3.4.1 Overview...................................................................58
3.4.2 Switch Settings............................................................ 58
3.4.3 U-Boot Environment Variables.............................................. 59
3.4.3.1 U-Boot Environment Variable “hwconfig”...................................59
3.4.3.2 Configuring U-Boot Network Parameters.....................................59
3.4.4 RCW (Reset Configuration Word) ........................................... 60
3.4.5 System Memory Map..........................................................62
3.4.6 Flash Bank Usage.......................................................... 63
3.4.7 Programming a New U-boot and RCW...........................................65
3.4.8 Deployment.................................................................66
3.4.8.1 FIT Image Deployment from TFTP.......................................... 66
3.4.8.2 FIT Image Deployment from Flash.......................................... 67
3.4.8.3 NFS Deployment...........................................................67
3.4.8.4 SD Deployment............................................................68
3.4.8.5 QSPI Deployment(only for QDS board)...................................... 70
3.4.9 Check 'Link Up' for Serial Interfaces......................................72
3.4.10 Basic Networking Ping Test................................................72
3.4.11 Hardware Setting for Special Purposes......................................73

Chapter 4 System Recovery.................... 79
4.1 Environment Setup.................... 79
4.1.1 Environment Setup (Common).................... 79
4.2 Image Recovery.................... 79
4.2.1 Recover system with already working U-Boot.................... 79
4.2.2 Recover system using CodeWarrior Flash Programmer.................... 80

Chapter 5 About Yocto Project.................... 83
5.1 Yocto Project Quick Start.................... 83
5.2 Application Development Toolkit User's Guide.................... 83
5.3 Board Support Packages - Developer's Guide.................... 83
5.4 Yocto Project Development Manual.................... 83
5.5 Yocto Project Linux Kernel Development Manual.................... 83
5.6 Yocto Project Profiling and Tracing Manual.................... 84
5.7 Yocto Project Reference Manual.................... 84

Chapter 6 Configuring DPAA Frame Queues.................... 85
6.1 Introduction.................... 85
6.2 FMan Network interface Frame Queue Configuration.................... 86
6.3 FMan network interface ingress FQs configuration.................... 86
6.4 Ingress FQs common configuration guidelines.................... 87
6.5 Dynamic load balancing with order preservation - ingress FQs configuration guidelines.................... 88
6.6 Dynamic load balancing with order restoration - ingress FQs configuration guidelines.................... 89
6.7 Static distribution - Ingress FQs Configuration Guidelines.................... 90
6.8 FMan network interface egress FQs configuration.................... 90
6.9 Accelerator Frame Queue Configuration.................... 91
6.10 DPAA Frame Queue Configuration Guideline Summary.................... 92

Chapter 7 DPAA Primer for Software Architecture.................... 97
7.1 DPAA Primer.................... 97
7.1.1 General Architectural Considerations.................... 97
7.1.2 Multicore Design.................... 97
7.1.3 Parse/classification Software Offload.................... 98
7.1.4 Flow Order Considerations.................... 98
7.1.5 Managing Flow-to-Core Affinity.................... 100
7.2 DPAA Goals.................... 102
7.3 FMan Overview.................... 102
7.4 QMan Overview.................... 104
7.5 QMan Scheduling.................... 110
7.6 BMan.................... 114
7.7 Order Handling.................... 115
7.8 Pool Channels.................... 118
7.9 Application Mapping.................... 122
7.10 FQ/WQ/Channel.................... 126

Chapter 8 DSPI Device Driver User Manual.................... 131
8.1 DSPI Device Driver User Manual.................... 131

Chapter 9 eDMA User Manual........................................................................ 133
9.1 eDMA User Manual........................................................................ 133

Chapter 10 Enhanced Secured Digital Host Controller (eSDHC)................. 137
10.1 eSDHC Driver User Manual........................................................................ 137

Chapter 11 Frame Manager ........................................................................ 143
11.1 Frame Manager Linux Driver API Reference........................................................................ 143

Chapter 12 Frame Manager Driver User's Guide ........................................145

Chapter 13 Frame Manager Configuration Tool User's Guide .....................147

Chapter 14 IEEE 1588 Device Driver User Manual..................................... 149
14.1 IEEE 1588 Device Driver User Manual........................................................................ 149
14.2 IXXAT IEEE 1588-2008 Quick Start........................................................................ 155

Chapter 15 IFC NOR Flash User Manual....................................................157
15.1 Integrated Flash Controller NOR Flash User Manual........................................................................157

Chapter 16 IFC NAND Flash User Manual................................................ 163
16.1 Integrated Flash Controller NAND Flash User Manual........................................................................163

Chapter 17 KVM/QEMU........................................................................ 173
17.1 Freescale KVM/QEMU Release Notes........................................................................173

Chapter 18 Libvirt Users Guide........................................................................ 175
18.1 Introduction to libvirt........................................................................ 175
18.1.1 Overview........................................................................175
18.1.2 For Further Information........................................................................175
18.1.3 Libvirt in the Freescale QorIQ SDK -- Supported Features........................................... 176
18.2 Build, Installation, and Configuration........................................................................ 178
18.2.1 Building Libvirt with Yocto........................................................................178
18.2.2 Running libvirtd........................................................................ 179
18.2.3 Libvirt Domain Lifecycle........................................................................179
18.2.4 Libvirt URIs........................................................................ 181
18.2.5 virsh........................................................................ 181
18.2.6 Libvirt xml........................................................................181
18.3 Examples........................................................................181
18.3.1 KVM Examples........................................................................ 181
18.3.1.1 Libvirt KVM/QEMU Example (Power Architecture)......................................... 181
18.3.1.2 Libvirt KVM/QEMU -- Adding Devices Example (Power Architecture)..............183
18.3.1.3 Libvirt KVM/QEMU Example (ARM Architecture)............................................ 186
18.3.2 Libvirt_lxc Examples........................................................................ 190
18.3.2.1 Basic Example........................................................................ 190

18.3.2.2 Custom Container Filesystem....192
18.3.2.3 Container Terminal Setup....193
18.3.2.4 Networking Examples....195
18.3.2.4.1 Shared Networking....195
18.3.2.4.2 Ethernet Bridging....195
18.3.2.4.3 MACVLAN....196
18.3.2.4.4 Direct Assignment....197
18.3.2.4.5 VLAN....198

Chapter 19 Linux Containers (LXC) for Freescale QorIQ User's Guide........201

19.1 Introduction to Linux Containers....201
19.1.1 Freescale LXC Release Notes....201
19.1.2 Overview....202
19.1.3 Comparing LXC and Libvirt....203
19.1.4 For Further Information....203
19.2 Build, Installation, and Configuration....204
19.2.1 Summary....204
19.2.2 LXC: Building with Yocto....204
19.2.3 Building Busybox....205
19.2.4 Building the Linux Kernel....206
19.2.5 Host Root Filesystem Configuration for Linux Containers....208
19.3 More Details....209
19.3.1 LXC: Command Reference....209
19.3.2 LXC: Configuration Files....210
19.3.3 LXC: Templates....211
19.3.4 Containers with Libvirt....211
19.3.5 Linux Control Groups (cgroups)....213
19.3.6 Linux Namespaces....214
19.3.7 POSIX Capabilities....214
19.4 LXC How To's....215
19.4.1 LXC: Getting Started (with a Busybox System Container)....215
19.4.2 LXC: How to configure non-virtualized networking (lxc-no-netns.conf)....219
19.4.3 LXC: How to assign a physical network interface to a container (lxc-phys.conf)....219
19.4.4 LXC: How to configure networking with virtual Ethernet pairs (lxc-veth.conf)....221
19.4.5 LXC: How to configure networking with macvlan (lxc-macvlan.conf)....222
19.4.6 LXC: How to configure networking using a VLAN (lxc-vlan.conf)....224
19.4.7 LXC: How to monitor containers....226
19.4.8 LXC: How to modify the capabilities of a container to provide additional isolation....227
19.4.9 LXC: How to use cgroups to manage and control a containers resources....228
19.4.10 LXC: How to run an application in a container with lxc-execute....229
19.4.11 LXC: How to run an unprivileged container....230
19.4.12 LXC: How to run containers with Seccomp protection....232
19.5 Libvirt How To's....234
19.5.1 Basic Example....234
19.6 USDPAA in LXC....236
19.6.1 General Setup....237
19.6.2 Running multiple USDPAA instances....239
19.6.3 Running Reflector in Containers....242
19.7 Appendix....244
19.7.1 LXC Configuration File Reference....244
19.7.2 Documentation/cgroups/cgroups.txt....260

Chapter 20 Linux Ethernet Driver for DPAA 1.x Family.... 273

20.1 Linux Ethernet Driver for DPAA 1.x Family........................................................................273
20.1.1 Introduction........................................................................273
20.1.2 Private DPAA Ethernet Driver........................................................................274
20.1.2.1 Configuration........................................................................274
20.1.2.1.1 Device Tree Configuration........................................................................275
20.1.2.1.2 Bootargs........................................................................275
20.1.2.1.3 Kconfig Options........................................................................277
20.1.2.1.4 ethtool Options........................................................................278
20.1.2.2 Features........................................................................279
20.1.2.2.1 Congestion Management........................................................................279
20.1.2.2.2 Scatter/Gather Support........................................................................279
20.1.2.2.3 GRO/GSO Support........................................................................279
20.1.2.2.4 Transmit Packet Steering........................................................................280
20.1.2.2.5 TX and RX Hardware Checksum........................................................................281
20.1.2.2.6 Pause Frames Flow Control ........................................................................282
20.1.2.2.7 Priority Flow Control ........................................................................283
20.1.2.2.8 Core Affined Queues........................................................................285
20.1.3 Ethernet Advanced Drivers........................................................................287
20.1.3.1 Macless DPAA Ethernet Driver........................................................................287
20.1.3.1.1 Configuration........................................................................287
20.1.3.1.2 Features........................................................................291
20.1.3.2 Shared DPAA Ethernet Driver........................................................................292
20.1.3.2.1 Configuration........................................................................292
20.1.3.2.2 Features........................................................................296
20.1.3.3 Proxy DPAA Ethernet Driver........................................................................296
20.1.3.3.1 Configuration........................................................................296
20.1.3.3.2 Features........................................................................297
20.1.3.4 Offload NIC Ethernet Driver........................................................................297
20.1.3.4.1 Configuration........................................................................298
20.1.3.4.2 Features........................................................................300
20.1.4 Offline Parsing Port Driver........................................................................301
20.1.4.1 Configuration........................................................................301
20.1.4.1.1 Kconfig Option........................................................................301
20.1.4.1.2 Device Tree Configuration ........................................................................301
20.1.4.2 Features........................................................................302
20.1.5 Link Management........................................................................303
20.1.5.1 Device Tree........................................................................303
20.1.5.2 Bootargs........................................................................303
20.1.5.3 Muxed MDIO........................................................................304
20.1.6 Debugging........................................................................304
20.1.6.1 Debugfs support........................................................................305
20.1.6.2 FMan address calculation........................................................................306
20.1.6.3 Sysfs support........................................................................306
20.1.7 Adding support for DPAA Ethernet in Topaz Hypervisor........................................................................306
20.1.8 MACsec........................................................................308
20.1.8.1 MACsec User-space Linux API........................................................................308
20.1.8.2 MACsec Quick Start Guide........................................................................319
20.1.9 Changes from previous versions........................................................................322
20.1.9.1 Known Issues........................................................................324
20.1.9.2 Good Questions........................................................................324

Chapter 21 Linux DPAA 1.x Ethernet Primer........................................................................327
21.1 Introduction........................................................................327
21.2 Intended Use Cases........................................................................327

21.2.1 Private Net Devices .......... 327
21.2.2 Shared-MAC Net Devices.......... 328
21.2.3 MAC-less Net Devices.......... 330
21.2.4 Choosing the Current Use Case.......... 331
21.3 The DPAA-Eth View of the World.......... 332
21.3.1 The Linux Kernel API's .......... 332
21.3.2 The Driver's Building Blocks .......... 334
21.3.2.1 Net Devices.......... 334
21.3.2.2 Frame Queues.......... 335
21.3.2.3 Buffer Pools.......... 336
21.4 DPAA Resources Initialization.......... 336
21.4.1 What, Why and How Resources are Initialized.......... 336
21.4.2 Hashing/PCD Frame Queues.......... 336
21.5 The (Simplified) Life of a Packet.......... 337
21.5.1 Private Net Device: Tx.......... 337
21.5.2 Private Net Device: Rx.......... 338
21.5.3 Shared MAC: Tx.......... 338
21.5.3.1 Shared MAC: Rx.......... 340
21.5.3.2 MAC-less Net Devices: Tx.......... 341
21.5.3.3 MAC-less Net Devices: Rx.......... 342
21.6 Advanced Drivers Use Cases.......... 343
21.6.1 MAC-less Over OH (Linux-USDPAA.......... 343
21.6.2 MAC-less Over OH (Linux-Linux).......... 344
21.6.3 ARP Handling in Shared MAC.......... 344
21.6.4 Multicast Support in Shared MAC.......... 344
21.7 Appendix A: Infrequently Asked Questions.......... 345
21.8 Appendix B: Frequently Asked Questions.......... 346

## Chapter 22 Low Power UART User Guide.......... 347
22.1 Low Power UART User Guide.......... 347

## Chapter 23 PCI/PCIe System User Manual.......... 351
23.1 PCI/PCIe System User Manual.......... 351

## Chapter 24 PCI Express Interface Controller.......... 355
24.1 PCI-e Remove and Rescan User Manual.......... 355

## Chapter 25 Power Management.......... 357
25.1 Power Management User Manual.......... 357
25.2 Thermal Management User Manual.......... 366
25.3 System Monitor.......... 369
25.3.1 Power Monitor User Manual.......... 369
25.3.2 Thermal Monitor User Manual.......... 374
25.3.3 Web-based System Monitor User Guide.......... 376

## Chapter 26 QDMA.......... 379
26.1 QDMA User Manual.......... 379

## Chapter 27 Queue Manager (QMan) and Buffer Manager (BMan).......... 381

27.1 QMan/BMan Drivers Release Notes........ 381

Chapter 28 QMan BMan API Reference........ 391
28.1 About this document........ 391
28.1.1 Suggested Reading........ 391
28.2 Introduction to the Queue Manager and the Buffer Manager........ 391
28.2.1 O/S specifics........ 391
28.3 Buffer Manager........ 391
28.3.1 BMan Overview........ 391
28.3.1.1 Buffer Manager's Function........ 391
28.3.1.2 BMan's interfaces........ 392
28.3.2 BMan configuration interface........ 393
28.3.2.1 BMan Device-Tree Node........ 393
28.3.2.1.1 Free Buffer Proxy Records........ 393
28.3.2.1.2 Logical I/O Device Number (BMan)........ 393
28.3.2.2 Buffer Pool Node........ 394
28.3.2.2.1 Buffer Pool ID........ 394
28.3.2.2.2 Seeding Buffer Pools........ 395
28.3.2.2.3 Depletion Thresholds........ 395
28.3.2.3 BMan Portal Device-Tree Node........ 395
28.3.2.3.1 Portal Initialization (BMan)........ 396
28.3.2.3.2 Portal sharing........ 396
28.4 BMan CoreNet portal APIs........ 396
28.4.1 BMan High-Level Portal Interface........ 396
28.4.1.1 Overview (BMan)........ 396
28.4.1.2 Portal management (BMan)........ 397
28.4.1.2.1 Modifying interrupt-driven portal duties (BMan)........ 397
28.4.1.2.2 Processing non-interrupt-driven portal duties (BMan)........ 398
28.4.1.2.3 Recovery support (BMan)........ 398
28.4.1.2.4 Determining if the release ring is empty........ 398
28.4.1.3 Pool Management........ 399
28.4.1.4 Releasing and Acquiring Buffers........ 400
28.4.1.5 Depletion State........ 401
28.5 Queue Manager........ 402
28.5.1 QMan Overview........ 402
28.5.1.1 Queue Manager's Function........ 402
28.5.1.2 Frame Descriptors........ 402
28.5.1.3 Frame Queue Descriptors (QMan)........ 403
28.5.1.4 Work Queues........ 403
28.5.1.5 Channels........ 403
28.5.1.6 Portals........ 403
28.5.1.7 Dedicated Portal Channels........ 403
28.5.1.8 Pool Channels........ 403
28.5.1.9 Portal Sub-Interfaces........ 404
28.5.1.10 Frame queue dequeuing........ 404
28.5.1.10.1 Unscheduled Dequeues........ 404
28.5.1.10.2 Scheduled Dequeues........ 404
28.5.1.10.3 Pull Mode........ 404
28.5.1.10.4 Push Mode........ 405
28.5.1.10.5 Stashing to Processor Cache........ 405
28.5.1.11 Frame Queue States........ 406
28.5.1.12 Hold active........ 407
28.5.1.12.1 Dequeue Atomicity........ 407
28.5.1.12.2 Parking Scheduled FQs........ 408

28.5.1.12.3 Order Preservation & Discrete Consumption Acknowledgement......408
28.5.1.13 Force Eligible........................................................................................ 408
28.5.1.14 Enqueue Rejections.............................................................................408
28.5.1.15 Order Restoration................................................................................ 409
28.5.2 QMan configuration interface........................................................................409
28.5.2.1 QMan device-tree node.......................................................................... 409
28.5.2.1.1 Frame Queue Descriptors...........................................................410
28.5.2.1.2 Packed Frame Descriptor Records.............................................410
28.5.2.1.3 Logical I/O Device Number (QMan)...........................................410
28.5.2.2 QMan pool channel device-tree node...................................................410
28.5.2.2.1 Channel ID................................................................................. 410
28.5.2.3 QMan portal device-tree node............................................................... 410
28.5.2.3.1 Portal Access to Pool Channels.................................................411
28.5.2.3.2 Stashing Logical I/O Device Number......................................... 411
28.5.2.3.3 Portal Initialization (QMan).........................................................411
28.5.2.3.4 Auto-Initialization........................................................................412
28.6 QMan portal APIs........................................................................................................ 412
28.6.1 QMan High-Level Portal Interface..................................................................... 412
28.6.1.1 Overview (QMan)...................................................................................412
28.6.1.2 Frame and Message Handling................................................................413
28.6.1.3 Portal management (QMan)................................................................... 413
28.6.1.3.1 Modifying interrupt-driven portal duties (QMan).........................413
28.6.1.3.2 Processing non-interrupt-driven portal duties (QMan)................ 414
28.6.1.3.3 Recovery support (QMan)..........................................................415
28.6.1.3.4 Stopping and restarting dequeues to the portal......................... 415
28.6.1.3.5 Manipulating the portal static dequeue command......................415
28.6.1.3.6 Determining if the enqueue ring is empty...................................416
28.6.1.4 Frame queue management....................................................................416
28.6.1.4.1 Querying a FQ object.................................................................417
28.6.1.4.2 Initialize a FQ............................................................................. 418
28.6.1.4.3 Schedule a FQ........................................................................... 419
28.6.1.4.4 Retire a FQ.................................................................................419
28.6.1.4.5 Put a FQ out of service.............................................................. 419
28.6.1.4.6 Query a FQD from QMan...........................................................420
28.6.1.4.7 Unscheduled (volatile) dequeuing of a FQ.................................420
28.6.1.4.8 Set FQ flow control state............................................................420
28.6.1.5 Enqueue Command (without ORP)....................................................... 421
28.6.1.6 Enqueue Command with ORP................................................................422
28.6.1.7 DCA Mode............................................................................................. 422
28.6.1.8 Congestion Management Records......................................................... 423
28.6.1.9 Zero-Configuration Messaging................................................................425
28.6.1.10 FQ allocation.........................................................................................426
28.6.1.10.1 Ad-hoc FQ allocator................................................................. 426
28.6.1.10.2 FQ range allocator................................................................... 426
28.6.1.10.3 Future FQ allocator changes.....................................................427
28.6.1.11 Helper functions.....................................................................................427
28.7 QMAN CEETM APIs..........................................................................................................428
28.7.1 QMAN CEETM Device-Tree Node..................................................................... 428
28.7.2 The token rate of CEETM shaper.......................................................................428
28.7.2.1 The token rate structure..........................................................................428
28.7.2.2 The APIs to convert token rate............................................................... 429
28.7.3 CEETM sub-portal.............................................................................................. 430
28.7.3.1 Claim/release sub-portal.........................................................................430
28.7.4 CEETM LNI - Logical Network Interface...............................................................431
28.7.4.1 Claim/release LNI.................................................................................. 431

28.7.4.2 Map sub-portal with LNI....431
28.7.4.3 Configure LNI shaper....432
28.7.4.4 Configure LNI traffic class flow control....434
28.7.5 CEETM class queue channel....434
28.7.5.1 Claim/release class queue channel....434
28.7.5.2 Configure the shaper of class queue channel....435
28.7.5.3 Configure the token limit as the weight for unshaped channel....436
28.7.5.4 Set CR/ER eligibility for CQs within a CEETM channel....437
28.7.6 CEETM class queue....437
28.7.6.1 Claim/release CQ....437
28.7.6.2 Change CQ weight....439
28.7.6.3 Query CQ statistics....440
28.7.7 CEETM Logical FQID....440
28.7.7.1 Claim/release LFQID....440
28.7.7.2 Configure/query dequeue context table....441
28.7.7.3 Create FQ for LFQ....441
28.7.8 CEETM Class Congestion Group(CCG)....442
28.7.8.1 Claim/release CCG....442
28.7.8.2 Configure CCG....443
28.7.8.3 Query CCG statistics....444
28.7.8.4 Set/get congestion state change notification target....444
28.8 Other QMan APIs....445
28.8.1 Waterfall Power Management....445
28.9 USDPAA-specific APIs....446
28.9.1 Overview....446
28.9.2 Thread initialization....446
28.9.3 FQID/BPID allocation....447
28.9.4 Interrupt handling....447
28.9.4.1 UIO file-descriptors....447
28.9.4.2 Processing interrupt-driven portal duties....447
28.9.5 Device-tree dependency....448
28.10 Sysfs and debugfs QMan/BMan interfaces....448
28.10.1 QMan sysfs....448
28.10.1.1 /sys/devices/ffe000000.soc/ffe318000.qman....448
28.10.1.2 /sys/devices/ffe000000.soc/ffe318000.qman/error_capture....448
28.10.1.3 /sys/devices/ffe000000.soc/ffe318000.qman/error_capture/sbec_< 0..6>....449
28.10.1.4 /sys/devices/ffe000000.soc/ffe318000.qman/sfdr_in_use....449
28.10.1.5 /sys/devices/ffe000000.soc/ffe318000.qman/pfdr_cfg....449
28.10.1.6 /sys/devices/ffe000000.soc/ffe318000.qman/idle_stat....449
28.10.1.7 /sys/devices/ffe000000.soc/ffe318000.qman/err_isr....450
28.10.1.8 /sys/devices/ffe000000.soc/ffe318000.qman/dcp< 0..3> _dlm_avg....450
28.10.1.9 /sys/devices/ffe000000.soc/ffe318000.qman/ci_rlm_avg....450
28.10.2 BMan sysfs....450
28.10.2.1 /sys/devices/ffe000000.soc/ffe31a000.bman....450
28.10.2.2 /sys/devices/ffe000000.soc/ffe31a000.bman/error_capture....450
28.10.2.3 /sys/devices/ffe000000.soc/ffe31a000.bman/error_capture/sbec_< 0..1>....451
28.10.2.4 /sys/devices/ffe000000.soc/ffe31a000.bman/pool_count....451
28.10.2.5 /sys/devices/ffe000000.soc/ffe31a000.bman/fbpr_fpc....451
28.10.2.6 /sys/devices/ffe000000.soc/ffe31a000.bman/err_isr....451
28.10.3 QMan debugfs....451
28.10.3.1 /sys/kernel/debug/qman....451
28.10.3.2 /sys/kernel/debug/qman/query_cgr....451
28.10.3.3 /sys/kernel/debug/qman/query_congestion....452
28.10.3.4 /sys/kernel/debug/qman/query_fq_fields....452
28.10.3.5 /sys/kernel/debug/qman/query_fq_np_fields....453

28.10.3.6 /sys/kernel/debug/qman/query_cq_fields........................................................454
28.10.3.7 /sys/kernel/debug/qman/query_ceetm_ccgr...................................................455
28.10.3.8 /sys/kernel/debug/qman/query_wq_lengths...................................................455
28.10.3.9 /sys/kernel/debug/qman/fqd/avoid_blocking_[enable | disable]......................456
28.10.3.10 /sys/kernel/debug/qman/fqd/prefer_in_cache_[enable | disable]..................456
28.10.3.11 /sys/kernel/debug/qman/fqd/cge_[enable | disable]....................................456
28.10.3.12 /sys/kernel/debug/qman/fqd/cpc_[enable | disable]....................................457
28.10.3.13 /sys/kernel/debug/qman/fqd/cred..................................................................457
28.10.3.14 /sys/kernel/debug/qman/fqd/ctx_a_stashing_[enable | disable]....................457
28.10.3.15 /sys/kernel/debug/qman/fqd/hold_active_[enable | disable]..........................458
28.10.3.16 /sys/kernel/debug/qman/fqd/orp_[enable | disable]......................................458
28.10.3.17 /sys/kernel/debug/qman/fqd/sfdr_[enable | disable]....................................458
28.10.3.18 sys/kernel/debug/qman/fqd/state_[active | oos | parked | retired | tentatively_sched | truly_sched]..................................................................... 459
28.10.3.19 /sys/kernel/debug/qman/fqd/tde_[enable | disable]......................................459
28.10.3.20 /sys/kernel/debug/qman/fqd/wq.................................................................. 459
28.10.3.21 /sys/kernel/debug/qman/fqd/summary.........................................................460
28.10.3.22 /sys/kernel/debug/qman/ccsrmempeek........................................................ 460
28.10.3.23 /sys/kernel/debug/qman/query_ceetm_xsfdr_in_use.................................. 461
28.10.4 BMan debugfs.........................................................................................................461
28.10.4.1 /sys/kernel/debug/bman.........................................................................................461
28.10.4.2 /sys/kernel/debug/bman/query_bp_state...................................................... 461
28.11 Error handling and reporting..........................................................................................463
28.11.1 Handling and Reporting..........................................................................................463
28.12 Operating system specifics............................................................................................463
28.12.1 Portal maintenance.................................................................................................464
28.12.2 Callback context..................................................................................................... 464
28.12.3 Blocking semantics.................................................................................................464

Chapter 29 QuadSPI Driver User Manual....................................................465
29.1 QuadSPI Driver User Manual..........................................................................................465

Chapter 30 QUICC Engine UCC UART User Manual..................................467
30.1 QUICC Engine UCC UART User Manual.......................................................................467

Chapter 31 QUICC Engine Time Division Multiplexing User Manual............471
31.1 QUICC Engine Time Division Multiplexing User Manual...............................................471

Chapter 32 Freescale Native SATA Driver User Manual.............................. 479
32.1 Freescale Native SATA Driver User Manual.................................................................. 479

Chapter 33 Security Engine (SEC) ..........................................................483
33.1 SEC Device Driver User Manual.................................................................................... 483

Chapter 34 User Enablement for Secure Boot - PBL Based Platforms........ 493
34.1 Preface........................................................................................................................... 493
34.2 Introduction.....................................................................................................................493
34.2.1 Purpose.................................................................................................................. 493
34.3 Secure Boot Process......................................................................................................494

34.4 Pre-Boot Phase........................................................................................................................ 495
34.5 ISBC Phase.............................................................................................................................497
34.5.1 Flow................................................................................................................................... 497
34.5.2 SRKs..................................................................................................................................498
34.5.3 Key Revocation..................................................................................................................498
34.5.4 Alternate Image support.................................................................................................... 499
34.5.5 ESBC with CSF Header.....................................................................................................499
34.6 ESBC Phase...........................................................................................................................500
34.6.1 Boot script.........................................................................................................................501
34.6.1.1 Where to place the boot script?........................................................................... 501
34.6.1.2 Chain of Trust....................................................................................................... 501
34.6.1.2.1 Sample Boot Script.............................................................................. 502
34.6.1.3 Chain of Trust with Confidentiality....................................................................... 503
34.6.1.3.1 Sample Encap Boot Script................................................................... 504
34.6.1.3.2 Sample Decap Boot Script...................................................................505
34.7 Next Executable Phase.......................................................................................................... 505
34.8 CST Tool................................................................................................................................. 505
34.8.1 Code Signing Tool Walkthrough........................................................................................505
34.8.2 KEY GENERATION...........................................................................................................507
34.8.2.1 gen_otpmk_drbg.................................................................................................. 507
34.8.2.1.1 Features...............................................................................................507
34.8.2.2 gen_drv_drbg....................................................................................................... 508
34.8.2.2.1 Features...............................................................................................509
34.8.2.3 gen_keys..............................................................................................................510
34.8.2.3.1 Features...............................................................................................510
34.8.3 CSF HEADER GENERATION...........................................................................................511
34.8.3.1 Verbose Mode.......................................................................................................511
34.8.3.2 Public Key/ SRK Hash Generation Only...............................................................512
34.8.3.3 Combined Usage of Various Options...................................................................513
34.8.3.4 Help......................................................................................................................513
34.8.3.5 Default Usage....................................................................................................... 513
34.8.3.5.1 Sample Input File and Output............................................................... 513
34.8.3.6 Key Extension......................................................................................................516
34.8.3.6.1 Introduction.......................................................................................... 516
34.8.3.6.2 How it works.........................................................................................516
34.8.3.6.3 IE Key Structure...................................................................................517
34.8.3.6.4 Sample Input File and Output............................................................... 518
34.8.3.6.5 Generate Header for Next Level Images............................................... 521
34.8.3.7 Image Hash Generation.......................................................................................523
34.8.3.7.1 Introduction.......................................................................................... 523
34.8.3.7.2 Features...............................................................................................524
34.8.3.8 Import Signature................................................................................................... 524
34.8.3.8.1 Introduction.......................................................................................... 524
34.8.3.8.2 Example............................................................................................... 524
34.8.3.9 Signature Calculation...........................................................................................525
34.8.3.9.1 Introduction.......................................................................................... 525
34.8.3.9.2 Example............................................................................................... 525
34.8.3.10 Signature Embedding......................................................................................... 525
34.8.3.10.1 Introduction........................................................................................ 525
34.8.3.10.2 Example............................................................................................. 526
34.9 Product execution...................................................................................................................526
34.9.1 Getting started.................................................................................................................. 526
34.9.1.1 Environment for Secure Boot...............................................................................526
34.9.1.2 SDK Images required for the demo..................................................................... 527
34.9.2 Chain of Trust.................................................................................................................... 527

34.9.2.1 Other Images Required for demo.......... 527
34.9.2.2 Bootscript and Signing the images.......... 528
34.9.2.2.1 Signing the images using same key pair.......... 529
34.9.2.2.2 Signing the images using different key pair.......... 530
34.9.2.3 Running secure boot (Chain of Trust).......... 532
34.9.3 Chain of Trust with Confidentiality.......... 533
34.9.3.1 Other Images Required for demo.......... 533
34.9.3.2 Encap Bootscript.......... 534
34.9.3.3 Decap Bootscript.......... 534
34.9.3.4 Creating CSF Headers.......... 535
34.9.3.5 Running secure boot (Chain of Trust with Confidentiality).......... 535
34.9.4 NAND Secure Boot (Chain of Trust).......... 537
34.9.4.1 Running NAND Secure boot (Chain of Trust).......... 538
34.9.5 NAND Secure Boot (Chain of Trust with Confidentiality).......... 539
34.9.5.1 Running NAND Secure boot (Chain of Trust with Confidentiality).......... 540
34.10 Troubleshooting.......... 542
34.11 CSF Header Data Structure Definition.......... 542
34.12 ISBC Validation error codes.......... 557
34.13 ESBC Validation error codes.......... 562
34.14 Address Map used for demo.......... 563
34.15 Useful U-Boot and CCS Commands.......... 568
34.16 Trust Architecture and SFP Information.......... 572
34.17 QCVS Tool Usage.......... 573

## Chapter 35 Universal Serial Bus Interfaces.......... 581

35.1 USB 3.0 Host/Peripheral Linux Driver User Manual.......... 581

## Chapter 36 USDPAA User Guide.......... 591

36.1 Introduction.......... 591
36.1.1 Intended audience.......... 591
36.1.2 USDPAA overview.......... 591
36.1.3 USDPAA and legacy Linux software.......... 592
36.1.3.1 Legacy user space applications.......... 592
36.1.3.2 Relationship to conventional kernel-based drivers.......... 592
36.2 USDPAA assumptions and use cases.......... 593
36.2.1 Assumptions.......... 593
36.2.1.1 General assumptions.......... 593
36.2.2 Use cases.......... 593
36.2.2.1 Run-to-completion.......... 593
36.2.2.2 Interrupt-driven.......... 594
36.3 USDPAA components.......... 594
36.3.1 Device-tree handling.......... 594
36.3.1.1 Device-tree initialization requirements.......... 594
36.3.2 QMan and BMan drivers and C API.......... 595
36.3.2.1 QMan driver overview.......... 595
36.3.2.2 QMan portals and the Linux device-tree.......... 596
36.3.2.3 Note on the current implementation.......... 597
36.3.2.4 Portal initialization requirements.......... 597
36.3.2.5 Buffer Manager (BMan).......... 597
36.3.2.6 Raw Portal APIs.......... 597
36.3.3 DMA memory management.......... 598
36.3.3.1 Current USDPAA solution.......... 598
36.3.3.2 DMA memory API.......... 599

36.3.4 Network configuration........................................................................................................ 599
36.3.4.1 Network configuration initialisation requirements...........................................599
36.3.5 CPU isolation........................................................................................................ 600
36.4 Relationship to SDK Linux ethernet subsystem........................................................................ 601
36.4.1 Selecting ethernet interfaces for USDPAA.................................................................. 601
36.4.2 FMC, FMD, and the ethernet Driver........................................................................ 603
36.5 Supported hardware platforms.................................................................................................. 603
36.5.1 P4080DS........................................................................................................ 603
36.5.2 P3041DS........................................................................................................ 604
36.5.3 P5020DS........................................................................................................ 604
36.5.4 P5040DS........................................................................................................ 604
36.5.5 P2041RDB........................................................................................................605
36.5.6 B4860QDS........................................................................................................605
36.5.7 T4240QDS........................................................................................................605
36.6 Example applications........................................................................................................605
36.7 USDPAA installation and execution........................................................................................ 606
36.7.1 Files needed to boot Linux on the P4080DS system.........................................606
36.7.2 About U-Boot and network interfaces.........................................................................606
36.7.3 P4080DS NOR flash banks........................................................................................ 609
36.7.4 Programming the P4080DS NOR flash bank 4.................................................... 609
36.7.5 Boot into bank 4 and set more variables................................................................ 610
36.7.6 Environment variable hwconfig and optical 10G...........................................................611
36.7.7 Booting Linux........................................................................................................611
36.7.8 Using tftp for the kernel, device-tree, and file-system...................................................612
36.8 Using configurations other than SerDes 0xe..............................................................................612
36.8.1 SGMII (4 x 1 Gbps) card and one XAUI (10 Gbps) card.............................................. 612
36.8.2 SGMII (4 x 1 Gbps) card and no XAUI (10 Gbps) card............................................... 613
36.9 Known limitations of this release............................................................................................... 613
36.10 Document history........................................................................................................614

## Chapter 37 USDPAA Multiple Processor Support User Guide..................... 617

37.1 USDPAA Multiple Process Support........................................................................................617
37.2 USDPAA User/Kernel Device Interface........................................................................................617
37.3 USDPAA Resource Management............................................................................................. 618
37.4 BMan and QMan API........................................................................................................620
37.5 USDPAA Thread and Global API............................................................................................... 622
37.6 USDPAA DMA API........................................................................................................ 623
37.7 USDPAA netcfg.h........................................................................................................625
37.8 Kernel configuration........................................................................................................625
37.9 Device Tree (Excluding QMan/BMan Resource Ranges)........................................................... 626
37.10 Device Tree (QMan/BMan Resource Ranges)..........................................................................626
37.11 USDPAA Boot Arguments........................................................................................................ 628
37.12 USDPAA Virtualisation and Partitioning ...................................................................................... 629
37.13 Multi-process PPAC Applications.............................................................................................. 629
37.14 Limitations........................................................................................................ 631

## Chapter 38 USDPAA Reflector and PPAC User Guide SDK........................633

38.1 Introduction........................................................................................................633
38.1.1 Intended audience........................................................................................................ 633
38.1.2 Change history........................................................................................................634
38.2 Overview of reflector........................................................................................................634
38.3 Overview of PPAC........................................................................................................634
38.4 PPAC details........................................................................................................ 635

38.4.1 IRQ mode for sleeping when idle....................635
38.4.2 Buffers....................635
38.4.3 Compile-time configuration....................636
38.4.3.1 Order preservation....................636
38.4.3.2 Monitoring Rx/Tx fill-levels via CGR....................637
38.4.3.3 Other settings....................637
38.5 Running reflector....................638
38.6 PPAC (and reflector) CLI commands....................639
38.7 Running hello_reflector....................640
38.8 Running hello_reflector....................640
38.9 Testing reflector....................641

Chapter 39 Freescale USDPAA IPFWD User Manual Rev. 1.2....................643
39.1 About this Book....................643
39.2 Introduction....................643
39.2.1 Purpose....................643
39.3 Overview....................643
39.3.1 USDPAA IPv4 forwarding application flow....................643
39.3.1.1 Overview of packet flow:....................644
39.4 Overview of PPAC....................644
39.5 IPFwd related PPAC Details....................645
39.5.1 Compile-time configuration....................645
39.5.1.1 Order Preservation in IPFWD....................645
39.5.1.2 Order Restoration in IPFWD....................645
39.5.1.3 Monitoring Rx/Tx fill-levels and flow-control via CGR....................646
39.6 PPAM related compile time configuration....................648
39.6.1 One million route support....................648
39.7 IPFWD Application Suite....................650
39.8 Possible configuration scenario for IPFWD....................651
39.9 Using Two Computers to Test the IPFWD Application Suite....................656
39.10 Flowchart for packet processing....................659
39.10.1 Description of Flow chart....................659
39.10.2 Running IPv4 forwarding on P4080DS board....................660
39.10.3 Running IPv4 forwarding on P3041/P5020 board....................663
39.10.4 Running IPv4 forwarding on T4240 board....................663
39.10.5 Running IPv4 forwarding on B4860 board....................664
39.10.6 Performance gap between 8 core and 6 core....................665
39.10.7 PPAC (and IPFwd) CLI commands....................665
39.11 IPv4 forward application Configuration command....................666
39.11.1 Syntax....................666
39.11.1.1 Command to show all enabled interfaces and their interface numbers....................666
39.11.1.2 Help for show all enabled interfaces command....................667
39.11.1.3 Assign IP address to interfaces....................668
39.11.1.4 Help for assign IP address to interfaces....................668
39.11.1.5 Adding a Route Entry....................669
39.11.1.6 Help for Route Entry Addition....................669
39.11.1.7 Deleting a Route Entry....................670
39.11.1.8 Help for Deleting a Route Entry....................670
39.11.1.9 Starting the Application Processing ....................670
39.11.1.10 Adding an ARP Entry....................671
39.11.1.11 Help for ARP Entry Addition....................671
39.11.1.12 Deleting an ARP Entry....................671
39.11.1.13 Help for Deleting an ARP Entry....................672
39.12 Traffic Generation....................672

39.13 References....................673

Chapter 40 Freescale USDPAA IPSecfwd User Manual.................... 675

40.1 Introduction....................675
40.1.1 Purpose....................675
40.1.2 Change History....................675
40.2 USDPAA IPSecfwd application....................675
40.2.1 Application Overview....................676
40.2.2 Packet Flow....................676
40.2.3 Overview of IPSecfwd packet processing....................677
40.2.3.1 Outbound processing:....................677
40.2.3.2 Inbound Processing:....................678
40.2.4 Flow chart for IpSecfwd packet processing....................681
40.3 Overview of PPAC....................682
40.4 IPSecfwd related PPAM Details....................682
40.4.1 In-Place Encryption/Decryption....................682
40.5 Secfwd application suite....................684
40.5.1 Using Two Computers to Test the IPFWD Application Suite....................685
40.5.2 Running IPSecfwd on P4080DS board....................687
40.5.3 Running IPv4 forwarding on P3041/P5020 board....................689
40.5.4 Running IPv4 forwarding on T4240 board....................689
40.5.5 Running IPv4 forwarding on B4860 board....................690
40.5.6 PPAC (and IPSecfwd) CLI commands....................691
40.5.7 IPSecfwd application Configuration command....................691
40.5.7.1 Syntax....................691
40.5.7.1.1 Command to show all enabled interfaces and their interface numbers692
40.5.7.1.2 Help for show all enabled interfaces command....................693
40.5.7.1.3 Assign IP address to interfaces....................693
40.5.7.1.4 Help for assign IP address to interfaces....................693
40.5.7.1.5 Adding an SA Entry....................694
40.5.7.1.6 Help for SA Entry Addition....................695
40.5.7.1.7 Deleting an SA Entry....................696
40.5.7.1.8 Help for Deleting an SA Entry....................696
40.5.7.1.9 Adding a Route Entry....................697
40.5.7.1.10 Help for Route Entry Addition....................697
40.5.7.1.11 Deleting a Route Entry....................698
40.5.7.1.12 Help for Deleting a Route Entry....................698
40.5.7.1.13 Adding an ARP Entry....................699
40.5.7.1.14 Help for ARP Entry Addition....................699
40.5.7.1.15 Deleting an ARP Entry....................699
40.5.7.1.16 Help for Deleting an ARP Entry....................700
40.5.7.1.17 Adding an ARP Entry....................700
40.6 References....................701
40.7 Revision History....................701

Chapter 41 Freescale Simple Crypto User Manual....................703

41.1 Introduction....................703
41.2 USDPAA Simple Crypto Application....................703
41.2.1 Overview....................703
41.2.2 Parameters to the application....................703
41.2.3 Packet Flow....................705
41.2.4 Throughput calculation....................706
41.2.5 Running Simple Crypto Application on board....................706

41.2.6 Simple Crypto command syntax....706
41.2.7 Snapshot of Simple Crypto output....708

Chapter 42 Freescale Simple Proto User Manual....711
42.1 Introduction....711
42.2 USDPAA Simple Proto Application....711
42.3 Overview....711
42.4 Parameters to the application....711
42.5 Packet Flow....713
42.6 Throughput calculation....714
42.7 Running Simple Proto Application on board....714
42.8 Simple Proto command syntax....715
42.9 MACSec protocol options....716
42.10 WiMAX protocol options....716
42.11 PDCP protocol options....717
42.12 RSA operations options....719
42.13 TLS protocol options....719
42.14 IPsec protocol options....719
42.15 MBMS protocol options....720

Chapter 43 SEC Descriptor construction library (DCL)....723
43.1 SEC Descriptor construction library (DCL)....723
43.2 DCL Description....723
43.3 DCL Packaging....723
43.4 DCL Files....723
43.5 DCL Functional Description....724
43.6 Command Generator....724
43.7 Descriptor Disassembler....724
43.8 Upper-Tier DCL Descriptor Constructors....725
43.9 API Reference....725
43.9.1 API Reference Command Generator....725
43.9.1.1 cmd_insert_shared_hdr()....725
43.9.1.2 cmd_insert_hdr()....726
43.9.1.3 cmd_insert_key()....727
43.9.1.4 cmd_insert_seq_key()....728
43.9.1.5 cmd_insert_proto_op_ipsec()....729
43.9.1.6 cmd_insert_proto_op_wimax()....729
43.9.1.7 cmd_insert_proto_op_wifi()....729
43.9.1.8 cmd_insert_proto_op_macsec()....730
43.9.1.9 cmd_insert_proto_op_unidir()....730
43.9.1.10 cmd_insert_alg_op()....730
43.9.1.11 cmd_insert_pkha_op()....731
43.9.1.12 cmd_insert_seq_in_ptr()....731
43.9.1.13 cmd_insert_seq_out_ptr()....732
43.9.1.14 cmd_insert_load()....732
43.9.1.15 cmd_insert_seq_load()....733
43.9.1.16 cmd_insert_fifo_load()....733
43.9.1.17 cmd_insert_seq_fifo_load()....734
43.9.1.18 cmd_insert_store()....734
43.9.1.19 cmd_insert_seq_store()....735
43.9.1.20 cmd_insert_fifo_store()....736
43.9.1.21 cmd_insert_seq_fifo_store()....736
43.9.1.22 cmd_insert_jump()....737

43.9.1.23 cmd_insert_math()........................................................................................ 737
43.9.1.24 cmd_insert_move()........................................................................................738
43.9.2 Descriptor Constructors........................................................................................738
43.9.2.1 Job Descriptor Constructors........................................................................... 739
43.9.2.1.1 cnstr_seq_jobdesc()........................................................................... 739
43.9.2.1.2 cnstr_jobdesc_blkcipher_cbc()........................................................... 739
43.9.2.1.3 cnstr_jobdesc_hmac()......................................................................... 740
43.9.2.1.4 cnstr_jobdesc_mdsplitkey()................................................................740
43.9.2.1.5 cnstr_jobdesc_aes_gcm()...................................................................741
43.9.2.1.6 cnstr_jobdesc_kasumi_f8()..................................................................742
43.9.2.1.7 cnstr_jobdesc_kasumi_f9()..................................................................742
43.9.2.1.8 cnstr_jobdesc_pkha_rsaexp()..............................................................743
43.9.2.1.9 cnstr_jobdesc_dsaverify()....................................................................743
43.9.2.2 Protocol/Shared Descriptor Constructors........................................................ 744
43.9.2.2.1 cnstr_pcl_shdesc_ipsec_cbc_decap()................................................ 744
43.9.2.2.2 cnstr_pcl_shdesc_ipsec_cbc_encap()................................................ 745
43.9.2.2.3 cnstr_shdsc_ipsec_encap()................................................................746
43.9.2.2.4 cnstr_shdsc_ipsec_decap()................................................................746
43.9.2.2.5 cnstr_shdsc_wifi_encap().................................................................... 747
43.9.2.2.6 cnstr_shdsc_wifi_decap().................................................................... 748
43.9.2.2.7 cnstr_shdsc_wimax_encap().............................................................. 748
43.9.2.2.8 cnstr_shdsc_wimax_decap().............................................................. 749
43.9.2.2.9 cnstr_shdsc_macsec_encap()............................................................750
43.9.2.2.10 cnstr_shdsc_macsec_decap()..........................................................750
43.9.2.2.11 cnstr_shdsc_snow_f8()......................................................................751
43.9.2.2.12 cnstr_shdsc_snow_f9()......................................................................752
43.9.2.2.13 cnstr_shdsc_kasumi_f8()...................................................................753
43.9.2.2.14 cnstr_shdsc_kasumi_f9()...................................................................754
43.9.2.2.15 cnstr_shdsc_cbc_blkcipher()............................................................ 754
43.9.2.2.16 cnstr_shdsc_hmac()......................................................................... 755
43.9.2.2.17 cnstr_pcl_shdsc_3gpp_rlc_decap()................................................... 756
43.9.2.2.18 cnstr_pcl_shdsc_3gpp_rlc_encap()................................................... 757
43.9.3 Disassembler........................................................................................................ 758
43.9.3.1 caam_desc_disasm()........................................................................................ 758

Chapter 44 Runtime Assembler Library Reference......................................759
44.1 Runtime Assembler Library Reference.........................................................................759

Chapter 45 USDPAA IPFwd Longest Prefix Match User Manual................. 761
45.1 Freescale P4080/P5020/P3041 USDPAA IPFwd Longest Prefix Match User Manual................ 761
45.1.1 Introduction........................................................................................................761
45.1.2 Overview.............................................................................................................761
45.1.3 How is it different from existing Route cache based IPFwd?.........................................762
45.1.4 Longest Prefix Match algorithm ........................................................................... 762
45.1.5 Shared MAC Overview......................................................................................... 764
45.1.6 How to run shared MAC interface ?.......................................................................765
45.1.7 MAC-less use case..............................................................................................767
45.1.8 How to ping MAC-less interface ?......................................................................... 767
45.1.9 USDPAA LPM based IPv4 forwarding application flow................................................ 768
45.1.10 Overview of packet flow:......................................................................................768
45.1.11 Overview of PPAC............................................................................................... 769
45.1.12 Compile-time configuration.................................................................................. 769
45.1.13 Order Preservation in LPM-IPFWD......................................................................... 769

45.1.14 Order Restoration in LPM IPFWD.......... 769
45.1.15 Monitoring Rx/Tx fill-levels and flow-control via CGR.......... 770
45.1.16 LPM IPFWD Application Suite.......... 772
45.1.17 Possible configuration scenario for LPM based IPFWD.......... 773
45.1.18 Using Two Computers to Test the IPFWD Application Suite.......... 778
45.1.19 Running LPM IPv4 forwarding on P4080DS board.......... 781
45.1.20 Running LPM IPv4 forwarding on P3041/P5020 board.......... 783
45.1.21 USDPAA LPM IP Fwd performance gap between 6 core and 8 core.......... 784
45.1.22 PPAC (and IPFwd) CLI commands.......... 784
45.1.23 Syntax.......... 785
45.1.24 Command to show all enabled interfaces and their interface numbers.......... 786
45.1.25 Help for show all enabled interfaces command.......... 786
45.1.26 Assign IP address to interfaces.......... 787
45.1.27 Help for assign IP address to interfaces.......... 788
45.1.28 Adding a Route Entry.......... 788
45.1.29 Help for Route Entry Addition.......... 789
45.1.30 Deleting a Route Entry.......... 789
45.1.31 Help for Deleting a Route Entry.......... 789
45.1.32 Adding an ARP Entry.......... 789
45.1.33 Help for ARP Entry Addition.......... 790
45.1.34 Deleting an ARP Entry.......... 790
45.1.35 Help for Deleting an ARP Entry.......... 791
45.1.36 References.......... 791

## Chapter 46 Watchdog Timers.......... 793

46.1 Watchdog Device Driver User Manual.......... 793

## Chapter 47 Open Data Plane (ODP) User Guide.......... 797

47.1 Introduction.......... 797
47.1.1 Intended Audience.......... 797
47.1.2 Definitions and Acronyms.......... 797
47.2 Test Setup.......... 798
47.3 ODP generator sample application (odp_generator).......... 798
47.4 ODP pktio sample application (odp_pktio).......... 801
47.5 ODP ipsec sample applications (odp_ipsec, odp_ipsec_proto).......... 803
47.6 ODP ‘packet classify’ sample application (odp_pkt_classify).......... 805
47.7 ODP timer sample application (odp_timer_test).......... 807
47.8 ODP L3 forwarding sample application (odp_l3_forwarding).......... 809
47.9 Appendix A. References.......... 810
47.10 Release Notes.......... 811

## Chapter 48 UEFI.......... 813

48.1 Introduction.......... 813
48.1.1 UEFI Boot Flow on LS1043.......... 813
48.1.2 Primary Protected Application.......... 815
48.1.3 How to Compile UEFI Image in Yocto.......... 819
48.1.4 SCT Overview.......... 820
48.1.5 How to Compile Linux Image in Yocto.......... 821
48.2 LS1043A UEFI Hardware.......... 822
48.2.1 Switch Settings.......... 822
48.2.2 RCW (Reset Configuration Word) and Ethernet Interfaces.......... 823
48.2.3 Memory Map.......... 825

48.2.3.1 LS1043A NAND Flash memory map for UEFI Bootloader.............................. 826
48.3 Booting UEFI.......................................................................................................................... 827
48.3.1 Booting to UEFI prompt via NOR Flash.......................................................................... 827
48.3.2 Booting to UEFI prompt via NAND Flash........................................................................ 834
48.4 LS1043 UEFI Software.......................................................................................................... 836
48.4.1 FAT32 Filesystem........................................................................................................... 836
48.4.2 Using Commands on the UEFI Shell.............................................................................. 839
48.4.3 Platform Configuration Database (PCD's) Used in LS1043 UEFI Implementation......... 841
48.4.4 How to Access BlockIO and I2C Devices....................................................................... 842

# Chapter 1 SDK Overview

## 1.1 What's New

Freescale Digital Networking is pleased to announce the release of Linux SDK for LS1043A, v0.5 supporting LS1043A rev1.0 processor on RDB board.

Highlights

- This release is incremental to v0.4.
- This release is still Beta release. The next SDK will be production release.

Processor and Board Support

- No Change from v0.4 release

Yocto and Toolchain

- No Change from v0.4 release

U-Boot Boot Loader

- Core frequency upgraded to 1.6GHz
- PSCI in u-boot: LPM20
- PPA integration
- PSCI in PPA: cpu_on/off, system reset

UEFI Boot Loader

- Non-XIP boot: SD
- PCIe NIC card enablement
- PPA – CPU OFF working with Linux
- Rebasing on latest master/shell versions from EDK2
- SMMU-500 in bypass mode
- UEFI_STUB support to boot Linux via UEFI

Linux Kernel Core and Virtualization

- Big endianess kernel

Linux Kernel Device Drivers

- DPAA Ethernet: macless support
- Fman classification (offline port)
- USB 3.0 (gadget with USB2.0 host )
- Thermal monitor: reading temperature through sysfs interface

User Space Datapath Acceleration Architecture (USDPAA) and Reference Applications

- IPsec ESN, high bandwidth
- LPM IPfwd

- Multi-processes
- Order preservation/Restoration
- Options in ppac framework: max frame size, MAC address modification, promiscuous mode, loopback mode
- DPAA offload
  - Classification (IPv4 and IPv6)
  - Header manipulations

Open Data Plane (ODP)

- Supports ODP API v1.3.0
- Implement Thread Grouping for the Scheduler
- Additional User Space ODP sample applications
  - odp_ipsec_proto
  - odp_l3_forwarding

Other Tools and Utilities

- No Change from v0.4 release

For a list of known issues for this release, see “Known and Fixed Issues” in the Overview section.

## 1.2 Components

Top-level components LS1043A SDK

- Yocto
- GNU Toolchain
- U-Boot Boot Loader
- UEFI Boot Loader
- Linux Kernel and Virtualization
- Linux Kernel and Device Drivers
- User Space Datapath Acceleration Architecture (USDPAA) and Applications
- Other Tools and Utilities

Yocto

- Yocto/Poky 1.6.1 "Daisy"
- 64-bit user space

GNU Toolchain

- arm: gcc-linaro-4.8.3-2014.04, eglibc-linaro-2.19-r2014.04, binutils-linaro-2.24-r2014.03, gdb-linaro-7.6.1-r2013.10
- Based on sources from Free Software Foundation

U-Boot Bootloader

- U-Boot: 2015.01
- OCRAM

- DUART, DDR4, I2C, PCIe, USB 3.0, eSDXC
- DSPI
- IFC access to NOR and NAND flash
- Boot from NOR, NAND flash, SD
- Networking support using FMAN Independent Mode
- NOR Secure Boot (ESBC)
- PPA integration, PSCI cpu_on/off, system reset
- PSCI in u-boot: LPM20
- QSGMII and XFI port
- QTA

UEFI Bootloader

- ARMv8 Core, CCI-400 and MMU support
- Boot from IFC NOR and NAND
- DDR4, DUART, I2C, eSDXC
- DSPI flash
- IFC NAND and NOR flash
- Interrupt model (GICv2 Configuration, ARMv8 Generic Timer interrupts and DUART Interrupts)
- Non-XIP boot: SD
- PCIe NIC card enablement
- PPA (EL3->EL2 transition), CPU OFF working with Linux
- Real Time Clock
- Rebasing on latest master/shell versions from EDK2
- SMMU-500 in bypass mode
- TZASC-380, CSU
- UEFI_STUB support to boot Linux via UEFI
- UEFI Shell
- Watchdog

Linux Kernel Core and Virtualization

- Linux kernel 3.19
- Little endianess and big endianess kernel
- SMP support
- ARM v8
- 64-bit effective addressing
- Linux Containers (LXC): osimple container creation, attaching, cloning, destroying, ocgroup tests and network tests
- KVM
  - Basic support (SMP, Oversubscription)

- I/O support: virtio-net with both virtio-mmio and virtio-pci, virtio-blk with file on the host , libvirt (basic boot + virtio net)

Linux Kernel Device Drivers

- Crypto driver supporting SEC 5 (CAAM): JR / QI / simple crypto / proto / IPSec
- DUART, DSPI, I2C
- Ethernet DPAA, macless support
- Flextimer
- Frame Manager (FMan), classification (offline port)
- Frame Manager uCode
- IEEE1588
- Integrated Flash Controller (IFC) NOR and NAND flash
- GIC-v2
- GPIO
- OCRAM
- PCIe
- PHY support: RGMII, XFI and QSGMII
- Power Management (PM) - CPU_ON/CPU_OFF(CPU hotplug) implementation
- QDMA
- Queue Manager (QMan) and Buffer Manager (BMan)
- QUICC Engine TDM, HDLC and PPPoHT
- Secured Digital Host Controller (eSDHC) and eSDXC support
- Thermal monitor: reading temperature through sysfs interface
- Universal Serial Bus (USB) 3.0, host and gadget with USB2.0 host
- Watchdog Timers

User Space Data Path Acceleration Architecture (USDPAA)

- Device-tree handling
- QMan and BMan drivers and C API
- DMA Memory Management
- Network Configuration
- CPU Isolation
- Drivers - BMan, QMan

USDPAA Reference Applications

- DPAA offload
  - Classification (IPv4 and IPv6)
  - Header manipulations
- Hello Reflector
- IP Forward (route cache)

- IP Forward (longest prefix match)
- IPSec Forward, ESN and high bandwidth
- LPM IPfwd
- Multi-processes
- PPAC Reflector
- Order preservation/Restoration
- Options in ppac framework: max frame size, MAC address modification, promiscuous mode, loopback mode
- Simple Crypto
- Simple Proto

Open Data Plane (ODP)

- Supports ODP API v1.3.0
- User Space ODP sample applications

Other Tools and Utilities

- FLIB/RTA - SEC descriptor creation library
- Frame Manager Configuration Tool (FMC) [DPAA processors]
- Power monitor
- UEFI FAT32 filesystem (source to be downloaded seperately)

## 1.3 Known Issues

Known issues for Linux SDK for LS1043A v0.5

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

Table 1: Known Issues

| ID | Description | Disposition | Opened In | Resolved In | Workaround |
|---|---|---|---|---|---|
| QUSDPA-797 | USDPAA IPFwd performance does not scale up from 2 cores to 4 cores. | Open | LS1043A SDK v0.5 | | |
| ODP-143 | The tunnel mode is not supported by odp_ipsec application | Open | LS1043A SDK v0.5 | | |
| QUSDPA-762 | When injecting 128B packet traffic at line rate, USDPAA RC IPFwd application will stop forwarding packets in short time. | Resolved | LS1043A SDK v0.3 | LS1043A SDK v0.5 | |

*Table continues on the next page...*

Table 1: Known Issues (continued)

| ID | Description | Disposition | Opened In | Resolved In | Workaround |
|---|---|---|---|---|---|
| QLINUX-3751 | Offline port is not supported in this release. | Resolved | LS1043A SDK v0.3 | LS1043A SDK v0.5 | |
| QLINUX-3740 | There are RX errors received with multiple netperf streams on the XFI port. | Resolved | LS1043A SDK v0.3 | LS1043A SDK v0.5 | |

# Chapter 2
# Getting Started with Yocto Project

Yocto Project is an open-source collaboration project for embedded Linux developers. Yocto Project uses the Poky build system to make Linux images. For complete information about Yocto Project, see About Yocto Project in the QorIQ SDK Knowledge Center.

## 2.1 Essential Build Instructions

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

See Additional Instructions for Developers for more information on using Yocto Project.

### 2.1.1 Install the SDK

How to install Yocto Project on the host machine.

1. Mount the ISO on your machine:

```
$ sudo mount -o loop Linux-LS1043A-SDK-<version>-<target>-<yyyymmdd>-yocto.iso /mnt/
cdrom
```

2. As a non-root user, install Yocto Project:

```
$ /mnt/cdrom/install
```

3. When you are prompted to input the install path, ensure that the current user has the correct permission for the install path.

There is no uninstall script. To uninstall Yocto Project, you can remove the <yocto_install_path>/Linux-LS1043A-SDK-<version>-<yyyymmdd>-yocto directory manually.

NOTE

* The source ISO contains the package source tarballs and yocto recipes. It can be installed and used to do non-cache build.

* The cache ISO contains the pre-built cache binaries. To avoid a long time build, you can install the source ISO and the cache ISO in the same installation folder.

* The image ISO includes all prebuilt images: flash images, standalone toolchain installer, HD rootfs images and small images.

* The source ISO can be used separately. The core ISO and the source ISO should work together.

### 2.1.2 Prepare the Host Environment

Yocto requires some packages to be installed on host.

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

1. 
```
$ cd <yocto_install_path>
```

2. 
```
$ ./poky/scripts/host-prepare.sh
```

The script "*host-prepare.sh*" is used to install the required packages on your host machine. Root permission and Internet access are required to run the script. The script only needs to be run once on each host.

Please follow instructions below to install Python 2.7.x in a custom path instead of overriding the system default python installation. If you override the system default python installation, some system utilities may no longer operate.

#### More Information on the "*host-prepare.sh*" script:

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

```
$ wget https://www.python.org/ftp/python/2.7.6/Python-2.7.6.tar.xz
[NOTE: Python 2.7.3 and python 2.7.5 can be used as well.]
$ tar -xf Python-2.7.6.tar.xz
$ cd Python-2.7.6
$ ./configure --prefix=/opt/python-2.7.6
$ make
$ sudo make install

Please run below export command to ensure python 2.7.x is used for Yocto build.
$ export PATH=/opt/python-2.7.6/bin:$PATH
```

Yocto Project supports typical Linux distributions:Ubuntu, Fedora, CentOS, Debian, OpenSUSE, etc. More Linux distributions are continually being verified. This SDK has been verified on following Linux distributions: Ubuntu 14.04, centos-7.0.1406,Mint-15,Debian 7.6, Fedora 20 and OpenSUSE 13.2

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

The following is the detailed package list on the Redhat and Centos hosts:

```
$ sudo yum groupinstall "Development Tools"
$ sudo yum install tetex gawk sqlite-devel vim-common redhat-lsb xz
python-devel  zlib-devel perl-String-CRC32 dos2unix python m4 make wget curl ftp
tar bzip2 gzip unzip perl texinfo texi2html diffstat openjade docbook-style-dsssl
sed docbook-style-xsl
docbook-dtds docbook-utils bc glibc-devel pcre pcre-devel groff linuxdoc-tools
patch cmake tcl-devel gettext ncurses apr SDL-devel  libtool xterm
mesa-libGL-devel mesa-libGLU-devel gnome-doc-utils autoconf automake
```

For the Fedora hosts:

```
$ sudo yum groupinstall "Development Tools"
$ sudo yum install tetex gawk sqlite-devel vim-common redhat-lsb xz
python-devel  zlib-devel perl-String-CRC32 dos2unix python m4 make wget curl ftp
tar bzip2 gzip unzip perl texinfo texi2html diffstat openjade
docbook-style-dsssl sed docbook-style-xsl docbook-dtds docbook-utils
bc glibc-devel pcre pcre-devel groff linuxdoc-tools patch cmake
```

```
tcl-devel gettext ncurses apr SDL-devel mesa-libGL-devel  xterm
mesa-libGLU-devel gnome-doc-utils autoconf automake libtool
$ sudo yum install ccache quilt perl-ExtUtils-MakeMaker ncurses-devel
```

For Ubuntu and Debian hosts:

```
$ sudo dpkg-reconfigure --terse -f readline dash
$ sudo apt-get install sed wget subversion git-core coreutils unzip
texi2html texinfo libsdl1.2-dev docbook-utils fop gawk python-pysqlite2
diffstat make gcc build-essential xsltproc g++ desktop-file-utils
chrpath libgl1-mesa-dev libglu1-mesa-dev autoconf automake groff libtool
xterm libxml-parser-perl vim-common xz-utils cvs tofrodos
libstring-crc32-perl ubuntu-minimal ubuntu-standard patch
libbonobo2-common libncurses5-dev
```

Extra packages are needed for Ubuntu-64b:

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

For OpenSUSE host:

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

## 2.1.3 Set Up Poky

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

```
$ source ./poky/fsl-setup-poky -m <machine>
```

For example:

```
$ source ./poky/fsl-setup-poky -m ls1043ardb
```

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

Usage:

```
source fsl-setup-poky <-m machine> [-l] [-h]
```

Where:

- `<-m machine>` is mandatory; `[-j jobs]`, `[-t tasks]`, `[-s string]` ,`[-p]`, `[-l]` and `[-h]` are optional
  - Supported Layerscape (ARM) machines: ls1043ardb
- `[-j jobs]`: number of jobs for make to spawn during the compilation stage.
- `[-t tasks]`: number of BitBake tasks that can be issued in parallel.
- `[-d path]`: non-default path of DL_DIR (downloaded source)
- `[-c path]`: non-default path of SSTATE_DIR (shared state Cache)
- `[-b path]`: non-default path of project folder (build_${machine}_release)

- `[-l]`: lite mode. To help conserve disk space, deletes the building directory once the package is built.
- `[-s string]`: append an extra string to project folder.
- `[-p]`: append fsl cache and source mirrors (For FSL Internal Use Only)
- `[-h]`: help

## 2.1.4 Builds

How to set up a cross compile environment and perform builds

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

1. ```
   $ cd <yocto_install_path>/build_<machine>_release/
   ```
2. ```
   $ bitbake <image-target>
   ```

   Where `<image-target>` is one of the following:

   - `fsl-image-minimal`: contains basic packages to boot up a board
   - `fsl-image-core`: contains common open source packages and FSL specific packages.
   - `fsl-image-full`: contains all packages in the full package list.
   - `fsl-image-flash`: contains all the user space apps needed to deploy the fsl-image-flash image to a USB stick, hard drive, or other large physical media.
   - `fsl-image-virt`: contains toolkit to interact with the virtualization capabilities of Linux
   - `fsl-image-kernelitb`: A FIT image comprising the Linux image, dtb and rootfs image
   - `fsl-image-x11`: Freescale image with a very basic X11 image with a terminal
   - `fsl-toolchain`: the cross compiler binary package
   - `uefi-ls1043a`: uefi binary package
   - `ppa`: ppa binary package
   - `package-name(usdpaa)`: build a specific package

NOTE

The UEFI build depends on FatPkg. FatPkg source code must been downloaded

a. git clone git://git.code.sf.net/p/tianocore/edk2-FatPkg

b. $ cd edk2-FatPkg

c. $ $ git reset --hard 8ff136aaa3fff82d81514fd3091961ec4a63c873

**Contents of the Built Images Directory:**

A Yocto build produces images that will be located in the following directory:

```
<yocto_install_patch>/build_<machine>_release/tmp/deploy/images/<machine>/
```

The following list shows the typical directory/image files (exact contents depend on the setting of the `<IMAGE_FSTYPES>` variable):

- `fsl-image-<machine>.ext2.gz.u-boot` - ramdisk image that can be loaded with U-Boot
- `fsl-image-<machine>.ext2.gz` - gzipped ramdisk image

- `fsl-image-<machine>.tar.gz` - gzipped tar archive of the image
- `kernel-<machine>.itb` - kernel (itb).
- `Image-<machine>.bin` - kernel binary of the image
- `u-boot-<machine>.bin` - U-Boot binary image that can be programmed into board Flash
- `Image-<machine>.dtb` - device tree binary (dtb).
- `fsl_fman_ucode_<machine>_<version>.bin` - fman ucode for <machine> board
- `hv/hv.uImage` - uImage for hypervisor
- `hv-cfg/*/*/hv.dtb` - dtb for hypervisor
- `rcw/*/rcw_*.bin` - rcw
- `ppa.itb` - PPA
- `<machine>_EFI.fd` - Uefi image

NOTE

For additional Yocto usage information, please refer to the "About Yocto" chaptelr.

## 2.2 Additional Instructions for Developers

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

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

### 2.2.1 U-Boot

How to configure and rebuild the U-Boot

1. Modify U-Boot source code
   a. `$ bitbake -c cleansstate u-boot`

      NOTE

      You can use the following shortcuts:

      `bitbake -c clean <target>`

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

      `bitbake -c cleansstate <target>`

      - Removes work directory in `build_<machine>_release/tmp/work`
      - Removes cache files in `<yocto_install_path>/sstate-cache/` directory.

      `bitbake -c cleanall <target>`

      - Removes work directory in `build_<machine>_release/tmp/work`
      - Removes cache files in `<yocto_install_path>/sstate-cache/` directory.
      - Removes the source code in `<yocto_install_path>/sources/` directory

   b. `$ bitbake -c patch u-boot`

c. `$ cd <S>` and modify the source code

> **NOTE**
> Use `bitbake -e <package-name> | grep ^S=` to get value of <S>.

2. Modify U-Boot configuration.
   a. `$ modify UBOOT_MACHINES`

      Values for `UBOOT_MACHINES` are listed in `meta-fsl-arm/conf/machine/<machine>.conf`

      e.g. `UBOOT_MACHINES = "ls1021atwr"`
3. Rebuild U-Boot image
   a. `$ cd build_<machine>_release`
   b. `$ bitbake -c compile -f u-boot`
   c. `$ bitbake u-boot`

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

## 2.2.2 Linux Kernel

How to Configure and Rebuild the Linux Kernel

1. Modify kernel source code
   a. `$ bitbake -c cleansstate virtual/kernel`
   b. `$ bitbake -c patch virtual/kernel`
   c. `$ cd <S>` and change the source code

> **NOTE**
> Use `bitbake -e <package-name> | grep ^S=` get the value of `<S>`.

2. Change the kernel defconfig
   a. `$ update KERNEL_DEFCONFIG variable in meta-fsl-arm/conf/machine/<machine>.conf`
3. Change dts
   a. `$ update KERNEL_DEVICETREE variable in meta-fsl-arm/conf/machine/<machine>.conf`
4. Do menuconfig
   a. `$ bitbake -c menuconfig virtual/kernel`

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

NOTE

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

For a non-X11 environment:

- `OE_TERMINAL = "screen"`

The following commands can be used for the other environments:

For a GNOME environment (default):

- `OE_TERMINAL = "gnome"`

For a KDE environment:

- `OE_TERMINAL = "konsole"`

For non-GNOME and non-KDE environments:

- `OE_TERMINAL = "xterm"`

5. Rebuild Kernel image
   a. `$ cd build_<machine>_release`
   b. $ bitbake -c compile -f virtual/kernel
   c. `$ bitbake virtual/kernel`

NOTE

Kernel images can be found in `build_<machine>_release/tmp/deploy/images/<machine>/`

## 2.2.3 Packages

How to Patch and Rebuild a Package

1. `$ cd <yocto_install_path>/build_<machine>_release`
2. `$ bitbake -c cleansstate <package-name>`
3. `$ cd <RECIPE_FOLDER>`

NOTE

Use bitbake <package-name> -e | grep ^FILE_DIR to get the value of <RECIPE_FOLDER>

4. `$ mkdir -p <RECIPE_FOLDER>/files`
5. Copy patch into `<RECIPE_FOLDER>/files`
6. Modify `<BB_FILE>` and add follow content in `package-name-<version>.bb` file

```
SRC_URI += "file://<name-of-patch1> \
                  file://<name-of-patch2> \
                              ... \
                  file://<name-of-patchn>"
```

7. Rebuild this package:

```
$ bitbake <package-name>
```

### 2.2.4 Customize Root Filesystem

How to Customize a Root Filesystem

Packages included in a rootfs can be customized by editing the corresponding recipe:

```
fsl-image-flash:meta-fsl-networking/images/fsl-image-flash.bb
fsl-image-core:meta-fsl-networking/images/fsl-image-core.bb
fsl-image-full:meta-fsl-networking/images/fsl-image-full.bb
fsl-image-virt: meta-fsl-networking/images/fsl-image-virt.bb
fsl-image-x11:meta-fsl-networking/images/fsl-image-x11.bb
fsl-image-kernelitb: meta-fsl-networking/images/fsl-image-kenelitb.bb
fsl-image-minimal:meta-fsl-networking/images/fsl-image-minimal.bb
fsl-toolchain:meta-fsl-networking/images/fsl-toolchain.bb
```

The rootfs type can be customized by setting the IMAGE_FSTYPES variable in the above recipes.

Supported rootfs types include the following:

```
cpio
cpio.gz cpio.xz
cpio.lzma
cramfs
ext2
ext2.gz
ext2.gz.u-boot
ext2.bz2.u-boot
ext3
ext3.gz.u-boot
ext2.lzma
jffs2
live
squashfs
squashfs-lzma
ubi
tar
tar.gz
tar.bz2
tar.xz
```

Path of source tarballs and patches:

- Package source tarballs are in the folder named `sources`, which is in the same folder level as `build_<machine>_release`.
- Patches are in the corresponding recipe folder

Specify the preferred version of package:

`<PREFERRED_VERSION_pkgname>` is used to configure the required version of a package.

If `<PREFERRED_VERSION>` is not defined, Yocto will pick up the recent version. For example, to downgrade Samba from 3.4.0 to 3.1.0: add `PREFERRED_VERSION_samba = "3.1.0"` in `meta-fsl-ppc/conf/machine/<machine>.conf`

Rebuild rootfs:

```
$ bitbake <image-target>
```

### 2.2.5 Native Packages

How to Build Native Packages for the Host

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

```
$ bitbake cst-native
```

NOTE

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

### 2.2.6 Extract Source Code

How to Extract the Source Code for a Package

To extract the source code of a package, do the following:

1. `$ bitbake -c cleansstate <package-name>`
2. `$ bitbake -c patch <package-name>`
3. `$ cd <S>`

NOTE

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

For example, to do a U-boot of a LS1043ARDB processor.

1. `$ bitbake -c cleansstate u-boot`
2. `$ bitbake -c patch u-boot`

NOTE

U-boot source code is installed in the folder: `build_ls1043ardb_release/tmp/work/ls1043aqds-fsl-linux/u-boot-ls1043a/2015.01+git-r0/`

### 2.2.7 Standalone toolchain

Build and install the standalone toolchain with Yocto:

1. ```
   $ source ./fsl-setup-poky -m <machine>
   ```
2. ```
   $ bitbake fsl-toolchain
   ```
3. ```
   $ cd build_<machine>_release/tmp/deploy/sdk
   ```
4. ```
   $ ./fsl-networking-eglibc-<host-system>-<core>-toolchain-<release>.sh
   ```

NOTE

The default installation path for standalone toolchain is `/opt/fsl-networking/`. The install folder can be specified during the installation procedure.

To use the installed toolchain, go the the location where the toolchain is installed and source the `environment-setup-<core>` file. This will set up the correct path to the build tools and also export some environment variables relevant for development (eg. `$CC`, `$ARCH`, `$CROSS_COMPILE`, `$LDFLAGS` etc).

To invoke the compiler, use the $CC variable (eg. `$CC <source files>`).

> NOTE
>
> This is a sysrooted toolchain. This means that GCC will start to look for target fragments and libraries (eg. crt*, libgcc.a) starting from the path specified by the sysroot. The default sysroot is preconfigured at build time to point to `/opt/fsl-networking/QorIQ-SDK-<sdk_version>/sysroots/<target_architecture>`. If the toolchain is installed in a location other than the default one (`/opt/fsl-networking/`), the `--sysroot=<path_to_target_sysroot>` parameter needs to be passed to GCC. When invoking the compiler through the `$CC` variable, there is no need to pass the `--sysroot` parameter as it is already included in the variable (check by running `echo $CC`).

## 2.2.8 Shared State (sstate) Cache

The shared state cache (sstate-cache), as pointed to by SSTATE_DIR.

1. Use the following setting in local.conf:

```
SSTATE_DIR="<absolute_path_of_cache_folder>"
e.g. SSTATE_DIR = "/home/yocto/sdk/sstate-cache/"
```

2. Some packages have no caches because the ISO uses 4GB of space. Thus, cache size is limited by ISO size. Some packages (e.g. u-boot, kernel, rcw, hv-cfg, fmc, depmodwrapper-cross, keymaps, base-files, merge-files, shadow-securetty, etc.) have no caches and building them requires about 20 minutes.

## 2.2.9 How to use hob

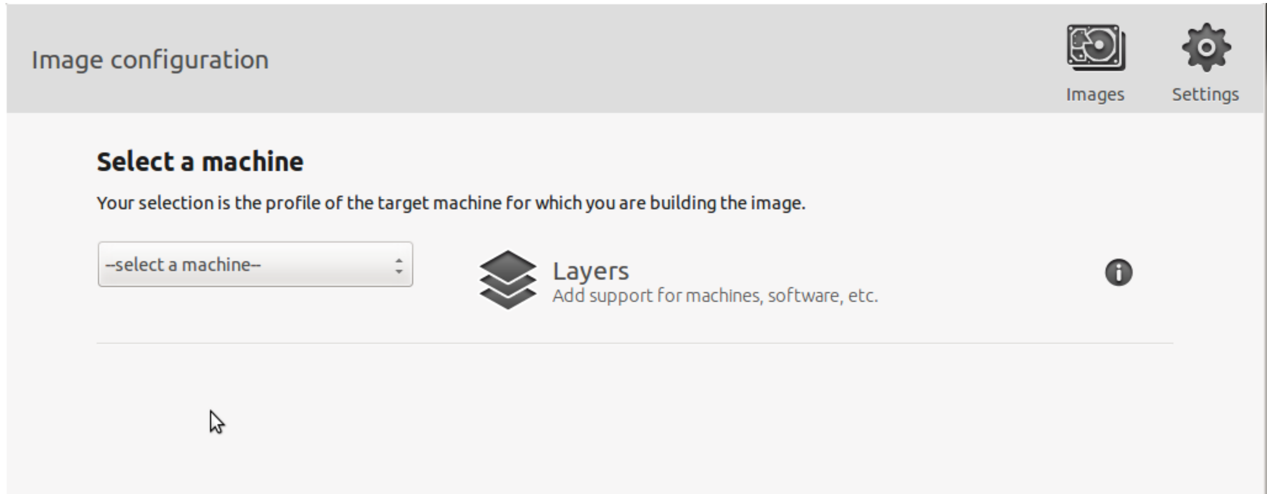
Hob is the graphical user interface for Yocto. Its primary goal is to enable a user to perform common tasks more easily.

1. Following is an example for hob usage

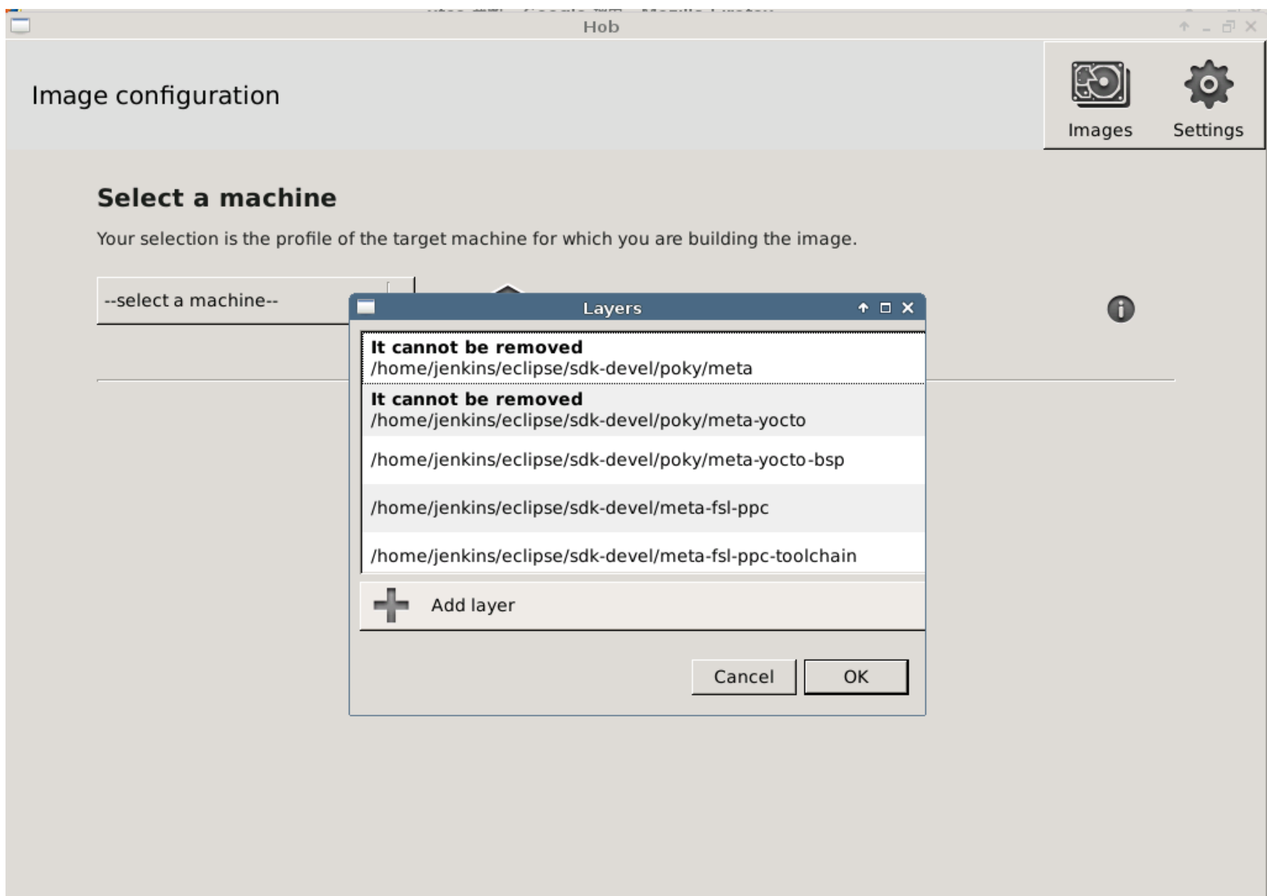
   $ source ./<yocto_install_path>/build_<machine>_release/SOURCE_THIS

   $ hob

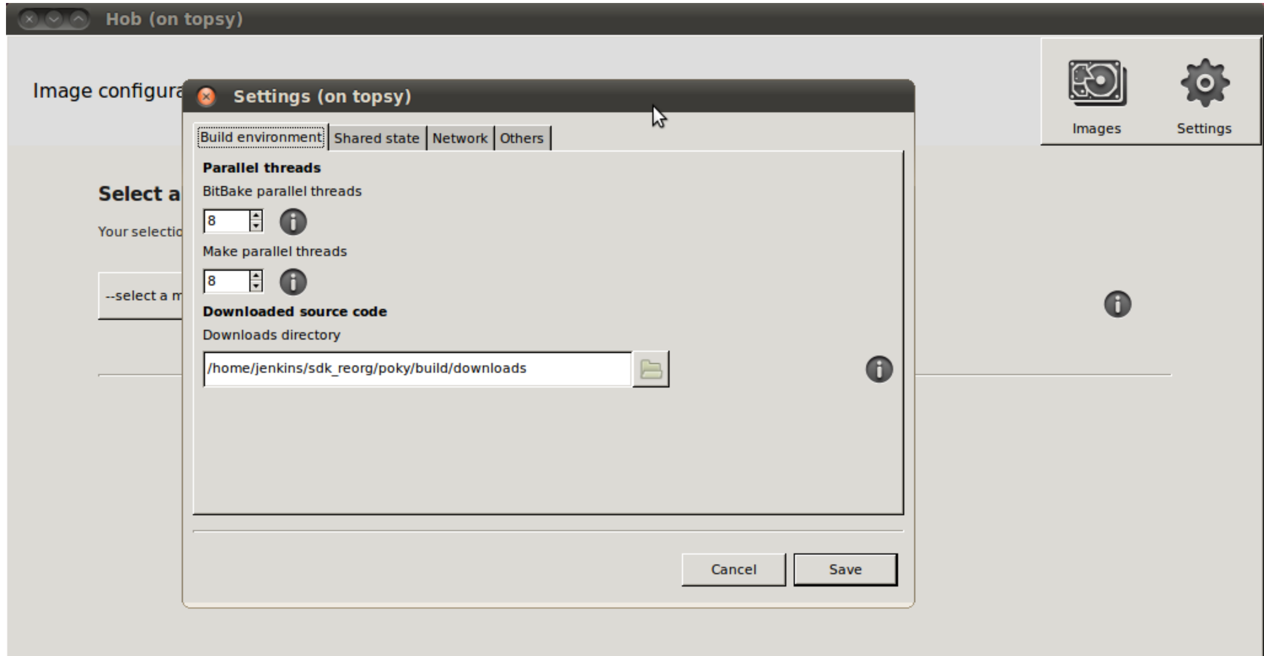
   the main screen for Hob appears:



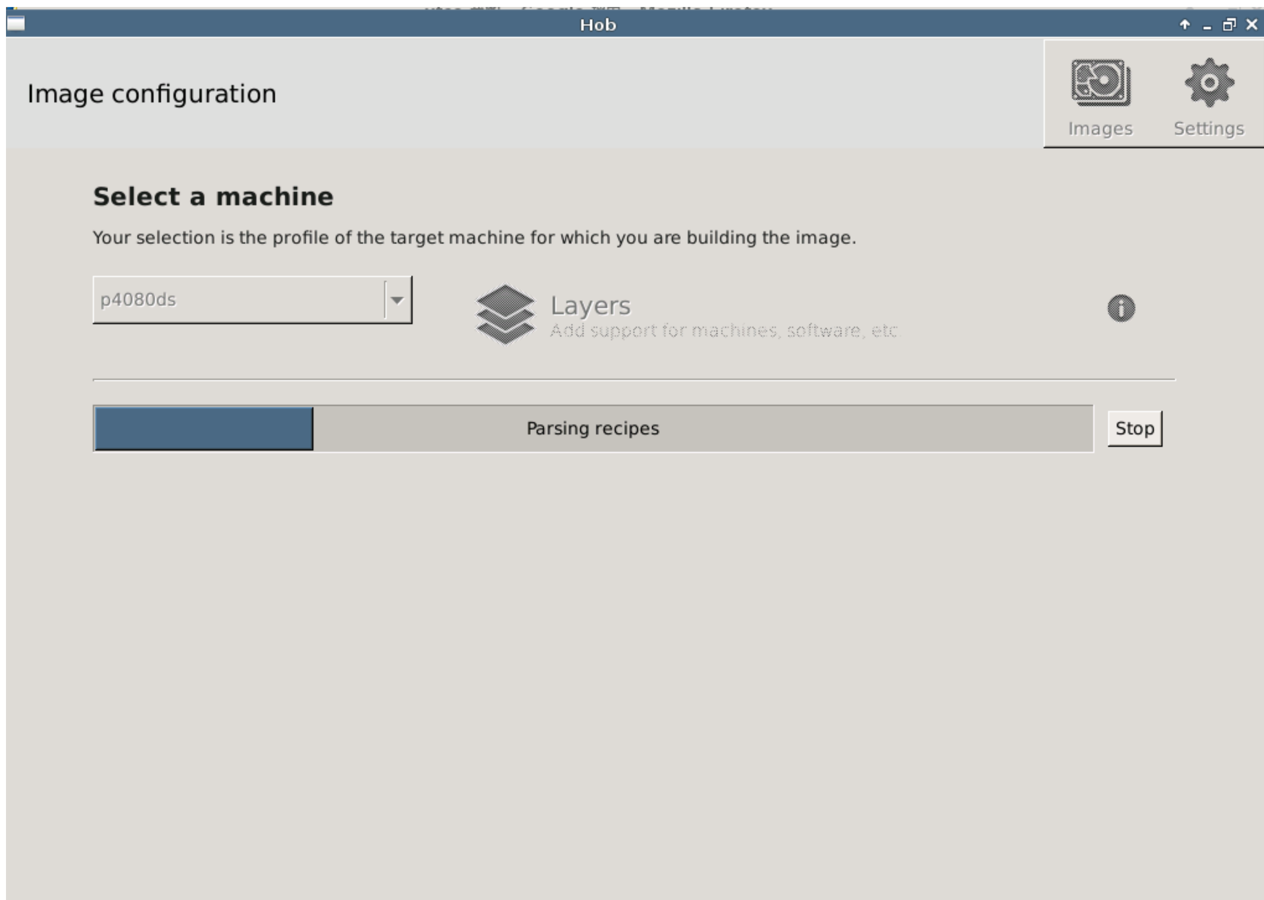

To get help information, roll your mouse over controls and buttons, or click the information icons throughout the interface. Here is an example that shows the help information for Layers::



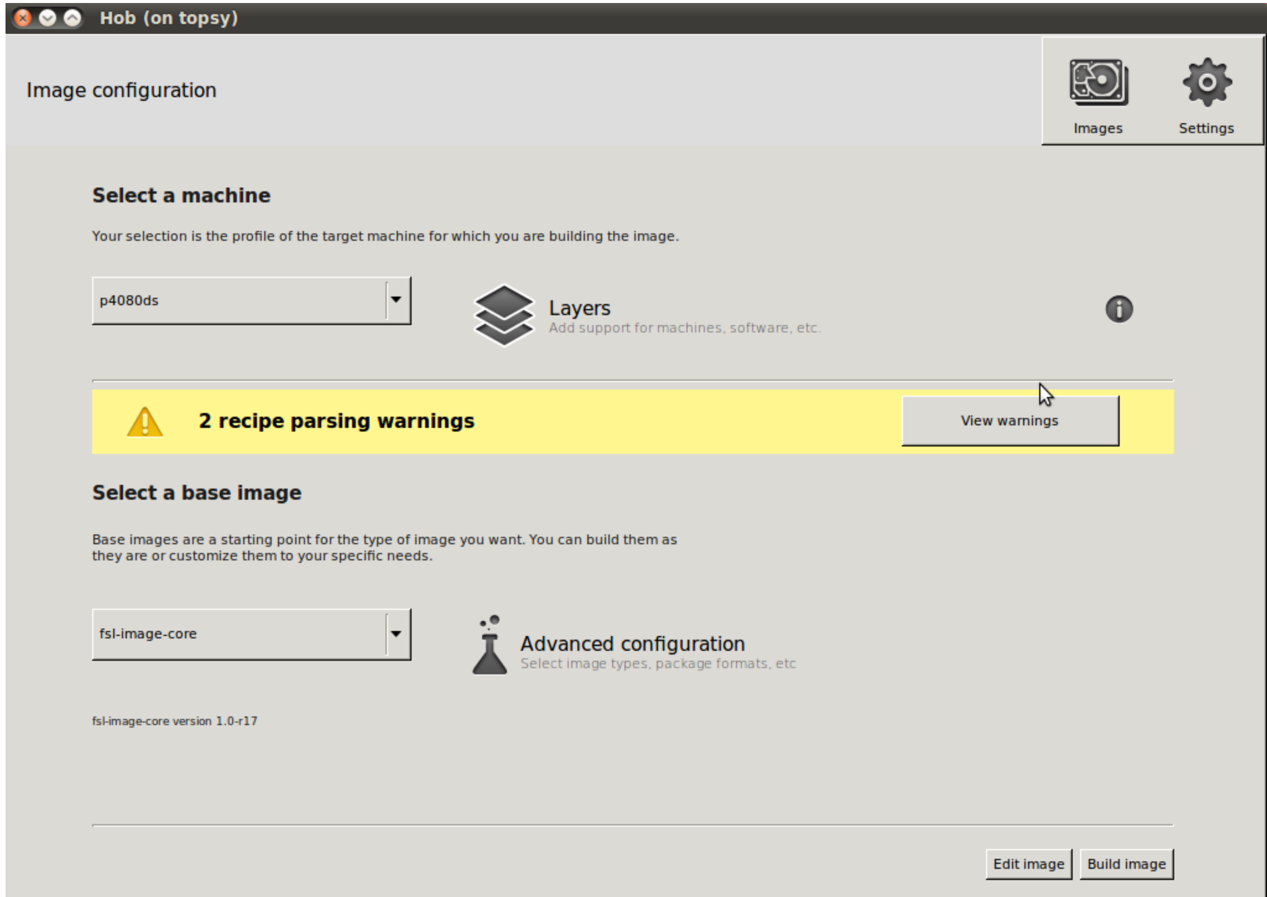

Once Hob has launched, you should go to the "Settings" dialog and be sure everything is set up for how you want your builds to occur. Clicking the "Settings" icon in the top-right corner of the main window reveals the "Settings" dialog:



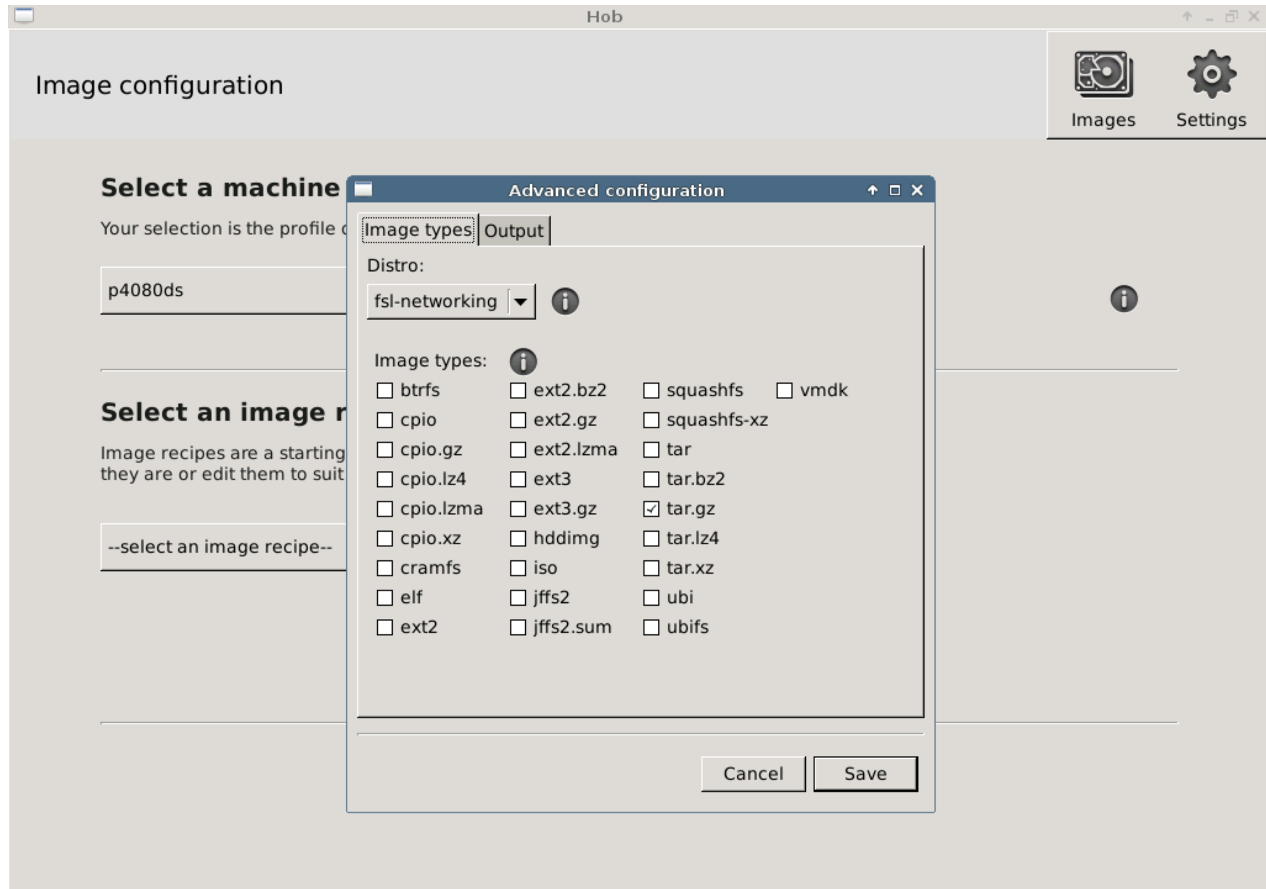

Once you pick the target machine, recipes are parsed:



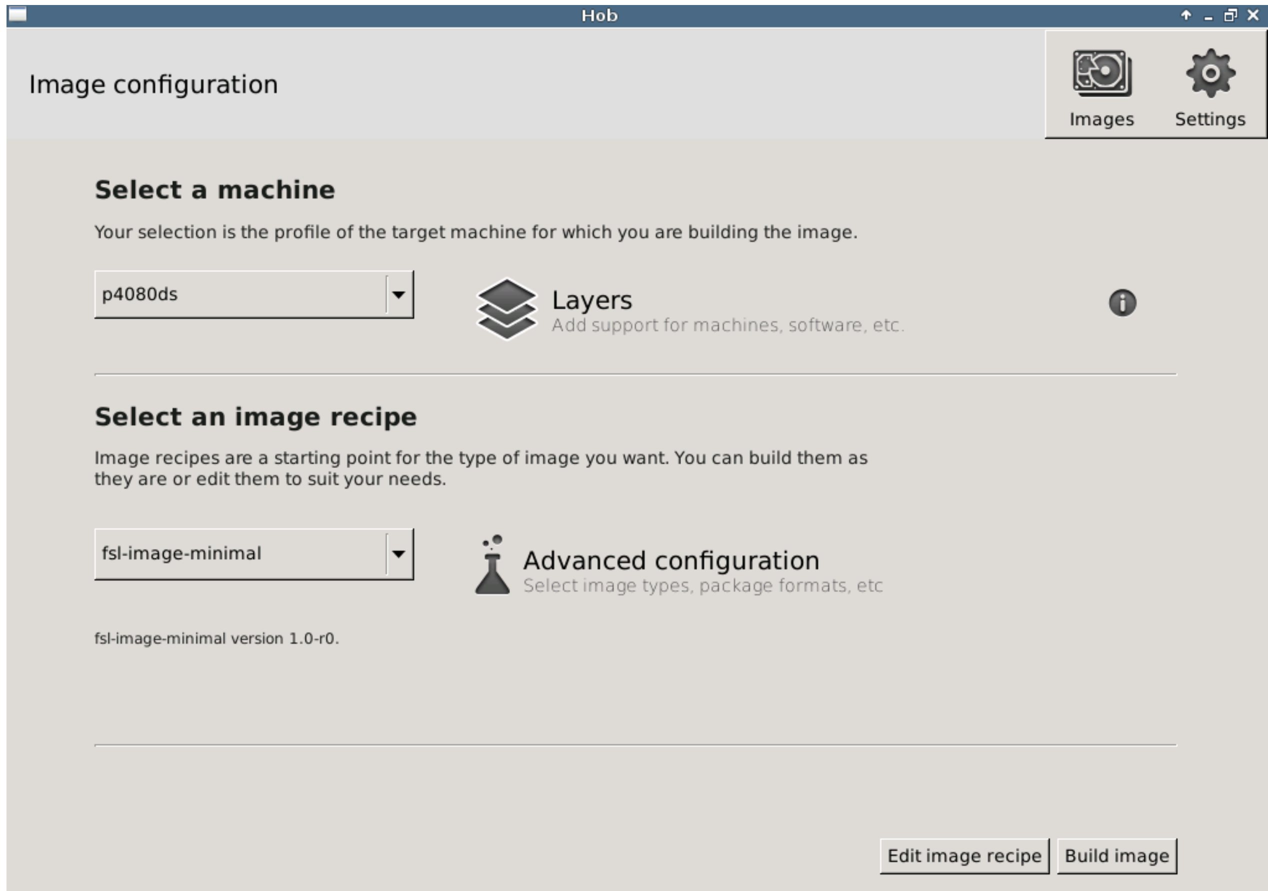

Once your settings are established and saved, you can pick the base image recipe :



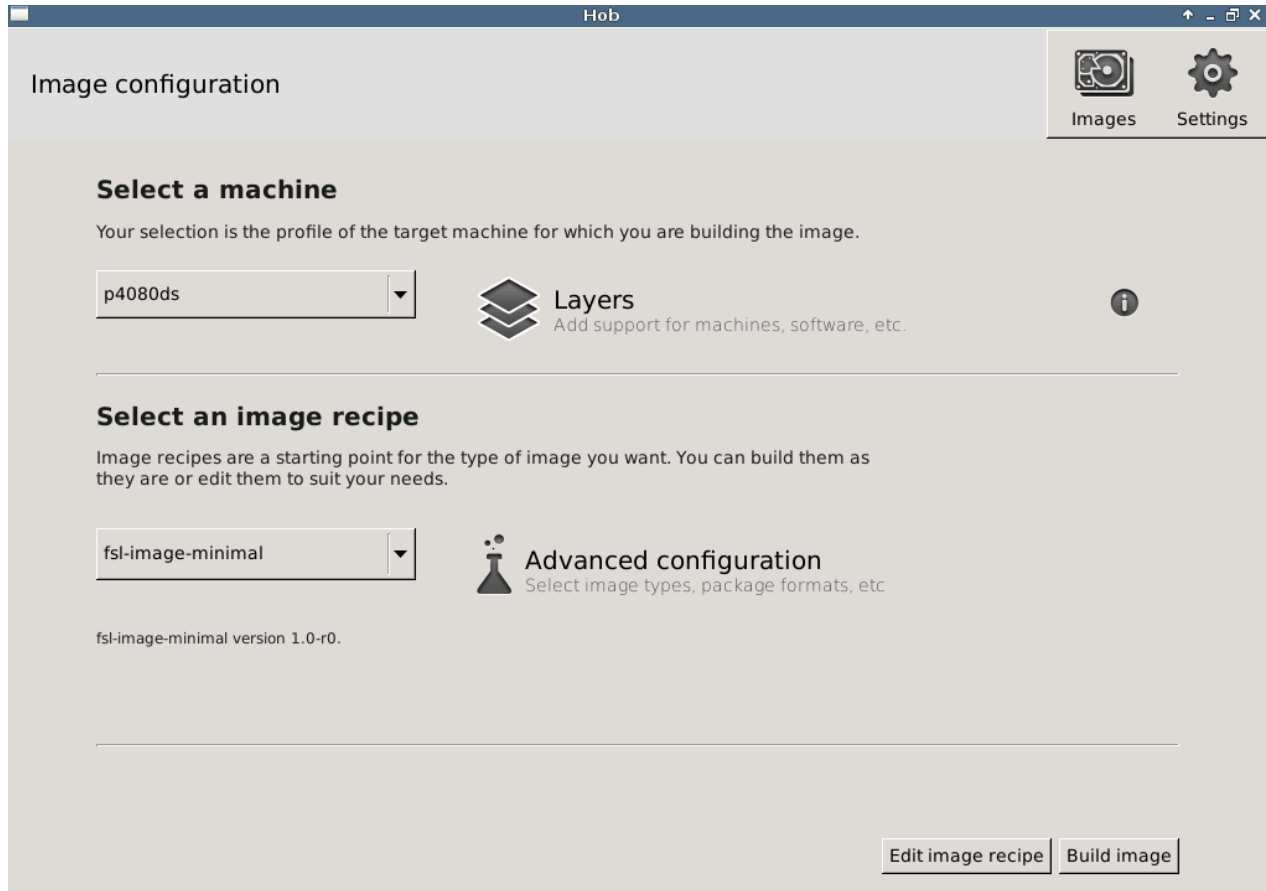

After the recipes are parsed, you can set image-related options using the "Advanced configuration" dialog. Clicking the button next to the "Select a base image" drop down menu reveals the "Advanced configuration" dialog:



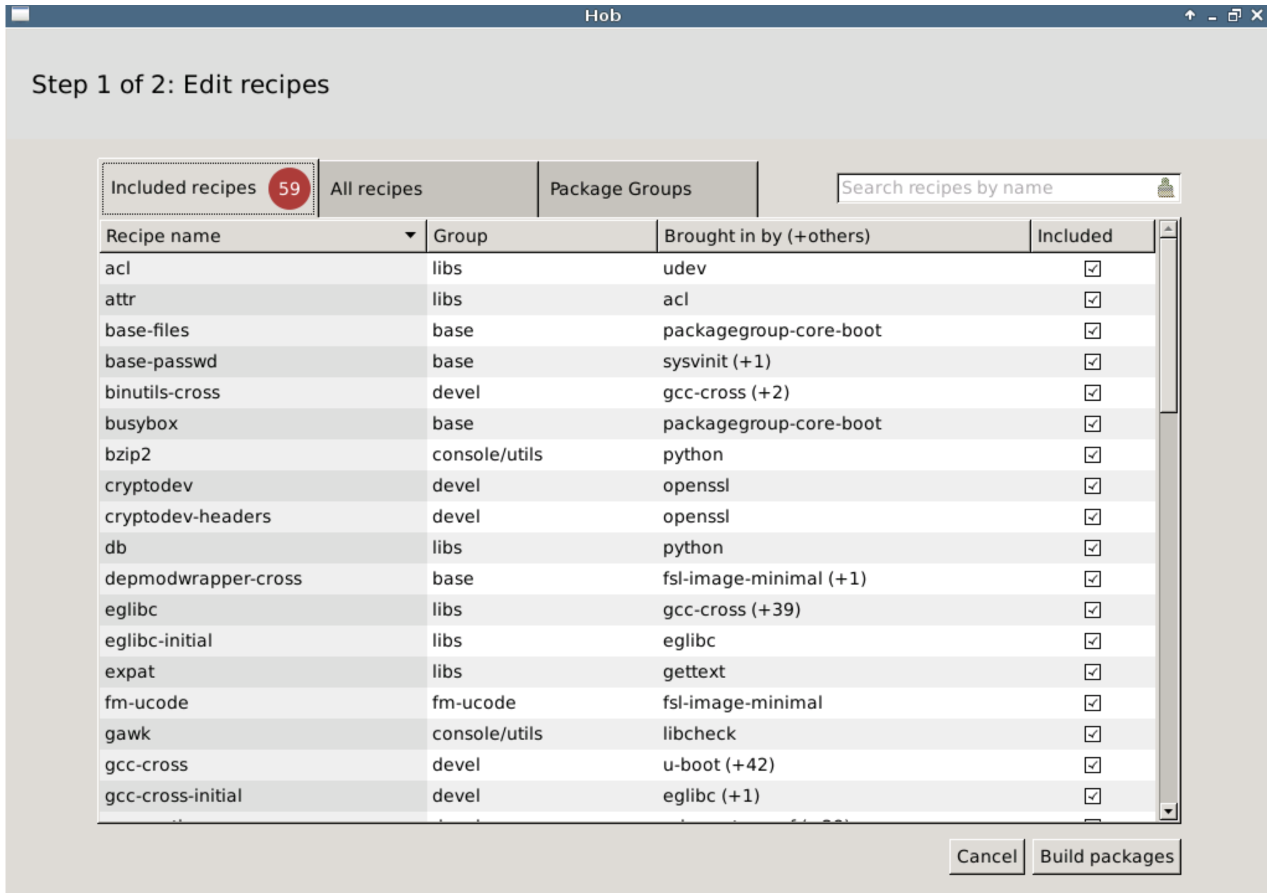

To build the image recipe as is, click the "Build image" button



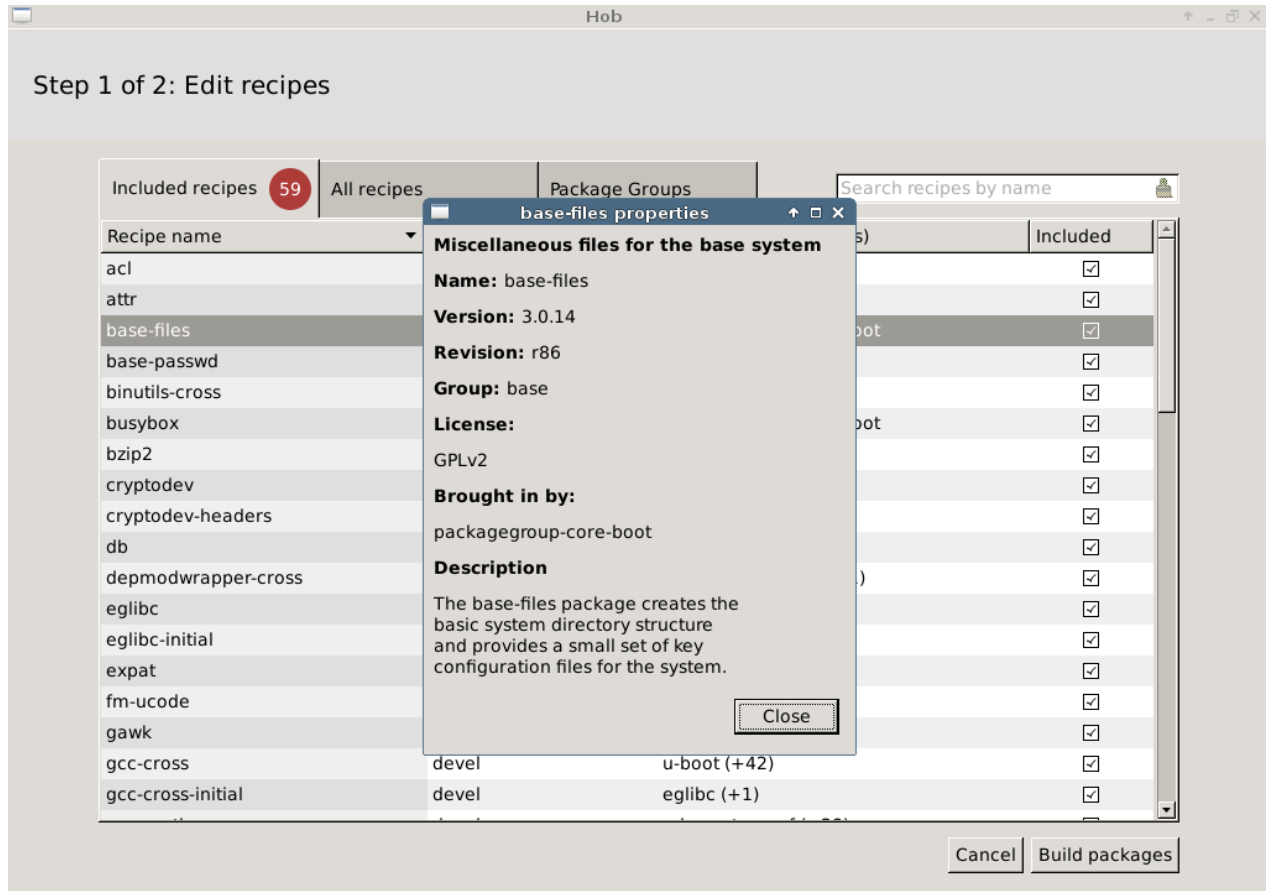

If you want to customize the recipes and packages included by default in your selected image recipe, click the "Edit image recipe" button.



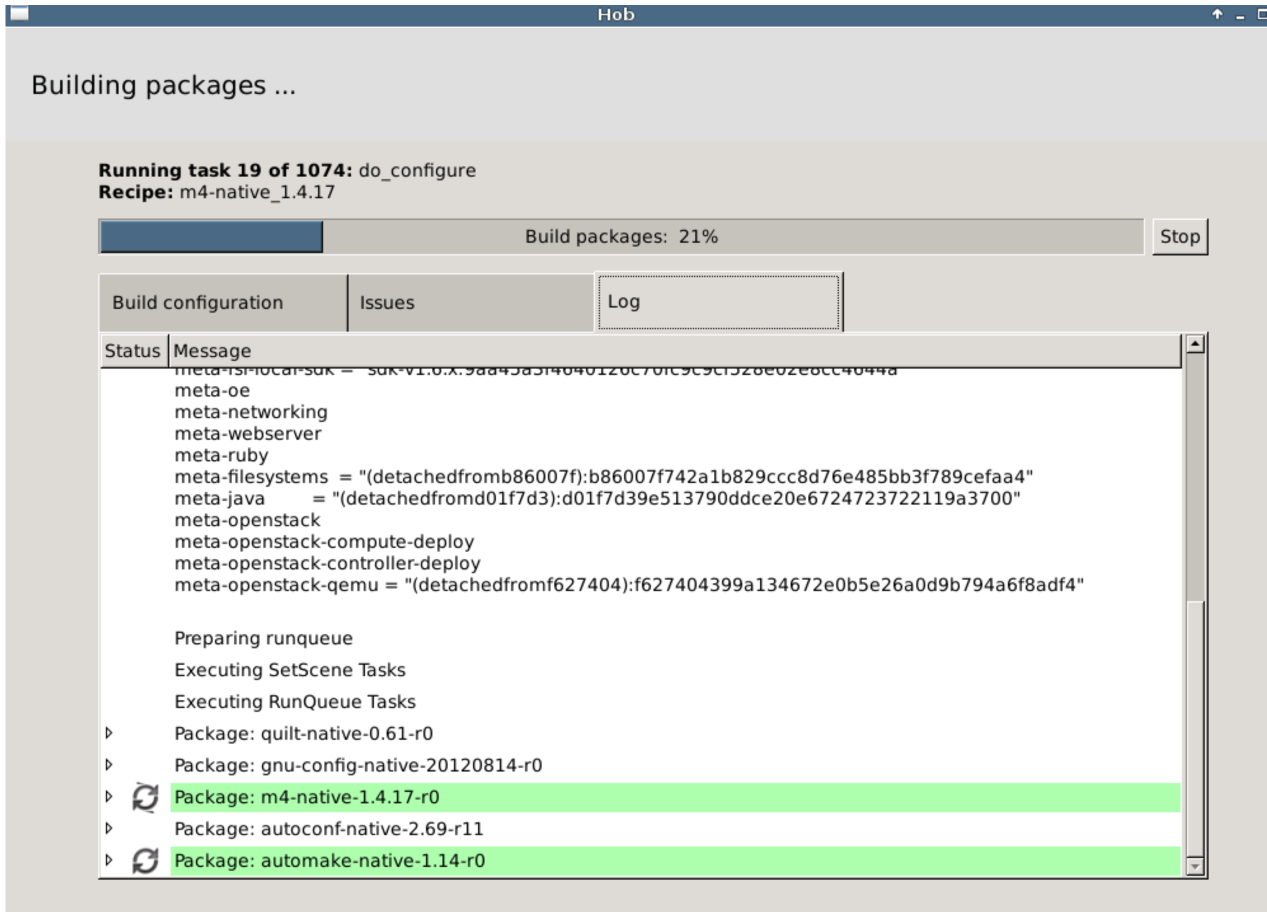

If you are not satisfied with the set of default recipes, you can use the "Edit recipes" screen to include or exclude recipes of your choice. You need to be aware that including and excluding recipes also includes or excludes dependent recipes. Use the check boxes in the "Included" column to include and exclude recipes.



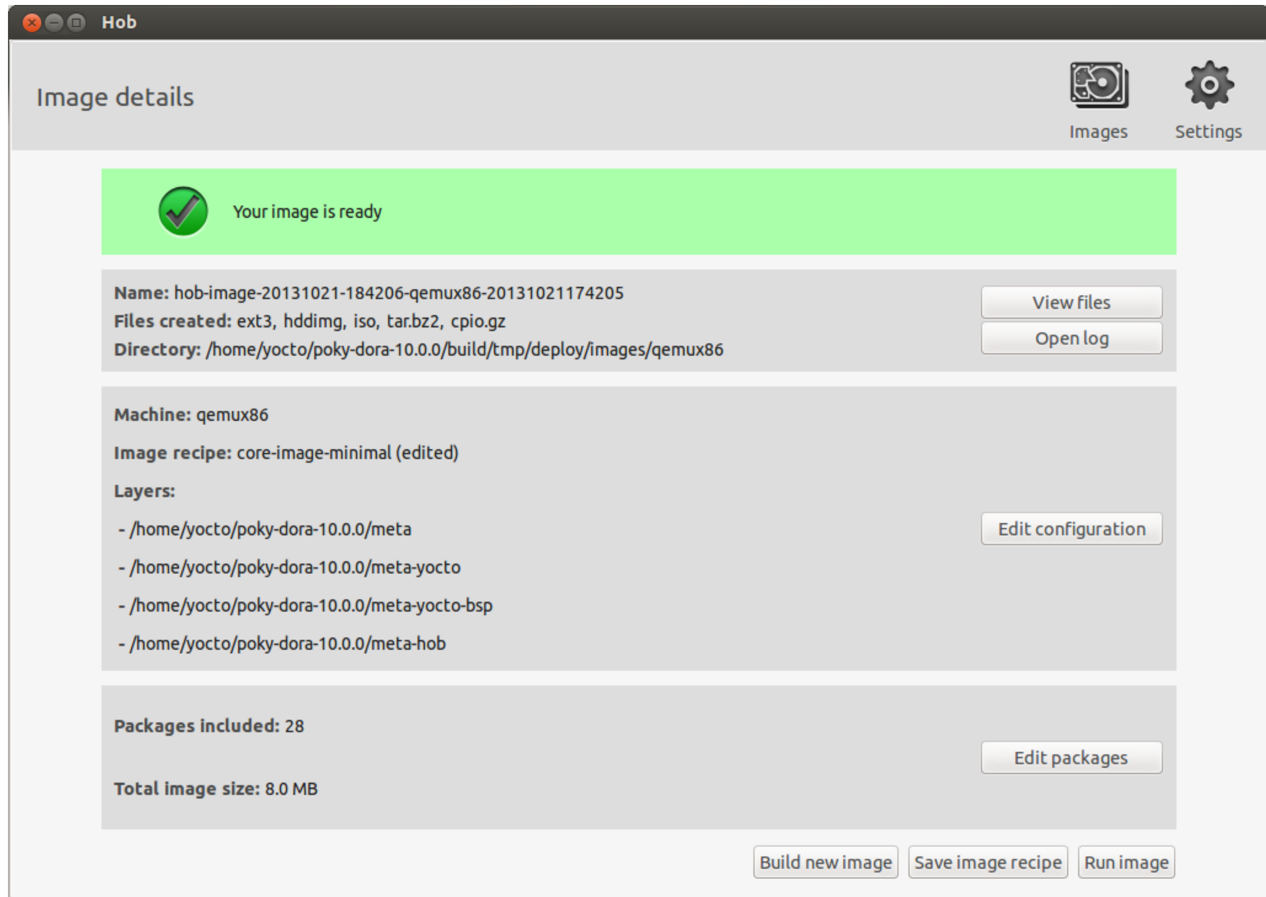

Click on a recipe to see some background information about it: description, homepage, license, and a full list of the recipes which bring in the selected recipe.



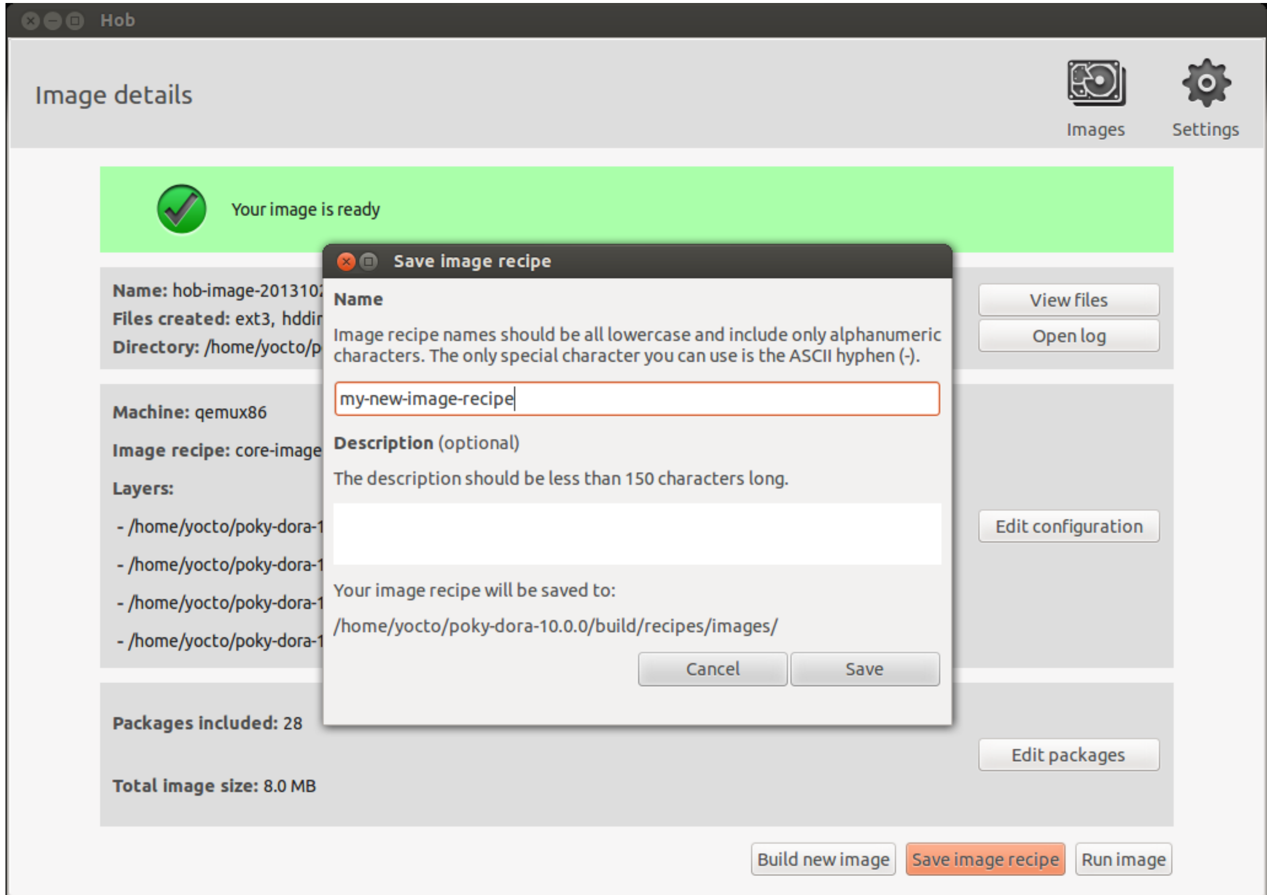

Once you are satisfied with the set of recipes, you can proceed by clicking the "Build packages" button. Or, you can cancel your changes and go back to the main screen by clicking the "Cancel" button. Here is what the screen looks like during package building:.



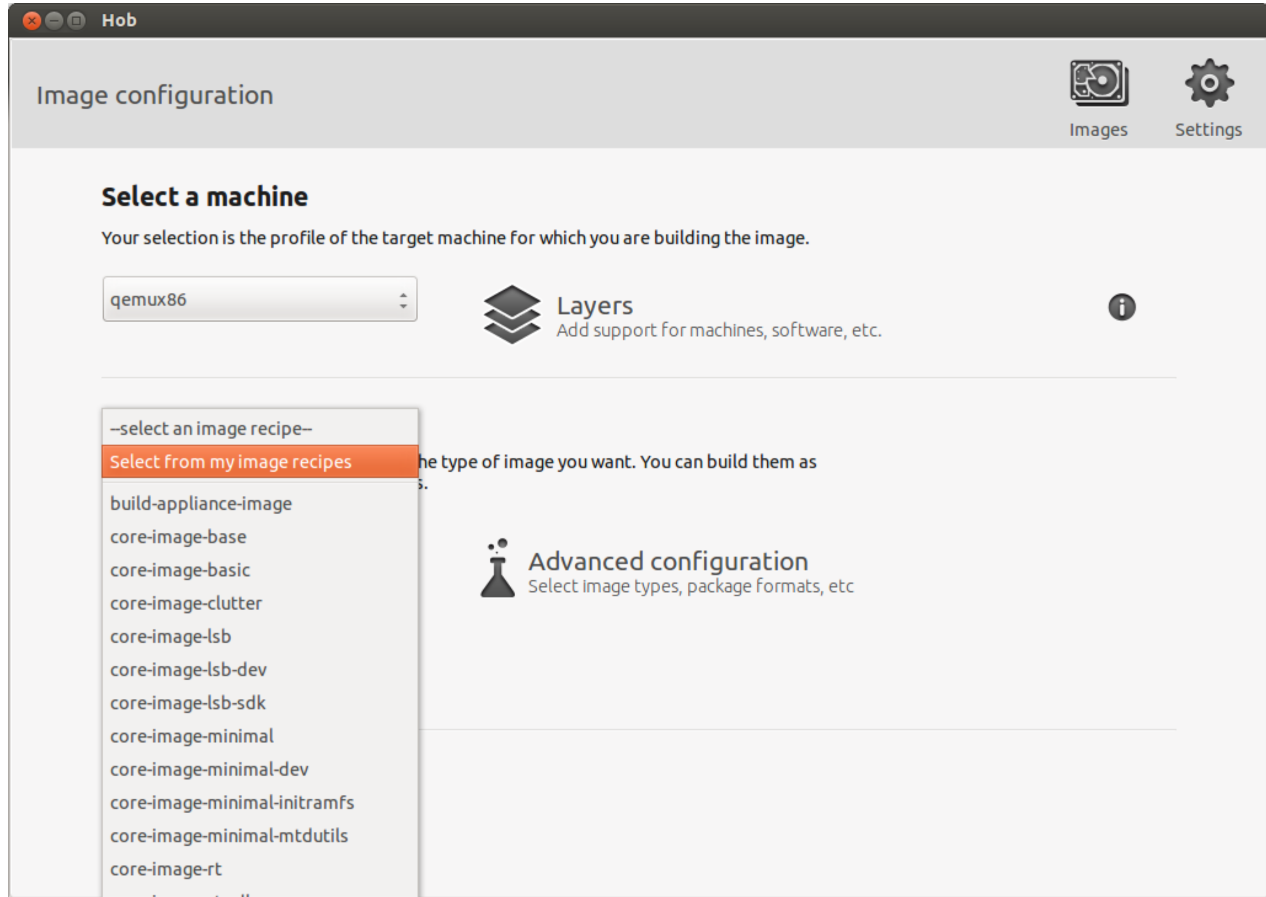

Once the build completes, you have several options. You can view the image files, check the build log, edit the configuration, edit the packages or build a new image.If you customized the content of you selected image recipe, Hob will also give you the option to save your changes as a new recipe.




Clicking the "Save image recipe" button in the lower-right hand corner of the screen displays a dialog where you enter the name and description for your image recipe. The name is mandatory, but the description is optional.




To build your image recipe, use the "Select from my image recipes" option that you'll find at the top of the image recipe selection menu.




## 2.2.10 FAQ

Frequently Asked Question about Yocto

Q: How to install source code, modify source code and rebuild u-boot/kernel?.

A: Use the following steps:

1. `$ cd <yocto_install_path>/build_<machine>_release`
2. If the source code has been installed, please skip this step.
   - `$ bitbake -c patch <package-name>`
   - `$ cd <S>` and do change

   **NOTE**

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

3. Modify configure (dts can also be modified):
   - Modify the `UBOOT_MACHINES` variable in `meta-fsl-ppc/conf/machine/<machine>.conf` or update the `KERNEL_DEFCONFIG` variable in `meta-fsl-ppc/conf/machine/<machine>.conf`
4. Rebuild images:
   - `$ cd <yocto_install_path>/build_<machine>_release`
   - `$ bitbake -c compile -f <package-name>`
   - `$ bitbake <package-name>`

**NOTE**

`u-boot.bin`, `uImage` and `dtb` files can be found in `build_<machine>_release/tmp/deploy/images/`.

Or set `<ARCH> and <CROSS_COMPILE>` and build the u-boot/kernel manually

Q: How to build u-boot/kernel with debugger (CodeWarrior support)?

A: For u-boot:

1. `$ cd <yocto_install_path>`
2. `$ bitbake -c cleansstate u-boot`
3. Modify the `u-boot_git.bb` file and add following content:
   - `$ cd meta-fsl-ppc/recipes-kernel/u-boot`
   - `$ add 'EXTRA_OEMAKE += "CONFIG_CW=1"' in u-boot_git.bb file`
4. Rebuild u-boot:
   - `$ bitbake u-boot`

For kernel:

1. `$ cd <yocto_install_path>`.
2. If kernel source code is not installed, install the kernel source code first.
   - `$ bitbake -c cleansstate virtual/kernel`
   - `$ bitbake -c patch virtual/kernel`
3. Configure kernel to enable CW support:
   - `$ bitbake -c menuconfig virtual/kernel`
   - Enable Kernel hacking -> Include CodeWarrior kernel debugging in kernel configuration UI.
4. Rebuild kernel:
   - `$ bitbake virtual/kernel`

Q: How to view the content of rootfs

A: The expanded rootfs is in `<IMAGE_ROOTFS>`

**NOTE**

Use `bitbake -e <rootfs-name> | grep ^IMAGE_ROOTFS=` to get the value of `<IMAGE_ROOTFS>`.

Q: How to display packages which are included in current rootfs image?

A: There are three methods :

- `$hob`
- `$bitbake -e <image-target> | grep IMAGE_INSTALL`
- `$bitbake -g <image-target>`

Q: How to add a pre-built ppc binary into the rootfs

A: Do the following:

1. `$ cd <yocto_install_path>`.
2. Add the files:

- `$ cd meta-fsl-networking/recipes-tools/merge-files`
- Put the files into files/merge, e.g. put `bash` into `files/merge/usr`, `bash` will be included in `usr/` of the new rootfs.

3. Build new rootfs image:
   - `$ bitbake <rootfs-target>`

Q: Example of using dtc to reverse to a dts from a dtb

A: Do the following

1. 
```
$ export PATH=<path-of-dtc>:$PATH
```
2. 
```
$ dtc -I dtb -O dts -o name-of-dts name-of-dtb
```

Q: How to build fsl-toolchain?

A: Do the following

1. 
```
$bitbake fsl-toolchain
```

> **NOTE**
> fsl-networking-eglibc-x86_64-<target>-toolchain-<VERSION>.sh can be found in `build_<machine>_release/tmp/deploy/sdk/`

> **NOTE**
> fsl-networking-eglibc-x86_64-<target>-toolchain-<VERSION>.sh runs and the toolchain can be installed in special path

Q: Fail to get source tarball of a packages with wget.

A: Use the `--no-check-certificates` optionwhen Yocto uses `wget` to fetch source tarball from internet, the option is only available when `wget` was configured at build time with SSH support. Ensure that `wget` on your host is built with SSH support.

Q: How to include toolchain in rootfs?

A: The toolchain runs on target board, which is the same exact version as the cross tools in this SDK, is only included in `fsl-image-full` rootfs, if you want to add toolchain into other rootfs images, do the following:

1. Edit `fsl-image-minimal.bb/fsl-image-flash.bb/fsl-image-core.bb` to add `packagegroup-core-buildessential` in the `IMAGE_INSTALL` variable.
2. `$ bitbake <rootfs-target>`.

Q: how to use standalone toolchain to build kernel?

A: If you want to build the linux kernel using toolchain, you can do the following:

1. 
```
cd <toolchain_install_path>
```
2. 
```
source environment-setup-<core>-fsl_networking-linux
```
3. 
```
export LDFLAGS=""
```
4. 
```
cd <linux_source_ path>
```

5.
```
make mrproper
```

6.
```
make <target_defconfig>
```

7.
```
make
```

### 2.2.11 BitBake User Manual

BitBake is a tool for executing tasks and managing metadata.

BitBake is, at its simplest, a tool for executing tasks and managing metadata. As such, its similarities to GNU make and other build tools are readily apparent. It was inspired by Portage, the package management system used by the Gentoo Linux distribution. BitBake is the basis of the OpenEmbedded project (www.openembedded.org), which is being used to build and maintain a number of embedded Linux distributions, including OpenZaurus and Familiar.

This document is not available in PDF form. However, you may obtain the latest document at http://docs.openembedded.org/bitbake

# Chapter 3
# Deployment Guides

## 3.1 Introduction

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

- Switch Settings
- U-Boot environment Variables
- Frame Manager Microcode (if applicable)
- Reset Configuration Word (RCW) and Ethernet Interfaces (if applicable)
- System Memory Map
- Flash Bank Usage

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

For reference design boards with more than one flash bank, the "Programming a New U-Boot, RCW, FMan Microcode" section describes how the user can individually or simultaneously update U-Boot, RCW, and FMan microcode by flashing them to the board's alternate bank prior to deployment.

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

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

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

## 3.2 Basic Host Set-up

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

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

1. Create the tftboot directory.

```
mkdir /tftpboot
```

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

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

3. Use Yocto to generate a tar.gz type file system, and uncompress it in <nfs_root_path>.
4. Edit /etc/exports and add the following line:

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

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

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

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

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

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

NOTE

(i) The Linux distribution running on your host will determine the specific instructions to use.

(ii) Steps 3 and 4 are only necessary when using NFS deployment.

## 3.3 Target Board Set-up

Once the host set-up is complete, follow these steps to set-up and power on the target board:

1. Power off the Target board system if the power is already on.
2. Connect the Target board to the network via an Ethernet port on the board.
3. Connect the Target board to the host machine via the serial port with an RS-232 cable and the joined Freescale adaptor cable, if needed.
4. Start the serial console tool on the host system.
5. Verify that all switches and jumpers are setup correctly as described in the board's Reference Manual/User Guide.
6. Power on the board.

Below is an example of a typical U-Boot log:

```
U-Boot 2016.01-rc1-00116-gb80a35f (Nov 21 2015 - 13:37:35 -0800)

CPU0:  P1022E, Version: 1.1, (0x80ee0011)
Core:  e500, Version: 5.1, (0x80211151)
Clock Configuration:
       CPU0:1066.656 MHz, CPU1:1066.656 MHz,
       CCB:533.328 MHz,
       DDR:333.330 MHz (666.660 MT/s data rate) (Asynchronous), LBC:33.333 MHz
L1:    D-cache 32 KiB enabled
       I-cache 32 KiB enabled
Board: P1022DS Sys ID: 0x19, Sys Ver: 0x03, FPGA Ver: 0x0a, vBank: 2
I2C:   ready
SPI:   ready
DRAM:  Detected UDIMM i-DIMM
2 GiB (DDR3, 64-bit, CL=6, ECC off)
Flash: 128 MiB
L2:    256 KiB enabled
NAND:  1024 MiB
MMC:   FSL_SDHC: 0
EEPROM: CRC mismatch (337047b4 != 00000000)
PCIe1: Root Complex of Slot 1, no link, regs @ 0xffe0a000
PCIe1: Bus 00 - 00
PCIe2: Root Complex of Slot 3, no link, regs @ 0xffe09000
PCIe2: Bus 01 - 01
PCIe3: Root Complex of Slot 2, x1 gen1, regs @ 0xffe0b000
  03:00.0     - 8086:10d3 - Network controller
PCIe3: Bus 02 - 03
In:    serial
Out:   serial
Err:   serial
Net:   e1000: 68:05:ca:0f:23:7d
       eTSEC1
Error: eTSEC1 address not set.
, eTSEC2
Error: eTSEC2 address not set.
, e1000#0 [PRIME]
Hit any key to stop autoboot:  0
```

NOTE

If the target board does not have a working U-Boot, see System Recovery on page 79.

# 3.4 Boards

## 3.4.1 Overview

The LS1043A reference design board(RDB) and QorIQ development system(QDS) are the high-performance computing,evaluation,and development platform that supports the QorIQ LS1043A LayerScape architecture processor.

This guide provides board-specific configuration and instructions for different methods of deploying U-Boot, Linux kernel and root file system to the target board.

## 3.4.2 Switch Settings

The RDB and QDS have user selectable switches for evaluating different boot options for the LS1043A device. Table below lists the default switch settings and the description of these settings.

1. Switch for RDB

Table 2: SWITCH Setting

| Boot Source | Switch |
|---|---|
| Nor(bank0) | SW4[1-8] +SW5[1] = 0b'00010010_1 SW5[4-6]= 0b'000 |
| Nor(bank4) | SW4[1-8] +SW5[1] = 0b'00010010_1 SW5[4-6]= 0b'001 |
| Nand | SW4[1-8] +SW5[1] = 0b'10000011_0 |
| Sd | SW4[1-8] +SW5[1] = 0b'00100000_0 |

2. Switch for QDS

Table 3: SWITCH Setting

| Boot Source | Switch |
|---|---|
| Nor (bank0) | SW1[1-8] = 0b'00010010 SW2[1] = 0b'1 SW6[1-4] = 0b'0000 |
| Nor (bank4) | SW1[1-8] = 0b'00100010 SW2[1] = 0b'1 SW6[1-4] = 0b'0100 |
| Nand | SW1[1-8] = 0b'10000011 SW2[1] = 0b'0 SW6[1-4] = 0b'1001 |
| Sd | SW1[1-8] = 0b'00100000 SW2[1] = 0b'0 SW6[1-4] = 0b'1111 |
| Qspi | SW1[1-8] = 0b'00100010 SW2[1] = 0b'1 SW6[1-4] = 0b'1111 |

## 3.4.3 U-Boot Environment Variables

### 3.4.3.1 U-Boot Environment Variable “hwconfig”

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

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

```
=>setenv hwconfig
```

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

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

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

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

Once you have appended the text, you should see the following:

```
=>print hwconfig
hwconfig = fsl_ddr:bank_intlv=auto
```

### 3.4.3.2 Configuring U-Boot Network Parameters

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

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

NOTE

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

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

```
=>setenv ethaddr 00:04:9f:ef:00:00
=>setenv eth1addr 00:04:9f:ef:01:01
=>setenv eth2addr 00:04:9f:ef:02:02
```

```
=>setenv eth3addr 00:04:9f:ef:03:03
=>saveenv
```

NOTE

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

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

### 3.4.4 RCW (Reset Configuration Word)

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

```
ab_cdef_g
```

Table 4: RCW irectories Naming Convention Legend

| Slot | Convention |
| --- | --- |
| a | 'R' indicates RGMII1@DTSEC3 is supported<br>'N' if not available/not used |
| b | 'R' indicates RGMII2@DTSEC4 is supported<br>'N' if not available/not used |
| c | What is available in the lane A(RDB) or slot1(QDS) |
| d | What is available in the lane B(RDB) or slot2(QDS) |
| e | What is available in the lane C(RDB) or slot3(QDS) |
| f | What is available in the lane D(RDB) or slot4(QDS) |
| g | Hex value of serdes1 protocol value |

Table 5: For the Lanes or Slot (c..f)

| Flag | Convention |
| --- | --- |
| 'N' | NULL, not available/not used |
| 'P' | PCIe |
| 'X' | XAUI |
| 'S' | SGMII |
| 'Q' | QSGMII |

*Table continues on the next page...*

Table 5: For the Lanes or Slot (c..f) (continued)

| | |
|---|---|
| '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 6: RCW Files Naming Convention Legend

| Code | | Convention |
|---|---|---|
| frequency | | Core frequency(MHZ) |
| specialsetting | bootmode | SD/NAND/NOR and so on |
| | special support | nand: Nand boot<br>sben: Secure boot<br>lpuart: lpuart1<br>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.
```

1. The following RCW binary for use 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
```

2. The following RCW binary for use on the ls1043aqds

```
RR_FQPP_1455/rcw_1600_lpuart.bin : lpuart
```

```
RR_FQPP_1455/rcw_1600_qspiboot.bin : Qspi boot
```

```
RR_SSPH_3358/rcw_1600.bin : Test SATA and SGMII
```

## 3.4.5 System Memory Map

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

| Start Physical Address | End Physical Address | Memory Type | Size |
|---|---|---|---|
| `0x00_0000_0000` | `0x00_00FF_FFFF` | Secure Boot ROM | 1MB |
| `0x00_0100_0000` | `0x00_0FFF_FFFF` | CCSR | 240MB |
| `0x00_1000_0000` | `0x00_1000_FFFF` | OCRAM0 | 64KB |
| `0x00_1001_0000` | `0x00_1001_FFFF` | OCRAM1 | 64 KB |
| `0x00_2000_0000` | `0x00_23FF_FFFF` | DCSR | 64MB |
| `0x00_4000_0000` | `0x00_5FFF_FFFF` | QSPI | 512MB |
| `0x00_6000_0000` | `0x00_7FFF_FFFF` | IFC region | 512MB |

*Table continues on the next page...*

*Table continued from the previous page...*

| Start Physical Address | End Physical Address | Memory Type | Size |
|---|---|---|---|
| `0x00_8000_0000` | `0x00_FFFF_FFFF` | DRAM | 2GB |
| `0x05_0000_0000` | `0x05_07FF_FFFF` | QMAN S/W Portal | 128M |
| `0x05_0800_0000` | `0x05_0FFF_FFFF` | BMAN S/W Portal | 128M |
| `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 Express1 | 32G |

### 3.4.6 Flash Bank Usage

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.

To protect the default U-Boot in bank 0, it is a convention employed by Freescale to deploy work images into the alternate bank, and then switch to the alternate bank for testing. 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 2015.01-gc297f5b (Dec 17 2015 - 16:09:30)

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 00001100 00000096 00000001
Board: LS1043AQDS
vBank: 4
Sys ID:0x36, Sys Ver: 0x11
FPGA:  v5 (LS1043QDS_2015_0427_1339), build 5
I2C:   ready
DRAM:  Initializing DDR....using SPD
SPD DQ mapping error fixed
Detected UDIMM 18ASF1G72AZ-2G1A1
4 GiB (DDR4, 32-bit, CL=11, ECC on)
    DDR Chip-Select Interleaving Mode: CS0+CS1
Retimer version id = 0x61
```

```
Using SERDES1 Protocol: 5205 (0x1455)
fman_port_enet_if:71: port(FM1_DTSEC3) is OK
fman_port_enet_if:77: port(FM1_DTSEC4) is OK
Flash: 128 MiB
NAND:  512 MiB
MMC:   FSL_SDHC: 0
PCIe1: disabled
PCIe2: Root Complex no link, regs @ 0x3500000
PCIe3: Root Complex no link, regs @ 0x3600000
In:    serial
Out:   serial
Err:   serial
SCSI:  Net:   Fman1: Uploading microcode version 106.4.15
Phy 9 not found
PHY reset timed out
FM1@DTSEC3, FM1@DTSEC4, FM1@TGEC1
Hit any key to stop autoboot:  0
```

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

- Switch to bank 0 for RDB:

```
=>cpld reset
```

- Switch to alternate bank for RDB:

```
=>cpld reset altbank
```

- Switch to bank 0 for QDS:

```
=>qixis_reset
```

- Switch to alternate bank for QDS:

```
=>qixis_reset altbank
```

The table below shows a memory map:

**Table 7: NOR Flash Memory Map for RDB and QDS**

| Start | End Offset | Description | Size |
|---|---|---|---|
| 0x60000000 | 0x600FFFFF | bank0 rcw + pbi | 1M |
| 0x60100000 | 0x601FFFFF | bank0 U-Boot image | 1 M |
| 0x60200000 | 0x602FFFFF | bank0 u-Boot Env | 1 M |
| 0x60300000 | 0x603FFFFF | bank0 Fman ucode | 1 M |
| 0x60400000 | 0x604FFFFF | bank0 UEFI | 1 M |
| 0x60500000 | 0x605FFFFF | bank0 PPA | 1 M |
| 0x60600000 | 0x606FFFFF | bank0 QE firmware | 1 M |
| 0x60700000 | 0x60EFFFFF | bank0 reserved | 8 M |

*Table continues on the next page...*

Table 7: NOR Flash Memory Map for RDB and QDS (continued)

| | | | |
|---|---|---|---|
| 0x60F00000 | 0x60FFFFFF | bank0 PHY firmware | 1 M |
| 0x61000000 | 0x610FFFFF | bank0 CORTINA PHY firmware | 1 M |
| 0x61100000 | 0x638FFFFF | bank0 FIT Image | 40M |
| 0x63900000 | 0x63FFFFFF | bank0 Unused | 7 M |
| 0x64000000 | 0x640FFFFF | bank4 rcw + pbi | 1M |
| 0x64100000 | 0x641FFFFF | bank4 U-Boot image | 1 M |
| 0x64200000 | 0x642FFFFF | bank4 u-Boot Env | 1 M |
| 0x64300000 | 0x643FFFFF | bank4 Fman ucode | 1 M |
| 0x64400000 | 0x644FFFFF | bank4 UEFI | 1 M |
| 0x64500000 | 0x645FFFFF | bank4 PPA | 1 M |
| 0x64600000 | 0x646FFFFF | bank4 QE firmware | 1 M |
| 0x64700000 | 0x64EFFFFF | bank4 reserved | 8 M |
| 0x64F00000 | 0x64FFFFFF | bank4 PHY firmware | 1 M |
| 0x65000000 | 0x650FFFFF | bank4 CORTINA PHY firmware | 1 M |
| 0x65100000 | 0x678FFFFF | bank4 FIT Image | 40M |
| 0x67900000 | 0x67FFFFFF | bank4 Unused | 7 M |

## 3.4.7 Programming a New U-boot and RCW

The following three sections will discuss how to individually update U-Boot, RCW. For specific addresses, please refer to the NOR Flash Memory Map as a reference. If the user intends to flash both two at once, there is no need to switch into the alternate bank after each configuration, i.e. type the command "cpld reset altbank" after U-Boot and RCW have both been programmed.

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

### Programming a New U-Boot

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

```
=>tftp 82000000 <u-boot_file_name>.bin
=>protect off 64100000 +$filesize
=>erase 64100000 +$filesize
=>cp.b 82000000 64100000 $filesize
=>protect on 64100000 +$filesize
=>cpld reset altbank
```

The commands above will only program a new U-Boot. Programming a new RCW will be discussed in the next sections.

#### Programming a New RCW

To program a new RCW, first switch to bank 0 by performing a hard reset or by typing *reset*. Next, load the new RCW to RAM by downloading it via TFTP and then copying it to flash at address 0x64000000. 0x64000000 is the location of RCW in the alternate bank. Execute the following commands at the U-Boot prompt to program the RCW to flash and reset to alternate bank.

```
=>tftp 82000000 <rcw_file_name>.bin
=>protect off 64000000 +$filesize
=>erase 64000000 +$filesize
=>cp.b 82000000 64000000 $filesize
=>protect on 64000000 +$filesize
=>cpld reset altbank
```

#### Programming U-boot to SD card

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

```
=>tftp 82000000 u-boot-with-spl-pbl.bin
=>mmc erase 8 0x800
=>mmc write 0x82000000 8 0x800
```

Program the image to SD card in Linux.

```
dd if=u-boot-with-spl-pbl-sd.bin of=/dev/sdb bs=1024 seek=8
```

## 3.4.8 Deployment

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

### 3.4.8.1 FIT Image Deployment from TFTP

1. **Setting U-Boot Environment**

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

   Refer to Configuring U-Boot Network Parameters on page 59 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for FIT image deployment from TFTP:

   ```
   =>setenv bootargs ‘root=/dev/ram rw earlycon=uart8250,0x21c0500,115200
   console=ttyS0,115200’
   =>saveenv
   ```

2. **Booting Up the System**

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

   ```
   =>tftp a0000000 < FIT_image_name>
   =>bootm a0000000
   ```

Now the board will boot into Linux .

### 3.4.8.2 FIT Image Deployment from Flash

1. **Setting U-Boot Environment**

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

   ```
   =>setenv  bootargs root=/dev/ram rw console=ttyS0,115200
   earlycon=uart8250,0x21c0500,115200
   =>setenv bootcmd   bootm <FIT_image_addr>
   =>saveenv
   ```

   Now U-Boot is ready for flash deployment.

2. **Programming FIT image to NOR Flash**

   The image should be downloaded to the RAM using TFTP then copied to the flash address <FIT_image_addr>. At the U-Boot prompt, use the following commands to program the image to flash:

   ```
   =>tftp 82000000 <FIT Image name>
   =>protect off <FIT_image_addr> +$filesize
   =>erase <FIT_image_addr> +$filesize
   =>cp.b 82000000 <FIT_image_addr> $filesize
   =>protect on <FIT_image_addr> +$filesize
   ```

3. **Booting Up the System**

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

   ```
   =>boot
   ```

   or

   ```
   => bootm <FIT_image_addr>
   ```

4. **FIT Image address on Flash Memory Map**

   ```
   > Bank0: FIT_image_addr = 0x0_6110_0000
   ```

   ```
   > Bank4: FIT_image_addr = 0x0_6510_0000
   ```

### 3.4.8.3 NFS Deployment

1. **Generating File System**

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

2. **Setting Host NFS Server Environment**

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

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

b. Restart the NFS service:

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

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

> **NOTE**
> nfs_root_path: the NFS root directory path on NFS server.

3. **Setting U-Boot Environment**

   The NFS file system generated by Yocto allows you to perform NFS deployment. Before performing NFS deployment, the U-Boot environment variables need to be configured. Refer to Configuring U-Boot Network Parameters on page 59 to set the U-Boot environment variables. In addition, execute the following commands at the U-Boot prompt to prepare for NFS deployment:

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

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

   Now U-Boot is ready for NFS deployment.

4. **Booting up the System**

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

```
=>tftp a0000000  kernel.itb;
=>bootm a0000000
```

   Now the board will boot up with NFS filesystem.

## 3.4.8.4 SD Deployment

### Partition SD Card

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

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

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

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

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

### FIT Kernel Image and Root File System Deployment from SD Card

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

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

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

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

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

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

5. Umount the temp director

```
#umount temp
```

### U-Boot Image Deployment from SD Card

- Form Linux Host PC
  1. Insert SD card into Host.
  2. Use the "dd" command

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

- Form ls1043ardb Board
  1. Insert SD card into target board and power on.
  2. Programming U-boot image to SD Card

```
=> tftpboot 82000000 u-boot-with-spl-pbl.bin
=> mmc write 82000000 8 800
=> cpld reset sd
```

### Setting U-Boot Environment

- Execute the following commands at the U-Boot prompt

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

- Using the Ramdisk as the Root File Systemp

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

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

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

- Saving the environment

```
=>saveenv
```

NOTE

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

## 3.4.8.5 QSPI Deployment(only for QDS board)

### Use an X-nor card

1. QDS X-nor card hardware configuration:

```
SW1[1:4]=1000
```

```
SW4[1:4]=1111
```

```
SW3[1:4]=0001
```

2. Insert the X-nor card into IFCCard slot.

### Prepare SD boot to program QSPI flash

1. Build U-Boot image for SD boot(enables QSPI):

```
$make distclean ARCH=aarch64 CROSS_COMPILE=${toolchain_path}/gcc-linaro-aarch64-
linux-gnu-4.9-2014.07_linux/bin/aarch64-linux-gnu-
```

```
$make ARCH=aarch64 ls1043aqds_sdcard_qspi_defconfig
```

```
$make CROSS_COMPILE=${toolchain_path}/gcc-linaro-aarch64-linux-
gnu-4.9-2014.07_linux/bin/aarch64-linux-gnu- -j4
```

2. Write the U-Boot image to SD card:

```
=>tftp 0x81000000 u-boot-with-spl-pbl.bin
```

```
=>mmc write 0x81000000 8 0x800
```

3. Switch to SD boot(enables QSPI):

```
=>qixis_reset sd_qspi
```

Or set SW1[1:8]=0010_0000,SW2[1]=0,SW6[1:4]=1111 and power on the board from SD boot.

**NOTE**

As QSPI and IFC are pin-multiplexed in LS1043A, use U-Boot booting from SD card that supports QSPI to write your images to QSPI flash.

### Build U-Boot image for QSPI boot

1. Compile QSPI boot image(enable QSPI):

```
$make distclean ARCH=aarch64 CROSS_COMPILE=${toolchain_path}/gcc-linaro-aarch64-
linux-gnu-4.9-2014.07_linux/bin/aarch64-linux-gnu-
```

```
$make ARCH=aarch64 ls1043aqds_qspi_defconfig
```

```
$make CROSS_COMPILE=${toolchain_path}/gcc-linaro-aarch64-linux-
gnu-4.9-2014.07_linux/bin/aarch64-linux-gnu- -j4
```

2. Swap the bytes for QSPI boot:

```
$tclsh byte_swap.tcl rcw_1500_qspiboot.bin rcw_1500_qspiboot_swap.bin 8
```

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

The byte_swap.tcl script is avaliable under ls1043-rcw/ls1043aqds/ directory.

3. Write RCW and U-Boot images to QSPI flash under SD boot(enables QSPI) mode:

```
=>sf probe 0:0
```

SF: Detected S25FL128S_64K with page size 256 Bytes, erase size 64 KiB, total 16 MiB

```
=>tftp 81000000 rcw_1500_qspiboot_swap.bin;sf erase 0 +$filesize;sf write 81000000
0 $filesize
```

```
=>tftp 82000000 u-boot_swap.bin;sf erase 10000 +$filesize;sf write 82000000
10000 $filesize
```

4. Switch to QSPI boot:

```
=>qixis_reset qspi
```

Or set SW1[1:8]=0010_0010,SW2[1]=1,SW6[1:4]=1111 and power on the board from QSPI boot.

### Write FMan ucode to QSPI flash to use FMan

Do it under SD boot(enbles QSPI) or QSPI boot:

```
=>sf probe 0:0
```

SF: Detected S25FL128S_64K with page size 256 Bytes, erase size 64 KiB, total 16 MiB

```
=>tftp 83000000 fsl_fman_ucode_t2080_r1.1_106_4_15.bin
```

```
=>sf probe 0:0;sf erase d0000 +$filesize;sf write 83000000 d0000 $filesize
```

## 3.4.9 Check 'Link Up' for Serial Interfaces

If you are experiencing problems with your Ethernet interfaces, this section provides some basic checks that can be performed in U-Boot to help diagnose the cause of the networking errors.

### Check Communication to External PHY

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

Example:

```
=> mdio list
FSL_MDIO0:
FSL_MDIO0:
1 - RealTek RTL8211F <--> FM1@DTSEC3
2 - RealTek RTL8211F <--> FM1@DTSEC4
4 - Vitesse VSC8514 <--> FM1@DTSEC1
5 - Vitesse VSC8514  <--> FM1@DTSEC2
6 - Vitesse VSC8514  <-->  FM1@DTSEC5
7 - Vitesse VSC8514 <--> FM1@DTSEC6
FM_TGEC_MDIO:
1 - Aquantia AQR105  <-->  FM1@TGEC1
```

The results from the above *mdio list* command show that U-Boot was able to see PHYs on each of the 6 DTSEC interfaces and on the 10GEC interface. If you see “Generic” reported, it is an indication that something is there but the ls1043ardb can’t communicate with the device/port.

### Check Link Status for External PHY

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

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

Example:

```
=> mdio read FM1@DTSEC1 1
Reading from bus FSL_MDIO0
PHY at address 2:
1 - 0x796d
```

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 0x7969.

## 3.4.10 Basic Networking Ping Test

In Linux, in order to check that your network driver is set up, follow the commands below:

```
#ip addr show
#ip addr add <ip address of board>/24 brd + dev <port in Linux>
#ip link set <port in Linux> up
#ping <serverip>
```

### 3.4.11 Hardware Setting for Special Purposes

The RDB and QDS has user selectable switches for evaluating different pin mux and boot options for the LS1043A device. Information below lists the hardware setting for those special purposes.

For RDB:

General UART port :

1. UART1(J4 bootm)
2. UART2(J4 top).

NOR boot setting

1. Set SW4[1:8]+SW5[1]=0b'0001_0010_1;
2. Bank0: SW5[4:6]=0b'000;
3. Bank1: SW3[4:6]=0b'001

SD boot setting

1. image:u-boot-with-spl-pbl.bin
2. Program the image to SD card in u-boot.

```
=>tftp 82000000 u-boot-with-spl-pbl.bin
=>=>mmc erase 8 0x800; mmc write 82000000 8 0x800
```

3. Program the image to SD card in Linux.

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

4. Insert SD card in adaptor;
5. Set SW4[1:8]+SW5[1]=0b'0010_0000_0

NAND boot setting

1. image:u-boot-with-spl-pbl.bin
2. Program the image to SD card in u-boot.

```
=>tftp 82000000 u-boot-with-spl-pbl.bin
=>=>nand erase.chip; nand write 82000000 0 100000
```

3. Set SW4[1:8]+SW5[1]=0b'1000_0011_0

**NOTE**

For more information about specific settings for the console (i.e. baud rate, ports used etc.), please refer to the ls1043ardb Quick Start Guide.

For QDS

General UART port:

1. UART1(J46 bootm)
2. UART2(J46 top)

Lpuart

1. Switch - Nor(bank0) boot

   ```
   SW1[1-8] = 0b'00010010
   ```

   ```
   SW2[1] = 0b'1
   ```

   ```
   SW6[1-4] = 0b'0000
   ```

2. RCW - support for lpuart

   a. Compile RCW

   ```
   #cd ls1043-rcw/
   ```

   ```
   #make
   ```

   **NOTE**
   ls1043aqds/RR_FQPP_1455/rcw_1500_lpuart.bin is the RCW image we need.

   b. Program the RCW image to altbank

   ```
   =>tftp 82000000 rcw_1500i_lpuart.bin;
   ```

   ```
   =>protect off 64000000 +$filesize;
   ```

   ```
   =>erase 64000000 +$filesize
   ```

   ```
   =>cp.b 0x82000000 64000000 $filesize
   ```

   ```
   =>protect on 64000000 +$filesize
   ```

3. U-boot - with lpuat console

   a. Compile Nor Boot Image with lpuart support:

   ```
   #export ARCH=arm64
   ```

   ```
   #export CROSS_COMPILE=aarch64-linux-gnu-
   ```

   ```
   #make ls1043aqds_lpuart_defconfig
   ```

   ```
   #make all -j4
   ```

   **NOTE**
   u-boot.bin is the image we need.

b. Program the Nor iamge to altbank

```
=>tftp 82000000 u-boot.bin
```

```
=>protect off 64100000 +$filesize
```

```
=>erase 64100000 +$filesize
```

```
=>cp.b 0x82000000 64100000 $filesize
```

```
=>protect on 64100000 +$filesize
```

4. Kernel
   a. Compile kernel image and dtb file

```
#export ARCH=arm64
```

```
#export CROSS_COMPILE=aarch64-linux-gnu-
```

```
#make ls1043a_defconfig
```

```
#make all  -j4
```

```
mkimage -f  kernel-ls1043a-qds.its kernel.itb
```

5. Boot using lpuart console
   a. Connect the serial port of PC to the JP1 with lpuart serial cable.
   b. Boot from current bank(bank0).
   c. Program the RCW and uboot image to altbank.
   d. Swith to bank4 using "qixis_reset altbank" command on uboot console.
   e. Set uboot environment variable.

```
=>setenv bootargs "console=ttyLP0,115200 root=/dev/ram0
earlycon=uart8250,0x21c0500,115200"
```

   f. Bootup kernel

```
=>tftp a0000000 kernel.itb
```

```
=>bootm a0000000
```

SATA

1. Switch - Nor(bank0) boot

   ```
   SW1[1-8] = 0b'00010010
   ```

   ```
   SW2[1] = 0b'1
   ```

   ```
   SW6[1-4] = 0b'0000
   ```

2. RCW - support for SATA
   a. Compile RCW

      ```
      #cd ls1043-rcw/
      ```

      ```
      #make
      ```

      **NOTE**
      ls1043aqds/RR_SSPH_3358/rcw_1500.bin is the RCW image we need.

   b. Program the RCW iamge to altbank

      ```
      tftp 82000000 rcw_1500.bin
      ```

      ```
      protect off 64000000 +$filesize
      ```

      ```
      erase 64000000 +$filesize
      ```

      ```
      cp.b 0x82000000 64000000 $filesize
      ```

      ```
      protect on 64000000 +$filesize
      ```

3. U-boot
   a. Compile Nor Boot Image.

      ```
      #export ARCH=arm64
      ```

      ```
      #export CROSS_COMPILE=aarch64-linux-gnu-
      ```

      ```
      #make ls1043aqds_nor_defconfig
      ```

      ```
      #make all -j4
      ```

      **NOTE**
      u-boot.bin is the image we need.

b. Program the Nor iamge to altbank

```
=>tftp 82000000 u-boot.bin
```

```
=>protect off 64100000 +$filesize
```

```
=>erase 64100000 +$filesize
```

```
=>cp.b 0x82000000 64100000 $filesize
```

```
=>protect on 64100000 +$filesize
```

4. Kernel

a. Compile kernel image and dtb file

```
#export ARCH=arm64
```

```
#export CROSS_COMPILE=aarch64-linux-gnu-
```

```
#make ls1043a_defconfig
```

```
#make all  -j4
```

```
mkimage -f  kernel-ls1043a-qds.its kernel.itb
```

5. Boot & Test

a. Boot from current bank(bank0).

b. Program the RCW and U-Boot image to bank4.

c. Enable SATA support

```
=>mw.b 7fb00052 0
```

d. Switch bank4

```
=>qixis_reset altbank
```

e. Test SATA on uboot

```
=>scsi info
```

f. Bootup kernel

```
=>tftp a0000000 kernel.itb && bootm a0000000
```

g. Test SATA on kernel

```
#hdparm -i /dev/sda
```

# Chapter 4
# System Recovery

## 4.1 Environment Setup

### 4.1.1 Environment Setup (Common)

The section describes the related setup for system recovery

1. Required Materials
   - Target board
   - The related recovery image files
2. Host PC setup

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

   a. Power off the target board system if the power is already on.

   b. If U-Boot runs on this board, and U-Boot commands will be used to reflash the U-Boot images, connect the target board to the network via the eTSEC port on the board.

   c. Connect the target board to the host machine via the serial port with an RS-232 cable and the joined Freescale adapter cable, if needed.

   d. Set up the serial terminal in the host machine with 115,200bps, 8-N-1, no flow control.

   e. Verify all the switches and jumpers are set up correctly to default values described in <Hardware configurations> as described in the Switch Settings section of the board's Software Deployment Guide

   f. Connect the JTAG cable for your CodeWarrior TAP or Gigabit TAP to the board if you will be using the CodeWarrior Flash Programmer to recover the board image.

   g. Power on the board.

## 4.2 Image Recovery

### 4.2.1 Recover system with already working U-Boot

Target Board Setup

1. Power off the target board system if the power is already on.
2. Connect the target board to the network via the eTSEC port on the board.
3. Connect the target board to the host machine via the serial port with an RS-232 cable and the joined Freescale adaptor cable, if needed.
4. Set up the serial terminal in the host machine with 115,200bps, 8-N-1, no flow control.

5. Verify all the switches and jumpers are setup correctly to default value described in the board's "Switch Settings" section in the board's Software Deployment Guide.
6. Power on the board.

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

## 4.2.2 Recover system using CodeWarrior Flash Programmer

### Environment Setup

### Required Materials

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

### Host PC setup

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

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

### Target board setup

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

### System Recovery

1. Start the CodeWarrior PA10 or ARMv7 IDE.
2. For LS102x targets, see Chapter 3 of *Getting Started for ARMv7 Processors.pdf* in the CodeWarrior ARMv7 installation for steps on creating a BareBoard Core0 project for the LS102x processor on this target board. For other QorIQ targets, see Quick Start for PA 10 Processors.pdf in the CodeWarrior PA10 installation for steps on creating a BareBoard AMP Core0 project for the QorIQ processor on this target board. In the "Debug Target Settings Page", of the procedure for creating a new project, uncheck the 'Download' option, and enable the 'Download SRAM' option, if available.
3. Select your CodeWarrior TAP or Gigabit TAP as your debug connection type. For CodeWarrior TAP, select “USB” or “Ethernet” as the connection medium.
4. Build the project.

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

**Table 8: T4240QDS NOR flash**

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

**Table 9: BSC9132QDS NAND flash**

| Binaries | Starting Address |
| --- | --- |
| U-Boot | 0x00000000 |
| RCW | not used |
| ucode | not used |
| dtb | 0x00300000 |
| RamDisk (jffs2 rfs) | 0x00C00000 |
| Linux Kernel | 0x00400000 |

8. Configure Flash Programmer.

a. Double-click on the file name that was imported with the flash profile, to bring up the Flash Programmer Task view.
b. Click on 'Add Action'>'Program/Verify'.
c. Set 'File Type' to "Binary".
d. Click on 'File System' and navigate to the folder containing the u-boot binary image.
e. Enable "Erase sectors before program".
f. Enable "Apply address offset", and enter the starting address where this binary recovery image will be flashed (see the tables in the previous step for examples).
g. (OPTIONAL) Enable 'Verify after program' to verify that the flash programming was successful.
h. Repeat steps (starting with Click 'Add Action') above for each binary image file to be programmed into flash.

9. Execute Flash Programming.
   a. In the Target Tasks view, right-click on the imported filename and select the green Execute button to launch the programmer.
   b. If Execute is not green, the debugger is not running. The debugger must be running for this flash programmer to work.
   c. When finished flashing, terminate the debugger.
10. This is the end of the process. Now the boot loader, kernel and root file system are programmed to flash.
11. Reset or power-cycle the board and verify that u-boot appears in the board’s serial terminal.

# Chapter 5 About Yocto Project

## 5.1 Yocto Project Quick Start

The Yocto Project Quick Start explains basic concepts and the use of its core components. Step through a simple example to show how to build a small image and run it using the QEMU emulator.

## 5.2 Application Development Toolkit User's Guide

The Application Development Toolkit consists of an architecture-specific cross-toolchain and a matching sysroot that are both built by the Yocto Project build system Poky.

For more information see the *Application Development Toolkit User's Guide* located in the following SDK directory: sdk_documentation/pdf/yocto/adt-manual.pdf

## 5.3 Board Support Packages - Developer's Guide

A Board Support Package (BSP) is a collection of information that defines how to support a particular hardware device, set of devices, or hardware platform. The BSP includes information about the hardware features present on the device and kernel configuration information along with any additional hardware drivers required.

For more information see the *Board Support Packages - Developer's Guide* located in the following SDK directory: sdk_documentation/pdf/yocto/bsp-guide.pdf

## 5.4 Yocto Project Development Manual

Use a Yocto Project to develop embedded Linux images and user-space applications to run on targeted devices.

For more information see the *Yocto Project Development Manual* located in the following SDK directory: sdk_documentation/pdf/yocto/dev-manual.pdf

## 5.5 Yocto Project Linux Kernel Development Manual

The Yocto Project Linux Kernel Development Manual describes how to work with Linux Yocto kernels and provides some conceptual information on the construction of the Yocto Linux kernel tree.

For more information see the *Yocto Project Linux Kernel Development Manual* located in the following SDK directory: sdk_documentation/pdf/yocto/kernel-dev.pdf

## 5.6 Yocto Project Profiling and Tracing Manual

The Yocto Project Profiling and Tracing Manual presents a set of common and generally useful tracing and profiling schemes along with their applications (as appropriate) to each tool.

For more information see the *Yocto Project Profiling and Tracing Manual* located in the following SDK directory: sdk_documentation/pdf/yocto/profile-manual.pdf

## 5.7 Yocto Project Reference Manual

The Yocto Project uses the Poky build tool to construct complete Linux images.

For more information see the *Yocto Project Reference Manual* located in the following SDK directory: sdk_documentation/pdf/yocto/poky-ref-manual.pdf

# Chapter 6
# Configuring DPAA Frame Queues

## 6.1 Introduction

Describes configurations of Queue Manager (QMan) Frame Queues (FQs) associated with Frame Manager (FMan) network interfaces for the QorIQ Data Path Acceleration Architecture (DPAA). The relationship of the FMan and the QMan channels and work queues are illustrated by examples.

The basic configuration examples for QMan FQs provided yield straightforward and reliable DPAA performance. These simple examples may then be fine tuned for special use cases. For additional information and understanding of advanced system level features please refer to the DPAA Reference Manual.

DPAA provides the networking specific I/Os, accelerator/offload functions, and basic infrastructure to enable efficient data passing, without locks or semaphores, within the multi-core QorIQ SoC between:

1. The Power Architecture cores (and software)
2. The network and I/O interfaces through which that data arrives and leaves
3. The accelerator blocks used by the software to assist in processing that data.

Hardware-managed queues which reside in and are managed by the QMan provide the basic infrastructure elements to enable efficient data path communication. The data resides in delimited work units of frames/packets between cores, hardware accelerators and network interfaces. These hardware-managed queues, known as Frame Queues (FQs), are FIFOs of related frames. These frames comprise buffers that hold a data element, generally a packet. Frames can be single buffers or multiple buffers (using scatter/gather lists).

FQ assignment to consumers i.e., cores, hardware accelerators, network interfaces, are programmable (not hard coded). Specifically, FQs are assigned to work queues which in turn are grouped into channels. Channels which represent a grouping of FQs from which a consumer can dequeue from, are of two types:

- Pool channel: a channel that can be shared between consumers which facilitates load balancing/spreading. (Applicable to cores only. Does not apply to hardware accelerators or 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 Power Architecture and DPAA technology
- The configuration information provided in this document assumes a top bin platform frequency.

## 6.2 FMan Network interface Frame Queue Configuration

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

Each network interface has an ingress and an egress direction. The ingress direction is defined as the direction from the network interface to the cores. The egress direction is defined as the direction from the cores to the network interfaces.

FQs associated with FMan network interfaces can be either ingress or egress FQs. Ingress FQs are referred to FQs used in the ingress direction to store packets received from network interfaces to be processed by the cores. Egress FQs are referred to FQs used in the egress direction to store packets to be transmitted by FMan out of its network interfaces.

## 6.3 FMan network interface ingress FQs configuration

Dependencies for configuration of the ingress Frame Queues (FQs) is dependent on the QMan mechanism used to load balance/spread received packets across the multiple cores in QorIQ DPAA.

Two mechanisms are offered:

1. Dynamic load balancing
    - Load spread the packets (from ingress FQs) to the cores based on actual core availability/readiness.
    - Achieved through the use of QMan pool channel (i.e. a channel which can be shared by multiple cores).
    - Maintaining packet ordering (e.g. when packets are being forwarded) is achieved through the following two mechanisms:
        a. Order preservation; ensures that related packets (e.g. a sequence of packets moving between two end points) are processed in order (and typically one at a time).
        b. Order restoration; allows packets to be processed out of order and then restores their order later on before they are transmitted out to the network interfaces.
    - Improves core work load balancing over a static distribution based approach scheme but will not maintain core affinity because a FQ may get processed by multiple cores.
2. Static distribution
    - Static association between FQs and cores; FQs are always processed by the same core.
    - Achieved through the use of QMan dedicated channel (i.e. a channel which supplies FQs to a specific core).
    - Static not dynamic, doesn't react to core load, assigns work to the cores in a static or fixed manner.
    - Does not 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.

## 6.4 Ingress FQs common configuration guidelines

Guidelines and examples for configuring ingress Frame Queues (FQs) in the QorIQ DPAA are shown.

Following guidelines apply regardless of the load balancing mechanism(s) configured:

- Maximum number of ingress FQs for all ingress interfaces on the device (including any of the separate FQs that are used to serve as an order restoration point (ORP)): 1024
- Maximum number of ingress FQs per work queue (FIFO of FQs):
- - 64 if the aggregate bandwidth of the configured network interface(s) on the device is higher than 10 Gbit/s.
  - 128 if the aggregate bandwidth of the configured network interface(s) on the device is 10 Gbit/s or lower.
- The aggregate bandwidth of the configured network interface(s) on the device receiving packets into FQs associated to the same work queue should not exceed 10 Gbit/s. In other words, the recommended maximum incoming rate into a single work queue is 10 Gbit/s. If the configured network interface(s) on the device is higher than 10 Gbit/s, then multiple work queues should be used.
- Since the Single Frame Descriptor Record (SFDRs) reservation scheme is recommended for the egress FQs (FMan network interface egress FQs configuration) and any other FQs assigned to high priority work queues will also use these reserved SFDRs, careful consideration should be given to the required number of ingress FQs assigned to the high priority work queues as SFDRs are a scarce QMan resource (there is a total of 2K SFDRs). One needs to leave sufficient SFDRs for FQs not using the reserved SFDRs (e.g. ingress FQs assigned to medium or low priority work queues).

As an example, if one allocates 1024 ingress FQs and the aggregate bandwidth of the configured network interface(s) on the device is higher than 10 Gbit/s, then a minimum of 16 work queues would be required based on the above guidelines. Assuming that all 1024 FQs are to be scheduled at the same priority using a dynamic load balancing scheme, a minimum of 6 pool channels would need to be used (based on the fact that up to 3 work queues can be used within a medium or low priority tier).

The guideline “maximum of 1024 ingress FQs for all ingress interfaces” results from the size of the internal memory in QMan that is used to cache Frame Queue Descriptors (FQDs). This internal memory is sized to 2K entries. To achieve high, deterministic and reliable performance under worst-case packet workload (back-to-back 64-byte packets enqueued to FQs on a rotating basis), all ingress FQDs must remain in the QMan internal cache. FQD cache misses increase the time required to enqueue packets as the FQD may need to be read from external memory. This in return could result in received packets being discarded by the MAC due MAC FIFO overflow condition as a result of the back-pressure applied by the FMan to the MAC as there is little buffering between the MAC and the point at which incoming packets are enqueued onto the ingress FQs.

Although a device configured with a number of ingress FQs higher than the size of the QMan FQD internal cache would operate at high performance with no packet discards if the incoming traffic exhibited some level of temporal locality, it is generally recommended that the device be engineered such that ingress path operates at line rate under worst case packet workload to avoid unnecessary packets losses and to make effective use of QMan to

prioritize and apply appropriate QoS if there is congestion in a downstream element (e.g. cores). Since all FQs defined on the device shared the QMan 2K internal FQD cache, the recommended maximum number of ingress and egress FQs is even more constrained so that there is adequate space left for caching FQDs assigned to accelerators.

With regards to congestion management, the default mechanism for managing ingress FQ lengths is through buffer management. Input to FQs is limited to the availability of buffers in the buffer pool used to supply buffers to the FQs. Although very efficient and simple, when a buffer pool is shared by multiple FQs, there is no protection between the FQs sharing the buffer pool and as a result a FQ could potentially occupy all the buffers.

Queue management mechanisms can be configured (e.g. tail drop/WRED) to improve congestion control however appropriate software must be in place to handle enqueue rejections as a result of queue congestion.

## 6.5 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.

## 6.6 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.

## 6.7 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. On P4080/P3041/P5020, due to errata number QMAN-A002, allowable values for ICS are: 0 and 15’ h7FFF.
  - Don’t set force SFDR allocate unless FQ needs performance optimization.
  - FQD CPC stashing enabled.
  - Dequeued Frame Data, Annotation, and FQ Context stashing: application dependent.
  - ORP disabled.

## 6.8 FMan network interface egress FQs configuration

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

FQ Configurations:

- Maximum number of egress FQs for all network interfaces: 128.
- Minimum number of egress FQs per network interface: 1.
- Maximum number of egress FQs per work queue: 8.

- Egress FQ descriptor attributes settings:
  - Prefer in cache.
  - Don't set hold active .
  - Don't set avoid blocking.
  - Set force SFDR allocate to ensure that egress queues make use of the reserved SFDRs; the SFDR reservation threshold field of the QMan SFDR configuration register must also be set accordingly (5 SFDRs per egress FQ + 3 extra SFDRs as required by QMan).
  - Intra-class scheduling set to zero (0) unless a more advanced scheduling scheme is required.
  - FQD CPC stashing enabled.
  - ORP disabled.

.

## 6.9 Accelerator Frame Queue Configuration

Configurations for Frame Queues (FQs) used to communicate with accelerators for QorIQ DPAA are shown.

FQ accelerator Guidelines:

- Since the Single Frame Descriptor Record (SFDRs) reservation scheme is recommended for the egress FQs (FMan network interface egress FQs configuration) and any other FQs assigned to high priority work queues will also use these reserved SFDRs, careful consideration should be given to the required number of accelerator FQs assigned to the high priority work queues as SFDRs are a scarce QMan resource (there is a total of 2K SFDRs). One needs to leave sufficient SFDRs for FQs not using the reserved SFDRs (e.g. accelerator FQs assigned to medium or low priority work queues).
- Accelerator FQ descriptor attributes settings:
  - Don't set prefer in cache.
  - Don't set hold active .
  - Don't set avoid blocking.
  - FQD CPC stashing enabled.
  - Intra-class scheduling (ICS) credit set to 0 unless a more advanced scheduling scheme is required.
  - Don't set force SFDR allocate unless FQ needs performance optimization.
  - Dequeued Frame Data, Annotation, and FQ Context stashing: application dependent.
  - ORP disabled.

Generally accelerators are used in a request/response manner and in cases where a pair of FQs is needed per session/flow to communicate with accelerators, one may need to allocate a very large number of FQs (in the order of thousands). At times when many FQs allocated to an accelerator are active, this situation can result in having significant amount of cache consumed for storing the corresponding FQ descriptors. This in turn may negatively impact overall system performance.

To ensure optimal resource utilization (e.g. QorIQ caches), maximize throughput and avoid overload, it is recommended that the number of outstanding requests/responses to an accelerator be regulated. Typically, for a given accelerator, regulating the number of outstanding requests/responses across all its FQs to a few hundredths should be sufficient to maintain high throughput without overloading the system. Regulating the number of outstanding requests/responses to an accelerator can be achieved through various methods.

One method is to keep track in software of the total number of outstanding requests/responses to an accelerator and once this number exceeds a threshold, software would stop sending requests to that accelerator.

Another method is to make use of the congestion management capabilities of QMan. Specifically, all FQs allocated to an accelerator can be aggregated into a congestion group. Each congestion group can be configured to track the number of Frames in all FQs in the congestion group. Once this number exceeds a configured threshold, the congestion group enters congestion. When a congestion group enters congestion, QMan can be configured to rejects enqueues to any FQs in the congestion group and/or sent notification indicating that the congestion group has entered congestion. If a Frame (or request) is not going to be enqueued, it will be returned to the configured destination via an enqueue rejection notification. Congestion state change notifications are generated when the congestion group either enters congestion or exits congestion. On software portals, the congestion state change notification is sent via an interrupt.

## 6.10 DPAA Frame Queue Configuration Guideline Summary

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

Four tables comprise this summary:

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

Table 10: Global Configuration Settings Summary

| Parameter or subject | Guideline |
|---|---|
| FQD stashing | Recommend QMan explicitly stash FQDs:<br>• 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.<br>• PAMU; PAACT tables used by PAMU also configured appropriately . |
| PFDR stashing | Recommend QMan explicitly stash PFDRs:<br>• QMan; the global CPC stash enable bit in the QMan PFDR_AR register must be set .<br>• PAMU; PAACT tables used by PAMU must also be configured appropriately . |
| SFDR reservation threshold | Set SFDR reservation threshold in QMan SFDR configuration register to:<br>• Total number of FQs using reserved SFDRs times 5 (5 SFDRs per FQ) plus 3 extra SFDRs as required by QMan.<br>Recommend that all egress FQs use reserved SFDRs . |

Table 11: Network Interface Ingress FQs Guidelines Summary

| Parameter or subject | Guideline |
| --- | --- |
| Maximum number of ingress FQs for all ingress interfaces on the device (including any of the separate FQs that are used to serve as an order restoration point (ORP)) | 1024 FQs |
| Maximum number of ingress FQs per work queue. | • 64 FQs per work queue if the aggregate bandwidth of the configured network interface(s) on the device is higher than 10 Gbit/s.<br>• 128 FQs per work queue if the aggregate bandwidth of the configured network interface(s) on the device is 10 Gbit/s or lower. |
| The maximum aggregate bandwidth of the configured network interface(s) on the device receiving packets into FQs associated to the same work queue | 10 Gbit/s |
| 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. |

*Table continues on the next page...*

Table 11: Network Interface Ingress FQs Guidelines Summary (continued)

| Parameter or subject | Guideline |
|---|---|
| Ingress FQ descriptor load balancing and performance related settings. | • Prefer_in_Cache: 1<br>• CPC Stash Enable: 1<br>• ORP_Enable: 0<br>• Avoid_Blocking:<br>  • 0 if static distribution or dynamic load balancing with order preservation.<br>  • 1 if dynamic load balancing with order restoration.<br>• Hold_Active<br>  • 0 if static distribution or dynamic load balancing with order restoration .<br>  • 1 if dynamic load balancing with order preservation.<br>• Force_SFDR_Allocate: 0 unless FQ needs performance optimization.<br>• 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<br>• CPC Stash Enable: 1<br>• ORP_Enable: 1<br>• Avoid_Blocking: 0<br>• Hold_Active: 0<br>• Force_SFDR_Allocate: 0<br>• ORP Restoration Window Size: 2 (corresponds to window size of 128 frames).<br>• Class Scheduling Credit: 0 |

Table 12: Network Interface Egress FQs Guidelines Summary

| Parameter or subject | Guideline |
|---|---|
| Maximum number of egress FQs for all network interfaces. | 128 FQs |
| Minimum number of egress FQs per network interface. | 1 FQ |
| Maximum number of egress FQs per work queue. | 8 FQs |

*Table continues on the next page...*

Table 12: Network Interface Egress FQs Guidelines Summary (continued)

| Parameter or subject | Guideline |
| --- | --- |
| Egress FQ descriptor performance related settings. | • Prefer_in_Cache: 1<br>• CPC Stash Enable: 1<br>• ORP_Enable: 0<br>• Avoid_Blocking: 0<br>• Hold_Active: 0<br>• Force_SFDR_Allocate: 1<br>• Class Scheduling Credit: 0 unless a more advanced scheduling scheme is required. |

Table 13: Accelerator FQs Guidelines Summary

| Parameter or subject | Guideline |
| --- | --- |
| Assignment to high priority work queues. | Should be limited enough to leave sufficient SFDRs for FQs not using the reserved SFDRs (e.g. accelerator FQs assigned to medium or low priority work queues). |
| Egress FQ descriptor performance related settings. | • Prefer_in_Cache: 0<br>• CPC Stash Enable: 1<br>• ORP_Enable: 0<br>• Avoid_Blocking: 0<br>• Hold_Active: 0<br>• Force_SFDR_Allocate: 0 unless FQ needs performance optimization .<br>• Class Scheduling Credit: 0 unless a more advanced scheduling scheme is required . |


Non-disclosure Agreement Required.

# Chapter 7
# DPAA Primer for Software Architecture

## 7.1 DPAA Primer

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

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

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

#### Benefits of the DPAA

The primary intent of the DPAA is to provide intelligence within the IO portion of the System On Chip (SOC) to route and manage the processing work associated with traffic flows to simplify ordering and load balance concerns associated with multi core processing. The DPAA hardware inspects ingress traffic and extracts user defined flows from the port traffic. It then steers specific flows (or related traffic) to a specific core or set of cores.

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

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

### 7.1.1 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.

### 7.1.2 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.

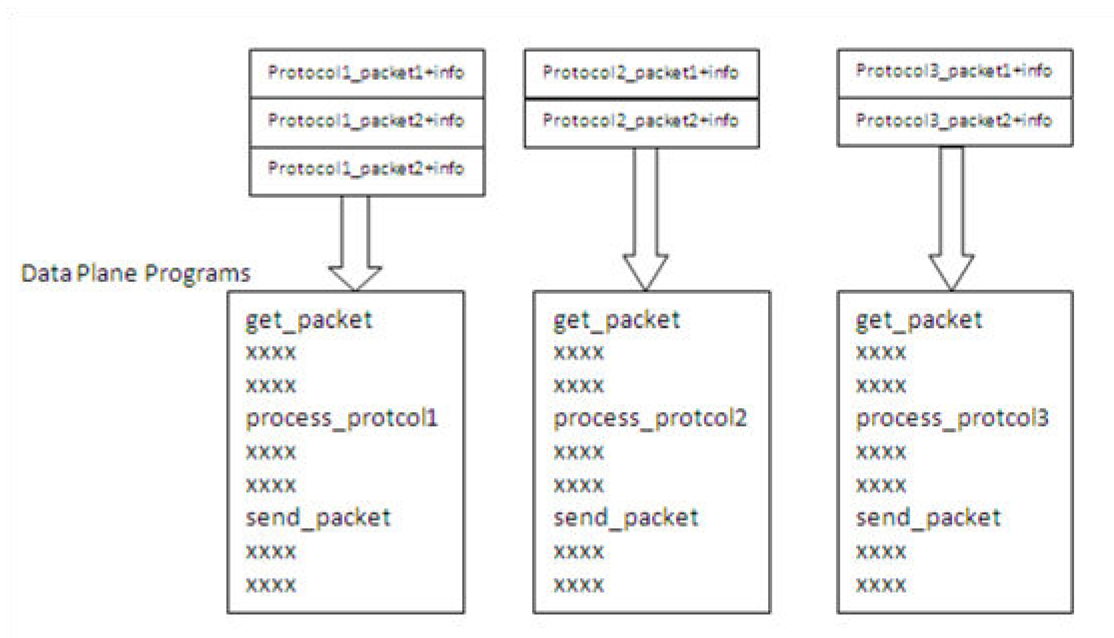
## 7.1.3 Parse/classification Software Offload

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

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

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

Figure 1: Hardware-sorted Protocol Flow

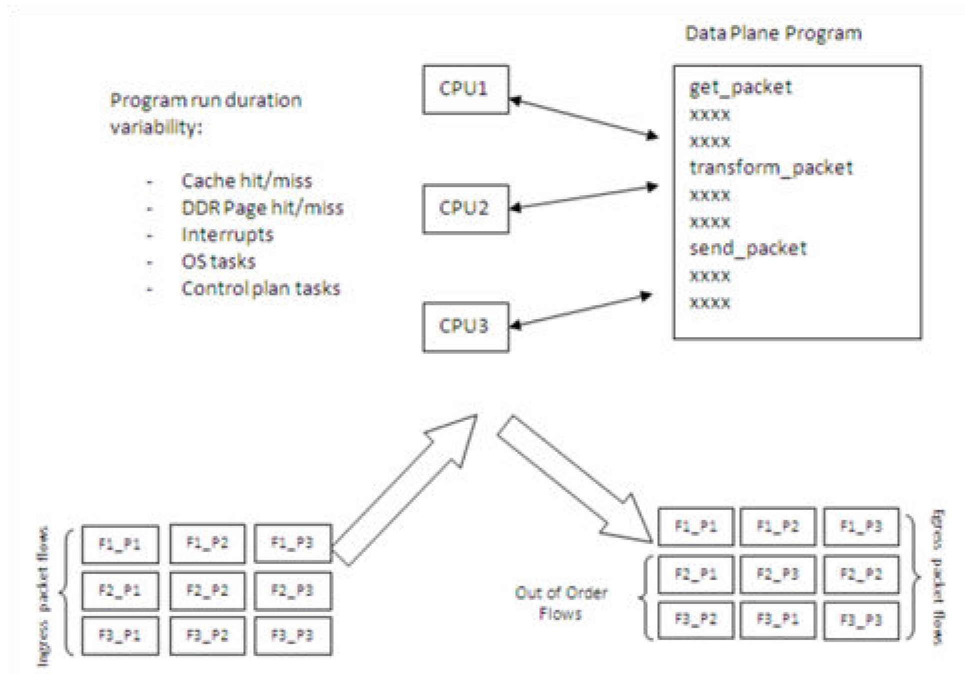



## 7.1.4 Flow Order Considerations

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

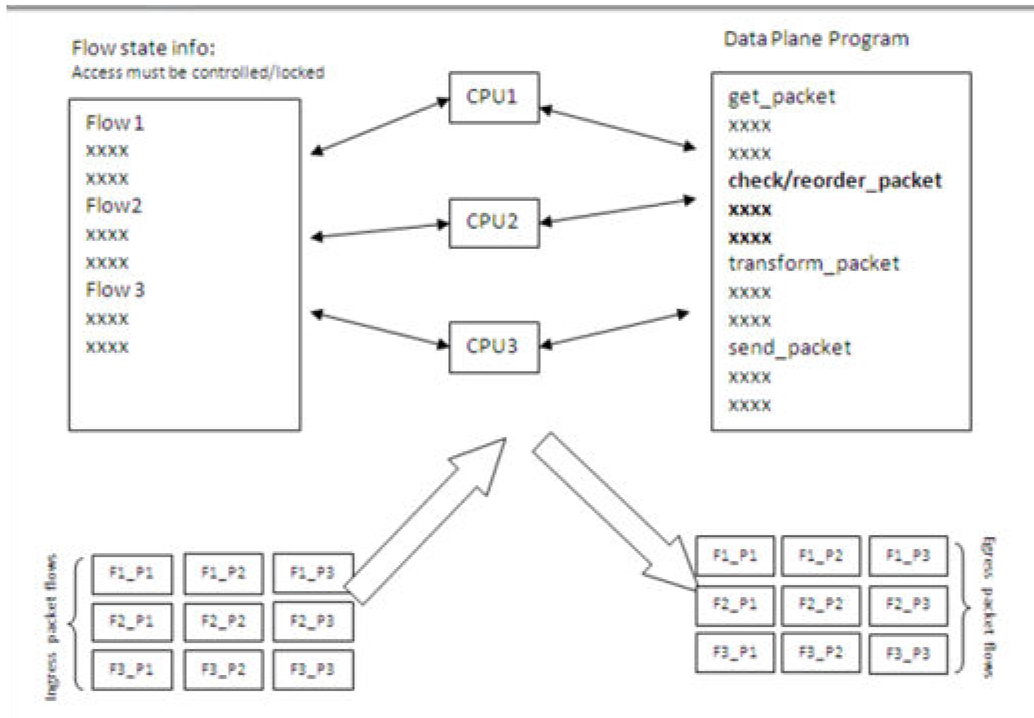
Even if the code flow is identical, factors such as cache hits/misses, DRAM page hits/misses, interrupts, control plane and OS tasks can cause some variability in the processing path, allowing egress packets to “pass” within the same flow, as shown in this figure

Figure 2: Multicore Flow Reordering




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

Figure 3: Implementing Order in Software



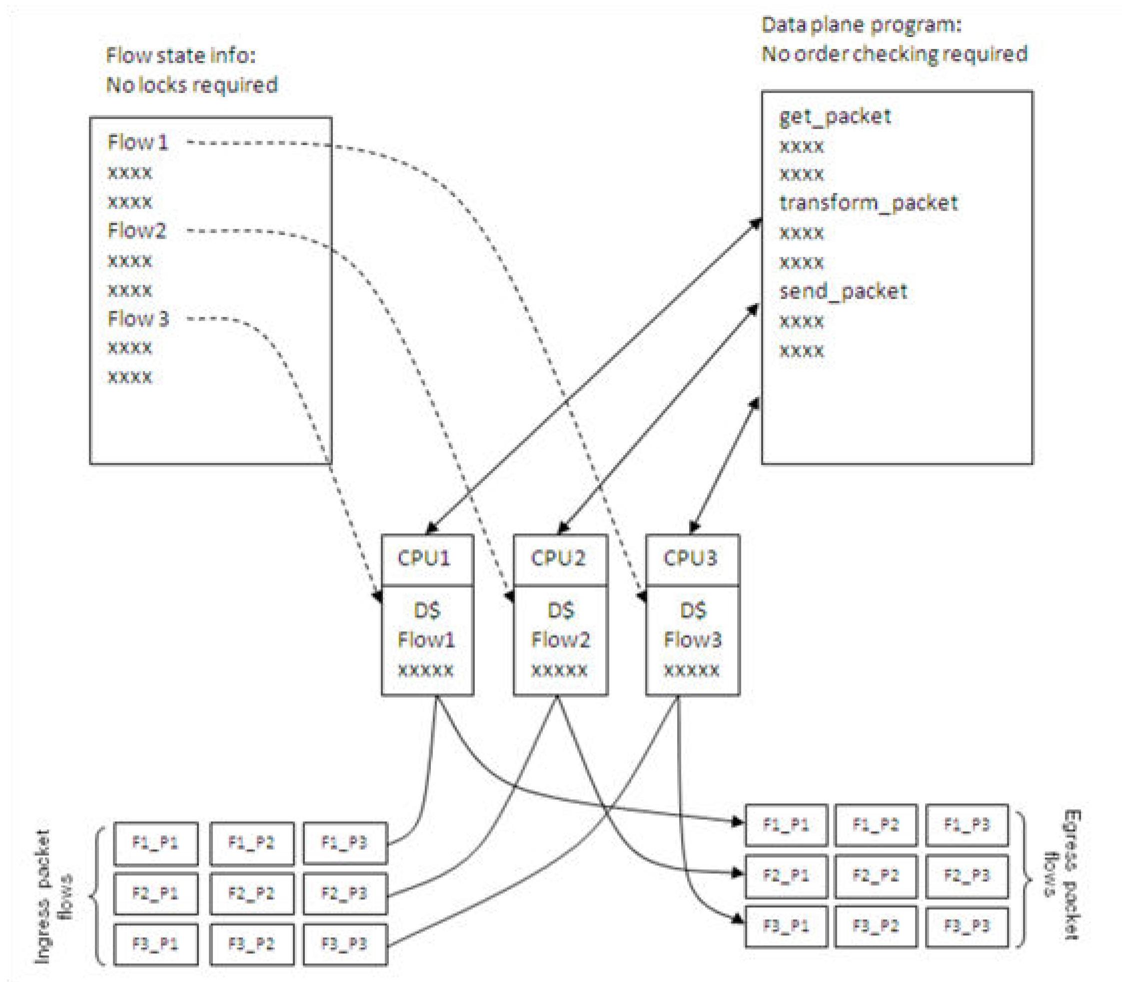

## 7.1.5 Managing Flow-to-Core Affinity

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

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

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

Figure 4: Managing Flow-to-Core Affinity




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

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


Non-disclosure Agreement Required.

# 7.2 DPAA Goals

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

The primary goals of the DPAA are as follows:

- To provide intelligence within the IO portion of the SoC.
- To route and manage the processing work associated with traffic flows.
- To simplify the ordering and load balance concerns associated with multicore processing.

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

# 7.3 FMan Overview

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

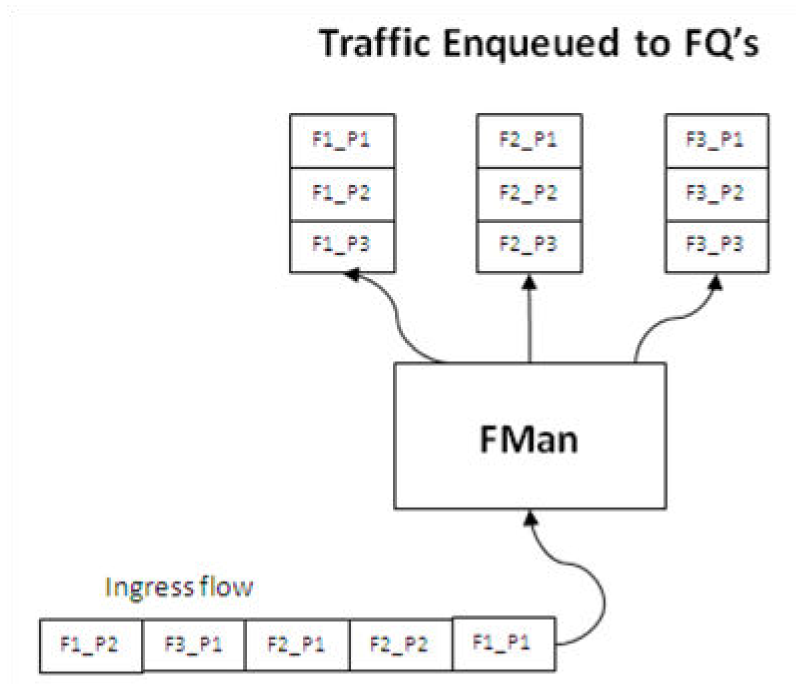
On ingress traffic, the FMan is configured to determine the traffic split required by the PCD (Parse, Classify, Distribute) function. This allows the user to decide how he wants to define his traffic: typically, by flows or classes of traffic. The PCD can be configured to route all traffic on one port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

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

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

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

Figure 5: Ingress FMan Flow




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

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

Figure 6: FMan Egress Flow

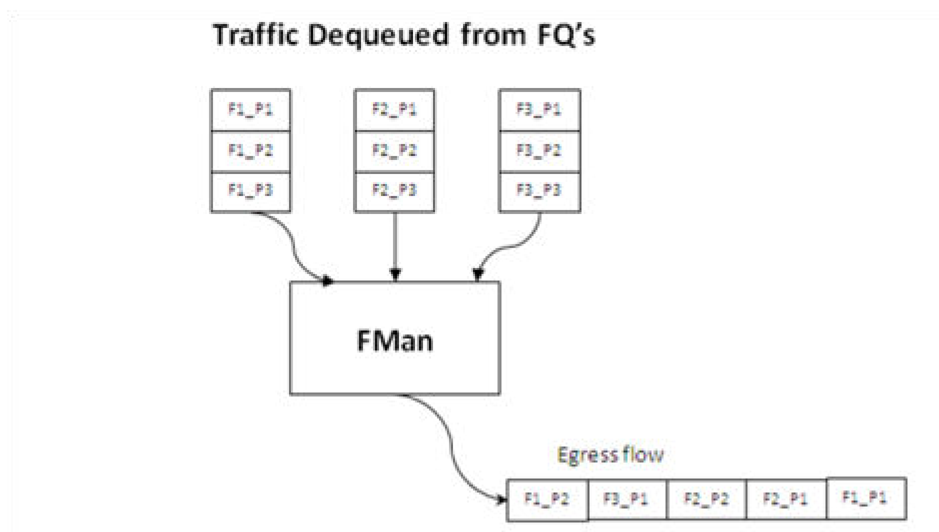
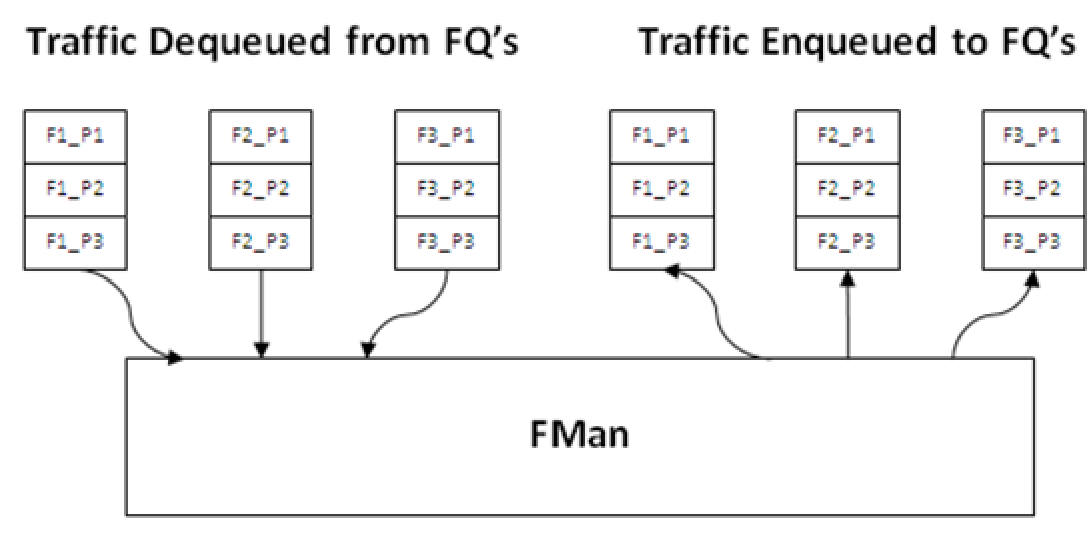


Figure 7: FMan Offline Flow

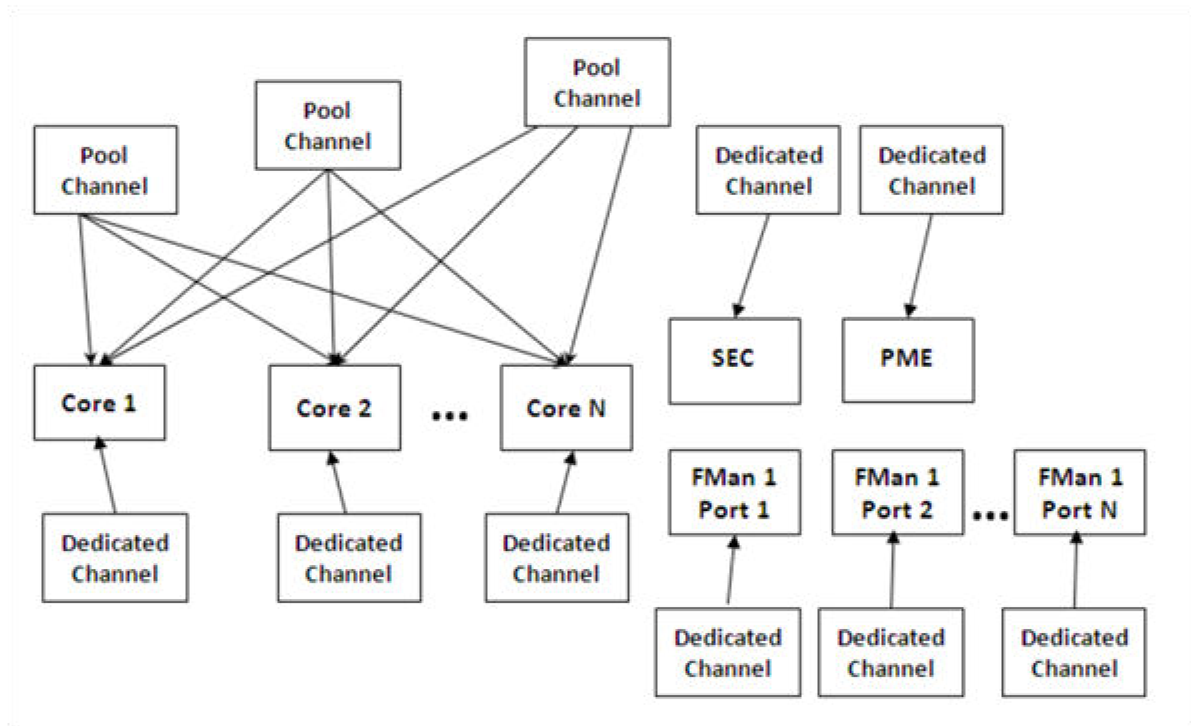



## 7.4 QMan Overview

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

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

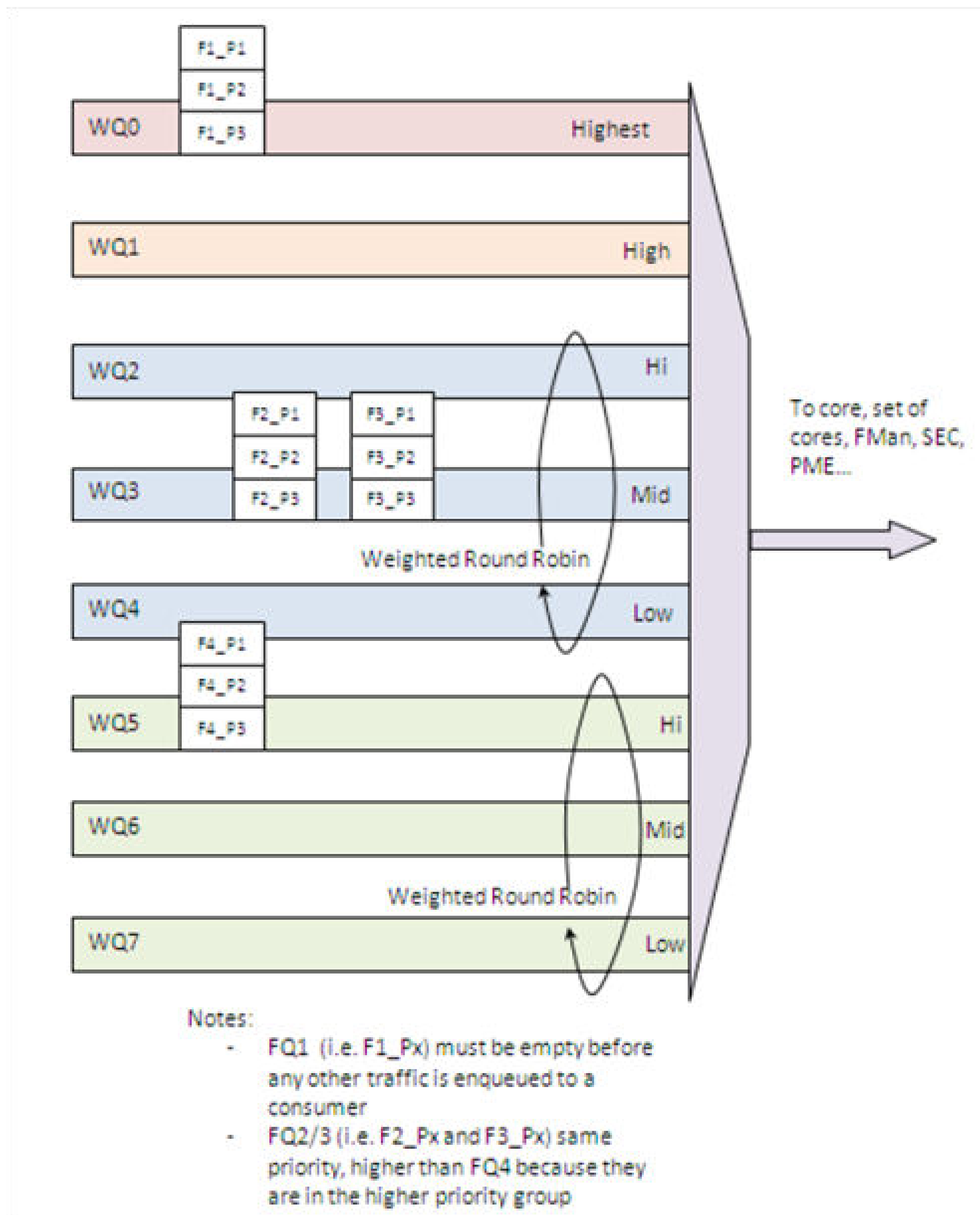
Figure 8: DPAA Channel Types




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

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

Figure 9: Prioritizing Work




### QMan: Portals

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

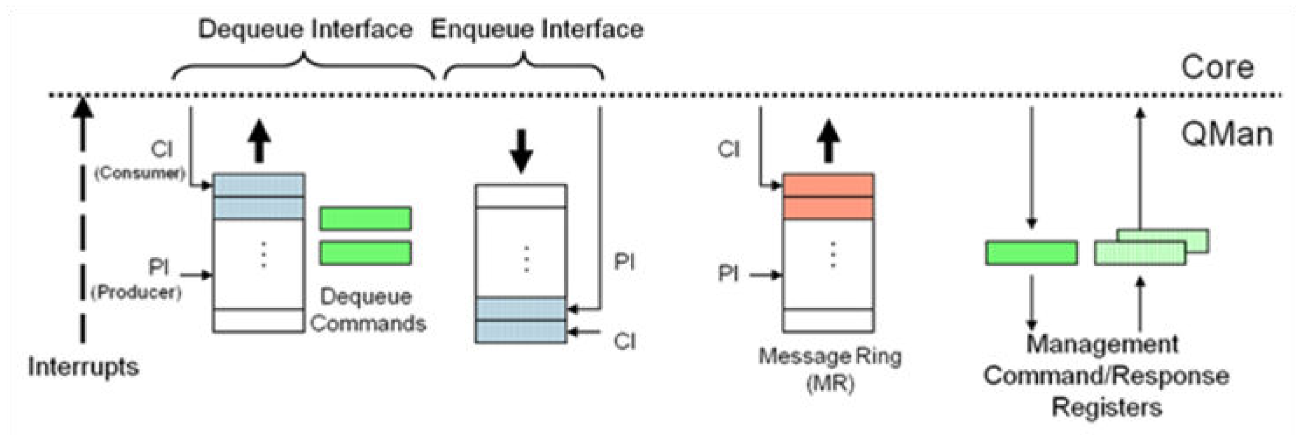
Software portals are associated with the processor cores and are, effectively, data structures that the cores use to pass (enqueue) packets to or receive (dequeue) packets from the channels associated with that portal (core). Each SoC has at least as many software portals as there are cores. Software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

The portal structure consists of the following:

- The Dequeue Response Ring (DQRR) determines the next packet to be processed.
- The Enqueue Command Ring (EQCR) sends packets from the core to the other elements.

- The Message Ring (MR) notifies the core of the action (for example, attempted dequeue rejected, and so on).
- The Management command and response control registers.

Figure 10: Processor Core Portal

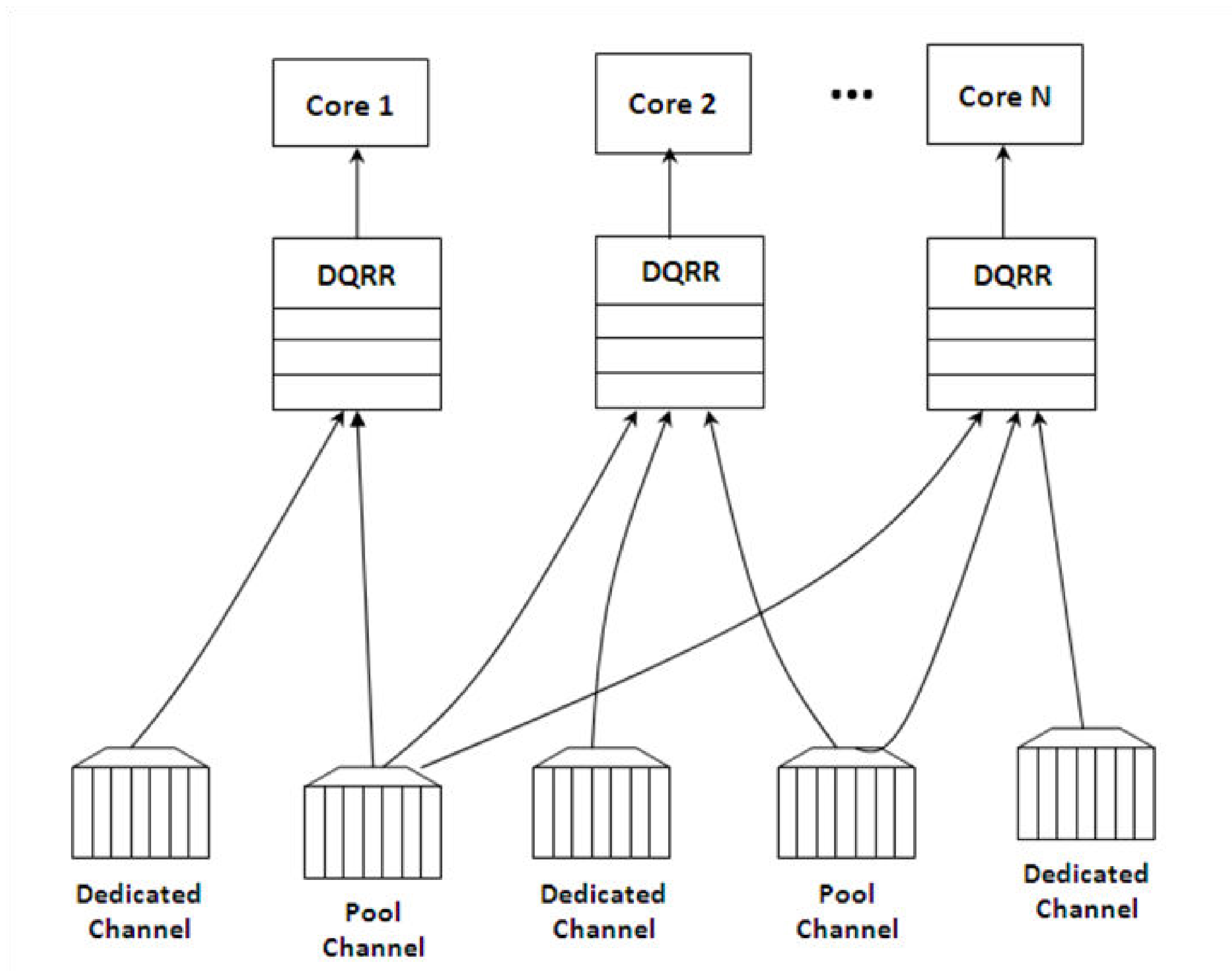



On ingress, the DQRR acts as a small buffer of incoming packets to a particular core. When a section of software performs a get packet type operation, it gets the packet from a pointer provided as an entry in the DQRR for the specific core running that software. Note that the DQRR consolidates all potential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

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

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

Figure 11: Ingress Channel to Portal Options




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

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

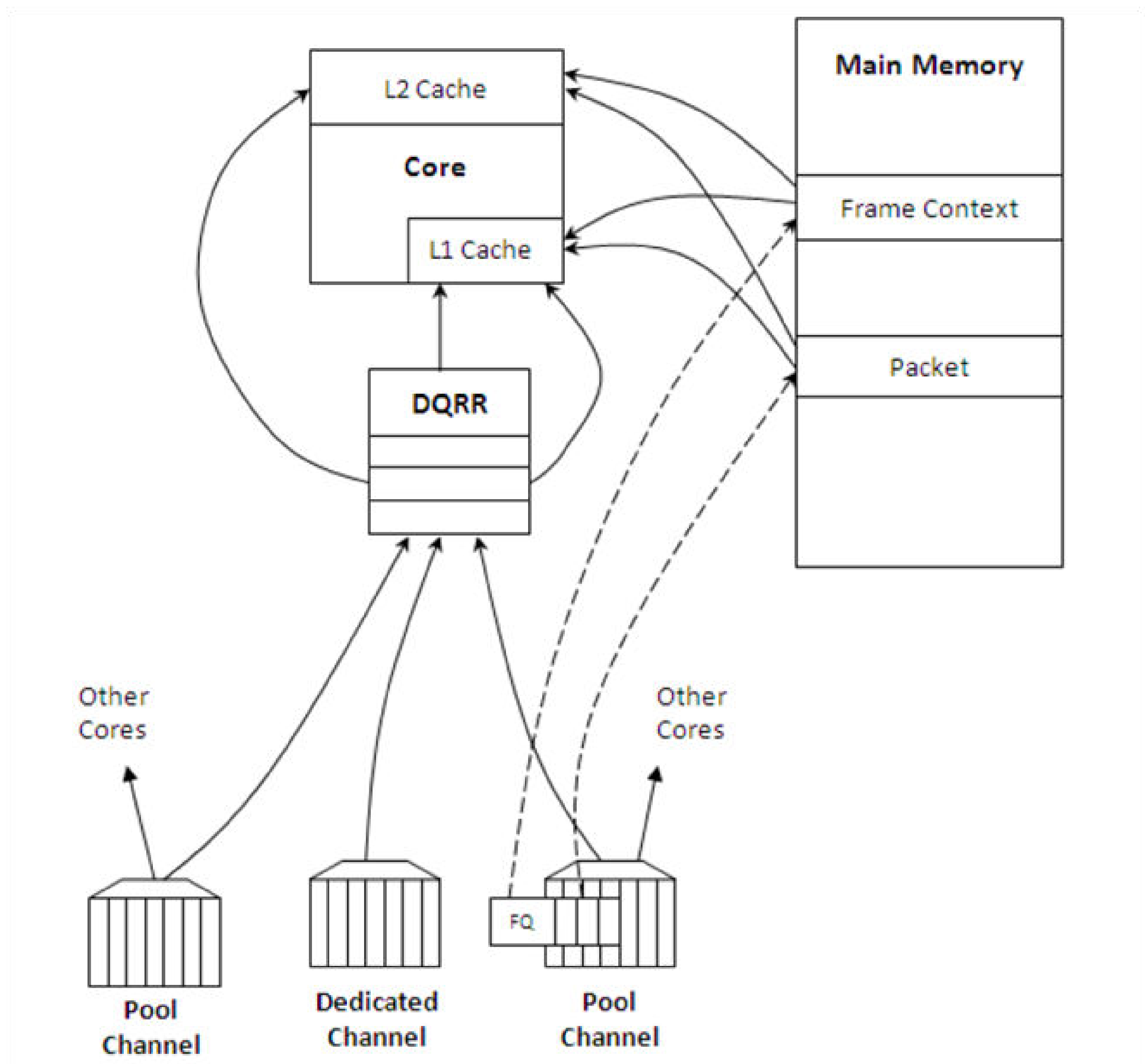
- The DQRR entry
- The packet or portion of the packet for a single buffer packet
- The scatter gather list for a multi-buffer packet
- Additional data added by FMan
- FQ context (A and B)

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

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

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

Figure 12: Cache Warming Options




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

## 7.5 QMan Scheduling

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

### QMan: Queue schedule options

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

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

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

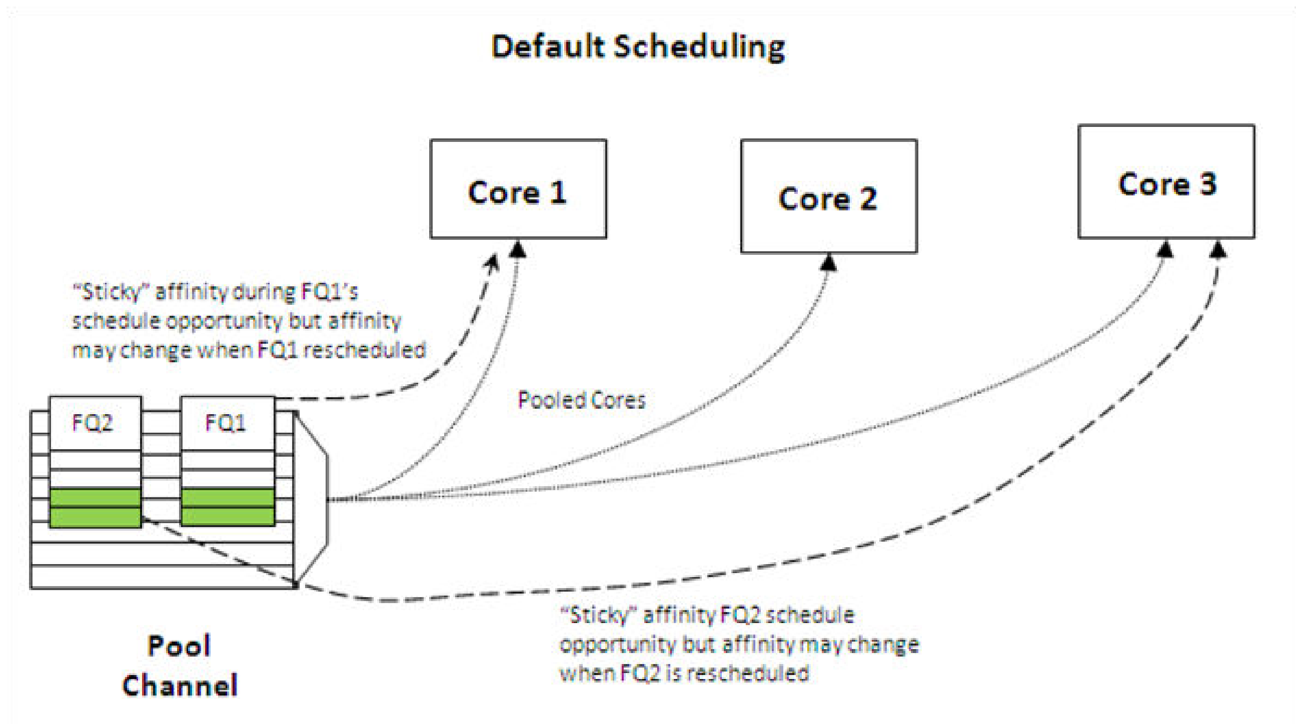
When the system is configured to use pool channels, a portal may get frames from more than one channel and a channel may provide frames (work) to more than one portal (core). QMan dequeues frames using the same mechanism described above (updating DQRR) and QMan also provides for some specific scheduling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

### QMan: Default Scheduling

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

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

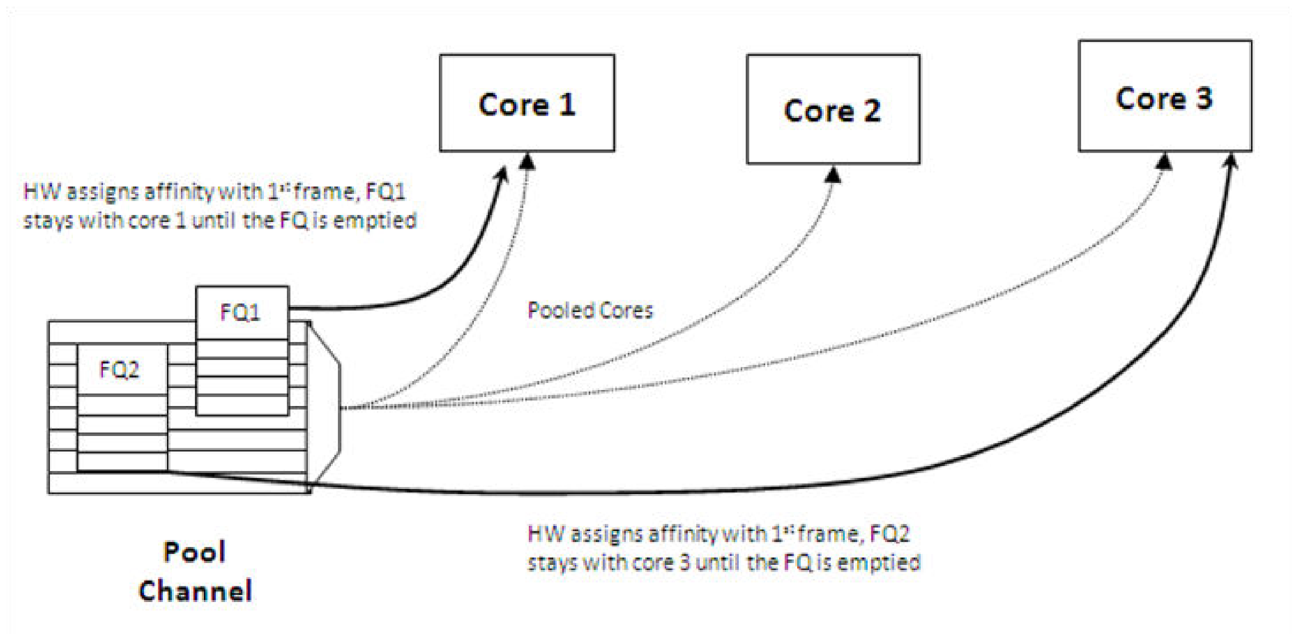
Figure 13: Default Scheduling




### QMan: Hold Active Scheduling

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

Figure 14: Hold Active Scheduling

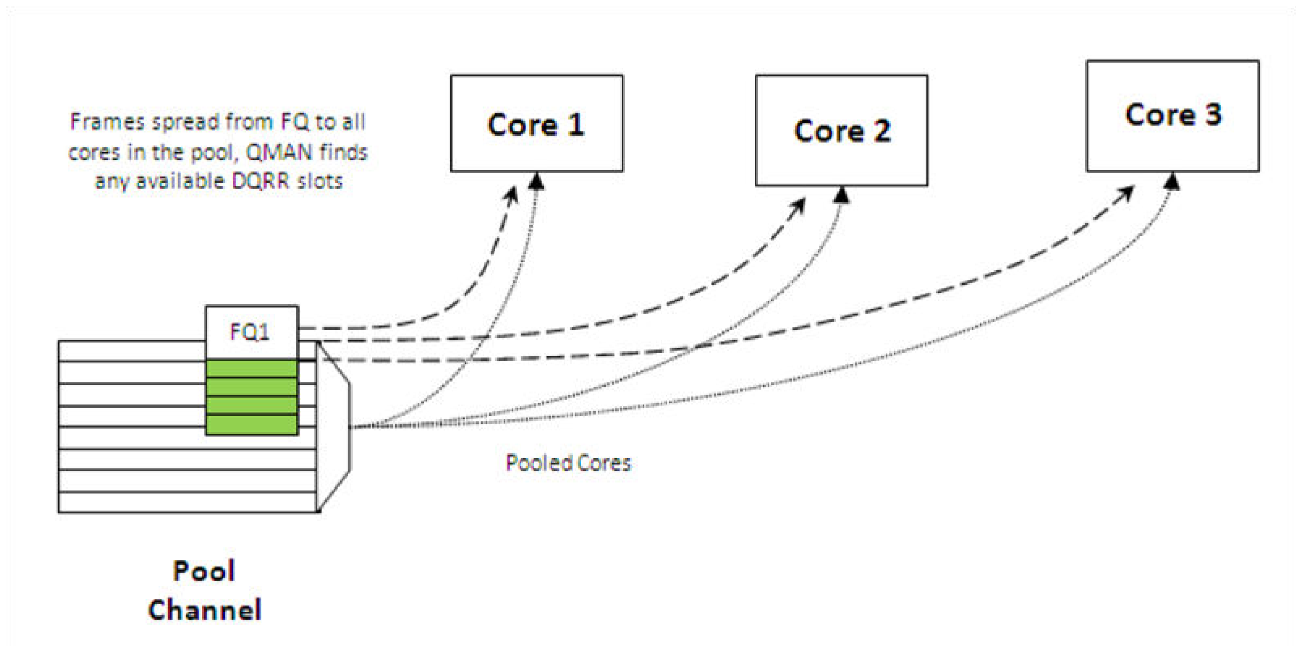



### QMan: Avoid blocking scheduling

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

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

Figure 15: Avoid Blocking Scheduling



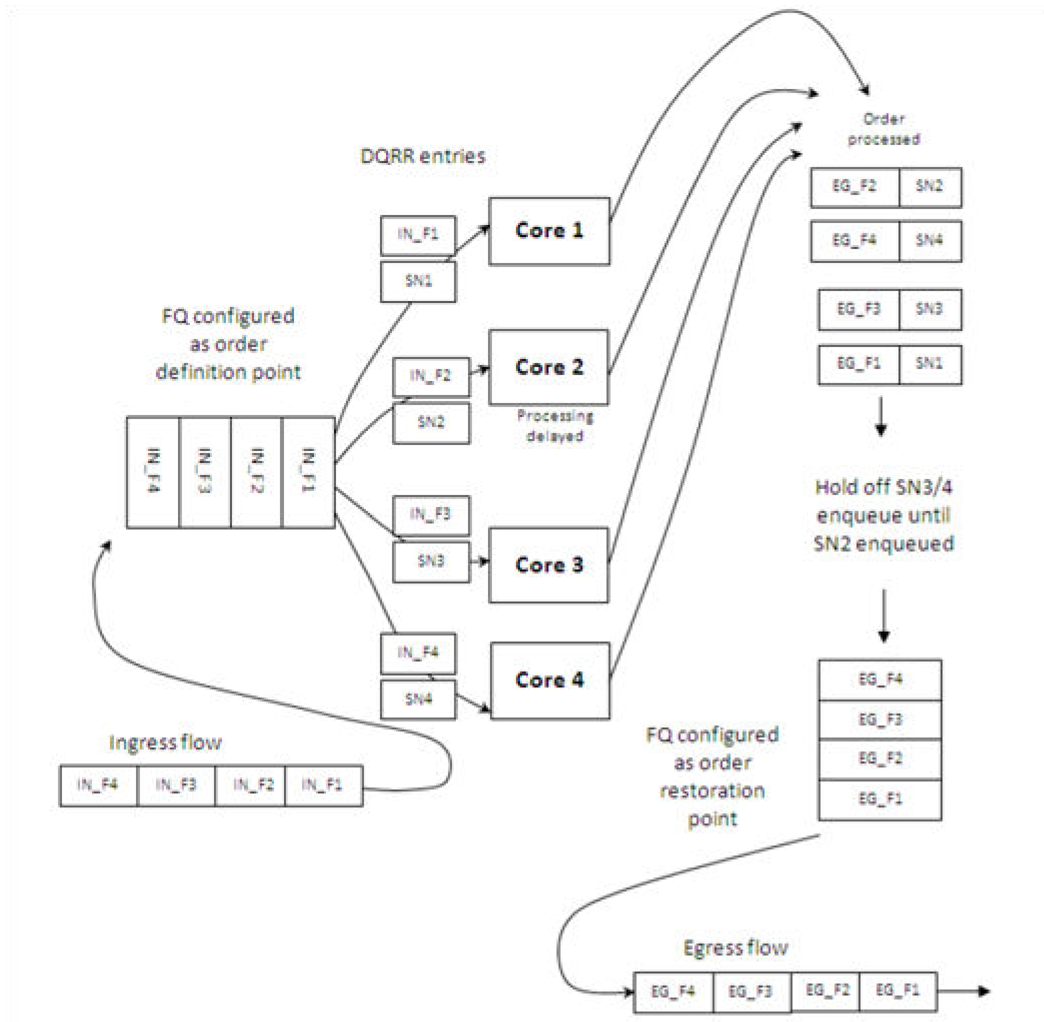

### QMan: Order Definition/ Restoration

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

NOTE

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

Figure 16: Order Definition/Restoration




As processors enqueue packets for egress, it is possible that they may skip a sequence number because of the nature of the protocol being processed. To handle this situation, each FQ that participates in the order restoration service maintains its own Next Expected Sequence Number (NESN). When the difference between the sequence number of the next expected and the most recently received sequence number exceeds the configurable ORP threshold, QMan gives up on the missing frame(s) and autonomously advances the NESN to bring the skew within threshold. This causes any deferred 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.

# 7.6 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,


Non-disclosure Agreement Required.

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).

## 7.7 Order Handling

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

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

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

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

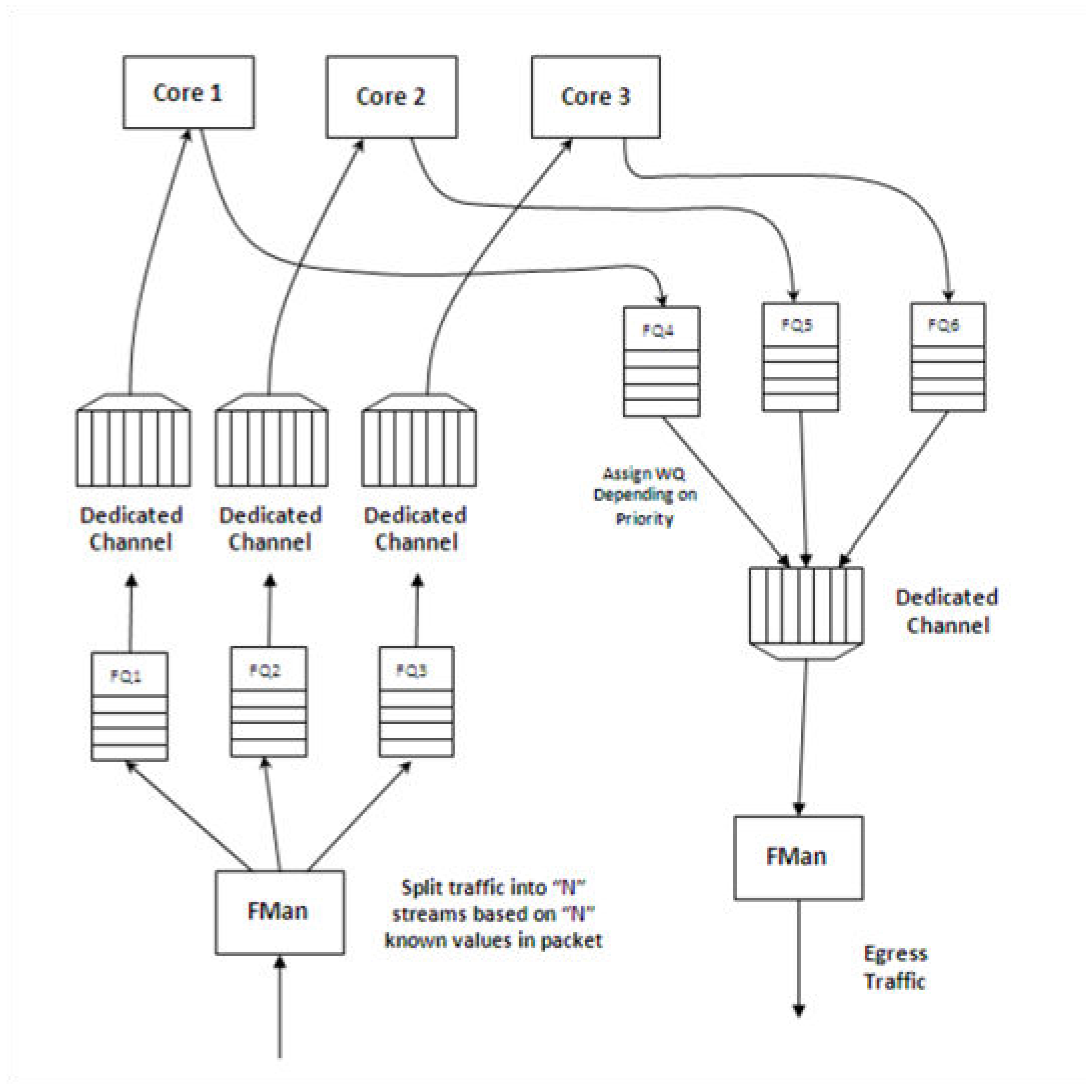
NOTE

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

#### Using the exact match flow definition to preserve order

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

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




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

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

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

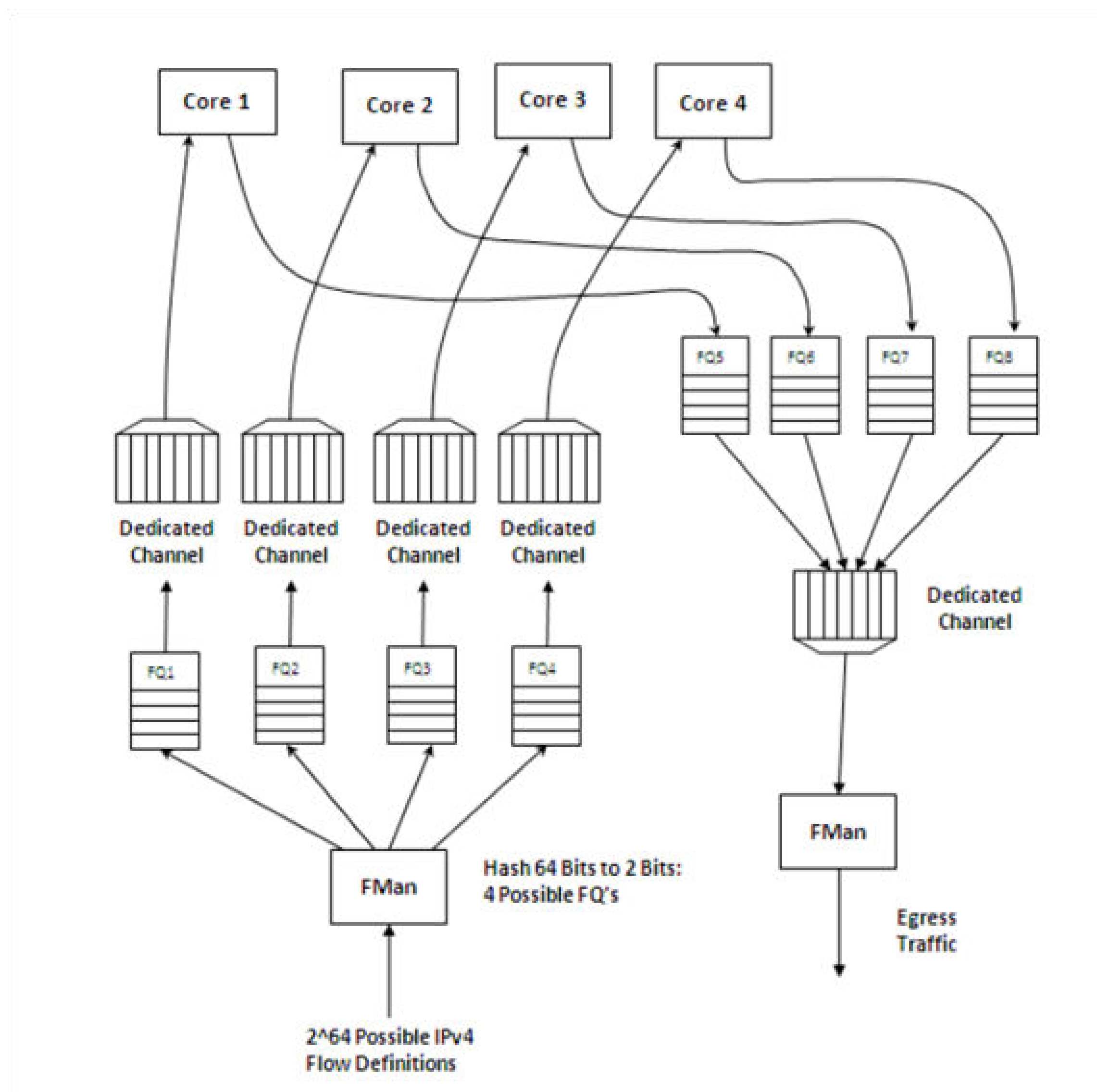
### Using hashing to distribute flows across cores

The FMan can be configured to extract data from a field or fields within the data frame, build a key from that, and then hash the resultant key into a smaller number. This is a useful technique to handle a larger number of flows while ensuring that a particular flow is always associated with a particular core. An example is to define a flow as an IPv4 source + IPv4 destination address. Both fields together constitute 64 bits, so there are 264 possible combinations for the flow in that definition. The FMan then uses a hash algorithm to compress this into

a manageable number of bits. Note that, because the hash algorithm is consistent, packets from a particular flow always go to the same FQ. By utilizing this technique, the flows can be spread in a pseudo-random, consistent (per flow) manner to a smaller number of FQs. For example, hashing the 64 bits down to 2 bits spreads the flows among four queues. Then these queues can be assigned to four separate cores by using a dedicated channel; effectively, this appears as a single-core implementation to any specific flow.

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

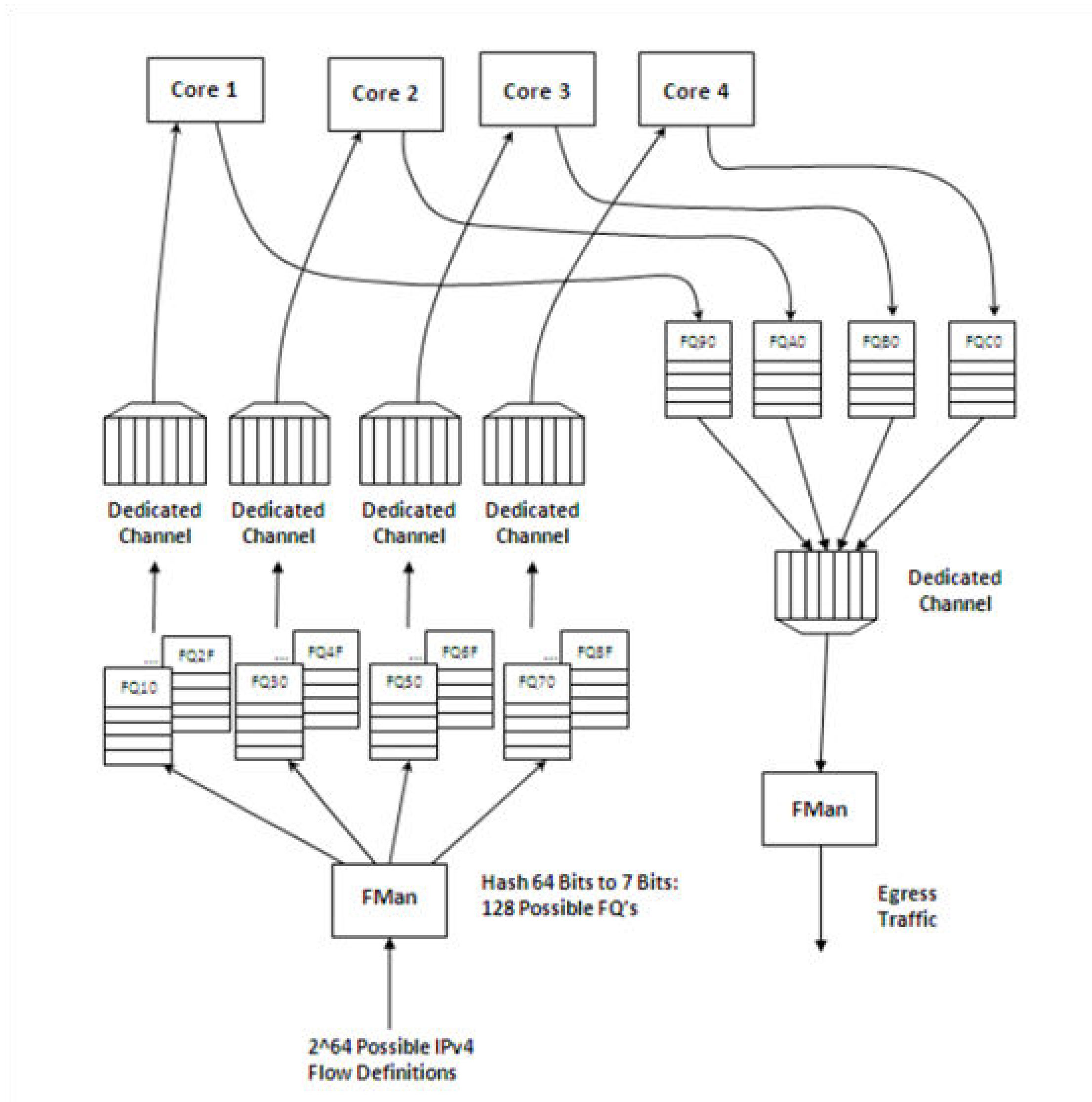
Figure 18: Simple flow distribution via hash (order preserved)




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

Note that there are 24 bits architected for the FQ ID, so there may be as many as 16 million FQs in the system. Although this total may be impractical, this does allow for the user to define more FQs than expected flows in order to reduce the likelihood of a hash collision; it also allows flexibility in assigning FQID's in some meaningful manner. It is also possible to hash some fields in the data frame and concatenate other parse results, possibly allowing a defined one-to-one flow to FQ implementation without hash collisions.

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



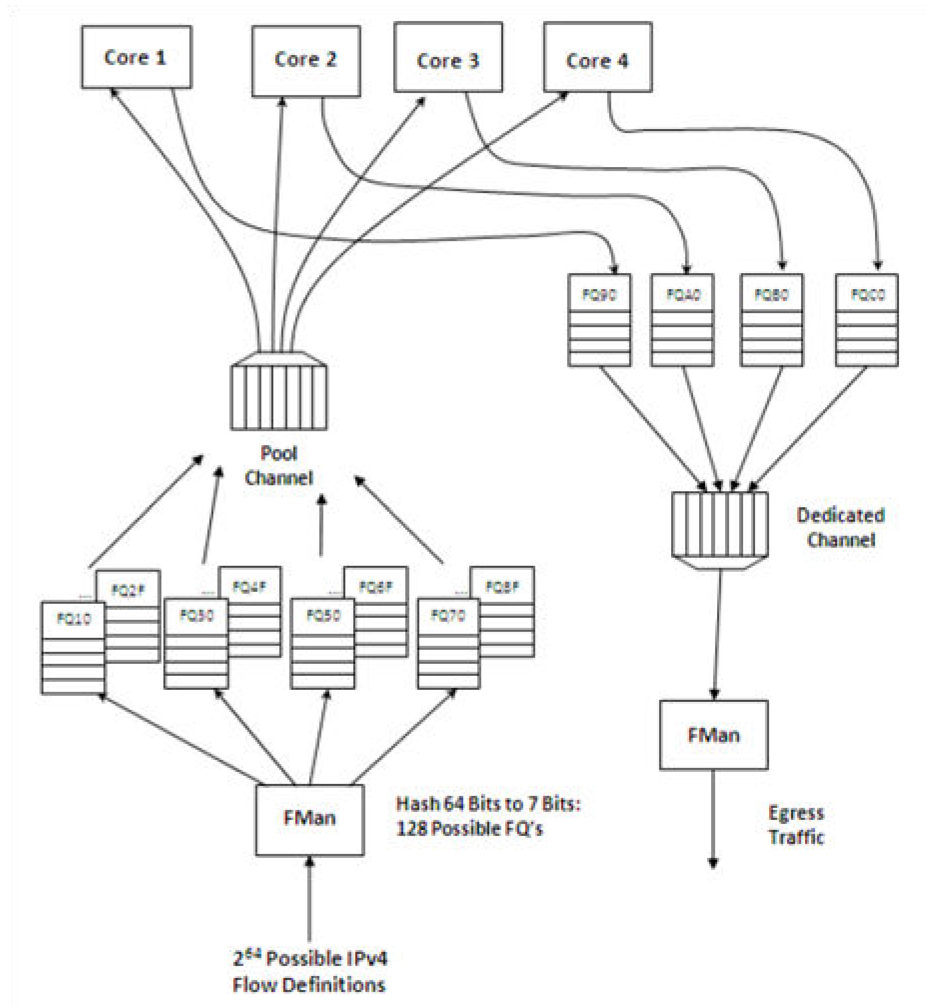

## 7.8 Pool Channels

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

So far, the techniques discussed in this white paper have involved assigning specific flows to the same core to ensure that the same core always processes the same flow or set of flows, thereby preserving flow order. However, depending on the nature of the flows being processed (that is, variable frame sizes, difficulty efficiently

spreading due to the nature of the flow contents, and so on), this may not effectively balance the processing load among the cores. Alternatively, a user may employ a pool channel approach where multiple cores may pool together to service a specific set of flows. This alternative approach allows potentially better processing balance, but increases the likelihood that packets may be processed out of order allowing egress packets to pass ingress packets. When the application does not require flows to be processed in order, the pool channel approach allows the easiest method for balancing the processing load. When a pool channel is used and order is required, the software must maintain order. The hardware order preservation may be used by the software to implement order without requiring locked access to shared state information. When the system uses a software lock to handle order then the default scheduling and hold active scheduling tends to minimize lock contention.

Figure 20: Using pool channel to balance processing




### Order preservation using hold active scheduling and DCA mode

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

When the FQ is emptied, QMan places the FQ in hold suspended state, which means that no further work for that FQ is enqueued to any core until all previously enqueued work is completed. Because DCA mode effectively

holds off the consumption notification (from the core to QMan) until the resultant processed frame is enqueued for egress, this implies processing is completely finished for any frames in flight to the core. After all the in-flight frames have been processed, QMan reschedules the FQ to the appropriate core.

NOTE

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

Figure 21: Hold active to held suspended mode

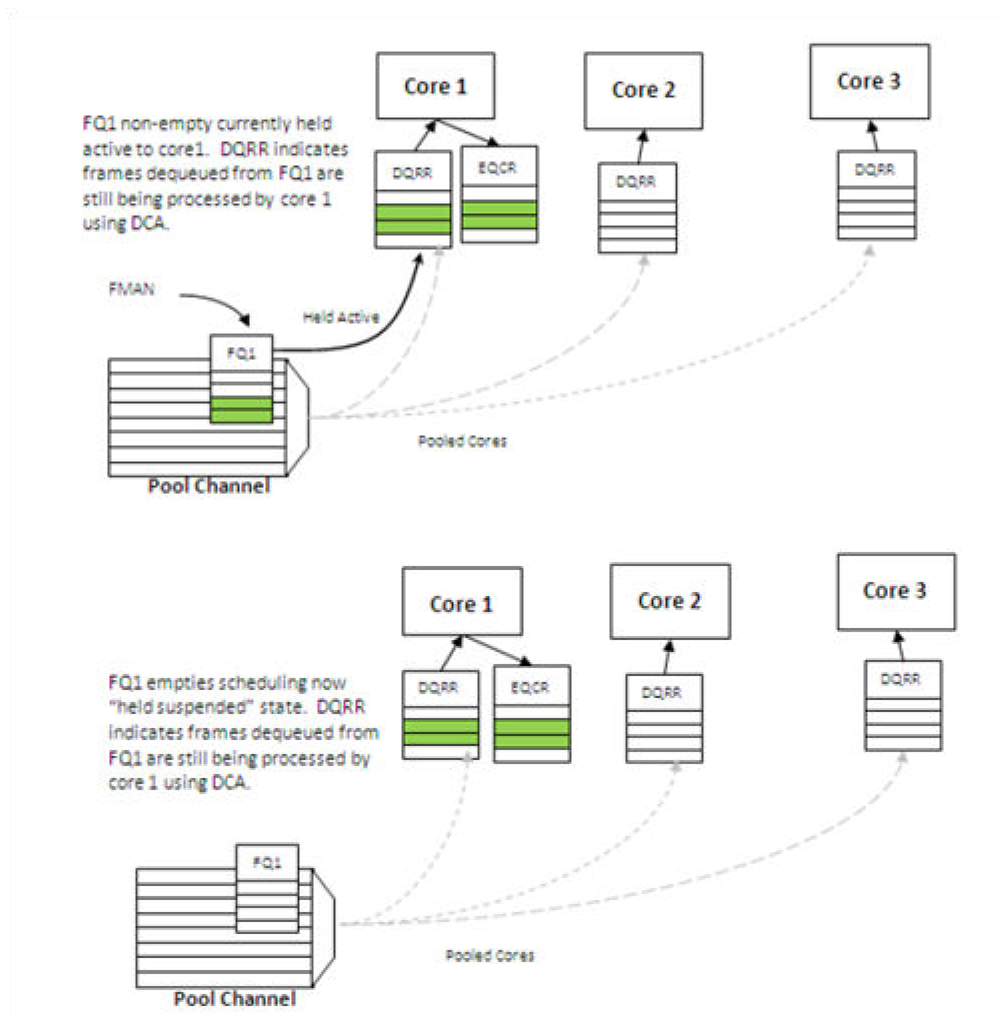
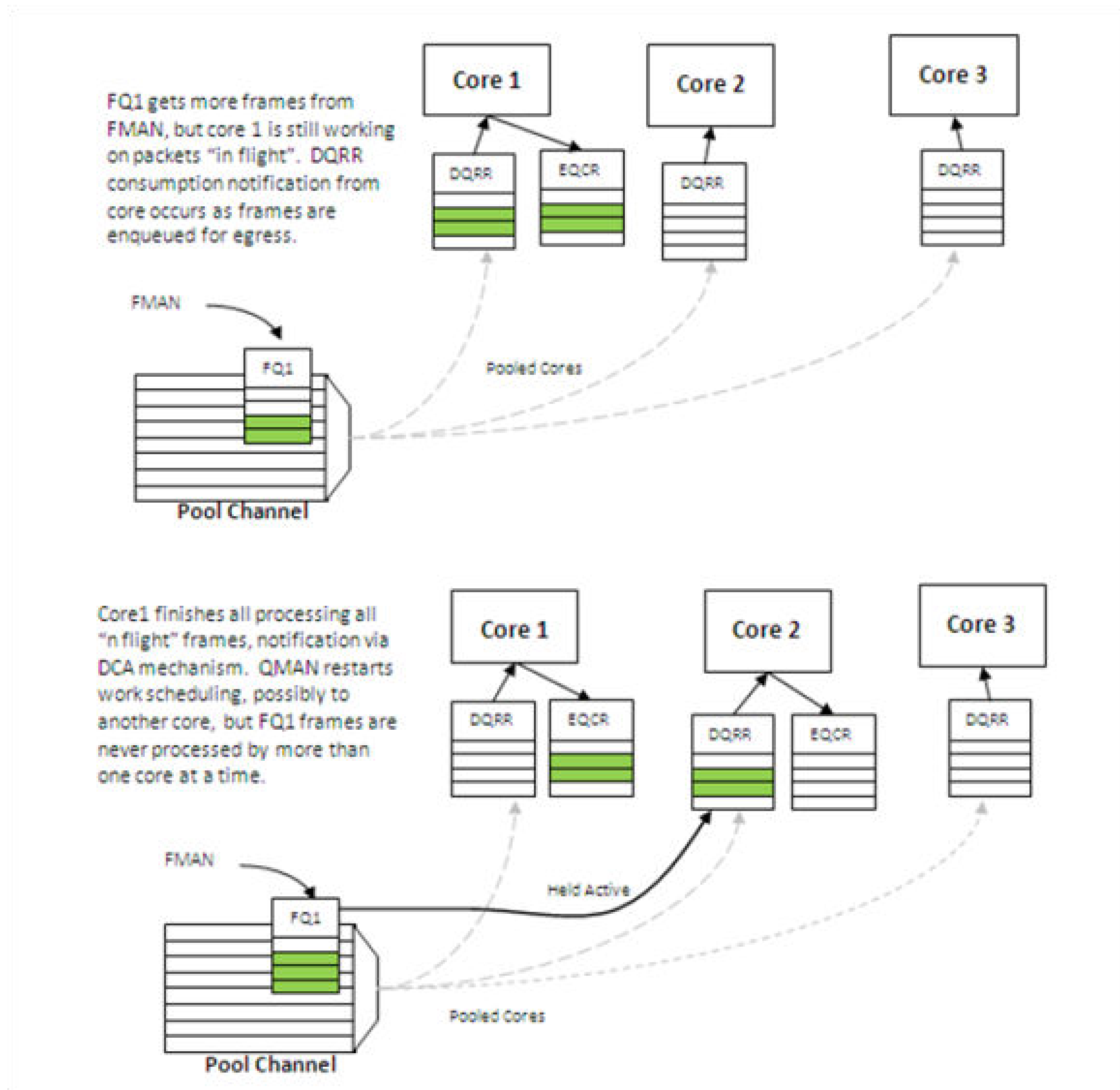


Figure 22: Held suspended to hold active mode




## Congestion management

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

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

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

attempting to manage the number of buffers in a buffer pool as they are used by a particular core or group of cores. The byte count is useful to manage the amount of system memory used by a particular core or group of cores.

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

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

## 7.9 Application Mapping

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

### Processor core assignment

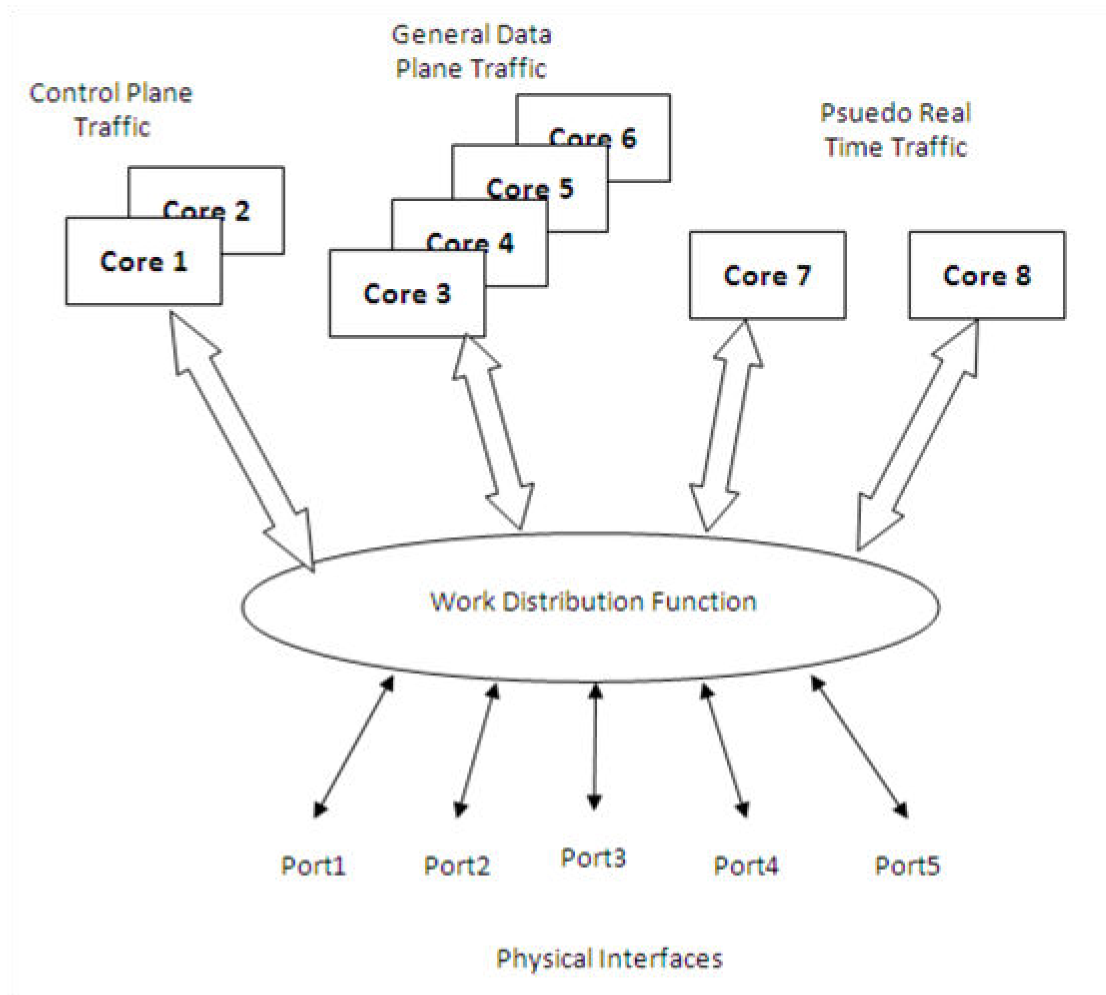
Consider a typical networking application with a set of distinct control and data plane functionality. Assigning two cores to perform control plane tasks and six cores to perform data plane tasks may be a reasonable partition in an eight-core device. When initially identifying the SoC required for the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

### Define flows

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

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

Figure 23: Example Application with Three Classes




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

Table 14: 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).<br>• May occur on ports 1, 2 or 3.<br>• Ingress control plane traffic on port three is higher priority than the other ports.<br>• Any ICMP packet on ports 1, 2 or 3 is considered control plane traffic.<br>• Control plane traffic makes up a small portion of the overall port bandwidth. |
| 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.<br>• The function performed is done on flows and egress packets must match the order of ingress packets.<br>• A flow is identified by the IP source address.<br>• The system can expect up to 50 flows at a time.<br>• All flows have the same priority and a lower priority than any control plane traffic.<br>• 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.<br>• 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.<br>• All valid ingress traffic on port 4 is to be processed by core 7, ingress traffic on port 5 processed by core 8.<br>• 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.<br>• The traffic on these flows are the highest priority. |

### Identify ingress and egress frame queues (FQs)

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

The general data plane ingress traffic is more complicated. Multiple options exist which maintain the required ordering for this traffic. While this traffic would certainly benefit from some of the control features of the QMan (cache warming, and so on), it is best to have one FQ per flow. Per the example, the flow is identified by the IP

source (32-bits), which consists of too many bits to directly use as the FQID. The hash algorithm can be used to reduce the 32-bits to a smaller number; in this case, six bits would generate 64 queues, which is more than the anticipated maximum flows at a given time. However, this is not significantly more than maximum flow expected, so more FQs can be defined to reduce hash collisions. Note that, in this case, a hash collision implies that two flows are assigned to the same FQ. As the ingress FQs fill directly from the port, the packet order is still maintained when there is a collision (two flows into one FQ). However, having two flows in the same FQ tends to minimize the impact of cache warming. There may be other possibilities to refine the definition of flows to ensure a one-to-one mapping of flows to FQs (for example, concatenating other fields in the frame) but for this example assume that an 8 bit hash (256 FQs) minimizes the likelihood of two flows in the FQ to an acceptable level.

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

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

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

#### Define PCD configuration for ingress FQs

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

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

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

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

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

## 7.10 FQ/WQ/Channel

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

Consider the following by class of traffic:

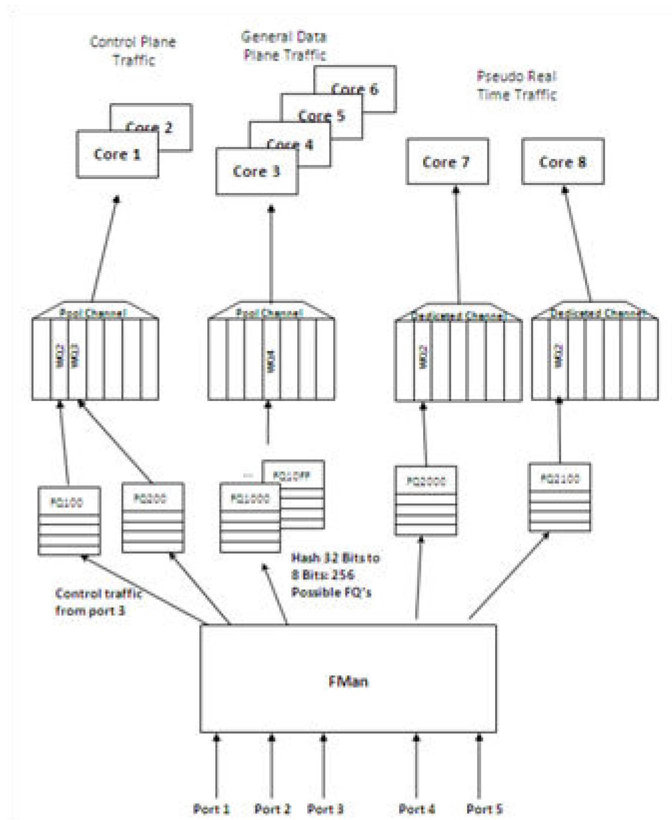
- The control traffic goes to a pool of two cores with priority given to traffic on port 3.
- The general data plane traffic goes to a pool of 4 cores.
- The pseudo real-time traffic goes to two separate cores as a dedicated channel.

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

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

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

Figure 24: Ingress application map




## Define egress FQ/WQ/channel configuration

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

For the example, the egress configuration is as follows:

- For control plane traffic, there needs to be separate queues for each port this traffic may use. These FQs must be assigned to a WQ that is higher in priority 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.

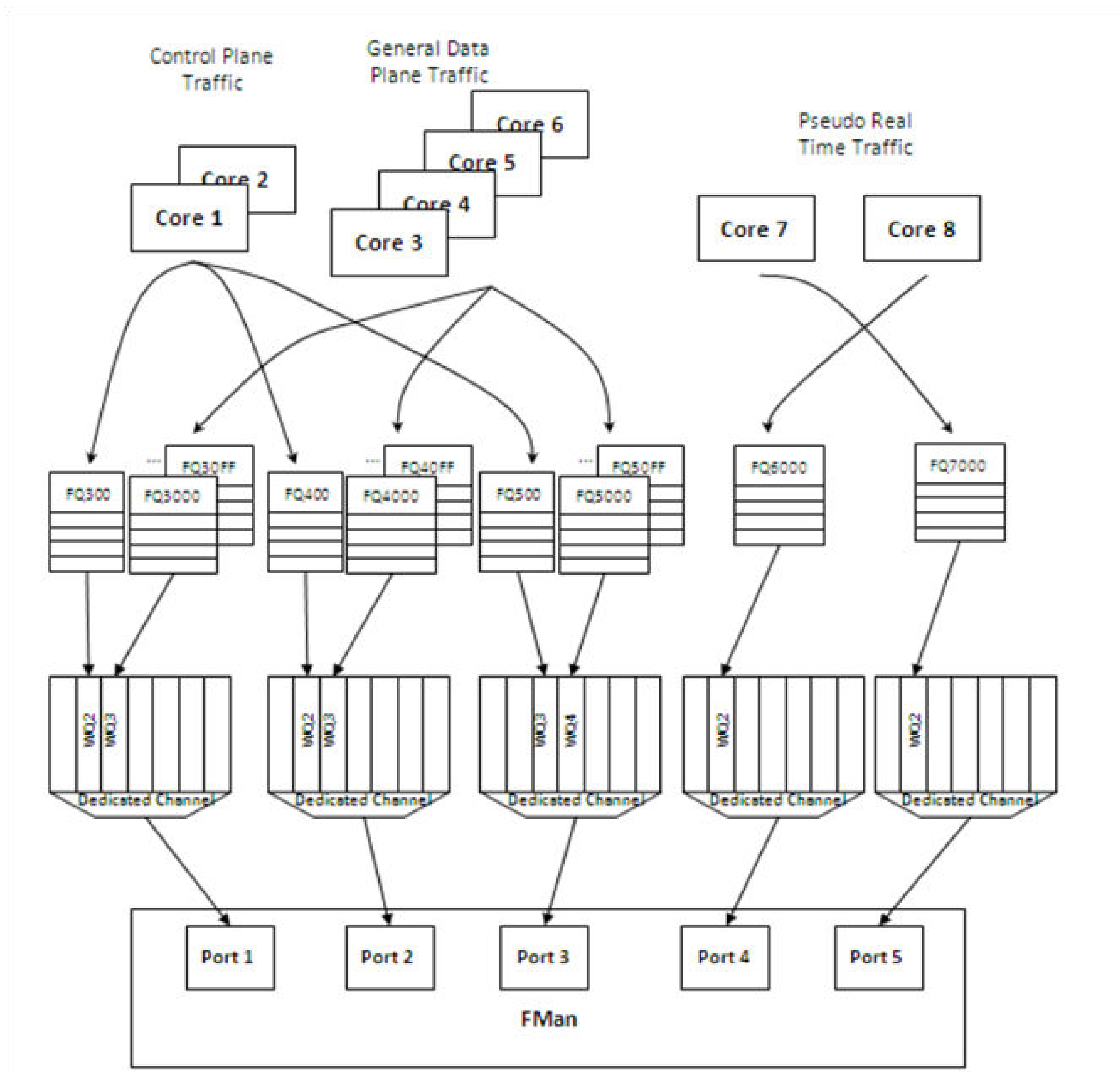
As stated before, the FQIDs can be whatever the user desires and should be selected to help keep track of what type of traffic the FQ's are associated. For this example:

- 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 25: Egress application map



## End of Document

DSPI Device Driver User Manual

# Chapter 8
# DSPI Device Driver User Manual

## 8.1 DSPI Device Driver User Manual

### Specifications

Linux-3.12.0.x + U-Boot-2014.07

### QDS X-nor card hardware configuration

```
SW1[1:4]=1000
SW4[1:4]=1111
SW3[1:4]=0001
```

J3 : short 1-2, J4 : short 1-2. This configuration can make DSPI device and QSPI device work at same time in X-nor card.

### Kernel Configure Tree View Options

```
Device Drivers --->
    [*] SPI support --->
        <*> Freescale DSPI controller
```

```
Device Drivers --->
     <*> Memory Technology Device (MTD) support --->
          Self-contained MTD device drivers --->
               <*> Support for AT45xxx DataFlash
```

### Compile-time Configuration Options

| Config | Values | Default Value | Description |
|---|---|---|---|
| `CONFIG_SPI_FSL_DSPI` | `y/n` | `y` | Enable DSPI module |
| `CONFIG_MTD_DATAFLASH` | `y/n` | `y` | Enables MTD devices for DSPI flash |

### Verification

#### Verification in U-Boot:

```
=> sf probe 1:0
SF: Detected AT45DB021D with page size 256 Bytes, erase size 2 KiB, total 256 KiB
```

DSPI Device Driver User Manual

```
=> sf erase 0 10000
SF: 65536 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

......
mtd_dataflash spi0.0: at45db021d (256 KBytes) pagesize 256 bytes (OTP)
6Freescale DSPI master initialized
......

Erase the DSPI flash

~ # mtd_debug erase /dev/mtd0 0x1100000 65536
Erased 65536 bytes from address 0x00000000 in flash


Write the DSPI 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 DSPI 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, the DSPI verification is passed.
```

eDMA User Manual

# Chapter 9
# eDMA User Manual

## 9.1 eDMA User Manual

### Description

The SoC integrates Freescale'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 --->`<br>`[*] DMA Engine support ---> --->`<br>`<*> 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>,
```

eDMA User Manual

```
                                <&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 15: 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???.???.
```

eDMA User Manual

```
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:~#
```

eDMA User Manual

# Chapter 10
# Enhanced Secured Digital Host Controller (eSDHC)

## 10.1 eSDHC Driver User Manual

### Description

The enhanced SD Host Controller(eSDHC) provides an interface between the host system and MMC/SD cards.

The eSDHC supports 1/4-bit data modes and bus clock frequency up to 50MHz

### Specifications

| Hardware: | eSDHC controller |
|---|---|
| Software: | Linux-3.0.x + u-boot-2011.12 |

### Module Loading

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

### U-boot Configuration

### Runtime options

| Env Variable | Env Description | Sub option | Option Description |
|---|---|---|---|
| hwconfig | Hardware configuration for u-boot | setenv hwconfig sdhc | Enable esdhc for the kernel |

### Kernel Configure Options

### Kernel Configure Tree View Options

| Kernel Configure Tree View Options | Description |
|---|---|
| Device Drivers ---><br><*> MMC/SD/SDIO card support ---><br><*> MMC block device driver<br>(8) Number of minors per block device<br>[*] Use bounce buffer for simple hosts | Enables SD/MMC block device driver support |
| *** MMC/SD/SDIO Host Controller Drivers ***<br><br><*> Secure Digital Host Controller Interface support<br><*> SDHCI platform and OF driver helper<br>[*] SDHCI OF support for the Freescale eSDHC controller | Enables freescale eSDHC driver support |

### Compile-time Configuration Options

| Option | Values | Default Value | Description |
|---|---|---|---|
| CONFIG_MMC | y/n | n | Enable SD/MMC bus protocol |
| CONFIG_MMC_BLOCK | y/n | y | Enable SD/MMC block device driver support |
| CONFIG_MMC_BLOCK_MINORS | integer | 8 | Number of minors per block device |
| CONFIG_MMC_BLOCK_BOUNCE | y/n | y | Enable continuous physical memory for transmit |
| CONFIG_MMC_SDHCI | y/n | y | Enable generic sdhc interface |
| CONFIG_MMC_SDHCI_PLTFM | y/n | y | Enable common helper function support for sdhci platform and OF drivers |
| CONFIG_MMC_SDHCI_OF_ESDHC | y/n | y | Enable Freescale eSDHC support |

### Device Tree Binding

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

Default node:

```
pq3-esdhc-0.dtsi:
sdhci@2e000 {
        compatible = "fsl,esdhc";
        reg = <0x2e000 0x1000>;
        interrupts = <72 0x2 0 0>;
        /* Filled in by U-Boot */
        clock-frequency = <0>;
};

For special platform (p1022ds as example):
/include/ "pq3-etsec2-1.dtsi"
        sdhc@2e000 {
                compatible = "fsl,p1022-esdhc", "fsl,esdhc";
                sdhci,auto-cmd12;
        };

NOTE: For different platform, the compatilbe can be difference.
      And the property "sdhci, auto-cmd12" is option.
```

### Source Files

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

| Source File | Description |
|---|---|
| drivers/mmc/host/sdhci.c | Linux SDHCI driver support |

*Table continues on the next page...*

*Table continued from the previous page...*

| Source File | Description |
|---|---|
| drivers/mmc/host/sdhci-pltfm.c | Linux SDHCI platform devices support driver |
| drivers/mmc/host/sdhci-of-esdhc.c | Linux eSDHC driver |

#### User Space Application

The following applications will be used during functional or performance testing. Please refer to the SDK UM document for the detailed build procedure.

| Command Name | Description | Package Name |
|---|---|---|
| iozone | IOzone is a filesystem benchmark tool. The benchmark generates and measures a variety of file operations.The benchmark tests file I/O performance for the following operations: Read, write, re-read, re-write, read backwards, read strided, fread, fwrite, random read, pread ,mmap, aio_read, aio_write. | iozone |

#### Verification in U-boot

The u-boot log:

```
=> mmcinfo
Device: FSL_ESDHC
Manufacturer ID: 3
OEM: 5344
Name: SD02G
Tran Speed: 25000000
Rd Block Len: 512
SD version 2.0
High Capacity: No
Capacity: 2032664576
Bus Width: 4-bit
=> mmc read 0 10000 0 1
MMC read: dev # 0, block # 0, count 1 ... 1 blocks read: OK
=> mmc part 0
Partition Map for MMC device 0  --   Partition Type: DOS
Partition     Start Sector     Num Sectors     Type


    1                  16         3970032       b
```

#### Verification in Linux

Add environment value

```
=> setenv hwconfig sdhc
```

The booting log

```
......
sdhci: Secure Digital Host Controller Interface driver
sdhci: Copyright(c) Pierre Ossman
mmc0: SDHCI controller on ffe2e000.sdhci-of [ffe2e000.sdhci-of] using PIO
......
mmc0: new SD card at address 87e2
mmcblk0: mmc0:87e2 SD02G 1.89 GiB
 mmcblk0: p1
```

Check the disk

```
~ # fdisk -l /dev/mmcblk0


disk /dev/mmcblk0: 2032 MB, 2032664576 bytes

63 heads, 62 sectors/track, 1016 cylinders

Units = cylinders of 3906 * 512 = 1999872 bytes

Device Boot      Start       End      Blocks  Id System

/dev/mmcblk0p1      1         1016     1984217   b Win95 FAT32   ~ #
```

Mount the file system and operate the card.

```
~ #

~ # mkdir /mnt/sd

~ # mount -t vfat /dev/mmcblk0p1 /mnt/sd

~ # ls /mnt/sd/

vim

~ # cp /bin/busybox /mnt/sd

~ # ls /mnt/sd

busybox  vim

~ # umount /mnt/sd

~ # mount -t vfat /dev/mmcblk0p1 /mnt/sd

~ # ls /mnt/sd

busybox  vim
```

```
~ #
```

## Benchmarking

```
~ #

~ # # iozone -Rab ./iosdresult/result -i 0 -i 1 -f test -n

512M -g 1G -r 64K

Iozone: Performance Test of File I/O

Version $Revision: 3.263 $

Compiled for 32 bit mode.

Build: linux-arm


Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins

Al Slater, Scott Rhine, Mike Wisner, Ken Goss

Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,

Randy Dunlap, Mark Montague, Dan Million,

Jean-Marc Zucconi, Jeff Blomberg,

Erik Habbinga, Kris Strecker, Walter Wong.


Run began: Wed Feb 16 20:33:04 2011


Excel chart generation enabled

Auto Mode

Using minimum file size of 524288 kilobytes.

Using maximum file size of 1048576 kilobytes.

Record Size 64 KB

Command line used: iozone -Rab ./iosdresult/result -i 0 -i 1 -f test -n 512M -g 1G -r
64K

Output is in Kbytes/sec

Time Resolution = 0.000005 seconds.

Processor cache size set to 1024 Kbytes.
```

```
Processor cache line size set to 32 bytes.

File stride size set to 17 * record size.

random random bkwd record stride

KB reclen write rewrite read reread read write read rewrite read fwrite frewrite fread
freread

524288 64 7040 7253 371022 372079

1048576 64 6537 6566 9857 10203
```

### Known Bugs, Limitations, or Technical Issues

1. P1/P2 platforms (except P2041) only support SDMA mode, to improve IO throughput, there is a way supported in SDK v1.2 can modify bounce buffer size (ADMA won’t use bounce buffer) at run-time:

# echo XXX > /sys/block/mmcblk0/bouncesz

generally XXX=262144 will reach the best performance, it will be improved by 30%.

2. Call trace when run "iozone" to test SDCARD performace on some platforms

workaround:increase the timeout value (in kernel configuration) and decrease the dirty_ratio in proc file system.

1) menuconfig:

Kernel hacking

(xxx) Default timeout for hung task detection (in seconds)

Note: the xxx may be 400 seconds or greater

2) modify 'proce file system':

echo xx > /proc/sys/vm/dirty_ratio

echo xx > /proc/sys/vm/dirty_background_ratio

Note: the xx may be 10 or 5, which meas 10% or 5%, the default is 20%.

3. The platform whose eMMC card is required to work on DDR mode needs a special rcw. (e.g. rcw_66_15_1800MHz_emmc_ddr.rcw for t2080qds)

Because of pin multiplexing with SPI, SPI would not work when eMMC card works on DDR mode

### Supporting Documentation

N/A

# Chapter 11 Frame Manager

## 11.1 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.

# Chapter 12
# Frame Manager Driver User's Guide


Non-disclosure Agreement Required.

# Chapter 13
# Frame Manager Configuration Tool User's Guide

IEEE 1588 Device Driver User Manual

# Chapter 14
# IEEE 1588 Device Driver User Manual

## 14.1 IEEE 1588 Device Driver User Manual

### Description

From IEEE-1588 perspective the components required are:

1. IEEE-1588 extensions to the gianfar driver or DPAA driver.
2. A stack application for IEEE-1588 protocol.

### Specifications

| Hardware: | Freescale Hardware Assist for IEEE1588 Compliant Timestamping |
|---|---|
| Software: | Freescale Linux SDK 1.4 for QorIQ Processors or above (Linux 3.8+) |

### Module Loading

IEEE 1588 device driver support either kernel built-in or module

### Kernel Configure Tree View Options

1. eTSEC - Using IXXAT stack

| Kernel Configure Tree View Options | Description |
|---|---|
| `Device Drivers  --->`<br>`  [*]Network device support  --->`<br>`     [*]Ethernet driver support --->`<br>`        <*> Gianfar Ethernet`<br>`        [*]       Gianfar 1588` | Enable 1588 driver for IXXAT stack |

2. eTSEC - Using PTPd stack

| Kernel Configure Tree View Options | Description |
|---|---|
| `Device Drivers  --->`<br>`  PTP clock support  --->`<br>`     <*> Freescale eTSEC as PTP clock` | Enable 1588 driver for PTPd stack |

3. DPAA - Using IXXAT stack

IEEE 1588 Device Driver User Manual

<table>
<tr><th>Kernel Configure Tree View Options</th><th>Description</th></tr>
<tr><td><pre>Device Drivers  --->
  [*]Network device support  --->
     [*]Ethernet driver support --->
        [*]Freescale devices --->
           <*>IEEE 1588-compliant timestamping
           Optimization choices for the DPAA Ethernet driver ---
>
               (X)Optimize for forwarding</pre></td><td>Enable IEEE 1588-compliant timestamping</td></tr>
</table>

4. DPAA - Using PTPd stack

<table>
<tr><th>Kernel Configure Tree View Options</th><th>Description</th></tr>
<tr><td><pre>Device Drivers  --->
  PTP clock support  --->
     <*> Freescale dTSEC as PTP clock</pre></td><td>Enable 1588 driver for PTPd stack</td></tr>
</table>

### Compile-time Configuration Options

For eTSEC (IXXAT)

| Option | Values | Default Value | Description |
|---|---|---|---|
| CONFIG_GIANFAR | y/n | y | Enable eTSEC driver support |
| CONFIG_FSL_GIANFAR_1588 | y/n | n | Enables 1588 driver support |

For eTSEC (PTPd)

| Option | Values | Default Value | Description |
|---|---|---|---|
| CONFIG_GIANFAR | y/n | y | Enable eTSEC driver support |
| CONFIG_PTP_1588_CLOCK_GIANFAR | y/n | y | Enables 1588 driver support |

For DPAA (IXXAT)

| Option | Values | Default Value | Description |
|---|---|---|---|
| CONFIG_FSL_DPAA_1588 | y/n | n | Enable IEEE 1588 support |
| CONFIG_FSL_DPAA_ETH | y/n | y | Enables DPAA driver support |

For DPAA (PTPd)

IEEE 1588 Device Driver User Manual

| Option | Values | Default Value | Description |
|---|---|---|---|
| CONFIG_PTP_1588_CLOCK_DPAA | y/n | n | Enable IEEE 1588 support |
| CONFIG_FSL_DPAA_ETH | y/n | y | Enables DPAA driver support |

### Source Files

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

1. eTSEC (for IXXAT)

| Source File | Description |
|---|---|
| drivers/net/ethernet/freescale/gianfar.c | IEEE 1588 hooks in the Ethernet driver |
| drivers/net/ethernet/freescale/gianfar_1588.c | IEEE 1588 driver |

2. eTSEC (for PTPd)

| Source File | Description |
|---|---|
| drivers/net/ethernet/freescale/gianfar.c | IEEE 1588 hooks in the Ethernet driver |
| drivers/net/ethernet/freescale/gianfar_ptp.c | IEEE 1588 driver |

3. DPAA (for IXXAT)

| Source File | Description |
|---|---|
| drivers/net/ethernet/freescale/dpa/dpaa_1588.c | IEEE 1588 driver support |
| drivers/net/ethernet/freescale/dpa/dpaa_1588.h | IEEE 1588 driver head file |
| drivers/net/ethernet/freescale/dpa/dpaa_eth.c | DPAA Ethdriver support |

4. DPAA (for PTPd)

| Source File | Description |
|---|---|
| drivers/net/ethernet/freescale/dpa/dpaa_ptp.c | IEEE 1588 driver support |
| drivers/net/ethernet/freescale/dpa/dpaa_eth.c | DPAA Ethdriver support |

### Device Tree Binding

1. eTSEC (for IXXAT)

| Property | Type | Status | Description |
|---|---|---|---|
| compatible | String | Required | Should be 'fsl,gianfar-ptp-timer' for IEEE 1588 driver |

```
Default node:
        ptp_timer: ptimer@24e00 {
               compatible = "fsl,gianfar-ptp-timer";
```

IEEE 1588 Device Driver User Manual

```
                reg = <0x24e00 0xb0>;
                fsl,ts-to-buffer;
                fsl,tmr-prsc = <0x2>;
                fsl,clock-source-select = <1>;
        };

        enet0: ethernet@24000 {
                ......
                ptimer-handle = <&ptp_timer>;
        };
```

2. eTSEC (for PTPd)

| Property | Type | Status | Description |
|---|---|---|---|
| compatible | String | Required | Should be 'fsl,etsec-ptp' for IEEE 1588 driver |

```
Default node:
         ptp_clock@b0e00 {
               compatible = "fsl,etsec-ptp";
               reg = <0xb0e00 0xb0>;
               interrupts = <68 2 0 0 69 2 0 0>;
               fsl,tclk-period = <10>;
               fsl,tmr-prsc    = <2>;
               fsl,tmr-add     = <0x80000016>;
               fsl,tmr-fiper1  = <0x3b9ac9f6>;
               fsl,tmr-fiper2  = <0x00018696>;
               fsl,max-adj     = <199999999>;
         };
```

3. DPAA

| Property | Type | Status | Description |
|---|---|---|---|
| compatible | String | Required | Should be 'fsl,fman-rtc' |
| reg | integer | Required | Register map |

```
Default node:
        fman0: fman@400000 {
                #address-cells = <1>;
                #size-cells = <1>;
                cell-index = <0>;
                compatible = "fsl,p4080-fman", "fsl,fman", "simple-bus";

                enet0: ethernet@e0000 {
                        cell-index = <0>;
                        compatible = "fsl,p4080-fman-1g-mac", "fsl,fman-1g-mac";
                        reg = <0xe0000 0x1000>;
                        fsl,port-handles = <&fman0_rx0 &fman0_tx0>;
                        tbi-handle = <&tbi0>;
                        phy-handle = <&phy0>;
                        phy-connection-type = "sgmii";
                        ptimer-handle = <&ptp_timer0>;
                };
```

IEEE 1588 Device Driver User Manual

```
		...enet1/2...

		enet3: ethernet@e6000 {
			cell-index = <3>;
			compatible = "fsl,p4080-fman-1g-mac", "fsl,fman-1g-mac";
			reg = <0xe6000 0x1000>;
			fsl,port-handles = <&fman0_rx3 &fman0_tx3>;
			tbi-handle = <&tbi3>;
			phy-handle = <&phy3>;
			phy-connection-type = "sgmii";
			ptimer-handle = <&ptp_timer0>;
		};

		ptp_timer0: rtc@fe000 {
			compatible = "fsl,fman-rtc";
			reg = <0xfe000 0x1000>;
		};
	};


	fman1: fman@500000 {
		#address-cells = <1>;
		#size-cells = <1>;
		cell-index = <1>;
		compatible = "fsl,p4080-fman", "fsl,fman", "simple-bus";

		enet5: ethernet@e0000 {
			cell-index = <0>;
			compatible = "fsl,p4080-fman-1g-mac", "fsl,fman-1g-mac";
			reg = <0xe0000 0x1000>;
			fsl,port-handles = <&fman1_rx0 &fman1_tx0>;
			tbi-handle = <&tbi5>;
			phy-handle = <&phy5>;
			phy-connection-type = "sgmii";
			ptimer-handle = <&ptp_timer1>;
		};

		...enet6/7...

		enet8: ethernet@e6000 {
			cell-index = <3>;
			compatible = "fsl,p4080-fman-1g-mac", "fsl,fman-1g-mac";
			reg = <0xe6000 0x1000>;
			fsl,port-handles = <&fman1_rx3 &fman1_tx3>;
			tbi-handle = <&tbi8>;
			phy-handle = <&phy8>;
			phy-connection-type = "sgmii";
			ptimer-handle = <&ptp_timer1>;
		};

		ptp_timer1: rtc@fe000 {
			compatible = "fsl,fman-rtc";
			reg = <0xfe000 0x1000>;
		};
	};
```

IEEE 1588 Device Driver User Manual

### Verification in Linux and test procedure

Connect two boards through crossover, ex, eth1 to eth1, and connect the other ethernet port on each board to switch or PC. One board runs as master, and the other as slave.

- **Using the IXXAT stack image**
  1. Enable IEEE-1588 support in the gianfar/DPAA driver, rebuild and reload it. You should get the following message during system boot:

```
...
IEEE1588: ptp 1588 is initialized.
...
```

  - On the master side:

```
# ifconfig eth1 192.168.1.100 allmulti up

# ./ptp -i 0:eth1 -do        //run stack application
```

  **Note:** On a DPAA platform, use fm1-gb1 instead of eth1

  - On the slave side :

```
# ifconfig eth1 192.168.1.200 allmulti up

#./ptp -i 0:eth1 -do        //run stack application
```

  **Note:** On a DPAA platform, use the desired interface(e.g. fm1-gb1) instead of eth1

  2. You should start getting synchronization messages on the slave side.

NOTE

A detailed configuration of the IEEE-1588 stack application is described in a Quick Start Guide provided with the stack application.

- **Using the PTPd stack image**
  1. Enable IEEE-1588 support in the gianfar driver, rebuild and reload it. You should get the following message during system boot:

```
...
pps pps0: new PPS source ptp0
...
```

  - On the master side:

```
# ifconfig eth0 up

# ifconfig eth0 192.168.1.100

# ./ptpd -i eth0 -MV        //run stack application
```

  **Note:** On a DPAA platform, use the desired interface(e.g. fm1-gb1) instead of eth0.

  -

On the slave side :

```
# ifconfig eth0 up

# ifconfig eth0 192.168.1.200

# ./ptpd -i eth0 -sV --servo:kp=0.32 --servo:ki=0.05        //run stack
application
```

**Note:** On a DPAA platform, use the desired interface(e.g. fm1-gb1) instead of eth0.

2. You should start getting synchronization messages on the slave side.

#### Known Bugs, Limitations, or Technical Issues

- For DPAA, the PTPd stack limits to use only one ptp timer, so only the interfaces on the second FMAN(such as fm2-gb2) are available for PTPd if the platform has two FMANs.
- The IEEE 1588 conflicts with SGMII 10/100M mode.1588 TimeStamp is supported in all Gbps modes(both RGMII and SGMII) and all full-duplex 10/100 modes except 10/100 SGMII mode.
- For eTSEC, the slave PPS signal (available on TSEC_1588_PULSE_OUT1 pin) is not phase aligned with master PPS signal. This is a known limitation.
- For eTSEC, running IEEE-1588 function need disable CONFIG_RX_TX_BUFF_XCHG. In default SDK kernel image, the ASF was enabled, so CONFIG_RX_TX_BUFF_XCHG was enabled too. you should disable CONFIG_RX_TX_BUFF_XCHG and rebuild kernel for 1588 function.
- VLAN feature was disable in default kernel configuration. So, if 1588 over VLAN is used in your application or solution, you need to enable VLAN in Linux kernel by yourself.
- For DPAA, the PTPd stack limits to use only one ptp timer, so only the interfaces on the second FMAN(such as fm2-gb2) are available for PTPd if the platform has two FMAN.

#### Supporting Documentation

N/A.

## 14.2 IEEE 1588-2008 Quick Start

This guide describes the setup of two P1025TWR evaluation boards to evaluate how the IXXAT stack runs on the P1025-TWR processor.

For more information see the *IEEE 1588-2008 Quick Start* located in the following SDK directory: sdk_documentation/pdf/ixxat_ieee1588_2008_qs.pdf

# Chapter 15
# IFC NOR Flash User Manual

## 15.1 Integrated Flash Controller NOR Flash User Manual

### Description

Freescale'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.

### Dependencies

| | |
|---|---|
| Hardware: | Platform must have NOR Flash connected using IFC |
| Software: | U-boot-2011+<br>Linux 2.6.35+ [1] |

### 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<br>CONFIG_SYS_FLASH_CFI<br>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

[1] For Linux and U-Boot version numbers see the Components section of the SDK Overiew.

<table>
<tr><th>Kernel Configure Tree View Options</th><th>Description</th></tr>
<tr><td><pre>
Device Drivers  --->
   <*> Memory Technology Device (MTD) support  --->
            [*]   MTD partitioning support
            [*]     Command line partition table parsing
            <*>     Flash partition map based on OF description
            <*>   Direct char device access to MTD devices
            -*-   Common interface to block layer for MTD
'translation layers'
            <*>   Caching block device access to MTD devices
            <*>   FTL (Flash Translation Layer) support
            RAM/ROM/Flash chip drivers  --->
                    <*> Detect flash chips by Common Flash
Interface (CFI) probe
                    <*> Support for Intel/Sharp flash chips
                    <*> Support for AMD/Fujitsu/Spansion flash
chips
            Mapping drivers for chip access  --->
                     <*> Flash device in physical memory map based
on OF description
</pre></td><td>These options enable CFI support for NOR Flash under MTD subsystem and Integrated Flash Controller support on Linux</td></tr>
<tr><td><pre>
File systems  --->
         [*] Miscellaneous filesystems  --->
                    <*>   Journalling Flash File System v2 (JFFS2)
support
</pre></td><td>This option enables JFFS2 file system support for MTD Devices</td></tr>
</table>

### Identifier

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

Special Configure needs to be enabled("Y") for LS1021 and LS1043. Please find in below table with default value as "N"

| Option | Values | Default Value | Description |
|---|---|---|---|
| CONFIG_FSL_IFC | Y/N | Y | Integrated Flash Controller support |
| CONFIG_MTD | Y/N | Y | Memory Technology Device (MTD) support |
| CONFIG_MTD_PARTITIONS | Y/N | Y | MTD partitioning support |
| CONFIG_MTD_CMDLINE_PARTS | Y/N | Y | Allow generic configuration of the MTD partition tables via the kernel command line. |
| CONFIG_MTD_OF_PARTS | Y/N | Y | This provides a partition parsing function which derives the partition map from the children of the flash nodes described in Documentation/powerpc/booting-without-of.txt |
| CONFIG_MTD_CHAR | Y/N | Y | Direct char device access to MTD devices |
| CONFIG_MTD_BLOCK | Y/N | Y | Caching block device access to MTD devices |
| CONFIG_MTD_CFI | Y/N | Y | Detect flash chips by Common Flash Interface (CFI) probe |
| CONFIG_MTD_GEN_PROBE | Y/N | Y | NA |
| CONFIG_MTD_MAP_BANK_WIDTH_1 | Y/N | Y | Support 8-bit buswidth |
| CONFIG_MTD_MAP_BANK_WIDTH_2 | Y/N | Y | Support 16-bit buswidth |
| CONFIG_MTD_MAP_BANK_WIDTH_4 | Y/N | Y | Support 32-bit buswidth |
| CONFIG_MTD_PHYSMAP_OF | Y/N | Y | Flash device in physical memory map based on OF description |
| CONFIG_FTL | Y/N | Y | 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 |
| CONFIG_MTD_CFI_ADV_OPTIONS | Y/N | N | Enable only for LS1021 and LS1043 |
| CONFIG_MTD_CFI_BE_BYTE_SWAP | Y/N | N | Enable only for LS1021 and LS1043 |

### Device Tree Binding

Documentation/devicetree/bindings/powerpc/fsl/ifc.txt

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.


Non-disclosure Agreement Required.

| Source File | Description |
|---|---|
| drivers/memory/fsl_ifc.c | Integrated Flash Controller driver to handle error interrupts |
| drivers/mtd/mtdpart.c | Simple MTD partitioning layer |
| drivers/mtd/mtdblock.c | Direct MTD block device access |
| drivers/mtd/mtdchar.c | Character-device access to raw MTD devices. |
| drivers/mtd/ofpart.c | Flash partitions described by the OF (or flattened) device tree |
| drivers/mtd/ftl.c | FTL (Flash Translation Layer) support |
| drivers/mtd/chips/cfi_probe.c | Common Flash Interface probe |
| drivers/mtd/chips/cfi_util.c | Common Flash Interface support |
| drivers/mtd/chips/cfi_cmdset_0001.c | Support for Intel/Sharp flash chips |
| drivers/mtd/chips/cfi_cmdset_0002.c | Support for AMD/Fujitsu/Spansion flash chips |

### Verification in U-boot

### Test the Read/Write/Erase functionality of NOR Flash

1. Boot the u-boot with above config options to get NOR Flash access enabled. Check this in boot log,

   ```
   FLASH: * MiB
   ```

   where * is the size of NOR Flash
2. Erase NOR Flash
3. Make test pattern on memory e.g. DDR
4. Write test pattern on NOR Flash
5. Read the test pattern from NOR Flash to memory e.g DDR
6. Compare the test pattern data to verify functionality.

Test Log :

Test log with initial u-boot log removed

```
--
--

FLASH: 32 MiB
L2:    256 KB enabled
--
--
/* u-boot prompt */
=> mw.b 1000000 0xa5 10000
=> md 1000000
01000000: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000010: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000020: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000030: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000040: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000050: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000060: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000070: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
```

```
01000080: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
01000090: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
010000a0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
010000b0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
010000c0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
010000d0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
010000e0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
010000f0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5    ................
=> protect off all
Un-Protect Flash Bank # 1
=> erase ee000000 ee01ffff

. done
Erased 1 sectors
=> cp.b 1000000 ee000000 10000
Copy to Flash... 9....8....7....6....5....4....3....2....1....done
=> cmp.b 1000000 ee000000 10000
Total of 65536 bytes were the same
=>
```

### Verification in Linux

To cross check whether IFC NOR driver has been configured in the kernel or not, see the kernel boot log with following entries. Please note mtd partition number can be changed depending upon device tree.

```

ee000000.nor: Found 1 x16 devices at 0x0 in 16-bit bank

Amd/Fujitsu Extended Query Table at 0x0040

number of CFI chips: 1

RedBoot partition parsing not available

Creating 4 MTD partitions on "ee000000.nor":

0x000000040000-0x000000080000 : "NOR DTB Image"

ata1: Signature Update detected @ 0 msecs

0x000000080000-0x000000780000 : "NOR Linux Kernel Image"

0x000000800000-0x000001c00000 : "NOR JFFS2 Root File System"

0x000001f00000-0x000002000000 : "NOR U-Boot Image"
```

To verify NOR flash device accesses see the following test,

```

[root@ root]# cat /proc/mtd

dev:    size   erasesize  name

mtd0: 00040000 00020000 "NOR DTB Image"

mtd1: 00700000 00020000 "NOR Linux Kernel Image"

mtd2: 01400000 00020000 "NOR JFFS2 Root File System"
```

```
mtd3: 00100000 00020000 "NOR U-Boot Image"

mtd4: 00100000 00004000 "NAND U-Boot Image"

mtd5: 00100000 00004000 "NAND DTB Image"

mtd6: 00400000 00004000 "NAND Linux Kernel Image"

mtd7: 00400000 00004000 "NAND Compressed RFS Image"

mtd8: 00f00000 00004000 "NAND JFFS2 Root File System"

mtd9: 00700000 00004000 "NAND User area"

mtd10: 00080000 00010000 "SPI (RO) U-Boot Image"

mtd11: 00080000 00010000 "SPI (RO) DTB Image"

mtd12: 00400000 00010000 "SPI (RO) Linux Kernel Image"

mtd13: 00400000 00010000 "SPI (RO) Compressed RFS Image"

mtd14: 00700000 00010000 "SPI (RW) JFFS2 RFS"

[root@ root]# flash_eraseall -j /dev/mtd2

Erasing 128 Kibyte @ 1400000 -- 100% complete. Cleanmarker written at 13e0000.

[root@P1010RDB root]# mount -t jffs2 /dev/mtdblock2 /mnt/

JFFS2 notice: (1202) jffs2_build_xattr_subsystem: complete building xattr subsystem, 0
of xdatum (0 unchecked, 0 orphan) and 0 of xref (0 dead, 0 orphan) found.

[root@ root]# cd /mnt/

[root@ mnt]# ls -l

[root@ mnt]# touch flash_file

[root@ root]# umount mnt
//ls must list local_file
[root@ root]# ls mnt
//mount again
[root@ root]# mount -t  jffs2 /dev/mtdblock2 /mnt/
JFFS2 notice: (1219) jffs2_build_xattr_subsystem: complete building xattr subsystem, 0
of xdatum (0 unchecked, 0 orphan) and 0 of xref (0 dead, 0 orphan) found.
//use ls ; it must show the created file
[root@ root]# ls /mnt/
flash_file
//unmount
[root@ root]# umount /mnt/
```

## Known Bugs, Limitations, or Technical Issues

N/A

## Supporting Documentation

N/A

# Chapter 16
# IFC NAND Flash User Manual

## 16.1 Integrated Flash Controller NAND Flash User Manual

### Description

Freescale'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.

### Dependencies

| Hardware: | Platform must have NAND Flash connected using IFC |
|---|---|
| Software: | U-boot-2011+<br>Linux 2.6.35+ [2] |

### 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 |

[2] For Linux and U-Boot version numbers see the Components section of the SDK Overview

## Kernel Configure Options

### Tree View

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

<table>
<tr><th>Kernel Configure Tree View Options</th><th>Description</th></tr>
<tr><td><pre>
Device Drivers  --->

     <*> Memory Technology Device (MTD) support  --->

     [*]  MTD partitioning support

     [*]    Command line partition table parsing=

     <*>     Flash partition map based on OF description

     <*>   Direct char device access to MTD devices

     -*-  Common interface to block layer for MTD 'translation
layers'

     <*>   Caching block device access to MTD devices

     <*>   NAND Device Support --->


        <*>   NAND support for Freescale IFC controller
Enable UBIFS filesystem in linux configuration
Device Drivers  --->
        <*> Memory Technology Device (MTD) support  --->

       UBI - Unsorted block images  --->


  <*> Enable UBI

  (4096) UBI wear-leveling threshold

  (1)   Percentage of reserved eraseblocks for bad eraseblocks
</pre></td><td>These options enable Integrated Flash Controller NAND support to work with MTD subsystem available on Linux.<br><br>Also UBIFS support needs to be enabled.</td></tr>
</table>

<table>
<tr><th>Kernel Configure Tree View Options</th><th>Description</th></tr>
<tr><td><pre>
handling

  < >   MTD devices emulation driver (gluebi)

          *** UBI debugging options ***

  [ ]   UBI debugging
File systems  --->
        [*] Miscellaneous filesystems  --->

       <*>   UBIFS file system support

       [*]     Extended attributes support
               [ ]     Advanced compression options
               [ ]     Enable debugging
</pre></td><td></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_FSL_IFC | y/n | y | Enable Integrated Flash Controller support |
| CONFIG_MTD_NAND_FSL_IFC | y/n | Y | Enable Integrated Flash Controller NAND Machine support |
| CONFIG_MTD_PARTITIONS | y/n | Y | MTD partitioning support |
| CONFIG_MTD_CMDLINE_PARTS | y/n | Y | Allow generic configuration of the MTD partition tables via the kernel command line. |
| CONFIG_MTD_OF_PARTS | y/n | Y | This provides a partition parsing function which derives the partition map from the children of the flash nodes described in Documentation/ powerpc/booting-without-of.txt |
| CONFIG_MTD_CHAR | y/n | Y | Direct char device access to MTD devices |
| CONFIG_MTD_BLOCK | y/n | Y | Caching block device access to MTD devices |
| CONFIG_MTD_GEN_PROBE | y/n | Y | NA |
| CONFIG_MTD_PHYSMAP_OF | y/n | Y | Flash device in physical memory map based on OF description |

### Device Tree Binding

`Documentation/devicetree/bindings/memory-controllers/fsl/ifc.txt`

Flash partitions are specified by platform device tree.

### Source Files

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

| Source File | Description |
|---|---|
| drivers/memory/fsl_ifc.c | Integrated Flash Controller driver to handle error interrupts |
| drivers/mtd/nand/fsl_ifc_nand.c | Integrated Flash Controller NAND Machine driver |
| include/linux/fsl_ifc.h | IFC Memory Mapped Registers |

### Verification in U-boot

#### Test the Read/Write/Erase functionality of NAND Flash

1. Boot the u-boot with above config options to get NAND Flash driver enabled. Check this in boot log,

   `NAND: * MiB`

   `Where * is NAND flash size`
2. Erase NAND Flash
3. Make test pattern on memory e.g. DDR
4. Write test pattern on NAND Flash
5. Read the test pattern from NAND Flash to memory e.g DDR
6. Compare the test pattern data to verify functionality.

#### Test Log :

```
...
...

Flash: 32 MiB

L2:    256 KB enabled

NAND:  32 MiB

...
...

/* U-boot prompt */
=> nand erase.chip



NAND erase.chip: device 0 whole chip

Bad block table found at page 65504, version 0x01 Bad block table found at page 65472,
version 0x01
```

```
Skipping bad block at  0x01ff0000

Skipping bad block at  0x01ff4000

Skipping bad block at  0x01ff8000

Skipping bad block at  0x01ffc000


OK
=> mw.b 1000000 0xa5 100000
=> md 1000000
01000000: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
01000010: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
01000020: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
01000030: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
01000040: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
01000050: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
01000060: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
01000070: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
01000080: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
01000090: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
010000a0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
010000b0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
010000c0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
010000d0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
010000e0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
010000f0: a5a5a5a5 a5a5a5a5 a5a5a5a5 a5a5a5a5   ................
=> nand write 1000000 0 100000

NAND write: device 0 offset 0x0, size 0x100000
 1048576 bytes written: OK
```

```
=> nand read 2000000 0 100000


NAND read: device 0 offset 0x0, size 0x100000

 1048576 bytes read: OK

=>  cmp.b 1000000 2000000 100000

Total of 1048576 bytes were the same
```

### Verification in Linux

To cross check whether IFC NAND driver has been configured in the kernel or not, check the following. Please note mtd partition numbers can be changed depending upon board device tree

```
[root@(none) root]# cat /proc/mtd

dev:   size   erasesize  name

mtd0: 00100000 00004000 "NAND U-Boot Image"

mtd1: 00100000 00004000 "NAND DTB Image"

mtd2: 00400000 00004000 "NAND Linux Kernel Image"

mtd3: 00400000 00004000 "NAND Compressed RFS Image"

mtd4: 00f00000 00004000 "NAND Root File System"

mtd5: 00700000 00004000 "NAND User area"

mtd6: 00080000 00010000 "SPI (RO) U-Boot Image"

mtd7: 00080000 00010000 "SPI (RO) DTB Image"

mtd8: 00400000 00010000 "SPI (RO) Linux Kernel Image"

mtd9: 00400000 00010000 "SPI (RO) Compressed RFS Image"

mtd10: 00700000 00010000 "SPI (RW) JFFS2 RFS"


[root@(none) root]# flash_eraseall /dev/mtd4 Erasing 16 Kibyte @ f00000 -- 100%
complete.


[root@(none) root]# ubiattach /dev/ubi_ctrl -m 4

UBI: attaching mtd4 to ubi0

UBI: physical eraseblock size:   16384 bytes (16 KiB)

UBI: logical eraseblock size:    15360 bytes
```

```
UBI: smallest flash I/O unit:    512

UBI: VID header offset:          512 (aligned 512)

UBI: data offset:                1024

UBI: empty MTD device detected

UBI: create volume table (copy #1)

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.

## Supporting Documentation

N/A

# Chapter 17 KVM/QEMU

## 17.1 Freescale KVM/QEMU Release Notes

This document describes current limitations in the release of KVM and QEMU for Freescale SoCs.

This document describes current limitations in the release of KVM and QEMU for Freescale SoCs.

Copyright (C) 2015 Freescale Semiconductor, Inc.

Freescale KVM/QEMU Release Notes

07/21/2015

```
Overview
--------
This document describes new features, current limitations, and
known issues in KVM and QEMU for Freescale QorIQ SDK 1.8.

Features
------------

- Linux and QEMU versions:
   o  KVM is based on the Linux kernel 3.19
   o  QEMU is based on QEMU 2.3.0

- SMP guests
- virtio-net and virtio-blk with both MMIO and PCI as transport
- in-kernel GICv2 emulation for the guest


Limitations
-----------

The following items describe known limitations with this release
of KVM/QEMU:

  o VFIO-PCI is not supported

  o GICv3 emulation is not supported

  o vhost-net is not supported

  o minimal support for guest debug
     - breakpoints are not supported

  o no support for MSIs for virtio-pci
```

# Chapter 18
# Libvirt Users Guide

## 18.1 Introduction to libvirt

### 18.1.1 Overview

This document is a guide and tutorial to using libvirt on Freescale SoCs.

Libvirt is an open source toolkit that enables the management of Linux-based virtualization technologies such as KVM/QEMU virtual machines and Linux containers.

The goal of the libvirt project (see http://libvirt.org) is to provide a stable, standard, hypervisor-agnostic interface for managing virtualization "domains" such as virtual machines and containers. Domains can be remote and libvirt provides full security for managing remote domains over a network. Libvirt is a layer intended to be used as a building block for higher level management tools and applications.

Libvirt provides:

- An interface to remotely manage the lifecycle of virtualization domains-- provisioning, start/stop, monitoring
- Support for a variety of hypervisors-- KVM/QEMU and Linux Containers are supported in the Freescale 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

### 18.1.2 For Further Information

Libvirt is an open source project and a great deal of technical and usage information is available on the libvirt.org website:

- http://libvirt.org/index.html

Additional references:

- Architecture: http://libvirt.org/intro.html
- Deployment: http://libvirt.org/deployment.html
- XML Format: http://libvirt.org/format.html
- virsh command reference: http://linux.die.net/man/1/virsh
- The libvirt wiki has user generated content: http://wiki.libvirt.org/page/Main_Page

**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

## 18.1.3 Libvirt in the Freescale QorIQ SDK -- Supported Features

The libvirt packages provides a huge number of capabilities and features. This section describes the features tested in the Freescale QorIQ SDK release.

The SDK supports QEMU/KVM and LXC and thus supports URIs for QEMU and LXC:

- qemu:///
- lxc:///

The following virsh commands are supported:

- Domain Management
    - *attach-device* - Attach a device from an XML file. To use --config option, then it will effect after the acitive domain restarted.
    - *attach-disk* - attach disk device
    - *attach-interface* - attach network interface
    - *autostart* - configure a domain to be automatically started at boot
    - *blkdeviotune* - set or query a block device I/O tuning parameters.
    - *console* - connect to the console of a domain
    - *cpu-stats* - show domain cpu statistics (need mount /cgroup/cpuacct).
    - *create* - creates a transient domain from an XML file and starts it
    - *define* - define a new persistent domain from an XML file
    - *desc* - show or modify the description and title of a domain
    - *destroy* - For persistent domains, stops the domain. For transient domains, the domain is destroyed. This command does not gracefully stop the domain.
    - *detach-device* - detach a device from an XML file. To use --config option, then it will effect after the acitive domain restarted.
    - *detach-disk* - detach disk device
    - *detach-interface* - detach network interface
    - *domid* - convert a domain name or UUID to domain id.
    - *domif-setlink* - set link state of a virtual interface.
    - *domiftune* - get/set parameters of a virtual interface.
    - *domname* - convert a domain id or UUID to domain name.
    - *domuuid* - convert a domain name or id to domain UUID.
    - *domxml-from-native* - convert a QEMU command line to libvirt XML
    - *domxml-to-native* - convert a libvirt XML file (for QEMU) to a native QEMU command line. Useful for debugging.
    - *dumpxml* - output the XML for the specified domain
    - *edit* - edit XML configuration for a domain.
    - *maxvcpus* - show connection vcpu maximum (for QEMU)
    - *memtune* - get or set memory parameters.
    - *qemu-monitor-command* - for QEMU/KVM domains allows sending commands to the QEMU monitor

- *reset* - reset a domain
- *restore* - restore a domain from a saved state in a file
- *resume* - resume a suspended domain. After *resume* the domain is in the "running" state.
- *save* - save a domain state to a file
- *schedinfo* - show/set scheduler parameters (need mount cgroup CPU controller).
- *setmaxmem* - change maximum memory limit.
- *setmem* - change memory allocation.
- *start* - start a domain
- *suspend* - suspend a running domain. After *suspend* the domain is the "paused" state.
- *ttyconsole* - show tty console.
- *undefine* - remove a domain (undo the effects of *define*)
- *vcpucount* - show domain vcpu counts.
- *vcpuinfo* - show detailed domain vcpu information (for QEMU).
- *vcpupin* - control or query domain vcpu affinity (for QEMU).
- *emulatorpin* - control or query domain emulator affinity.

- Domain Monitoring
  - *domblkerror* - show errors on block devices (for QEMU).
  - *domblkinfo* - show domain block device size information.
  - *domblklist* - list all domain blocks.
  - *domblkstat* - get device block stats for a domain.
  - *domcontrol* - show domain control interface state.
  - *domif-getlink* - get link state of a virtual interface.
  - *domiflist* - list all domain virtual interfaces.
  - *domifstat* - get network interface stats for a domain.
  - *dominfo* - show domain information.
  - *dommemstat* - get memory statistics for a domain.
  - *domstate* - show domain state.
  - *list* - show the status of all domains
- Host and Hypervisor
  - *capabilities* - show capabilities.
  - *hostname* - print the hypervisor hostname.
  - *nodecpumap* - show node cpu map.
  - *nodecpustats* - print cpu stats of the node.
  - *nodeinfo* - show node information.
  - *nodememstats* - print memory stats of the node.
  - *sysinfo* - print the hypervisor sysinfo.
  - *uri* - print the hypervisor canonical URI.

- *version* - show version.
- Snapshot
  - *snapshot-create* - Create a snapshot from XML
  - *snapshot-create-as* - Create a snapshot from a set of args
  - *snapshot-current* - Get or set the current snapshot
  - *snapshot-delete* - Delete a domain snapshot
  - *snapshot-dumpxml* - Dump XML for a domain snapshot
  - *snapshot-edit* - edit XML for a snapshot
  - *snapshot-info* - snapshot information
  - *snapshot-list* - List snapshots for a domain
  - *snapshot-parent* - Get the name of the parent of a snapshot
  - *snapshot-revert* - Revert a domain to a snapshot
- Virsh itself
  - *cd* - change the current directory.
  - *connect* - (re)connect to hypervisor.
  - *echo* - echo arguments.
  - *exit* - quit this interactive terminal.
  - *help* - print help.
  - *pwd* - print the current directory.
  - *quit* - quit this interactive terminal.

Other virsh commands may operate correctly, but have not been specifically validated in the QorIQ SDK.

# 18.2 Build, Installation, and Configuration

## 18.2.1 Building Libvirt with Yocto

Libvirt is a Linux user space package that can easily be added to a root filesystem using the Yocto build system.

In the Freescale SDK, Libvirt and all pre-requisite user space packages are included when building the "virt" and "full" image type:

```
bitbake fsl-image-virt
bitbake fsl-image-full
```

Libvirt can be easily added to any rootfs image by updating the IMAGE_INSTALL_append variable in the **conf/local.conf** file in the Yocto build environment. For example, append the following line to **local.conf**:

```
IMAGE_INSTALL_append = " libvirt libvirt-libvirtd libvirt-virsh"
```

After libvirt is included in a root filesystem the libvirtd daemon will be automatically started by the system init scripts.

## 18.2.2 Running libvirtd

### Running libvirtd

The libvirtd daemon is installed as part of a libvirt packages installation. By default the target system init scripts should start libvirtd.

Running libvirtd on the target system is a pre-requisite to running any management tools such as virsh.

The libvirtd daemon can be manually started like this:

```
$ /etc/init.d/libvirtd start
```

In some circumstances the daemon may need to be restarted such as after mounting cgroups or hugetlbfs. Daemon restart can be done like this:

```
$ /etc/init.d/libvirtd restart
```

The libvirtd daemon can be configured in /etc/libvirt/libvirtd.conf. The file is self-documented and has detailed comments on the configuration options available.

### The libvird Daemon and Logging

The libvirt daemon logs data to /var/log/libvirt/

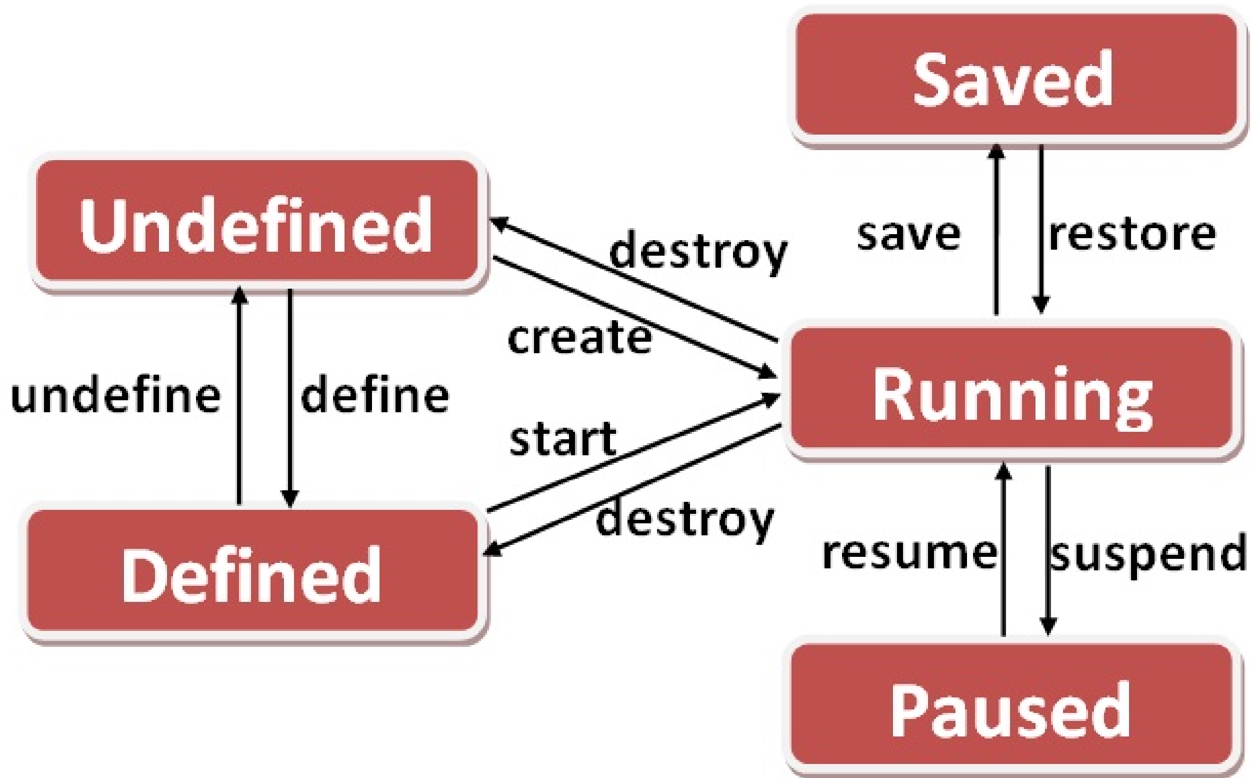
- General libvirtd log messages are in: */var/log/libvirt/libvirtd.log*
- QEMU/KVM domain logs are in: */var/log/libvirt/qemu/[domain-name].log*
- LXC domains logs are in: */var/log/libvirt/lxc/[domain-name].log*

The verbosity of logging can be controlled in */etc/libvirt/libvirtd.conf*.

## 18.2.3 Libvirt Domain Lifecycle

Two types of libvirt domains are supported in the QorIQ SDK-- KVM/QEMU virtual machines and Linux containers.

The following state diagram illustrates the lifecycle of a domain, the states that domains can be in and the **virsh** commands that move the domain between states.




Domain States

1. *Undefined*

There are two types of domains-- persistent and transient domains. All domains begin in the "undefined" state where they are defined in XML definition file, and libvirt is unaware of them.

2. *Defined*

Persistent domains begin with being "defined". This adds the domain to libvirt, but it is not running. This state can also be conceptually thought of as "stopped". The output of **virsh list --all** shows the domain as being "shut off".

3. *Running*

The "running" state is the normal state of an active domain after it has been started. The **start** command is used to move persistent domains into this state. Transient domains go from being undefined to "running" through the **create** command.

4. *Paused*

The domain execution has been suspended. The domain is unaware of being in this state.

5. *Saved*

The domain state has been saved and could be restored again.

See the Libvirt KVM/QEMU Example (Power Architecture) on page 181 article for an example of a KVM/QEMU domain lifecycle.

See the Basic Example on page 190 article for an example of a container domain lifecycle.

### 18.2.4 Libvirt URIs

Because libvirt supports managing multiple types of virtualization domains (possibly remote) it uses uniform resource identifiers (URIs) to describes the target "node" to manage and the type of domain being managed.

A URI is specified when tools such as **virsh** makes a connection to a target node running **libvirtd**.

Two types of URIs are supported in the QorIQ SDK-- QEMU/KVM and LXC.

QEMU/KVM URIs are in the form:

- For a local node: *qemu:///system*
- For a remote node: *qemu[+transport]://[hostname]/system*

For Linux containers:

- For a local node: *lxc:///*
- For a remote node: *lxc[+transport]://[hostname]/*

A default URI can be specified in the environment (LIBVIRT_DEFAULT_URI) or in the /etc/libvirt/libvirtd.conf config file.

For further information about URIs:

- http://libvirt.org/uri.html
- http://libvirt.org/remote.html#Remote_URI_reference

### 18.2.5 virsh

The virsh command is a command line tool provided with the libvirt package for managing libvirt domains. It can be used to create, start, pause, shutdown domains. The general command format is:

```
virsh [OPTION]... <command> <domain> [ARG]...
```

### 18.2.6 Libvirt xml

The libvirt XML format is defined at: http://libvirt.org/format.html.

## 18.3 Examples

### 18.3.1 KVM Examples

#### 18.3.1.1 Libvirt KVM/QEMU Example (Power Architecture)

The following example shows the lifecycle of a simple KVM/QEMU libvirt domain called *kvm1*.

In this example the default URI is qemu:///system, and because of this default an explicit URI is not used in the virsh commands.

1. We begin with a simple QEMU command line in a text file named kvm1.args:

```
$ cat kvm1.args
```

```
/usr/bin/qemu-system-ppc -m 256 -nographic -M ppce500 -kernel /boot/uImage -initrd /home/
root/my.rootfs.ext2.gz -append "root=/dev/ram rw console=ttyS0,115200" -serial pty -
enable-kvm -name kvm1
$
```

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 kvm1.args  > kvm1.xml
$
```

The content of the newly created domain XML file is shown below:

```
$ cat kvm1.xml
<domain type='kvm'>
  <name>kvm1</name>
  <uuid>f5b7cf86-41eb-eb78-4284-16501ff9f0e1</uuid>
  <memory unit='KiB'>262144</memory>
  <currentMemory unit='KiB'>262144</currentMemory>
  <vcpu placement='static'>1</vcpu>
  <os>
    <type arch='ppc' machine='ppce500'>hvm</type>
    <kernel>/boot/uImage</kernel>
    <initrd>/home/root/my.rootfs.ext2.gz</initrd>
    <cmdline>root=/dev/ram rw console=ttyS0,115200</cmdline>
  </os>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-ppc</emulator>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <memballoon model='virtio'/>
  </devices>
</domain>
```

3. Now the domain can be defined:

```
# virsh define kvm1.xml
Domain kvm1 defined from kvm1.xml

# virsh list --all
 Id    Name                           State
----------------------------------------------------
 -     kvm1                           shut off
```

4. Next start the domain. This starts the VM and boots the guest Linux.

```
# virsh start kvm1
Domain kvm1 started

# virsh list
 Id    Name                           State
----------------------------------------------------
 3     kvm1                           running
```

5. The virsh console command can be used to connect to the console of the running Linux domain.

```
# virsh console kvm1
Connected to domain kvm1
Escape character is ^]

Poky 9.0 (Yocto Project 1.4 Reference Distro) 1.4 model : qemu ppce500 ttyS0

model : qemu ppce500 login:
```

Press CTRL + ] to exit the console.

6. To stop the domain use the destroy command:

```
# virsh destroy kvm1
Domain kvm1 destroyed

root@p4080ds:~# virsh list --all
 Id    Name                           State
----------------------------------------------------
 -     kvm1                           shut off
```

7. To remove the domain from libvirt, use the undefine command:

```
# virsh undefine kvm1
Domain kvm1 has been undefined

root@p4080ds:~# virsh list --all
 Id    Name                           State
----------------------------------------------------
```

## 18.3.1.2 Libvirt KVM/QEMU -- Adding Devices Example (Power Architecture)

This example shows how devices (virtual network, USB, and PCI) can be added to a libvirt domain. The steps are identical to the simple domain example shown in Libvirt KVM/QEMU Example (Power Architecture) on page 181 , (except some additional preparation is needed for step #1):

1. Create a text file containing the QEMU command line
2. Use **virsh domxml-from-native** to create the libvirt XML
3. Use **virsh define** to define the QEMU domain
4. Use **virsh start** to start the domain

When defining the QEMU command line options in step #1 to add the network, USB, and PCI devices it is necessary to explicitly specify the PCI slot number for each device (all the devices in this example appear on the VM's virtual PCI bus). The remainder of this example explains how to do this.

The normal QEMU command line arguments to assign these devices to a virtual machine would look something like:

- for a virtual network interface

```
-netdev tap,id=tap0,script=/home/root/qemu-ifup,downscript=/home/root/qemu-
ifdown,vhost=on -device virtio-net-pci,netdev=tap0
```

- for the passthrough of USB device on bus #2, port #1

```
-device usb-ehci,id=ehci -device usb-host,bus=ehci.0,hostbus=2,hostport=1
```

- for the passthrough of PCI device 0000:01:00.0

```
-device vfio-pci,host=0000:01:00.0
```

(For further information on these command line argument, please refere to the KVM/QEMU documentation)

All these devices will appear on the virtual PCI bus in the virtual machine. However, in the context of libvirt it is necessary to explicitly assign the PCI slot for each device and therefore it is necessary to determine what slot are occupied and free.

First, start the virtual machine with libvirt **without the new devices** so that the default PCI slot assignment can be viewed (the example is for a domain called "kvm1" on a p4080ds):

```
$ echo '/usr/bin/qemu-system-ppc -m 512 -nographic -M ppce500 -kernel /home/root/uImage
-initrd /home/root/fsl-image-minimal-p4080ds.rootfs.ext2.gz  -append "root=/dev/ram rw
console=ttyS0,115200" -enable-kvm -smp 8 -mem-path /var/lib/hugetlbfs/pagesize-16MB -
name kvm1 -serial pty' > kvm1.args

$ virsh domxml-from-native qemu-argv kvm1.args  > kvm1.xml
$ virsh define kvm1.xml
$ virsh start kvm1
```

Next, view the devices on the PCI bus:

```
$ virsh qemu-monitor-command --hmp kvm1 'info pci'
  Bus  0, device   0, function 0:
    PCI bridge: PCI device 1957:0030
      BUS 0.
      secondary bus 0.
      subordinate bus 0.
      IO range [0x0000, 0x0fff]
      memory range [0x00000000, 0x000fffff]
      prefetchable memory range [0x00000000, 0x000fffff]
      BAR0: 32 bit memory at 0xc0000000 [0xc00fffff].
      id ""
  Bus  0, device   1, function 2:
    USB controller: PCI device 8086:7020
      IRQ 0.
      BAR4: I/O at 0xffffffffffffffff [0x001e].
      id "usb"
  Bus  0, device   3, function 0:
    Class 0255: PCI device 1af4:1002
      IRQ 0.
```

```
      BAR0: I/O at 0x1000 [0x101f].
      id "balloon0"
```

In this example it can be seen that the highest occupied slot is 0x3, which means that slots 0x4 and higher are available.

The explicit definition of the PCI slot numbers on the QEMU command line looks like this:

- for a virtual network interface (PCI slot/addr 0x4)

```
-netdev tap,id=tap0,script=/home/root/qemu-ifup,downscript=/home/root/qemu-
ifdown,vhost=on -device virtio-net-pci,netdev=tap0,bus=pci.0,addr=0x4
```

- for the passthrough of USB device on bus #2, port #1 (PCI slot/addr 0x5)

```
-device usb-ehci,id=ehci,bus=pci.0,addr=0x5 -device usb-host,bus=ehci.
0,hostbus=2,hostport=1
```

- for the passthrough of PCI device 0000:01:00.0 (PCI slot/addr 0x6)

```
-device vfio-pci,host=0000:01:00.0,bus=pci.0,addr=0x06
```

The virtual machine's PCI bus is "pci.0".

The complete command sequence to define the domain and start it looks like:

```
$ echo '/usr/bin/qemu-system-ppc -m 512 -nographic -M ppce500 -kernel /home/root/uImage
-initrd /home/root/fsl-image-minimal-p4080ds.rootfs.ext2.gz  -append "root=/dev/ram rw
console=ttyS0,115200" -enable-kvm -smp 8 -mem-path /var/lib/hugetlbfs/pagesize-16MB -
name kvm1 -serial pty -netdev tap,id=tap0,script=/home/root/qemu-ifup,downscript=/home/
root/qemu-ifdown,vhost=on -device virtio-net-pci,netdev=tap0,bus=pci.0,addr=0x4 -device
usb-ehci,id=ehci,bus=pci.0,addr=0x5 -device usb-host,bus=ehci.0,hostbus=2,hostport=1 -
device vfio-pci,host=0000:01:00.0,bus=pci.0,addr=0x06' > kvm1.args

$ virsh domxml-from-native qemu-argv kvm1.args  > kvm1.xml
$ virsh define kvm1.xml
$ virsh start kvm1
```

Libvirt also supports adding devices into a domain by modifying the domain XML file directly.

Shutdown the guest domain and edit the guest XML definition with the virsh edit command

```
$ virsh edit kvm1
```

- for a virtual network interface (PCI slot/addr 0x4)

```
<interface type="ethernet">
  <script path="/home/root/qemu-ifup"/>
  <model type="virtio"/>
  <driver name="vhost"/>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
</interface>
```

- for the passthrough of PCI device 0000:01:00.0 (PCI slot/addr 0x6)

```
<hostdev mode="subsystem" type="pci" managed="yes">
  <driver name="vfio"/>
  <source>
```

```
    <address domain="0x0000" bus="0x01" slot="0x06" function="0x0"/>
  </source>
</hostdev>
```

Start the domain again.

It is also possible to add and remove devices by defining the devicde in a standalone XML file. The **virsh attach-device/detach-device** commands commands can be used when the domain is not active.

For example,

```
$ cat vhost_on.xml
<interface type='ethernet'>
  <script path='/home/root/qemu-ifup'/>
  <model type='virtio'/>
  <driver name='vhost'/>
  <address type='pci' domain='0x0000' bus='0x00' slot='0x04' function='0x0'/>
</interface>
```

Attach device to the guest domain using **virsh attach-device.** The resulting XML file can be dumped to find the added device:

```
$ virsh attach-device --config kvm1 vhost_on.xml
$ virsh dumpxml kvm1
```

### 18.3.1.3 Libvirt KVM/QEMU Example (ARM Architecture)

The following example shows the lifecycle of a simple KVM/QEMU libvirt domain called **kvm**. In this example the default URI is qemu:///system, and because of this default an explicit URI is not used in the virsh commands

Libvirt has the possibility to convert qemu command line arguments in xml. However, the current version of libvirt does not have full support for all qemu arguments. It can be used to generate a basic xml file, but there is currently a problem that it generates a default USB node which should be removed.

1. We begin with a simple QEMU command line in a text file named **kvm.args**:

- 32 bit ARMv7:

```
echo "/usr/bin/qemu-system-arm -name kvm -smp 2 -enable-kvm -m 512  -nographic -cpu
host -machine type=virt -kernel /boot/zImage  -serial pty -initrd /boot/fsl-image-core-
ls1021atwr.ext2.gz -append 'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk' "
> kvm.args
```

- 64 bit ARMv8:

```
echo "/usr/bin/qemu-system-aarch64 -name kvm -smp 2 -enable-kvm -m 512 -nographic -cpu
host -machine type=virt -kernel /boot/Image  -serial pty -initrd /boot/
guest.rootfs.ext2.gz -append 'root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk'
" > kvm.args
```

**NOTE**

The serial console is a tty, not a telnet server. The -name option is required and specifies the name of the virtual machine.

2. Before defining the domain the QEMU command line must be converted to libvirt XML format:

```
virsh domxml-from-native qemu-argv kvm.args > kvm.xml
```

**NOTE**

For ARMv7 paltforms the generated xml file has to be manually changed to remove the USB node.

The content of the newly created domain XML file is shown below:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  <name>kvm</name>
  <uuid>12a3391e-9cd4-43dd-b8b7-27b1fb193378</uuid>
  <memory unit='KiB'>524288</memory>
  <currentMemory unit='KiB'>524288</currentMemory>
  <vcpu placement='static'>2</vcpu>
  <os>
    <type machine='virt'>hvm</type>
    <kernel>/boot/zImage</kernel>
    <initrd>/boot/fsl-image-core-ls1021atwr.ext2.gz</initrd>
    <cmdline>root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk</cmdline>
  </os>
  <cpu mode='custom' match='exact'>
    <model fallback='allow'>host</model>
  </cpu>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-arm</emulator>
    <controller type='usb' index='0'/>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
  </devices>
 </domain>
```

A more complex example including devices can be found bellow.

The example contains two devices:

- a virtio network interface
- a virtio block device (disk)

**NOTE**

This example will use MMIO as transport for virtio. Currently libvirt has no support for PCI transport, but it can be used using passthrough QEMU command line arguments (see the next example)

Device example (using virtio with MMIO as transport):

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  <name>kvm</name>
  <uuid>12a3391e-9cd4-43dd-b8b7-27b1fb193378</uuid>
```

```
  <memory unit='KiB'>524288</memory>
  <currentMemory unit='KiB'>524288</currentMemory>
  <vcpu placement='static'>2</vcpu>
  <os>
    <type machine='virt'>hvm</type>
    <kernel>/boot/zImage</kernel>
    <initrd>/boot/fsl-image-core-ls1021atwr.ext2.gz</initrd>
    <cmdline>root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk</cmdline>
  </os>
  <cpu mode='custom' match='exact'>
    <model fallback='allow'>host</model>
  </cpu>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-arm</emulator>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>

    <interface type='ethernet'>
      <mac address='52:54:00:b0:39:28'/>
      <script path='/home/root/qemu-ifup'/>
    </interface>
    <disk type='file' device='disk'>
      <driver name='qemu' type='raw' cache='none'/>
      <source file='/home/root/my_guest_disk'/>
      <target dev='vda' bus='virtio'/>
    </disk>

  </devices>
  <qemu:commandline>
    <qemu:arg value='-mem-path'/>
    <qemu:arg value='/var/lib/hugetlbfs/pagesize-2MB'/>
  </qemu:commandline>
</domain>
```

Device example (using virtio with PCI as transport):

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/qemu/1.0'>
  <name>kvm</name>
  <uuid>faa89ddc-e156-46c0-9d0f-029c322f9bf7</uuid>
  <memory unit='KiB'>1048576</memory>
  <currentMemory unit='KiB'>1048576</currentMemory>
  <vcpu placement='static'>4</vcpu>
  <os>
    <type arch='aarch64' machine='virt'>hvm</type>
    <kernel>/boot/Image</kernel>
    <initrd>/images/fsl-image-core-ls1043ardb.ext2.gz</initrd>
    <cmdline>root=/dev/ram0 rw console=ttyAMA0 rootwait earlyprintk</cmdline>
  </os>
  <cpu mode='custom' match='exact'>
    <model fallback='allow'>host</model>
```

```
  </cpu>
  <clock offset='utc'/>
  <on_poweroff>destroy</on_poweroff>
  <on_reboot>restart</on_reboot>
  <on_crash>destroy</on_crash>
  <devices>
    <emulator>/usr/bin/qemu-system-aarch64</emulator>
    <serial type='pty'>
      <target port='0'/>
    </serial>
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <memballoon model='none'/>
  </devices>
  <qemu:commandline>

    <qemu:arg value='-netdev'/>
    <qemu:arg value='tap,id=tap0,script=/home/root/qemu-ifup,downscript=no,ifname=tap0'/
>
    <qemu:arg value='-device'/>
    <qemu:arg value='virtio-net-pci,netdev=tap0'/>

  </qemu:commandline>
</domain>
```

3. Now the domain can be defined:

```
# virsh define kvm.xml
Domain kvm defined from kvm.xml

# virsh list --all
 Id    Name                           State
----------------------------------------------------
 -     kvm                            shut off
```

4. Next start the domain. This starts the VM and boots the guest Linux.

```
# virsh start kvm
Domain kvm started

# virsh list
 Id    Name                           State
----------------------------------------------------
 3     kvm                            running
```

5. The **virsh console** command can be used to connect to the console of the running Linux domain.

```
# virsh console kvm
Connected to domain kvm
Escape character is ^]

Poky (Yocto Project Reference Distro) 1.5 ls1021aqds /dev/ttyAMA0

ls1021aqds login: root
```

6. To stop the domain use the destroy command:

```
# virsh destroy kvm
Domain kvm destroyed

root@p4080ds:~# virsh list --all
 Id    Name                           State
----------------------------------------------------
 -     kvm                           shut off
```

7. To remove the domain from libvirt, use the undefine command:

```
# virsh undefine kvm
Domain kvm has been undefined

root@p4080ds:~# virsh list --all
 Id    Name                           State
----------------------------------------------------
```

## 18.3.2 Libvirt_lxc Examples

### 18.3.2.1 Basic Example

The following example shows the lifecycle of a simple LXC libvirt domain called *container1*. The virsh tool is used for managing the lxc domain lifecycle.

**1. Confirm the host Linux configuration.** Begin by confirming that the host kernel is configured correctly and that rootfs setup such as mounting cgroups has been done. This can be done with the lxc-checkconfig command.

```
# lxc-checkconfig
--- Namespaces ---
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
Pid namespace: enabled
User namespace: missing
Network namespace: enabled
Multiple /dev/pts instances: enabled

--- Control groups ---
Cgroup: enabled
Cgroup clone_children flag: enabled
Cgroup device: enabled
Cgroup sched: enabled
Cgroup cpu account: enabled
Cgroup memory controller: enabled
Cgroup cpuset: enabled

--- Misc ---
Veth pair device: enabled
Macvlan: enabled
Vlan: enabled
File capabilities: enabled
```

2. **Create a libvirt XML file defining the container.** The example below shows a very simple container defined in container1.xml that runs the command /bin/sh and has a console:

```
# cat container1.xml
<domain type='lxc'>
  <name>container1</name>
  <memory>500000</memory>
  <os>
    <type>exe</type>
    <init>/bin/sh</init>
  </os>
  <devices>
    <console type='pty'/>
  </devices>
</domain>
```

3. **Define the container.** The *virsh define* command processes the XML and makes creates the new libvirt domain.

```
# virsh -c lxc:/// define container1.xml
Domain container1 defined from container1.xml

# virsh -c lxc:/// list --all
 Id    Name                           State
----------------------------------------------------
 -     container1                     shut off
```

4. Start the container.

```
# virsh -c lxc:/// start container1
Domain container1 started

# virsh -c lxc:/// list
 Id    Name                           State
----------------------------------------------------
 3196  container1                     running
```

5. Connect to the console.

```
# virsh -c lxc:/// console container1
Connected to domain container1
Escape character is ^]
sh-4.2#
sh-4.2# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 18:25 pts/2    00:00:00 /bin/sh
root         3     1  0 18:36 pts/2    00:00:00 ps -ef

sh-4.2# ifconfig br0
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.171.73.123  netmask 255.255.254.0  broadcast 10.171.73.255
        inet6 fe80::a00:27ff:fe01:fe07  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:01:fe:07  txqueuelen 0  (Ethernet)
        RX packets 865838  bytes 104029354 (99.2 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 104446  bytes 43998714 (41.9 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

```
sh-4.2#
```

Press CTRL + ] to exit the console.

The following aspects must be noted:

- the processes inside the container are running in a separate namespace, hence the different process hierarchy
- since no network configuration for the domain is explicitly specified, all networking interfaces are shared with the host (all the other interfaces are present too - br0 is mentioned as an example)
- since no filesystem configuration is specified for the domain, the filesystem is shared with the host-- all host mounts are present in the container as well.

6. To stop the container use the destroy command:

```
# virsh -c lxc:/// destroy container1
Domain container1 destroyed

# virsh -c lxc:/// list  --all
 Id    Name                           State
----------------------------------------------------
 -     container1                     shut off
```

7. To remove the domain from libvirt, use the undefine command.

```
# virsh -c lxc:/// undefine container1
Domain container1 has been undefined
```

### 18.3.2.2 Custom Container Filesystem

The libvirt documentation (http://libvirt.org/formatdomain.html#elementsFilesystems) details the flavors and usage of the **filesystem** tag in order to assign particular types of filesystem mounts to the domain.

Mounts

The <mount> tag specifies private root filesystem, available on host in a specific directory. It will be the rootfs of the container. The filesystem can be handcrafted, installed from media, debootstrapped, etc. Below is an example snippet of the XML that assigns a filesystem to a container:

```
<domain type='lxc'>
  [ ... ]
  <devices>
    [ ... ]
    <filesystem type='mount'>
      <source dir='/var/lib/lxc/foo/rootfs'/>
      <target dir='/'/>
    </filesystem>
  </devices>
</domain>
```

The result is that the domain is started with the root mounted at */var/lib/lxc/foo/rootfs*.

For ease of use, the example that follows will use the standard LXC command lxc-create to build a container Busybox rootfs. However, the default rootfs created by lxc-create will not work with libvirt tools as-is, and some

additional terminal setup must be done. This will be detailed in the next example: Container Terminal Setup on page 193.

### 18.3.2.3 Container Terminal Setup

Each LXC domain needs at least one console device defined in the XML. By default, libvirt will link this console to the process and make it available with **virsh console** command.

Libvirt also offers support for multiple consoles in a container. This section provides an example describing how libvirt can be used to start four processes inside the container and assign each one of these a private terminal. This will be done by using:

- a Busybox container filesystem, built with lxc-create
- a modified inittab
- the container XML configuration

Libvirt will mount a tmpfs on /dev and a devpts on /dev/pts and it will create all the device nodes itself inside the domain. The domain definition will be altered so that it will start the busybox-init as the init process.

The init process reads the /etc/inittab file inside the rootfs and will determine additional processes to start, and the terminals to link them to. Here is an example inittab that specifies the four processes to start and the separate tty device assigned to each:

```
::sysinit:/etc/init.d/rcS
tty1::askfirst:/bin/sh
tty2::respawn:/bin/getty -L tty2 115200 vt100
tty3::respawn:/bin/getty -L tty3 115200 vt100
tty4::respawn:/bin/getty -L tty4 115200 vt100
```

The domain XML configuration that shows the /sbin/init as the initial program to run and describes the four tty devices looks like this:

```
<domain type='lxc'>
  [ ... ]
  <os>
    <type>exe</type>
    <init>/sbin/init</init>
  </os>
  [ ... ]
  <devices>
    [ ... ]
    <console type='pty'>
      <target type='serial' port='0'/>
    </console>
    <console type='pty'>
      <target type='serial' port='1'/>
    </console>
    <console type='pty'>
      <target type='serial' port='2'/>
    </console>
    <console type='pty'>
      <target type='serial' port='3'/>
    </console>
  </devices>
</domain>
```

Using the inittab above and XML configs, **virsh start** will start the following process hierarchy:

```
# ps axf
[ ... ]
18450 ?        Ss   0:00 /usr/libexec/libvirt_lxc --name foo --console 24 --console 25 --
console 26 --console 27 --security=selinux --handshake 30 --background
18451 pts/0    Ss+ 0:00  \_ init
18454 ?        Ss   0:00      \_ /bin/syslogd
18459 ?        Ss   0:00      \_ /bin/sh
18460 pts/1    Ss+ 0:00      \_ /bin/getty -L tty2 115200 vt100
18461 pts/2    Ss+ 0:00      \_ /bin/getty -L tty3 115200 vt100
18462 pts/3    Ss+ 0:00      \_ /bin/getty -L tty4 115200 vt100
```

By running **virsh -c lxc:/// dumpxml foo**, we can see what alias libvirt has assigned to each console device:

```
<domain type='lxc' id='18450'>
  [ ... ]
  <devices>
    [ ... ]
    <console type='pty' tty='/dev/pts/3'>
      <source path='/dev/pts/3'/>
      <target type='serial' port='0'/>
      <alias name='console0'/>
    </console>
    <console type='pty'>
      <source path='/dev/pts/4'/>
      <target type='serial' port='1'/>
      <alias name='console1'/>
    </console>
    <console type='pty'>
      <source path='/dev/pts/5'/>
      <target type='serial' port='2'/>
      <alias name='console2'/>
    </console>
    <console type='pty'>
      <source path='/dev/pts/6'/>
      <target type='serial' port='3'/>
      <alias name='console3'/>
    </console>
  </devices>
</domain>
```

To connect to a particular console, one must run

```
virsh -c lxc:/// console --devname <console_alias> <domain_name>
```

```
[root@everest][~]# virsh -c lxc:/// console --devname console2 foo
Connected to domain foo
Escape character is ^]

everest.ea.freescale.net login: root
#
#
#
```

#### 18.3.2.4 Networking Examples

Libvirt offers extensive support when it comes to networking. The purpose of this section is to provide some insight of how standard LXC networking scenarios can be achieved with Libvirt - to provide the same functionality. In its further versions, this document could include details regarding other, more advanced networking options as well.

##### 18.3.2.4.1 Shared Networking

**Shared Networking**

By default libvirt containers share network interfaces and the network namespace with the host. To share the hosts network interfaces, simply do not define any network interfaces in the domain XML.

When at least one interface is defined, the container will be started in a new network namespace, with the defined interface and a loopback.

**No Networking**

In order to remove all access to the host's network interfaces, start the container in a new network namespace, without specifying any interface. To do this use the <privnet> XML tag as seen in the example below. This will a loopback interface in the new namespace, but the host network interfaces are not visible in the container.

```
<domain type='lxc'>
  [ ... ]
  <features>
    <privnet/>
  </features>
</domain>
```

##### 18.3.2.4.2 Ethernet Bridging

This is the recommended configuration for general guest activity on hosts with static wired networking configurations. It relies on 802.1d Ethernet Bridging, and it provides a bridge from the VM directly into the LAN. This assumes there is a bridge device on the host with one or more of the host's physical NICs enslaved to it. The guest VM will have an associated tun device created with a name of vnetN, which can also be overridden with the **target** element in the XML config file. This tun device will also be enslaved to the bridge. The IP range / network configuration is whatever is used on the LAN. This provides the guest VM full incoming and outgoing net access just like a physical machine. The bridge normally is a Linux bridge - but this can be configured to be an open vSwitch as well, if it is supported on the host. This is done by adding some further parameters in the config file.

**Host Configuration**

The physical interface **fm1-gb1** is added to a bridge device - **br0**:

```
ifconfig fm1-gb1 0.0.0.0 up
brctl addbr br0
ifconfig br0 192.168.1.141 up
brctl addif br0
```

**XML Description of the Interface**

Only the relevant part is described here - defining the interface of the guest in XML:

```
<domain type='lxc'>
  [ ... ]
  <devices>
    [ ... ]
    <interface type="bridge">
```

```
      <source bridge="br0" />
    </interface>
  </devices>
</domain>
```

### Guest Configuration and Testing

After booting the container the network interface can be managed normally:

```
~ # ifconfig
eth0      Link encap:Ethernet  HWaddr 52:54:00:83:78:F4
          inet6 addr: fe80::5054:ff:fe83:78f4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:22 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:3104 (3.0 KiB)  TX bytes:636 (636.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

~ # ifconfig eth0 192.168.1.143
~ # ping -c 3 192.168.1.1 # gateway
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: seq=0 ttl=64 time=3.557 ms
64 bytes from 192.168.1.1: seq=1 ttl=64 time=0.220 ms
64 bytes from 192.168.1.1: seq=2 ttl=64 time=0.227 ms

--- 192.168.1.1 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.220/1.334/3.557 ms
~ # ping -c 3 192.168.1.141 # host
PING 192.168.1.141 (192.168.1.141): 56 data bytes
64 bytes from 192.168.1.141: seq=0 ttl=64 time=3.493 ms
64 bytes from 192.168.1.141: seq=1 ttl=64 time=0.050 ms
64 bytes from 192.168.1.141: seq=2 ttl=64 time=0.036 ms

--- 192.168.1.141 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.036/1.193/3.493 ms
~ #
```

## 18.3.2.4.3 MACVLAN

Macvlan is a relatively new Linux kernel technology used to ease the task of networking in virtual machines. See the following article for some useful overview information: http://seravo.fi/2012/virtualized-bridged-networking-with-macvtap.

Macvlan is device driver consisting of 2 components:

- the **macvlan** driver - which makes it possible to create **virtual network interfaces** that "cling on" a physical network interface. Each virtual interface has its own MAC address distinct from the physical interface's MAC

address. Frames sent to or from the virtual interfaces are mapped to the physical interface, which is called **the lower interface**.

- the **tap** interface - a software-only interface, using to pass Ethernet frames. Instead of passing frames to and from a physical Ethernet card, the frames are read and written by a userspace program. The kernel makes the Tap interface available via the */dev/tapN* device file, where N is the index of the network interface.

Libvirt uses the macvtap device technology to attach virtual network interfaces to physical ones. This has no impact on the functionality of the physical interfaces on the host. The macvtap device has a different approach than the bridge. While the latter provides connectivity from the host to the virtual devices, the former isolates them. The bridge unifies the physical interface with the virtual ones, thus providing a common addressing space and connectivity between each 2 endpoints. The macvtap device will put the virtual interfaces in separate MAC-based VLAN's, isolated from the host.

The macvtap device can function in three different modes: **vepa**, **bridge** and **private** - libvirt supports all three of them. In order to configure networking using a macvtap device, the type attribute of the interface is set to "direct" which can be seen in the following XML example:

```
<domain type='lxc'>
  [ ... ]
    <devices>
    [ ... ]
    <interface type="direct">
      <source dev="p7p1" mode="vepa" />
    </interface>
  </devices>
</domain>
```

In the above example, **p7p1** is a physical network interface on the host. The **mode** tag specifies the mode of the macvtap device - and it can be one of the following:

- **vepa** (Virtual Ethernet Port Aggregator) - in this mode, which is the default, data between endpoints on the same lower device are sent via the lower device (physical Ethernet card), to the physical switch the lower device is connected to. This mode requires that the switch supports *Reflective Relay* mode, also known as *Hairpin* mode. Reflective Relay means that the switch can send back a frame on the same port it received it on. The reason this mode exists is to leverage the switching computation to an external switch (and thus freeing the host).
- **bridge** - in this mode, the endpoints can communicate directly without sending the data out via the lower device. There is no isolation between endpoints on the same lower device, but there is isolation between them and the lower device itself.
- **private** - the nodes on the same Macvtap device can never talk to each other. This is used when you want to isolate the virtual machines connected to the endpoints from each other, but not from the outside network.

## 18.3.2.4.4 Direct Assignment

This options enables a container to have private access to a host interface directly. It is similar to the **lxc-phys** networking configuration option. Technically, this option will move a host network interface from the host network namespace to the container's. Once the container is stopped (destroyed), the interface will be assigned back to the host network namespace. While the container is running, the interface will not be available from the host.

**XML Description of the Interface**

Assuming the host has interface **fm2-gb0**, this is the XML snippet that assigns it to the container:

```
<domain type='lxc'>
  [ ... ]
```

```
  <devices>
    [ ... ]
    <hostdev mode='capabilities' type='net'>
      <source>
        <interface>fm2-gb0</interface>
      </source>
    </hostdev>
  </devices>
</domain>
```

#### Guest Configuration and Testing

Once the container is started, the inteface must be configured with IP, netmask, etc. Then it can be used just like it would have been on the host.

```
~ # ifconfig fm2-gb0
fm2-gb0   Link encap:Ethernet  HWaddr 00:04:9f:00:02:05
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:18 errors:0 dropped:0 overruns:0 frame:0
          TX packets:56 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1094 (1.0 KiB)  TX bytes:4068 (3.9 KiB)
          Memory:fe5e0000-fe5e0fff

~ # ifconfig fm2-gb0 20.0.0.3 netmask 255.0.0.0
~ # ping -c 3 20.0.0.1
PING 20.0.0.1 (20.0.0.1) 56(84) bytes of data.
64 bytes from 20.0.0.1: icmp_req=1 ttl=64 time=0.377 ms
64 bytes from 20.0.0.1: icmp_req=2 ttl=64 time=0.186 ms
64 bytes from 20.0.0.1: icmp_req=3 ttl=64 time=0.198 ms

--- 20.0.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.186/0.253/0.377/0.089 ms
~ #
```

When returning to the host network namespace, the interface loses its configuration and it is down.

### 18.3.2.4.5 VLAN

Libvirt does not directly provide VLAN configuration support-- there is no equivalent to the capability in user space LXC where a container can be configured to have a specific VLAN assigned only to itself (see LXC: How to configure networking using a VLAN (lxc-vlan.conf) on page 224.

Libvirt containers does not offer equivalent support, but VLANs can be used with libvirt containers using traditional VLAN tools such as vconfig, and the **bridge** or **macvtap** network sharing modes.

This leads to the following different configuration scenarios for using VLAN based network interfaces:

1. Configuring a VLAN on the host:

   ```
   vconfig add eth0 2
   ifconfig eth0.2 192.168.20.2
   ```

   then exposing the **eth0.2** sub-interface in the containers, using **bridging** or direct attachment through a **macvtap** device. These would provide the same functionality as if **eth0.2** were a normal physical interface on the host, but it will only provide connectivity inside the VLAN.
2. Configuring VLAN in container shared with macvtap - in this scenario we assume that the domain has been configured to attach directly to the eth0 interface, and we create the VLAN inside the container. Note

that this scenario works only if eth0 does not have a sub-interface in the same VLAN defined on the host. As stated in the Macvtap section, ping will work only between virtual endpoints and the outside network.

3. Configuring VLAN in container shared with bridge - in this scenario we assume that the domain has been configured to attach to a host bridge. This bridge previously had a physical interface attached to it. When starting the domain, virtual endpoints will be created and attached to this bridge inside the domain. We create the VLAN sub-interfaces inside the created containers. Note that this scenario works only if there has not been configured a sub-interface on the physical interface with the same VLAN id.

# Chapter 19 Linux Containers (LXC) for Freescale QorIQ User's Guide

## 19.1 Introduction to Linux Containers

### 19.1.1 Freescale LXC Release Notes

This document describes current limitations in the release of LXC for Freescale SoCs.

```
Copyright (C) 2015 Freescale Semiconductor, Inc.

Freescale LXC Release Notes
05/12/2015

Overview
--------
This document describes new features, current limitations, and
working demos in Linux Containers (LXC) for Freescale QorIQ SDK 1.8.

New Features
------------
  o Add Seccomp support to LXC containers.

Fixes
-----
  o Reboot for Busybox containers.

Limitations
-----------

  o initramfs support.  Linux containers are not supported on initramfs
    filesystems because bind mounts and the pivot_root() system call
    are required by LXC, but not supported on this type of filesystem.

  o Currently seccomp support cannot be activated on ARMv8 platforms,
    due to this feature missing from the LXC package.

  o Currently unprivileged containers (making use of the user namespace)
    are not working 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.
```

### 19.1.2 Overview

This document is a guide and tutorial to using Linux Containers on Freescale e500-based, ARMv7 and ARMv8-based SoCs.

Linux Containers is a lightweight virtualization technology that allows the creation of environments in Linux called "**containers**" in which Linux applications can be run in isolation from the rest of the system and with fine grained control over resources allocated to the container (e.g. CPU, memory, network).

There are 2 implementations of containers in the QorIQ SDK:

- LXC. LXC is a user space package that provides a set of commands to create and manage containers and uses existing Linux kernel features to accomplish the desired isolation and control.
- Libvirt. The libvirt package is a virtualization toolkit that provides a set of management tools for managing virtual machines and Linux containers. See the ***Libvirt Users Guide*** chapter for general information regarding libvirt. The libvirt driver for containers is called "lxc", but the libvirt "lxc" driver is distinct from the user space LXC package.

Applications in a container run in a "sandbox" and can be restricted in what they can do and what visibility they have. In a container:

- An application "sees" only other processes that are in the container.
- An application has access only to network resources granted to the container.
- If configured as such, an application "sees" only a container-specific root filesystem. In addition to limiting access to data in the system's host rootfs, by limiting the */dev* entries that exist in the containers rootfs this limits the devices that the container can access.
- The file POSIX capabilities available to programs are controlled and configured by the system administrator.
- The container's processes run in what is known as a "control group" which the system administrator can use to monitor and control the container's resources.

Why are containers useful? Below are a few examples of container use cases:

- **Application partitioning** -- control CPU utilization between high priority and low priority applications, control what resources applications can access.
- **Virtual private server** -- boot multiple instances of user space, each which effectively looks like a private instance of a server. This approach is commonly used in website infrastructure.
- **Software upgrade** -- run Linux user space in a container, when it becomes necessary to upgrade applications in the system, create and test upgraded software in a new container. The old container can be stopped and the new container can be started as desired.
- **Terminal servers** -- user accesses the system with a thin client, with containers on the server providing applications. Each user gets a private, sandboxed workspace.

There are two general usage models for containers:

- **application containers**: Running a single application in a container. In this scenario, a single executable program is started in the container.

- **system containers**: Booting an instance of user space in a container. Booting multiple system containers allows multiple isolated instances of user space to run at the same time.

Containers are conceptually different than virtual machine technologies such as QEMU/KVM. Virtual machines emulate a hardware platform and are capable of booting an operating system kernel. A container is a mechanism to isolate Linux applications. In a system using containers there is only one Linux kernel running -- the host Linux kernel.

### 19.1.3 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.

### 19.1.4 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.

See the chapter ***Libvirt Users Guide*** for general information about libvirt.

#### Web

- libvirt LXC driver: http://libvirt.org/drvlxc.html
- Getting started with LXC using libvirt : https://www.berrange.com/posts/2011/09/27/getting-started-with-lxc-using-libvirt/
- LXC: Official web page for the LXC project: https://linuxcontainers.org/
- LXC: Overview article on LXC on IBM developerWorks (2009): http://www.ibm.com/developerworks/linux/library/l-lxc-containers/
- Article on POSIX file capabilities: http://www.friedhoff.org/posixfilecaps.html
- SUSE LXC tutorial: https://www.suse.com/documentation/sles11/singlehtml/lxc_quickstart/lxc_quickstart.html
- LXC Linux Containers, presentation: http://www.slideshare.net/samof76/lxc-17456998

- Stephane Graber's LXC 1.0 blog posts: https://www.stgraber.org/2013/12/20/lxc-1-0-blog-post-series/
- Linux Plumbers 2013 videos: https://www.youtube.com/channel/UCIxsmRWj3-795FMlrsikd3A/videos

**Containers and Security**

If using containers to sandbox untrusted applications, a thorough understanding is needed of the capabilities granted to a container and the security vulnerabilities they may imply. The following references are helpful for understanding container security:

- Ubuntu's security issues and mitigations with LXC, https://wiki.ubuntu.com/LxcSecurity
- Emeric Nasi, Exploiting capabilities, http://www.sevagas.com/IMG/pdf/exploiting_capabilities_the_dark_side.pdf
- 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
```

## 19.2 LXC: Build, Installation, Configuration

### 19.2.1 Summary

To prepare a root filesystem with the needed components for using LXC-based or libvirt containers the following steps are required:

1. Build and include the LXC and/or libvirt packages in the rootfs. For LXC see: LXC: Building with Yocto on page 204). For libvirt see the chapter ***Libvirt Users Guide*** .
2. If using busybox, update the busybox configuration and build to included features needed by containers (see Building Busybox on page 205).
3. Update the Linux kernel configuration and build to included features needed to support containers (see Building the Linux Kernel on page 206).
4. Update the rootfs so the system is ready to support containers (see Host Root Filesystem Configuration for Linux Containers on page 208).

### 19.2.2 LXC: Building with Yocto

LXC is a Linux user space package that can easily be added to a rootfs using the Yocto build system.

In the Freescale SDK, LXC and all pre-requisite user space packages are included when building the "full" and "virt" image types:

```
bitbake fsl-image-full
bitbake fsl-image-virt
```

LXC can be easily added to any rootfs image by updating the IMAGE_INSTALL_append variable in the **conf/local.conf** file in the Yocto build environment. For example, append the following line to **local.conf**:

```
IMAGE_INSTALL_append = " lxc"
```

In order to build LXC with seccomp support, add the following line to **local.conf**:

```
PACKAGECONFIG_append_pn-lxc = " seccomp"
```

## 19.2.3 Building Busybox

If using the busybox LXC template for creating the rootfs for containers, the busybox application must be built statically.

To configure and rebuild busybox:

```
bitbake busybox -c cleansstate
bitbake busybox -c menuconfig
bitbake busybox
bitbake [rootfs image type]
```

Ensure that the following configuration options are enabled in the menuconfig step:

**Busybox Configuration Options**

```
Busybox Settings --->
    Build Options --->
        [*] Build BusyBox as a static binary (no shared libs)
Coreutils  --->
    [*] touch
    [*]   Add support for -h
    [*]   Add support for SUSV3 features (-d -t -r)
    [*] sha3sum
    [*] sleep
    [*]   Enable multiple arguments and s/m/h/d suffixes
    [*]     Enable fractional arguments
Init Utilities --->
    [*] init
    [*]   Support reading an inittab file
    [*]   Run commands with leading dash with controlling tty
    [*]   Enable init to write to syslog
    [*]       Be _extra_ quiet on boot
    [*]   Support dumping core for child processes (debugging only)
    [*]   Support running init from within an initrd (not initramfs)
Login/Password Management Utilities --->
    [*] Support for shadow passwords
    [*] Use internal password and group functions rather than system functions
    [*]   Use internal shadow password functions
    [*] adduser
    [*]   Enable long options
    [*] addgroup
    [*]   Enable long options
    [*] deluser
    [*] delgroup
```

| Busybox Configuration Options |
|---|
| `[*] login`<br>`[*]   Support for login script`<br>`[*]   Support for /etc/nologin`<br>`[*]   Support for /etc/securetty`<br>`[*] passwd`<br>`[*] cryptpw`<br>`[*] chpasswd`<br>`[*] su`<br>`[*] sulogin`<br>`Networking Utilities  -->`<br>`[*] vconfig` |

The "RPC services" option must be disabled in Busybox:

| Busybox Configuration Options | Description |
|---|---|
| `Networking Utilities`<br>`[*] inetd`<br>`[ ]   Support RPC services` | Disable support for RPC services |

## 19.2.4 Building the Linux Kernel

In order to use LXC the Linux kernel must be configured with options to enable cgroups, namespaces, POSIX file capabilities, and options to support networking in containers.

To configure and build the Linux kernel:

```
bitbake virtual/kernel -c cleansstate
bitbake virtual/kernel -c menuconfig
bitbake virtual/kernel
```

Ensure that the following configuration options are enabled:

| Kernel Configuration Options | Description |
|---|---|
| `General setup --->`<br>`[*] Control Group support -->`<br>`[*] Example debug cgroup subsystem`<br>`[*] Freezer cgroup subsystem`<br>`[*] Device controller for cgroups`<br>`[*] Cpuset support`<br>`[*] Simple CPU accounting cgroup subsystem`<br>`[*] Resource counters`<br>`[*]     Memory Resource Controller for Control Groups`<br>`[*]        Memory Resource Controller Swap Extension`<br>`[*]           Memory Resource Controller Swap Extension enabled by default (NEW)` | Control Group settings |

<table>
<tr><th>Kernel Configuration Options</th><th>Description</th></tr>
<tr><td><pre>
        [*]        Memory Resource Controller Kernel Memory
accounting (EXPERIMENTAL)
        [*]     HugeTLB Resource Controller for Control Groups
        [*] Enable perf_event per-cpu per-container group
(cgroup) monitoring
        [*] Group CPU scheduler
        [*] Block IO controller
</pre></td><td></td></tr>
</table>

<table>
<tr><th>Kernel Configuration Options</th><th>Description</th></tr>
<tr><td><pre>
General setup --->
    [*] Namespaces support --->
        [*] UTS namespace
        [*] IPC namespace
        [*] User namespace
        [*] PID Namespaces
        [*] Network namespace
</pre></td><td>Namespaces settings</td></tr>
</table>

<table>
<tr><th>Kernel Configuration Options</th><th>Description</th></tr>
<tr><td><pre>
Device Drivers --->
    [*] Network device support --->
        <*> MAC-VLAN support (EXPERIMENTAL)
        <*>     MAC-VLAN based tap driver (EXPERIMENTAL)
        <*> Virtual ethernet pair device
</pre></td><td>Network Device Drivers settings</td></tr>
</table>

<table>
<tr><th>Kernel Configuration Options</th><th>Description</th></tr>
<tr><td><pre>
Device Drivers --->
    [*] Character devices --->
        [*] Unix98 PTY support
        [*]     Support multiple instances of devpts
</pre></td><td>Character Device Drivers settings</td></tr>
</table>

<table>
<tr><th>Kernel Configuration Options</th><th>Description</th></tr>
<tr><td><pre>
[*] Networking support --->
    Networking options --->
        <*> 802.1d Ethernet Bridging
        [*]       IGMP/MLD snooping (NEW)
        <*> 802.1Q VLAN Support
        [*]     GVRP (GARP VLAN Registration Protocol) support
</pre></td><td>Networking support settings</td></tr>
</table>

| Kernel Configuration Options | Description |
| --- | --- |
| `File systems --->`<br>`    <*> Second extended fs support`<br>`    [*]    Ext2 extended attributes`<br>`    [*]        Ext2 POSIX Access Control Lists`<br>`    <*> Ext3 journalling file system support`<br>`    [*] Ext3 extended attributes`<br>`    [*]    Ext3 POSIX Access Control Lists`<br>`    <*> The Extended 4 (ext4) filesystem`<br>`    [*]    Ext4 POSIX Access Control Lists` | File System settings |

| Kernel Configuration Options | Description |
| --- | --- |
| `[*] Enable the block layer --->`<br>`        [*] Block layer bio throttling support`<br>`        IO Schedulers --->`<br>`                <*> CFQ I/O scheduler>`<br>`                [*]   CFQ Group Scheduling support` | Block layer settings |

| Kernel Configuration Options | Description |
| --- | --- |
| `Kernel Features --->`<br>`        [*] Enable seccomp to safely compute untrusted bytecode` | Seccomp kernel support |

## 19.2.5 Host Root Filesystem Configuration for Linux Containers

In order to use containers, mount the ‘cgroups’ pseudo-filesystem. When booting kernels compiled with cgroups support, there is a default directory for mounting them - /sys/fs/cgroup. We will use this to mount our cgroups controllers.

We must mount the cgroup controllers separately, so that the setup is compatible with the libvirt lxc driver as well. In order to do this, we create a temporary tmpfs mount at /sys/fs/cgroup, so that we can create a directory for each independent controller to be mounted.

```
mount -t tmpfs tmpfs-cgroups /sys/fs/cgroup
mkdir /sys/fs/cgroup/{freezer,devices,memory,cpuacct,cpuset,cpu,blkio}
mount -t cgroup -ofreezer cgroup /sys/fs/cgroup/freezer
mount -t cgroup -odevices cgroup /sys/fs/cgroup/devices
mount -t cgroup -omemory cgroup /sys/fs/cgroup/memory
mount -t cgroup -ocpuacct cgroup /sys/fs/cgroup/cpuacct
mount -t cgroup -ocpuset cgroup /sys/fs/cgroup/cpuset
mount -t cgroup -ocpu cgroup /sys/fs/cgroup/cpu
mount -t cgroup -oblkio cgroup /sys/fs/cgroup/blkio
```

# 19.3 More Details

## 19.3.1 LXC: Command Reference

This section contains links to available open source documentation for the commands in the LXC user space package.

For a description of the libvirt commands for managing containers see the chapter **Libvirt Users Guide**.

Table 16:

| 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 |
| 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 | LXC Configuration File Reference on page 244 |
| 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

## 19.3.2 LXC: Configuration Files

NOTE
This section is applicable to LXC only, not to libvirt.

For LXC, configuration files are used to configure aspects of a container at the time it is created. The configuration file defines what resources are private to the container and what is shared. By default the following resources are private to a container:

- process IDs
- sysv ipc mechanisms
- mount points

This means for example, that by default the container will share network resources and the filesystem with the host system, but will have it's own private process IDs.

The container configuration file allows additional isolation to be specified through configuration in the following areas:

- network
- console
- mount points and the backing store for the root filesystem
- control groups (cgroups)
- POSIX capabilities

See the LXC Configuration File Reference on page 244 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 215 how-to article for an introduction to the container lifecycle and how configuration files are used when creating containers.

Several example configuration files are provided with LXC:

```
/usr/share/doc/lxc/examples/lxc-empty-netns.conf
/usr/share/doc/lxc/examples/lxc-complex.conf
```

```
        /usr/share/doc/lxc/examples/lxc-no-netns.conf
        /usr/share/doc/lxc/examples/lxc-vlan.conf
        /usr/share/doc/lxc/examples/lxc-macvlan.conf
        /usr/share/doc/lxc/examples/lxc-veth.conf
        /usr/share/doc/lxc/examples/lxc-phys.conf
```

## 19.3.3 LXC: Templates

NOTE

This section is applicable to LXC only, not to libvirt.

For LXC, When a container is "created" a directory for the container (which has the same name as the container) is created under /var/lib/lxc. This is where the container's configuration file is stored and can be edited.

For system containers (containers created with **lxc-create**), the default is for the root filesystem structure of the container to be stored here as well.

Creating containers is simplified by the use of example "templates" provided with the LXC. Template examples are provided for a number of different Linux distributions. A template is a script invoked by **lxc-create** that creates the root filesystem structure and sets up the container's config file.

The following example templates are provided with LXC and can be referred to for the expected template structure:

```
/usr/share/lxc/templates/lxc-alpine
/usr/share/lxc/templates/lxc-altlinux
/usr/share/lxc/templates/lxc-archlinux
/usr/share/lxc/templates/lxc-busybox
/usr/share/lxc/templates/lxc-centos
/usr/share/lxc/templates/lxc-cirros
/usr/share/lxc/templates/lxc-debian
/usr/share/lxc/templates/lxc-download
/usr/share/lxc/templates/lxc-fedora
/usr/share/lxc/templates/lxc-gentoo
/usr/share/lxc/templates/lxc-openmandriva
/usr/share/lxc/templates/lxc-opensuse
/usr/share/lxc/templates/lxc-oracle
/usr/share/lxc/templates/lxc-plamo
/usr/share/lxc/templates/lxc-sshd
/usr/share/lxc/templates/lxc-ubuntu
/usr/share/lxc/templates/lxc-ubuntu-cloud
```

For the Freescale Linux SDK for QorIQ the busybox template is recommended and has been tested with Yocto-created root filesystems.

The how-to examples provided in this user guide that create system containers use the busybox template.

## 19.3.4 Containers with Libvirt

This section provides an overview to using libvirt-based containers.

For an general introduction to libvirt, please see the chapter ***Libvirt Users Guide***. Also, see the container information available on the libvirt website: http://libvirt.org/drvlxc.html.

With libvirt, a container "domain" is specified in an XML file. The XML is used to "define" the container, which then allows the container to be managed with the standard libvirt domain lifecycle.

**Libvirt XML**

The XML for the simplest functional container would look like the example below:

```
<domain type='lxc'>
  <name>container1</name>
  <memory>500000</memory>
  <os>
    <type>exe</type>
    <init>/bin/sh</init>
  </os>
  <devices>
    <console type='pty'/>
  </devices>
</domain>
```

Refer to the XML reference information available on the libvirt website for detailed reference information: http://libvirt.org/formatdomain.html

The <domain> element must specify a type attribute of "lxc" for a container/lxc domain. There are 4 additional sub-nodes required:

- <name> - specifies the name of the container
- <memory> - specifies the maximum memory the container may use
- <os> - identifies the initial program to run. In the example this is /bin/sh. For an application based container this is the name of the application. If booting an instance of Linux user space this would typically by /sbin/init.
- <devices> - specifies any devices, in the above example there is just a console

To see a working example using this XML, see the how to article: Basic Example on page 190.

**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

## 19.3.5 Linux Control Groups (cgroups)

Linux control groups (or cgroups) is a feature of the Linux kernel that allows the allocation, prioritization,control, and monitoring of resources such as CPU time, memory, network bandwidth among groups of Linux processes.

Cgroups is one of the underlying Linux kernel features that LXC is built upon. LXC automatically creates a cgroup for each container when it is started. A pre-requisite for using LXC is mounting the cgroup virtual filesystem. Mounting the cgroup filesystem is presented in section Host Root Filesystem Configuration for Linux Containers on page 208.

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.

For an overview of cgoups, see the Linux kernel documentation overview here: Documentation/cgroups/cgroups.txt on page 260.

You may also check out the Red Hat documentation on cgroups here: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/ch-Subsystems_and_Tunable_Parameters.html.

Cgroup subsystems can be configured within the configuration file used when creating a container. The configuration file accepts cgroup configuration in the following form:

```
lxc.cgroup.[subsystem name] = <value>
```

See the LXC Configuration File Reference on page 244 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 228.

## 19.3.6 Linux Namespaces

Linux namespaces is a feature in the Linux kernel that allows one to unshare and isolate a processes' resources like UTS, PID, IPC, file system mount and network from their parent. To achieve this the kernel places the resources in different namespaces.

When LXC spawns the container's main process it unshares all these resources except the network. The network is controlled from the configuration file and is shared by default.

A network namespace provides an isolated view of the networking stack (network device interfaces, IPv4 and IPv6 protocol stacks, IP routing tables, firewall rules, the /proc/net and /sys/class/net directory trees, sockets, etc.). A physical network device can live in exactly one network namespace. A virtual network device ("veth") pair provides a pipe-like abstraction that can be used to create tunnels between network namespaces, and can be used to create a bridge to a physical network device in another namespace. When a network namespace is freed (i.e., when the last process in the namespace terminates), its physical network devices are moved back to the initial network namespace (not to the parent of the process).

Each namespace is documented in the Linux **clone** man page. See: clone (2)

## 19.3.7 POSIX Capabilities

Linux supports the concept of file "capabilities" which provides fine grained control over what executable programs are permitted to do. Instead of the "all or nothing" paradigm where a super-user or "root" has the power to perform all operations, capabilities provide a mechanism to grant a specific program specific capabilities.

LXC uses this feature of the kernel to implement containers. By default processes running in a container will have **all** capabilities, but this can be configured. Capabilities can be dropped in the container's configuration file. See LXC: Configuration Files on page 210.

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)

## 19.4 LXC: How To's

### 19.4.1 LXC: Getting Started (with a Busybox System Container)

The following article describes steps to run a simple container example. All the command below are issued from a host Linux command prompt.

1. Confirm that your kernel environment is configured correctly using **lxc-checkconfig**. All options should show as 'enabled.

```
# lxc-checkconfig
--- Namespaces ---
Namespaces: enabled
Utsname namespace: enabled
Ipc namespace: enabled
Pid namespace: enabled
User namespace: missing
Network namespace: enabled
Multiple /dev/pts instances: enabled

--- Control groups ---
Cgroup: enabled
Cgroup clone_children flag: enabled
Cgroup device: enabled
Cgroup sched: enabled
Cgroup cpu account: enabled
Cgroup memory controller: enabled
Cgroup cpuset: enabled

--- Misc ---
Veth pair device: enabled
Macvlan: enabled
Vlan: enabled
File capabilities: enabled

Note : Before booting a new kernel, you can check its configuration
usage : CONFIG=/path/to/config /usr/bin/lxc-checkconfig
```

If the cgroup namespace option shows as required:

```
Cgroup namespace: required
```

...most likely the /cgroup directory needs to be created and or mounted. See #unique_160.

NOTE: the User namespace is reported as missing. Although initial support for User Namespaces has been enabled in Linux 3.8, a lot of work still had to be done after this release and the USER_NS config flag could not be enabled. This has no impact on the functionality of containers, since User namespace support was not implemented - just the flag was there.

2. Create a container

   Create a system container using lxc-create and specify the busybox template and lxc-empty-netns.conf config file. lxc-empty-netns.conf is a simple config file with no networking:

```
# lxc-create -n foo -t busybox -f /usr/share/doc/lxc/examples/lxc-empty-netns.conf
setting root password to "root"
Password for 'root' changed
#
```

   By default, LXC will try to install the dropbear ssh utility, if it's available on the host system. The Busybox template also has support for installing OpenSSH (assuming it's installed on the host Linux) in the container. This needs to be passed explicitly using a command line parameter:

```
# lxc-create -n foo -t busybox -f /usr/share/doc/lxc/examples/lxc-empty-netns.conf --
-s openssh
setting root password to "root"
Password for 'root' changed
'OpenSSH' ssh utility installed
#
```

3. List containers that exist

```
# lxc-ls -l
drwxr-xr-x 3 root root 1024 May 30 15:37 foo
```

4. From a shell on the host Linux, start the container. When prompted, press 'Enter'.

```
# lxc-start -n foo

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 is the default one for starting and connecting to a container, 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 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 -d
# 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
#
```

### 19.4.2 LXC: How to configure non-virtualized networking (lxc-no-netns.conf)

One approach to providing networking capability to a container is to simply allow the container to use existing host network interfaces. To accomplish this, a configuration file is created with no networking setup (i.e. the **lxc.network.type** property is not set) and the default will be to allow the container to access the host's networking interfaces.

With this approach no network namespace is created for the container.

An example config is provided:

```
/usr/share/doc/lxc/examples/lxc-no-netns.conf
```

The contents of lxc-no-netns.conf look like this:

```
# Container with non-virtualized network
lxc.network.type = none
lxc.utsname = delta
```

The example below shows starting an application container (running bash) with this config file and shows that the host network interface fm2-mac5 is inherited and accessible by the container:

```
# lxc-execute -n mytest -f /usr/share/doc/lxc/examples/lxc-no-netns.conf -- /bin/bash
bash-4.3# ifconfig
fm2-mac5  Link encap:Ethernet  HWaddr 00:04:9f:02:7a:3b
          inet addr:10.0.0.1  Bcast:10.255.255.255  Mask:255.0.0.0
          inet6 addr: fe80::204:9fff:fe02:7a3b/64 Scope:Link
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:508 (508.0 B)
          Memory:fe5e8000-fe5e8fff

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:272 (272.0 B)  TX bytes:272 (272.0 B)
bash-4.2#
```

### 19.4.3 LXC: How to assign a physical network interface to a container (lxc-phys.conf)

One approach to providing networking capability to a container is to directly assign an available, unused network interface to the container. The interface is not shared, it becomes the private resource of the container.

An example LXC configuration file is provided to configure this type of networking:

```
    /usr/share/doc/lxc/examples/lxc-phys.conf
```

The contents of the default lxc-phys.conf example are show below:

```
# Container with network virtualized using a physical network device with name
# 'eth0'
lxc.utsname = gamma
lxc.network.type = phys
lxc.network.flags = up
lxc.network.link = eth0
lxc.network.hwaddr = 4a:49:43:49:79:ff
lxc.network.ipv4 = 10.2.3.6/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3297
```

**Note:** The network type is set to: **phys.** Make a copy of the example config file and update it with the name of the Ethernet interface to be assigned, an appropriate IP address, and any other appropriate changes (e.g. mac address). For example, the change (in universal diff format) to set the interface fm2-gb0 and IP address 192.168.10.3 would look like:

```
--- /usr/share/doc/lxc/examples/lxc-phys.conf
+++ lxc-phys.conf
@@ -3,7 +3,6 @@
 lxc.utsname = gamma
 lxc.network.type = phys
 lxc.network.flags = up
-lxc.network.link = eth0
-lxc.network.hwaddr = 4a:49:43:49:79:ff
-lxc.network.ipv4 = 10.2.3.6/24
-lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3297
+lxc.network.link = fm2-mac5
+lxc.network.hwaddr = 00:e0:0c:00:93:05
+lxc.network.ipv4 = 192.168.10.3/24
```

A simple way to test the new config file and the network interface is to run /bin/bash as a command with lxc-execute, which will provide a shell running in the container:

```

# lxc-execute -n mytest -f lxc-phys.conf -- /bin/bash
bash-4.2#
```

In the container, use the fm1-gb4 interface normally:

```
bash-4.3# ifconfig
fm2-mac5  Link encap:Ethernet  HWaddr 00:e0:0c:00:93:05
          inet addr:192.168.10.3  Bcast:192.168.10.255  Mask:255.255.255.0
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:508 (508.0 B)
          Memory:fe5e8000-fe5e8fff

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

```
bash-4.2# ping -c 3 192.168.10.1
PING 192.168.10.1 (192.168.10.1) 56(84) bytes of data.
64 bytes from 192.168.10.1: icmp_req=1 ttl=64 time=0.385 ms
64 bytes from 192.168.10.1: icmp_req=2 ttl=64 time=0.207 ms
64 bytes from 192.168.10.1: icmp_req=3 ttl=64 time=0.187 ms

--- 192.168.10.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.187/0.259/0.385/0.090 ms
```

## 19.4.4 LXC: How to configure networking with virtual Ethernet pairs (lxc-veth.conf)

One approach to providing a virtual network interface to a container is using the "Virtual ethernet pair device" feature of the Linux kernel in conjunction with a network bridge.

See the veth description in LXC Configuration File Reference on page 244 for additional details on this approach to networking.

With this approach LXC creates a new network namespace for the container.

The example configuration file lxc-veth.conf demonstrates this approach:

```
/usr/share/doc/lxc/examples/lxc-veth.conf
```

The contents of the default lxc-veth.conf example are show below:

```
# Container with network virtualized using a pre-configured bridge named br0 and
# veth pair virtual network devices
lxc.utsname = beta
lxc.network.type = veth
lxc.network.flags = up
lxc.network.link = br0
lxc.network.hwaddr = 4a:49:43:49:79:bf
lxc.network.ipv4 = 10.2.3.5/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3597
```

Note, the network type value is: **veth** and the link property value is **br0**.

First, create a network bridge which is attached to a physical network interface and assign the bridge an IP address. The bridge becomes one endpoint In the example below the bridge br0 is created, interface fm2-gb1 is added to it, and the bridge is assigned an IP address of 192.168.20.2.

```
# brctl addbr br0
# ifconfig br0 192.168.20.2 up
# ifconfig fm2-mac5 up
# brctl addif br0 fm2-mac5
```

Make a copy of the example config file and update it with an appropriate IP address and any other appropriate changes (e.g. mac address). For example, the change (in universal diff format) to update the IP address to 192.168.20.3 would look like:

```
--- /usr/share/doc/lxc/examples/lxc-veth.conf
+++ lxc-veth.conf
@@ -5,5 +5,5 @@
 lxc.network.flags = up
 lxc.network.link = br0
```

```
 lxc.network.hwaddr = 4a:49:43:49:79:bf
-lxc.network.ipv4 = 10.2.3.5/24
+lxc.network.ipv4 = 192.168.20.3/24
 lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3597
```

A simple way to test the new config file and the network interface is to run /bin/bash as a command with lxc-execute, which will provide a shell running in the container:

```
# lxc-execute -n mytest -f lxc-veth.conf -- /bin/bash
bash-4.2#
```

In the container, use the virtual network interface (eth0 in this example) normally:

```
bash-4.2# ifconfig
eth0      Link encap:Ethernet  HWaddr 4a:49:43:49:79:bf
          inet addr:192.168.20.3  Bcast:192.168.20.255  Mask:255.255.255.0
          inet6 addr: fe80::4849:43ff:fe49:79bf/64 Scope:Link
          inet6 addr: 2003:db8:1:0:214:1234:fe0b:3597/64 Scope:Global
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:468 (468.0 B)  TX bytes:586 (586.0 B)

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

bash-4.2# ping -c 3 192.168.20.1
PING 192.168.20.1 (192.168.20.1) 56(84) bytes of data.
64 bytes from 192.168.20.1: icmp_req=1 ttl=64 time=0.433 ms
64 bytes from 192.168.20.1: icmp_req=2 ttl=64 time=0.221 ms
64 bytes from 192.168.20.1: icmp_req=3 ttl=64 time=0.228 ms

--- 192.168.20.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.221/0.294/0.433/0.098 ms
```

### 19.4.5 LXC: How to configure networking with macvlan (lxc-macvlan.conf)

An LXC container can be provided with a virtual network interface using the "MAC-VLAN" feature of the Linux kernel (see kernel config option CONFIG_MACVLAN). MAC-VLAN allows virtual interfaces to be created that route packets to or from a MAC address to a physical network interface.

See the macvlan description in LXC Configuration File Reference on page 244 for some additional details on this approach to networking.

The example configuration file lxc-veth.conf demonstrates this approach:

```
/usr/share/doc/lxc/examples/lxc-macvlan.conf
```

The contents of the provided lxc-phys.conf example configuration file are show below:

```
# Container with network virtualized using the macvlan device driver
lxc.utsname = alpha
lxc.network.type = macvlan
lxc.network.flags = up
lxc.network.link = eth0
lxc.network.hwaddr = 4a:49:43:49:79:bd
lxc.network.ipv4 = 10.2.3.4/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
```

Make a copy of the example config file and update it with the physical network interface to be used, an appropriate IP address, and any other appropriate changes (e.g. mac address). For example, the change (in universal diff format) to specify the fm1-gb4 interface and update the IP address to 192.168.1.24 would look like:

```
--- /usr/share/doc/lxc/examples/lxc-macvlan.conf
+++ lxc-macvlan.conf
@@ -2,7 +2,7 @@
 lxc.utsname = alpha
 lxc.network.type = macvlan
 lxc.network.flags = up
-lxc.network.link = eth0
+lxc.network.link = fm2-mac5
 lxc.network.hwaddr = 4a:49:43:49:79:bd
-lxc.network.ipv4 = 10.2.3.4/24
+lxc.network.ipv4 = 192.168.10.3/24
 lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
```

Put the network interface in promiscuous mode:

```
# ifconfig fm2-mac5 promisc
# ifconfig fm2-mac5
fm2-gb0   Link encap:Ethernet  HWaddr 00:e0:0c:00:93:05
          inet addr:192.168.10.2  Bcast:192.168.10.255  Mask:255.255.255.0
          inet6 addr: fe80::2e0:cff:fe00:9305/64 Scope:Link
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:5 errors:0 dropped:0 overruns:0 frame:0
          TX packets:17 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:344 (344.0 B)  TX bytes:1314 (1.2 KiB)
          Memory:fe5e0000-fe5e0fff
```

Test the MAC-VLAN interface by starting an application container running /bin/bash:

```
# lxc-execute -n mytest -f lxc-macvlan.conf -- /bin/bash
bash-4.2#
```

Note: the shell prompt above ("bash-4.2") is in the container.

Test the interface in the now running container:

```
bash-4.2# ifconfig
eth0      Link encap:Ethernet  HWaddr 4a:49:43:49:79:bd
          inet addr:192.168.10.3  Bcast:192.168.10.255  Mask:255.255.255.0
          inet6 addr: fe80::4849:43ff:fe49:79bd/64 Scope:Link
          inet6 addr: 2003:db8:1:0:214:1234:fe0b:3596/64 Scope:Global
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
```

 Non-disclosure Agreement Required. 

```
          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
```

### 19.4.6 LXC: How to configure networking using a VLAN (lxc-vlan.conf)

A container can be provided with a virtual network interface using VLANs.

See the vlan description in LXC Configuration File Reference on page 244 for some additional details on this approach to networking.

The example configuration file lxc-veth.conf demonstrates this approach:

```
/usr/share/doc/lxc/examples/lxc-vlan.conf
```

The contents of the provided lxc-vlan.conf example configuration file are show below:

```
 # Container with network virtualized using the vlan device driver
lxc.utsname = alpha
lxc.network.type = vlan
lxc.network.vlan.id = 1234
lxc.network.flags = up
lxc.network.link = eth0
lxc.network.hwaddr = 4a:49:43:49:79:bd
lxc.network.ipv4 = 10.2.3.4/24
lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
```

Make a copy of the example config file and update it with the physical network interface to be used and the vlan ID, an appropriate IP address, and any other appropriate changes. For example, the change (in universal diff format) to specify the fm2-gb0 interface, a VLAN id of 2, and an IP address of 192.168.30.2 would look like:

```
--- /usr/share/doc/lxc/examples/lxc-vlan.conf   2013-05-30 14:22:14.980406375 +0300
+++ lxc-vlan.conf       2013-06-03 13:26:38.477580000 +0300
@@ -1,9 +1,9 @@
 # Container with network virtualized using the vlan device driver
 lxc.utsname = alpha
 lxc.network.type = vlan
```

```
-lxc.network.vlan.id = 1234
+lxc.network.vlan.id = 2
 lxc.network.flags = up
-lxc.network.link = eth0
+lxc.network.link = fm2-mac5
 lxc.network.hwaddr = 4a:49:43:49:79:bd
-lxc.network.ipv4 = 10.2.3.4/24
+lxc.network.ipv4 = 192.168.30.2/24
 lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
```

In this setup, the host is connected to a test machine through physical interface fm2-gb0. On the test machine, the following commands have been issued (interface p7p1 on this machine has physical link to fm2-gb0):

```
[root@everest][~]# modprobe 8021q
[root@everest][~]# lsmod | grep 8021q
8021q                  23476  0
garp                   13763  1 8021q
[root@everest][~]# vconfig add p7p1 2
Added VLAN with VID == 2 to IF -:p7p1:-
[root@everest][~]# ifconfig p7p1.2 192.168.30.1 up
```

Test the VLAN interface by starting an application container running /bin/bash:

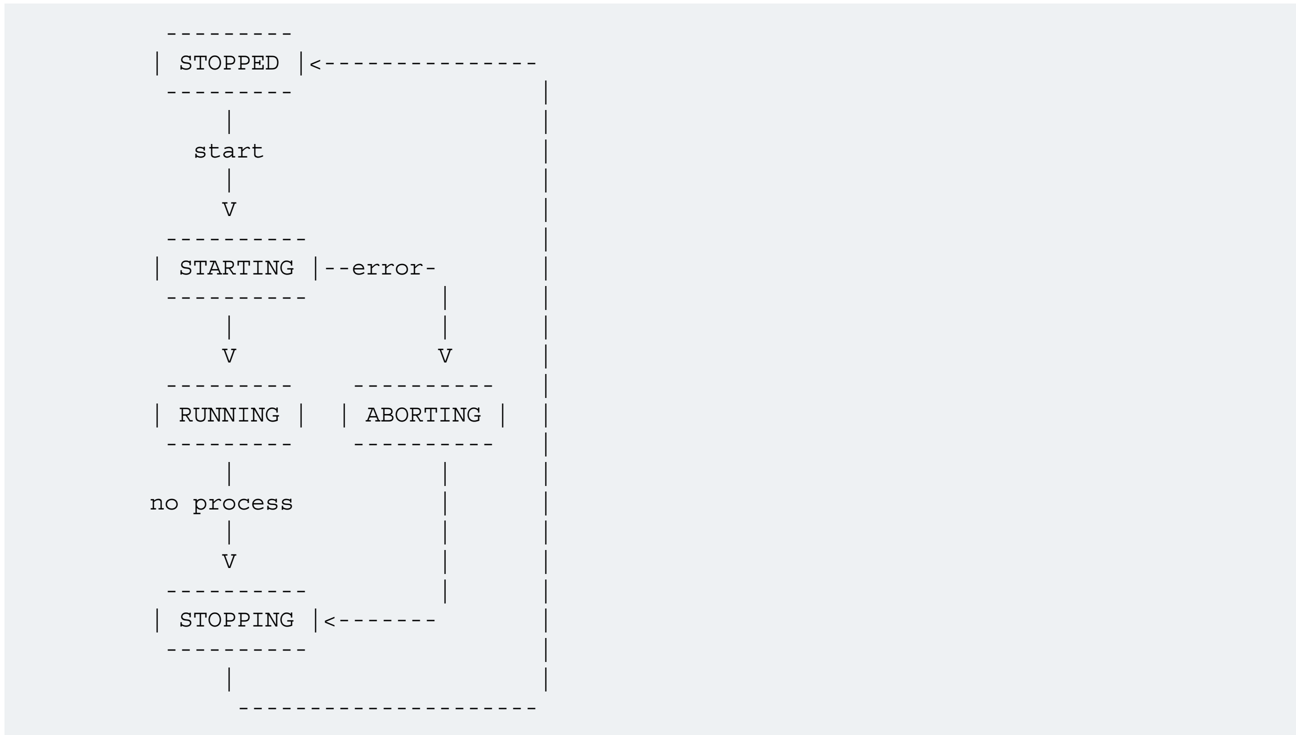
```
# lxc-execute -n mytest -f lxc-vlan.conf -- /bin/bash
bash-4.2#
```

Test the interface in the now running container:

```
bash-4.2# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.30.2  netmask 255.255.255.0  broadcast 192.168.30.255
        inet6 fe80::21e:c9ff:fe49:bb93  prefixlen 64  scopeid 0x20<link>
        ether 00:1e:c9:49:bb:93  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 6  bytes 468 (468.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 16436
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 0  (Local Loopback)
        RX packets 4  bytes 200 (200.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 4  bytes 200 (200.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

bash-4.2# ping -c 3 192.168.30.1
PING 192.168.30.1 (192.168.30.1) 56(84) bytes of data.
64 bytes from 192.168.30.1: icmp_req=1 ttl=64 time=0.338 ms
64 bytes from 192.168.30.1: icmp_req=2 ttl=64 time=0.372 ms
64 bytes from 192.168.30.1: icmp_req=3 ttl=64 time=0.355 ms

--- 192.168.30.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.338/0.355/0.372/0.013 ms
```

## 19.4.7 LXC: How to monitor containers

Containers transition through a set of well defined states. After a container is created it is in the "stopped" state.




A number of commands are available in LXC to monitor the state of a container. The following examples provide an introduction and demonstrate the capabilities of these commands.

1. lxc-info

   The lxc-info command shows the current state of the container.

   In the example below, a container called "foo" has already been created but not started and the container is stopped:

   ```
   # lxc-info -n foo
   Name:           foo
   State:          STOPPED
   After the container is started lxc-info shows the container in the running state:
   ```

   ```
   # lxc-start -n foo -d
   # 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 -d
# 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
```

## 19.4.8 LXC: How to modify the capabilities of a container to provide additional isolation

As described in POSIX Capabilities on page 214 ,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
```

### 19.4.9 LXC: How to use cgroups to manage and control a containers resources

This example demonstrates how to use control croups to control which CPU's a container is scheduled on and the percentage of CPU time allocated to a container.

In this example we'll examine and change:

- the **cpuset** subsystem's **cpus** parameter which controls which physical CPUs the container's processes will run on
- the **cpu** subsystem's **shares** parameter which controls the percentage of the CPU to be allocated to the container

1. Start two application containers each running /bin/bash:

   First container:

   ```
   # lxc-execute -n foo1 -f lxc-no-netns.conf -- /bin/bash
   bash-4.2#
   ```

   Second container:

   ```
   # lxc-execute -n foo2 -f lxc-no-netns.conf -- /bin/bash
   bash-4.2#
   ```

2. In both containers start a process that will put a 100% load on the CPUs:

   ```
   (while true; do true; done) &
   ```

3. The **cpuset.cpus** subsystem/value specifies which physical CPUs the container's processes run on. From a host shell, examine this with the **lxc-cgroup** command:

   ```
   # lxc-cgroup -n foo1 cpuset.cpus
   0-7
   ```

   In this example the host system has 4 CPUs.

   This can also be seen directly through the /cgroup filesystem:

   ```
   # cat /cgroup/cpuset/lxc/foo1/cpuset.cpus
   0-7
   ```

4. Change both containers to run only on CPU 2:

   ```
   # lxc-cgroup -n foo1 cpuset.cpus 2
   # lxc-cgroup -n foo2 cpuset.cpus 2
   #
   ```

   The top command now shows CPU 2 with 100% utilization. The bash commands running in each container, each have about 50% of the CPU:

   ```
   top - 17:14:41 up 10 min,  4 users,  load average: 1.64, 0.61, 0.23
   Tasks: 100 total,   3 running,  97 sleeping,   0 stopped,   0 zombie
   Cpu0  :  0.0%us,  0.3%sy,  0.0%ni, 99.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
   Cpu1  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
   Cpu2  :100.0%us, 0.0%sy, 0.0%ni, 0.0%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
   Cpu3  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
   Cpu4  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
   ```

```
Cpu5  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu6  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu7  :  0.0%us,  0.0%sy,  0.0%ni,100.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   3996400k total,   189836k used,  3806564k free,     1652k buffers
Swap:        0k total,        0k used,        0k free,    26180k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2875 root      20   0  3624  416  164 R   50  0.0   1:28.12 bash
 2874 root      20   0  3624  424  168 R   50  0.0   1:31.06 bash
```

5. The cpu.shares subsystem/value specifies the percentage of the CPU allocated to the cgroup/container. By default each container has a shares value of 1024:

```
# lxc-cgroup -n foo1 cpu.shares
1024
# lxc-cgroup -n foo2 cpu.shares
1024
```

6. Change container "foo2" to have about 10% of the CPU:

```
# lxc-cgroup -n foo2 cpu.shares 100
# lxc-cgroup -n foo1 cpu.shares 900
```

Now the top command output reflects the new CPU allocation:

```
  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2874 root      20   0  3624  424  168 R   90  0.0   2:53.63 bash
 2875 root      20   0  3624  416  164 R   10  0.0   2:11.36 bash
```

7. Stop the containers

```
# lxc-stop -n foo1 -k
# lxc-stop -n foo2 -k
#
```

## 19.4.10 LXC: How to run an application in a container with lxc-execute

The **lxc-execute** command allows a single application to be run in a container (as contrasted with a system container which boots an instance of Linux user space starting with System V style init).

In the example below an instance of a QEMU/KVM virtual machine is started in a container called foo.

Note, it is not required to explicitly create (and destroy) a container when running application containers with lxc-execute. The containers will automatically created and destroyed.

1. Start QEMU in the container with lxc-execute:

```
# lxc-execute -n foo -f lxc-no-netns.conf -- qemu-system-ppc -enable-kvm -smp 2 -m 256M
-nographic -M ppce500 -kernel uImage -initrd rootfs.ext2.gz -append "root=/dev/ram rw
console=ttyS0,115200" -serial tcp::4445,server,telnet
```

NOTE: For 64bit platforms, please replace qemu-system-ppc with qemu-system-ppc64.

Some notes:

- The QEMU command line follows the double dash ("--") 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
```

## 19.4.11 LXC: How to run an unprivileged container

With the addition of the user namespace in the Linux kernel, a normal user on a Linux host can create and run container instances. This feature has been integrated in the LXC package, starting from version 1.0.

The steps below detail the necessary steps required in order to configure and manage an unprivileged container.

**NOTE:** Before running these steps, make sure that the host is properly configured for container use, by running **lxc-checkconfig** (cgroups, namespaces, etc.). If some of the options are missing, please refer to the guidelines in LXC: Build, Installation, Configuration on page 204.

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

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.**

## 19.4.12 LXC: How to run containers with Seccomp protection

A large number of system calls are exposed to every userland process with many of them going unused for the entire lifetime of the process. As system calls change and mature, bugs are found and eradicated. A certain subset of userland applications benefit by having a reduced set of available system calls. The resulting set reduces the total kernel surface exposed to the application. System call filtering is meant for use with those applications.

Seccomp (short for *secure compute*) is a system call filtering mechanism present in the kernel. Initially it has been thought to be a sandboxing mechanism that would allow userspace processes to issue a very limited set of system calls - read(), write(), exit() and sigreturn(). This has been further known to be **seccomp mode 1**, and while it is strong on the security side, it doesn't leave much room for flexibility.

The next addition to seccomp was to allow filtering (or **seccomp mode 2**) based on the kernel BPF (Berkeley Packet Filter) infrastructure. This allows the system administrator to define complex and granular policies, per system call and its arguments. This is an extension to the BPF mechanism, that allows filtering to apply to system call numbers and their arguments, besides its original purpose (socket packets). The defined filter results in a seccomp policy which is attached to the userspace process in the form of a BPF program. Each time the process issues a system call, it is checked against this policy in order to determine how it will be handled:

- **SECCOMP_RET_KILL** - the task exits immediately without executing the system call.
- **SECCOMP_RET_TRAP** - the kernel sends a SIGSYS to the triggering task without executing the system call.
- **SECCOMP_RET_ERRNO** - a custom errno is returned to userspace without executing the system call.
- **SECCOMP_RET_TRACE** - causes the kernel to attempt to notify a ptrace-based tracer prior to executing the system call. The tracer can skip the system call or change it to a valid syscall number.
- **SECCOMP_RET_ALLOW** - results in the system call being executed.

In order to make the secure computing facility more userspace-friendly, the libseccomp library has been developed, which is meant to make it easier for applications to take advantage of the packet-filter-based seccomp mode. Prior to this, userspace applications had to define the BPF filter themselves. libseccomp restructures this approach into a simple and straightforward API which userspace applications can use. The latest version of libseccomp adds support for Python bindings as well, and is designed to work on multiple architectures (ARM, MIPS). PowerPC support has also been merged on a separate branch, and is expected to be included in future releases.

**Using seccomp with LXC containers**

Please refer to Building LXC for information on how to build LXC with seccomp support in the SDK.

**Note:** Currently LXC seccomp support is not available for ARM64 architectures.

Seccomp filtering integrates well with processes sandboxed as containers, as they can be assigned to untrusted users and exposed with a limited set of allowed system calls. This is a portable and granular low-level security mechanism which can be used to increase container security. The seccomp policy file needs to be applied only to the init process in the container, and will be inherited by all its children.

The seccomp policy for the container is specified using the container configuration file, in the form of a single line containing:

```
lxc.seccomp = /var/lib/lxc/lxc_seccomp.conf
```

An example lxc_seccomp policy file can look as follows:

```
2
blacklist
[ppc64]
mknod errno 120
sched_setscheduler trap
fchmodat kill
[ppc]
mknod
```

The elements in the policy file represent the following:

1. Version number (1/2) - a single integer containing a single number, 1 or 2. Version 1 only allows to define a set of system calls which are allowed (whitelisted) in the container, specified by syscall number. This version is limited in configurability and portability, since it's only used to specify allowed syscall numbers, which may

differ from arch to arch. Version 2 allows the policies to be either a whitelist (default deny, except mentioned syscalls) or a blacklist (default allow, except mentioned syscalls), and the syscalls can be expressed by name.

2. Policy type (whitelist/blacklist) - with an option of a default policy action (errno #, trap, kill, allow). The policy type is per seccomp context, and can be either whitelist or blacklist, not both.
3. Architecture tag [optional] - mentions that the following set of system calls will only be applied to a specific architecture. There can be multiple architecture tags and associated syscalls. These tags allow the same seccomp policy file to be used on multiple platforms, treating each one differently with respect to the set of system calls.
4. System calls - which can be expressed by number (in version 1) or name (in version 2). Optionally, an action can be expressed after the system call (errno #, trap, kill, allow), specifying the desired seccomp behavior. If this is omitted, the 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).

# 19.5 Libvirt How To's

## 19.5.1 Basic Example

The following example shows the lifecycle of a simple LXC libvirt domain called *container1*. The virsh tool is used for managing the lxc domain lifecycle.

**1. Confirm the host Linux configuration.** Begin by confirming that the host kernel is configured correctly and that rootfs setup such as mounting cgroups has been done. This can be done with the lxc-checkconfig command.

```
# lxc-checkconfig
--- Namespaces ---
Namespaces: enabled
Utsname namespace: enabled
```

```
Ipc namespace: enabled
Pid namespace: enabled
User namespace: missing
Network namespace: enabled
Multiple /dev/pts instances: enabled

--- Control groups ---
Cgroup: enabled
Cgroup clone_children flag: enabled
Cgroup device: enabled
Cgroup sched: enabled
Cgroup cpu account: enabled
Cgroup memory controller: enabled
Cgroup cpuset: enabled

--- Misc ---
Veth pair device: enabled
Macvlan: enabled
Vlan: enabled
File capabilities: enabled
```

2. **Create a libvirt XML file defining the container.** The example below shows a very simple container defined in container1.xml that runs the command /bin/sh and has a console:

```
# cat container1.xml
<domain type='lxc'>
  <name>container1</name>
  <memory>500000</memory>
  <os>
    <type>exe</type>
    <init>/bin/sh</init>
  </os>
  <devices>
    <console type='pty'/>
  </devices>
</domain>
```

3. **Define the container.** The *virsh define* command processes the XML and makes creates the new libvirt domain.

```
# virsh -c lxc:/// define container1.xml
Domain container1 defined from container1.xml

# virsh -c lxc:/// list --all
 Id    Name                           State
----------------------------------------------------
 -     container1                     shut off
```

4. Start the container.

```
# virsh -c lxc:/// start container1
Domain container1 started

# virsh -c lxc:/// list
 Id    Name                           State
----------------------------------------------------
 3196  container1                     running
```

5. Connect to the console.

```
# virsh -c lxc:/// console container1
Connected to domain container1
Escape character is ^]
sh-4.2#
sh-4.2# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 18:25 pts/2    00:00:00 /bin/sh
root         3     1  0 18:36 pts/2    00:00:00 ps -ef

sh-4.2# ifconfig br0
br0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.171.73.123  netmask 255.255.254.0  broadcast 10.171.73.255
        inet6 fe80::a00:27ff:fe01:fe07  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:01:fe:07  txqueuelen 0  (Ethernet)
        RX packets 865838  bytes 104029354 (99.2 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 104446  bytes 43998714 (41.9 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

sh-4.2#
```

Press CTRL + ] to exit the console.

The following aspects must be noted:

- the processes inside the container are running in a separate namespace, hence the different process hierarchy
- since no network configuration for the domain is explicitly specified, all networking interfaces are shared with the host (all the other interfaces are present too - br0 is mentioned as an example)
- since no filesystem configuration is specified for the domain, the filesystem is shared with the host-- all host mounts are present in the container as well.

6. To stop the container use the destroy command:

```
# virsh -c lxc:/// destroy container1
Domain container1 destroyed

# virsh -c lxc:/// list  --all
 Id    Name                           State
----------------------------------------------------
 -     container1                     shut off
```

7. To remove the domain from libvirt, use the undefine command.

```
# virsh -c lxc:/// undefine container1
Domain container1 has been undefined
```

# 19.6 USDPAA in LXC

This section describes running the USDPAA driver software in LXC containers. It presents the configuration steps for this scenario, along with observed issues and raised questions. Running USDPAA in containers is

intended to be a proof of concept and a supported feature for the QorIQ SDK starting from v1.4. USDPAA is only available for DPAA v1.x platforms.

The P4080DS RefMan was also used to determine the proper SerDes Lane Configuration for the test board.

## 19.6.1 General Setup

**Prerequisites**

The test environment consisted of the following components in the **FSR Boardfarm**:

- a P4080DS board - **dumbo**
- a Linux test machine - **zro04qoriq02** with 2 test ports

The **dumbo** test board presents the following port configuration:

```
[b43198@zro04qoriqsrv ~]$ lars info dumbo
Name            : dumbo
Template        : P4080DS
Category        : board
Status          : reserved
Current user    : Purcareata Bogdan-B43198
Usable          : True

Station     : <Station(176)>
Console     : telnet://10.171.77.74:51201
Power       : adam5000://10.171.77.131/18
Reset       : adam5000://10.171.77.131/18
JTAG        : 10161212

Installed devices:
dumbo.duart_p1   : PORT
dumbo.rgmii_p1   : SERVICE
dumbo.sgmii_s3p1: MCCPORT
dumbo.sgmii_s3p2: MCCPORT
dumbo.sgmii_s3p3: SWITCH
dumbo.usb20_p1   : PORT
dumbo.xaui_s4p1 : PORT
```

The board exposes 2 SGMII ports available for testing, managed by Fman: *dumbo.sgmii_s3p1* and *dumbo.sgmii_s3p2*. These ports have been connected to the test machine as follows:

```
[b43198@zro04qoriqsrv][~]$ lars list links
zro04qoriq02.p1      <--->     dumbo.sgmii_s3p1
zro04qoriq02.p2      <--->     dumbo.sgmii_s3p2
```

The following software components have been compiled from the SDK:

- fman microcode - fsl_fman_ucode.bin
- rcw - R_PPSXN_0x10/rcw_13g_1500mhz_rev2.bin
- u-boot
- kernel - uImage--3.8-r15.1-p4080ds-20130527105557.bin
- device tree blob - uImage_sdk_1.4_23.05_usdpaa_multiple_1pool.dtb
- rootfs - fsl-image-core-p4080ds-20130527121546.rootfs.ext2.gz.u-boot

The device tree has been slightly modified in order to support multiple running instances of the USDPAA application. This modification will be detailed in subsection Running multiple USDPAA instances on page 239.

### Booting the Board

```
setenv kernelimage; setenv dtbimage; setenv rootfsimage;setenv bootargs
console=ttyS0,115200;setenv tftpram 'tftp 1000000 $kernelimage;tftp
c00000 $dtbimage;tftp 2000000 $rootfsimage'; setenv bootram 'bootm 1000000 2000000
c00000'

setenv kernelimage b43198/uImage--3.8-r15.1-p4080ds-20130527105557.bin; setenv dtbimage
b43198/uImage_sdk_1.4_23.05_usdpaa_multiple_1pool.dtb; setenv rootfsimage b43198/fsl-
image-core-p4080ds-20130527121546.rootfs.ext2.gz.u-boot

setenv bootargs root=/dev/ram rw console=ttyS0,115200 usdpaa_mem=256M bportals=s0-1
qportals=s0-1

run tftpram; run bootram
```

### Running a Single Reflector Process

According to the documentation, it takes 2 steps:

- running the Frame Manager Configuration tool (**fmc**) - NOTE: documentation states that this tool can be used only once
- running the reflector process

Both applications require 2 files:

- FMC Policy Definition File - PCD
- FMC Configuration Source File

These files are passed as arguments and must be the same for the two applications. The policy definition file can be the same regardless of SerDes configuration. The configuration source file is SerDes config dependent. Since the **dumbo** board exposes 2 SGMII ports, it has been booted with SerDes config 0x10 and the config file **usdpaa_config_p4_serdes_0x10.xml** has been modified as follows:

```
<cfgdata>
 <config>
  <engine name="fm1">
   <port type="1G" number="0" policy="hash_ipsec_src_dst_spi_policy6"/>
   <port type="1G" number="1" policy="hash_ipsec_src_dst_spi_policy7"/>
  </engine>
 </config>
</cfgdata>
```

This translates into calling **fmc** and **reflector** for interfaces **fm2-gb0** and **fm2-gb1**.

```
root@p4080ds:~# fmc -c usdpaa_config_p4_serdes_0x10.xml -p usdpaa_policy_hash_ipv4.xml -
a
root@p4080ds:~# reflector -c usdpaa_config_p4_serdes_0x10.xml -p
usdpaa_policy_hash_ipv4.xml -b 120:0:0 0..7
Found /fsl,dpaa/dpa-fman0-oh@1, Tx Channel = 46, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@5, Tx Channel = 61, FMAN = 1, Port ID = 0
Found /fsl,dpaa/ethernet@6, Tx Channel = 62, FMAN = 1, Port ID = 1
Found /fsl,dpaa/ethernet@9, Tx Channel = 60, FMAN = 1, Port ID = 0
WARN:Can't find Qman clock frequency
Configuring for 2 network interfaces
Allocated DMA region size 0x1000000
reflector starting
Released 120 bufs to BPID 8
Released 120 bufs to BPID 9
```

```
Thread uid:0 alive (on cpu 0)
Thread uid:1 alive (on cpu 1)
Thread uid:2 alive (on cpu 2)
Thread uid:3 alive (on cpu 3)
Thread uid:4 alive (on cpu 4)
Thread uid:5 alive (on cpu 5)
Thread uid:6 alive (on cpu 6)
Thread uid:7 alive (on cpu 7)
reflector>
```

**Testing**

Testing has been done using the steps indicated in he documentation. The **zro04qoriq04** test workstation has been configured as follows:

- IP addresses:

```
ifconfig eth3 192.168.10.2 up
ifconfig eth4 192.168.20.2 up
```

- ARP entries:

```
arp -s 192.168.10.1 00:E0:0C:00:93:05
arp -s 192.168.20.1 00:E0:0C:00:93:06
```

Test commands:

```
[b43198@zro04qoriq04 ~]$ 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_seq=1 ttl=64 time=0.227 ms
64 bytes from 192.168.10.1: icmp_seq=2 ttl=64 time=0.216 ms
64 bytes from 192.168.10.1: icmp_seq=3 ttl=64 time=0.338 ms

--- 192.168.10.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2034ms
rtt min/avg/max/mdev = 0.216/0.260/0.338/0.056 ms

[b43198@zro04qoriq04 ~]$ 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_seq=1 ttl=64 time=0.298 ms
64 bytes from 192.168.20.1: icmp_seq=2 ttl=64 time=0.204 ms
64 bytes from 192.168.20.1: icmp_seq=3 ttl=64 time=0.229 ms

--- 192.168.20.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2032ms
rtt min/avg/max/mdev = 0.204/0.243/0.298/0.043 ms
```

## 19.6.2 Running multiple USDPAA instances

According to reference "Running multiple distinct PPAM application processes (or "instances" as they are sometimes described) requires that each process have its own dedicated set of FMAN interfaces, buffer pools, and cores".

The following resources need to be partitioned:

- interfaces - by using an additional **-i** parameter to the reflector application. Since Since there are only 2 FMan interfaces available on the board - **fm2-gb0** and **fm2-gb1** - the maximum number of USDPAA instances that can be run is 2.

- cores - by using an additional **cpu-range** parameter to the reflector application, in the form of **'index'** or **'first .. last'**. The reflector will start threads affine to the mentioned cores.
- buffer pools - this is more delicate and will be discussed in detail in the next subsection.

**Partitioning Buffer Pools**

Buffer pools are associated to Ethernet interfaces and the mappings are done in the device tree blob.

There should be distinct buffer pools for distinct USDPAA processes. When sharing buffer pools among different running processes, they fail with SEGFAULT.

Documentation states that *"Each PPAC-based application process will initialize the (usually 3) buffer pools used by each FMan interface that belongs to it. (If a buffer pool is used by more than one interface, it will only be initialised once.) By default the number of buffers to allocate for the triplet of pools used by an FMan interface is 0 for the first two pools and 1728 for the third. The default allocation triplet can be overridden via the -b option."*

One can define 1, 2 or 3 buffer pools per FMan interface. When doing custom seeding via the -b parameter - which only accepts a triplet x:y:z - the first value (x) corresponds to the first bufer, the second (y) corresponds to the second buffer and the third value (z) to the third buffer. If an interface only has 1 associated buffer pool, passing *-b 120:0:0* is valid, as it will seed only the 1st buffer pool - the interface's only one; if it has 2 buffer pools - *-b 120:120:0* is also valid. You may also pass other values than 0 in the triplets - since they don't have any buffer pool correspondence, they will be discarded. One must seed the buffer pool with respect to the DMA memory size allocated for USDPAA and passed via the **usdpaa_mem** boot argument.

The following is a snippet from the device tree containing the buffer pools for interfaces **fm2-gb0** (*ethernet@5*) and **fm2-gb1** (*ethernet@6*).

```
[ ... ]

2167     ethernet@5 {
2168       compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
2169       fsl,fman-mac = <0x59>;
2170       fsl,bman-buffer-pools = <0x53>;
2171       fsl,qman-frame-queues-rx = <0x5c 0x1 0x5d 0x1>;
2172       fsl,qman-frame-queues-tx = <0x7c 0x1 0x7d 0x1>;
2173     };
2174
2175     ethernet@6 {
2176       compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
2177       fsl,fman-mac = <0x5a>;
2178       fsl,bman-buffer-pools = <0x54>;
2179       fsl,qman-frame-queues-rx = <0x5e 0x1 0x5f 0x1>;
2180       fsl,qman-frame-queues-tx = <0x7e 0x1 0x7f 0x1>;
2181     };

[ ... ]

2214   buffer-pool@7 {
2215     compatible = "fsl,p4080-bpool", "fsl,bpool";
2216     fsl,bpid = <0x7>;
2217     fsl,bpool-ethernet-cfg = <0x0 0x0 0x0 0xc0 0x0 0xdeadbeef>;
2218     fsl,bpool-thresholds = <0x400 0xc00 0x0 0x0>;
2219     linux,phandle = <0x52>;
2220     phandle = <0x52>;
2221   };
2222
2223   buffer-pool@8 {
2224     compatible = "fsl,p4080-bpool", "fsl,bpool";
2225     fsl,bpid = <0x8>;
2226     fsl,bpool-ethernet-cfg = <0x0 0x0 0x0 0x240 0x0 0xabbaf00d>;
```

```
2227     fsl,bpool-thresholds = <0x100 0x300 0x0 0x0>;
2228     linux,phandle = <0x53>;
2229     phandle = <0x53>;
2230   };
2231
2232   buffer-pool@9 {
2233     compatible = "fsl,p4080-bpool", "fsl,bpool";
2234     fsl,bpid = <0x9>;
2235     fsl,bpool-ethernet-cfg = <0x0 0x0 0x0 0x6c0 0x0 0xfeedabba>;
2236     fsl,bpool-thresholds = <0x100 0x300 0x0 0x0>;
2237     linux,phandle = <0x54>;
2238     phandle = <0x54>;
2239   };

[ ... ]
```

### Running 2 Reflectors

To run 2 instances of the reflector application, one must:

- run the fmc tool - only once
- run 2 reflector processes

The fmc tool is passed an .xml config file containing both FMan interfaces present on the board. In order to provide each reflector instance with a different interface, you can either:

- pass a .xml config file describing only that particular interface;
- pass a .xml config file with both interfaces, but also pass an additional **-i** parameter, specifying the designated one.

The differential config files may look as follows:

- usdpaa_config_p4_serdes_0x10_0.xml - only fm2-gb0 is present:

```
<cfgdata>
 <config>
  <engine name="fm1">
   <port type="1G" number="0" policy="hash_ipsec_src_dst_spi_policy6"/>
  </engine>
 </config>
</cfgdata>
```

- usdpaa_config_p4_serdes_0x10_1.xml - only fm2-gb1 is present:

```
<cfgdata>
 <config>
  <engine name="fm1">
   <port type="1G" number="1" policy="hash_ipsec_src_dst_spi_policy7"/>
  </engine>
 </config>
</cfgdata>
```

In order to run 2 reflectors, you can issue the following commands - one in each terminal:

```
reflector -c usdpaa_config_p4_serdes_0x10.xml -p usdpaa_policy_hash_ipv4.xml -b 120:0:0
-i fm2-gb0 0..1
reflector -c usdpaa_config_p4_serdes_0x10.xml -p usdpaa_policy_hash_ipv4.xml -b 120:0:0
-i fm2-gb1 4..5
```

... or these:

```
reflector -c usdpaa_config_p4_serdes_0x10_0.xml -p usdpaa_policy_hash_ipv4.xml -b
120:0:0 0..1
reflector -c usdpaa_config_p4_serdes_0x10_1.xml -p usdpaa_policy_hash_ipv4.xml -b
120:0:0 4..5
```

## 19.6.3 Running Reflector in Containers

### Inspecting the Reflector App

The first step in investigating reflector behavior was to **strace** it. Since strace exposes application syscalls, and containers are running with the same kernel as the host, this should be an appropriate approach.

Looking at the results, the reflector opens the following files:

- /lib/* - library files;
- /proc/device-tree/* - the entire board device tree (.dtb);
- the passed .xml config files;
- /dev/mem - access portal memory mappings;
- /dev/fsl-usdpaa - manage the kernel DPAA resource allocator;
- /dev/fsl-usdpaa-irq - interrupt controller device (replacement for /dev/uio* entries);
- /etc/terminfo/* - terminal settings.

In order to run the test, all these files must be present in the container filesystem. Each container filesystem is configured to contain the /dev/mem and /dev/fsl-usdpaa devices - these are shared. Since SDK 1.4, the /dev/bman-uio and /dev/qman-uio entries have been replaced with a common **/dev/fsl-usdpaa-irq** device used by the processes to register to receiving the interrupts. This device is shared among the containers as well. These devices will then be created with mknod commands inside each container.

### Running the test

NOTE: commands are shown for one container only; commands for the other container are issued symmetrically.

1. mount the croups filesystem on host - requirement of LXC (see Host Root Filesystem Configuration for Linux Containers on page 208).
2. run the fmc tool on host - the fmc tool is used to initialize the FMan kernel driver and should be run only once after each successful boot, regardless of how many USDPAA application instances are started/stopped afterwards. It has no particular importance whether this is run on the host or inside a container, since its purpose is to configure kernel space variables - and these are shared between the host and the containers. Considering this aspect, and the fact that fmc should be run only once, it is best for it to be run on the host, before starting any USDPAA aplication:

   ```
   # fmc -c usdpaa_config_p4_serdes_0x10.xml -p usdpaa_policy_hash_ipv4.xml -a
   ```

3. edit the LXC container config file - **lxc-usdpaa-1.conf** - doing the following:
   - set container name
   - enable device rights
   - set dedicated cores

   ```
   lxc.utsname = usdpaa1

   # You may add some other parameters here if you like
   ```

```
# First deny all device access , then allow specific ones
lxc.cgroup.devices.deny = a

# Enable console and terminal devices
lxc.cgroup.devices.allow = c 5:* rwm # console
lxc.cgroup.devices.allow = c 4:* rwm # tty0
lxc.cgroup.devices.allow = c 136:* rwm # tty1
# Enable other devices access
lxc.cgroup.devices.allow = b 1:0 rwm # ram0
lxc.cgroup.devices.allow = c 1:3 rwm # null
lxc.cgroup.devices.allow = c 1:9 rwm # urandom
# Enable usdpaa
lxc.cgroup.devices.allow = c 10:61 rwm # fsl-usdpaa-irq
lxc.cgroup.devices.allow = c 10:62 rwm # fsl-usdpaa
lxc.cgroup.devices.allow = c 1:1 rwm # mem

# Dedicated cores
lxc.cgroup.cpuset.cpus = 0,1,2,3
```

4. create and start the container:

```
root@p4080ds:~# lxc-create -n foo1 -t busybox -f lxc-usdpaa-1.conf
setting root password to "root"
Password for 'root' changed
'busybox' template installed
'foo1' created
root@p4080ds:~# lxc-start -n foo1 -d
root@p4080ds:~# lxc-console -n foo1

Type <Ctrl+a q> to exit the console, <Ctrl+a Ctrl+a> to enter Ctrl+a itself

foo1 login: root
Password: root
~ #
```

5. copy binaries and xml files from host:

```
# reflector binary
root@p4080ds:~# mkdir -p /var/lib/lxc/foo1/rootfs/usr/bin
root@p4080ds:~# cp /usr/bin/reflector /var/lib/lxc/foo1/rootfs/usr/bin/
# usdpaa local xmls
root@p4080ds:~# cp usdpaa_* /var/lib/lxc/foo1/rootfs/root/
# terminfo settings
root@p4080ds:~# mkdir -p /var/lib/lxc/foo1/rootfs/etc/terminfo
root@p4080ds:~# cp -R /etc/terminfo/* /var/lib/lxc/foo1/rootfs/etc/terminfo/
```

6. setup the container - create devices in /dev:

```
~ # mknod /dev/fsl-usdpaa-irq c 10 61
~ # mknod /dev/fsl-usdpaa c 10 62
~ # mknod /dev/mem c 1 1
```

7. run reflector in container:

```
~ # reflector -c usdpaa_config_p4_serdes_0x10.xml -p usdpaa_policy_hash_ipv4.xml -b
120:0:0 -i fm2-gb0 0..3
Found /fsl,dpaa/dpa-fman0-oh@1, Tx Channel = 46, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@5, Tx Channel = 61, FMAN = 1, Port ID = 0
```

```
Found /fsl,dpaa/ethernet@6, Tx Channel = 62, FMAN = 1, Port ID = 1
Found /fsl,dpaa/ethernet@9, Tx Channel = 60, FMAN = 1, Port ID = 0
WARN:Can't find Qman clock frequency
Configuring for 1 network interface
Allocated DMA region size 0x1000000
reflector starting
Released 120 bufs to BPID 8
Thread uid:0 alive (on cpu 0)
Thread uid:1 alive (on cpu 1)
Thread uid:2 alive (on cpu 2)
Thread uid:3 alive (on cpu 3)
reflector>
```

8. test from Linux PC:

```
[b43198@zro04qoriq04 ~]$ 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_seq=1 ttl=64 time=0.177 ms
64 bytes from 192.168.10.1: icmp_seq=2 ttl=64 time=0.217 ms
64 bytes from 192.168.10.1: icmp_seq=3 ttl=64 time=0.321 ms

--- 192.168.10.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2037ms
rtt min/avg/max/mdev = 0.177/0.238/0.321/0.061 ms
```

By following the same steps for the second container - with slight modications regarding the container name and assigned resources - another reflector instance can be successfully started and run in parallel with the first one, for the second FMan interface **fm2-gb1**.

# 19.7 Appendix

## 19.7.1 LXC Configuration File Reference

The text below comes from the lxc.conf.5 man page:

```
LXC.CONF(5)                                                  LXC.CONF(5)
NAME

       lxc.conf - Configuration files for LXC.
DESCRIPTION

       LXC configuration is split in two parts. Container configuration and
       system configuration.

   CONTAINER CONFIGURATION
       The container configuration is held in the config stored in the
       container's directory.

       A basic configuration is generated at container creation time with
       the default's recommended for the chosen template as well as extra
       default keys coming from the default.conf file.

       That default.conf file is either located at
       /usr/local/etc/lxc/default.conf or for unprivileged containers at
       ~/.config/lxc/default.conf.
```

```
       Details about the syntax of this file can be found in:
       lxc.container.conf(5)

   SYSTEM CONFIGURATION
       The system configuration is located at /usr/local/etc/lxc/lxc.conf or
       ~/.config/lxc/lxc.conf for unprivileged containers.

       This configuration file is used to set values such as default lookup
       paths and storage backend settings for LXC.

       Details about the syntax of this file can be found in:
       lxc.system.conf(5)
SEE ALSO

       lxc(1), lxc.container.conf(5), lxc.system.conf(5), lxc-usernet(5)
AUTHOR

       Stéphane Graber <stgraber@ubuntu.com>
COLOPHON

       This page is part of the lxc (Linux containers) project.  Information
       about the project can be found at http://linuxcontainers.org/.  If
       you have a bug report for this manual page, send it to
       lxc-devel@lists.linuxcontainers.org.  This page was obtained from the
       project's upstream Git repository (git://github.com/lxc/lxc) on
       2014-05-21.  If you discover any rendering problems in this HTML
       version of the page, or you believe there is a better or more up-to-
       date source for the page, or you have corrections or improvements to
       the information in this COLOPHON (which is not part of the original
       manual page), send a mail to man-pages@man7.org

                        Wed May 21 10:30:15 CEST 2014            LXC.CONF(5)
```

The text below comes from the lxc.container.conf.5 man page:

```
LXC.CONTAINER.CONF(5)                                  LXC.CONTAINER.CONF(5)
NAME

       lxc.container.conf - LXC container configuration file

DESCRIPTION

       The linux containers (lxc) are always created before being used. This
       creation defines a set of system resources to be virtualized /
       isolated when a process is using the container. By default, the pids,
       sysv ipc and mount points are virtualized and isolated. The other
       system resources are shared across containers, until they are
       explicitly defined in the configuration file. For example, if there
       is no network configuration, the network will be shared between the
       creator of the container and the container itself, but if the network
       is specified, a new network stack is created for the container and
       the container can no longer use the network of its ancestor.

       The configuration file defines the different system resources to be
       assigned for the container. At present, the utsname, the network, the
       mount points, the root file system, the user namespace, and the
       control groups are supported.
```

```
        Each option in the configuration file has the form key = value
        fitting in one line. The '#' character means the line is a comment.

    CONFIGURATION
        In order to ease administration of multiple related containers, it is
        possible to have a container configuration file cause another file to
        be loaded. For instance, network configuration can be defined in one
        common file which is included by multiple containers. Then, if the
        containers are moved to another host, only one file may need to be
        updated.

        lxc.include
               Specify the file to be included. The included file must be in
               the same valid lxc configuration file format.

    ARCHITECTURE
        Allows one to set the architecture for the container. For example,
        set a 32bits architecture for a container running 32bits binaries on
        a 64bits host. This fixes the container scripts which rely on the
        architecture to do some work like downloading the packages.

        lxc.arch
               Specify the architecture for the container.

               Valid options are x86, i686, x86_64, amd64

    HOSTNAME
        The utsname section defines the hostname to be set for the container.
        That means the container can set its own hostname without changing
        the one from the system. That makes the hostname private for the
        container.

        lxc.utsname
               specify the hostname for the container

    HALT SIGNAL
        Allows one to specify signal name or number, sent by lxc-stop to the
        container's init process to cleanly shutdown the container. Different
        init systems could use different signals to perform clean shutdown
        sequence. This option allows the signal to be specified in kill(1)
        fashion, e.g. SIGPWR, SIGRTMIN+14, SIGRTMAX-10 or plain number. The
        default signal is SIGPWR.

        lxc.haltsignal
               specify the signal used to halt the container

    STOP SIGNAL
        Allows one to specify signal name or number, sent by lxc-stop to
        forcibly shutdown the container. This option allows signal to be
        specified in kill(1) fashion, e.g. SIGKILL, SIGRTMIN+14, SIGRTMAX-10
        or plain number.  The default signal is SIGKILL.

        lxc.stopsignal
               specify the signal used to stop the container

    NETWORK
        The network section defines how the network is virtualized in the
        container. The network virtualization acts at layer two. In order to
        use the network virtualization, parameters must be specified to
        define the network interfaces of the container. Several virtual
```

```
    interfaces can be assigned and used in a container even if the system
    has only one physical network interface.

    lxc.network.type
           specify what kind of network virtualization to be used for the
           container. Each time a lxc.network.type field is found a new
           round of network configuration begins. In this way, several
           network virtualization types can be specified for the same
           container, as well as assigning several network interfaces for
           one container. The different virtualization types can be:

           none: will cause the container to share the host's network
           namespace. This means the host network devices are usable in
           the container. It also means that if both the container and
           host have upstart as init, 'halt' in a container (for
           instance) will shut down the host.

           empty: will create only the loopback interface.

           veth: a peer network device is created with one side assigned
           to the container and the other side is attached to a bridge
           specified by the lxc.network.link. If the bridge is not
           specified, then the veth pair device will be created but not
           attached to any bridge. Otherwise, the bridge has to be setup
           before on the system, lxc won't handle any configuration
           outside of the container. By default lxc choose a name for the
           network device belonging to the outside of the container, this
           name is handled by lxc, but if you wish to handle this name
           yourself, you can tell lxc to set a specific name with the
           lxc.network.veth.pair option.

           vlan: a vlan interface is linked with the interface specified
           by the lxc.network.link and assigned to the container. The
           vlan identifier is specified with the option
           lxc.network.vlan.id.

           macvlan: a macvlan interface is linked with the interface
           specified by the lxc.network.link and assigned to the
           container.  lxc.network.macvlan.mode specifies the mode the
           macvlan will use to communicate between different macvlan on
           the same upper device. The accepted modes are private, the
           device never communicates with any other device on the same
           upper_dev (default), vepa, the new Virtual Ethernet Port
           Aggregator (VEPA) mode, it assumes that the adjacent bridge
           returns all frames where both source and destination are local
           to the macvlan port, i.e. the bridge is set up as a reflective
           relay. Broadcast frames coming in from the upper_dev get
           flooded to all macvlan interfaces in VEPA mode, local frames
           are not delivered locally, or bridge, it provides the behavior
           of a simple bridge between different macvlan interfaces on the
           same port. Frames from one interface to another one get
           delivered directly and are not sent out externally. Broadcast
           frames get flooded to all other bridge ports and to the
           external interface, but when they come back from a reflective
           relay, we don't deliver them again. Since we know all the MAC
           addresses, the macvlan bridge mode does not require learning
           or STP like the bridge module does.

           phys: an already existing interface specified by the
           lxc.network.link is assigned to the container.
```

```
        lxc.network.flags
               specify an action to do for the network.

               up: activates the interface.

        lxc.network.link
               specify the interface to be used for real network traffic.

        lxc.network.mtu
               specify the maximum transfer unit for this interface.

        lxc.network.name
               the interface name is dynamically allocated, but if another
               name is needed because the configuration files being used by
               the container use a generic name, eg. eth0, this option will
               rename the interface in the container.

        lxc.network.hwaddr
               the interface mac address is dynamically allocated by default
               to the virtual interface, but in some cases, this is needed to
               resolve a mac address conflict or to always have the same
               link-local ipv6 address.  Any "x" in address will be replaced
               by random value, this allows setting hwaddr templates.

        lxc.network.ipv4
               specify the ipv4 address to assign to the virtualized
               interface. Several lines specify several ipv4 addresses.  The
               address is in format x.y.z.t/m, eg. 192.168.1.123/24. The
               broadcast address should be specified on the same line, right
               after the ipv4 address.

        lxc.network.ipv4.gateway
               specify the ipv4 address to use as the gateway inside the
               container. The address is in format x.y.z.t, eg.
               192.168.1.123.  Can also have the special value auto, which
               means to take the primary address from the bridge interface
               (as specified by the lxc.network.link option) and use that as
               the gateway. auto is only available when using the veth and
               macvlan network types.

        lxc.network.ipv6
               specify the ipv6 address to assign to the virtualized
               interface. Several lines specify several ipv6 addresses.  The
               address is in format x::y/m, eg.
               2003:db8:1:0:214:1234:fe0b:3596/64

        lxc.network.ipv6.gateway
               specify the ipv6 address to use as the gateway inside the
               container. The address is in format x::y, eg. 2003:db8:1:0::1
               Can also have the special value auto, which means to take the
               primary address from the bridge interface (as specified by the
               lxc.network.link option) and use that as the gateway. auto is
               only available when using the veth and macvlan network types.

        lxc.network.script.up
               add a configuration option to specify a script to be executed
               after creating and configuring the network used from the host
               side. The following arguments are passed to the script:
               container name and config section name (net) Additional
```

```
            arguments depend on the config section employing a script
            hook; the following are used by the network system: execution
            context (up), network type (empty/veth/macvlan/phys),
            Depending on the network type, other arguments may be passed:
            veth/macvlan/phys. And finally (host-sided) device name.

            Standard output from the script is logged at debug level.
            Standard error is not logged, but can be captured by the hook
            redirecting its standard error to standard output.

     lxc.network.script.down
            add a configuration option to specify a script to be executed
            before destroying the network used from the host side. The
            following arguments are passed to the script: container name
            and config section name (net) Additional arguments depend on
            the config section employing a script hook; the following are
            used by the network system: execution context (down), network
            type (empty/veth/macvlan/phys), Depending on the network type,
            other arguments may be passed: veth/macvlan/phys. And finally
            (host-sided) device name.

            Standard output from the script is logged at debug level.
            Standard error is not logged, but can be captured by the hook
            redirecting its standard error to standard output.

 NEW PSEUDO TTY INSTANCE (DEVPTS)
     For stricter isolation the container can have its own private
     instance of the pseudo tty.

     lxc.pts
            If set, the container will have a new pseudo tty instance,
            making this private to it. The value specifies the maximum
            number of pseudo ttys allowed for a pts instance (this
            limitation is not implemented yet).

 CONTAINER SYSTEM CONSOLE
     If the container is configured with a root filesystem and the inittab
     file is setup to use the console, you may want to specify where the
     output of this console goes.

     lxc.console
            Specify a path to a file where the console output will be
            written. The keyword 'none' will simply disable the console.
            This is dangerous once if have a rootfs with a console device
            file where the application can write, the messages will fall
            in the host.

 CONSOLE THROUGH THE TTYS
     This option is useful if the container is configured with a root
     filesystem and the inittab file is setup to launch a getty on the
     ttys. The option specifies the number of ttys to be available for the
     container. The number of gettys in the inittab file of the container
     should not be greater than the number of ttys specified in this
     option, otherwise the excess getty sessions will die and respawn
     indefinitely giving annoying messages on the console or in
     /var/log/messages.

     lxc.tty
            Specify the number of tty to make available to the container.
```

```
CONSOLE DEVICES LOCATION
    LXC consoles are provided through Unix98 PTYs created on the host and
    bind-mounted over the expected devices in the container.  By default,
    they are bind-mounted over /dev/console and /dev/ttyN. This can
    prevent package upgrades in the guest. Therefore you can specify a
    directory location (under /dev under which LXC will create the files
    and bind-mount over them. These will then be symbolically linked to
    /dev/console and /dev/ttyN.  A package upgrade can then succeed as it
    is able to remove and replace the symbolic links.

    lxc.devttydir
           Specify a directory under /dev under which to create the
           container console devices.

/DEV DIRECTORY
    By default, lxc creates a few symbolic links (fd,stdin,stdout,stderr)
    in the container's /dev directory but does not automatically create
    device node entries. This allows the container's /dev to be set up as
    needed in the container rootfs. If lxc.autodev is set to 1, then
    after mounting the container's rootfs LXC will mount a fresh tmpfs
    under /dev (limited to 100k) and fill in a minimal set of initial
    devices.  This is generally required when starting a container
    containing a "systemd" based "init" but may be optional at other
    times. Additional devices in the containers /dev directory may be
    created through the use of the lxc.hook.autodev hook.

    lxc.autodev
           Set this to 1 to have LXC mount and populate a minimal /dev
           when starting the container.

ENABLE KMSG SYMLINK
    Enable creating /dev/kmsg as symlink to /dev/console. This defaults
    to 1.

    lxc.kmsg
           Set this to 0 to disable /dev/kmsg symlinking.

MOUNT POINTS
    The mount points section specifies the different places to be
    mounted. These mount points will be private to the container and
    won't be visible by the processes running outside of the container.
    This is useful to mount /etc, /var or /home for examples.

    lxc.mount
           specify a file location in the fstab format, containing the
           mount information. The mount target location can and in most
           cases should be a relative path, which will become relative to
           the mounted container root. For instance,

           proc proc proc nodev,noexec,nosuid 0 0

           Will mount a proc filesystem under the container's /proc,
           regardless of where the root filesystem comes from. This is
           resilient to block device backed filesystems as well as
           container cloning.

           Note that when mounting a filesystem from an image file or
           block device the third field (fs_vfstype) cannot be auto as
           with mount(8) but must be explicitly specified.
```

```
lxc.mount.entry
       specify a mount point corresponding to a line in the fstab
       format.

lxc.mount.auto
       specify which standard kernel file systems should be
       automatically mounted. This may dramatically simplify the
       configuration. The file systems are:

       · proc:mixed (or proc): mount /proc as read-write, but remount
         /proc/sys and /proc/sysrq-trigger read-only for security /
         container isolation purposes.

       · proc:rw: mount /proc as read-write

       · sys:ro (or sys): mount /sys as read-only for security /
         container isolation purposes.

       · sys:rw: mount /sys as read-write

       · cgroup:mixed: mount a tmpfs to /sys/fs/cgroup, create
         directories for all hierarchies to which the container is
         added, create subdirectories there with the name of the
         cgroup, and bind-mount the container's own cgroup into that
         directory.  The container will be able to write to its own
         cgroup directory, but not the parents, since they will be
         remounted read-only

       · cgroup:ro: similar to cgroup:mixed, but everything will be
         mounted read-only.

       · cgroup:rw: similar to cgroup:mixed, but everything will be
         mounted read-write. Note that the paths leading up to the
         container's own cgroup will be writable, but will not be a
         cgroup filesystem but just part of the tmpfs of
         /sys/fs/cgroup

       · cgroup (without specifier): defaults to cgroup:rw if the
         container retains the CAP_SYS_ADMIN capability, cgroup:mixed
         otherwise.

       · cgroup-full:mixed: mount a tmpfs to /sys/fs/cgroup, create
         directories for all hierarchies to which the container is
         added, bind-mount the hierarchies from the host to the
         container and make everything read-only except the
         container's own cgroup. Note that compared to cgroup, where
         all paths leading up to the container's own cgroup are just
         simple directories in the underlying tmpfs, here
         /sys/fs/cgroup/$hierarchy will contain the host's full
         cgroup hierarchy, albeit read-only outside the container's
         own cgroup.  This may leak quite a bit of information into
         the container.

       · cgroup-full:ro: similar to cgroup-full:mixed, but everything
         will be mounted read-only.

       · cgroup-full:rw: similar to cgroup-full:mixed, but everything
         will be mounted read-write. Note that in this case, the
         container may escape its own cgroup. (Note also that if the
         container has CAP_SYS_ADMIN support and can mount the cgroup
```

```
              filesystem itself, it may do so anyway.)

            · cgroup-full (without specifier): defaults to cgroup-full:rw
              if the container retains the CAP_SYS_ADMIN capability,
              cgroup-full:mixed otherwise.

     Note that if automatic mounting of the cgroup filesystem is enabled,
     the tmpfs under /sys/fs/cgroup will always be mounted read-write (but
     for the :mixed and :ro cases, the individual hierarchies,
     /sys/fs/cgroup/$hierarchy, will be read-only). This is in order to
     work around a quirk in Ubuntu's mountall(8) command that will cause
     containers to wait for user input at boot if /sys/fs/cgroup is
     mounted read-only and the container can't remount it read-write due
     to a lack of CAP_SYS_ADMIN.

     Examples:

                lxc.mount.auto = proc sys cgroup
                lxc.mount.auto = proc:rw sys:rw cgroup-full:rw

 ROOT FILE SYSTEM
     The root file system of the container can be different than that of
     the host system.

     lxc.rootfs
            specify the root file system for the container. It can be an
            image file, a directory or a block device. If not specified,
            the container shares its root file system with the host.

            For directory or simple block-device backed containers, a
            pathname can be used. If the rootfs is backed by a nbd device,
            then nbd:file:1 specifies that file should be attached to a
            nbd device, and partition 1 should be mounted as the rootfs.
            nbd:file specifies that the nbd device itself should be
            mounted. overlayfs:/lower:/upper specifies that the rootfs
            should be an overlay with /upper being mounted read-write over
            a read-only mount of /lower.  aufs:/lower:/upper does the same
            using aufs in place of overlayfs. loop:/file tells lxc to
            attach /file to a loop device and mount the loop device.

     lxc.rootfs.mount
            where to recursively bind lxc.rootfs before pivoting. This is
            to ensure success of the pivot_root(8) syscall. Any directory
            suffices, the default should generally work.

     lxc.rootfs.options
            extra mount options to use when mounting the rootfs.

     lxc.pivotdir
            where to pivot the original root file system under lxc.rootfs,
            specified relatively to that. The default is mnt.  It is
            created if necessary, and also removed after unmounting
            everything from it during container setup.

 CONTROL GROUP
     The control group section contains the configuration for the
     different subsystem. lxc does not check the correctness of the
     subsystem name. This has the disadvantage of not detecting
     configuration errors until the container is started, but has the
     advantage of permitting any future subsystem.
```

```
    lxc.cgroup.[subsystem name]
           specify the control group value to be set. The subsystem name
           is the literal name of the control group subsystem. The
           permitted names and the syntax of their values is not dictated
           by LXC, instead it depends on the features of the Linux kernel
           running at the time the container is started, eg.
           lxc.cgroup.cpuset.cpus

CAPABILITIES
    The capabilities can be dropped in the container if this one is run
    as root.

    lxc.cap.drop
           Specify the capability to be dropped in the container. A
           single line defining several capabilities with a space
           separation is allowed. The format is the lower case of the
           capability definition without the "CAP_" prefix, eg.
           CAP_SYS_MODULE should be specified as sys_module. See
           capabilities(7),

    lxc.cap.keep
           Specify the capability to be kept in the container. All other
           capabilities will be dropped.

APPARMOR PROFILE
    If lxc was compiled and installed with apparmor support, and the host
    system has apparmor enabled, then the apparmor profile under which
    the container should be run can be specified in the container
    configuration. The default is lxc-container-default.

    lxc.aa_profile
           Specify the apparmor profile under which the container should
           be run. To specify that the container should be unconfined,
           use

           lxc.aa_profile = unconfined

SELINUX CONTEXT
    If lxc was compiled and installed with SELinux support, and the host
    system has SELinux enabled, then the SELinux context under which the
    container should be run can be specified in the container
    configuration. The default is unconfined_t, which means that lxc will
    not attempt to change contexts.

    lxc.se_context
           Specify the SELinux context under which the container should
           be run or unconfined_t. For example

           lxc.se_context = unconfined_u:unconfined_r:lxc_t:s0-s0:c0.c1023

SECCOMP CONFIGURATION
    A container can be started with a reduced set of available system
    calls by loading a seccomp profile at startup. The seccomp
    configuration file must begin with a version number on the first
    line, a policy type on the second line, followed by the
    configuration.

    Versions 1 and 2 are currently supported. In version 1, the policy is
    a simple whitelist. The second line therefore must read "whitelist",
```

```
    with the rest of the file containing one (numeric) sycall number per
    line. Each syscall number is whitelisted, while every unlisted number
    is blacklisted for use in the container

    In version 2, the policy may be blacklist or whitelist, supports per-
    rule and per-policy default actions, and supports per-architecture
    system call resolution from textual names.

    An example blacklist policy, in which all system calls are allowed
    except for mknod, which will simply do nothing and return 0
    (success), looks like:

    2
    blacklist
    mknod errno 0

    lxc.seccomp
           Specify a file containing the seccomp configuration to load
           before the container starts.

UID MAPPINGS
    A container can be started in a private user namespace with user and
    group id mappings. For instance, you can map userid 0 in the
    container to userid 200000 on the host. The root user in the
    container will be privileged in the container, but unprivileged on
    the host. Normally a system container will want a range of ids, so
    you would map, for instance, user and group ids 0 through 20,000 in
    the container to the ids 200,000 through 220,000.

    lxc.id_map
           Four values must be provided. First a character, either 'u',
           or 'g', to specify whether user or group ids are being mapped.
           Next is the first userid as seen in the user namespace of the
           container. Next is the userid as seen on the host. Finally, a
           range indicating the number of consecutive ids to map.

CONTAINER HOOKS
    Container hooks are programs or scripts which can be executed at
    various times in a container's lifetime.

    When a container hook is executed, information is passed both as
    command line arguments and through environment variables.  The
    arguments are:

    · Container name.

    · Section (always 'lxc').

    · The hook type (i.e. 'clone' or 'pre-mount').

    · Additional arguments In the case of the clone hook, any extra
      arguments passed to lxc-clone will appear as further arguments to
      the hook.

    The following environment variables are set:

    · LXC_NAME: is the container's name.

    · LXC_ROOTFS_MOUNT: the path to the mounted root filesystem.
```

```
    · LXC_CONFIG_FILE: the path to the container configuration file.

    · LXC_SRC_NAME: in the case of the clone hook, this is the original
      container's name.

    · LXC_ROOTFS_PATH: this is the lxc.rootfs entry for the container.
      Note this is likely not where the mounted rootfs is to be found,
      use LXC_ROOTFS_MOUNT for that.

    Standard output from the hooks is logged at debug level.  Standard
    error is not logged, but can be captured by the hook redirecting its
    standard error to standard output.

    lxc.hook.pre-start
           A hook to be run in the host's namespace before the container
           ttys, consoles, or mounts are up.

    lxc.hook.pre-mount
           A hook to be run in the container's fs namespace but before
           the rootfs has been set up. This allows for manipulation of
           the rootfs, i.e. to mount an encrypted filesystem. Mounts done
           in this hook will not be reflected on the host (apart from
           mounts propagation), so they will be automatically cleaned up
           when the container shuts down.

    lxc.hook.mount
           A hook to be run in the container's namespace after mounting
           has been done, but before the pivot_root.

    lxc.hook.autodev
           A hook to be run in the container's namespace after mounting
           has been done and after any mount hooks have run, but before
           the pivot_root, if lxc.autodev == 1.  The purpose of this hook
           is to assist in populating the /dev directory of the container
           when using the autodev option for systemd based containers.
           The container's /dev directory is relative to the
           ${LXC_ROOTFS_MOUNT} environment variable available when the
           hook is run.

    lxc.hook.start
           A hook to be run in the container's namespace immediately
           before executing the container's init. This requires the
           program to be available in the container.

    lxc.hook.post-stop
           A hook to be run in the host's namespace after the container
           has been shut down.

    lxc.hook.clone
           A hook to be run when the container is cloned to a new one.
           See lxc-clone(1) for more information.

CONTAINER HOOKS ENVIRONMENT VARIABLES
    A number of environment variables are made available to the startup
    hooks to provide configuration information and assist in the
    functioning of the hooks. Not all variables are valid in all
    contexts. In particular, all paths are relative to the host system
    and, as such, not valid during the lxc.hook.start hook.

    LXC_NAME
```

```
            The LXC name of the container. Useful for logging messages in
            common log environments. [-n]

     LXC_CONFIG_FILE
            Host relative path to the container configuration file. This
            gives the container to reference the original, top level,
            configuration file for the container in order to locate any
            additional configuration information not otherwise made
            available. [-f]

     LXC_CONSOLE
            The path to the console output of the container if not NULL.
            [-c] [lxc.console]

     LXC_CONSOLE_LOGPATH
            The path to the console log output of the container if not
            NULL.  [-L]

     LXC_ROOTFS_MOUNT
            The mount location to which the container is initially bound.
            This will be the host relative path to the container rootfs
            for the container instance being started and is where changes
            should be made for that instance.  [lxc.rootfs.mount]

     LXC_ROOTFS_PATH
            The host relative path to the container root which has been
            mounted to the rootfs.mount location.  [lxc.rootfs]

 LOGGING
     Logging can be configured on a per-container basis. By default,
     depending upon how the lxc package was compiled, container startup is
     logged only at the ERROR level, and logged to a file named after the
     container (with '.log' appended) either under the container path, or
     under /usr/local/var/log/lxc.

     Both the default log level and the log file can be specified in the
     container configuration file, overriding the default behavior. Note
     that the configuration file entries can in turn be overridden by the
     command line options to lxc-start.

     lxc.loglevel
            The level at which to log. The log level is an integer in the
            range of 0..8 inclusive, where a lower number means more
            verbose debugging. In particular 0 = trace, 1 = debug, 2 =
            info, 3 = notice, 4 = warn, 5 = error, 6 = critical, 7 =
            alert, and 8 = fatal. If unspecified, the level defaults to 5
            (error), so that only errors and above are logged.

            Note that when a script (such as either a hook script or a
            network interface up or down script) is called, the script's
            standard output is logged at level 1, debug.

     lxc.logfile
            The file to which logging info should be written.

 AUTOSTART
     The autostart options support marking which containers should be
     auto-started and in what order. These options may be used by LXC
     tools directly or by external tooling provided by the distributions.
```

```
       lxc.start.auto
              Whether the container should be auto-started.  Valid values
              are 0 (off) and 1 (on).

       lxc.start.delay
              How long to wait (in seconds) after the container is started
              before starting the next one.

       lxc.start.order
              An integer used to sort the containers when auto-starting a
              series of containers at once.

       lxc.group
              A multi-value key (can be used multiple times) to put the
              container in a container group. Those groups can then be used
              (amongst other things) to start a series of related
              containers.

EXAMPLES

       In addition to the few examples given below, you will find some other
       examples of configuration file in /usr/local/share/doc/lxc/examples

   NETWORK
       This configuration sets up a container to use a veth pair device with
       one side plugged to a bridge br0 (which has been configured before on
       the system by the administrator). The virtual network device visible
       in the container is renamed to eth0.

            lxc.utsname = myhostname
            lxc.network.type = veth
            lxc.network.flags = up
            lxc.network.link = br0
            lxc.network.name = eth0
            lxc.network.hwaddr = 4a:49:43:49:79:bf
            lxc.network.ipv4 = 10.2.3.5/24 10.2.3.255
            lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3597

   UID/GID MAPPING
       This configuration will map both user and group ids in the range
       0-9999 in the container to the ids 100000-109999 on the host.

            lxc.id_map = u 0 100000 10000
            lxc.id_map = g 0 100000 10000

   CONTROL GROUP
       This configuration will setup several control groups for the
       application, cpuset.cpus restricts usage of the defined cpu,
       cpus.share prioritize the control group, devices.allow makes usable
       the specified devices.

            lxc.cgroup.cpuset.cpus = 0,1
            lxc.cgroup.cpu.shares = 1234
            lxc.cgroup.devices.deny = a
            lxc.cgroup.devices.allow = c 1:3 rw
            lxc.cgroup.devices.allow = b 8:0 rw

   COMPLEX CONFIGURATION
       This example show a complex configuration making a complex network
       stack, using the control groups, setting a new hostname, mounting
```

```
       some locations and a changing root file system.

            lxc.utsname = complex
            lxc.network.type = veth
            lxc.network.flags = up
            lxc.network.link = br0
            lxc.network.hwaddr = 4a:49:43:49:79:bf
            lxc.network.ipv4 = 10.2.3.5/24 10.2.3.255
            lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3597
            lxc.network.ipv6 = 2003:db8:1:0:214:5432:feab:3588
            lxc.network.type = macvlan
            lxc.network.flags = up
            lxc.network.link = eth0
            lxc.network.hwaddr = 4a:49:43:49:79:bd
            lxc.network.ipv4 = 10.2.3.4/24
            lxc.network.ipv4 = 192.168.10.125/24
            lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3596
            lxc.network.type = phys
            lxc.network.flags = up
            lxc.network.link = dummy0
            lxc.network.hwaddr = 4a:49:43:49:79:ff
            lxc.network.ipv4 = 10.2.3.6/24
            lxc.network.ipv6 = 2003:db8:1:0:214:1234:fe0b:3297
            lxc.cgroup.cpuset.cpus = 0,1
            lxc.cgroup.cpu.shares = 1234
            lxc.cgroup.devices.deny = a
            lxc.cgroup.devices.allow = c 1:3 rw
            lxc.cgroup.devices.allow = b 8:0 rw
            lxc.mount = /etc/fstab.complex
            lxc.mount.entry = /lib /root/myrootfs/lib none ro,bind 0 0
            lxc.rootfs = /mnt/rootfs.complex
            lxc.cap.drop = sys_module mknod setuid net_raw
            lxc.cap.drop = mac_override
SEE ALSO

       chroot(1), pivot_root(8), fstab(5), capabilities(7)

SEE ALSO

       lxc(7), lxc-create(1), lxc-destroy(1), lxc-start(1), lxc-stop(1),
       lxc-execute(1), lxc-console(1), lxc-monitor(1), lxc-wait(1),
       lxc-cgroup(1), lxc-ls(1), lxc-info(1), lxc-freeze(1),
       lxc-unfreeze(1), lxc-attach(1), lxc.conf(5)

AUTHOR

       Daniel Lezcano <daniel.lezcano@free.fr>

COLOPHON

       This page is part of the lxc (Linux containers) project.  Information
       about the project can be found at http://linuxcontainers.org/.  If
       you have a bug report for this manual page, send it to
       lxc-devel@lists.linuxcontainers.org.  This page was obtained from the
       project's upstream Git repository (git://github.com/lxc/lxc) on
       2014-05-21.  If you discover any rendering problems in this HTML
       version of the page, or you believe there is a better or more up-to-
       date source for the page, or you have corrections or improvements to
       the information in this COLOPHON (which is not part of the original
       manual page), send a mail to man-pages@man7.org
```

```
                         Wed May 21 10:30:15 CEST 2014  LXC.CONTAINER.CONF(5)
```

The text below comes from the lxc.system.conf.5 man page:

```
LXC.SYSTEM.CONF(5)                                        LXC.SYSTEM.CONF(5)
NAME

       lxc.system.conf - LXC system configuration file

DESCRIPTION

       The system configuration is located at /usr/local/etc/lxc/lxc.conf or
       ~/.config/lxc/lxc.conf for unprivileged containers.

       This configuration file is used to set values such as default lookup
       paths and storage backend settings for LXC.

   CONFIGURATION PATHS
       lxc.lxcpath
              The location in which all containers are stored.

       lxc.default_config
              The path to the default container configuration.

   CONTROL GROUPS
       lxc.cgroup.use
              Comma separated list of cgroup controllers to setup.

       lxc.cgroup.pattern
              Format string used to generate the cgroup path (e.g. lxc/%n).

   LVM
       lxc.bdev.lvm.vg
              Default LVM volume group name.

       lxc.bdev.lvm.thin_pool
              Default LVM thin pool name.

   ZFS
       lxc.bdev.zfs.root
              Default ZFS root name.
        top

       lxc(1), lxc.container.conf(5), lxc.system.conf(5)

SEE ALSO

       lxc(7), lxc-create(1), lxc-destroy(1), lxc-start(1), lxc-stop(1),
       lxc-execute(1), lxc-console(1), lxc-monitor(1), lxc-wait(1),
       lxc-cgroup(1), lxc-ls(1), lxc-info(1), lxc-freeze(1),
       lxc-unfreeze(1), lxc-attach(1), lxc.conf(5)

AUTHOR

       Stéphane Graber <stgraber@ubuntu.com>

COLOPHON
```

```
       This page is part of the lxc (Linux containers) project.  Information
       about the project can be found at http://linuxcontainers.org/.  If
       you have a bug report for this manual page, send it to
       lxc-devel@lists.linuxcontainers.org.  This page was obtained from the
       project's upstream Git repository (git://github.com/lxc/lxc) on
       2014-05-21.  If you discover any rendering problems in this HTML
       version of the page, or you believe there is a better or more up-to-
       date source for the page, or you have corrections or improvements to
       the information in this COLOPHON (which is not part of the original
       manual page), send a mail to man-pages@man7.org

                        Wed May 21 10:30:15 CEST 2014     LXC.SYSTEM.CONF(5)
```

## 19.7.2 Documentation/cgroups/cgroups.txt

The following documentation is from the Linux kernel (Documentation/cgroups/cgroups.txt)

```
				CGROUPS
				-------

Written by Paul Menage <menage@google.com> based on
Documentation/cgroups/cpusets.txt

Original copyright statements from cpusets.txt:
Portions Copyright (C) 2004 BULL SA.
Portions Copyright (c) 2004-2006 Silicon Graphics, Inc.
Modified by Paul Jackson <pj@sgi.com>
Modified by Christoph Lameter <clameter@sgi.com>

CONTENTS:
=========

1. Control Groups
  1.1 What are cgroups ?
  1.2 Why are cgroups needed ?
  1.3 How are cgroups implemented ?
  1.4 What does notify_on_release do ?
  1.5 What does clone_children do ?
  1.6 How do I use cgroups ?
2. Usage Examples and Syntax
  2.1 Basic Usage
  2.2 Attaching processes
  2.3 Mounting hierarchies by name
3. Kernel API
  3.1 Overview
  3.2 Synchronization
  3.3 Subsystem API
4. Extended attributes usage
5. Questions

1. Control Groups
=================

1.1 What are cgroups ?
----------------------

Control Groups provide a mechanism for aggregating/partitioning sets of
tasks, and all their future children, into hierarchical groups with
```

```
specialized behaviour.

Definitions:

A *cgroup* associates a set of tasks with a set of parameters for one
or more subsystems.

A *subsystem* is a module that makes use of the task grouping
facilities provided by cgroups to treat groups of tasks in
particular ways. A subsystem is typically a "resource controller" that
schedules a resource or applies per-cgroup limits, but it may be
anything that wants to act on a group of processes, e.g. a
virtualization subsystem.

A *hierarchy* is a set of cgroups arranged in a tree, such that
every task in the system is in exactly one of the cgroups in the
hierarchy, and a set of subsystems; each subsystem has system-specific
state attached to each cgroup in the hierarchy.  Each hierarchy has
an instance of the cgroup virtual filesystem associated with it.

At any one time there may be multiple active hierarchies of task
cgroups. Each hierarchy is a partition of all tasks in the system.

User-level code may create and destroy cgroups by name in an
instance of the cgroup virtual file system, specify and query to
which cgroup a task is assigned, and list the task PIDs assigned to
a cgroup. Those creations and assignments only affect the hierarchy
associated with that instance of the cgroup file system.

On their own, the only use for cgroups is for simple job
tracking. The intention is that other subsystems hook into the generic
cgroup support to provide new attributes for cgroups, such as
accounting/limiting the resources which processes in a cgroup can
access. For example, cpusets (see Documentation/cgroups/cpusets.txt) allow
you to associate a set of CPUs and a set of memory nodes with the
tasks in each cgroup.

1.2 Why are cgroups needed ?
----------------------------

There are multiple efforts to provide process aggregations in the
Linux kernel, mainly for resource-tracking purposes. Such efforts
include cpusets, CKRM/ResGroups, UserBeanCounters, and virtual server
namespaces. These all require the basic notion of a
grouping/partitioning of processes, with newly forked processes ending
up in the same group (cgroup) as their parent process.

The kernel cgroup patch provides the minimum essential kernel
mechanisms required to efficiently implement such groups. It has
minimal impact on the system fast paths, and provides hooks for
specific subsystems such as cpusets to provide additional behaviour as
desired.

Multiple hierarchy support is provided to allow for situations where
the division of tasks into cgroups is distinctly different for
different subsystems - having parallel hierarchies allows each
hierarchy to be a natural division of tasks, without having to handle
complex combinations of tasks that would be present if several
unrelated subsystems needed to be forced into the same tree of
cgroups.
```

```
At one extreme, each resource controller or subsystem could be in a
separate hierarchy; at the other extreme, all subsystems
would be attached to the same hierarchy.

As an example of a scenario (originally proposed by vatsa@in.ibm.com)
that can benefit from multiple hierarchies, consider a large
university server with various users - students, professors, system
tasks etc. The resource planning for this server could be along the
following lines:

       CPU :          "Top cpuset"
                       /       \
               CPUSet1         CPUSet2
                  |               |
               (Professors)    (Students)

               In addition (system tasks) are attached to topcpuset (so
               that they can run anywhere) with a limit of 20%

       Memory : Professors (50%), Students (30%), system (20%)

       Disk : Professors (50%), Students (30%), system (20%)

       Network : WWW browsing (20%), Network File System (60%), others (20%)
                               / \
               Professors (15%)  students (5%)

Browsers like Firefox/Lynx go into the WWW network class, while (k)nfsd goes
into the NFS network class.

At the same time Firefox/Lynx will share an appropriate CPU/Memory class
depending on who launched it (prof/student).

With the ability to classify tasks differently for different resources
(by putting those resource subsystems in different hierarchies),
the admin can easily set up a script which receives exec notifications
and depending on who is launching the browser he can

    # echo browser_pid > /sys/fs/cgroup/<restype>/<userclass>/tasks

With only a single hierarchy, he now would potentially have to create
a separate cgroup for every browser launched and associate it with
appropriate network and other resource class.  This may lead to
proliferation of such cgroups.

Also let's say that the administrator would like to give enhanced network
access temporarily to a student's browser (since it is night and the user
wants to do online gaming :))  OR give one of the student's simulation
apps enhanced CPU power.

With ability to write PIDs directly to resource classes, it's just a
matter of:

       # echo pid > /sys/fs/cgroup/network/<new_class>/tasks
       (after some time)
       # echo pid > /sys/fs/cgroup/network/<orig_class>/tasks

Without this ability, the administrator would have to split the cgroup into
multiple separate ones and then associate the new cgroups with the
```

```
new resource classes.



1.3 How are cgroups implemented ?
---------------------------------

Control Groups extends the kernel as follows:

 - Each task in the system has a reference-counted pointer to a
   css_set.

 - A css_set contains a set of reference-counted pointers to
   cgroup_subsys_state objects, one for each cgroup subsystem
   registered in the system. There is no direct link from a task to
   the cgroup of which it's a member in each hierarchy, but this
   can be determined by following pointers through the
   cgroup_subsys_state objects. This is because accessing the
   subsystem state is something that's expected to happen frequently
   and in performance-critical code, whereas operations that require a
   task's actual cgroup assignments (in particular, moving between
   cgroups) are less common. A linked list runs through the cg_list
   field of each task_struct using the css_set, anchored at
   css_set->tasks.

 - A cgroup hierarchy filesystem can be mounted for browsing and
   manipulation from user space.

 - You can list all the tasks (by PID) attached to any cgroup.

The implementation of cgroups requires a few, simple hooks
into the rest of the kernel, none in performance-critical paths:

 - in init/main.c, to initialize the root cgroups and initial
   css_set at system boot.

 - in fork and exit, to attach and detach a task from its css_set.

In addition, a new file system of type "cgroup" may be mounted, to
enable browsing and modifying the cgroups presently known to the
kernel.  When mounting a cgroup hierarchy, you may specify a
comma-separated list of subsystems to mount as the filesystem mount
options.  By default, mounting the cgroup filesystem attempts to
mount a hierarchy containing all registered subsystems.

If an active hierarchy with exactly the same set of subsystems already
exists, it will be reused for the new mount. If no existing hierarchy
matches, and any of the requested subsystems are in use in an existing
hierarchy, the mount will fail with -EBUSY. Otherwise, a new hierarchy
is activated, associated with the requested subsystems.

It's not currently possible to bind a new subsystem to an active
cgroup hierarchy, or to unbind a subsystem from an active cgroup
hierarchy. This may be possible in future, but is fraught with nasty
error-recovery issues.

When a cgroup filesystem is unmounted, if there are any
child cgroups created below the top-level cgroup, that hierarchy
will remain active even though unmounted; if there are no
child cgroups then the hierarchy will be deactivated.
```

```
No new system calls are added for cgroups - all support for
querying and modifying cgroups is via this cgroup file system.

Each task under /proc has an added file named 'cgroup' displaying,
for each active hierarchy, the subsystem names and the cgroup name
as the path relative to the root of the cgroup file system.

Each cgroup is represented by a directory in the cgroup file system
containing the following files describing that cgroup:

 - tasks: list of tasks (by PID) attached to that cgroup.  This list
   is not guaranteed to be sorted.  Writing a thread ID into this file
   moves the thread into this cgroup.
 - cgroup.procs: list of thread group IDs in the cgroup.  This list is
   not guaranteed to be sorted or free of duplicate TGIDs, and userspace
   should sort/uniquify the list if this property is required.
   Writing a thread group ID into this file moves all threads in that
   group into this cgroup.
 - notify_on_release flag: run the release agent on exit?
 - release_agent: the path to use for release notifications (this file
   exists in the top cgroup only)

Other subsystems such as cpusets may add additional files in each
cgroup dir.

New cgroups are created using the mkdir system call or shell
command.  The properties of a cgroup, such as its flags, are
modified by writing to the appropriate file in that cgroups
directory, as listed above.

The named hierarchical structure of nested cgroups allows partitioning
a large system into nested, dynamically changeable, "soft-partitions".

The attachment of each task, automatically inherited at fork by any
children of that task, to a cgroup allows organizing the work load
on a system into related sets of tasks.  A task may be re-attached to
any other cgroup, if allowed by the permissions on the necessary
cgroup file system directories.

When a task is moved from one cgroup to another, it gets a new
css_set pointer - if there's an already existing css_set with the
desired collection of cgroups then that group is reused, otherwise a new
css_set is allocated. The appropriate existing css_set is located by
looking into a hash table.

To allow access from a cgroup to the css_sets (and hence tasks)
that comprise it, a set of cg_cgroup_link objects form a lattice;
each cg_cgroup_link is linked into a list of cg_cgroup_links for
a single cgroup on its cgrp_link_list field, and a list of
cg_cgroup_links for a single css_set on its cg_link_list.

Thus the set of tasks in a cgroup can be listed by iterating over
each css_set that references the cgroup, and sub-iterating over
each css_set's task set.

The use of a Linux virtual file system (vfs) to represent the
cgroup hierarchy provides for a familiar permission and name space
for cgroups, with a minimum of additional kernel code.
```

```
1.4 What does notify_on_release do ?
------------------------------------

If the notify_on_release flag is enabled (1) in a cgroup, then
whenever the last task in the cgroup leaves (exits or attaches to
some other cgroup) and the last child cgroup of that cgroup
is removed, then the kernel runs the command specified by the contents
of the "release_agent" file in that hierarchy's root directory,
supplying the pathname (relative to the mount point of the cgroup
file system) of the abandoned cgroup.  This enables automatic
removal of abandoned cgroups.  The default value of
notify_on_release in the root cgroup at system boot is disabled
(0).  The default value of other cgroups at creation is the current
value of their parents' notify_on_release settings. The default value of
a cgroup hierarchy's release_agent path is empty.

1.5 What does clone_children do ?
---------------------------------

This flag only affects the cpuset controller. If the clone_children
flag is enabled (1) in a cgroup, a new cpuset cgroup will copy its
configuration from the parent during initialization.

1.6 How do I use cgroups ?
--------------------------

To start a new job that is to be contained within a cgroup, using
the "cpuset" cgroup subsystem, the steps are something like:

 1) mount -t tmpfs cgroup_root /sys/fs/cgroup
 2) mkdir /sys/fs/cgroup/cpuset
 3) mount -t cgroup -ocpuset cpuset /sys/fs/cgroup/cpuset
 4) Create the new cgroup by doing mkdir's and write's (or echo's) in
    the /sys/fs/cgroup virtual file system.
 5) Start a task that will be the "founding father" of the new job.
 6) Attach that task to the new cgroup by writing its PID to the
    /sys/fs/cgroup/cpuset/tasks file for that cgroup.
 7) fork, exec or clone the job tasks from this founding father task.

For example, the following sequence of commands will setup a cgroup
named "Charlie", containing just CPUs 2 and 3, and Memory Node 1,
and then start a subshell 'sh' in that cgroup:

  mount -t tmpfs cgroup_root /sys/fs/cgroup
  mkdir /sys/fs/cgroup/cpuset
  mount -t cgroup cpuset -ocpuset /sys/fs/cgroup/cpuset
  cd /sys/fs/cgroup/cpuset
  mkdir Charlie
  cd Charlie
  /bin/echo 2-3 > cpuset.cpus
  /bin/echo 1 > cpuset.mems
  /bin/echo $$ > tasks
  sh
  # The subshell 'sh' is now running in cgroup Charlie
  # The next line should display '/Charlie'
  cat /proc/self/cgroup

2. Usage Examples and Syntax
============================
```

```
2.1 Basic Usage
---------------

Creating, modifying, using cgroups can be done through the cgroup
virtual filesystem.

To mount a cgroup hierarchy with all available subsystems, type:
# mount -t cgroup xxx /sys/fs/cgroup

The "xxx" is not interpreted by the cgroup code, but will appear in
/proc/mounts so may be any useful identifying string that you like.

Note: Some subsystems do not work without some user input first.  For instance,
if cpusets are enabled the user will have to populate the cpus and mems files
for each new cgroup created before that group can be used.

As explained in section `1.2 Why are cgroups needed?' you should create
different hierarchies of cgroups for each single resource or group of
resources you want to control. Therefore, you should mount a tmpfs on
/sys/fs/cgroup and create directories for each cgroup resource or resource
group.

# mount -t tmpfs cgroup_root /sys/fs/cgroup
# mkdir /sys/fs/cgroup/rg1

To mount a cgroup hierarchy with just the cpuset and memory
subsystems, type:
# mount -t cgroup -o cpuset,memory hier1 /sys/fs/cgroup/rg1

While remounting cgroups is currently supported, it is not recommend
to use it. Remounting allows changing bound subsystems and
release_agent. Rebinding is hardly useful as it only works when the
hierarchy is empty and release_agent itself should be replaced with
conventional fsnotify. The support for remounting will be removed in
the future.

To Specify a hierarchy's release_agent:
# mount -t cgroup -o cpuset,release_agent="/sbin/cpuset_release_agent" \
  xxx /sys/fs/cgroup/rg1

Note that specifying 'release_agent' more than once will return failure.

Note that changing the set of subsystems is currently only supported
when the hierarchy consists of a single (root) cgroup. Supporting
the ability to arbitrarily bind/unbind subsystems from an existing
cgroup hierarchy is intended to be implemented in the future.

Then under /sys/fs/cgroup/rg1 you can find a tree that corresponds to the
tree of the cgroups in the system. For instance, /sys/fs/cgroup/rg1
is the cgroup that holds the whole system.

If you want to change the value of release_agent:
# echo "/sbin/new_release_agent" > /sys/fs/cgroup/rg1/release_agent

It can also be changed via remount.

If you want to create a new cgroup under /sys/fs/cgroup/rg1:
# cd /sys/fs/cgroup/rg1
# mkdir my_cgroup
```

```
Now you want to do something with this cgroup.
# cd my_cgroup

In this directory you can find several files:
# ls
cgroup.procs notify_on_release tasks
(plus whatever files added by the attached subsystems)

Now attach your shell to this cgroup:
# /bin/echo $$ > tasks

You can also create cgroups inside your cgroup by using mkdir in this
directory.
# mkdir my_sub_cs

To remove a cgroup, just use rmdir:
# rmdir my_sub_cs

This will fail if the cgroup is in use (has cgroups inside, or
has processes attached, or is held alive by other subsystem-specific
reference).

2.2 Attaching processes
-----------------------

# /bin/echo PID > tasks

Note that it is PID, not PIDs. You can only attach ONE task at a time.
If you have several tasks to attach, you have to do it one after another:

# /bin/echo PID1 > tasks
# /bin/echo PID2 > tasks
    ...
# /bin/echo PIDn > tasks

You can attach the current shell task by echoing 0:

# echo 0 > tasks

You can use the cgroup.procs file instead of the tasks file to move all
threads in a threadgroup at once. Echoing the PID of any task in a
threadgroup to cgroup.procs causes all tasks in that threadgroup to be
attached to the cgroup. Writing 0 to cgroup.procs moves all tasks
in the writing task's threadgroup.

Note: Since every task is always a member of exactly one cgroup in each
mounted hierarchy, to remove a task from its current cgroup you must
move it into a new cgroup (possibly the root cgroup) by writing to the
new cgroup's tasks file.

Note: Due to some restrictions enforced by some cgroup subsystems, moving
a process to another cgroup can fail.

2.3 Mounting hierarchies by name
--------------------------------

Passing the name=<x> option when mounting a cgroups hierarchy
associates the given name with the hierarchy.  This can be used when
mounting a pre-existing hierarchy, in order to refer to it by name
rather than by its set of active subsystems.  Each hierarchy is either
```

```
nameless, or has a unique name.

The name should match [\w.-]+

When passing a name=<x> option for a new hierarchy, you need to
specify subsystems manually; the legacy behaviour of mounting all
subsystems when none are explicitly specified is not supported when
you give a subsystem a name.

The name of the subsystem appears as part of the hierarchy description
in /proc/mounts and /proc/<pid>/cgroups.


3. Kernel API
=============

3.1 Overview
------------

Each kernel subsystem that wants to hook into the generic cgroup
system needs to create a cgroup_subsys object. This contains
various methods, which are callbacks from the cgroup system, along
with a subsystem ID which will be assigned by the cgroup system.

Other fields in the cgroup_subsys object include:

- subsys_id: a unique array index for the subsystem, indicating which
  entry in cgroup->subsys[] this subsystem should be managing.

- name: should be initialized to a unique subsystem name. Should be
  no longer than MAX_CGROUP_TYPE_NAMELEN.

- early_init: indicate if the subsystem needs early initialization
  at system boot.

Each cgroup object created by the system has an array of pointers,
indexed by subsystem ID; this pointer is entirely managed by the
subsystem; the generic cgroup code will never touch this pointer.

3.2 Synchronization
-------------------

There is a global mutex, cgroup_mutex, used by the cgroup
system. This should be taken by anything that wants to modify a
cgroup. It may also be taken to prevent cgroups from being
modified, but more specific locks may be more appropriate in that
situation.

See kernel/cgroup.c for more details.

Subsystems can take/release the cgroup_mutex via the functions
cgroup_lock()/cgroup_unlock().

Accessing a task's cgroup pointer may be done in the following ways:
- while holding cgroup_mutex
- while holding the task's alloc_lock (via task_lock())
- inside an rcu_read_lock() section via rcu_dereference()

3.3 Subsystem API
-----------------
```

```
Each subsystem should:

- add an entry in linux/cgroup_subsys.h
- define a cgroup_subsys object called <name>_subsys

If a subsystem can be compiled as a module, it should also have in its
module initcall a call to cgroup_load_subsys(), and in its exitcall a
call to cgroup_unload_subsys(). It should also set its_subsys.module =
THIS_MODULE in its .c file.

Each subsystem may export the following methods. The only mandatory
methods are css_alloc/free. Any others that are null are presumed to
be successful no-ops.

struct cgroup_subsys_state *css_alloc(struct cgroup *cgrp)
(cgroup_mutex held by caller)

Called to allocate a subsystem state object for a cgroup. The
subsystem should allocate its subsystem state object for the passed
cgroup, returning a pointer to the new object on success or a
ERR_PTR() value. On success, the subsystem pointer should point to
a structure of type cgroup_subsys_state (typically embedded in a
larger subsystem-specific object), which will be initialized by the
cgroup system. Note that this will be called at initialization to
create the root subsystem state for this subsystem; this case can be
identified by the passed cgroup object having a NULL parent (since
it's the root of the hierarchy) and may be an appropriate place for
initialization code.

int css_online(struct cgroup *cgrp)
(cgroup_mutex held by caller)

Called after @cgrp successfully completed all allocations and made
visible to cgroup_for_each_child/descendant_*() iterators. The
subsystem may choose to fail creation by returning -errno. This
callback can be used to implement reliable state sharing and
propagation along the hierarchy. See the comment on
cgroup_for_each_descendant_pre() for details.

void css_offline(struct cgroup *cgrp);
(cgroup_mutex held by caller)

This is the counterpart of css_online() and called iff css_online()
has succeeded on @cgrp. This signifies the beginning of the end of
@cgrp. @cgrp is being removed and the subsystem should start dropping
all references it's holding on @cgrp. When all references are dropped,
cgroup removal will proceed to the next step - css_free(). After this
callback, @cgrp should be considered dead to the subsystem.

void css_free(struct cgroup *cgrp)
(cgroup_mutex held by caller)

The cgroup system is about to free @cgrp; the subsystem should free
its subsystem state object. By the time this method is called, @cgrp
is completely unused; @cgrp->parent is still valid. (Note - can also
be called for a newly-created cgroup if an error occurs after this
subsystem's create() method has been called for the new cgroup).

int can_attach(struct cgroup *cgrp, struct cgroup_taskset *tset)
```

```
(cgroup_mutex held by caller)

Called prior to moving one or more tasks into a cgroup; if the
subsystem returns an error, this will abort the attach operation.
@tset contains the tasks to be attached and is guaranteed to have at
least one task in it.

If there are multiple tasks in the taskset, then:
  - it's guaranteed that all are from the same thread group
  - @tset contains all tasks from the thread group whether or not
    they're switching cgroups
  - the first task is the leader

Each @tset entry also contains the task's old cgroup and tasks which
aren't switching cgroup can be skipped easily using the
cgroup_taskset_for_each() iterator. Note that this isn't called on a
fork. If this method returns 0 (success) then this should remain valid
while the caller holds cgroup_mutex and it is ensured that either
attach() or cancel_attach() will be called in future.

void cancel_attach(struct cgroup *cgrp, struct cgroup_taskset *tset)
(cgroup_mutex held by caller)

Called when a task attach operation has failed after can_attach() has succeeded.
A subsystem whose can_attach() has some side-effects should provide this
function, so that the subsystem can implement a rollback. If not, not necessary.
This will be called only about subsystems whose can_attach() operation have
succeeded. The parameters are identical to can_attach().

void attach(struct cgroup *cgrp, struct cgroup_taskset *tset)
(cgroup_mutex held by caller)

Called after the task has been attached to the cgroup, to allow any
post-attachment activity that requires memory allocations or blocking.
The parameters are identical to can_attach().

void fork(struct task_struct *task)

Called when a task is forked into a cgroup.

void exit(struct task_struct *task)

Called during task exit.

void bind(struct cgroup *root)
(cgroup_mutex held by caller)

Called when a cgroup subsystem is rebound to a different hierarchy
and root cgroup. Currently this will only involve movement between
the default hierarchy (which never has sub-cgroups) and a hierarchy
that is being created/destroyed (and hence has no sub-cgroups).

4. Extended attribute usage
===========================

cgroup filesystem supports certain types of extended attributes in its
directories and files.  The current supported types are:
    - Trusted (XATTR_TRUSTED)
    - Security (XATTR_SECURITY)
```

```
Both require CAP_SYS_ADMIN capability to set.

Like in tmpfs, the extended attributes in cgroup filesystem are stored
using kernel memory and it's advised to keep the usage at minimum.  This
is the reason why user defined extended attributes are not supported, since
any user can do it and there's no limit in the value size.

The current known users for this feature are SELinux to limit cgroup usage
in containers and systemd for assorted meta data like main PID in a cgroup
(systemd creates a cgroup per service).

5. Questions
============

Q: what's up with this '/bin/echo' ?
A: bash's builtin 'echo' command does not check calls to write() against
   errors. If you use it in the cgroup file system, you won't be
   able to tell whether a command succeeded or failed.

Q: When I attach processes, only the first of the line gets really attached !
A: We can only return one error code per call to write(). So you should also
   put only ONE PID.
```

Linux Ethernet Driver for DPAA 1.x Family

# Chapter 20
# Linux Ethernet Driver for DPAA 1.x Family

## 20.1 Linux Ethernet Driver for DPAA 1.x Family

### 20.1.1 Introduction

This document describes the Linux drivers which enable support for Ethernet on processors with the Datapath Acceleration Architecture (DPAA). The focus is on the theory and operation behind using Ethernet. It provides only limited discussion of the BMan, QMan, and FMan, describing instead the layer of software which allows all of these to interoperate. For the purposes of this document, all these drivers will be referred to as the DPAA Ethernet Drivers. Enablement, configuration and debugging for all DPAA Ethernet Drivers are described in this document.

#### Purpose

The DPAA Ethernet Driver is meant to manage the use of the Datapath hardware for communication via the Ethernet protocol. This includes facilities for:

- Allocating buffer pools and buffers
- Allocating frame queues
- Assigning frame queues and buffer pools to specified FMan ports
- Transferring packets between frame queues and the Linux stack
- Controlling Link Management features

#### Overview

Ethernet in the Datapath is realized by interconnecting BMan, QMan, and FMan. The primary interaction for the DPAA Ethernet Driver is between the kernel and the QMan. Ethernet frames are delivered to the driver from the frame queue via the QMan portal, and the driver delivers Ethernet frames to the outgoing frame queue via the QMan portal. For some use cases, that is the only interaction.

Usually, the frame queues are connected to a FMan port. Each FMan port has two queues which must be assigned to them: a default queue and an error queue. This assignment can be specified in the device tree, or created dynamically by the driver on initialization.

The Ethernet frames are often stored in buffers which are associated with a BMan buffer pool. The driver sets up this pool, and either seeds it with buffers, or maps the buffers which are put into the pool. Depending on the pool, the buffers may be allocated and freed by the kernel during network activity, or they may be allocated once, and return to the pool when not in use by the Datapath hardware.

#### DPAA Ethernet Driver types

The complexity of DPAA allows a variety of possible use cases. Although speed is the key factor for performance in most use cases, customization and usability are preferred in others. Building a single Ethernet driver to cover all these tasks was, in the ever-changing world of the Linux kernel, a task that was becoming harder day-by-day. Instead we divided the single basic block into several specialized and simpler Ethernet drivers which can be combined in complex configurations:

- The Private DPAA Ethernet Driver resembles the common Linux Ethernet driver. It is highly improved for performance and uses all the features that DPAA offers;

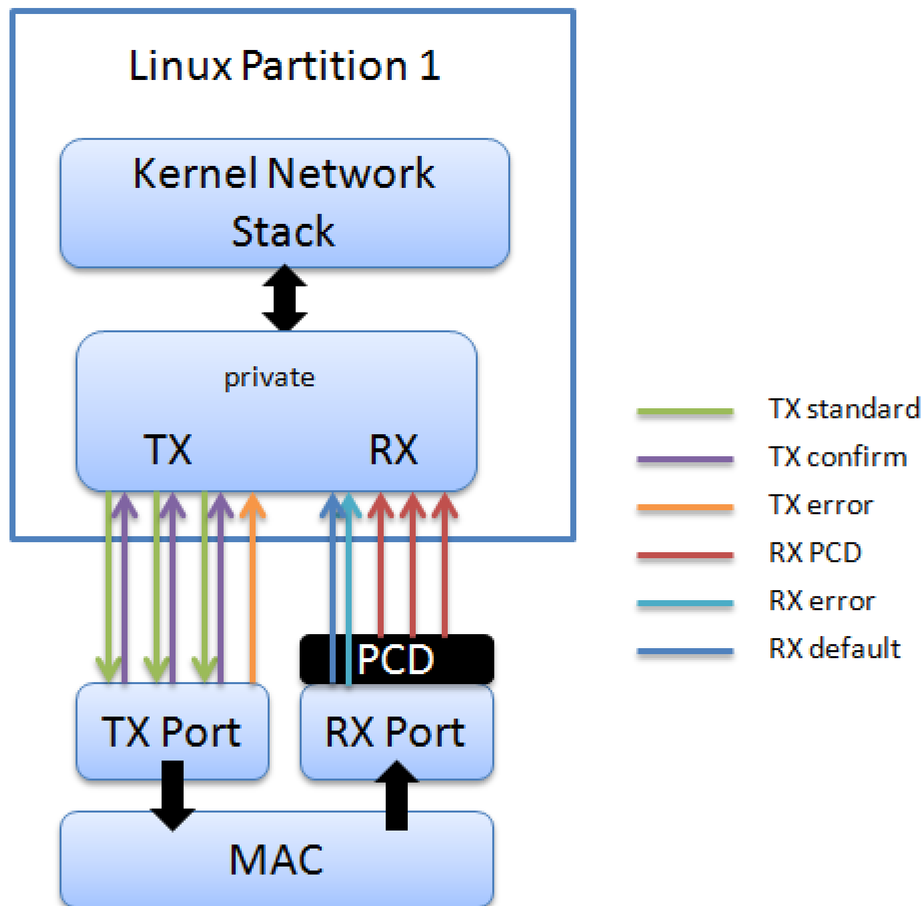
Linux Ethernet Driver for DPAA 1.x Family

- The Macless DPAA Ethernet Driver and the Shared DPAA Ethernet Driver flavors are used in virtualized multi-partition scenarios (with the Topaz Hypervizor) or in custom user-space traffic analyzer use cases;
- The Proxy DPAA Ethernet Driver will do the entire preliminary work in scenarios where all the control is passed to user-space, bypassing the standard Linux kernel standard.
- The Offload NIC Ethernet Driver is similar to the Macless Driver and can be used in the same scenarios. In addition, it has offloading capabilities that the Macless Driver lacks, and uses Offline Parsing/Host Command Ports.
- The MACsec Driver is used to configure the FMan MACsec hardware block that is capable of offloading the IEEE 802.1AE's protocol features.

## 20.1.2 Private DPAA Ethernet Driver

The Private DPAA Ethernet Driver manages the network interfaces which are fully owned by the Linux partition who runs them. Therefore, it is possible to take advantage of the DPAA facilities in order to increase the performance in both termination and forwarding scenarios.

The Private DPAA Ethernet Driver will be further referenced as the Private Driver.




### 20.1.2.1 Configuration

This chapter present the configuration options for the Private DPAA Ethernet Driver.

Linux Ethernet Driver for DPAA 1.x Family

#### 20.1.2.1.1 Device Tree Configuration

The compatible string used to define a private interface in device tree is „fsl,dpa-ethernet”. The default structure for the device tree node that specifies a Private interface should be similar to the below snippet of a B4860QDS device tree node:

```
ethernet@4 {
        compatible = "fsl,b4860-dpa-ethernet", "fsl,dpa-ethernet";
        fsl,fman-mac = <&fm1mac5>;
};
```

“fsl,fman-mac” is the reference to the MAC device connected to this interface. This property is used to determine which RX and TX ports are connected to this interface.

**Buffer pools**

A single buffer pool is currently defined and used by all the Private interfaces. The buffer pool ID is dynamically allocated and provided by the Buffer Manager. The number and size of the buffers in the pool is decided internally by the Private driver therefore no device tree configuration is accepted.

**Frame queues**

The frame queues are allocated by the Private driver with IDs dynamically allocated and provided by the Queue Manager. The frame queues can also be statically defined using two additional device tree properties:

```
ethernet@0 {
        compatible = "fsl,b4860-dpa-ethernet", "fsl,dpa-ethernet";
        fsl,fman-mac = <&fm1mac5>;
        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.

#### 20.1.2.1.2 Bootargs

Two bootarg parameters are defined for the Frame Manager driver but they also influence the behavior of the Private driver:

- fsl_fm_max_frm
- fsl_fm_rx_extra_headroom

**fsl_fm_max_frm**

The Frame Manager discards both Rx and Tx frames that are larger than a specific Layer2 MAXFRM value. The DPAA Ethernet driver won't allow one to set an interface’s MTU too high such that it would produce Ethernet frames larger than MAXFRM. The maximum value one can use as the MTU for any interface is (MAXFRM - 22) bytes, where 22 is the size of an Eth+VLAN header (18 bytes), plus the Layer2 FCS (4 bytes).

Linux Ethernet Driver for DPAA 1.x Family

Currently, the value of MAXFRM is set at boot-time and cannot be changed without rebooting the system.

The default MAXFRM is 1522, allowing for MTUs up to 1500. If a larger MTU is desired, one would have to reboot and reconfigure the system as described next. The maximum MAXFRM is 9600

*The MAXFRM can be set in two ways:*

- as a Kconfig option (CONFIG_FSL_FM_MAX_FRAME_SIZE):

```
Device Drivers
+-> Network device support
  +-> Ethernet driver support
    +-> Freescale devices
       +-> Frame Manager support
          +-> Freescale Frame Manager (datapath) support
            +-> Maximum L2 frame size
```

- as a bootarg:
  - In no-HV scenarios: In the u-boot environment, add "fsl_fm_max_frm=<your_MAXFRM>" directly to the "bootargs" variable.
  - In Hypervisor-based scenarios: Add or modify the "chosen" node in Hypervisor device tree having a "bootargs" property specifying "fsl_fm_max_frm=<your_MAXFRM>;".

Note that any value set directly in the kernel bootargs will override the Kconfig default. If not explicitly set in the bootargs, the Kconfig value will be used.

*Symptoms of Misconfigured MAXFRM*

MAXFRM directly influences the partitioning of FMan's internal MURAM among the available Ethernet ports, because it determines the value of an FMan internal parameter called "FIFO Size". Depending on the value of MAXFRM and the number of ports being probed, some of these may not be probed because there is not enough MURAM for all of them. In such cases, one will see an error message in the boot console.

#### fsl_fm_rx_extra_headroom

Configure this to tell the Frame Manager to reserve some extra space at the beginning of a data buffer on the receive path, before Internal Context fields are copied. This is in addition to the private data area already reserved for driver internal use. The option does not affect in any way the layout of transmitted buffers. The default value (64bytes) offers best performance for the case when forwarded frames are being encapsulated (e.g. IPSec).

The RX extra headroom can be set in two ways:

- as a Kconfig option (CONFIG_FSL_FM_RX_EXTRA_HEADROOM):

```
Device Drivers
+-> Network device support
  +-> Ethernet driver support
    +-> Freescale devices
      +-> Frame Manager support
        +-> Freescale Frame Manager (datapath) support
          +-> Add extra headroom at beginning of data buffers
```

- as a bootarg:
  - In no-HV scenarios: In the u-boot environment, add "fsl_fm_rx_extra_headroom=< your_rx_extra_headroom>" directly to the "bootargs" variable.
  - In Hypervisor-based scenarios: Add or modify the "chosen" node in Hypervisor device tree having a "bootargs" property specifying " fsl_fm_rx_extra_headroom=<your_rx_extra_headroom>;".

Linux Ethernet Driver for DPAA 1.x Family

### 20.1.2.1.3 Kconfig Options

The private driver has a number of parameters which can be tuned at compile time from menuconfig. These can be found in:

```
Device Drivers
    +- Network Device Support
        +- Ethernet Driver Support
            +- Freescale Devices
                +- DPAA Ethernet
```

**FSL_DPAA_ETH_JUMBO_FRAME - "Optimize for jumbo frames"**

Optimizes the DPAA Ethernet driver throughput for large frames termination traffic (e.g. 4K and above).

On FMan v2 platforms this option requires modifications to the device tree, otherwise the ping will not work. Include 'qoriq-fman-0-chosen-fifo-resize.dtsi' in 'qoriq-fman-0.dtsi' and 'qoriq-fman-1-chosen-fifo-resize.dtsi' in 'qoriq-fman-1.dtsi'

```
/include/ "qoriq-fman-0-chosen-fifo-resize.dtsi"
```

Using this option in combination with small frames increases significantly the driver's memory footprint and may even deplete the system memory.

**FSL_DPAA_TS - "Linux compliant timestamping"**

Enables IEEE1588 support code.

**FSL_DPAA_TS - "Linux compliant timestamping"**

Enables Linux API compliant timestamping support.

**FSL_DPAA_ETH_WQ_ASSIGN - "WorkQueue assignment scheme for FrameQueues"**

Selects the FrameQueue to WorkQueue assignment scheme.

**FSL_DPAA_ETH_WQ_LEGACY - "Legacy WQ assignment"**

Statically-defined FQIDs are round-robin assigned to all WQs (0..7). PCD queues are always in this category. Other frame queues may be those used for "MAC-less" or "shared MAC" configurations of the driver. Dynamically-defined FQIDs all go to WQ7.

**FSL_DPAA_ETH_WQ_MULTI - "Multi-WQ assignment"**

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

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

The DPAA-Ethernet driver defines a ndo_select_queue() callback for optimal selection of the egress FQ. That will override the XPS support for this netdevice. If for whatever reason you want to be in control of the egress FQ-to-CPU selection and mapping, or simply don't want to use the driver's ndo_select_queue() callback, then unselect this and use the standard XPS support instead.

**FSL_DPAA_ETH_MAX_BUF_COUNT - "Maximum nuber of buffers in private bpool"**

Defaults to "128". The maximum number of buffers to be by default allocated in the DPAA-Ethernet private port's buffer pool. One needn't normally modify this, as it has probably been tuned for performance already. This cannot be lower than DPAA_ETH_REFILL_THRESHOLD.

Linux Ethernet Driver for DPAA 1.x Family

**FSL_DPAA_ETH_REFILL_THRESHOLD - "Private bpool refill threshold"**

Defaults to "128". The maximum number of buffers to be by default allocated in the DPAA-Ethernet private port's buffer pool. One needn't normally modify this, as it has probably been tuned for performance already. This cannot be lower than DPAA_ETH_REFILL_THRESHOLD.

**FSL_DPAA_CS_THRESHOLD_1G - "Egress congestion threshold on 1G ports"**

The size in bytes of the egress Congestion State notification threshold on 1G ports. Ranges from 0x1000 to 0x10000000. Defaults to 0x06000000. This option can help when:

- the device stays congested for a prolonged time (risking the netdev watchdog to fire - see also the tx_timeout module param)
- preventing the Tx cores from tightly-looping (as if the congestion threshold was too low to be effective)

This might also implies some risks:

- affecting performance of protocols such as TCP, which otherwise behave well under the congestion notification mechanism
- running out of memory if the CS threshold is set too high

**FSL_DPAA_CS_THRESHOLD_10G - "Egress congestion threshold on 10G ports"**

The size in bytes of the egress Congestion State notification threshold on 10G ports. Ranges from 0x1000 to 0x20000000. Defaults to 0x10000000.

**FSL_DPAA_INGRESS_CS_THRESHOLD - "Ingress congestion threshold on FMan ports"**

The size in bytes of the ingress tail-drop threshold on FMan ports. Defaults to 0x10000000. Traffic piling up above this value will be rejected by QMan and discarded by FMan.

**FSL_DPAA_ETH_DEBUGFS - "DPAA Ethernet debugfs interface"**

This option compiles debugfs code for the DPAA Ethernet driver.

**FSL_DPAA_ETH_DEBUG - "DPAA Ethernet Debug Support"**

This option compiles debug code for the DPAA Ethernet driver.

## 20.1.2.1.4 ethtool Options

The private driver implements the following ethtool operations

```
-a --show-pause
    Queries the specified Ethernet device for pause parameter information.
-A --pause
    Changes the pause parameters of the specified private devices.
    rx on|off
        Specifies whether RX pause should be enabled.
    tx on|off
        Specifies whether TX pause should be enabled.
-g --show-ring
    The DPAA driver does not control rings effectively, so all the values returned are
0.
-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.
```

Linux Ethernet Driver for DPAA 1.x Family

## 20.1.2.2 Features

This chapter present the Private DPAA Ethernet Driver features.

### 20.1.2.2.1 Congestion Management

QMan offers 3 methods of managing congestion:

- WRED
- congestion state tail drop (CSTD)
- FQ tail drop (FQTD)

The Private driver implements CSTD both on TX and on RX. When the number of bytes residing in a TX FQ congestion group reaches a congestion threshold (high watermark), the QMan rejects any further incoming frames, until the sum of all the frames contained in the congestion groups drops under a low watermark, which is 7/8 of the high watermark. The high watermark can be configured from menuconfig. See section "Kconfig options" for more details.

### 20.1.2.2.2 Scatter/Gather Support

On the Rx path, the first S/G entry is used to build the skb linear part and the other entries are used as fragments.

The Private driver can access the egress skbufs allocated in high memory (e.g. mapped directly from user-space, as is the case of the sendfile() system call). This eliminates the kernel need to copy such skbufs into newly-allocated low memory buffers, allowing zero-copy on the egress path.

#### 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 accomodate 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 (EXPERIMENTAL)
```

In addition to enabling this feature from menuconfig, the user is required to set fsl_fm_max_frm=9600 in the bootargs, otherwise the configuration is not valid.

### 20.1.2.2.3 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.

Linux Ethernet Driver for DPAA 1.x Family

GRO is commonly supported in hardware as a set of "GRO assists", rather than full packet coalescing. The following features count as GRO assists:

- RX hardware checksum validation
- Receive Traffic Distribution (RTD)
- Multiple RX/TX queues
- Receive Traffic Hashing
- Header prefetching
- Header separation
- Core affinity
- Interrupt affinity

Note: With the exception of header separation, the DPAA platforms feature all other hardware assists. Most notably, they are implicitly achieved through the mechanisms that accompany PCDs.

Generic Segmentation Offload (GSO) is also a well-established feature in the Linux kernel. Normally, a TCP segment is composed in the Layer 4 of the Linux stack, based on the current MSS (Maximum Segment Size) connection setting. It has been observed, though, that delaying segmentation is a better approach in terms of CPU load, because fewer headers are processed. Linux has taken an optimization approach, called 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 don't support this. Ethtool version 3.0 does - and possibly others before it, too.

Generic optimizations that enhance the driver's performance in the general case also apply to the GRO/GSO-enabled driver. PCD support is therefore recommended in this regard. We have found that these optimizations yield the best results on 10Gbps traffic, and to a lesser extent (if any) on 1Gbps traffic. TCP tests, especially, can benefit from GRO by shedding CPU load and upping the network throughput. The improvements are the more visible with smaller network MTU - with MTU=1500 and below, the benefits are higher, while starting from MTU=4k they are no longer observable.

One optimization that boosts GSO performance is the zero-copy egress path. That is available thanks to the *sendfile()* system call, which may be used instead of the plain *send()* syscall, and which certain benchmark applications know about. Netperf for instance has *sendfile* support in its *TCP_SENDFILE* tests.

GRO and GSO are no panacea, one-button-fix-all kind of optimization. While under most circumstances they should be transparent (this being why GRO is by default enabled in the Linux kernel), there are scenarios and configurations where they may in fact under-perform. Traffic on 1Gbps ports sees little benefit from GRO/GSO. GRO on P1023 behaves worse. Also, if the Private Driver detects that PCDs are not in place, GRO is automatically by-passed.

## 20.1.2.2.4 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.

Linux Ethernet Driver for DPAA 1.x Family

The Private driver, however, matches the two realms by mapping the DPAA FQs onto kernel's own queue structures. To that end, the Private driver provides a standard callback (net-device operation, or NDO) called ndo_select_queue(), which the stack can interrogate to find out the specific queue mapping it needs for transmitting a frame. The existence of that NDO (which is otherwise optional) overrides the kernel queue selection via XPS. This is why the Private driver provides a compile-time choice to disable the ndo_select_queue() callback, leaving it to the stack to choose a transmission queue.

To use the Private driver's builtin ndo_select_queue() callback, select the Kconfig option FSL_DPAA_ETH_USE_NDO_SELECT_QUEUE.

To disable the Private driver's queue selection mechanism and use XPS instead, unselect this Kconfig option. Further on, the users can configure their own txq-to-cpu mapping, as described in the LWN article above.

## 20.1.2.2.5 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 17: 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.

Linux Ethernet Driver for DPAA 1.x Family

## 20.1.2.2.6 Pause Frames Flow Control

FMan supports IEEE 802.3x flow control. Whenever the FMan RX FIFO threshold is exceeded, FMan transmits PAUSE frames to the other peer on the link. In Linux, the transmission and reception of PAUSE frames can be enabled or disabled using ethool.

To display PAUSE frames settings in use for an interface

```
ethtool -a intf_name
```

### Triggering PAUSE frames ON/OFF

PAUSE frames can be enabled/disabled on RX/TX using ethtool -A, like in the following examples

```
ethtool -A intf_name rx on
ethtool -A intf_name tx off
ethtool -A intf_name rx off tx off
```

### Autonegotiation

Starting with SDK 1.6, the DPAA Private driver supports PAUSE frame autonegotiation.

When autonegotiation is enabled and the user enables/disables PAUSE frames on RX/TX, these will not automatically be triggered on/off. Instead, the local and the peer PAUSE symmetric/asymmetric capabilities will be considered. If the peer does not match the local capabilities, the following commands may have no effect:

```
ethtool -A intf_name rx on
ethtool -A intf_name rx off
ethtool -A intf_name tx on
ethtool -A intf_name tx ff
```

When autonegotiation is disabled, ethtool settings override the result of link negotiation.

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

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

### FMAN v3 platforms

On the following platforms: T4, B4, and T1040, 802.1Qbb Priority Flow Control is used instead of 802.3x PAUSE frames. The ethtool controls are the same, but the structure of the frames is different.

Find about DPAA Private support of PFC in the following section:

Priority Flow Control on page 283

Linux Ethernet Driver for DPAA 1.x Family

#### 20.1.2.2.7 Priority Flow Control

Beginning with SDK 1.6, the DPAA Ethernet Driver offers experimental support for IEEE standards 802.1Qbb (Priority Flow Control) and 802.1p.

These standards aim to implement lossless Ethernet, in which the highest-priority classes of traffic benefit from maximum bandwidth and minimum delay. Up to 8 classes of service can be used, but only a minimum of 3 is required.

The terms “Class of Service (CoS)” and “priority” will be used interchangeably in this section.

802.1Qbb PFC frames are available only on platforms with FMan v3, namely T4, B4 and T1040. For the other platforms 802.3x PAUSE frames are used instead for Ethernet flow control.

##### Enabling PFC Support

To enable PFC support, enable the following options from menuconfig

```
Device Drivers
+ Network device support
  + Ethernet driver support
    + Freescale devices
      + Frame Manager support
        + Freescale Frame Manager (datapath) support
          + FMan PFC support (EXPERIMENTAL)
            + (3)     Number of PFC Classes of Service
            + (65535) The pause quanta for PFC CoS 0
            + (65535) The pause quanta for PFC CoS 1
            + (65535) The pause quanta for PFC CoS 2
```

The number of Classes of Service can range between 1 and 4. It defines the number of Work Queues used and the number of priorities that are set when a PFC frame is issued. 3 is the default value. Changing this value also changes the number of WQs and priorities.

The pause time can be adjusted for each CoS individually.

Enabling and disabling CoS and their pause time is unavailable at runtime. It is only possible at compile time in this release.

##### Selecting the Class of Service

When PFC support is enabled, the egress traffic flowing on a DPAA Private interface is distributed on the first 3 Work Queues of a TX port, namely WQ0, WQ1 and WQ2.

These function in strict priority. WQ0 has the highest priority and WQ2 the lowest priority. FMan cannot dequeue frames from WQ1 unless WQ0 is empty and from WQ2 unless WQ1 and WQ0 are empty.

The work queue a frame will be enqueued on is determined from the socket buffer priority. skb_prio is just an internal tag that the kernel applies to the frames on the egress path and is not visible to the receiver.

| skb_prio | CoS |
|---|---|
| 0 | 0 |
| 1 | 1 |
| ≥2 | 2 |

Linux Ethernet Driver for DPAA 1.x Family

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
```

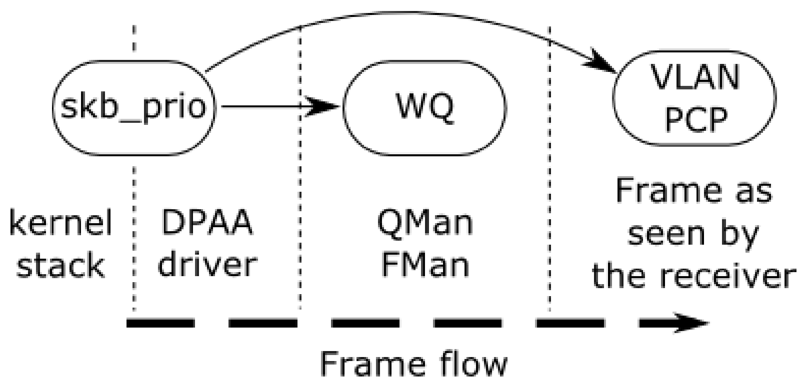
### VLAN tagging

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:



Linux Ethernet Driver for DPAA 1.x Family

#### 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 DPAA Private interfaces share a single buffer pool which accounts for the buffers in which the frames are stored upon receiving.

When the Buffer Pool reaches the refill/depletion threshold, PFC frames are sent back to the sender in order to pause frames transmission and thus avoid frame loss.

FMan sends PFC frames that pause all Classes of Traffic defined. The only difference between the classes is the pause time.

The pause time can be configured from menuconfig. A pause time of 0 disables that Class of Service.

When the common buffer pool depletes, issued PFC frames look like this.

| Class-Enable Vector | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| Pause Quanta Class 0 | | | | | | | |
| Pause Quanta Class 1 | | | | | | | |
| Pause Quanta Class 2 | | | | | | | |
| 0 | | | | | | | |
| 0 | | | | | | | |
| ... | | | | | | | |

FMan issues either 802.1Qbb PFC or 802.3x PAUSE frames depending on the platform, but there is no difference in controlling their transmission and reception via ethtool. For more detatils, see the chapter on PAUSE frames support.

#### Enabling and disabling PFC using ethtool

FMan issues either 802.1Qbb PFC or 802.3x PAUSE frames depending on the platform, but there is no difference in controlling their transmission and reception via ethtool. For more detatils, see the chapter on PAUSE frames support.

Pause Frames Flow Control on page 282

### 20.1.2.2.8 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:

Linux Ethernet Driver for DPAA 1.x Family

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

Following are the values for various QorIQ DPAA platforms:

Table 18: FMAN v2 devices core affined queues

| Interface | FQID base | P4080 | P5020 | P5040 | P3041 | P2041 | P1023 | T1040 (FMAN v3) |
|---|---|---|---|---|---|---|---|---|
| fm1-gb0 | 0x3800 | Y | Y | Y | Y | Y | 0x7800 | Y |
| fm1-gb1 | 0x3880 | Y | Y | Y | Y | Y | 0x7880 | Y |
| fm1-gb2 | 0x3900 | Y | Y | Y | Y | Y | | Y |
| fm1-gb3 | 0x3980 | Y | Y | Y | Y | Y | | Y |
| fm1-gb4 | 0x3a00 | | Y | Y | Y | | | Y |
| fm1-10g | 0x3c00 | Y | Y | Y | | | | |
| fm2-gb0 | 0x7800 | Y | | Y | | | | |
| fm2-gb1 | 0x7880 | Y | | Y | | | | |
| fm2-gb2 | 0x7900 | Y | | Y | | | | |
| fm2-gb3 | 0x7980 | Y | | Y | | | | |
| fm2-gb4 | 0x7a00 | | | Y | | | | |
| fm2-10g | 0x7c00 | Y | | Y | | | | |

Table 19: FMAN v3 devices core affined queues

| Interface | FQID base | T4240 | T4160 | B4860 | B4420 | T2080 |
|---|---|---|---|---|---|---|
| fm1-mac1 | 0x3800 | Y | Y | Y | Y | Y |
| fm1-mac2 | 0x3880 | Y | Y | Y | Y | Y |
| fm1-mac3 | 0x3900 | Y | Y | Y | Y | Y |
| fm1-mac4 | 0x3980 | Y | Y | Y | Y | Y |
| fm1-mac5 | 0x3a00 | Y | Y | Y | | |
| fm1-mac6 | 0x3a80 | Y | Y | Y | | |
| fm1-mac9 | 0x3c00 | Y | Y | Y | | Y |

*Table continues on the next page...*

Linux Ethernet Driver for DPAA 1.x Family

Table 19: FMAN v3 devices core affined queues (continued)

| Interface | FQID base | T4240 | T4160 | B4860 | B4420 | T2080 |
|---|---|---|---|---|---|---|
| fm1-mac10 | 0x3c80 | Y | | Y | | Y |
| fm2-mac1 | 0x7800 | Y | Y | | | |
| fm2-mac2 | 0x7880 | Y | Y | | | |
| fm2-mac3 | 0x7900 | Y | Y | | | |
| fm2-mac4 | 0x7980 | Y | Y | | | |
| fm2-mac5 | 0x7a00 | Y | Y | | | |
| fm2-mac6 | 0x7a80 | Y | Y | | | |
| fm2-mac9 | 0x7c00 | Y | Y | | | |
| fm2-mac10 | 0x7c80 | Y | | | | |

These queues are assigned to cores in a round-robin fashion. For instance, if there are 8 cores, 0x3800 will be serviced by core 0, 0x3801 by core 1, 0x3808 by core 0, etc. Currently, if one specifies extra RX PCD queues in the device tree, these queues will **also** be assigned in this round-robin fashion.

## 20.1.3 Ethernet Advanced Drivers

Enter a short description of your topic here (optional).

This is the start of your topic.

### 20.1.3.1 Macless DPAA Ethernet Driver

Macless DPAA Ethernet Driver is a virtual Ethernet driver, hence not using an Ethernet controller to transmit frames. This type of DPAA Ethernet driver is a lightweight version of Private DPAA Ethernet Driver that does not have MAC device control primitives, but maintains the same structure. Macless DPAA Ethernet Driver is used with many different names, “macless”, “MAC-less” or previous SDK name, “Virtual Ethernet Driver”. This DPAA Ethernet Driver is used in simple scenarios, like communication between USDPAA and Linux or communication between two Linux partitions or more complex ones, like Offloading Architecture and Shared MAC scenarios. All these scenarios are presented in the Configuration chapter.

#### 20.1.3.1.1 Configuration

The main configuration options are offered, like in any other embedded Linux Ethernet driver, by the device tree configuration and the common interface offered by the Linux kernel (ifconfig, ethtool, etc.). All configuration capabilities are presented in this chapter.

##### 20.1.3.1.1.1 Device Tree Configuration

The Macless DPAA Ethernet Driver has *fsl,dpa-ethernet-macless* as compatible string in the device tree. For example, the standard structure for the device tree node that specifies a Macless interface in a B4860QDS device tree looks like:

```
ethernet@16 {
    compatible = "fsl,b4860-dpa-ethernet-macless", "fsl,dpa-ethernet-macless";
```

Linux Ethernet Driver for DPAA 1.x Family

```
    fsl,bman-buffer-pools = <&bp10>;
    fsl,qman-frame-queues-rx = <0xfa0 0x8>;
    fsl,qman-frame-queues-tx = <0xfa8 0x8>;
    local-mac-address = [00 11 22 33 44 55];
};
```

The properties that a Macless Driver device tree node can have are:

- *fsl,bman-buffer-pools* - a list of buffer pools used by this interface. The Macless DPAA Ethernet Driver will use only static defined buffer pools because the frames are transmitted between different memory spaces. See Note 2 for more details;
- *fsl,qman-frame-queues-rx* - a list of base/count pairs of frame queues that will transmit frames to the Macless Driver. These queues are initialized as PCD queues. By default there are 8 Rx queues because there are 8 CPUs on B4860QDS boards;
- *fsl,qman-frame-queues-tx* - a list of base/count pairs of frame queues that fetch the frames from Macless Driver to the next hardware device (TX Port, O/H Port) or to another Macless Driver. These queues will only be created, but not initialized. It is the role of the next software module in the data path to initialize these queues;
- *local-mac-address* - this is a virtual L2 address used to identify the driver in the Linux kernel network stack.

Note 1: The Macless DPAA Ethernet Driver does not have a MAC device to control. Because of this the “fsl,fman-mac” property is missing from the device tree specification. This property exists for all other DPAA Ethernet Drivers.

Note 2: A static defined buffer pool should be declared as exemplified below for a B4860QDS board:

```
bp7: buffer-pool@7 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <7>;
    fsl,bpool-ethernet-cfg = < 0 256 0 192 0 0x40000000>;
    fsl,bpool-thresholds = <0x400 0xc00 0x0 0x0>;
};
```

Although the above device tree node is a representation for the BMan Driver, *fsl,bpool-ethernet-cfg* property is parsed solely by the DPAA Ethernet Driver that has a reference to this buffer pool. This property has the following meaning:

```
fsl,bpool-ethernet-cfg = <count size base_address>;
```

- *count* - represents the number of buffers from the buffer pool;
- *size* - buffer size;
- *base_address* - physical address of the buffer pool. Because two Linux partition have different memory spaces, this physical address will be mapped in both partitions. In scenarios where a single partition is used, this address will be invalid, particularly 0, and a dynamic mapping from user-space to kernel-space will be done.

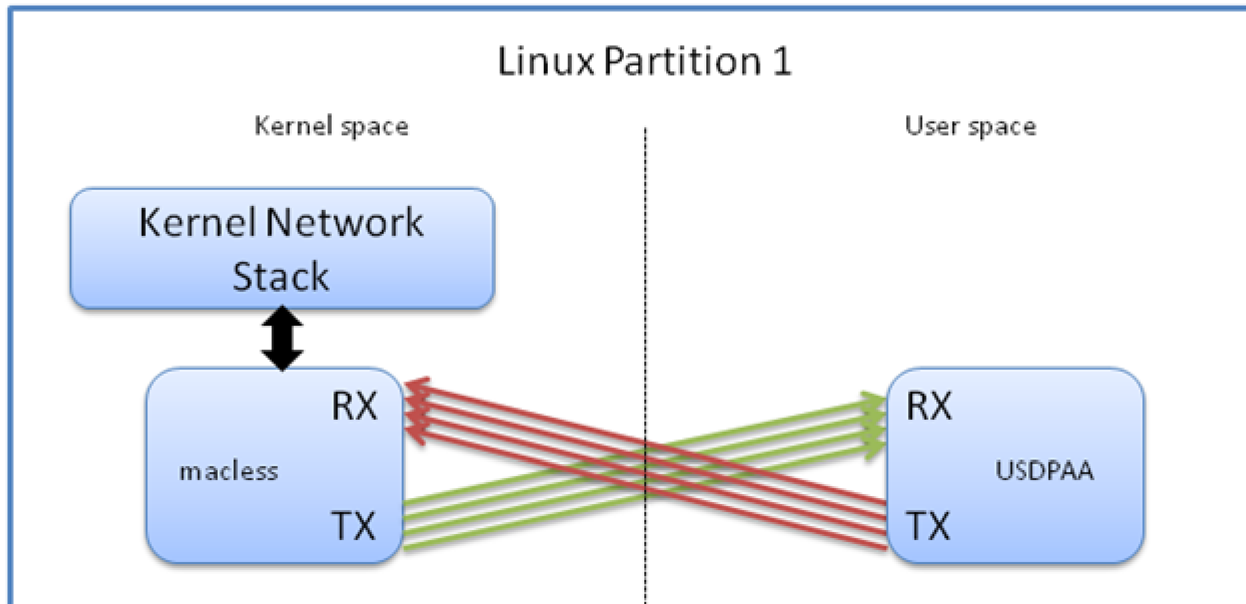
The example above declares a buffer pool with buffer pool ID 7, and describes a pool with 256 192-byte buffers, occupying the memory region from 0x40000000 to 0x40000000 + 256*192. The reason there are two numbers per each of *count*, *size*, and *base_address* is that we support 36-bit addresses on the P4080 (and 64-bit on P5020). It should be noted that the size of those parameters should be set by the root node's *#address-cells* and *#size-cells* properties. The *fsl,bpool-ethernet-seeds* property is there to tell the driver which is using the buffer pool whether to seed the pool with the declared buffers, or not.

The above generic device tree structure can be modified, depending on the scenario that Macless Ethernet Driver is used into. These scenarios are:

**Communication inside a single Linux partition between USDPAA and Linux Network Stack through a Macless DPAA Ethernet Driver.**

Linux Ethernet Driver for DPAA 1.x Family

For some applications, USDPAA needs the benefits and flexibility of Linux networking capabilities. To connect to the Linux Network Stack, it uses a Macless DPAA Ethernet Driver. This scenario is represented in the following picture:




The device tree configuration should be similar to the one below extracted from a B4860QDS device tree configuration:

```
ethernet@16 {
    compatible = "fsl,b4860-dpa-ethernet-macless", "fsl,dpa-ethernet-macless";
    fsl,bman-buffer-pools = <&bp16>;
    fsl,qman-frame-queues-rx = <0xfa0 0x8>;
    fsl,qman-frame-queues-tx = <0xfa8 0x8>;
    local-mac-address = [00 11 22 33 44 55];
};

bp16: buffer-pool@16 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <0x10>;
    fsl,bpool-ethernet-cfg = <0x0 0x800 0x0 0x6c0 0x0 0x0>;
    fsl,bpool-thresholds = <0x100 0x300 0x0 0x0>;
};
```

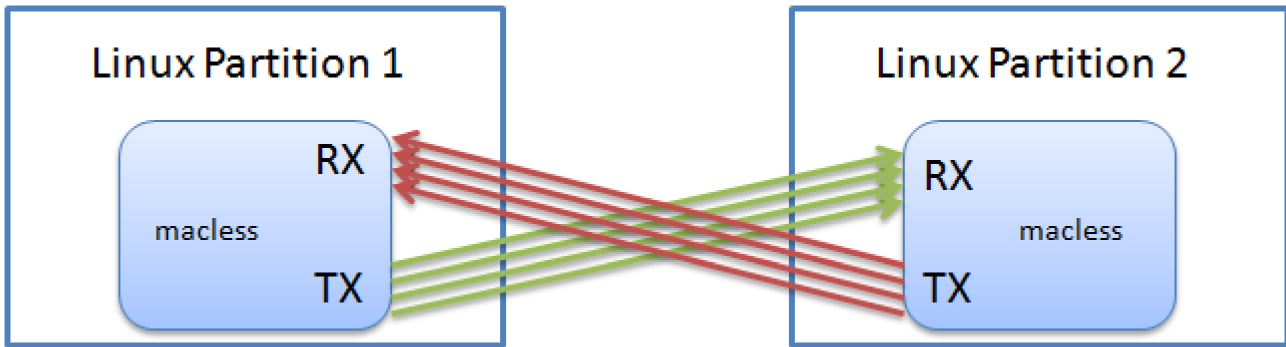
In the above device tree snippet, there are two device tree nodes, one for the Macless DPAA Ethernet Driver and one for the static definition of the buffer pool used in the transmission. USDPAA will also parse these nodes and it will manage the buffer pool used in the communication. Additionally, because a Macless DPAA Ethernet Driver initializes only its RX frame queues, USDPAA has the additional role of initializing Macless DPAA Ethernet Driver's TX frame queues also. Another important configuration is the base address of the buffer pool. Because the communication occurs inside a single Linux partition, there is no need for the buffer pool to advertise its physical address, hence the 0 address representing an invalid physical address. The mapping of the Macless kernel memory space to USDPAA's user-space memory space is done through the kernel *kmap*/*kunmap* primitives.

#### Communication between two Linux partitions through a Macless DPAA Ethernet Driver

With the help of the Freescale virtualization solution, called Topaz, two or more partitions can run on the same board. Every communication between partitions is supervised by Topaz. Although network communication between partitions is possible through external connections, one direct connection can be created inside the

Linux Ethernet Driver for DPAA 1.x Family

board using Macless DPAA Ethernet Driver and hardware frame queues. To create a fast networking communication between partitions, two or more Macless DPAA Ethernet Drivers should be used as depicted below:




The TX frame queues from one partition should be the RX frame queues from the other partition. The device tree nodes in each partition are represented below.

First partition's device tree representation:

```
dpa-ethernet@10 {
    compatible = "fsl,p4080-dpa-ethernet", "fsl,dpa-ethernet-macless";
    fsl,qman-frame-queues-rx = <0x4000 8>;
    fsl,qman-frame-queues-tx = <0x4008 8>;
    local-mac-address = [02 00 c0 a8 6f fe];
    fsl,bman-buffer-pools = <&bp10>;
};
bp10: buffer-pool@10 {
    compatible = "fsl,b4080-bpool", "fsl,bpool";
    fsl,bpid = <10>;
    fsl,bpool-ethernet-cfg = <0 0x80 0 1728 0 0x80000000>;
    fsl,bpool-ethernet-seeds;
};
```

Device tree from the second partition:

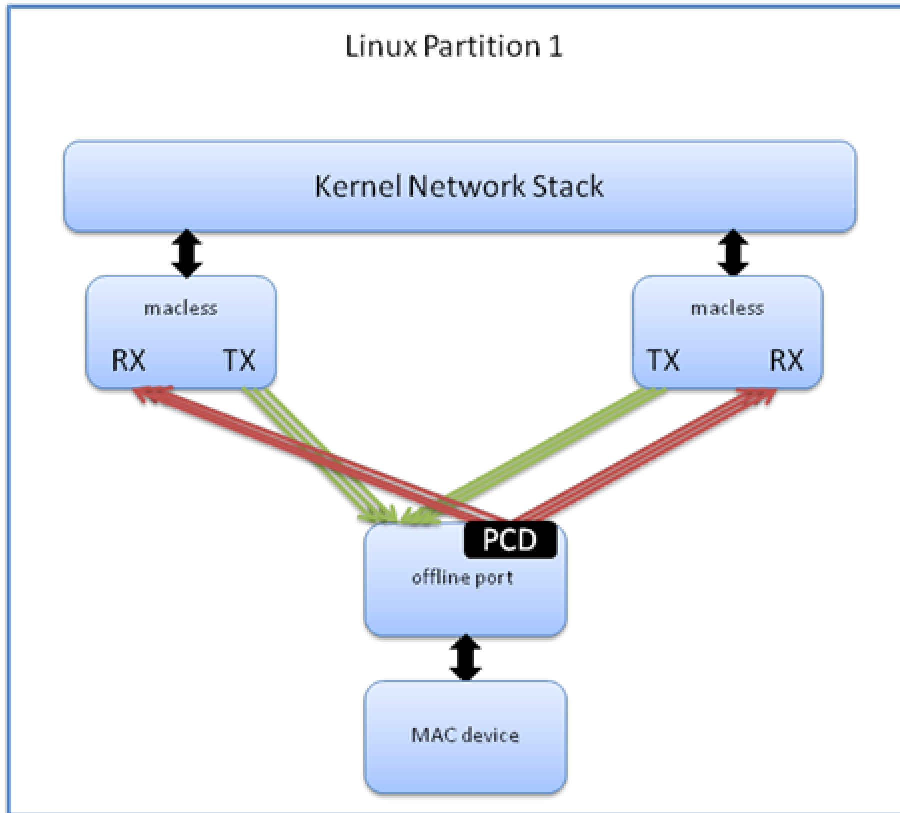
```
dpa-ethernet@10 {
    compatible = "fsl,p4080-dpa-ethernet", "fsl,dpa-ethernet-macless";
    fsl,qman-frame-queues-rx = <0x4008 8>;
    fsl,qman-frame-queues-tx = <0x4000 8>;
    local-mac-address = [02 00 c0 a8 79 fe];
    fsl,bman-buffer-pools = <&bp10>;
};
bp10: buffer-pool@10 {
    compatible = "fsl,b4080-bpool", "fsl,bpool";
    fsl,bpid = <10>;
    fsl,bpool-ethernet-cfg = <0 0x80 0 1728 0 0x80000000>;
};
```

In the above configuration there are 8 TX frame queues and 8 RX frame queues used in interchangeable roles in each partition. Both interfaces share one statically defined buffer pool. The same buffer pool definition should be used in both partitions, except for the *fsl,bpool-ethernet-seeds* property. In order to easily map from one memory address space from one partition to another partition a valid physical base address must be configured. The mapping from physical address to each partition’s virtual kernel address is done with *ioremap* kernel primitive. The seeding of the buffer pool will be done by the Macless DPAA Ethernet Driver from the partition that has *fsl,bpool-ethernet-seeds* in its device tree node. In the above configuration, the first partition will seed the buffer pool.

Linux Ethernet Driver for DPAA 1.x Family

**Communication inside a Linux partition using Offline Ports and Macless DPAA Ethernet Driver**

DPAA has many hardware offload capabilities. One of them is the Offline Parsing/Host Command Port. This hardware portal can fetch traffic from multiple destinations (such as Macless DPAA Ethernet Drivers) and send traffic through many configurable destinations, like MAC devices. For example, this scenario can be used to offload into DPAA stack functionality, like IPSec, IP fragmentation and reassembly or TCP segmentation and reassembly.




Because the Macless DPAA Ethernet Driver does not have an Ethernet controller attached, it can be used in many hardware configuration scenarios similar to the one used in the figure above. In order to do that, special attention should be paid to Frame Queues configuration and the Buffer Pool configuration.

**Communication of different Linux partitions through a single MAC device, using Macless DPAA Ethernet Driver and Shared DPAA Ethernet Driver**

In this scenario, a new type of DPAA Ethernet Driver is needed. This is called Shared DPAA Ethernet Driver and its configuration is described in Shared DPAA Ethernet Driver configuration chapter. Shared DPAA Ethernet Driver together with Macless DPAA Ethernet Driver are used in a typical shared MAC scenario, described in the Shared DPAA Ethernet Driver chapter.

## 20.1.3.1.1.2 Configuration available through Linux interfaces

Because Macless driver does not manage a MAC device, some *ethtool* and *ifconfig* options will not be available. All Layer 3 configurations, like setting an IP address, are generally available.

## 20.1.3.1.2 Features

This chapter describes the features of the Macless DPAA Ethernet Driver, their configuration options, fine-tuning and known limitations.

**MAC device control capability**

Linux Ethernet Driver for DPAA 1.x Family

As of SDK 1.5, the Macless DPAA Ethernet Driver is capable of controlling a MAC device on behalf of an user-space application, like USDPAA. This will add an additional control mechanism to the amount of customization available through Macless DPAA Ethernet Driver. For example, in the scenario depicted in *Communication inside a Linux partition using Offline Ports and Macless DPAA Ethernet Driver* scenario, it is desired that one of the Macless DPAA Ethernet Drivers should control the MAC device.

Some of the MAC devices are initialized by the Proxy DPAA Ethernet Driver to be used by USDPAA. By setting the following attribute in the device tree specification of the Macless DPAA Ethernet Driver

```
proxy = <&proxy1>;
```

a reference to the MAC device initialized by the Proxy driver can be obtained in the Macless driver. This will allow basic operations regarding L2 address of the MAC device using Macless driver Linux interface. Currently, the supported operations are enablement and disablement of the MAC device and setting multicast or unicast addresses.

**Scatter/Gather**

The current implementation supports DPAA Scatter/Gather only on the RX path. Therefore, the Macless DPAA Ethernet Driver will know how to handle Scatter/Gather frames, but the driver does not advertise TX Scatter/Gather capability to the Linux Network Core and only linear socket buffers will be accepted on the TX path.

**Hardware checksum**

As previously presented in the Private DPAA Ethernet Driver's Features chapter, the FMan supports computation of the L3 and/or L4 checksum for certain protocols (mainly TCP/IP and UDP/IP). Because the same transmission procedure is used to activate hardware checksum in both Private and Macless drivers, the same limitations presented in the *Private DPAA Ethernet Driver* chapter apply to Macless driver. Currently, at frame reception, the Macless Driver is not able to fetch the checksum computed by FMan.

## 20.1.3.2 Shared DPAA Ethernet Driver

Shared DPAA Ethernet Driver is similar to Macless Driver, only it has a MAC device in its control. Although it has similar structures with Private DPAA Ethernet Driver, it does not have the same optimization and feature set available. This is because it can be used in pair with a Macless Driver from another Linux partition or it can be used in pair with USDPAA. Therefore, it needs portability and easiness, qualities that Private DPAA Ethernet Driver sacrifices in exchange for higher performance. Shared DPAA Ethernet Driver is also known as *shared* or *Shared Controller* in the previous SDK release. In this document Shared DPAA Ethernet Driver will be referred as *Shared Driver*. This flavor of DPAA Ethernet Driver is used in two scenarios. First, when a MAC device is shared between different partitions under a Hypervisor and second, when a MAC device is shared with a User Space DPAA application in a single Linux partition. These use cases are presented in the Configuration chapter below.

### 20.1.3.2.1 Configuration

The main configuration options are offered by the device tree configuration and common Linux interfaces (*ifconfig*, *ethtool*, etc.). All configuration capabilities are presented in this chapter.

#### 20.1.3.2.1.1 Device Tree Configuration

The Shared Driver has the *fsl,dpa-ethernet-shared* string as compatible string in the device tree. Therefore, the standard structure for the device tree node that specifies a Shared interface should be similar to the below snippet of a B4860QDS device tree node:

```
ethernet@9 {
    compatible = "fsl,b4860-dpa-ethernet-shared", "fsl,dpa-ethernet-shared";
    fsl,fman-mac = <&fm1mac10>;
    fsl,bman-buffer-pools = <&bp17>;
    fsl,qman-frame-queues-rx = <0x5e 1 0x5f 1 0x2000 3>;
```

Linux Ethernet Driver for DPAA 1.x Family

```
    fsl,qman-frame-queues-tx = <0 1 0 1 0x3000 8>;
};
```

Following are the properties of a Shared Driver’s device tree node:

- *fsl,fman-mac* - this is the MAC device reference that is in Shared Driver’s control;
- *fsl,bman-buffer-pools* - as in Macless Driver specification this is a list of buffer pools used by this interface. Since the frames are transmitted between different memory spaces, Shared Driver will use only static defined buffer pools in both use cases;
- *fsl,qman-frame-queues-rx* - a list of base/count pairs of frame queues from which the Shared Driver dequeues frames. Because the Shared Driver has a MAC device in its control, the device tree format of the frame queues is similar to the Private Driver’s device tree format. The MAC device must have at least two types of frame queues specified at initialization. These are the error frame queue which is the frame queue on which received erroneous frames will be placed and the default frame queue which is the frame queue where the frame will be placed if no other frame queue is selected for transmission. The next batch represents the PCD queues. These queues will be used by the PCD rules configured by the user. The above device tree example defines one error queue with ID 0x5e, one default queue with ID 0x5f and 3 configurable PCD queues, 0x2000, 0x2001, 0x2002;
- *fsl,qman-frame-queues-tx* - a list of base/count pairs of frame queues used by the Shared Driver send the frames to the MAC device. As in Private DPAA Ethernet Driver, these are the TX error frame queue, TX confirmation frame queue and the standard TX frame queues. In the above example, a value of 0 for TX error and TX confirmation queues enables the dynamic allocation of values for queues’ IDs, letting the QMan assign available values. Besides one dynamic TX error and one dynamic TX confirmation queue, 8 standard TX queues will be created, with IDs between 0x3000 and 0x3007. It is recommended to specify up to NR_CPUs frame queues and have a direct mapping between frame queue and CPU. In the above example, B4860QDS has 8 CPUs.

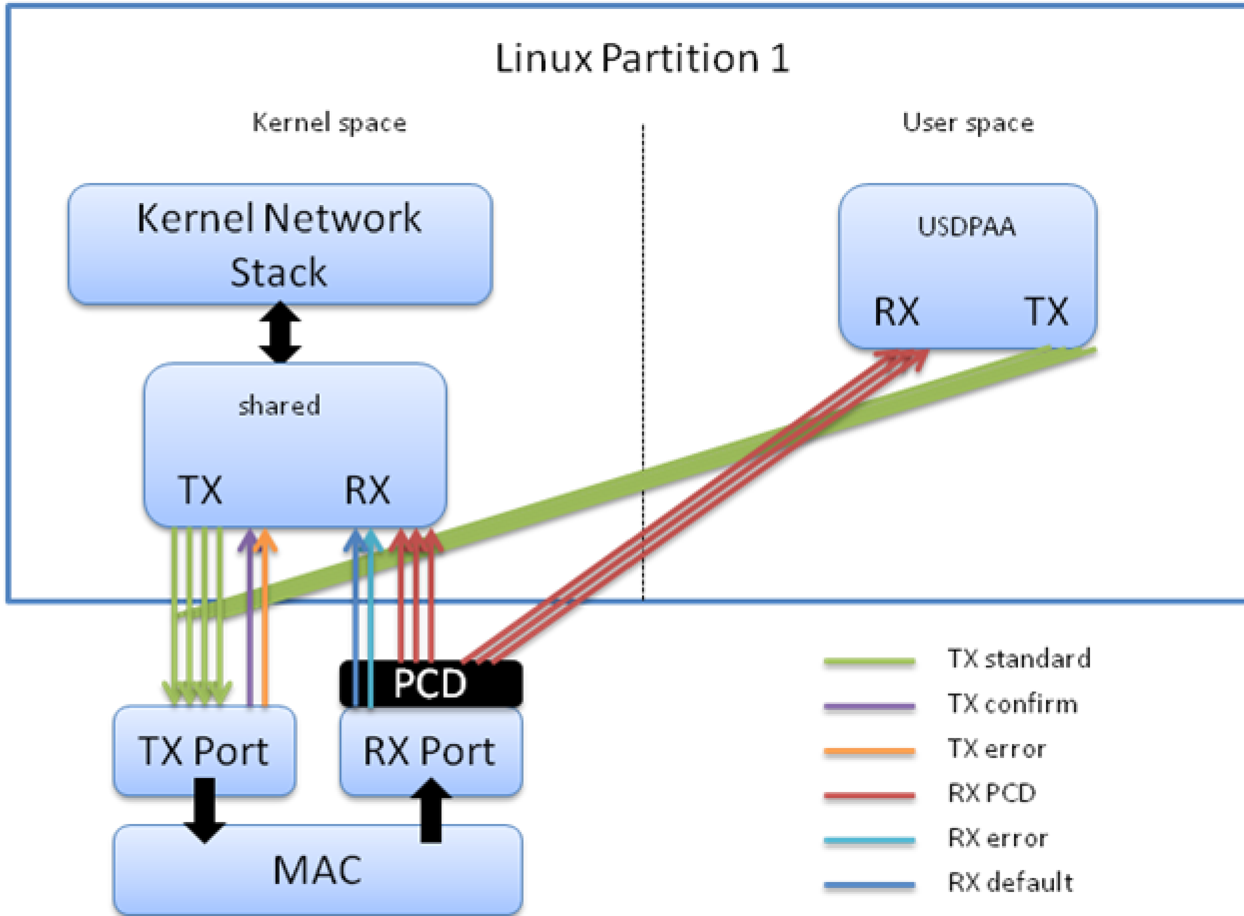
Note 1: The static defined buffer pool has the same representation as defined in the Macless Driver Configuration chapter. As stated above, the Shared Driver will be used in two different scenarios. Each scenario involves entities that have different memory address spaces, but must share the buffer pools’ configuration. This is the reason for not using dynamic buffer pools, like in Private DPAA Ethernet Driver.

Following are the scenarios in which Shared Driver can be used:

**USDPAA and Linux stack communicating through a single MAC device, using Shared Driver inside a single Linux partition**

USDPAA applications handle only a certain type of traffic, the rest of the packets being handled by Linux Network Stack. The simplest solution is to have a Shared Driver that knows about the traffic division. In order to split the traffic, PCD rules must be applied on the incoming port. This scenario is presented below.

Linux Ethernet Driver for DPAA 1.x Family




The device tree configuration should be similar to the one below extracted from a B4860QDS device tree configuration.

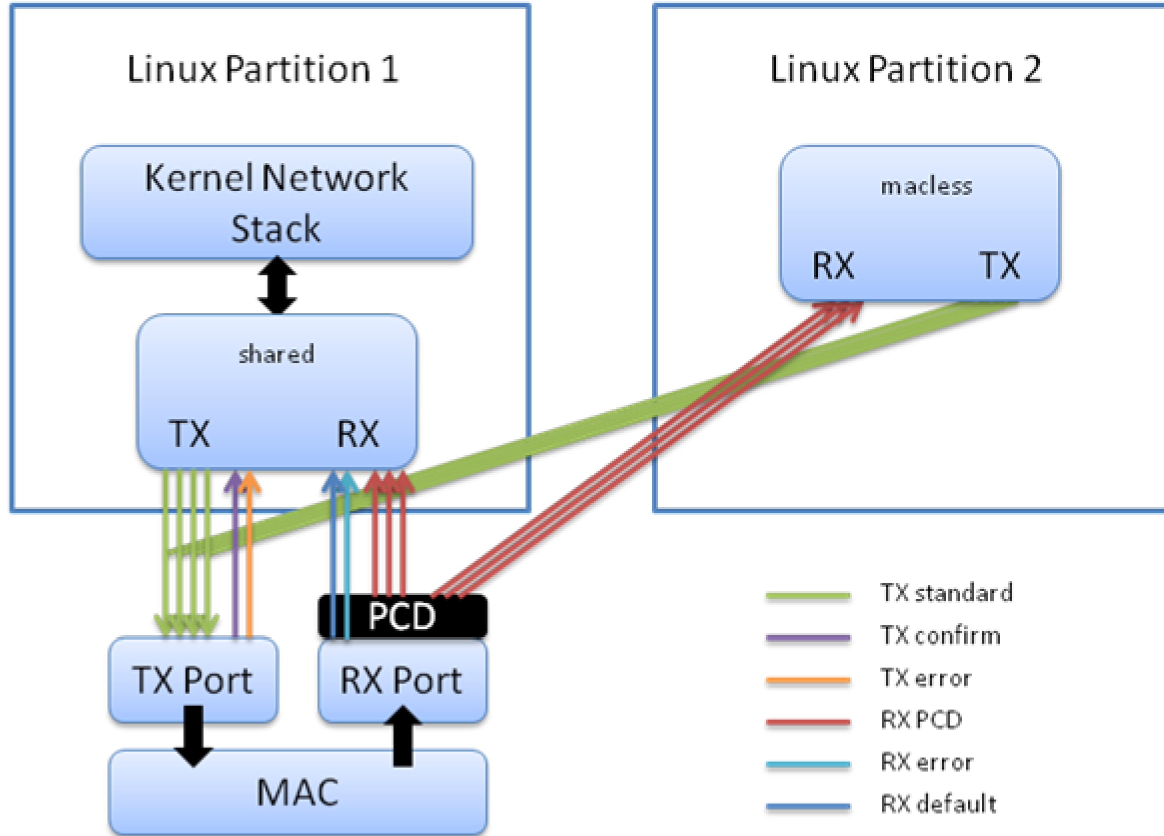
```
ethernet@9 {
    compatible = "fsl,b4860-dpa-ethernet-shared", "fsl,dpa-ethernet-shared";
    fsl,bman-buffer-pools = <&bp17>;
    fsl,qman-frame-queues-rx = <0x5e 1 0x5f 1 0x2000 3>;
    fsl,qman-frame-queues-tx = <0 1 0 1 0x3000 8>;
    fsl,fman-mac = <&fm1mac10>;
};
bp17: buffer-pool@17 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <17>;
    fsl,bpool-ethernet-cfg = <0 2048 0 1728 0 0>;
    fsl,bpool-thresholds = <0x100 0x300 0x0 0x0>;
};
```

In the above device tree snippet, there are two device tree nodes, one for the Shared Driver and one for the static definition of the buffer pool used for transmission. In this scenario, USDPAA allocates the buffer pool used in the communication, therefore the above device tree nodes will be parsed by USDPAA also. One important configuration is the base address of the buffer pool. Because the communication occurs inside a single Linux partition, there is no need for the buffer pool to advertise its physical address, hence the 0 address representing an invalid physical address. Because the buffers are allocated by USDPAA, these are in the virtual address space of USDPAA. The mapping of the USDPAA's user-space memory space to kernel space is done through kernel *kmap*/*kunmap* primitives.

Linux Ethernet Driver for DPAA 1.x Family

### Shared communication between two Linux partitions through a single MAC device using a Shared Driver and Macless Driver

In scenarios with multiple Linux partitions managed by a Hypervisor, it is sometimes desired to split one MAC device's traffic between partitions. To achieve this, one partition should use the Shared Driver which will manage the MAC device and the other Linux partition should use the Macless Driver as depicted below.




In scenarios with multiple Linux partitions managed by a Hypervisor, it is sometimes desired to split one MAC device's traffic between partitions. To achieve this, one partition should use the Shared Driver which will manage the MAC device and the other Linux partition should use the Macless Driver as depicted below.

First partition:

```
dpa-ethernet@4 {
    compatible = "fsl,b4860-dpa-ethernet", "fsl,dpa-ethernet-shared";
    fsl,qman-frame-queues-rx = <0x340 1 0x341 1 0x320 8>;
    fsl,qman-frame-queues-tx = <0x342 1 0x343 1 0x300 8>;
    fsl,bman-buffer-pools = <&part1_bp11>;
};
part1_bp11: buffer-pool@11 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <11>;
    fsl,bpool-ethernet-cfg = <0 0x80 0 1728 0 0x80036000>;
    fsl,bpool-ethernet-seeds;
};
```

Second partition:

```
dpa-ethernet@20 {
    compatible = "fsl,b4860-dpa-ethernet", "fsl,dpa-ethernet-macless";
```

Linux Ethernet Driver for DPAA 1.x Family

```
    fsl,qman-frame-queues-rx = <0x350 8>;
    fsl,qman-frame-queues-tx = <0x300 8>;
    local-mac-address = [02 00 c0 a8 a1 fe];
    fsl,bman-buffer-pools = <&part2_bp11>;
};
part2_bp11: buffer-pool@11 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <11>;
    fsl,bpool-ethernet-cfg = <0 0x80 0 1728 0 0x80036000>;
};
```

In the above configuration the standard 8 TX queues from the first partition are the same as the standard 8 TX queues from the second partition. Both interfaces share one static buffer pool and the same buffer pool definition is used in both partitions. As in other multi-partition scenarios, the buffers are allocated from the physical memory, indicated by the base address parameter. In the above configuration, the base address is 0x80036000. When a frame arrives to one partition, its address should be mapped to partition's kernel address space. The mapping from physical address to each partition's virtual kernel address is done with *ioremap* kernel primitive. The seeding of the buffer pool will be done by the Shared Driver from the first partition as it has *fsl,bpool-ethernet-seeds* in its device tree node.

#### 20.1.3.2.1.2 Configuration available through Linux interfaces

The Shared Driver has the same data structures as the Private Driver, therefore it has the same configuration capabilities available through standard Linux interfaces (*ethtool*, *ifconfig*, etc.) as the Private Driver.

### 20.1.3.2.2 Features

The same transmission primitives are used both by the Shared and the Macless driver. Therefore, as stated in the Macless Driver's *Features* chapter, the Shared Driver is capable of receiving Scatter/Gather frames and is able to offload checksum computation on transmission. On reception, Shared Driver is not able to fetch the checksum computed by FMan.

## 20.1.3.3 Proxy DPAA Ethernet Driver

USDPAA applications achieve high speeds for particular types of traffic, bypassing the processing done by the Linux Network Stack. These applications reside in user-space and need kernel-space configuration in order to initialize the necessary hardware modules used along the data path. The configuration of buffer pools and frame queues and initialization of MAC devices used by the USDPAA are delegated to the Proxy DPAA Ethernet Driver. This type of DPAA Ethernet Driver runs once at system startup and does not have a data structure at run-time like the other DPAA Ethernet Drivers. Proxy DPAA Ethernet Driver is used with many different names, *proxy* or *Initialization Manager*, as it was known in the previous SDK release. In this document, it will be referred to as *Proxy Driver*.

### 20.1.3.3.1 Configuration

The only configuration for this type of DPAA Ethernet Driver can be made through device tree node configuration.

#### 20.1.3.3.1.1 Device Tree Configuration

The Proxy Driver has *fsl,dpa-ethernet-init* as compatible string in the device tree. For example, the standard structure for the device tree node that specifies a Proxy interface in a B4860QDS device tree is depicted below:

```
ethernet@8 {
    compatible = "fsl,b4860-dpa-ethernet-init", "fsl,dpa-ethernet-init";
    fsl,fman-mac = <&fm1mac10>;
    fsl,bman-buffer-pools = <&bp7 &bp8 &bp9>;
    fsl,qman-frame-queues-rx = <0x5c 0x1 0x5d 0x1>;
```

Linux Ethernet Driver for DPAA 1.x Family

```
    fsl,qman-frame-queues-tx = <0x7c 0x1 0x7d 0x1>;
};
```

All the above properties are the same as those used by Shared Driver that has a MAC device reference:

- *fsl,fman-mac* - this is the MAC device reference that will be initialized by the Proxy Driver;
- *fsl,bman-buffer-pools* - each port needs a buffer pool to get buffers from. Since it has a MAC device reference, Proxy Driver will need a statically defined buffer pool. This buffer cannot be dynamically created because the Proxy Driver does not know the memory address space of the software entity that will use the initialized infrastructure.
- *fsl,qman-frame-queues-rx* - a list of base/count pairs of frame queues. The device tree format of this property is the same as the format used in Private and Shared Drivers. The MAC device must have at least two queues specified at initialization. These are:
  - error queue which is the queue on which received erroneous frames will be placed;
  - default queue which is the queue where the frame will be placed if none of the PCD queues is selected for transmission;
  - PCD queues are optional. These queues will be used by the PCD rules configured by the user. In the above device tree example no PCD queues are defined.
- *fsl,qman-frame-queues-tx* - a list of base/count pairs of frame queues used by the software entity to send the frames to the MAC device. As in Private or Shared Drivers, these are TX error queue, TX confirmation queue and standard TX queues. In the above example, there are no standard TX queues.

Note 1: The statically defined buffer pools have the same device tree structure as that used for other DPAA Ethernet Drivers.

The classical scenario where a Proxy Driver is used is the one in which it initializes the MAC device on behalf of USDPAA inside a single Linux partition. Another scenario is where the initialization is made on behalf of another software entity or even another Linux partition.

#### 20.1.3.3.2 Features

Proxy Driver is used only to initialize some of the hardware modules and does not have advanced features like other DPAA Ethernet Drivers.

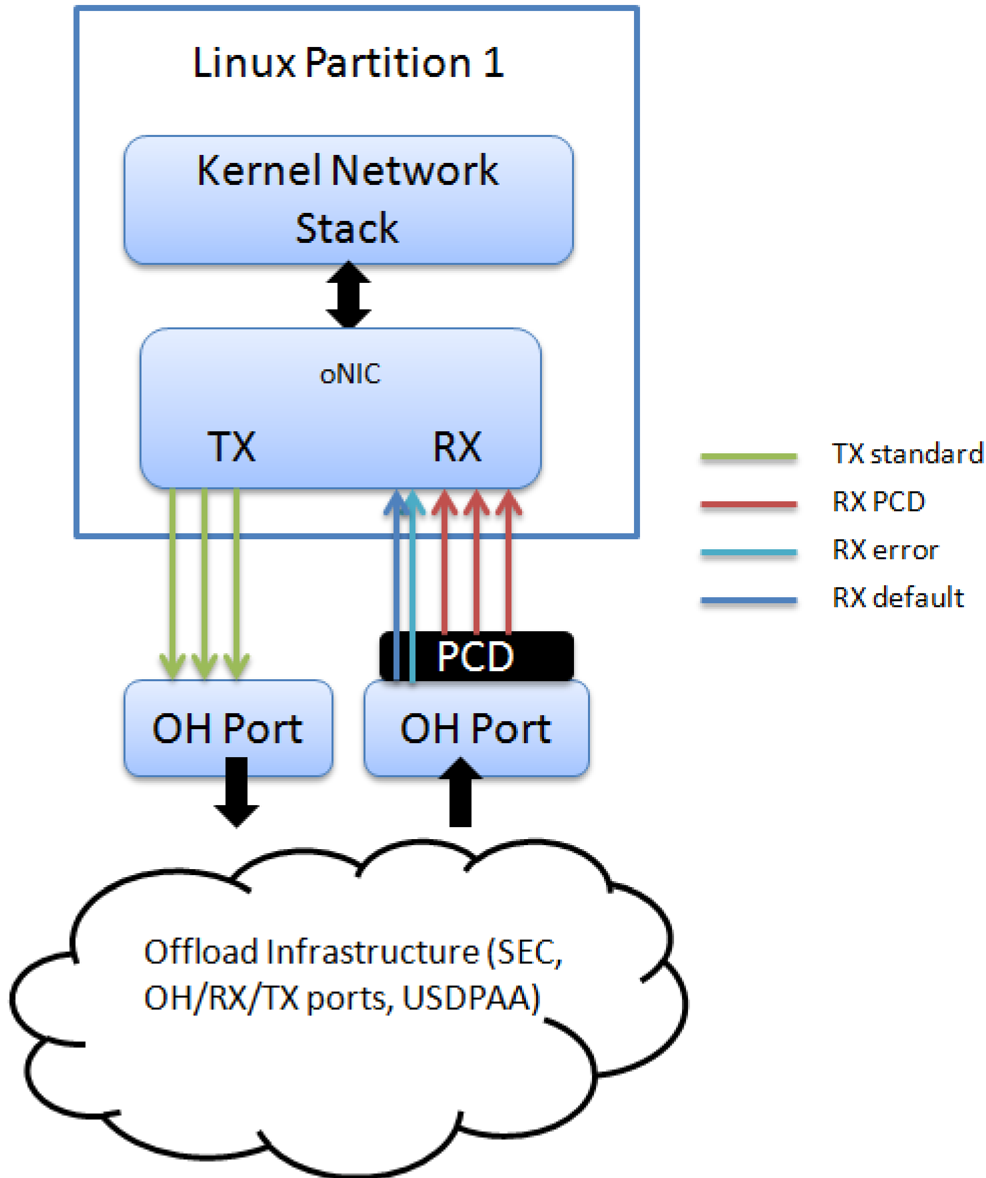
### 20.1.3.4 Offload NIC Ethernet Driver

Offload NIC Ethernet Driver is a virtual Ethernet driver, similar to Macless. This DPAA Ethernet Driver has CSUM offload and zero-copy, features that cannot be achieved with Macless. This DPAA Ethernet Driver is used in complex scenarios, like IPSec offload or communication with USDPAA where memory space isolation between kernel space and underlying offload architecture (OH ports, SEC, USDPAA) space is needed. Some of these scenarios are presented in the Configuration chapter.

This type of DPAA Ethernet Driver needs two OH ports with VSP capabilities enabled in order to achieve zero-copy.

Offload NIC Ethernet Driver will be referred in this documentation as "oNIC".

Linux Ethernet Driver for DPAA 1.x Family




### 20.1.3.4.1 Configuration

The main configuration options are offered, like in any other embedded Linux Ethernet driver, by the device tree configuration and the common interface offered by the Linux kernel (ifconfig, ethtool, etc.). All configuration capabilities are presented in this chapter.

#### 20.1.3.4.1.1 Device tree configuration

oNIC Ethernet Driver has *fsl,dpa-ethernet-generic* as compatible string in the device tree. For example, the standard structure for the device tree node that specifies an oNIC interface in a B4860QDS device tree looks like:

```
ethernet@16 {
    compatible = "fsl,b4860-dpa-ethernet-generic", "fsl,dpa-ethernet-generic";
    fsl,qman-frame-queues-tx = <4000 8x8>;
    fsl,qman-frame-queues-rx = <4008 8x8>;
    fsl,oh-ports = <&oh1 &oh2>;
    fsl,disable_buff_dealloc;
```

Linux Ethernet Driver for DPAA 1.x Family

```
    local-mac-address = [00 11 22 33 44 55];
};
```

The properties that the oNIC device tree node can have are:

- *fsl,qman-frame-queues-tx* - a list of base/count pairs of frame queues that forwards the frame buffers generated by oNIC to the Tx O/H port; it is recommended to have NR_CPUS Tx frame queues;
- *fsl,qman-frame-queues-rx* - a list of base/count pairs of frame queues called PCD queues that fetch the frames from the Rx O/H port to oNIC; besides these queues, oNIC uses the default and the error queue of the Rx O/ H port; the PCD queues can be used in PCD schemes applied on the Rx O/H port.
- *fsl,oh-ports* – links to O/H port device tree representation. The first value is a reference to the Rx O/H port used on ingress path, the second value is a reference to the Tx O/H port used on egress path. The device tree representation for the O/H port is described below;
- *fsl,disable_buff_dealloc* - in some scenarios the TX OH port cannot enable VSP because of the OH port limitation (O/H port cannot have VSP capability and IP Fragmentation enabled in the same time), therefore another hardware module along the TX path should release the buffers generated by oNIC into the draining buffer pool. With this flag, oNIC will configure the TX OH port to NOT release the buffers into the draining buffer pool. If other hardware modules do not release the buffers into the draining pool the device will leak available memory. This is an optional attribute.
- *local-mac-address* - this is a virtual L2 address used to identify the driver in the Linux kernel network stack.

Note 1: Like Macless DPAA Ethernet Driver, oNIC does not have a MAC device to control, therefore the “fsl,fman-mac” property is missing from the device tree specification.

oNIC does not have a buffer pool specified in its device tree representation. It will use the default buffer pools specified in the O/H port device tree node. The O/H port device tree representation looks like:

```
oh2: dpa-fman0-oh@2 {
     compatible = "fsl,dpa-oh";
     fsl,bman-buffer-pools = <&bp1 &bp2 &bp3 &bp4>;
     fsl,qman-frame-queues-oh = <110 1 111 1>;
     fsl,fman-oh-port = <&fman0_oh2>;
};
```

The properties of the O/H port device tree node can have are:

- *fsl,bman-buffer-pools* - the default buffer pools managed by oNIC; up to four static buffer pools can be specified. If this O/H port has VSPs, these buffer pools will be used in the default VSP. This property is parsed by oNIC that will initialize the default buffer pools;
- *fsl,qman-frame-queues-oh* - the default egress queues. The default queues are represented by one default frame queue and one error frame queue. This property will be parsed by oNIC that will initialize both queues and use them in case PCD frame queues are not defined;
- *fsl,fman-oh-port* - reference to FMan port device tree representation; this field is mandatory for Offline Port Driver, but it is not used by oNIC;

To activate VSP capability on a port, the user will have to configure the chosen node in the device tree. One valid entry looks like:

```
fman0_oh2-extd-args {
    cell-index = <0x1>;
    compatible = "fsl,fman-port-op-extended-args";
    vsp-window = <0x8 0x0>;
};
```

The most important property is:

Linux Ethernet Driver for DPAA 1.x Family

- *vsp-window* - the number of VSPs that this port can have and the default VSP (starting VSP index). In the above example 8 VSPs can be added to the O/H port (with ids 0, 1, ..., 8).

The device tree node of the O/H port use statically defined buffer pools. oNIC will parse the device tree representation of the default buffer pools of the Rx O/H port and will seed them. The device tree representation of the buffer pool is:

```
bp7: buffer-pool@7 {
    compatible = "fsl,b4860-bpool", "fsl,bpool";
    fsl,bpid = <7>;
    fsl,bpool-ethernet-cfg = <0 256 0 192 0 0>;
    fsl,bpool-thresholds = <0x400 0xc00 0x0 0x0>;
};
```

oNIC parses *fsl,bpid* and *fsl,bpool-ethernet-cfg* properties to initialize its internal buffer pools structures. As in other DPAA Ethernet Drivers, *fsl,bpool-ethernet-cfg* property has the following meaning:

```
fsl,bpool-ethernet-cfg = <count size base_address>;
```

- *count* - represents the number of buffers from the buffer pool;
- *size* - buffer size;
- *base_address* - this value is ignored by oNIC.

Note 2: oNIC will seed its configured buffer pools when the Ethernet interface is raised up. Because of this the *fsl,bpool-ethernet-seeds* property is ignored.

Note 3: All the buffers are allocated in the kernel memory space, therefore the *base_address* value is ignored.

The example above declares a buffer pool with buffer pool ID 7, and describes a pool with 256 192-byte buffers, occupying the memory in the kernel space. The *base_address* with <0 0> value is ignored. It should be noted that the size of those parameters should be set by the root node's *#address-cells* and *#size-cells* properties.

## 20.1.3.4.1.2 Configuration available through Linux interfaces

Because oNIC does not manage a MAC device, some *ethtool* and *ifconfig* options will not be available. All Layer 3 configurations, like setting an IP address, are generally available.

## 20.1.3.4.2 Features

This chapter describes the features of the oNIC Ethernet Driver, their configuration options, fine-tuning and known limitations.

### Hardware Checksum

The OH ports are able to validate and compute L3 and/or L4 checksums for certain protocols (mainly TCP/IP and UDP/IP). Because oNIC uses OH ports as communication points with the lower architecture, CSUM validation and computation are supported.

### Zero Copy

oNIC, representing kernel space, works in memory "isolation". This capability is achieved through OH ports with VSP capability enabled, which are able to copy the frames from one memory space to another. oNIC code and the underlying offload architecture code are not aware of the existence of another memory space, therefore simplifying the communication procedures.

Linux Ethernet Driver for DPAA 1.x Family

## 20.1.4 Offline Parsing Port Driver

Offline Parsing/Host Command Port, also known as *O/H Port* or simply *O/H*, is an FMan hardware module that is capable of multiplexing and de-multiplexing network traffic inside DPAA. Its behavior and programming model is similar to an “online” FMan port, supporting Parse-Classify-Distribute (PCD) function and buffer copy from one buffer pool to another. It is useful in complex DPAA scenarios, where plenty of hardware offload is desired, like IPSec or TCP fragmentation and reassembly. Similar to most DPAA hardware modules, a Linux kernel driver for O/H Ports is needed for initialization and configuration. In this document, the Offline Parsing Port Driver will be referred to as *Offline Port Driver*.

### 20.1.4.1 Configuration

Like other DPAA Ethernet Drivers, the Offline Port Driver is a standard Linux platform device driver, whose configuration is available through device tree. Also, the support for Offline Port Driver is enabled through a kernel Kconfig option.

#### 20.1.4.1.1 Kconfig Option

The Offline Port Driver is enabled in the Kbuild Linux configuration by default. It can be toggled via the following menuconfig option:

```
Device Drivers
    ---> Network device support
        ---> Ethernet driver support
            ---> Freescale devices
                ---> DPAA Ethernet
                    ---> Offline Ports support
```

#### 20.1.4.1.2 Device Tree Configuration

The Offline Port Driver has *fsl,dpa-oh* as compatible string in the device tree. For example, the standard structure for the device tree node that specifies an Offline Port Driver in a B4860QDS device tree is depicted below:

```
dpa-fman0-oh@2 {
    compatible = "fsl,dpa-oh";
    fsl,bman-buffer-pools = <&bp10>;
    fsl,qman-frame-queues-oh = <0x6e 0x1 0x6f 0x1>;
    fsl,qman-frame-queues-tx = <0x70 0x5>;
    fsl,qman-frame-queues-ingress = <base_id1 count1 ... base_idn countn>;
    fsl,qman-frame-queues-egress = <base_id1 count1 ... base_idn countn>;
    fsl,qman-channel-ids-egress = <channel_id1 ... channel_idn>;
    fsl,fman-oh-port = <&fman0_oh2>;
};
```

This device tree node resides inside *fsl,dpaa* device tree node, near DPAA Ethernet Driver device tree node specification.

Historically the offline port driver did not initialize the frame queues that entered and exited the OH port and relied on software components (Ethernet driver, USDPAA, other kernel modules) to initialize those queues. The offline port driver added capabilities to initialize ingress queues (queues that enter the Offline Port) simplifying the work for the Ethernet driver, but maintaining the same complexity in USDPAA and/or other kernel modules. Full capability for initializing both ingress and egress queues has been added, eliminating the dependency on USDPAA and/or other kernel modules. With it, complex offload architectures that have OH ports can be largely initialized by the offline port driver.

Linux Ethernet Driver for DPAA 1.x Family

The additional device tree attributes, qman-frame-queues-ingress, qman-frame-queues-egress and qman-channel-ids-egress are optional and do not interfere with existing code. Therefore, backward compatibility is maintained.

Following are the properties that could be used within Offline Port Driver's device tree node:

- *fsl,bman-buffer-pools* - a list of buffer pools used by this O/H Port. The O/H Port will use these statically defined buffer pools in case it will do IP Fragmentation or buffer copy from one buffer pool to another. For IP Fragmentation, one incoming frame is divided into multiple frames. In order to copy data from a buffer pool to another, VSP (Virtual Storage Profile) properties must be configured on this port and the destination buffer pool is initialized and managed by the destination software entity. Therefore, these buffer pools are not initialized or managed by the Offline Port Driver. In some scenarios, this can be done by one of DPAA Ethernet Drivers. If the O/H Port does not use IP Fragmentation or VSP capabilities, this attribute will not be used;
- *fsl,qman-frame-queues-oh* - a list of base/count pairs of frame queues through which the frames are transmitted from the O/H Port to the next hardware or software module. These queues are not initialized, but are solely used as ID references for O/H Port initialization. This is necessary because the initialization of an FMan Port (online or offline) requires a TX error and a TX default frame queues. These frame queues must be initialized by other software module. This device tree property is mandatory;
- *fsl,qman-frame-queues-tx* - deprecated
- *fsl,qman-frame-queues-ingress* - a base/count pair of frame queues to be initialized by the Offline Port Driver that will fetch the frames from the hardware or software entity to the O/H Port.
- *fsl,qman-frame-queues-egress* - a list of base/count pairs of frame queues through which the frames are transmitted from the O/H Port to the next hardware or software module. These queues are connected only to DC portals therefore the property below (qman-channel-ids-egress) is mandatory. Frame queues connected to SW portals should be created from another software entity. Also, if a frame queue uses different initialization parameters it should be created by another software entity.
- *fsl,qman-channel-ids-egress* - a list of channel ids used together with "fsl,qman-frame-queues-egress" option to configure the egress frame queues.
- *fsl,fman-oh-port* - this is the reference to the O/H Port device tree node and has the same meaning as *fsl,fman-mac* from DPAA Ethernet Driver device tree specification. This attribute is mandatory.

Note 1: It should be noted that the Offline Port Driver does not initialize frame queues specified through *fsl,qman-frame-queues-oh* attribute. When the port is probed, the O/H Port is enabled and configured with the values from the device tree, but no action is taken on the declared frame queues.

Note 2: The statically defined buffer pool should be declared in a similar way as for the DPAA Ethernet Drivers.

The O/H Ports can be used in multiple scenarios as nodes that fetch traffic from multiple sources or as intermediate nodes that multiplexes incoming traffic to multiple destinations.

## 20.1.4.2 Features

The Offline Port Driver is an initialization driver for the O/H Ports. This chapter presents only the Offline Port Driver initialization capabilities, not the hardware features offered by the O/H Ports.

#### TX queues initialization

Before SDK 1.5, no frame queue referenced by the Offline Port Driver's device tree node was initialized. It was other driver's job to initialize the frame queues used by the O/H Port. As of SDK 1.5, *fsl,qman-frame-queues-tx* and *fsl,qman-channel-id* attributes were introduced. The frame queues referenced by the above mentioned *fsl,qman-frame-queues-tx* attribute are managed by the Offline Port Driver, decreasing thus the complexity of other drivers.

Linux Ethernet Driver for DPAA 1.x Family

## 20.1.5 Link Management

The FMan has two types of ethernet controllers: 1G and 10G. The two types of ethernet conform to different standards (802.3 Clause 22 and Clause 45, respectively) for link management, so there are two corresponding types of MDIO controller for those standards. Each ethernet controller has its own MDIO device, however only one of each type are pinned out on currently shipping parts (10/2011). On P4080, for example, only the MDIO on FM1's first 1G MAC is pinned out, as well as the MDIO on FM1's 10G MAC. This is true even if FM1's first 1G MAC or 10G MAC is disabled.

On the 1G MACs, the MDIO controllers serve a second purpose -- configuring the SERDES link for SGMII between the MAC and the external PHY. This is done via management commands to an on-chip "TBI" PHY. This PHY is configured to sit at the address written to the TBIPA register in the MAC, and all MDIO transactions from that MAC to that address will be intercepted by the TBI PHY. This means that the setting of this register on the first 1G MAC is very relevant to system architects, as the address must not conflict with the address of an external PHY, or that PHY will not be reachable by MDIO management commands.

### 20.1.5.1 Device Tree

The MDIO nodes are described in the device tree, and look like this:

```
           mdio0: mdio@e1120 {
                    #address-cells = <1>;
                    #size-cells = <0>;
                    compatible = "fsl,fman-mdio";
                    reg = <0xe1120 0xee0>;
                    interrupts = <100 1 0 0>;
                    ethernet-phy@0 {
                               reg = <0x0>;
                    };
              tbi0: tbi-phy@8 {
                     reg = <0x8>;
                     device_type = "tbi-phy";
              };
       };
```

This is one of the 1G MDIO nodes for a P4080 (Not the DS, but more on that later). It specifies that there is a PHY at address 0, and the TBI PHY will sit at address 8. Note that it is the first MAC, which is the only one whose pins are connected outside the chip. The other 7 MACs would have a similar node, but with only a TBI PHY in each. This is how one would set up PHYs under a standard MDIO bus topology; the first MAC connects to all of the PHYs (up to 31 of them), and also has its assigned internal TBI PHY.

In order to associate an ethernet controller with a PHY, one uses the phy-handle property

### 20.1.5.2 Bootargs

```
fsl_fm_max_frm
```

Configure this to set the L2 Maximum Frame Size. This influences the Frame Manager FIFO size resources.

```
fsl_fm_rx_extra_headroom
```

Configure this to tell the Frame Manager to reserve some extra space at the beginning of a data buffer on the receive path, before Internal Context fields are copied. This is in addition to the private data area already reserved

Linux Ethernet Driver for DPAA 1.x Family

for driver internal use. The option does not affect in any way the layout of transmitted buffers. The default value (64bytes) offers best performance for the case when forwarded frames are being encapsulated (e.g. IPSec). For plain forwarding or termination cases, a value of zero is recommended for optimum performance.

The current size of the buffers in USDPAA dts files is 1728bytes (odd_cache_line * cache_line_size) which is calculated according to the default value (64bytes) of the dpa_extra_headroom. If the dpa_extra_headroom is set to 0 then the buffer size will be 1600bytes (also multiple of odd cache line). For any extra headroom values ranging between 1 and 128bytes the buffer size is 1728bytes. For values ranging between 129bytes and 256bytes, the buffer size is 1856bytes and so on.

### 20.1.5.3 Muxed MDIO

The Development Systems for the QorIQ product lines do not have a standard MDIO topology. In order to support the variety of configurations that are possible under the QorIQ product line, the PHYs often reside on riser cards, many of which use the same addresses; this means that it is not possible to keep all of the PHYs on the same bus. To work around this problem, the MDIO buses have been muxed so that, at any one time, only a subset of the PHYs will be connected to the MDIO buses. In order to send an MDIO transaction to a PHY, it is therefore necessary to first modify the MUX. This is done via a set of "virtual" MDIO interfaces. Each MUX setting is considered a separate bus, each with its own PHYs. That means that the P4080DS's MDIO node looks like this:

```
mdio0: mdio@e1120 {
        gpios = <&gpio0 0 0
                  &gpio0 1 0>;
        tbi0: tbi-phy@8 {
                reg = <0x8>;
                device_type = "tbi-phy";
        };
        p4080mdio0: p4080ds-mdio0 {
                #address-cells = <1>;
                #size-cells = <0>;
                compatible = "fsl,p4080ds-mdio";
                fsl,mdio-handle = <&mdio0>;
                fsl,muxval = <0>;
                phyrgmii: ethernet-phy@0 {
                        reg = <0x0>;
                };
        };
};
```

The bus now has a subnode, "p4080ds-mdio0", which represents the bus topology when the MUX is set to 0. That bus connects to the RGMII PHY at address 0. Other buses connect to the SGMII PHYs in the 3 riser card slots. It's important to note that it is not at all required to use a muxed MDIO bus on QorIQ products; this is only a solution to the problem of trying to access PHYs which can be in multiple places. The other important thing to note is that U-Boot modifies the device tree. In the example of the P4080DS, U-Boot uses the RCW to determine which interfaces are enabled, and which PHYs those interfaces are connected to. Therefore the connections in the dts file may not be the final arrangements of the PHYs.

## 20.1.6 Debugging

This chapter describes the debugging capabilities of the DPAA Ethernet driver.

Linux Ethernet Driver for DPAA 1.x Family

## 20.1.6.1 Debugfs support

### Debugfs Introduction

Debugfs is a virtual filesystem used to export debugging information to user space. It is available in Linux kernel since 2.6.10-rc3.

### Activating debugfs support

To compile a Linux kernel with the debugfs facility, the CONFIG_DEBUG_FS option must be set. Additionally, at runtime, debugfs is typically mounted in /sys/kernel/debug with the following command line:

```
mount -t debugfs none /sys/kernel/debug/
```

### Functionality

There is an entry in debugfs for each interface in the system showing various counters 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 a debugfs output:

```
root@p4080ds:~# cat /sys/kernel/debug/fsl_dpa/eth0

DPA counters for fm1-gb1:
CPU           irqs        rx        tx   recycle   confirm     tx sg    tx err   ...
     0/0         5         2         2         0         3         0         0
     1/1         6         2         0         0         4         0         0
     2/2         6         5         4         0         1         0         0
     3/3         6         2         1         0         4         0         0
     4/4         6         5        12         0         1         0         0
     5/5         6         2         0         0         4         0         0
     6/6         6         4         3         0         2         0         0
     7/7         6         3         0         0         3         0         0
Total           47        25        22         0        22         0         0

Device congestion stats:
Device has been congested for 0 ms.
CGR id 0 avg count: 0
Device entered congestion 0 times. Current congestion state is: not congested.

DPA RX Errors:
CPU        dma err  phys err  size err   hdr err  csum err
     0/0         0         0         0         0         0
     1/1         0         0         0         0         0
     2/2         0         0         0         0         0
     3/3         0         0         0         0         0
     4/4         0         0         0         0         0
     5/5         0         0         0         0         0
     6/6         0         0         0         0         0
     7/7         0         0         0         0         0
Total            0         0         0         0         0

DPA ERN counters:
  CPU     cg_td      wred  err_cond   early_w    late_w     fq_td    fq_ret   ...
  0/0         0         0         0         0         0         0         0
  1/1         0         0         0         0         0         0         0
  2/2         0         0         0         0         0         0         0
```

Linux Ethernet Driver for DPAA 1.x Family

```
  3/3         0         0         0         0         0         0         0
  4/4         0         0         0         0         0         0         0
  5/5         0         0         0         0         0         0         0
  6/6         0         0         0         0         0         0         0
  7/7         0         0         0         0         0         0         0
Total         0         0         0         0         0         0         0
```

### 20.1.6.2 Read/Write of FMan Registers

Most of the FMan configuration registers are mapped into the system memory space. Efficient debugging and testing can be done by making read/write operations on the registers through specialized tools. For example, the number of pause frames received on a particular MAC device can be computed summing the base relative address of every component:

```
0xffe000000 (the address of the SoC) +
   0x400000 (FMan 1) +
    0xe8000 (MAC 5) +
      0x234 (RX pause frames) =
-------------------------------------
0xffe4e8234
```

A memory print of the 0xffe4e8234 address will display the number of pause frames received by the fifth MAC device from the first FMan on a P4080DS platform.

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

### 20.1.6.3 Sysfs support

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

```
root@p4080ds:~# cat /sys/devices/fsl,dpaa.16/ethernet.18/net/fm2-gb0/bpids
32
root@p4080ds:~# cat /sys/devices/fsl,dpaa.16/ethernet.17/net/fm1-gb1/fqids
Rx error: 262
Rx default: 263
Rx PCD: 14464 - 14591
Tx confirmation (mq): 264 - 271
Tx error: 272
Tx default confirmation: 273
Tx: 274 - 281
```

## 20.1.7 Adding support for DPAA Ethernet in Topaz Hypervisor

We start from the assumption that hv.dts file for the platform has already been created starting from an existing platform. The practice is to place the device tree under the following folder hierarchy: hv-cfg/"platform_name"/"RCW_name" It means that the RCW gives the number and types of ports that will be added in hv.dts. We will consider T1040RDB hv.dts as a reference throughout this guide.

Following are the steps to be followed in order to add DPAA ports (private, shared-mac and macless) in hv.dts:

Linux Ethernet Driver for DPAA 1.x Family

1. Add bman-portal and qman-portal nodes under "part 1" node. E.g.:

```
bman-portal@0 {
    device = "/bman-portals/bman-portal@0";
};

qman-portal@0 {
    device = "/qman-portals@ff6000000/qman-portal@0";

    stash-mem {
        liodn-index = <1>;
        dma-window = <&dw_linux1>;
        operation-mapping = <0>;
        stash-dest = <3>;
    };

    stash-dqrr {
        liodn-index = <0>;
        dma-window = <&dw_dqrr_qportal0>;
        operation-mapping = <0>;
        stash-dest = <3>;
    };
};
```

   Make sure dma-window "dw_dqrr_qportal0" exists under "dma-windows" node:

```
// DMA window for stash_dqrr for qman-portal0
dw_dqrr_qportal0: window3 {
    compatible = "dma-window";
    guest-addr = <0xf 0xf6000000>;
    size = <0 0x4000>;
};
```

   All bman-portal and qman-portal nodes must be added except at least one which should be added to the second partition. For T1040RDB, bman-portal@1c000 and qman-portal@1c000 were added in the second partition. The list of available bman-portal and qman-portal nodes for T1040RDB can be retrieved from the following dtsi file included by the platform device tree: arch/powerpc/boot/dts/fsl/t1040si-post.dtsi bman-portals are located under bportals node and qman-portals under qportals node.

2. For the current RCW that is used, add the appropriate dpaa ethernet node under "part 1" parent node. For T1040RDB the following ports are added:

```
// FMAN0 RGMII -- ethernet 3 assigned to this partition
dpa-ethernet@3 {
    device = "/fsl,dpaa/ethernet@3";
};


// FMAN0 RGMII -- ethernet 4 assigned to this partition
dpa-ethernet@4 {
    device = "/fsl,dpaa/ethernet@4";
};
```

Linux Ethernet Driver for DPAA 1.x Family

The full list of available ethernet ports can be retrieved from the platform device tree: arch/powerpc/boot/dts/t1040rdb.dts

3. bman, qman, fman0 and fman1 (in case there are two FMANs) nodes must be also added under "part 1" parent node. See T1040RDB example.
4. fman0 and fman1 (if applicable) must be added under "portal-devices" node. See T1040RDB example.
5. Under "node-update" node make sure you define at least two buffer pools for macless and shared-mac interfaces. While here, make sure gpma1 is defined and guest-addr reserves the inter-partition memory area. Also check that dw_linux1 and dw_linux2 both have sub-window nodes with the same memory area region matching the inter-patition area. See T1040RDB example.
6. Make sure "fsl,dpaa" node exists under "node-update" node and check or create a macless and a shared-mac interface. The compatible string for the "fsl,dpaa" node must be: compatible = "fsl,t1040-dpaa", "fsl,dpaa"; replace t1040 with the name of the platform you want to add. The compatible string for the macless interface must be: compatible = "fsl,t1040-dpa-ethernet", "fsl,dpa-ethernet-macless"; The compatible string for the macless interface must be: compatible = "fsl,t1040-dpa-ethernet", "fsl,dpa-ethernet-shared"; Make sure the macless node includes the local-mac-address identifier. See T1040RDB example.
7. Make sure "fsl,dpaa" node exists under "node-update-phandle" node. Here the buffer pools are associated with macless and shared-mac interfaces. See T1040RDB example.
8. Under "part 1" node check or add the following nodes: bman-bpids, qman-fqids@0, qman-fqids@1, qman-pools, qman-cgrids The values used by these nodes can be retrieved from arch/powerpc/boot/dts/fsl/qoriq-dpaa-res3.dtsi For T1040RDB two partitions were created therefore the resources, bpids, fqids, etc were splitted among the two partitions. In case more partitions are created the resources must be splitted accordingly. See T1040RDB example.
9. Repeat steps 3-8 for "part 2" node in case of 2 partitions scenario. The same steps must be followed in case more than 2 partitions are added. Exception for step 6: Two macless nodes must be added under "fsl,dpaa" node. The first node is the "pair" of the macless interface in the first partition and the second is the interface connected to the shared-mac port (the port shared with the first partition). See T1040RDB example.

## 20.1.8 MACsec

#### Introduction

This chapter provides information about the MACsec kernel module and user space API that can be used to configure and control the MACsec hardware block inside the Frame Manager. The Quick Start Guide lists the steps that need to be taken in order to enabled MACsec after integrating the API into the hostapd and wpa_supplicant applications. The API Reference section details the user space MACsec API and gives examples on how to use it.

#### Intended Audience

This chapter is intended for software developers and architects who want to develop software applications that implement the 802.1X and 802.1AE (MACsec) protocols.

### 20.1.8.1 MACsec User Space API Reference

The following API is implemented in the hostapd and wpa_supplicant user space applications and can be reached by including the src/drivers/dpaa_utils_macsec.h header.

#### MACsec API data structures

Linux Ethernet Driver for DPAA 1.x Family

1. enable_macsec_t

Member's description:

- if_name - char* that holds the name of the interface on which MACsec is to be enabled
- if_name_length - size_t that holds the length of the if_name field, including the terminating null character
- config_unknown_sci_treatment - boolean that selects between the default value and the one in unknown_sci_treatment
- unknown_sci_treatment - enum that selects how frames received with unknown SCI are treated (default is DISCARD_BOTH - discarded on both the controlled and uncontrolled ports)
- config_invalid_tag_treatment - boolean that selects between the default value and the one in deliver_uncontrolled
- deliver_uncontrolled - boolean that selects if frames received with an invalid SecTAG or a zero PN value should be delivered on the uncontrolled port (default is FALSE)
- config_kay_frame_treatment - boolean that selects between the default value and the one in discard_uncontrolled
- discard_uncontrolled - boolean that selects if frames received with the Encryption (E) bit set and the Changed Text (C) bit clear (reserved for use by the KaY) should be discarded on the uncontrolled port (default is FALSE)
- config_untag_treatment - boolean that selects between the default value and the one in untag_treatment
- untag_treatment - enum that selects how frames received without the MAC SecTAG are treated (default is UNTAG_DELIVER_UNCTRL_DISCARD_CTRL)
- config_pn_exhaustion_threshold - boolean that selects between the default value and the one in pn_threshold
- pn_threshold - u32 that sets the TX PN exhaustion threshold (default is 0xffffffff)
- config_keys_unreadable - boolean that selects if the RX and TX keys and hash values should be unreadable (default is FALSE)
- config_sectag_without_sci - boolean that selects if the maximum frame length should not consider the SCI's size in the SecTAG (default is FALSE)
- config_exception - boolean that selects between the default values and the ones in enable_exception and exception
- enable_exception - boolean that selects if the exception mask set by the exception field should be enabled or not
- exception - enum that defines the mask of the exception that is to be enabled or disabled (default is SINGLE_BIT_ECC | MULTI_BIT_ECC - both masks are enabled)

2. enable_secy_t

Member's description:

- macsec_id - int value returned by the dpa_macsec_enable() call that identifies one MACsec instance per interface
- sci - 64bit value made from the MAC address of the interface on which MACsec has been enabled, concatenated with a Port Identifier
- config_insertion_mode - boolean that selects between the default value and the one in sci_insertion_mode
- sci_insertion_mode - enum that selects if the SCI will be included in the SecTag or not (default is SCI_INSERTION_MODE_EXPLICIT_SECTAG - the SCI is always included)

Linux Ethernet Driver for DPAA 1.x Family

- config_protect_frames - boolean that selects between the default value and the one in protect_frames
- protect_frames - boolean that selects if the packet will be encapsulated with MACsec header (default is TRUE)
- config_replay_window - boolean that selects between the default values and the ones in replay_window and replay_protect
- replay_protect - boolean that selects if the replay protection algorithm is enabled or not (default is FALSE)
- replay_window - unsigned integer that represents the replay window dimension (default is 0)
- config_validation_mode - boolean that selects between the default value and the one in validate_frames
- validate_frames - enum that selects the behaviour of MACsec from frames validation point of view (default is VALID_FRAME_BEHAVIOR_STRICT - strictly filter out invalid frames)
- config_confidentiality - boolean that selects between the default values and the ones in confidentiality_enable and confidentiality_offset
- confidentiality_enable - boolean that selects if the packages will be encrypted or not (default is FALSE)
- confidentiality_offset - unsigned integer that represents the offset from which the data will be encrypted (default is 0)
- config_point_to_point - boolean that selects if MACsec must be configured only for point-to-point communication (default is FALSE)
- config_exception - boolean that selects between the default values and the ones in enable_exception and exception
- enable_exception - boolean that selects if the exception mask set by the exception field should be enabled or not
- exception - enum that defines the mask of the exception that is to be enabled or disabled (default is FRAME_DISCARDED)
- config_event - boolean that selects between the default values and the ones in enable_event and event
- enable_event - boolean that selects if the event mask set by the event field should be enabled or not
- event - enum that defines the mask of the event that is to be enabled or disabled (default is NEXT_PN)

**MACsec API functions**

**1. dpa_macsec_enable**

```
int dpa_macsec_enable(enable_macsec_t en_macsec)
```

Description:

- used to configure and enable MACsec on a certain interface

Parameters:

- en_macsec - MACsec data structure including the name of the interface to enable MACsec on and several configuration options

Return:

- returns the macsec_id that will be used in further calls of the API's functions
- returns a negative value in case something went wrong

**2. dpa_macsec_secy_en**

```
int dpa_macsec_secy_en(enable_secy_t enable_secy)
```

Linux Ethernet Driver for DPAA 1.x Family

Description:

- used for configuring and creating SecY; also creates a corresponding TxSc (secure channel for TX)
- sets the default values for cipher suite, SCI insertion mode, protect frames, replay protection, replay window dimension, confidentiality, confidentiality offset and point-to-point connection, as IEEE802.1AE-2006 states

Parameters:

- enable_secy - SecY data structure

Return:

- returns the sc_id allocated by MACsec (a value from 15 to 0, or simply 0, if the setup is point-to-point) that uniquely identifies the SecY
- returns a negative value in case something went wrong

3. dpa_macsec_secy_create_tx_sa

```
int dpa_macsec_secy_create_tx_sa(int macsec_id, u8 an, u8 *sak, u32 sak_len)
```

Description:

- used for creating a TxSa (secure association for TX)

Parameters:

- macsec_id - MACsec interface identifier
- an - association number
- sak - secure association key (the key used for encryption)
- sak_len - length of the key (must be at most 32)

Return:

- returns 0 on success or a negative value in case something went wrong

4. dpa_macsec_secy_activate_tx_sa

```
int dpa_macsec_secy_activate_tx_sa(int macsec_id, u8 an)
```

Description:

- used for activating the secure association for TX channel

Parameters:

- macsec_id - MACsec interface identifier
- an - association number

Return:

- returns 0 on success or a negative value in case something went wrong

5. dpa_macsec_secy_create_rx_sc

```
int dpa_macsec_secy_create_rx_sc(int macsec_id, u64 sci)
```

Description:

- creates a RxSc (secure channel for RX)

Parameters:

Linux Ethernet Driver for DPAA 1.x Family

- macsec_id - MACsec interface identifier
- sci - a 64 bit value made of the MAC address of the destination(48 bits) and a port number(16 bits)

Return:

- returns the rx_sc_id allocated by MACsec (a value from 15 to 0, or simply 0, if the setup is point-to-point) that uniquely identifies the peer to whom we are discussing MACsec
- returns a negative value in case something went wrong

#### 6. dpa_macsec_secy_create_rx_sa

```
int dpa_macsec_secy_create_rx_sa(int macsec_id, u32 rx_sc_id, u8 an, u32 lpn, u8 *sak,
u32 sak_len)
```

Description:

- creates RxSa (secure association for RX)

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number
- lpn - lowest packet number, value used in anti-replay algorithm
- sak - secure association key (the key used for decryption)
- sak_len - length of the key (must be at most 32)

Return:

- returns 0 on success or a negative value in case something went wrong

#### 7. dpa_macsec_secy_activate_rx_sa

```
int dpa_macsec_secy_activate_rx_sa(int macsec_id, u32 rx_sc_id, u8 an)
```

Description:

- used for activating the secure association for RX channel

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number

Return:

- returns 0 on success or a negative value in case something went wrong

#### 8. dpa_macsec_secy_disable_rx_sa

```
int dpa_macsec_secy_disable_rx_sa(int macsec_id, u32 rx_sc_id, u8 an)
```

Description:

- used for disabling the secure association for RX channel

Parameters:

- macsec_id - MACsec interface identifier

Linux Ethernet Driver for DPAA 1.x Family

- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number

Return:

- returns 0 on success or a negative value in case something went wrong

**9. dpa_macsec_secy_delete_rx_sa**

```
int dpa_macsec_secy_delete_rx_sa(int macsec_id, u32 rx_sc_id, u8 an)
```

Description:

- used for deleting the secure association for RX channel

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number

Return:

- returns 0 on success or a negative value in case something went wrong

**10. dpa_macsec_secy_delete_rx_sc**

```
int dpa_macsec_secy_delete_rx_sc(int macsec_id, u32 rx_sc_id)
```

Description:

- used for deleting the secure channel for RX

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call

Return:

- returns 0 on success or a negative value in case something went wrong

**11. dpa_macsec_secy_delete_tx_sa**

```
int dpa_macsec_secy_delete_tx_sa(int macsec_id, u8 an)
```

Description:

- used for deleting the secure association for TX channel

Parameters:

- macsec_id - MACsec interface identifier
- an - association number

Return:

- returns 0 on success or a negative value in case something went wrong

**12. dpa_macsec_secy_disable**

```
int dpa_macsec_secy_disable(int macsec_id)
```

Linux Ethernet Driver for DPAA 1.x Family

Description:

- used for deleting the SecY and disabling the associated TxSc

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns 0 on success or a negative value in case something went wrong

**13. dpa_macsec_disable**

```
int dpa_macsec_disable(int macsec_id)
```

Description:

- used for disabling MACsec

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns 0 on success or a negative value in case something went wrong

**14. dpa_macsec_disable_all**

```
int dpa_macsec_disable_all(int macsec_id)
```

Description:

- used as a single call for disabling all that was enabled for MACsec to work
- is the equivalent for calling dpa_macsec_secy_disable_rx_sa, dpa_macsec_secy_delete_rx_sa, dpa_macsec_secy_delete_rx_sc, dpa_macsec_secy_delete_tx_sa, dpa_macsec_secy_disable and dpa_macsec_disable one by one

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns 0 on success or a negative value in case something went wrong

**15. dpa_macsec_secy_get_txsc_phys_id**

```
int dpa_macsec_secy_get_txsc_phys_id(int macsec_id)
```

Description:

- obtain the ID associated with the TX SC by the Fman

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns the tx_sc_id corresponding to the SecY from Fman
- returns a negative value in case something went wrong

Linux Ethernet Driver for DPAA 1.x Family

16. dpa_macsec_secy_modify_txsa_key

```
int dpa_macsec_secy_modify_txsa_key(int macsec_id, u8 an, u8 *sak, u32 sak_len)
```

Description:

- used to change the encryption key

Parameters:

- macsec_id - MACsec interface identifier
- an - association number
- sak - new secure association key (the key used for encryption)
- sak_len - length of the new key (must be at most 32)

Return:

- returns 0 on success or a negative value in case something went wrong

17. dpa_macsec_secy_modify_rxsa_key

```
int dpa_macsec_secy_modify_rxsa_key(int macsec_id, u32 rx_sc_id, u8 an, u8 *sak, u32
sak_len)
```

Description:

- used to change the decryption key

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number
- sak - new secure association key (the key used for decryption)
- sak_len - length of the new key (must be at most 32)

Return:

- returns 0 on success or a negative value in case something went wrong

18. dpa_macsec_secy_get_tx_sa_active_an

```
int dpa_macsec_secy_get_tx_sa_active_an(int macsec_id)
```

Description:

- used to get the active association number for this TxSc

Parameters:

- macsec_id - MACsec interface identifier

Return:

- returns the association number on success or a negative value in case something went wrong

19. dpa_macsec_secy_get_rxsc_phys_id

```
int dpa_macsec_secy_get_rxsc_phys_id(int macsec_id, u32 rx_sc_id)
```

Description:

Linux Ethernet Driver for DPAA 1.x Family

- obtain the ID associated with the RX SC by the Fman

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call

Return:

- returns the rx_sc_id corresponding to the peer's SecY from Fman
- returns a negative value in case something went wrong

### 20. dpa_macsec_secy_rx_sa_update_npn

```
int dpa_macsec_secy_rx_sa_update_npn(int macsec_id, u32 rx_sc_id, u8 an, u32 pn)
```

Description:

- used to set the next packet number that MACsec should handle on RX

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number
- pn - packet number

Return:

- returns 0 on success or a negative value in case something went wrong

### 21. dpa_macsec_secy_rx_sa_update_lpn

```
int dpa_macsec_secy_rx_sa_update_lpn(int macsec_id, u32 rx_sc_id, u8 an, u32 pn)
```

Description:

- used to set the lowest packet number that MACsec should handle on RX

Parameters:

- macsec_id - MACsec interface identifier
- rx_sc_id - the one received from MACsec after dpa_macsec_secy_create_rx_sc call
- an - association number
- pn - packet number

Return:

- returns 0 on success or a negative value in case something went wrong

### 22. dpa_macsec_get_revision

```
int dpa_macsec_get_revision(int macsec_id)
```

Description:

- used to find the current revision of MACsec

Parameters:

- macsec_id - MACsec interface identifier

Linux Ethernet Driver for DPAA 1.x Family

Return:

- returns the revision corresponding to the macsec_id
- returns a negative value in case something went wrong

### 23. dpa_macsec_set_exception

```
int dpa_macsec_set_exception(int macsec_id, bool enable_ex, macsec_exception ex)
```

Description:

- used to enable/disable a trigger for a certain exception

Parameters:

- macsec_id - MACsec interface identifier
- enable_ex - bool that will select if the exception given by the third argument will be enabled or not
- ex - the exception for wich a trigger will be activated/deactivated

Return:

- returns 0 on success or a negative value in case something went wrong

### MACsec API usage examples

### 1. Basic usage

```
        enable_macsec_t en_macsec;
        enable_secy_t enable_secy;
        int i, rx_sc_id;
        u8 an = 0;
        u32 lpn = 0;
        u32 sa_key_len = 32;
        u16 port_src = 1;
        u16 port_dst = 1;
        u64 sci_src; //MAC + port - 64b
        u64 sci_dest; //MAC + port - 64b

        u8 *sa_key = malloc(sa_key_len * sizeof(u8));
        for (i = 0; i < sa_key_len; i ++)
            sa_key[i] = 0x12;

        sci_src = (mac_src << 16) | port_src;
        sci_dest = (mac_dst << 16) | port_dest;

        en_macsec.if_name = ifname;
        en_macsec.if_name_length = strlen(ifname) + 1;

        en_macsec.config_unknown_sci_treatment = FALSE;
        en_macsec.config_invalid_tag_treatment = FALSE;
        en_macsec.config_kay_frame_treatment = FALSE;
        en_macsec.config_untag_treatment = FALSE;
        en_macsec.config_pn_exhaustion_threshold = FALSE;
        en_macsec.config_keys_unreadable = FALSE;
        en_macsec.config_sectag_without_sci = FALSE;
        en_macsec.config_exception = FALSE;
        en_macsec.enable_exception = FALSE;

        macsec_id = dpa_macsec_enable(en_macsec);
```

Linux Ethernet Driver for DPAA 1.x Family

```
        enable_secy.macsec_id = macsec_id;
        enable_secy.sci = sci_src;

        enable_secy.config_insertion_mode = FALSE;
        enable_secy.config_protect_frames = FALSE;
        enable_secy.config_replay_window = FALSE;
        enable_secy.config_validation_mode = FALSE;
        enable_secy.config_confidentiality = FALSE;
        enable_secy.config_point_to_point = FALSE;
        enable_secy.config_exception = FALSE;
        enable_secy.config_event = FALSE;

        dpa_macsec_secy_en(enable_secy);
        dpa_macsec_secy_create_tx_sa(macsec_id, an, sa_key, sa_key_len);
        dpa_macsec_secy_activate_tx_sa(macsec_id, an);

        rx_sc_id = dpa_macsec_secy_create_rx_sc(macsec_id, sci_dest);
        dpa_macsec_secy_create_rx_sa(macsec_id, rx_sc_id, an, lpn, sa_key, sa_key_len);
        dpa_macsec_secy_activate_rx_sa(macsec_id, rx_sc_id, an);

        /* ... */

        free(sa_key);

        dpa_macsec_disable_all(macsec_id);

        /* Or:
        * dpa_macsec_secy_disable_rx_sa(macsec_id, rx_sc_id, an);
        * dpa_macsec_secy_delete_rx_sa(macsec_id, rx_sc_id, an);
        * dpa_macsec_secy_delete_rx_sc(macsec_id, rx_sc_id);
        * dpa_macsec_secy_delete_tx_sa(macsec_id, an);
        * dpa_macsec_secy_disable(macsec_id);
        * dpa_macsec_disable(macsec_id);
        */
```

## 2. Enable confidentiality

Enable the confidentiality flag and set the confidentiality offset in the enable_secy_t structure.

```
        enable_secy.macsec_id = macsec_id;
        enable_secy.sci = sci_src;

        enable_secy.config_insertion_mode = FALSE;
        enable_secy.config_protect_frames = FALSE;
        enable_secy.config_replay_window = FALSE;
        enable_secy.config_validation_mode = FALSE;

        enable_secy.config_confidentiality = TRUE;
        enable_secy.confidentiality_enable = TRUE;
        enable_secy.confidentiality_offset = 0;

        enable_secy.config_point_to_point = FALSE;
        enable_secy.config_exception = FALSE;
        enable_secy.config_event = FALSE;
```

## 3. Modify the TxSa and the RxSa keys

Linux Ethernet Driver for DPAA 1.x Family

Generate a new set of keys and call the appropriate API functions.

```
        for(i = 0; i < sa_key_len; ++i) {
            txsa_key[i] = 0x23;
        }

        dpa_macsec_secy_modify_txsa_key(macsec_id, an, txsa_key, sa_key_len);

        for(i = 0; i < sa_key_len; ++i) {
            rxsa_key[i] = 0x57;
        }

        dpa_macsec_secy_modify_rxsa_key(macsec_id, rx_sc_id, an, rxsa_key, sa_key_len);
```

## 20.1.8.2 MACsec Quick Start Guide

### Software Installation and Build

- Compile MACsec as a module:

```
        Device Drivers
            +-> Network device support (NETDEVICES [=y])
                +-> Ethernet driver support (ETHERNET [=y])
                    +-> Freescale devices (NET_VENDOR_FREESCALE [=y])
                        +-> DPAA Ethernet (FSL_DPAA_ETH [=y])
                            +-> DPAA Macsec [=m]
```

- Increase the Fman maximum frame size to 1554 bytes. This is needed in order to accommodate the MACsec header without decreasing the MTU:

```
        Device Drivers
            +-> Network device support (NETDEVICES [=y])
                +-> Ethernet driver support (ETHERNET [=y])
                    +-> Freescale devices (NET_VENDOR_FREESCALE [=y])
                        +-> Frame Manager Support
                            +-> Freescale Frame Manager (datapath) support (FSL_FMAN 
[=y])
                                +-> Maximum L2 frame size [=1554]
```

- Activate Fman_v3l (only needed by for T1040 platforms):

```
        Device Drivers
            +-> Network device support (NETDEVICES [=y])
                +-> Ethernet driver support (ETHERNET [=y])
                    +-> Freescale devices (NET_VENDOR_FREESCALE [=y])
                        +-> Frame Manager Support
                            +-> Freescale Frame Manager (datapath) support (FSL_FMAN 
[=y])
                                +-> FMAN_V3L like T1040, T1042, T1020, T1022 [=y]
```

Linux Ethernet Driver for DPAA 1.x Family

- Build the HostAP and wpa_supplicant

  Create a .bbappend file as described in Yocto's documentation and add the two packages to the image:

```
IMAGE_INSTALL += "hostapd"
IMAGE_INSTALL += "wpa-supplicant"
```

### Execution setup

- Generate a set of OpenSSL keys and certificates for the 802.1X authentication.
- Create a configuration file for the HostAP.

  A separate configuration file is needed for each interface on which the hostapd will run. hostapd.conf example:

```
ctrl_interface=/var/run/hostapd
ctrl_interface_group=0
use_pae_group_addr=1

# Replace the following with the appropriate interface name
interface=fm1-gb3

driver=wired
logger_syslog=-1
logger_syslog_level=2
logger_stdout=-1
logger_stdout_level=2
dump_file=/tmp/hostapd.dump

eapol_version=2
ieee8021x=1
eap_server=1

# Replace the following with the path to the eap_user_file
eap_user_file=/etc/hostapd.eap_user

# Replace the following with your CA certificate path
ca_cert=/etc/server.crt
server_cert=/etc/server.crt
private_key=/etc/server.key
```

- Create an eap_user_file for the HostAP.

  hostapd.eap_user example:

```
# Phase 1 users
* PEAP
# Replace the following with your desired credentials
"test" MSCHAPV2,MD5,PEAP "password" [2]
```

- Create a configuration file for the wpa_supplicant.

Linux Ethernet Driver for DPAA 1.x Family

wpa_supplicant.conf example:

```
        ctrl_interface=/var/run/wpa_supplicant
        ctrl_interface_group=0
        ap_scan=0
        fast_reauth=1
        eapol_version=2

        network={
            ssid=""
            key_mgmt=IEEE8021X
            scan_ssid=0
            eap=PEAP
            priority=100
            phase2="auth=MSCHAPV2"

            # Replace the following with the credentials from the HostAP's
eap_user_file
            identity="test"
            password="password"

            # Replace the following with your CA certificate path
            ca_cert="/etc/server.crt"
        }
```

- Boot the board according to the platform's documentation.
- Insert the fsl-dpa-macsec module.

  Specify in a comma separated list the interfaces on which MACsec will be enabled:

```
        modprobe fsl-dpa-macsec ifs="fm1-gb3,fm1-gb4"
```

- Start the HostAP.

```
        # Replace with the path to the configuration file
        /usr/sbin/hostapd /etc/hostapd.conf &
```

- Start the wpa_supplicant.

```
        # Replace with the path to the configuration file and the appropriate
interface name
        wpa_supplicant -Dwired -ifm1-gb3 -c/etc/wpa_supplicant.conf &
```

- Inspect MACsec traffic statistics in debugfs.

```
        cat /sys/kernel/debug/fsl_macsec_dpa/fm1-gb3
        Macsec counters for fm1-gb3:
        CPU       tx        rx
        0         0         0
```

Linux Ethernet Driver for DPAA 1.x Family

```
        1        0        0
        Total:     0        0
```

**Limitations and known issues**

- MACsec statistics for RX path are not supported in debugfs.
- MACsec is not functional on ports connected to the internal L2switch on T1040 platforms. Consult your platform's specifications to determine which ports have MACsec support.

## 20.1.9 Changes from previous versions

- SDK1.8
  - *MACsec*

    Added MACsec kernel driver and a set of User-space Linux APIs to allow developers to configure the MACsec hardware block.
  - *DPAA Ethernet as loadable modules*

    Added loadable module support for the DPAA Ethernet driver.
  - *Adding support for DPAA Ethernet in Topaz Hypervisor*

    This is a documentation section enumerating the steps to follow in order to enable DPAA Ethernet in Topaz Hypervisor.
- SDK1.7
  - *PCD configuration files*

    Included in the SDK package PCD configuration and policy files (.xml files) for each platform supported by the SDK. The configuration files include all the ports enabled by the RCWs associated with each platform.
- SDK1.6
  - *oNIC device driver - DPAA hardware offloading aware Ethernet net device*

    Added oNIC netdevice, based on OH FMAN ports in order to offer advanced DPAA offloading capabilities like: IPSec offload, zero-copy frames between USDPAA and kernel stack, OH CSUM offload.
  - *DPAA OH port driver update*

    Previous versions of the offline port driver did not initialize the frame queues that entered and exited the OH port and relied on other software components (Ethernet driver, USDPAA, or other kernel modules) to initialize those queues. This update adds full capability for initializing both ingress and egress queues, eliminating the dependency on USDPAA and/or other kernel modules. With it, complex offload architectures that have OH ports can be largely initialized by the offline port driver.
  - *Priority Flow Control (PFC) - 802.1Qbb*

    Experimental feature added on T4240/T2080 for traffic priority classes and MAC support for 802.1Qbb. PFC provides the ability to issue and respond to Pause Frames on a priority-flow basis and prevents a single flow from consuming the entire port's bandwidth. This is distinct from standard flow control which turns on or off the Ethernet port, stopping all flows.
  - *Sleep/DeepSleep support and Wake on LAN (WoL) support*

    Implement the suspend/resume features that allow stopping the Ethernet port before system enters the sleep state and resuming the port to normal state when the system is waken up. Updated the Ethernet driver to work together with the Wakeup-on-LAN options of ethtool utility. More information about this can be found at Documentation/power/devices.txt in the kernel source tree.

Linux Ethernet Driver for DPAA 1.x Family

- SDK1.5
  - *Single driver for termination and forwarding*

    There is only one Ethernet driver codebase now, based on the “optimized for termination” version which implements the support for S/G frames. But the driver has undergone an optimization process such that its “IP forwarding” performance is similar now to that of the removed “optimized for forwarding” driver. In addition, the “termination” performance has been improved.

    All source code related to the “forwarding” driver is removed, along with the following Kconfig options: FSL_DPAA_ETH_OPTIMIZE_FOR_IPFWD, FSL_DPAA_ETH_OPTIMIZE_FOR_TERM and CONFIG_FSL_DPAA_ETH_SG_SUPPORT. The “termination” driver support for S/G frames requires a different socket buffer layout. The ASF and IEEE1588 modules has been adapted to the new buffer layout, so the existing feature set has been preserved.
  - *Buffer recycling algorithm updates*

    The most important part of the private driver performance improvement strategy consist in data buffer fragments recycling and skb structure recycling. These techniques, formerly deployed in “forwarding ” driver, has been adapted for the S/G support and the new skb layout.
  - *CPU hotplug support*

    In order to support CPU Hotplug, the DPAA Ethernet driver has been adapted to a new QMAN API and implemented a new NAPI logic which maps the portals to each available CPU. When a CPU goes offline the portal interrupts are seamlessly migrated and processed by the boot-CPU (CPU #0)..
  - *Control MAC multicast group using socket API on MAC-less netdevice.*

    The MAC-less interface is now able to configure a MAC device by using the Proxy DPAA Ethernet Driver . The proxy interface offers a simple API to MAC-less interface for enablement and disablement of the MAC device and for adding and removing mac unicast and multicast addresses. This MAC-less feature is optional and can be activated by setting "proxy" attribute in the device tree node of the MAC-less interfaces.
  - *ASF is the default SDK configuration*

    For specific use cases, ASF can provide superior performance than the upstream IP stack, bypassing the normal Ethernet driver and stack processing. Normal stack features may become unavailable in ASF configuration. But, if one needs to compile out the ASF support, the variable KERNEL_DELTA_DEFCONFIG must be set to empty.
- SDK1.4
  - Added pause frame control support through ethtool
  - Added netpoll support
  - Moved "QDisc bypass for performance reasons" option to ASF
  - Added Linux standard API for hardware timestamping (IEEE1588)
  - Moved kernel config options for DPAA Ethernet driver from "Device Drivers -> Network device support -> Ethernet (10000 Mbit) -> Freescale Data Path Frame Manager Ethernet" to "Device Drivers->Network device support->Ethernet driver support->Freescale devices -> DPAA Ethernet "
- SDK1.3
  - The bootarg which controls the maximum frame size (previously "fsl_fman_phy_max_frm") has been renamed as "fsl_fm_max_frm" and is now available in the menuconfig via: Device Drivers --> Frame Manager support --> Freescale Frame Manager (datapath) support --> Maximum L2 frame size (CONFIG_FSL_FM_MAX_FRAME_SIZE)
  - The bootarg which controls the extra headroom ("fsl_fm_rx_extra_headroom") has been moved in the Kconfig and is now available via: Device Drivers --> Frame Manager support --> Freescale Frame

Linux Ethernet Driver for DPAA 1.x Family

Manager (datapath) support --> Add extra headroom at beginning of data buffers (CONFIG_FSL_FM_RX_EXTRA_HEADROOM)

- Scatter/Gather support is activated by the driver's "Optimize for termination" compile-time choice. It is no longer explicitly accessible via the Kconfig.

### 20.1.9.1 Known Issues

- The MTU currently defaults to a maximum of 1522. If you want a higher MTU, it is necessary to pass fsl_fm_max_frm=N on the kernel bootargs, where "N" is the desired maximum MTU + 22 .

### 20.1.9.2 Good Questions:

1. What channel are the FQs assigned to?

   A: Each interface uses by default one pool channel across all Software Portals and also the dedicated channels of each CPU. Note that any of these channels may be shared with other DPAA Eth net devices, and even with other DPAA drivers such as SEC. The *default* and *error* FQs are assigned to the pool channel. The TX queues are assigned to the (direct connect) channel linked to the TX port associated with the interface. Any other statically-defined queues will be assigned in a round-robin fashion to the core-affine portals.

2. What work queue are the FQs assigned to?

   A:

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

3. How do I use the core-affined queues?

   The anticipated way of using the core-affined queues is to use one of the default FMC policy files:

   `/etc/fmc/config/private/common/policy_ipv4.xml`

   `/etc/fmc/config/private/common/policy_ipv6.xml`

   Default FMC configuration files are provided for each reference board:

   ```
   /etc/fmc/config/private/<name of reference board>/<RCW directory>/<name of
   configuration file>
   ```

   Here are two examples showing FMC commands using the default configuration and policy files:

   ```
   (1)    fmc -c /etc/fmc/config/private/t2080rdb/RRFFXX_P_66_15/config.xml  -
   p /etc/fmc/config/private/t2080rdb/RRFFXX_P_66_15/policy_ipv4.xml -a
   ```

   Note that `/etc/fmc/config/private/t2080rdb/RRFFXX_P_66_15/policy_ipv4.xml` is a soft link to `/etc/fmc/config/private/common/policy_ipv4.xml`.

   ```
   (2)    fmc -c /etc/fmc/config/private/t4240rdb/SSFFPPH_27_55_1_9/config_20g.xml  -
   p /etc/fmc/config/private/common/policy_ipv4.xml  -a
   ```

   Note that `/etc/fmc/config/private/t4240rdb/RRFFXX_P_66_15/policy_ipv4.xml` is a soft link to `/etc/fmc/config/private/common/policy_ipv4.xml`.

   If you create a configuration file instead of using one of the default configuration files, be sure to use the appropriate policies found in the default policy files:

   `/etc/fmc/config/private/common/policy_ipv4.xml`

Linux Ethernet Driver for DPAA 1.x Family

/etc/fmc/config/private/common/policy_ipv6.xml.

Linux Ethernet Driver for DPAA 1.x Family

# Chapter 21
# Linux DPAA 1.x Ethernet Primer

## 21.1 Introduction

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

The primary concepts of the DPAA-Ethernet driver architecture are presented without going into such intricate details as device-tree configuration or code structure. The current document is neither a Linux Device Drivers tutorial, nor a replacement to the **Linux Ethernet Driver** document in the SDK, but a quick start guide which provides context for users.

The DPAA-Ethernet driver software shipped with the standard QorIQ Linux SDK is described.

## 21.2 Intended Use Cases

This chapter presents a high-level view of the standard use-cases (based on the QorIQ Linux SDK requirements) that the DPAA-Ethernet driver currently supports.

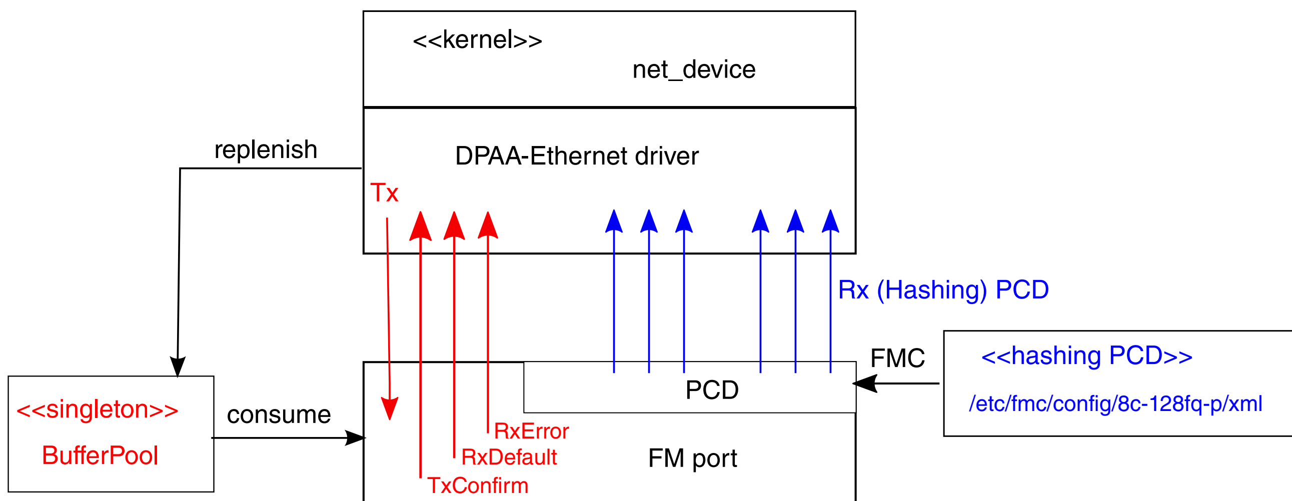
### 21.2.1 Private Net Devices

This is the primary driver and the one which currently delivers the best performance. Characteristics of this driver are:

- The private driver is a multiqueue driver - it uses 1 TX queue per CPU
- All private interfaces use a single BPID - usually dynamically allocated
- The FQIDs for the common types of queues - RX, TX, RX Error, TX Error, TX Confirm - are dynamically allocated
- The Hashing/PCD frame queues are hardcoded in the device tree. The private driver imports the PCD configuration from device tree at startup
- The above resources are allocated and visible only to the private driver

In the case of private interfaces, all network traffic takes place between the Linux kernel and the physical FMan port private to that partition.

Figure 26: Network Traffic Between the Linux Kernel and the Physical FMan Port




## 21.2.2 Shared-MAC Net Devices

A shared-MAC device is one that can be used from two (or potentially more) Linux and/or USDPAA partitions. A shared-MAC encompasses one of the following partitioning scenarios:

Figure 27: Two (or more) Linux separate partitions, under control of the TOPAZ hypervisor.

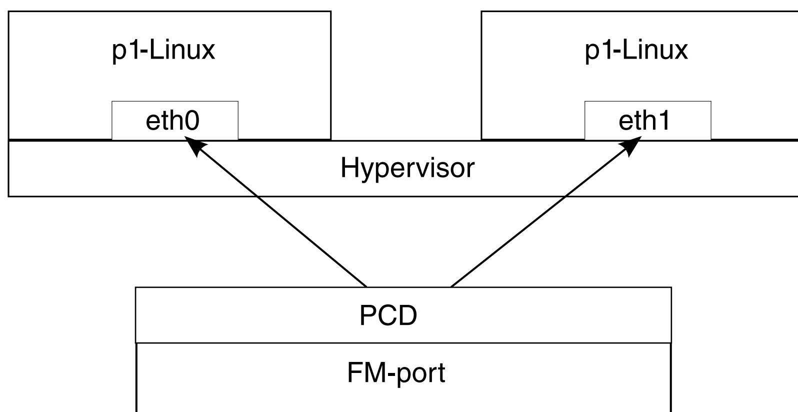
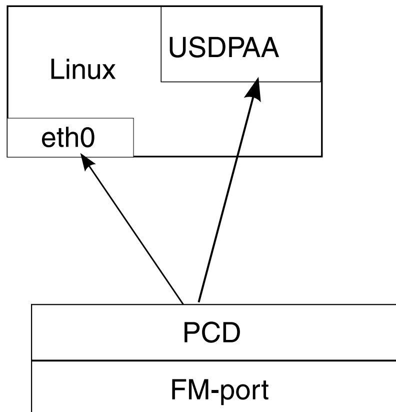


Figure 28: One Linux and one USDPAA running in the same partition




Points to emphasize

- In all cases, there is exactly one physical MAC. Linux always “owns” the MAC, which it initializes.
- A shared-MAC between two Linux interfaces (say, eth0 and eth1) from the same partition is **not supported** unless in a hypervised scenario (and has never been intended as a use-case, anyway).

Configuration and ownership of a shared-MAC is asymmetric and driven by a number of hardware and software constraints.

The following constraints and design assumptions apply to Buffer Pools in a shared-MAC device:

- FMan v2 (not supporting virtual storage profiles) picks the buffer to store the Rx frame based solely on the ingress frame size, regardless of the result of PCD.
- In shared-MAC devices used by Linux and USDPAA, to avoid the need for the USDPAA fastpath to remap the ingress buffers, the same Buffer Pool is shared between Linux and USDPAA.
- In shared-MAC devices used by two Linux partitions (no USDPAA involved), it is necessary that the Linux guests have the same true-physical to guest-physical mappings, in order for FMan-DMA to work seamlessly regardless of the destination partition. Moreover, that presumes that the buffer space seen by the two partitions is identical, which comes down to the Buffer Pools being physically shared between the two Linux partitions.
- Sharing a Buffer Pool between two or several Linux and/or USDPAA partitions requires that the BPID be statically defined (identically hard-coded) in the `.dts` configurations of all partitions.
- The convention between Linux and USDPAA is that USDPAA initializes and seeds the shared Buffer Pool. Linux dynamically remaps ingress buffers received on a shared-MAC, copies them into a buffer dynamically allocated in its own memory space, then releases the ingress buffer back into the shared Buffer Pool.
- In the case of a shared-MAC between two Linux partitions (in a hypervised scenario), only one partition will initialize and seed the shared Buffer Pool. That partition is determined by means of a special property in the `hv.dts` partition configuration. Refer to **Linux Ethernet Driver - Buffer Pools** for details.

The following constraints and design assumptions apply to Frame Queues in a shared-MAC device as seen by the Linux DPAA-Ethernet (i.e. not USDPAA) driver:

- Rx traffic is driven to either partition via a PCD configuration applied on the shared physical port. This means that, for Linux partitions using a shared-MAC, at least one Rx Frame Queue has to be statically declared in the `.dts`. That may be the Rx Default Frame Queue and/or the (automatically initialized) 128 core-affine **Hashing PCD Frame Queues**.
- In the Linux-Linux shared-MAC scenario, all Tx Frame Queues must be statically specified and be the same in both partitions. The reasoning is explained in the **Linux Ethernet Driver - Virtual/Shared Controller** document. .

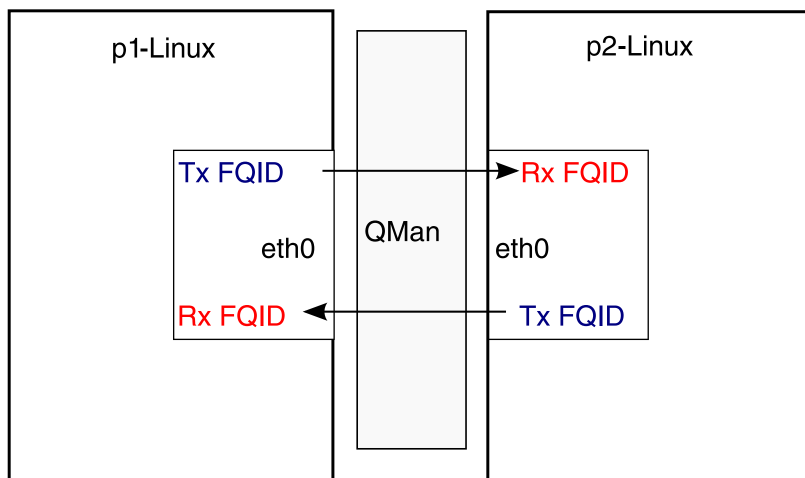
- In the Linux-USDPAA shared-MAC scenarios, the Linux partition will always initialize its own Tx Frame Queues, be they dynamically or statically allocated. It is up to the USDPAA application to choose its own Tx FQIDs.
- The Tx Confirm Frame Queues are always used in shared-MAC scenarios; all egress buffers are confirmed. See **Shared MAC: Tx** packet lifecycle for details.

## 21.2.3 MAC-less Net Devices

A MAC-less device, also called a Virtual Controller in the **Linux Ethernet Driver - Virtual/Shared Controller**, appears to Linux as a regular net device (Ethernet interface). It is “virtual” in the sense that it does not have a physical FMan port (be it an online or an OH port) underlying, but that is transparent to the Linux kernel and userspace applications – only the DPAA-Ethernet driver is aware of the difference.

The figure below shows the typical MAC-less use-case is as a communication device between two Linux partitions:

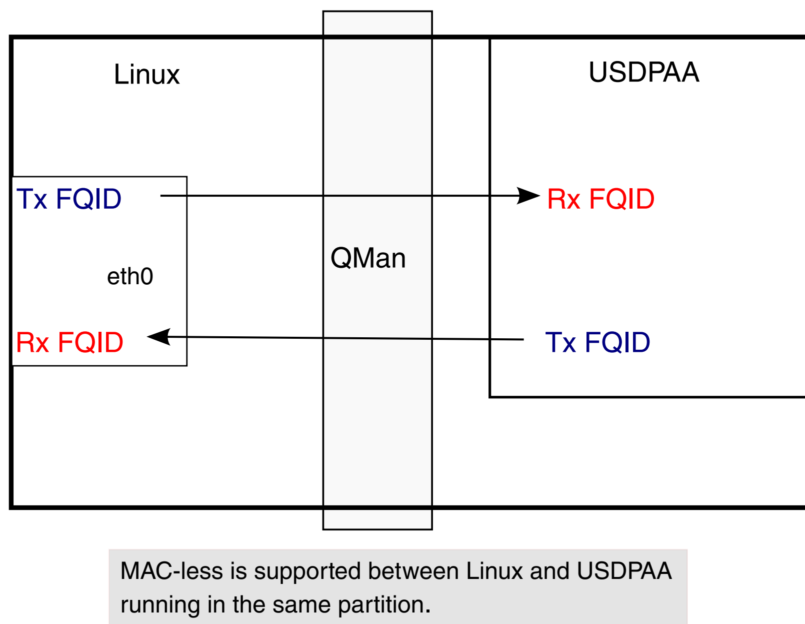
**Figure 29: Communication device between two Linux partitions**




Typical MAC-less setup between two Linux partitions
(Both under control of the hypervisor)

… or between Linux and USDPAA:

Figure 30: Communication device between one Linux and one USDPAA in the same partition




MAC-less is supported between Linux and USDPAA running in the same partition.

From the DPAA resource configuration standpoint, a MAC-less net device is very similar to a **Shared MAC: Tx**. The same constraints apply to Buffer Pools and Frame Queues as in the case of a shared-MAC, with one notable difference:

- Because the Tx FQs of a MAC-less device always sink into another partition (or USDPAA) instead of a physical FMan port, there is the convention that the DPAA-Ethernet driver of a MAC-less node **only initializes its Rx FQs**. In other words, each partition initializes its own Rx FQs, because it has to bind local dequeue callbacks to them.

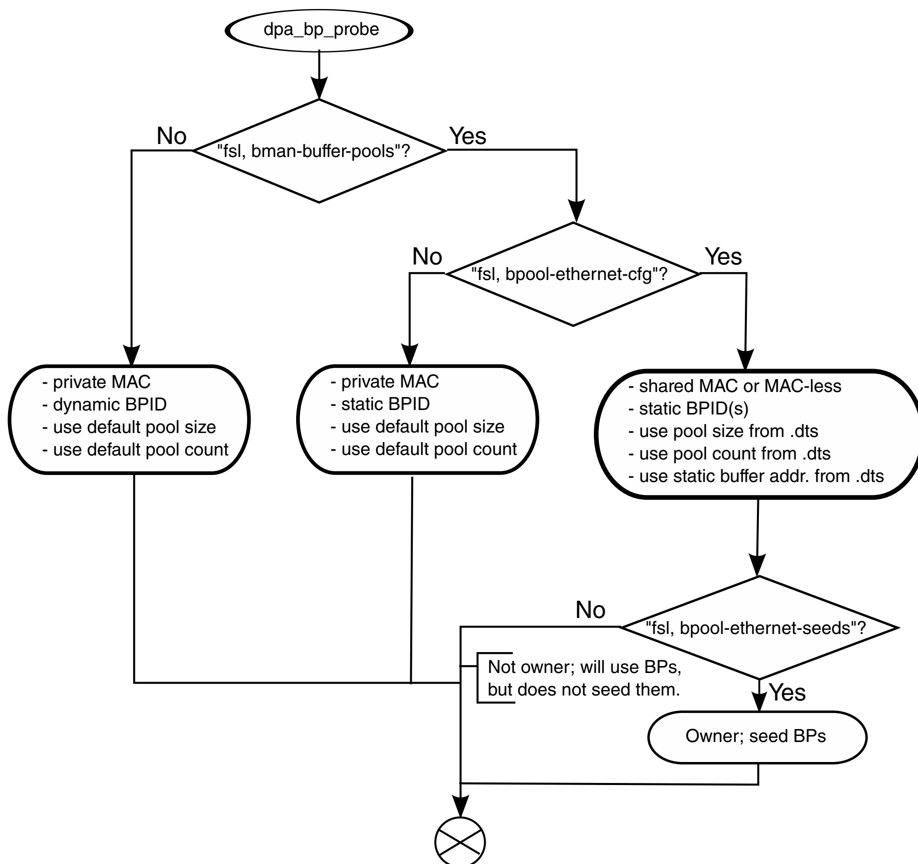
In a MAC-less setup, one endpoint's Tx FQIDs are the other endpoint's Rx FQIDs and vice versa.

While the previous diagrams have shown typical MAC-less use-cases, one can design more complex scenarios by interposing various processing blocks between the two MAC-less endpoints. Refer to the **Extended Use Cases** chapter for a discussion.

## 21.2.4 Choosing the Current Use Case

The following activity diagram describes the configuration selection logic in the DPAA-Ethernet driver, based on properties found in the `.dts` specification:

Figure 31: Configuration Selection Logic




## 21.3 The DPAA-Eth View of the World

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

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

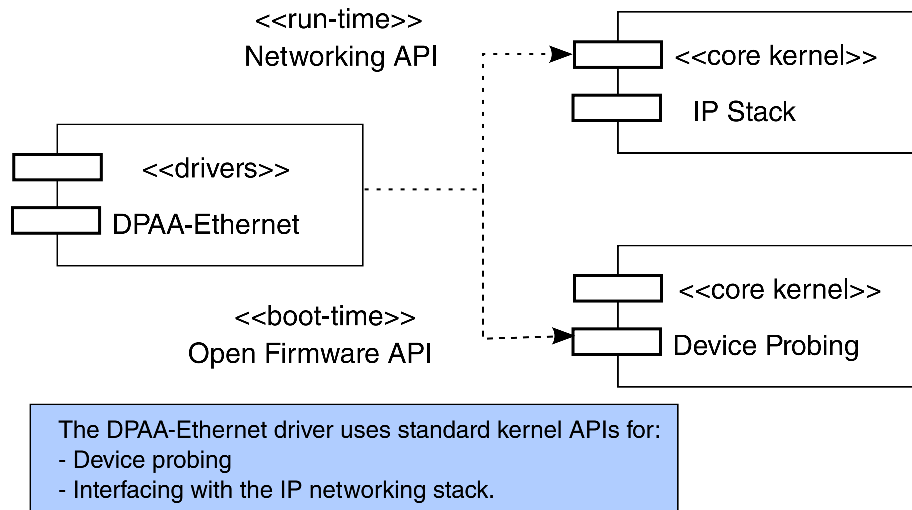
### 21.3.1 The Linux Kernel API's

The DPAA-Ethernet drivers interface with the Linux kernel via the latter's networking stack APIs. This is a strong requirement, mandated by the integration with the Linux kernel.

Another type of interaction with the kernel code is at boot-time, via the Open-Firmware API. That API is used to parse the PowerPC platform device tree and discover the hardware modules that need to be configured. In particular, the DPAA-Ethernet drivers use the platform device tree to discover:

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

Figure 32: Platform Device Tree

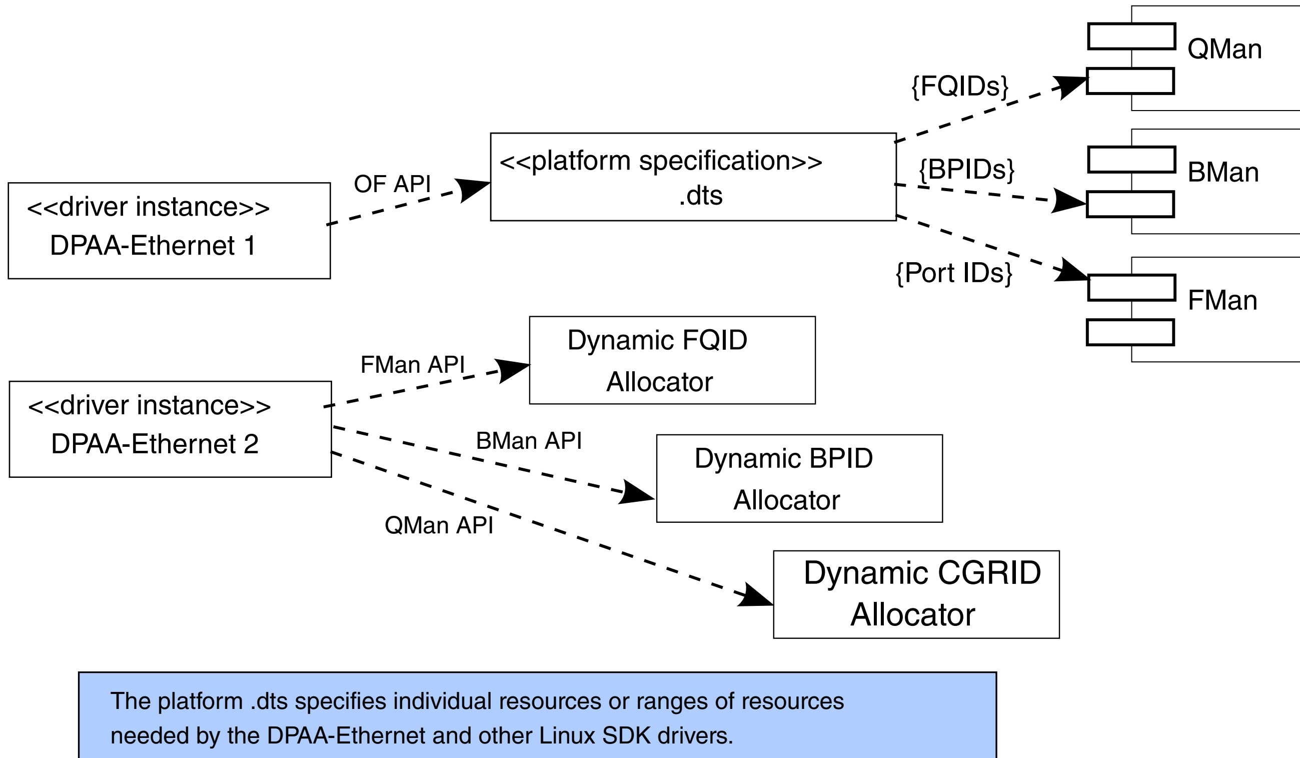



Generally, we prefer drivers configurations to be dynamic and transparent to the rest of the system. Among the benefits of dynamic resource allocations, we count:

- Portability of the drivers across multiple QorIQ platforms;
- Seamless support of platform changes (e.g. via booting with different RCWs);
- Seamless support of multiple partitions under the control of a hypervisor;
- Cohabitation with other DPAA drivers (e.g. a SEC driver) in the SDK.

In certain scenarios, however, configurations are statically defined and are extracted directly from the .dts specification. This is for instance the case of shared MAC and MAC-less devices.

Figure 33: Shared MAC and MAC-less Devices




## 21.3.2 The Driver's Building Blocks

This chapter presents the main structures and data entities with which the DPAA-Ethernet driver operates.

The driver's building blocks are part of the interfaces of the driver with the relating components, i.e.:

- The kernel’s IP stack;
- The DPAA hardware blocks and their drivers.

The DPAA Ethernet driver is actually a series of drivers, each tailored for a specific use case:

- Private driver - maps Linux kernel network interfaces to physical ports
- Shared driver - gives simultaneous control of a physical port to the Linux kernel and user space applications
- Macless driver - although it appears to the Linux kernel as a regular net device, it does not control a physical port. It is typically used as a virtual communication device between two Linux partitions
- Proxy driver - an interface through which the Linux kernel configures a physical port, only to pass its control to user space applications
- Offline port driver - controls Offline Parsing / Host Command port (OH)

Each of the above drivers has its own code base and implements the Linux kernel API by providing its own callback functions.

### 21.3.2.1 Net Devices

A net device (`struct net_device` in C representation) is the fundamental structure of any Linux network device driver.

A net device describes a (physical or virtual) device capable of sending and receiving packets over a (virtual or physical) network. All incoming and outgoing traffic is accounted and processed on behalf of the net device it comes or goes on.

Each supported type of net device has its own kernel driver. If there are several such devices present in a system, there will be as many device driver instances.

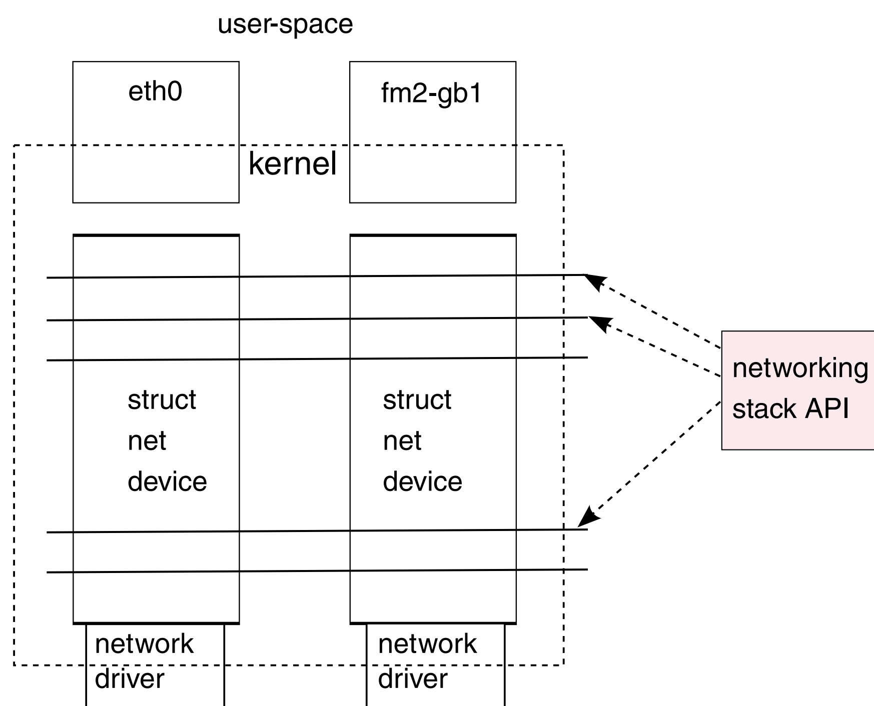
A net device is accessible to the Linux user via the standard tools, such as ‘ifconfig’ or ‘ethtool’.

Not all net devices have real underlying hardware; tunnel endpoints, for examples, are represented by net devices but are not directly backed by hardware. Same holds for drivers such as “bonding” or “dummy”.

It is worth emphasizing, however, that **every** Linux interface is represented by a net device. This is a fundamental design aspect of all Linux networking drivers, including DPAA-Ethernet. One can describe the Linux IP stack as being a **netdev-centric** construction. Nearly all of the kernel networking APIs receive a `struct net_device` as a parameter. The `net_device` structure is the handle through which the driver and the network stack communicate.

The following diagram illustrates what has just been described:

Figure 34: Every Linux Interface is Represented by a Net Device




The kernel networking APIs are generally netdevice-centric.
A network driver interfaces with the IP stack on behalf of a net device

## 21.3.2.2 Frame Queues

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

Ingress frames received by the DPAA-Ethernet driver on one of the Frame Queues it is servicing are sent to the IP stack on behalf of the net device structure that the driver is associated with. Conversely, outgoing frames coming from the IP stack into the driver are enqueued to one of the egress Frame Queues.

NOTE

Depending on its configuration (see usecase), the DPAA-Ethernet driver may initialize some of its Frame Queues and make assumptions over what entity initializes the others.

#### 21.3.2.3 Buffer Pools

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

Unlike the Frame Queue utilization – which is more flexible – the Buffer Pool utilization is conditioned by several design assumptions:

- The source and ownership of the ingress frame buffers are presumed by the DPAA-Ethernet driver. The exact allocation logic depends on the driver configuration (see usecase), but in all cases the driver makes hard assumptions on how the buffers were allocated.

  For instance, the “private MAC” driver configuration seeds the Buffer Pools at predefined checkpoints on the Rx path. There are also buffer utilization counters maintained by the driver, which influence the buffer allocation logic.

  The “shared MAC” and “MAC-less” driver configurations only work with the Buffer Pools hard-coded in the platform `.dts`.
- The layout of incoming frames is also presumed by the driver. Depending on its configuration (see usecase), the driver expects the frame layout to conform to its own allocation logic. The actual buffer layout is outside the scope of this document and should not be assumed upon by driver users.
- The existence of a static Buffer Pool configuration in the platform `.dts` determines the driver configuration. The DPAA-Ethernet driver currently assumes that, if a static Buffer Pool is configured in its device tree node (as number/size/address of buffers), then it is describing a “shared MAC” or a “MAC-less” configuration.

## 21.4 DPAA Resources Initialization

The rationale behind the “what”s, “why”s and “how”s of DPAA resource initializations made by the DPAA-Ethernet driver are presented. This description does not go into the full detail of driver configuration (please refer to the **Linux Ethernet Driver** for that level of detail).

### 21.4.1 What, Why and How Resources are Initialized

DPAA resources initialized by the various configurations of the DPAA-Ethernet driver are:

- FQs and FQIDs (where static config applies);
- BPs and BPIDs (where static config applies);
- Buffers (not quite “DPAA” resources, rather “system” resources);;
- CGRs (CGRIDs are always dynamic);
- FMan’s online ports (note: the offline ports are configured by a different driver than DPAA-Ethernet).

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

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

### 21.4.2 Hashing/PCD Frame Queues

Among the Frame Queues initialized by the DPAA-Ethernet driver, there is a predefined set of 128 core-affined Rx FQs, automatically initialized by the driver for user’s convenience. They are there because most performance-enhanced setups must use a PCD configuration; to that end, the standard QorIQ Linux SDK provides a “hashing PCDs” configuration that can be applied by the user via the FMC tool. Since FMC does not support dynamic FQID specification in its .xml configuration files, the “hashing PCD” Frame Queues also have static, hard-coded FQIDs.

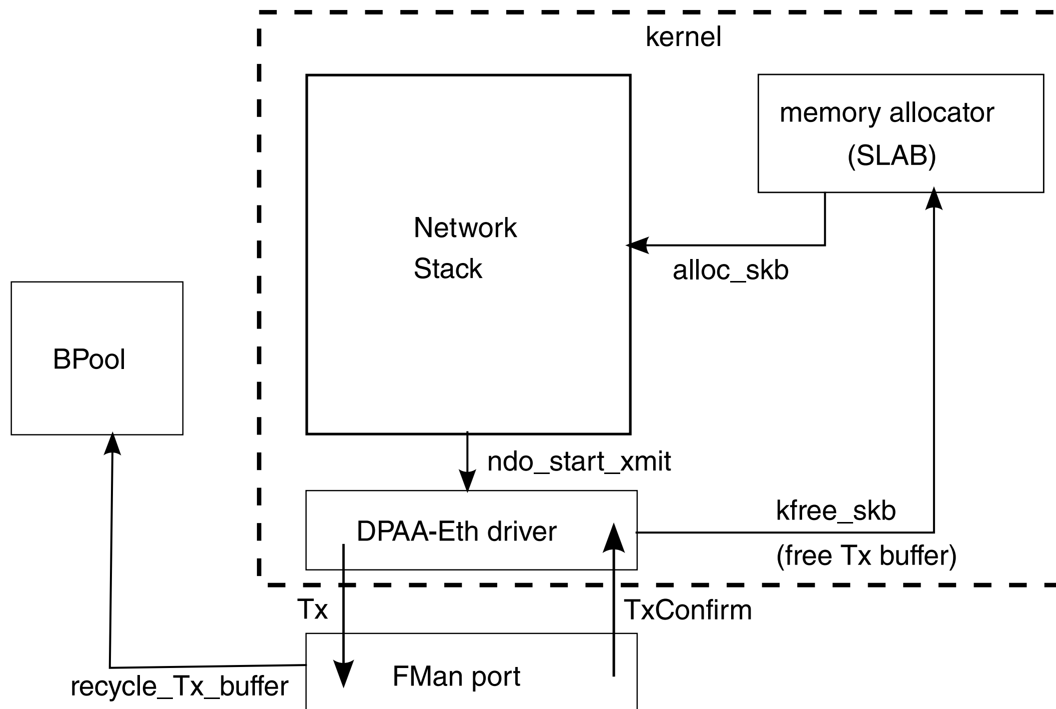
For details about the “hashing PCD” Frame Queues, refer to the **Linux Ethernet Driver - Core Affined Queues**.

## 21.5 The (Simplified) Life of a Packet

This chapter presents a packet’s lifecycle in various configurations of the DPAA-Ethernet driver.

### 21.5.1 Private Net Device: Tx

Figure 35: DPAA-Ethernet driver enqueues the packet to the FMan port



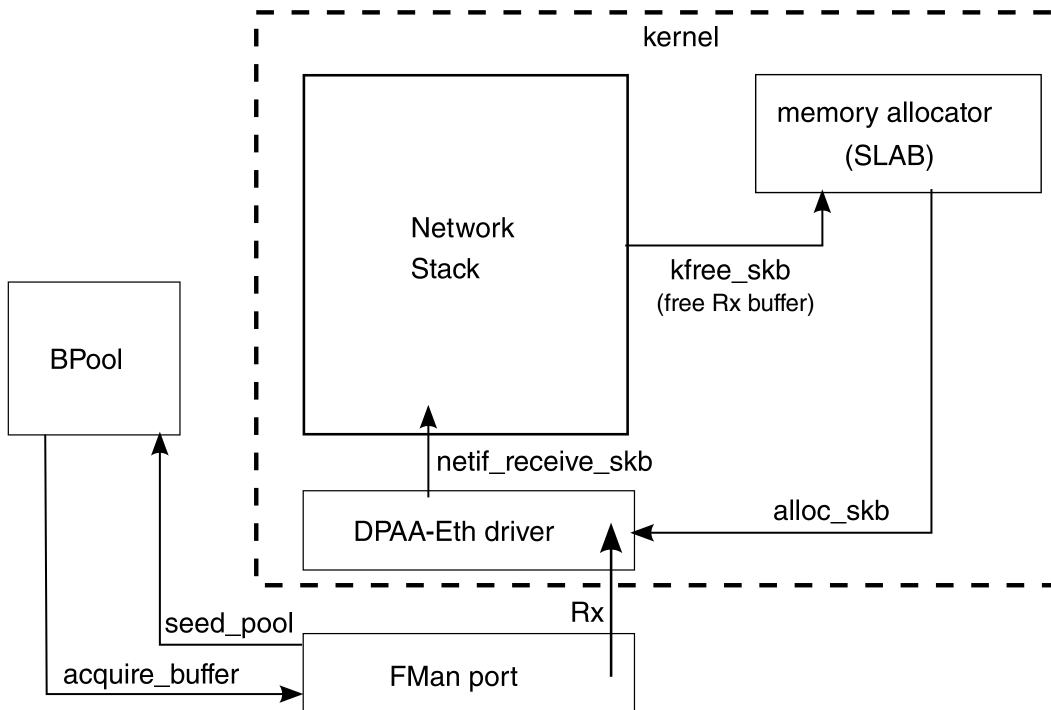

Arrows in the diagram above represent the direction of the buffer/packet flow.

A packet on the egress path is allocated by the network stack using the kernel’s standard memory allocator. The DPAA-Ethernet driver enqueues the packet to the FMan port with an indication to recycle the buffer if possible. If recycling is not possible, the DPAA-Ethernet driver itself frees the buffer memory back to the kernel’s allocator, when Tx delivery is confirmed by FMan.

 Non-disclosure Agreement Required. 

## 21.5.2 Private Net Device: Rx

Figure 36: Buffers on the Ingress Path

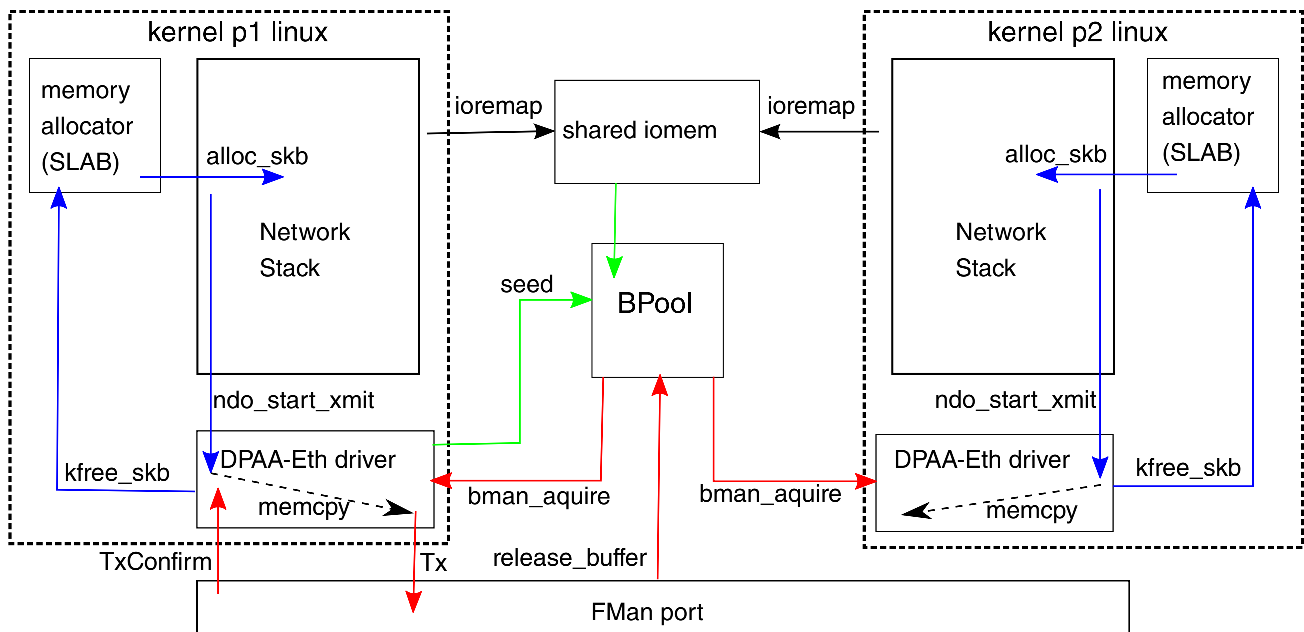



Buffers on the ingress path are acquired by FMan directly from a Buffer Pool which was seeded by the DPAA-Ethernet driver. Buffer layout is important to the driver, which assumes ownership on the BP. Arrows in the diagram above represent the direction of the buffer/packet flow.

## 21.5.3 Shared MAC: Tx

The following diagram presents the lifecycle of an egress buffer/packet in the case of a shared-MAC device used by two Linux partitions (under the control of the Hypervisor, not shown in the picture):

Figure 37: A Shared-MAC Device Used by Two Linux Partitions




There are essentially two buffer circuits in this scenario:

- The in-stack socket buffer lifecycle (colored in blue);
- The bpool buffer lifecycle (colored in red).

Socket buffers are dynamically allocated from the kernel memory. The DPAA-Ethernet driver memcopies them in buffers acquired from the Buffer Pool, then releases the socket buffers back into the kernel memory.

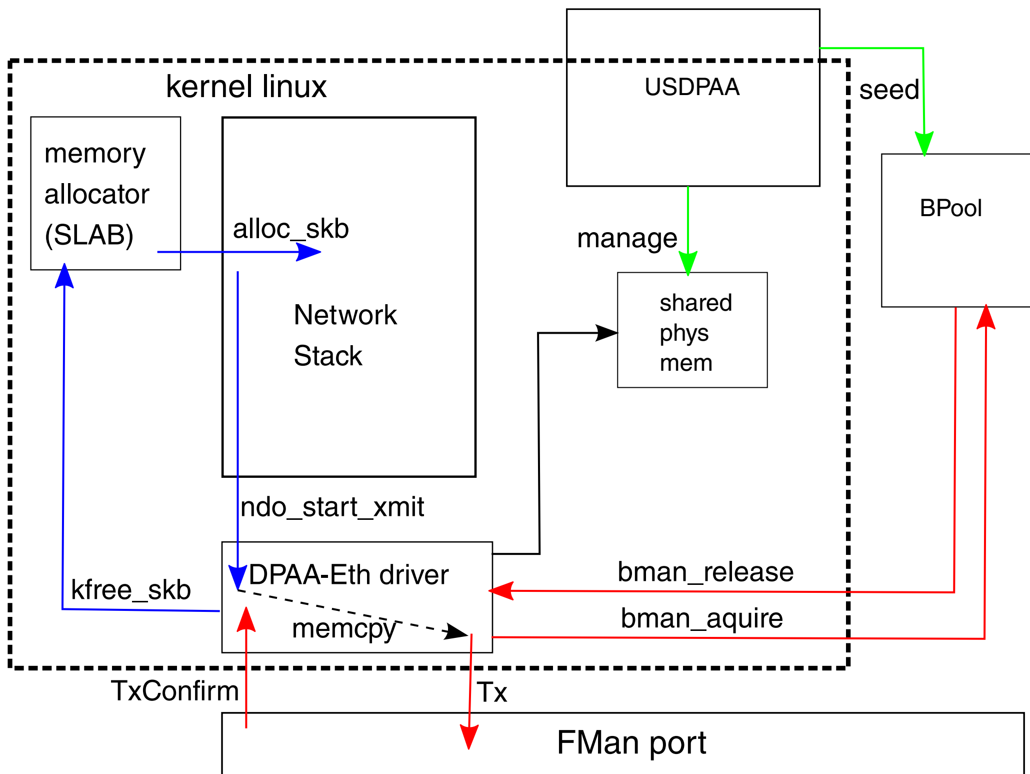
The shared Buffer Pool is seeded by one (and only one) of the Linux partitions, and is never again replenished by the software. Memory used for seeding the shared Buffer Pool is statically defined in the guest `.dts` configuration and is presented to the kernel as device memory (not as physical memory). Both DPAA-Ethernet drivers statically ioremap the entire shared Buffer Pool space (as per the static configuration in the .dts) at probe time. No run-time mapping/unmapping is effected afterwards.

After transmission, FMan confirms the frame and the Linux DPAA-Ethernet driver releases the buffers back into the Shared Buffer Pool.

Similarly to the above, the next diagram the lifecycle of an egress buffer/packet in the case of a shared-MAC device used by Linux and USDPAA:


Non-disclosure Agreement Required.

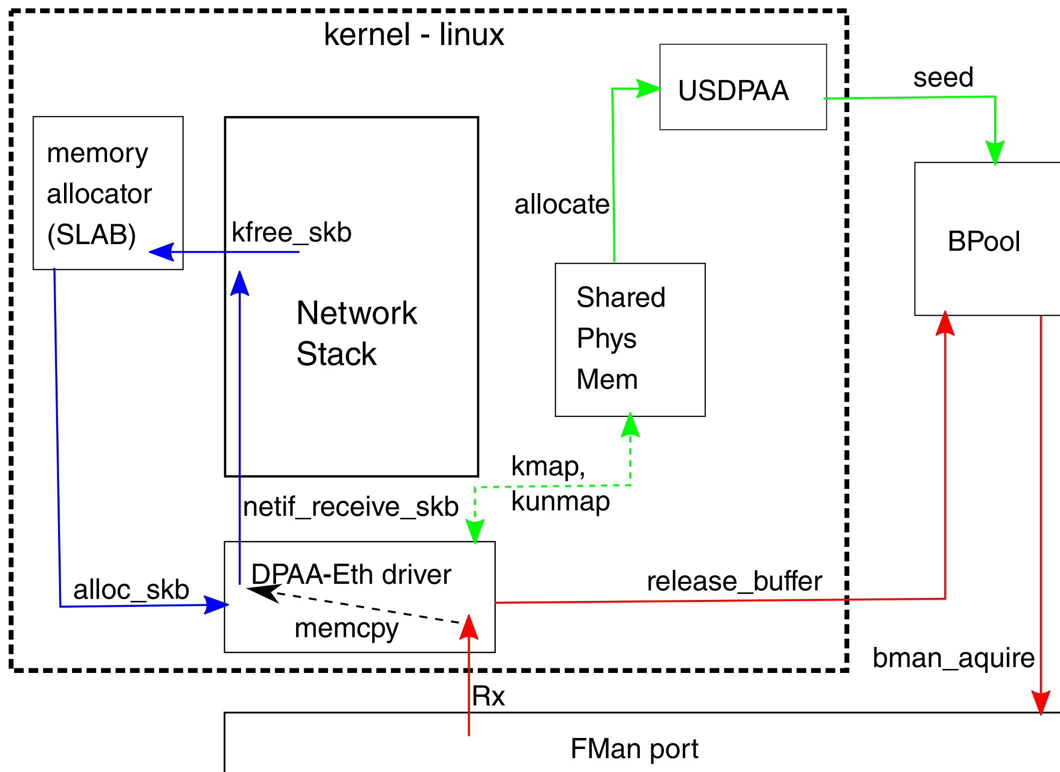
Figure 38: Shared-MAC Device Used by Linux and USDPAA




## 21.5.3.1 Shared MAC: Rx

The diagram below presents the lifecycle of a buffer/packet in the case of a shared-MAC device used by Linux and USDPAA running in the same partition.

Figure 39: Shared-MAC Device Used by Linux and USDPAA




The Linux driver is the one to initialize the shared FMan port, but it is the USDPAA partition that allocates and seeds the buffers in the Shared Buffer Pool. To that end, USDPAA uses a block of physical memory reserved at boot time, such that the kernel can still map it, but not allocate from it (via either the page allocator or the object cache allocator).

Each incoming frame is dynamically mapped by the DPAA-Ethernet driver. As in the case of shared-MAC Tx, a memcopy is involved in the driver, from each incoming frame into a newly allocated socket buffer (sk_buff). The DPAA-Ethernet driver subsequently unmaps the buffer and releases it back into the shared Buffer Pool, for later reuse.

One should note the invariant that, as in all MAC-less and shared-MAC scenarios, the Shared Buffer Pool is initialized and seeded exactly once, at init-time; at run-time, the total number of in-flight buffers and buffers available in the pool is constant. One must preallocate a large enough number of buffers in the Shared Buffer Pool, such that to prevent its run-time depletion.

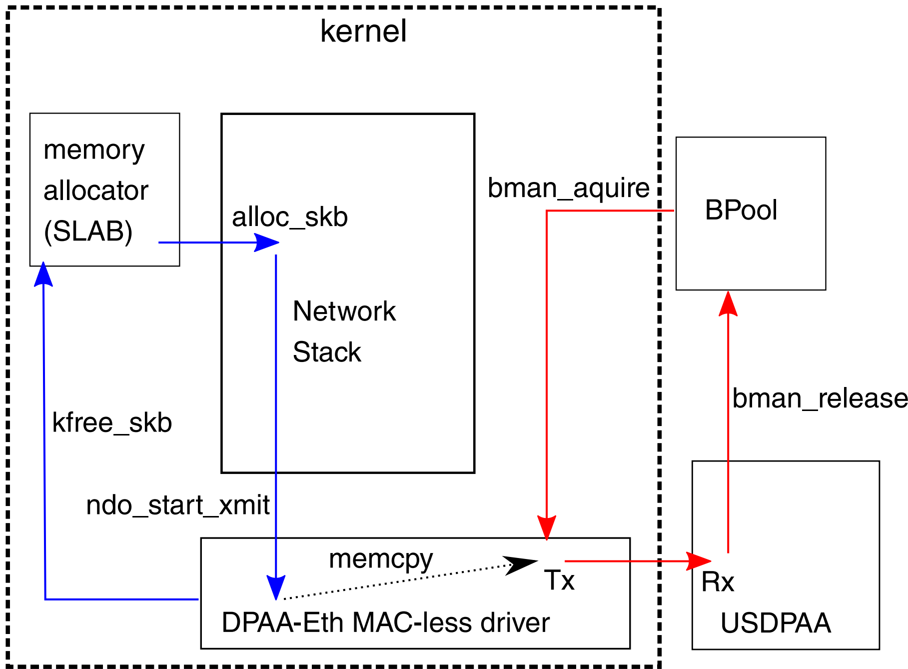
NOTE

A similar diagram can be obtained in the case of a shared-MAC between two different Linux partitions, with the principal change that the shared memory from which the Buffer Pool is seeded is seen as iomem (rather than physical memory), and is statically ioremapped all at once, at boot-time.

## 21.5.3.2 MAC-less Net Devices: Tx

The following diagram presents the lifecycle of a buffer/packet in the case of a MAC-less device used for communication from Linux to USDPAA:

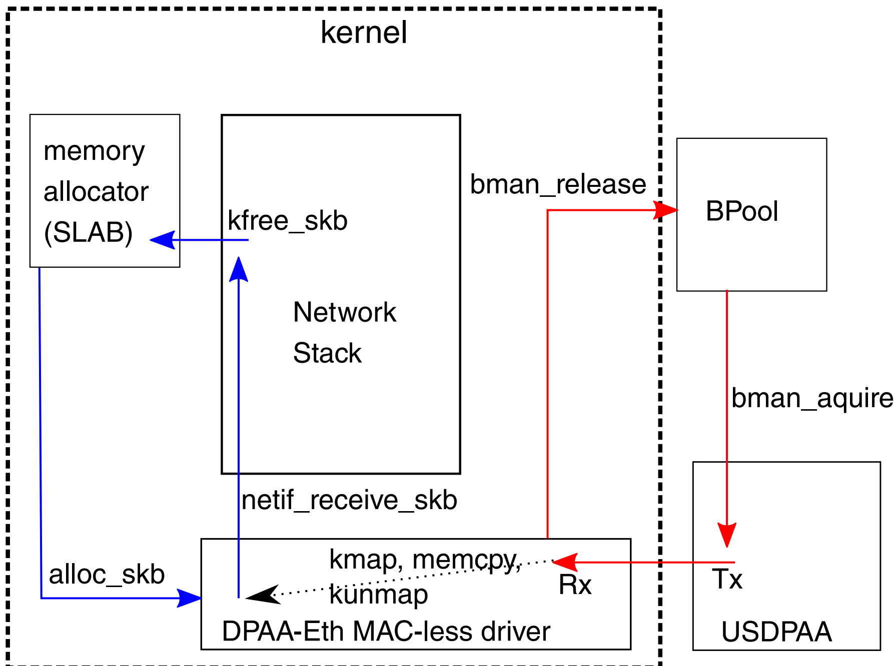
Figure 40: MAC-less Device Used for Communication from Linux to USDPAA




## 21.5.3.3 MAC-less Net Devices: Rx

The following diagram presents the lifecycle of a buffer/packet in the case of a MAC-less device used for communication from USDPAA to Linux:

Figure 41: a MAC-less Device Used for Communication from USDPAA to Linux



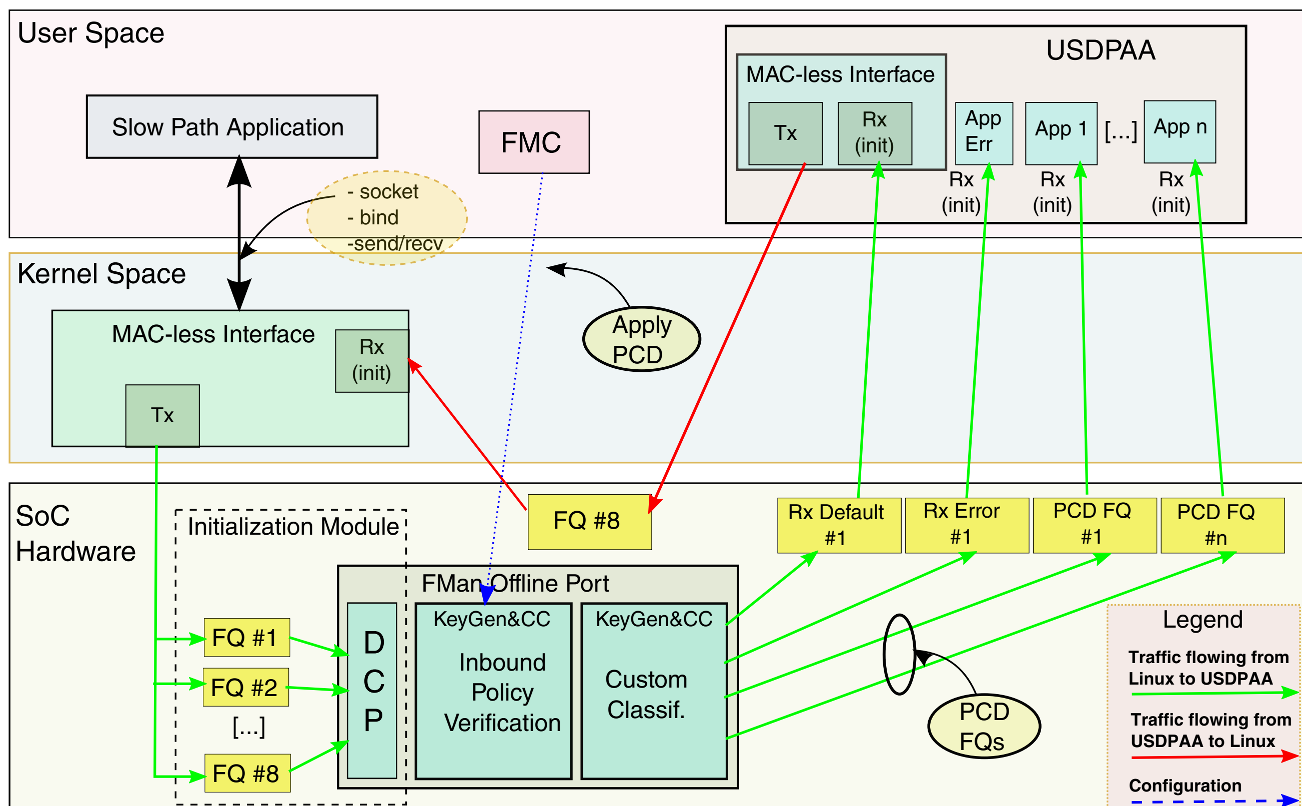
# 21.6 Advanced Drivers Use Cases

This chapter is meant as an initial guideline toward building more complex use-cases, based on the predefined built-in capabilities of the DPAA-Ethernet driver that have been explained so far. (Note: Such use-cases are not currently part of the standard Linux QorIQ SDK.)

## 21.6.1 MAC-less Over OH (Linux-USDPAA

One can interpose an OH port between a Linux MAC-less net device and a USDPAA MAC-less interface, where the Linux-to-USDPAA path goes through the OH port, while the USDPAA-to-Linux path is direct:

Figure 42: Linux MAC-less net device and a USDPAA MAC-less interface




In the current implementation, the DPAA Ethernet Driver initializes Linux MAC-less Rx FQs (= USDPAA's MAC-less Tx FQs).

Because there is no symmetric MAC-less endpoint to initialize the Linux endpoint's Tx FQs (as in the case of a standard MAC-less Linux-Linux setup), a separate kernel module is necessary in order to:

- Initialize the Linux MAC-less Tx FQs; and
- Add them to the DCP of the Offline Port.

The diagram shows:

Table 20: DPAA Ethernet Driver initializes FQs

| # | Description | Detail |
| --- | --- | --- |
| 1 | A MAC-less Linux interface with: | • 8 Tx FQs – initialized by the Offline Port Initialization Module and placed in the OH port's DCP;<br>• 8 Rx FQs – initialized by the MAC-less interface and connected to the Tx FQs of the USDPAA MAC-less interface. |
| 2 | A MAC-less USDPAA interface with: | • 8 Tx FQs – initialized by the Linux MAC-less interface. (Note: USDPAA currently asserts that there are exactly 8 Tx FQs. This can be changed if a different number of frame queues are requested).<br>• 8 Rx FQs - initialized by the MAC-less interface. Unlike in the standard Linux-Linux MAC-less scenario, these are different from the Tx FQs of the Linux MAC-less interface, because of the interposing OH port. The Offline Port's Rx Default and Rx Error FQs (and possibly the PCD FQs) use some of these FQIDs. |
| 3 | An Offline Port Initialization Module. | Responsible with initializing the 8 Tx FQs of the Linux MAC-less interface and connecting them to the DCP of the OH port. |

The common idea of this design is that the entity which uses the queues for Rx is responsible with their initialization and (for software portals) specifying their dequeue callbacks. On the kernel side, this is the responsibility of the MAC-less interface (DPAA-Ethernet driver) and on USDPAA the responsibility is of the application that uses the FQs.

## 21.6.2 MAC-less Over OH (Linux-Linux)

Similarly to the previous use-case, one can use an Offline Port's interface with the QMan for interposing an OH between two MAC-less endpoints and effectively build a MAC-less-over-OH net device.

## 21.6.3 ARP Handling in Shared MAC

1. Linux-Linux: Currently (FMan-v2), ARP packets arriving on a shared-MAC interface are (in absence of relevant PCD configuration) routed to the Linux partition.
2. Linux-USDPAA: This is entirely handled by USDPAA infrastructure. Please refer to the USDPAA documentation.

## 21.6.4 Multicast Support in Shared MAC

This is not a supported feature in the FMan-v2- and FMan-v3-based DPAA-Ethernet drivers.

## 21.7 Appendix A: Infrequently Asked Questions

Table 21: Q and A

| # | Question | Answer |
|---|---|---|
| 1 | How do I send a frame up the network stack? | The frame-processing network stack only exists in the context of a net device. So, “sending a frame into the stack” is an inaccurate statement: the frame must first be associated to a net device, and then the respective instance of the Ethernet driver will deliver the frame to the stack, on behalf of that net device. To achieve that, the frame must arrive via the physical device that underlies the driver (or via a MAC-less device’s ingress FQs). |
| 2 | Can I allocate a buffer and inject it as a frame into a private interface’s ingress queues? | This is probably a mistake. The DPAA-Ethernet driver makes hard assumptions on buffer ownership, allocation and layout. In addition, the driver expects FMan Parse Results to be placed in the frame preamble, at an offset which is implementation-dependent. In short, while a carefully crafted code might work, it would make for *very* brittle design, and hard to maintain, too.<br><br>NOTE<br>A MAC-less interface (potentially modified to interpose an OH port), however, will support this case, as long as: a) the buffer originates from the interface’s shared Buffer Pool; and b) the frame is a valid Ethernet frame. |
| 3 | But can I acquire a buffer directly from a private interface’s Buffer Pool, and inject it as such into the private interface’s Rx FQs? | While this works on a MAC-less or shared-MAC interface, it is not an intended use-case for private interfaces. |

*Table continues on the next page...*

Table 21: Q and A (continued)

| # | Question | Answer |
| --- | --- | --- |
| 4 | What format must an ingress frame have, from the standpoint of the DPAA-Ethernet driver and the Linux kernel stack? | The DPAA-Ethernet driver is expected to perform an initial validation of the ingress frame, but does not look at the Layer-2 fields directly. The current kernel networking code does make a check on the MAC addresses of the frame and the protocol (Ethertype) field. One should not make assumptions on such details of frame processing, because the kernel stack implementation is not bound by any contract. |

## 21.8 Appendix B: Frequently Asked Questions

The right place to look at the driver FAQs is the **Linux Ethernet Driver** document in the SDK.

Low Power UART User Guide

# Chapter 22
# Low Power UART User Guide

## 22.1 Low Power UART User Guide

### Description

The LS1043A SoC integrate Freescale Low Power UART module. LS1043A QDS supports the lpuart port as the alternate console.

### Dependencies

| Hardware: | Freescale LS1043A QDS board |
|---|---|
| Software: | Linux 3.19 & U-boot v2015.01 |

### RCW configuration

To support the lpuart module, specific rcw binary is need for LS1043A QDS board. So the rcw source should be built first. Refer to below table for the rcw binary files name and location after built.

| board | RCW binary |
|---|---|
| LS1043A QDS | ls1043aqds/RR_FQPP_1455/rcw_1600_lpuart.bin |

### 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 Freescale lpuart support |
| CONFIG_LPUART_32B_REG | Select 32-bit lpuart register mode |

Choosing predefined u-boot board configs:

| board | build configuration to select lpuart as u-boot console |
|---|---|
| LS1021A QDS | make ls1021aqds_nor_lpuart_defconfig |
| LS1021A TWR | make ls1021atwr_nor_lpuart_defconfig |

### Runtime options

Low Power UART User Guide

| 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 ---><br>Character devices ---><br>Serial drivers ---><br><*> Freescale lpuart serial port support<br>[*] 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.

Low Power UART User Guide

| Source File | Description |
| --- | --- |
| drivers/tty/serial/fsl_lpuart.c | The LPUART driver file |

### Jump or switch setting

| Board | Jump | Switch | lpuart port |
| --- | --- | --- | --- |
| LS1021A TWR | J19 1-2; J20: 1-2 | sw3[5] on | J5 |
| LS1021A QDS | N/A | sw13[6:8] = 001 | JP2 |

### 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. for LS1021 QDS board, run below u-boot command to update rcw:
   - dhcp 82000000 <tftpboot dir>/rcw_1000_lpuart.bin
   - protect off all;erase 0x64000000 0x6401ffff;cp.b 0x82000000 0x64000000 100;
3. for LS1021A TWR board, run below u-boot command to update rcw:
   - dhcp 82000000 <tftpboot dir>/rcw_1000_lpuart.bin
   - protect off all;erase 0x64000000 0x6401ffff;cp.b 0x82000000 0x64000000 100;
4. for u-boot update, first make sure the u-boot is built with the proper board configuration, then run below u-boot command:
   - dhcp 81000000 <tftpboot dir>/u-boot.bin
   - protect off all;erase 0x64100000 0x641fffff;cp.b 0x81000000 0x64100000 80000;
5. after all is updated, run u-boot command boot_bank4 in the current prompt 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
Board: LS1021AQDS
Sys ID:0x2b, Sys Ver: 0x12, vBank: 4
FPGA:  v13 (QIXIS_LS1021QDS_2014_08_27_1658), build 65386
I2C:   ready
Initializing DDR....using SPD
Detected UDIMM 18KSF51272AZ-1G6K1
DRAM:  2 GiB (DDR3, 32-bit, CL=11, ECC on)
Using SERDES1 Protocol: 96 (0x60)
Flash: 128 MiB
NAND:  512 MiB
MMC:   FSL_SDHC: 0
Not a microcode
```

Low Power UART User Guide

```
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
scanning bus for devices...
Found 0 device(s).
Net:   eTSEC1 is in sgmii mode.
eTSEC2 is in sgmii mode.
Phy not found
PHY reset timed out
eTSEC1, eTSEC2, eTSEC3 [PRIME]
=>
```

### 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>/ls1021aqds.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 QDS 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)

[...]

ls1021aqds login: root
root@ls1021aqds:~#
```

2. After the kernel boot up to the console, You can type any shell command in the LPUART TERMINAL.

# Chapter 23
# PCI/PCIe System User Manual

## 23.1 PCI/PCIe System User Manual

### Description

PCI and PCI Express Controller is integrated with the CPU.

### Specifications

| | |
|---|---|
| Hardware: | FSL Freescale PCI/PCIe controller and PCIe Ethernet card |
| PCI Express Supported Modes: | 32/64 bit, x4, x2 and x1 link support on SerDes |
| Software: | Linux 3.12.0+ |

### Module Loading

The PCI/PCIe host bridge support code is compiled into the kernel. It is not available as a module.

### Kernel Configure Tree View Options

- Configuration for PCI

| Kernel Configure Tree View Options | Description |
|---|---|
| `Bus support --->`<br>`        [*] PCI support` | Enable PCI host bridge |
| `Network device support --->`<br>`        Ethernet driver support  --->`<br>`                [*]   Intel devices`<br>`                <*>     Intel(R) PRO/100+`<br>`support` | PCI Intel(R) PRO/100 support |
| `Network device support --->`<br>`        Ethernet driver support  --->`<br>`                [*]   Intel devices`<br>`                <*>     Intel(R) PRO/1000 Gigabit Ethernet`<br>`support` | PCI Intel Pro/1000 support and |

- Configuration for PCIe with MSI interrupts. (If using intx interrupts just deselect the Mesage Signaled Interrupts option)

| Kernel Configure Tree View Options | Description |
| --- | --- |
| `Bus support ---> [*] PCI support [*] Message Signaled Interrupts (MSI and MSI-X)` | Enable PCIe support<br>Enable MSI suppport |
| `Bus support ---> [*] PCI support [*] Message Signaled Interrupts (MSI and MSI-X) PCI host controller drivers ---> [*] Freescale Layerscape PCIe controller` | Enable Layerscape PCI host bridge (only for Layerscape platform)<br>Enable MSI suppport |
| `Network device support ---> Ethernet driver support ---> [*] Intel devices <*> Intel(R) PRO/1000 PCI-Express Gigabit Ethernet support` | PCIe Intel Pro/1000 support and |

### Compile-time Configuration Options

· configuration for PCI

| Option | Values | Default Value | Description |
| --- | --- | --- | --- |
| CONFIG_PCI | y/n | y | Enable PCI host bridge |
| CONFIG_PCI_DOMAINS | y/n | y | Enable PCI domains |
| CONFIG_NET_PCI | y/n | y | Enable network PCI |
| CONFIG_E100 | y/m/n | y | Enable Intel Pro/100+ driver |
| CONFIG_E1000 | y/m/n | y | Enable Intel Pro/1000 driver |

· configuration for PCIe

| Option | Values | Default Value | Description |
| --- | --- | --- | --- |
| CONFIG_PCIEPORTBUS | y/n | y | Enable PCIe general support |
| CONFIG_PCIE_LAYERSCAPE | y/n | y | Enable Layerscape PCIe general support(only for Layerscape platform) |
| CONFIG_PCI_MSI | y/n | y | Enable MSI support |
| CONFIG_E1000E | y/m/n | m | Intel(R) PRO/1000 PCI Express Gigabit Ethernet support |

### Source Files

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

PCI/PCIe System User Manual

| Source File | Description |
| --- | --- |
| arch/powerpc/sysdev/fsl_pci.c | The e500 platform PCI/PCIe host bridge support source |
| drivers/pci/host/pcie-layerscape.c | The Layerscape platform PCIe host bridge support source |
| drivers/net/ethernet/intel/e100.c | Intel Pro/100+ driver source code |
| drivers/net/ethernet/intel/e1000/ | Intel Pro/1000 driver source code |
| drivers/net/ethernet/intel/e1000e/ | Intel Pro/1000 PCI Express Gigabit Ethernet Card driver source code |

### Verification in U-boot

```
u-boot log

PCIE1 connected to Slot 2 as Root Complex (base addr ffe0a000)

  01:00.0     - 8086:10b9 - Network controller  /*if pluged Intel Pro/e1000 network
card,  the pci device can be found */

PCIE1: Bus 00 - 01

...

=> pci 1

Scanning PCI devices on bus 1

BusDevFun  VendorId   DeviceId   Device Class       Sub-Class

_____________________________________________________________

01.00.00   0x8086     0x10b9     Network controller      0x00
```

### Verification in Linux

- Plug Intel Pro/100+ network card or Intel Pro/1000 network card into standard PCI slot on the board. After linux bootup, ifconfig ethx ip address and netmask, then do ping testing.
- Plug Intel Pro/1000 PCIe network card into standard PCIe slot on the 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 Pro/1000 network card is eth0.

### Known Bugs, Limitations, or Technical Issues

- PCIe INTx mode is not compatible with P1022DS due to hardware limitation.
- When run IPFWD from E1000-PCIe to eTSEC on P2020RDB, call trace about msi interrupts will occur sometimes. As a workaround, set 'pci=nomsi' in othbootargs can resolve it.


Non-disclosure Agreement Required.

PCI/PCIe System User Manual

# Chapter 24
# PCI Express Interface Controller

## 24.1 PCI-e Remove and Rescan User Manual

### Description

Describes how to remove and rescan a PCI-e device under runtime Linux system.

### Dependencies

| | |
|---|---|
| Hardware: | All Freescale powerpc platform with PCI-e controller, Intel PCI-e e1000e network card |
| Software: | Freescale linux SDK 1.4 |

### 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 ---><br>[*] Network device support---><br>[*] Ethernet (1000 Mbit) ---><br>[*] 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

The support of PCI-e device remove rescan on powerpc platform is first added in Freescale LInux SDK 1.4(kernel version: 3.8.4). If it fail, the PCI-e device will be rescaned, but the driver of the device will fail to loaded.

### Supporting Documentation

N/A

# Chapter 25
# Power Management

## 25.1 Power Management User Manual

### Linux SDK for QorIQ Processors

### Abbreviations and Acronyms

WFI: Wait For Interrupt

DFS: Dynamic Frequency Scaling

WoL: Wake on LAN

PMC: Power Management Control

RCPM: Run Control and Power Management

FTM: FlexTimer Module

P1, P2, T4, B4, ... : stand for P1xxx, P2xxx, T4xxx, B4xxx processors

### Description

QorIQ Processors have features to minimize power consumption at several different levels. All processors based on e500v2, e500mc, e5500 and e6500 cores support DOZE (PH10), NAP (PH15) and SLEEP mode (LPM20). Some processors, such as P1022, T1040, also support deep sleep mode. The processors based on e6500 also support PW10(thread), PW20(core), PH20(core), PCL10(cluster) mode. The processors based on ARM core support WFI mode.

The following power management features are supported on various QorIQ processors:

- Dynamic power management
- Shutting down unused IP blocks
- Cores enter low power modes (such as DOZE, NAP)
- Processors enter low power state (SLEEP, DEEP SLEEP)
  - SLEEP mode: most of processor clocks are shut down
  - DEEP SLEEP mode: power is removed to cores, cache and most IP blocks of the processor such as DIU, eLBC, PEX, eTSEC2, USB2, SATA, eSDHC etc.
- CPU hotplug: If cores are down at runtime, they will enter low power state.
- DFS: Change the frequency of cores dynamically

The wake-up event sources caused quitting from low power mode are listed as below:

- Wake on LAN (WoL) using magic packet
- Wakeup by user defined packets, such as ARP request packet, unicast packet, etc.
- MPIC timer or FlexTimer
- Internal and external interrupts
- USB, such as device plug/unplug, remote wakeup

In QorIQ processors, there are two kinds of IP blocks to control the power management feature. They are called PMC and RCPM. For example, PMC is used in P1020, P2020, etc. and RCPM is used in P4080, P5020, etc.

For more information on a specific processor, refer to processor Reference Manual.

### U-boot Configuration

#### boot arguments passed to kernel

| boot arguments | Description |
| --- | --- |
| resume=/dev/sda2 root=/dev/$bdev rw | In hibernation case, add these arguments to the env variable "bootargs". |

### Kernel Configure Tree View Options

For Powerpc platform

<table>
<tr><th>Kernel Configure Tree View Options</th><th>Description</th></tr>
<tr><td><pre>
Kernel options -->
  [*] Suspend to RAM and standby
</pre></td><td>Enable sleep and deep sleep feature</td></tr>
<tr><td><pre>
Kernel options -->
  [*] Hibernation (aka 'suspend to disk')
  () Default resume partition (NEW)
</pre></td><td>Enable hibernation feature</td></tr>
<tr><td><pre>
Platform support  -->
  [*] MPIC Global Timer
  <*>   Freescale MPIC global timer wakeup driver
</pre></td><td>Enable the MPIC global timer and enable the MPIC timer wakeup driver.</td></tr>
<tr><td><pre>
Platform support -->
  CPU Frequency scaling -->
    [*] CPU Frequency scaling
    <*> CPU frequency translation statistics
    Default CPUFreq governor (userspace) -->
    -*- 'userspace' governor for userspace frequency scaling
        PowerPC CPU frequency scaling drivers  --->
          <*> CPU frequency scaling driver for Freescale
QorIQ SoCs

  CPU Frequency drivers -->
    [*] Support for Freescale MPC85xx CPU freq
</pre></td><td>Enable the CPU frequency driver for DFS</td></tr>
</table>

For Layerscape platform

<table>
<tr><th>Kernel Configure Tree View Options</th><th>Description</th></tr>
<tr><td><pre>
Power management options  -->
  [*] Suspend to RAM and standby
</pre></td><td>Enable sleep and deep sleep feature</td></tr>
<tr><td><pre>
Device Drivers  -->
  SOC (System On Chip) specific Drivers -->
    [*] Freescale Soc Drivers
    [*]   LS1021A Soc Drivers
    [*]     FTM alarm driver
</pre></td><td>Enable the FTM alarm driver.</td></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 |
|---|---|---|---|
| Suspend - standby | sleep | ALL | CONFIG_SUSPEND |
| Suspend - mem | deep sleep | P102x, T104x, LS1021 | CONFIG_SUSPEND |
| | wake by MPIC timer | P1, P2, P3, P4, P5, T4, B4 | CONFIG_MPIC_TIMER, CONFIG_FSL_MPIC_TIMER_WAKEUP |
| | wake by Flextimer | LS1021 | CONFIG_FTM_ALARM |
| Suspend - disk | | ALL(except LS1) | CONFIG_HIBERNATION |
| CPU idle | DOZE/PW10/WFI | ALL | |
| CPU hotplug | NAP/PH20/WFI | ALL | CONFIG_HOTPLUG_CPU |
| cpufreq | DFS | ALL | CONFIG_CPU_FREQ, CONFIG_CPU_FREQ_DEFAULT_GOV_USERSPACE |
| cpufreq | DFS | P1, P2 | CONFIG_MPC85xx_CPUFREQ |
| cpufreq | DFS | P3, P4, P5, T4, B4 | CONFIG_QORIQ_CPUFREQ |


Non-disclosure Agreement Required.

### 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 |
|---|---|---|
| ethtool | This utility allows querying and changing of ethernet card settings, such as speed, port, auto-negotiation, and PCI locations. | ethtool |

### Device Tree Binding

| Property | Type | Status | Description |
|---|---|---|---|
| fsl,wake-on-filer | String | Required | Enable wake up on user-defined packet |
| fsl,magic-packet | String | Required | Enable wake up on magic-packet |
| compatible | String | Required | Should be "fsl,mpc8536-pmc" or "fsl,mpc8548-pmc" |
| compatible | String | Required | Should be "fsl,mpic-global-timer" for MPIC timer |
| clk-handle | String | Required | Enable the ethernet clock |
| sleep | unsigned int | Required | Require if the IP block can work as a wakeup source. |

For processors integrated PMC:

```
enet0: ethernet@b0000 {
        .
        .
        .
        fsl,magic-packet;
        fsl,wake-on-filer;
        fsl,pmc-handle = <&etsec1_clk>;
        .
        .
        .
};
        .
        .
        .

power@e0070 {
        compatible = "fsl,p1022-pmc", "fsl,mpc8536-pmc", "fsl,mpc8548-pmc";
        reg = <0xe0070 0x20>;

        etsec1_clk: soc-clk@24 {
                fsl,pmcdr-mask = <0x00000080>;
        };
        etsec2_clk: soc-clk@25 {
                fsl,pmcdr-mask = <0x00000040>;
        };
        etsec3_clk: soc-clk@26 {
                fsl,pmcdr-mask = <0x00000020>;
        };
};
```

For processors integrated RCPM

```
        rcpm: global-utilities@e2000 {
                compatible = "fsl,t1040-rcpm", "fsl,qoriq-rcpm-2.0";
                reg = <0xe2000 0x1000>;
        };

        gpio@130000 {
                sleep = <&rcpm 0x00000040>;
        };
```

### Source Files

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

| Source File | Description |
| --- | --- |
| arch/powerpc/sysdev/fsl_pmc.c | The interface of other power management module with platform specified code. |
| arch/powerpc/sysdev/fsl_rcpm.c | The interface of other power management module with platform specified code for RCPM integrated devices. |
| arch/powerpc/platforms/85xx/sleep.S | Low level platform specified suspend and resume code |
| arch/powerpc/platforms/85xx/deepsleep.c | deep sleep related functions |
| arch/powerpc/platforms/85xx/cpufreq-jog.c | CPU frequency scaling driver for MPC8536 and P1022 |
| drivers/cpufreq/qoriq_cpufreq.c | CPU frequency scaling driver for qoriq chips |
| arch/powerpc/kernel/idle_e500.S | DOZE/NAP low power mode support code for e500 core |
| drivers/net/gianfar.c | eTSEC driver supports wake-up on user-defined packet (ARP, unicast) and wake-up on magic packet and |
| drivers/platform/fsl/sleep_fsm.c | configure the EPU FSM for deep sleep |
| arch/arm/mach-imx/pm-ls1.c | the sleep and deep sleep process code for LS1 |
| arch/arm/mach-imx/sleep-ls1.S | the low level code for entering deep sleep for LS1 |

### Verification in Linux

- DOZE&NAP

  When cores are in the idle state, kernel will put them into a low power state. This low power state depends on "powersave-nap" variable in kernel. If powersave-nap is zero (default value), the low power state is the DOZE mode, otherwise the NAP mode. The NAP mode is more power-saving than the DOZE mode. (only available on processors integrated e500v2, such as P2020)

  ```
  /* Enter NAP mode when cores idle */
  # echo 1 >/proc/sys/kernel/powersave-nap

  /* Enter DOZE mode when cores idle */
  # echo 0 >/proc/sys/kernel/powersave-nap
  ```

- Cpuidle Driver

QorIQ processor support multiple idle states that are differentiated by varying exit latencies and power consumption during idle. 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 which cpuidle driver is used. For LS1, the output is "ls1_cpuidle". */
# 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/
/sys/devices/system/cpu/cpu1/cpuidle/state0/
/sys/devices/system/cpu/cpu1/cpuidle/state1/
```

- SLEEP

  When the system is in the SLEEP mode, the ASLEEP LED will be light.

```
/* Enter SLEEP mode */
# echo standby > /sys/power/state
```

- DEEP SLEEP

  When the system is in the DEEP SLEEP mode, the ASLEEP will be light, LED VCORE and LED VSERDES will be dark.

```
/* Enter DEEP SLEEP mode */
# echo mem > /sys/power/state
```

- Wake up by MPIC timer or FTM timer

  The system can be wake up by the MPIC timer or FTM timer.

  Note: MPIC timer be supported on PowerPC. FTM timer be supported on ARM.

```
/* Start a MPIC timer. It will trigger an interrupt to wake up the system after 5
seconds. */
# echo 5 > /sys/devices/system/mpic/timer_wakeup

/* Start a FTM timer. It will trigger an interrupt to wake up the system after 5
seconds. */
/* LS1021A */
# echo 5 > /sys/devices/soc.2/29d0000.ftm0/ftm_alarm
/* LS2085A */
# echo 5 > /sys/devices/soc.2/2800000.ftm0/ftm_alarm

/* Enter SLEEP/DEEP SLEEP mode */
# echo standby > /sys/power/state
or
# echo mem > /sys/power/state
```

- Wake up by magic packets

```
/* Configure IP address */
```

```
# ifconfig eth0 192.168.20.168

/* Enable to wake up by magic packets */
# ethtool -s eth0 wol g

/* Enter SLEEP/DEEP SLEEP mode */
# echo standby > /sys/power/state
or
# echo mem > /sys/power/state

/* Use a magic packet tool on other device to send magic packets to the eth0 port. */

/* Use the following command to disable Wake-on-LAN */
# ethtool -s eth0 wol d
```

- Wake up by ARP request packets

```
/* Configure IP address */
# ifconfig eth0 192.168.20.168

/* Enable to wake up by ARP request packets */
# ethtool -s eth0 wol a

/* Enter SLEEP/DEEP SLEEP mode */
# echo standby > /sys/power/state
or
# echo mem > /sys/power/state

/* If using the ping command on PC to generate unicast packets, flush the arp cache
before executing the ping command. */
c:\> arp -d *
c:\> ping 192.168.20.168

/* Use the following command to disable Wake-on-LAN */
# ethtool -s eth0 wol d
```

- Wake up by unicast packets

```
/* Configure IP address */
# ifconfig eth0 192.168.20.168

/* Enable to wake up by unicast packets */
# ethtool -s eth0 wol u

/* Enter SLEEP/DEEP SLEEP mode */
# echo standby > /sys/power/state
or
# echo mem > /sys/power/state

/* If using the ping command on PC to generate arp request packets, add the arp entry
before executing the ping command. */
c:\> arp -s 192.168.20.168 xx:xx:xx:xx:xx:xx
c:\> ping 192.168.20.168

/* Use the following command to disable Wake-on-LAN */
# ethtool -s eth0 wol d
```

Note: ARP and unicast wake-up can be used together. But magic packet wake-up must be used solely.

- Wake up by USB event

```
/* Enable USB wakeup */
# echo enabled > /sys/bus/usb/devices/usb1/power/wakeup

/* Enter SLEEP mode */
# echo standby > /sys/power/state

/* Plug a USB device to USB1 port, or pull out a USB device. The system will wake up.
*/
```

- CPU Hotplug

```
You can offline cpu at runtime.

/* Offline cpu */
echo 0 > /sys/devices/system/cpu/cpu#/online
Note: cpu# is the cpu number you want to offline. cpu0 could not plug off as boot
core.

/* Online cpu */
echo 1 > /sys/devices/system/cpu/cpu#/online

Note: For e6500 based cores, if all cores in one cluster are offline the cluster will
automatically enter PCL10 state to save more power.
```

- 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.
```

- HIBERNATION mode

```
Assume /dev/sda2 is a SWAP partition. Add "resume=/dev/sda2" to the "bootargs"
environment variable in UBOOT.
For example:
The U-boot Environment:
=> setenv bootargs 'resume=/dev/sda2 root=/dev/$bdev rw console=$consoledev,$baudrate
$othbootargs'
=> saveenv

The Linux Kernel Environment:
[root@p1022ds root]#
[root@p1022ds root]# mkswap /dev/sda2
Setting up swapspace version 1, size = 1998738944 bytes

[root@p1022ds root]# swapon /dev/sda2
Adding 1951888k swap on /dev/sda2.  Priority:-1 extents:1 across:1951888k

[root@p1022ds root]# echo disk > /sys/power/state
PM: Syncing filesystems ... done.
Freezing user space processes ... (elapsed 0.00 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.00 seconds) done.
PM: Preallocating image memory... done (allocated 11659 pages)
PM: Allocated 46636 kbytes in 0.22 seconds (211.98 MB/s)
Suspending console(s) (use no_console_suspend to debug)
sd 0:0:0:0: [sda] Synchronizing SCSI cache
Disabling non-boot CPUs ...
Breaking affinity for irq 58
Cannot set affinity for irq 247
PM: Creating hibernation image:
PM: Need to copy 11316 pages
PM: Hibernation image created (11316 pages copied)
Enabling non-boot CPUs ...
setting frequency for cpu 0 to 999990 kHz, PLL ratio is 4/2
PMJCR request 04043c00 at CPU 0
PORPLLSR core freq 999MHz at CPU 0
Processor 1 found.
0: 499995kHz
1: 749992kHz
2: 999990kHz
setting frequency for cpu 1 to 999990 kHz, PLL ratio is 4/2
PMJCR request 04043c00 at CPU 1
PORPLLSR core freq 999MHz at CPU 1
setting frequency for cpu 1 to 999990 kHz, PLL ratio is 4/2
PMJCR request 04043c00 at CPU 1
PORPLLSR core freq 999MHz at CPU 1
setting frequency for cpu 1 to 999990 kHz, PLL ratio is 4/2
PMJCR request 04043c00 at CPU 1
PORPLLSR core freq 999MHz at CPU 1
CPU1 is up
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)
sd 0:0:0:0: [sda] Starting disk
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
PM: Saving image data pages (11328 pages) ...  76%
```

```
PHY: mdio@ffe24000:01 - Link is Down                                   done
PM: Wrote 45312 kbytes in 0.74 seconds (61.23 MB/s)
PM: S|
sd 0:0:0:0: [sda] Synchronizing SCSI cache
sd 0:0:0:0: [sda] Stopping disk
Disabling non-boot CPUs ...
Breaking affinity for irq 58
Power down.
System Halted, OK to turn off power

Reboot the kernel:
……………

CPU1 is up
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)
sd 0:0:0:0: [sda] Starting disk
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
Restarting tasks ...

done.

[root@p1022ds root]#
```

### Known Bugs, Limitations, or Technical Issues

- When using the MMC driver with the power management feature, CONFIG_MMC_UNSAFE_RESUME should also be enabled in linux. During the sleep period, do not remove the MMC/SD Card.
- On T4 series, when the CCB frequency is 666.667MHz, CPU frequency scaling can't work correctly. When CCB frequency is 600MHz, CPU frequency scaling works fine.
- The I/O peripherals such as PCIe and eTSEC may lose packets during the jog mode frequency transition.
- If you never use some function blocks, such as eTSEC2, USB2, SEC etc, you can permanently disable these unused blocks in the DEVDISR register to save more power. You must remove the unused nodes related to the unused blocks from .dts file, otherwise it can cause the system hang when the kernel probes these devices.

### Supporting Documentation

- QorIQ processor reference manuals

## 25.2 Thermal Management User Manual

### Description

The thermal management function is based on TMU (Thermal Monitoring Unit).

The driver sets two threshold for management function. If the CPU temperature crosses the first one (85 C), the driver will trigger CPU frequency limitation auto-scaling according to the temperature trend; If the CPU temperature crosses the second one (95 C, critical for core) the driver will shut down the system.

This driver also provides monitor function. The user space lm-sensors tools can get and display the CPU temperature value.

### Specifications

| | |
|---|---|
| Target boards: | T1040RDB, T1042RDB, T1023RDB, T1024RDB, LS1021ATWR. |
| Operating system: | Linux 3.12+ |

### Kernel Configure Tree View Options (For PowerPC platform)

| Kernel Configure Tree View Options | Description |
|---|---|
| `Platform support  --->`<br>`    CPU Frequency scaling  --->`<br>`        PowerPC CPU frequency scaling drivers  --->`<br>`            <*> CPU frequency scaling driver for Freescale QorIQ SoCs` | Enable CPUfreq driver. |
| `Device Drivers  --->`<br>`    [*] <*> Hardware Monitoring support  --->`<br>`    [*] Generic Thermal sysfs driver  --->`<br>`        [*] generic cpu cooling support`<br>`        [*] 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  --->`<br>`    CPU Frequency scaling  --->`<br>`        ARM CPU frequency scaling drivers  --->`<br>`            <*> CPU frequency scaling driver for Freescale QorIQ SoCs` | Enable CPUfreq driver. |
| `Device Drivers  --->`<br>`    [*] <*> Hardware Monitoring support  --->`<br>`    [*] Generic Thermal sysfs driver  --->`<br>`        [*] generic cpu cooling support`<br>`        [*] 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_HWMON | y/m/n | y | Enable hardware monitor support |

*Table continues on the next page...*

*Table continued from the previous page...*

| Option | Values | Default Value | Description |
|---|---|---|---|
| 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 (85 C), 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 (85 C), system will shutdown.

[Monitor:]

```
1. You can manually read the thermal interfaces in sysfs:
~$ cat /sys/class/hwmon/hwmon1/devices/temp1_input
35000

2. You can use lm_sensors tools as follows.
~ # sensors

tmu_thermal_zone-virtual-0
Adapter: Virtual device
temp1:        +35.0 C  (crit = +85.0 C)
```

# 25.3 System Monitor

## 25.3.1 Power Monitor User Manual

#### Description

There are two methods currently we can use to measure the power consumption which are called online and offline power monitoring respectively. The difference between them is that offline power monitoring support measuring power consumption during sleep or deep sleep.

The Power Monitor can be supported on P1022DS/P2020DS/P4080DS/P5020DS/P5040DS/BSC9131QDS/T4240QDS board.

This User guide uses the T4240QDS board as an example.

#### Specifications

| Target board: | Freescale P5020DS, P0122DS, P4080DS, P2020DS, P5040DS, BSC9131QDS, T4240QDS |
|---|---|
| CPU: | Freescale P5020/P0122/P4080/P5040/BSC9131/T4240/P2020 |
| Software: | Linux 3.8.13 or later |

### 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

<table>
<tr><th>Option</th><th>Description</th></tr>
<tr><td><pre>Device Drivers --->
 <*> Hardware Monitoring support  --->
    <*>   Texas Instruments INA219 and compatibles</pre></td><td>Enables INA220</td></tr>
<tr><td><pre>Device Drivers --->
    [*]   Enable compatibility bits for old user-space
    <*>   I2C device interface
    [*]   Autoselect pertinent helper modules
          I2C Hardware Bus support  --->
              <*> MPC107/824x/85xx/512x/52xx/83xx/86xx</pre></td><td>Enables I2C block device driver support</td></tr>
<tr><td><pre>Device Drivers --->
      <*>   I2C bus multiplexing support
          Multiplexer I2C Chip support  --->
              <*> Philips PCA954x I2C Mux/switches</pre></td><td>Enables I2C bus multiplexing PCA9547</td></tr>
</table>

### Compile-time Configuration Options

| Option | Values | Default Value | Description |
|---|---|---|---|
| CONFIG_I2C_MPC | y/n | y | Enable I2C bus protocol |
| SENSORS_INA2XX | y/n | y | Enables INA220 |
| CONFIG_I2C_MUX_PCA954x | y/n | y | Enables I2C multiplexing PCA9547 |

### Device Tree Binding

| Property | Type | Status | Description |
|---|---|---|---|
| compatible | String | Required | "Philips,pca9547" for pca9547 |
| reg | integer | Required | reg = <0x77> |
| compatible | String | Required | "ti,ina220" for ina220 |
| reg | integer | Required | reg = <the i2c address of ina220> |

```
Default node:
        i2c@118000 {
              pca9547@77 {
                  compatible = "philips,pca9547";
                  reg = <0x77>;
```

```
            #address-cells = <1>;
            #size-cells = <0>;

            channel@2 {
                #address-cells = <1>;
                #size-cells = <0>;
                reg = <0x2>;

                ina220@40 {
                    compatible = "ti,ina220";
                    reg = <0x40>;
                    shunt-resistor = <1000>;
                };

                ina220@41 {
                    compatible = "ti,ina220";
                    reg = <0x41>;
                    shunt-resistor = <1000>;
                };

                ina220@44 {
                    compatible = "ti,ina220";
                    reg = <0x44>;
                    shunt-resistor = <1000>;
                };

                ina220@45 {
                    compatible = "ti,ina220";
                    reg = <0x45>;
                    shunt-resistor = <1000>;
                };

                ina220@46 {
                    compatible = "ti,ina220";
                    reg = <0x46>;
                    shunt-resistor = <1000>;
                };

                ina220@47 {
                    compatible = "ti,ina220";
                    reg = <0x47>;
                    shunt-resistor = <1000>;
                };
            };
        };
```

### Source Files

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

| Source File | Description |
| --- | --- |
| drivers/i2c/muxes/i2c-mux-pca954x.c | PCA9547 driver |
| drivers/hwmon/ina2xx.c | ina220 driver |

### Test Procedure

Do the following to validate under the kernel

1. The bootup information is displayed:

```
......
i2c /dev entries driver
mpc-i2c ffe118000.i2c: timeout 1000000 us
mpc-i2c ffe118100.i2c: timeout 1000000 us
mpc-i2c ffe119000.i2c: timeout 1000000 us
mpc-i2c ffe119100.i2c: timeout 1000000 us
i2c i2c-0: Added multiplexed i2c bus 6
i2c i2c-0: Added multiplexed i2c bus 7
i2c i2c-0: Added multiplexed i2c bus 8
i2c i2c-0: Added multiplexed i2c bus 9
i2c i2c-0: Added multiplexed i2c bus 10
i2c i2c-0: Added multiplexed i2c bus 11
i2c i2c-0: Added multiplexed i2c bus 12
i2c i2c-0: Added multiplexed i2c bus 13
pca954x 0-0077: registered 8 multiplexed busses for I2C mux pca9547
ina2xx 8-0040: power monitor ina220 (Rshunt = 1000 uOhm)
ina2xx 8-0041: power monitor ina220 (Rshunt = 1000 uOhm)
ina2xx 8-0045: power monitor ina220 (Rshunt = 1000 uOhm)
ina2xx 8-0046: power monitor ina220 (Rshunt = 1000 uOhm)
ina2xx 8-0047: power monitor ina220 (Rshunt = 1000 uOhm)
ina2xx 8-0044: power monitor ina220 (Rshunt = 1000 uOhm)
......
```

2.

```

# sensors
ina220-i2c-8-40
Adapter: i2c-0-mux (chan_id 2)
in0:          +0.08 V
in1:          +1.06 V
power1:       35.00 W
curr1:       +32.77 A

ina220-i2c-8-41
Adapter: i2c-0-mux (chan_id 2)
in0:          +0.01 V
in1:          +0.01 V
power1:       60.00 mW
curr1:        +6.62 A

ina220-i2c-8-45
Adapter: i2c-0-mux (chan_id 2)
in0:          +0.01 V
in1:          +0.00 V
power1:       60.00 mW
curr1:        +8.20 A

ina220-i2c-8-46
Adapter: i2c-0-mux (chan_id 2)
in0:          +0.01 V
in1:          +0.01 V
power1:       60.00 mW
curr1:        +6.60 A

ina220-i2c-8-47
Adapter: i2c-0-mux (chan_id 2)
in0:          +0.01 V
in1:          +0.02 V
```

```
power1:       140.00 mW
curr1:         +7.40 A

ina220-i2c-8-44
Adapter: i2c-0-mux (chan_id 2)
in0:           +0.00 V
in1:           +1.51 V
power1:         1.86 W
curr1:         +1.23 A
```

NOTE

Please make sure to include the "sensors" command in your rootfs

### Offline Power Monitoring

Inside the FPGA of some Freescale QorIQ (PowerPC) reference boards is a microprocessor called the General Purpose Processor (GSMA). Running on the GSMA is the Data Collection Manager (DCM), which is used to periodically read and tally voltage, current, and temperature measurements from the on-board sensors. You can use this feature to measure power consumption while running tests, without having the host CPU perform those measurements.

This method support measuring power consumption when kernel is in sleep or deep sleep status. It gets the average power value of period from the time DCM starts to the time it ends.

### Module Loading

The device driver support either kernel built-in or module.

### Kernel Configure Tree View Options

| Kernel Configure Tree View Options | Description |
|---|---|
| `Device Drivers --->`<br>`[*] Misc devices  --->`<br>`    <*>   Freescale Data Collection Manager (DCM) driver` | Enables DCM driver |

### Compile-time Configuration Options

| Option | Values | Default Value | Description |
|---|---|---|---|
| CONFIG_FSL_DCM | y/n | y | Enable DCM module |

### Device Tree Binding

| Property | Type | Status | Description |
|---|---|---|---|
| compatible | String | Required | "fsl,t4240qds-fpga", "fsl,fpga-qixis" |
| reg | Integer | Required | reg = <3 0 0x300> |

```
Default node:
        ifc: localbus@ffe124000 {
               board-control@3,0 {
               compatible = "fsl,t4240qds-fpga", "fsl,fpga-qixis";
```

```
            reg = <3 0 0x300>;
        };
```

### Source Files

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

| Source File | Description |
| --- | --- |
| drivers/misc/fsl_dcm.c | DCM driver |

### Test Procedure

Do the following to validate under the Kernel:

1. The bootup information is displayed:

```
......
Freescale Data Collection Module is installed.
......
```

2. Start measuring measure power

```
# echo 1 > /sys/devices/platform/fsl-dcm.0/control
```

3. Stop measuring power

```
#echo 0 > /sys/devices/platform/fsl-dcm.0/control
```

4. Display the average power consumption

```

#cat /sys/devices/platform/fsl-dcm.0/result

Name                         Average
====================         ================
CPU voltage:                 1068    (mV)
CPU current:                 25910   (mA)
DDR voltage:                 1348    (mV)
DDR current:                 740     (mA)
CPU temperature:             38      (C)
```

### Supporting Documentation

- OCM_DCM_SoftwareOverview.pdf

## 25.3.2 Thermal Monitor User Manual

### Description

The Temperature Monitoring function is provided by the chip ADT7461.

This driver exports the values of Temperature to SYSFS. The user space lm-sensors tools can get and display these values.

### Specifications

| | |
|---|---|
| Target boards: | T1024RDB, T1040RDB, T1042RDB, T208xQDS, T208xRDB, B4QDS, T4240QDS, C293PCIe, P1022DS, P2041RDB, P3041DS, P4080DS, P5020DS, P5040DS, LS1021AQDS |
| Operating system: | Linux 2.6.35+ |

### Kernel Configure Tree View Options

| Kernel Configure Tree View Options | Description |
|---|---|
| `Device Drivers  --->`<br>`    [*] Hardware Monitoring support  --->`<br>`        [*] National Semiconductor LM90 and compatibles` | Enable thermal monitor chip driver like ADT7461. |
| `Device Drivers  --->`<br>`    <*> I2C bus multiplexing support  --->`<br>`        Multiplexer I2C Chip support  --->`<br>`            <*> 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 |

*Table continues on the next page...*

*Table continued from the previous page...*

| Source File | Description |
| --- | --- |
| drivers/hwmon/lm90.c | ADT7461 chip driver |
| drivers/i2c/i2c-mux.c | I2C bus multiplexing support |
| drivers/i2c/muxes/pca954x.c | PCA954x chip driver |

### Verification in Linux

There are two ways to get temperature results.

```
1. You can manually read the thermal interfaces in sysfs:
~$ ls /sys/class/hwmon/hwmon1/devices
alarms            temp1_crit        temp1_min_alarm   temp2_max_alarm
driver            temp1_crit_alarm  temp2_crit        temp2_min
hwmon             temp1_crit_hyst   temp2_crit_alarm  temp2_min_alarm
modalias          temp1_input       temp2_crit_hyst   temp2_offset
name              temp1_max         temp2_fault       uevent
power             temp1_max_alarm   temp2_input       update_interval
subsystem         temp1_min         temp2_max

~$ cat /sys/class/hwmon/hwmon1/devices/temp1_input
29000

2. You can use lm_sensors tools as follows.
~ # sensors

adt7461-i2c-1-4c
Adapter: MPC adapter
temp1:        +34.0 C  (low  =  +0.0 C, high = +85.0 C)
                       (crit = +85.0 C, hyst = +75.0 C)
temp2:        +48.5 C  (low  =  +0.0 C, high = +85.0 C)
                       (crit = +85.0 C, hyst = +75.0 C)
```

lm_sensors is integrated into Yocto file system by default. If there is no "sensors" command in your rootfs just add lmsensors-sensors package and build your own rootfs using Yocto:

```
IMAGE_INSTALL += "lmsensors-sensors"
```

## 25.3.3 Web-based System Monitor User Guide

Monitors the health of a system using a web browser in real time.

### Description

Web-based System Monitor is a tool for monitoring the health of your system using a web browser in real time. The following procedures will guide you to setup the system monitor.

### Kernel requirements

The raw data of this monitor system is collected from hardware monitor chips. So before you setup this monitor system you should enable the hwmon subsystem and drivers of monitor chips in the kernel. Kernel configure details are listed below.

 Non-disclosure Agreement Required. 

<table>
<tr><th>Kernel Configure Tree View Options</th><th>Description</th></tr>
<tr><td><pre>Device Drivers  --->
    [*] Hardware Monitoring support  --->
        [*] National Semiconductor LM90 and compatibles
        [*] Texas Instruments INA219 and compatibles</pre></td><td>Enable monitor chip drivers like ADT7461(ADT7481)/INA220, etc.</td></tr>
<tr><td><pre>Device Drivers  --->
    <*> I2C bus multiplexing support  --->
        Multiplexer I2C Chip support  --->
            <*> Philips PCA954x I2C Mux/switches</pre></td><td>Enable I2C PCA954x multiplexer support</td></tr>
</table>

Some monitor chips may not be included in the device tree. In this case you could add the device manually. Take adt7461 on T4240QDS as an example: the monitor is attached to I2C multiplexer PCA954x channel 3 in address 0x43. T4240QDS has 4 I2C controllers so the channel index of multiplexer start from 4 (represent the channel 0). ADT7461 is connected to channel 3 which indexed as 7. Use the flowing command to add the device to kernel:

```
~$ echo adt7461 0x4c > /sys/bus/i2c/devices/i2c-7/new_device
```

### Rootfs requirements

You could use the fsl-image-full rootfs in which all packages needed are included.

Or you can build your own rootfs using Yotco. Please follow the steps below.

1. Add following package group to your rootfs recipes like fsl-image-core.bb:

   ```
   IMAGE_INSTALL += "packagegroup-fsl-monitor"
   ```

2. If you are using ramdisk boot please add following settings to local.conf to get enough space for monitor systems:

   ```
   IMAGE_ROOTFS_EXTRA_SPACE = "100000"
   ```

   **NOTE**

   This will add 100000KB (100MB) more space to rootfs for monitor database. Each sensor needs about 10MB more space for logging raw data.

### Setting up system monitor

The monitor system will be setup automatically. What you need to do is to make sure that the network on board is working. Then you can monitor the system via any web browser by visiting: http://your.ip.address/senspix/sensors.cgi. This results page will refresh itself for every 10 seconds.

If you need to re-setup the system you could enter /usr/rrd directory and run:

```
$ make clean
$ make
```

**NOTE**

The System Monitor only works when system time is right. So you should guarantee that.

### How to configure the system monitor

The monitor results you see is based on the configuration file: "monitor.conf". It's automatically generated by scanning the hwmon subsystem. You could manually modify it too. Here is how:

1. Each line of this configuration file represents one monitor curve. It contains four fields formatted as follows:

   ```
   SENSDEV:MONITOR_TERM:DURATION:DESCRIPTION
   ```

   SENSDEV: The sensor data can be monitored from /sys/class/hwmon/ interfaces. Each sensor has a corresponding folder distinguished by hwmon# like hwmon0 or hwmon1. SENSDEV is the folder name.

   MONITOR_TERM: This is the item you want to read like temp1 or temp2 etc.

   DURATION: This is how long you want to see the results. You can set minute/hour/day here.

   DESCRIPTION: This DESCRIPTION will show up on the result picture helping you to understand the contents of the curve.

2. You could add/remove/resort the configuration file. After modifying it, you could enter the /usr/rrd directory and run:

   ```
   $ make config
   ```

3. Then the monitor results will be updated to what you configured.

### Run demo

We also provide scripts to cycle the system through different PM low power states to form a out-of-box demo for PM features. Please enter /usr/pm_demo directory and simply run:

```
$ ./pm_demo.sh
```

The output of the script will state the current PM features. It helps you to understand the system monitor results better.

**NOTE**

The demo could be terminated by CTRL-C and will apply the default PM features back.

# Chapter 26
# QDMA

## 26.1 QDMA User Manual

### Description

The LS SoC integrates Freescale's Queue Direct Memory Access Controller module. The qDMA controller transfers blocks of data between one source and one or more destinations. The blocks of data transferred can be represented in memory as contiguous or non-contiguous using scatter/gather table(s). Channel virtualization is supported through enqueuing of DMA jobs to, or dequeuing DMA jobs from, different work queues.

### Specifications

| Hardware: | Freescale LS1043, LS2085 |
|---|---|
| Software: | Linux 3.12+ |

### Source Files

The driver source is maintained in the Linux source tree.

| Source File | Description |
|---|---|
| drivers/dma/fsl-qdma.c | Freescale qDMA driver |

### 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 ---><br>    [*] DMA Engine support  ---> ---><br>        <*>   Freescale qDMA engine support | DMA engine subsystem driver and qDMA 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_QDMA | y/m/n | n | qDMA Driver |

### Verification in Linux

Enable CONFIG_DMA_TEST option when build kernel.

| Kernel Configure Tree View Options | Description |
| --- | --- |
| `Device Drivers --->`<br>`[*] DMA Engine support ---> --->`<br>`<*> Freescale qDMA engine support`<br>`<*> DMA Test client` | DMA engine subsystem driver and qDMA driver support |

```
root@ls1043aqds:~# echo 16384 > /sys/module/dmatest/parameters/test_buf_size
root@:~# echo 8 > /sys/module/dmatest/parameters/threads_per_chan
root@ls1043aqds:~# echo 4 > /sys/module/dmatest/parameters/max_channels
root@ls1043aqds:~# echo 1000 > /sys/module/dmatest/parameters/iterations
root@ls1043aqds:~# echo 1 > /sys/module/dmatest/parameters/run
dmatest: Started 8 threads using dma0chan0
root@ls1043aqds:~#
dmatest: dma0chan0-copy5: summary 1000 tests, 0 failures 1266 iops 10245 KB/s (0)
dmatest: dma0chan0-copy2: summary 1000 tests, 0 failures 1224 iops 9741 KB/s (0)
dmatest: dma0chan0-copy1: summary 1000 tests, 0 failures 1221 iops 9681 KB/s (0)
dmatest: dma0chan0-copy3: summary 1000 tests, 0 failures 1201 iops 9723 KB/s (0)
dmatest: dma0chan0-copy7: summary 1000 tests, 0 failures 1197 iops 9508 KB/s (0)
dmatest: dma0chan0-copy4: summary 1000 tests, 0 failures 1195 iops 9679 KB/s (0)
dmatest: dma0chan0-copy6: summary 1000 tests, 0 failures 1175 iops 9334 KB/s (0)
dmatest: dma0chan0-copy0: summary 1000 tests, 0 failures 1171 iops 9303 KB/s (0)
root@ls1043aqds:~#
```

### KnownBugs, Limitations, or Technical Issues

Just only support legacy mode now.

# Chapter 27
# Queue Manager (QMan) and Buffer Manager (BMan)

## 27.1 QMan/BMan Drivers Release Notes

### Description

This document describes Linux and USDPAA drivers for the QMan and BMan hardware blocks underlying the QorIQ(P4080/P3041/P5020/P5040/P2041/P1023/T4240/B4860) data path. QMan and BMan have independent drivers but their implementation and interfaces are very much analogous due to the similar CCSR and Corenet programming interfaces for each. As such, we will describe here "the driver", when in fact the description applies to both the QMan and BMan drivers equally and independently.

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

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

### CCSR, or "global config"

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

### Corenet portals

Use of QMan/BMan is via a number of independent portals(10 for P2041/P3041/P4080/P5020/P5040, 3 for P1023, 50 for T4240, 25 for B4860) located within sub-regions of a corenet memory map. Each portal has its own copy of the device interfaces that allow independent and parallel use of QMan/BMan functionality, possibly by different operating systems or by different processors (or threads) within a single operating system. When a hypervisor is managing multiple guests, each guest device tree indicates to the guest the portal it has access to. The device tree specifies a one to one mapping between cores and QMan/BMan portals. This mapping is assumed by the high-level QMan/BMan APIs. This should not be changed unless only low-level QMan/BMan APIs are to be used.

### Specifications

| | |
|---|---|
| Target board: | P4080DS/P3041DS/P5020DS/P5040DS/P2041RDB/P1023RDB/T4240QDS/ B4860QDS |
| CPU: | Freescale P4080/P3041/P5020/P5040/P2041/ P1023/T4240/B4860 |
| Operating system: | Linux3.12. Tested with kernel version supplied with this BSP. |

### 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 “Queue Manager, Buffer Manager API Reference Manual”).

#### DPAA allocator

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

#### Sysfs Interface

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

Debugfs Interface

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

### Module Loading

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

### QMan and BMan Kernel Configure Options

| Common Kernel Configure Options | Description |
|---|---|
| CONFIG_STAGING | Required in order to make “staging” drivers such as QMan/BMan available. |
| CONFIG_FSL_DPA | Required to build either QMan and/or BMan drivers. |
| CONFIG_FSL_DPA_CHECKING | Compiles in additional sanity-checks, at the expense of minor performance degradation. Recommended for debugging, but not for benchmarking. |

*Table continues on the next page...*

*Table continued from the previous page...*

| Common Kernel Configure Options | Description |
|---|---|
| CONFIG_FSL_DPA_CAN_WAIT | Compiles in support for interfaces and functionality that allow callers to optionally be put to “sleep” waiting for temporarily blocked resources to become available rather than returning errors. Eg. enqueuing when an enqueue ring is full. This is enabled unconditionally on linux. |
| CONFIG_FSL_DPA_CAN_WAIT_SYNC | Similar to “_CAN_WAIT”, but supports additional API flags for waiting for asynchronous operatoins to complete. Eg. after starting a volatile dequeue, wait for all dequeues to complete. This is enabled unconditionally on linux. |
| 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). |

*Table continues on the next page...*

*Table continued from the previous page...*

| QMan Kernel Configure Options | Description |
|---|---|
| CONFIG_FSL_QMAN_TEST_ERRATA | This requires the presence of cpu-affine portals, and performs testing that handling for known hardware-errata is correct. |
| CONFIG_FSL_QMAN_DEBUGFS | This option enables files in the debugfs filesystem. |

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

### Device-tree nodes

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

### CCSR, or "global config"

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

```
soc {
      #address-cells = <1>;
      #size-cells = <1>;
      device_type = "soc";
      compatible = "simple-bus";

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

      qman: qman@318000 {
          compatible = "fsl,qman";
          reg = <0x318000 0x1000>;
          interrupts = <16 2 1 3>;
```

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

### Contiguous memory

The `fsl,qman-fqd`, `fsl,qman-pfdr`, and `fsl,bman-fbpr` properties can be used to specify which contiguous sub-regions of memory should be used for the various memory requirements of QMan/BMan. The properties use 64-bit values, so 4 cells express the address/size 2-tuple to use. In the above example, if uncommented, the QMan/BMan resources would be allocated in the range `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 and BMan Corenet portals

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

```
qman-portals@ffe4200000 {
    #address-cells = <0x1>;
    #size-cells = <0x1>;
    compatible = "simple-bus";
    qportal0: qman-portal@0 {
        [...]
    };
    [...]
    qportal3: qman-portal@c000 {
        cell-index = <0x3>;
        compatible = "fsl,qman-portal";
        reg = <0xc000 0x4000 0x103000 0x1000>;
        interrupts = <110 0x2 0 0>;
        fsl,qman-channel-id = <0x3>;
    };
    [...]
};
```

### QMan FQID-range allocation

The "fsl,fqid-range" node (i.e. these "compatible" property types) indicates a range of FQIDs to use for FQID allocation by the QMan driver. The range within the node is specified using a property of the same name, and

whose two cells are the starting FQID value and the count. Multiple nodes can be provided to seed the allocator with a 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.

#### Linux

| Source Files | Description |
|---|---|
| include/linux/fsl_qman.h | The QMan driver APIs |
| include/linux/fsl_bman.h | The BMan driver APIs |
| drivers/staging/fsl_qbman/dpa_sys.h | Linux-specific definitions shared by the QMan and BMan drivers. |
| drivers/staging/fsl_qbman/dpa_alloc.c | Kernel support for dpa allocator |
| drivers/staging/fsl_qbman/qman_*.* | The QMan driver (excepting qman_test*.*, qman_debugfs*.*) |
| drivers/staging/fsl_qbman/qman_test*.* | The QMan driver self-tests |
| drivers/staging/fsl_qbman/qman_debugfs.* | The QMan debugfs interface |
| drivers/staging/fsl_qbman/bman_*.* | The BMan driver (excepting bman_test*.*, bman_debugfs*.*) |
| drivers/staging/fsl_qbman/bman_test*.* | The BMan driver self-tests |

*Table continues on the next page...*

*Table continued from the previous page...*

| Source Files | Description |
|---|---|
| drivers/staging/fsl_qbman/bman_debugfs.* | The BMan debugfs interface |
| drivers/staging/fsl_qbman/fsl_usdpaa.* | The USDPAA interface |
| arch/powerpc/boot/dts/p4080ds.dts<br>arch/powerpc/boot/dts/p3041ds.dts<br>arch/powerpc/boot/dts/p5020ds.dts<br>arch/powerpc/boot/dts/p5040ds.dts<br>arch/powerpc/boot/dts/p2041rdb.dts<br>arch/powerpc/boot/dts/p1023rdb.dts<br>arch/powerpc/boot/dts/t4240qds.dts<br>arch/powerpc/boot/dts/b4860qds.dts | Multiple device trees are provided to support the Reference System and Simulation targets. There are 32-bit and 36-bit variations of the device trees for guest use and native use (not under hypervisor) respectively. When booting with hypervisor, partition device-trees are described in hypervisor configuration. See hypervisor documentation for details. |
| arch/powerpc/boot/dts/p4080ds-usdpaa.dts<br>arch/powerpc/boot/dts/p3041ds-usdpaa.dts<br>arch/powerpc/boot/dts/p5020ds-usdpaa.dts<br>arch/powerpc/boot/dts/p5040ds-usdpaa.dts<br>arch/powerpc/boot/dts/p2041rdb-usdpaa.dts<br>arch/powerpc/boot/dts/t4240qds-usdpaa.dts<br>arch/powerpc/boot/dts/b4860qds-usdpaa.dts | USDPAA-specific device-trees. The differences with these is that a subset of portals and ethernet interfaces are reserved for use by USDPAA applications in user space. |

USDPAA

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

Build Procedure

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

### Test Procedure

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

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

Detecting the CCSR and portal device-tree nodes;

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

BMan portal initialised, cpu 0

BMan portal initialised, cpu 1

BMan portal initialised, cpu 2

BMan portal initialised, cpu 3

BMan portal initialised, cpu 4

BMan portal initialised, cpu 5

BMan portal initialised, cpu 6

BMan portal initialised, cpu 7

BMan portals initialised

BMan: BPID allocator includes range 32:32

QMan err interrupt handler present

QMan portal initialised, cpu 0

QMan portal initialised, cpu 1

QMan portal initialised, cpu 2

QMan portal initialised, cpu 3

QMan portal initialised, cpu 4

QMan portal initialised, cpu 5

QMan portal initialised, cpu 6
```

```
QMan portal initialised, cpu 7

QMan portals initialised

QMan: FQID allocator includes range 256:256

QMan: FQID allocator includes range 32768:32768

QMan: CGRID allocator includes range 0:256

QMan: pool channel allocator includes range 33:15

[...]
```

Running the QMan and BMan self-tests;

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

Running the BMan threshold test;

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

```
thread 7: exiting
bman_test_thresh: done
[...]
```

Running the QMan hot potato test;

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

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

### Changes since SDKv1.7

- Add new CEETM APIs to allow the user to set the LNI and channel shaping rate in bps format directly.

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

### Known Bugs, Limitations, or Technical Issues

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

### Supporting Documentation

1. Pxxxx QorIQ Integrated Multicore Communication Processor Reference Manual
2. T4240 QorIQ Advanced Multiprocessing Processor Reference Manual
3. B4860 QorIQ Qonverge Multicore Baseband Communication Processor Reference Manual
4. Queue Manager, Buffer Manager API reference Manual
5. USDPAA User Guide

# Chapter 28
# QMan BMan API Reference

## 28.1 About this document

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

### 28.1.1 Suggested Reading

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

## 28.2 Introduction to the Queue Manager and the Buffer Manager

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

### 28.2.1 O/S specifics

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 463for more details.

## 28.3 Buffer Manager

### 28.3.1 BMan Overview

#### 28.3.1.1 Buffer Manager's Function

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

In particular;

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

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

## 28.3.1.2 BMan's interfaces

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

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

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

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

Each portal presents the following BMan functionality;

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

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

## 28.3.2 BMan configuration interface

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

### 28.3.2.1 BMan Device-Tree Node

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

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

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

#### 28.3.2.1.1 Free Buffer Proxy Records

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

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

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

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

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

#### 28.3.2.1.2 Logical I/O Device Number (BMan)

Reads and writes to BMan's FBPR memory are subject to processing by the PAMU IO-MMU configuration of the SoC. In particular, BMan has an LIODN (Logical I/O Device Number) register setting that will be used by

PAMU 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.

**NOTE**

QorIQ P1023 doesn't have a PAMU, so LIODNs are not applicable for P1023.

### 28.3.2.2 Buffer Pool Node

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

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

#### 28.3.2.2.1 Buffer Pool ID

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

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

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

#### 28.3.2.2.2 Seeding Buffer Pools

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

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

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

#### 28.3.2.2.3 Depletion Thresholds

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

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

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

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

### 28.3.2.3 BMan Portal Device-Tree Node

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

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

```
bman-portal@0 {
        compatible = "fsl,bman-portal";
```

```
                reg = <0xe4000000 0x4000 0xe4100000 0x1000>;
                interrupts = <0x69 2>;
                interrupt-parent = <&mpic>;
                cell-index = <0x0>;
                cpu-handle = <&cpu3>;
        };
```

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

#### 28.3.2.3.1 Portal Initialization (BMan)

The driver is informed of the BMan portals that are available to it via the device-tree passed to the system from the boot process. For those portals that aren't reserved for 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.

#### 28.3.2.3.2 Portal sharing

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

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

## 28.4 BMan CoreNet portal APIs

The following sections describe interfaces provided by the BMan driver for manipulating portals, as defined in #unique_302.

### 28.4.1 BMan High-Level Portal Interface

#### 28.4.1.1 Overview (BMan)

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

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

NOTE

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

#### 28.4.1.2 Portal management (BMan)

The portal management API provides bman_affine_cpus(), which returns a mask that indicates which CPUs have auto-initialized portals associated with them. See #unique_306. 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);
```

##### 28.4.1.2.1 Modifying interrupt-driven portal duties (BMan)

Portals have various servicing duties they must perform in reaction to hardware events. The portal management API allows applications to control which of these duties/events are triggered by interrupt-handling versus those which are performed at the application's explicit request via bman_poll(). If portal-sharing is in effect (see #unique_308), these APIs won't succeed when called from a slave CPU.

```
#define BM_PIRQ_RCRI    0x00000002      /* RCR Ring (below threshold) */
#define BM_PIRQ_BSCN    0x00000001      /* Buffer depletion State Change */
/**
 * bman_irqsource_get - return the portal work that is interrupt-driven
 *
 * Returns a bitmask of BM_PIRQ_**I processing sources that are currently
 * enabled for interrupt handling on the current cpu's affine portal. These
 * sources will trigger the portal interrupt and the interrupt handler (or a
 * tasklet/bottom-half it defers to) will perform the corresponding processing
 * work. The bman_poll_***() functions will only process sources that are not in
 * this bitmask. If the current CPU is sharing a portal hosted on another CPU,
 * this always returns zero.
 */
u32 bman_irqsource_get(void);
/**
 * bman_irqsource_add - add processing sources to be interrupt-driven
 * @bits: bitmask of BM_PIRQ_**I processing sources

 * Adds processing sources that should be interrupt-driven, (rather than
* processed via bman_poll_***() functions). Returns zero for success, or
* -EINVAL if the current CPU is sharing a portal hosted on another CPU.
*/
int bman_irqsource_add(u32 bits);
/**
* bman_irqsource_remove - remove processing sources from being interrupt-driven
* @bits: bitmask of BM_PIRQ_**I processing sources
*
* Removes processing sources from being interrupt-driven, so that they will
* instead be processed via bman_poll_***() functions. Returns zero for success,
* or -EINVAL if the current CPU is sharing a portal hosted on another CPU. */
int bman_irqsource_remove(u32 bits);
```

#### 28.4.1.2.2 Processing non-interrupt-driven portal duties (BMan)

If portal-sharing is in effect (see #unique_308), these APIs won't succeed when called from a slave CPU.

```
/**
 * bman_poll_slow - process anything that isn't interrupt-driven.
 *
 * This function does any portal processing that isn't interrupt-driven. NB,
 * unlike the legacy wrapper bman_poll(), this function will deterministically
 * check for the presence of portal processing work and do it, which implies
 * some latency even if there's nothing to do. The bman_poll() wrapper on the
 * other hand (like the qman_poll() wrapper) attenuates this by checking for
 * (and doing) portal processing infrequently. Ie. such that qman_poll() and
 * 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);
```

#### 28.4.1.2.3 Recovery support (BMan)

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

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

#### 28.4.1.2.4 Determining if the release ring is empty

```
/**
 * bman_rcr_is_empty - Determine if portal's RCR is empty
 *
 * For use in situations where a cpu-affine caller needs to determine when all
 * releases for the local portal have been processed by BMan but can't use the
 * BMAN_RELEASE_FLAG_WAIT_SYNC flag to do this from the final bman_release().
```

```
 * The function forces tracking of RCR consumption (which normally doesn't
 * happen until release processing needs to find space to put new release
 * commands), and returns zero if the ring still has unprocessed entries,
 * non-zero if it is empty.
 */
int bman_rcr_is_empty(void);
```

### 28.4.1.3 Pool Management

To work with BMan buffer pools, a pool object must be created. As explained in Depletion State on page 401, 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);
```

### 28.4.1.4 Releasing and Acquiring Buffers

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

```
/* Flags to bman_release() */
#define BMAN_RELEASE_FLAG_WAIT       0x00000001 /* wait if RCR is full */
```

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

### 28.4.1.5 Depletion State

It is possible for portals to track depletion state changes to any of the 64 buffer pools supported in BMan. As described in Pool Management on page 399, 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) [...]
```

# 28.5 Queue Manager

## 28.5.1 QMan Overview

### 28.5.1.1 Queue Manager's Function

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

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

### 28.5.1.2 Frame Descriptors

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

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

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

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

As with BMan not caring whether the 48-bit tokens it manages are real buffer addresses or not, the same is mostly true for QMan with respect to the frame descriptors it manages. QMan ignores the memory addresses

present in the frame descriptor, unless it is dequeued via a portal configured for data stashing and is dequeued from a frame queue that is configured for frame data (or annotation) stashing. However QMan always pays attention to the length field of frame descriptors. In general, the only field that can be safely used as a "pass-through" value without any QMan consequences is the 32-bit cmd/status field.

## 28.5.1.3 Frame Queue Descriptors (QMan)

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

## 28.5.1.4 Work Queues

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

## 28.5.1.5 Channels

A channel is a fixed, hardware-defined association of 8 work queues, also thought of as "priority work queues". This grouping is convenient in that QMan provides sophisticated prioritization support for 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.

## 28.5.1.6 Portals

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

## 28.5.1.7 Dedicated Portal Channels

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

## 28.5.1.8 Pool Channels

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

### 28.5.1.9 Portal Sub-Interfaces

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

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

### 28.5.1.10 Frame queue dequeuing

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

#### 28.5.1.10.1 Unscheduled Dequeues

One can dequeue from a specific frame queue, but that frame queue must necessarily be "idle" - or in QMan terminology, "unscheduled". It is an illegal action to attempt to dequeue directly from a frame queue that is in a "scheduled" state. Specifically, unscheduled dequeues require the frame queue to be in the "Parked" or "Retired" state (described in Frame Queue States on page 406).

#### 28.5.1.10.2 Scheduled Dequeues

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

#### 28.5.1.10.3 Pull Mode

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

clobber the previous command before QMan has prepared its execution). The PDQCR command register can perform scheduled and unscheduled dequeues.

### 28.5.1.10.4 Push Mode

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

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

### 28.5.1.10.5 Stashing to Processor Cache

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

NOTE

P1023 doesn’t support per-frame-descriptor information stashing as described above. The FLIODN is not applicable to P1023.

## 28.5.1.11 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.

## 28.5.1.12 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.

### 28.5.1.12.1 Dequeue Atomicity

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

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

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

#### 28.5.1.12.2 Parking Scheduled FQs

As noted above in #unique_336, if a FQ is currently "held active" in the portal, software can request that it be move to the Parked state once its final DQRR entry is consumed, rather than rescheduled which is the normal behavior. As we will also see in Force Eligible on page 408, 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.

#### 28.5.1.12.3 Order Preservation & Discrete Consumption Acknowledgement

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

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

Note, QMan has other functionality called Order Restoration that is completely unrelated to the above - Order Restoration is a mechanism to restore frames into their intended order once they been allowed to get out of order, using sequence numbers and "reassembly windows" within QMan, see Order Restoration on page 409. The above "hold active" mechanisms are to prevent frames from getting out of order in the first place.

### 28.5.1.13 Force Eligible

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

When QMan processes a Force Eligible command, it does two things - it tags the frame queue descriptor with a flag that is visible in subsequent DQRR entries, and, if the frame queue is Tentatively Scheduled (because it is empty), it will move the frame queue to the Truly Scheduled state (linked to a work queue). The result is that the frame queue "will receive dequeue processing soon", whether that was already happening or not. Fundamentally, when QMan is dequeuing from the FQ a short while later, it will treat the frame queue as "hold active", even if it isn't configured for hold active treatment. As such, software can request that the FQ be parked rather than rescheduled once the DQRR entry is consumed. See #unique_340.

### 28.5.1.14 Enqueue Rejections

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

### 28.5.1.15 Order Restoration

Frame queue descriptors can serve one or both of two complimentary purposes. A small subset of fields in the FQDs are used to implement an "Order Restoration Point", which allows an FQD to act as a reassembly window for out-of-sequence enqueues. FQDs also contain a sequence number field that generates increasing sequence numbers for all frames dequeued from the FQ. This dequeue activity sequence number is also called an "Order Definition Point". The idea is that frames dequeued from a given FQ (ODP) may get out-of-sequence during processing before they're enqueued onto an egress FQ, so the enqueue function allows one to not only specify the 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 421, 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 416, 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.

## 28.5.2 QMan configuration interface

The QMan configuration interface is an encapsulation of the QMan CCSR register space and the global/error interrupt source. Whereas QMan portals provide independent channels for accessing QMan functionality, the configuration interface represents the QMan device itself. The QMan configuration interface is presently limited to the device-tree node that represents it.

### 28.5.2.1 QMan device-tree node

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

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

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

#### 28.5.2.1.1 Frame Queue Descriptors

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

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

The treatment and alignment requirements of this property are the same as in #unique_347.

#### 28.5.2.1.2 Packed Frame Descriptor Records

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

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

The treatment and alignment requirements of this property are the same as in #unique_347.

#### 28.5.2.1.3 Logical I/O Device Number (QMan)

This property is the same as described in #unique_350, but for use by QMan when accessing FQD and PFDR memory (rather than BMan's FBPR memory).

### 28.5.2.2 QMan pool channel device-tree node

Each QMan software portal has its own dedicated channel of work queues. QMan also provides "pool channels" that all software portals can optionally dequeue from - this is described in Portals on page 403. 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>;
};
```

#### 28.5.2.2.1 Channel ID

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

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

### 28.5.2.3 QMan portal device-tree node

The QMan Corenet portal interface in QorIQ P4080 provides up to 10 distinct memory-mapped interfaces for use by software to interact efficiently with QMan functionality. These are described in Portals on page 403

and Portal Sub-Interfaces on page 404. 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 #unique_354, the other QMan-specific portal properties are described below.

## 28.5.2.3.1 Portal Access to Pool Channels

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

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

## 28.5.2.3.2 Stashing Logical I/O Device Number

This property, when used in QMan portal nodes, declares two LIODN values for use by QMan when performing dequeue stashing to processor cache. These are documented in #unique_357. 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>;
```

## 28.5.2.3.3 Portal Initialization (QMan)

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

As with the BMan driver, the QMan driver will automatically associate initialised QMan portals with the CPU to which they are configured, only one a one-per-CPU basis (if multiple portals are configured for the same CPU,

only one is used). Please see #unique_308 for an explanation of this behaviour in the BMan documentation, the QMan behaviour is identical.

#### 28.5.2.3.4 Auto-Initialization

As with the BMan driver, the QMan driver will, by default, automatically initialize QMan portals as they are parsed out of the device-tree. Please see #unique_360 for an explanation of this behavior in the BMan documentation. The QMan behavior is identical.

## 28.6 QMan portal APIs

The following sections describe interfaces provided by the QMan driver for manipulating portals. These are defined in QMan portal device-tree node on page 410, and described in Portals on page 403 and Portal Sub-Interfaces on page 404.

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.

### 28.6.1 QMan High-Level Portal Interface

#### 28.6.1.1 Overview (QMan)

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

This interface provides locking and arbitration of portal operations from multiple software contexts and/or threads (ie. the portal is shared). In cases where a resource is busy, the interface also gives callers the option of blocking/ sleeping until the 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.

### 28.6.1.2 Frame and Message Handling

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

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

### 28.6.1.3 Portal management (QMan)

The portal management API provides qman_affine_cpus(), which returns a mask that indicates which CPUs have auto-initializaed portals associated with them. See QMan portal device-tree node on page 410. 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);
```

#### 28.6.1.3.1 Modifying interrupt-driven portal duties (QMan)

Portals have various servicing duties they must perform in reaction to hardware events. The portal management API allows applications to control which of these duties/events are triggered by interrupt-handling versus those which are performed at the application's explicit request via qman_poll() (or more specifically, via qman_poll_dqrr() and qman_poll_slow()). If portal-sharing is in effect (see #unique_308), 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);
```

### 28.6.1.3.2 Processing non-interrupt-driven portal duties (QMan)

If portal-sharing is in effect (see #unique_308), 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 Freescale Confidential Proprietary Freescale Semiconductor
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);
```

## 28.6.1.3.3 Recovery support (QMan)

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

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

## 28.6.1.3.4 Stopping and restarting dequeues to the portal

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

## 28.6.1.3.5 Manipulating the portal static dequeue command

```
/**
 * qman_static_dequeue_add - Add pool channels to the portal SDQCR
 * @pools: bit-mask of pool channels, using QM_SDQCR_CHANNELS_POOL(n)
 *
 * Adds a set of pool channels to the portal's static dequeue command register
```

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

#### 28.6.1.3.6 Determining if the enqueue ring is empty

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

### 28.6.1.4 Frame queue management

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

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

```
#define QMAN_FQ_FLAG_NO_ENQUEUE      0x00000001 /* can't enqueue */
#define QMAN_FQ_FLAG_NO_MODIFY       0x00000002 /* can only enqueue */
#define QMAN_FQ_FLAG_TO_DCPORTAL     0x00000004 /* consumed by CAAM/PME/FMan */
#define QMAN_FQ_FLAG_LOCKED          0x00000008 /* multi-core locking */
#define QMAN_FQ_FLAG_AS_I            0x00000010 /* query h/w state */
#define QMAN_FQ_FLAG_DYNAMIC_FQID    0x00000020 /* (de)allocate fqid */
struct qman_fq {
```

```
        /* Caller of qman_create_fq() provides these demux callbacks */
        struct qman_fq_cb {
                qman_cb_dqrr dqrr;      /* for dequeued frames */
                qman_cb_mr ern;         /* for s/w ERNs */
                qman_cb_mr dc_ern;      /* for diverted h/w ERNs */
                qman_cb_mr fqs;         /* frame-queue state changes*/
        } cb;
          /* Internal to the driver, don't touch. */
          [...]
};
/**
 * qman_create_fq - Allocates a FQ
 * @fqid: the index of the FQD to encapsulate, must be "Out of Service"
 * @flags: bit-mask of QMAN_FQ_FLAG_*** options
 * @fq: memory for storing the 'fq', with callbacks filled in
 *
 * Creates a frame queue object for the given @fqid, unless the
 * QMAN_FQ_FLAG_DYNAMIC_FQID flag is set in @flags, in which case a FQID is
 * dynamically allocated (or the function fails if none are available). Once
 * created, the caller should not touch the memory at 'fq' except as extended to

 * adjacent memory for user-defined fields (see the definition of "struct
 * qman_fq" for more info). NO_MODIFY is only intended for 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);
```

### 28.6.1.4.1 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);
```

## 28.6.1.4.2 Initialize a FQ

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

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

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

## 28.6.1.4.3 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);
```

## 28.6.1.4.4 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);
```

## 28.6.1.4.5 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);
```

### 28.6.1.4.6 Query a FQD from QMan

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

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

### 28.6.1.4.7 Unscheduled (volatile) dequeuing of a FQ

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

### 28.6.1.4.8 Set FQ flow control state

```
/**
 * qman_fq_flow_control - Set the XON/XOFF state of a FQ
 * @fq: the frame queue object to be set to XON/XOFF state, must not be 'oos',
 * or 'retired' or 'parked' state
 * @xon: boolean to set fq in XON or XOFF state
 *
```

```
 * The frame should be in Tentatively Scheduled state or Truly Schedule sate,
 * otherwise the IFSI interrupt will be asserted.
 */
int qman_fq_flow_control(struct qman_fq *fq, int xon);
```

### 28.6.1.5 Enqueue Command (without ORP)

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

### 28.6.1.6 Enqueue Command with ORP

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

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

 *   of a frame. */
#define QMAN_ENQUEUE_FLAG_NLIS       0x01000000
/* - this flag performs no enqueue but fills in an ORP sequence number that
 *   would otherwise block it (eg. if a frame has been dropped). */
#define QMAN_ENQUEUE_FLAG_HOLE       0x02000000
/* - this flag performs no enqueue but advances NESN to the given sequence
 *   number. */
#define QMAN_ENQUEUE_FLAG_NESN       0x04000000
/*
 * qman_enqueue_orp - Enqueue a frame to a frame queue using an ORP
 * @fq: the frame queue object to enqueue to
 * @fd: a descriptor of the frame to be enqueued
 * @flags: bit-mask of QMAN_ENQUEUE_FLAG_*** options
 * @orp: the frame queue object used as an order restoration point.
 * @orp_seqnum: the sequence number of this frame in the order restoration path
 *
 * Similar to qman_enqueue(), but with the addition of an Order Restoration
 * Point (@orp) and corresponding sequence number (@orp_seqnum) for this
 * enqueue operation to employ order restoration. Each frame queue object acts
 * as an Order Definition Point (ODP) by providing each frame dequeued from it
 * with an incrementing sequence number, this value is generally ignored unless
 * that sequence of dequeued frames will need order restoration later. Each
 * frame queue object also encapsulates an Order Restoration Point (ORP), which
 * is a re-assembly context for re-ordering frames relative to their sequence
 * numbers as they are enqueued. The ORP does not have to be within the frame
 * queue that receives the enqueued frame, in fact it is usually the frame
 * queue from which the frames were originally dequeued. For the purposes of
 * order restoration, multiple frames (or "fragments") can be enqueued for a
 * single sequence number by setting the QMAN_ENQUEUE_FLAG_NLIS flag for all
 * enqueues except the final fragment of a given sequence number.  Ordering
 * between sequence numbers is guaranteed, even if fragments of different
 * sequence numbers are interlaced with one another. Fragments of the same
 * sequence number will retain the order in which they are enqueued. If no
 * enqueue is 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);
```

### 28.6.1.7 DCA Mode

As described in #unique_382, 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 421, 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);
```

## 28.6.1.8 Congestion Management Records

QMan supports a fixed number[3] 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;
};
/* 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;
```

[3] 256 for P4080/P5020/P3041, 64 for P1023

```
                     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);
```

## 28.6.1.9 Zero-Configuration Messaging

As described in Overview (QMan) on page 412, 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 418).

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);
```

# 28.6.1.10 FQ allocation

## 28.6.1.10.1 Ad-hoc FQ allocator

As described in #unique_387, BMan buffer pool ID zero is currently reserved for use as an ad-hoc FQ allocator. As seen in Frame queue management on page 416, 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).

## 28.6.1.10.2 FQ range allocator

The following APIs allow software to allocate and release arbitrary ranges of FQIDs, but it should be noted that the current version of the Freescale 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);
```

#### 28.6.1.10.3 Future FQ allocator changes

Please note that a future version of the Freescale 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.

### 28.6.1.11 Helper functions

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

**NOTE**

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

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

```
        int err;
        err = qman_query_fq_np(fq, &np);
        if (err)
                return err;
        if ((np.state & QM_MCR_NP_STATE_MASK) == QM_MCR_NP_STATE_OOS)
                return 1;
        return 0;
}
```

# 28.7 QMan CEETM APIs

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

## 28.7.1 QMan CEETM Device-Tree Node

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

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

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

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

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

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

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

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

## 28.7.2 The token rate of CEETM shaper

### 28.7.2.1 The token rate structure

The QMan CEETM provides two rate shapers – CR and ER shapers to shape the traffic with the given rate. Both shapers use the token credit rate and credit update reference period to determine the shaper’s output rate (in

bits/second). The token rate is specified in bytes with an 11 bit integer and a 13 bit fractional part, and can be configured via CEETM shaper configuration commands. Here is the definition of the token rate structure:

```
/* Token Rate Structure
/* The shaping rates are based on a "credit" system and a pre-configured h/w
 * internal timer. The following type represents a shaper "rate" parameter as a
 * fractional number of "tokens". Here's how it works. This (fractional) number
 * of tokens is added to the shaper's "credit" every time the h/w timer elapses
 * (up to a limit which is set by another shaper parameter). Every time a frame
 * is enqueued through a shaper, the shaper deducts as many tokens as there are
 * bytes of data in the enqueued frame. A shaper will not allow itself to
 * enqueue any frames if its token count is negative. As such:
 *
 *         The rate at which data is enqueued is limited by the
 *         rate at which tokens are added.
 *
 * Therefore if the user knows the period between these h/w timer updates in
 * seconds, they can calculate the maximum traffic rate of the shaper (in
 * bytes-per-second) from the token rate. And vice versa, they can calculate
 * the token rate to use in order to achieve a given traffic rate.
 */
struct qm_ceetm_rate {
        /* The token rate is; whole + (fraction/8192) */
        u32 whole:11; /* 0..2047 */
        u32 fraction:13; /* 0..8191 */
};
```

## 28.7.2.2 The APIs to convert token rate and shapers output rate

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

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

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

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

/**
 * qman_ceetm_tokenrate2bps - Given a 'token_rate', compute the corresponding
```

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

## 28.7.3 CEETM Sub-portal

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

### 28.7.3.1 Claim/release sub-portal

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

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

## 28.7.4 CEETM LNI - Logic Network Interface

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

### 28.7.4.1 Claim/release LNI

```
/**
 * qman_ceetm_lni_claim - Claims an unclaimed LNI.
 * @lni: the returned LNI object, if successful.
 * @dcp_id: specifies the desired FMan block (and thus the relevant CEETM instance).
 * @lni_idx: the desired LNI index.
 *
 * Returns zero for success, or -EINVAL for failure, which will happen
 * if the LNI is not available or has already been claimed (and not yet
 * successfully released),or lni_idx is out of range.
 *
 * Note that there may be multiple driver domains (or instances) that need to transmit
 * out the same LNI, so this claim is only guaranteeing exclusivity within the
 * domain of the driver being called. See qman_ceetm_sp_claim() and
 * qman_ceetm_sp_get_lni() for more information.
 */
int qman_ceetm_lni_claim(struct qm_ceetm_lni **lni,
                         enum qm_dc_portal dcp_id,
                         unsigned int lni_idx);

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

### 28.7.4.2 Map sub-portal with LNI

```
/**
 * qman_ceetm_sp_set_lni
 * qman_ceetm_sp_get_lni – Set/get the LNI that the sub-portal is currently mapped to.
 * @sp: the given sub-portal.
 * @lni(in “set” function): the LNI object which the sub-portal will be mapped to.
 * @lni_idx(in “get” function): the LNI index which the sub-portal is mapped to.
 *
 * Returns zero for success. * Returns –EINVAL for “set” function when this sp-lni
mapping has been set, or
 * configure mapping command returns error, and
 * -EINVAL for “get” function when this sp-lni mapping is not set, or the query
 * mapping command returns error.
 *
```

```
 * This may be useful in situations where multiple driver
 * domains have access to the same sub-portals in order to all be able to
 * transmit out the same physical interface (perhaps they're on different IP
 * addresses or VPNs, so FMan is splitting Rx traffic and here we need to
 * converge Tx traffic). In that case, a control-plane is likely to use
 * qman_ceetm_lni_claim() followed by qman_ceetm_sp_set_lni() to configure the
 * sub-portal, and other domains are likely to use qman_ceetm_sp_get_lni()
 * followed by qman_ceetm_lni_claim() in order to determine the LNI that the
 * control-plane had assigned. This is why the "get" returns an index, whereas
 * the "set" takes an (already claimed) LNI object.
 */
int qman_ceetm_sp_set_lni(struct qm_ceetm_sp *sp,
                          struct qm_ceetm_lni *lni);
int qman_ceetm_sp_get_lni(struct qm_ceetm_sp *sp,
                          unsigned int *lni_idx);
```

### 28.7.4.3 Configure LNI shaper

The LNI provides two rate shapers that can be used to shape the traffic from all class queue channels which are shaped on input to the channel scheduler. The LNI shapers are used to shape or rate limit the aggregate of the class queue channels which have each been shaped. The two shaper rates are applied only to frames shaped by the corresponding shaper at the channel shaper level.

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

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

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

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

#### 28.7.4.4 Configure LNI traffic class flow control

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

### 28.7.5 CEETM class queue channel

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

#### 28.7.5.1 Claim/release class queue channel

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

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

### 28.7.5.2 Configure the shaper of class queue channel

```
/**
 * qman_ceetm_channel_enable_shaper
 * qman_ceetm_channel_disable_shaper - Enable/Disable shaping on the given channel.
 * @channel: the given channel.
 * @coupled: indicates whether surplus tokens should be added to the excess-rate
 * token count (up to the excess-rate token limit) when the number of (unused)
 * committed-rate tokens reach the committed-rate token limit.
 *
 * Whenever a claimed channel is first enabled for shaping, its committed and
 * excess token rates and limits are zero, so will need to be changed to do
 * anything useful. The shaper can subsequently be enabled/disabled without
 * resetting the shaping parameters, but the shaping parameters will be reset
 * when the channel is released.
 *
 * Returns 0 for success and -EINVAL for failure if the channel shaping has been
 * enabled or disabled, or the management command returns error
 */
int qman_ceetm_channel_enable_shaper(struct qm_ceetm_channel *channel, int coupled);
int qman_ceetm_channel_disable_shaper(struct qm_ceetm_channel *channel);


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


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

```
int qman_ceetm_channel_set_commit_rate(struct qm_ceetm_channel *channel,
                                 const struct qm_ceetm_rate *token_rate,
                                 u16 token_limit);
int qman_ceetm_channel_get_commit_rate(struct qm_ceetm_channel *channel,
                                 struct qm_ceetm_rate *token_rate,
                                 u16 *token_limit);
int qman_ceetm_channel_set_excess_rate(struct qm_ceetm_channel *channel,
                                 const struct qm_ceetm_rate *token_rate,
                                 u16 token_limit);
int qman_ceetm_channel_get_excess_rate(struct qm_ceetm_channel *channel,
                                 struct qm_ceetm_rate *token_rate,
                                 u16 *token_limit);

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

#### 28.7.5.3 Configure the token limit as the weight for the unshaped channel

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

```
/**
 * qman_ceetm_channel_set_weight
 * qman_ceetm_channel_get_weight - Set/get the weight for unshaped channel.
 * @channel: the given channel.
 * @token_limit: the desired token limit as the weight of unshaped channel for “set”
 * function, or the queried token limit for “get” function.
 *
 * Returns zero for success, or -EINVAL if the channel is a shaped channel, or
```

```
 * the management command returns error
 */
int qman_ceetm_channel_set_weight(struct qm_ceetm_channel *channel,
                                  u16 token_limit);
int qman_ceetm_channel_get_weight(struct qm_ceetm_channel *channel,
                                  u16 *token_limit);
```

#### 28.7.5.4 Set CR/ER eligibility for CQs within a CEETM channel

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

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

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

### 28.7.6 CEETM class queue

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

#### 28.7.6.1 Claim/release CQ

```
/**
 * qman_ceetm_cq_claim - Claims an individual class queue.
 * @cq: the returned class queue object, if successful.
 * @channel: the given class queue channel.
 * @idx: is from 0 to 7 (representing CQ0 to CQ7).
 * @ccg: represents the class congestion group that this class queue
 * should be subscribed to, or NULL if no congestion group membership is desired.
```

```
 *
 * Returns zero for success, or -EINVAL if @idx is out of range 0 to 7 or this class
 * queue has been claimed, or configure class queue command returns error, or
 * returns -ENOMEM if allocating CQ memory fails.
 */
int qman_ceetm_cq_claim(struct qm_ceetm_cq **cq,
                            struct qm_ceetm_channel *channel,
                            unsigned int idx,
                            struct qm_ceetm_ccg *ccg);

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

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

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

/**
 * qman_ceetm_drain_cq - drain the CQ till it is empty.
```

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

### 28.7.6.2 Change CQ weight

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


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

/**
 * qman_ceetm_ratio2wbfs - Given a weight, find the nearest possible weight code.
 * @numerator: numerator part of the given weight.
 * @denominator: denominator part of the given weight.
 * @weight_code: the weight code computed from the given weight.
 *
 * If the user needs to know how close this is, convert the resulting weight code
```

```
 * back to a weight and compare.
 *
 * Returns zero for success, or -ERANGE if "numerator/denominator" is outside the
 * range of weights.
 */
int qman_ceetm_ratio2wbfs(u32 numerator,
                          u32 denominator,
                          struct qm_ceetm_weight_code *weight_code);

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

#### 28.7.6.3 Query CQ statistics

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

### 28.7.7 CEETM logical FQID

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

#### 28.7.7.1 Claim/release LFQID

```
/**
```

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

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

## 28.7.7.2 Configure/Query Dequeue Context Table

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

## 28.7.7.3 Create FQ for LFQ

```
/**
 * qman_ceetm_create_fq - Initialise a FQ object for the LFQ.
 * @lfq: the given logic FQ.
 * @fq: the FQ object created for the LFQ
 *
 * The FQ object can be used in qman_enqueue() and qman_enqueue_orp() APIs to target
 * a logical FQID (and the class queue it is associated with). Note that this FQ
 * object can only be used for enqueues, and in the case of qman_enqueue_orp()
 * it can not be used as the 'orp' parameter, only as 'fq'. This FQ object can not
 * (and shouldn't) be destroyed, it is only valid as long as the underlying 'lfq'
```

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

## 28.7.8 CEETM Class Congestion Group(CCG)

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

### 28.7.8.1 Claim/release CCG

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

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

```
int qman_ceetm_ccg_release(struct qm_ceetm_ccg *ccg);
```

## 28.7.8.2 Configure CCG

```
/* This struct is used to specify attributes for a CCG. The 'we_mask' field
 * controls which CCG attributes are to be updated, and the remainder specify
 * the values for those attributes. A CCG counts either frames or the bytes
 * within those frames, but not both ('mode'). A CCG can optionally cause
 * enqueues to be rejected, due to tail-drop or WRED, or both (they are
 * independent options, 'td_en' and 'wr_en_g,wr_en_y,wr_en_r'). Tail-drop can be
 * level-triggered due to a single threshold ('td_thres') or edge-triggered due
 * to a "congestion state", but not both ('td_mode'). Congestion state has
 * distinct entry and exit thresholds ('cs_thres_in' and 'cs_thres_out'), and
 * notifications can be sent to software the CCG goes in to and out of this
 * congested state ('cscn_en').
 * The struct qm_cgr_cs_thres has already been defined in fsl_qman.h
 */
struct qm_ceetm_ccg_params {
        /* Boolean fields together in a single bitfield struct */
        struct {
                /* Whether to count bytes or frames. 1==frames */
                int mode:1;
                /* En/disable tail-drop. 1==enable */
                int td_en:1;
                /* Tail-drop on congestion-state or threshold. 1=threshold */
                int td_mode:1;
                /* Generate congestion state change notifications. 1==enable */
                int cscn_en:1;
                /* Enable WRED rejections (per colour). 1==enable */
                int wr_en_g:1;
                int wr_en_y:1;
                int wr_en_r:1;
        } __packed;
        /* Tail-drop threshold. See qm_cgr_thres_[gs]et64(). */
        struct qm_cgr_cs_thres td_thres;
        /* Congestion state thresholds, for entry and exit. */
        struct qm_cgr_cs_thres cs_thres_in;
        struct qm_cgr_cs_thres cs_thres_out;
        /* Overhead accounting length. Per-packet "tax", from -128 to +127 */
        signed char oal;
        /* WRED parameters. */
        struct qm_cgr_wr_parm wr_parm_g;
        struct qm_cgr_wr_parm wr_parm_y;
        struct qm_cgr_wr_parm wr_parm_r;
};

/* Bits used in 'we_mask' to qman_ceetm_ccg_set(), controls which attributes of
 * the CCGR are to be updated. */
#define QM_CCGR_WE_CDV     0x0000 /* cdv */
#define QM_CCGR_WE_MODE         0x0001 /* mode (bytes/frames) */
#define QM_CCGR_WE_TD_EN        0x0004 /* congestion state tail-drop enable */
#define QM_CCGR_WE_TD_MODE      0x4000 /* tail-drop mode (state/threshold) */
#define QM_CCGR_WE_TD_THRES     0x2000 /* tail-drop threshold */
#define QM_CCGR_WE_CS_THRES_IN  0x0002 /* congestion state entry threshold */
#define QM_CCGR_WE_CS_THRES_OUT 0x1000 /* congestion state exit threshold */
#define QM_CCGR_WE_CSCN_EN      0x0010 /* congestion notification enable */
#define QM_CCGR_WE_CSCN_TUPD    0x0008 /* CSCN target update */
```

```
#define QM_CCGR_WE_OAL          0x0800 /* overhead accounting length */
#define QM_CCGR_WE_WR_PARM_G    0x0400 /* WRED parameters - green */
#define QM_CCGR_WE_WR_PARM_Y    0x0200 /* WRED parameters - yellow */
#define QM_CCGR_WE_WR_PARM_R    0x0100 /* WRED parameters - red */
#define QM_CCGR_WE_WR_EN_G      0x0080 /* WRED enable - green */
#define QM_CCGR_WE_WR_EN_Y      0x0040 /* WRED enable - yellow */
#define QM_CCGR_WE_WR_EN_R      0x0020 /* WRED enable - red */

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

## 28.7.8.3 Query CCG statistics

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

## 28.7.8.4 Set/get Congestion State Change Notification Target

```
/** qman_ceetm_cscn_swp_get - Query whether a given software portal index is
  * in the cscn target mask.
  * @ccg: the give CCG object.
  * @swp_idx: the index of the software portal.
  * @cscn_enabled: 1: cscn is enabled in this swp. 0: cscn is not enabled
  * in this swp.
  *
  * Return 0 for success, or -EINVAL if command in set/get function fails.
  */
int qman_ceetm_cscn_swp_get(struct qm_ceetm_ccg *ccg,
                            u16 swp_idx,
                            unsigned int *cscn_enabled);

/** qman_ceetm_cscn_dcp_set - Add or remove a direct connect portal from the\
```

```
  * target mask.
  * qman_ceetm_cscn_swp_get - Query whether a given direct connect portal index
  * is in the cscn target mask.
  * @ccg: the give CCG object.
  * @dcp_idx: the index of the direct connect portal.
  * @cscn_enabled: 1: Set the dcp to be cscn target. 0: remove the dcp from
  * the target mask.
  *
  * Return 0 for success, or -EINVAL if command in set/get function fails.
  */
int qman_ceetm_cscn_dcp_set(struct qm_ceetm_ccg *ccg,
                            u16 dcp_idx,
                            unsigned int cscn_enabled);
int qman_ceetm_cscn_dcp_get(struct qm_ceetm_ccg *ccg,
                            u16 dcp_idx,
                            unsigned int *cscn_enabled);
```

## 28.8 Other QMan APIs

The following sections describe the interfaces provided by the QMan driver for manipulating QMan device.

### 28.8.1 Waterfall Power Management

Waterfall power management is a mechanism that works between the QMan and the e6500's Drowsy Core mode. The basic idea is that when using multiple cores and pool channels to forward frames, and the QMan receives less traffic than required to keep all cores busy, it no longer assigns to the higher numbered cores (i.e. software portals) in a pool. This feature is supported from QMan3.0, and the APIs below can only be called at SoC with QMan3.0

```
/**
 * qman_set_wpm - Set waterfall power management
 *
 * @wpm_enable: boolean, 1 = enable wpm, 0 = disable wpm.
 *
 * Return 0 for success, return -ENODEV if QMan misc_cfg register is not
 * accessible.
 */
int qman_set_wpm(int wpm_enable);
/**
 * qman_get_swp - Query the waterfall power management setting
 *
 * @wpm_enable: boolean, 1 = enable wpm, 0 = disable wpm.
 *
 * Return 0 for success, return -ENODEV if QMan misc_cfg register is not
 * accessible.
 */
int qman_get_wpm(int *wpm_enable);
```

# 28.9 USDPAA-specific APIs

## 28.9.1 Overview

The USDPAA SDK includes a port of the QMan and BMan drivers to linux user space, and assumes a pthreads-based programming model, and the use of a single application/process instance.

Unlike conventional user space interfaces to hardware (in which the kernel manipulates hardware interfaces on behalf of user space processes), the USDPAA QMan and BMan drivers operate on the hardware directly from user space. The underlying mechanisms for this are based on the USDPAA Linux character device. Put briefly, the kernel exposes the devices to user space as character devices with various attributes, which allow the user space drivers to mmap() the Corenet portal regions directly. Interrupt handling works by disabling the portal interrupt when it asserts and conveying the interrupt event to user-space via the USDPAA device's file-descriptor (the user-space portal driver can re-enable the interrupt whenever it so chooses).

The key distinction between the QMan (or BMan) drivers found in the Linux kernel and the one in USDPAA is that the latter does not perform a global initialization step to parse all available portals and initialize and assign them to CPUs/threads. The model in USDPAA assumes that the application is responsible for creating its own pthreads, and is responsible for making those threads affine to the appropriate CPUs (if indeed it chooses to). Such applications simply call into a USDPAA-specific API to "enable" the thread for QMan/BMan portal usage (specifying the desired CPU-affinity for the portal), and the USDPAA driver at that point will "find" an unused portal suitable for the requested CPU and initialize and bind it to the pthread via thread-local storage, or fail if no such portal was available. Portals can likewise be released from threads when they are no longer required, without requiring the pthread itself to be destroyed.

Not only do both the Linux kernel and USDPAA drivers initialize all portals at start-up, but their portals are managed as CPU-*local* associations. USDPAA on the other hand manages portals as *thread-local* associations. Nonetheless, portals are (typically) configured for a particular CPU-affinity, meaning that ring and data stashing will target the designated CPU, so USDPAA applications are advised to run their threads on the CPUs for which their portals are configured - failing to do so will not break the system nor the application, but will yield significant performance degradation relative to an optimized system.

NOTE

*The "recovery_mode" parameters in the following APIs are currently unsupported, they are reserved until a future release, so should always be set to 0 (or "FALSE").*

## 28.9.2 Thread initialization

An application pthread can request that the QMan drivers initialize and bind a portal for use by that thread by calling the following APIs. This API must be called and return success prior to calling any of the interfaces mentioned in QMan portal APIs on page 412 or BMan CoreNet portal APIs on page 396. Note, this interface is dependent on the device-tree layer having been initialized, see Device-tree dependency on page 448.

```
int qman_thread_init(void);
int qman_thread_init_idx(uint32_t idx);
int bman_thread_init(void);
int bman_thread_init_idx(uint32_t idx);
```

Note that the 'idx' parameter influences the driver's search for the portal with the specified portal index. The pthread is required to already be affine to the given CPU.

As usual with "int"-valued APIs, a return value of zero indicates success, otherwise a standard negative error number is returned in event of failure.

Likewise, the thread-portal association can be ended (and thus make the underlying portal available for another user/thread) by calling the following APIs.

```
int qman_thread_finish(void);
int bman_thread_finish(void);
```

## 28.9.3 FQID/BPID allocation

As of the current release of USDPAA, support for allocation of FQIDs and BPIDs is via hard-coded ranges encoded within the drivers. This will be revised in a future release.

To enable the allocation mechanisms in the USDPAA QMan (and BMan) drivers, the following APIs must be called once from a thread that has already successfully bound to a QMan (or BMan) portal via qman_thread_init() (or bman_thread_init(), respectively).

```
int qman_global_init(int recovery_mode);
int bman_global_init(int recovery_mode);
```

Note that this API does not need to be called from all threads, only from one, but that it should be called before using any QMan/BMan APIs that require dynamic allocation of FQIDs or BPIDs.

## 28.9.4 Interrupt handling

### 28.9.4.1 USDPAA file-descriptors

Interrupt handling in USDPAA is done by reading a standard file descriptor that is created when a portal is assigned to a thread. The application should use the poll() or select() API to determine when the file descriptor becomes readable which indicates an interrupt has occurred. The following APIs return the current USDPAA file-descriptors for portals bound to the calling thread.

```
int qman_thread_fd(void);
int bman_thread_fd(void);
```

### 28.9.4.2 Processing interrupt-driven portal duties

It should be noted that the QMan and BMan drivers allow applications to dynamically configure which portal "duties" are to be triggered and performed by interrupts versus polling. See Modifying interrupt-driven portal duties (BMan) on page 397 and Modifying interrupt-driven portal duties (QMan) on page 413 for details. Prior to going into a blocking read(), select(), poll(), [...] on the file-descriptor, the application will need to ensure that the "duties" it wishes to wait for are put into IRQ mode (ie. added to the portal's "irqsource") prior to blocking, otherwise the presence of such work will not trigger any interrupt and so will not wake up the sleeping process. Likewise, if going back into polling mode and expecting polling APIs to process such portal duties, the application will need to remove such duties from the "irqsource".

For portal duties that are in the "irqsource", and after any USDPAA-supported file-descriptor operation that indicates that an interrupt was received, the application pthread can call the following APIs to post-process any such interrupt-driven duties:

```
/* Post-process interrupts. NB, the kernel IRQ handler disables the interrupt
 * line before notifying us, and this post-processing re-enables it once
 * processing is complete. As such, it is essential to call this before going
 * into another blocking read/select/poll. */
void qman_thread_irq(void);
void bman_thread_irq(void);
```

### 28.9.5 Device-tree dependency

The QMan/BMan drivers “find” portals by referring to information extracted from the device-tree, as represented in the procfs filesystem located at /proc/device-tree. This information is handled by a USDPAA “of” driver, which must be initialised before any attempts are made to initialise threads for QMan/BMan use as described in Thread initialization on page 446. As of the current USDPAA release, the “of” driver must be explicitly initialised by applications prior to QMan/BMan thread initialisation, though this may change in future releases. (As a side note, other USDPAA mechanisms, such as application-side configuration parsing may also be dependent on this device-tree layer, and so the “of” init should occur prior to all such dependencies.)

The API to initialise the “of” driver is of_init(), which parses the /proc/device-tree representation into an internal representation of C data-structures.

```
#ifndef OF_INIT_DEFAULT_PATH
#define OF_INIT_DEFAULT_PATH "/proc/device-tree"
#endif
int of_init_path(const char *dt_path);
/* Use of this wrapper is recommended. */
static inline int of_init(void)
{
        return of_init_path(OF_INIT_DEFAULT_PATH);
}
```

Conversely, the of_finish() API cleans up all data structures and handles created by of_init(), which may be useful to satisfy leak-checking tools, or persistent process/thread recycling schemes.

```
void of_finish(void);
```

## 28.10 Sysfs and debugfs QMan/BMan interfaces

This chapter describes the qman and bman interfaces available via sysfs and debugfs.

> NOTE
>
> P1023 has different CCSRBAR address and QMan/BMan offsets within CCSRBAR, thus the paths for sysfs are different under /sys/devices/ on P1023. For non-P4080 devices, it is generally recommended to determine these from the device-tree, the SoC-specific Reference Manual, and/or by examining the sysfs filesystem at run-time.

### 28.10.1 QMan sysfs

#### 28.10.1.1 /sys/devices/ffe000000.soc/ffe318000.qman

Description:

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

#### 28.10.1.2 /sys/devices/ffe000000.soc/ffe318000.qman/error_capture

Description:

This directory contains a snapshot of error related qman attributes.

#### 28.10.1.3 /sys/devices/ffe000000.soc/ffe318000.qman/error_capture/sbec_<0..6>

Description:

Provides a count of the number of single bit ECC errors that have occurred when reading from one of the QMan internal memories. The range <0..6> represent a QMAN internal memory region defined as follows:

0: FQD cache memory

1: FQD cache tag memory

2: SFDR memory

3: WQ context memory

4: Congestion Group Record memory

5: Internal Order Restoration List memory

6: Software Portal ring memory

This file is read-reset.

#### 28.10.1.4 /sys/devices/ffe000000.soc/ffe318000.qman/sfdr_in_use

Description:

Reports the number of SFDR currently in use. The minimum value is 1. This file is not available on Rev 1.0 of P4080 QorIQ.

This file is read-only

/sys/devices/ffe000000.soc/ffe318000.qman/pfdr_fpc

Description:

Total Packed Frame Descriptor Record Free Pool Count in external memory.

This file is read-only

#### 28.10.1.5 /sys/devices/ffe000000.soc/ffe318000.qman/pfdr_cfg

Description:

Used to read the configuration of the dynamic allocation policy for PFDRs. The value is used to account for PFDR that may be required to complete any currently executing operations in the sequencers.

This file is read-only.

#### 28.10.1.6 /sys/devices/ffe000000.soc/ffe318000.qman/idle_stat

Description:

This file can be used to determine when QMan is both idle and empty. The possible values are:

0: All work queues in QMan are NOT empty and QMan is NOT idle.

1: All work queues in QMan are NOT empty and QMan is idle.

2: All work queues in QMan are empty

3: All work queues in QMan are empty and QMan is idle.

This file is read-only.

#### 28.10.1.7 /sys/devices/ffe000000.soc/ffe318000.qman/err_isr

Description:

QMan contains one dedicated interrupt line for signaling error conditions to software. This file identifies the source of the error interrupt within QMan. The value is displayed in hexadecimal format. Refer to the appropriate QorIQ SOC Reference Manual for a description of the QMAN_ERR_ISR register.

This file is read-only.

#### 28.10.1.8 /sys/devices/ffe000000.soc/ffe318000.qman/dcp<0..3>_dlm_avg

Description:

These files contain an EWMA (exponentially weighted moving average) of dequeue latency samples for dequeue commands received on the sub portal. The range <0..3> refers to each of the direct-connect portals. The display format is as follows: <avg_interger>.<avg_fraction>

This file can be seeded with a interger value. The input interger is processed in the following manner: <avg_fraction> = lowest 8 bits / 256 , <avg_interger> = next 12 bits

ex: echo 0x201 > dcp0_dlm_avg

cat dcp0_dlm_avg

0.00390625

This file is read-write

#### 28.10.1.9 /sys/devices/ffe000000.soc/ffe318000.qman/ci_rlm_avg

Description:

This file contains an EWMA (exponentially weighted moving average) of read latency samples for reads on CoreNet initiated by QMan. The display format is as follows: <avg_interger>.<avg_fraction>

This file can be seeded with a interger value. The input interger is processed in the following manner: <avg_fraction> = lowest 8 bits / 256 , <avg_interger> = next 12 bits

ex: echo 0x201 > ci_rlm_avg

cat ci_rlm_avg

0.00390625

This file is read-write

### 28.10.2 BMan sysfs

#### 28.10.2.1 /sys/devices/ffe000000.soc/ffe31a000.bman

Description:

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

#### 28.10.2.2 /sys/devices/ffe000000.soc/ffe31a000.bman/error_capture

Description:

This directory contains a snapshot of error related BMan attributes.

#### 28.10.2.3 /sys/devices/ffe000000.soc/ffe31a000.bman/error_capture/sbec_<0..1>

Description:

Provides a count of the number of single bit ECC errors that have occurred when reading from one of the BMan internal memories. The range <0..1> represent a BMAN internal memory region defined as follows:

0: Stockpile memory 0

1: Software Portal ring memory

This file is read-reset.

#### 28.10.2.4 /sys/devices/ffe000000.soc/ffe31a000.bman/pool_count

Description:

This directory contains a snapshot of the number of free buffers available in any of the buffer pools.

#### 28.10.2.5 /sys/devices/ffe000000.soc/ffe31a000.bman/fbpr_fpc

Description:

This file returns a snapshot of the Free Buffer Proxy Record free pool size. Total Free Buffer Proxy Record Free Pool Count in external memory.

This file is read-only

#### 28.10.2.6 /sys/devices/ffe000000.soc/ffe31a000.bman/err_isr

Description:

BMan contains one dedicated interrupt line for signaling error conditions to software. This file identifies the source of the error interrupt within BMan. The value is displayed in hexadecimal format. Refer to the appropriate QorIQ SOC Reference Manual for a description of the BMAN_ERR_ISR register.

This file is read-only.

### 28.10.3 QMan debugfs

#### 28.10.3.1 /sys/kernel/debug/qman

Description:

This directory contains various QMan device debugging attributes.

#### 28.10.3.2 /sys/kernel/debug/qman/query_cgr

Description:

Query the entire contents of a Congestion Group Record. The file takes as input the Congestion Group Record ID. The output of the file returns the various CGR fields.

For example, if we want to query cgr_id 10 we would do the following:

# echo 10 > query_cgr

# cat query_cgr

Query CGR id 0xa

wr_parm_g MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0

wr_parm_y MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0

wr_parm_r MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0

wr_en_g: 0, wr_en_y: 0, we_en_r: 0

cscn_en: 0

cscn_targ: 0

cstd_en: 0

cs: 0

cs_thresh_TA: 0, cs_thresh_Tn: 0

i_bcnt: 0

a_bcnt: 0

## 28.10.3.3 /sys/kernel/debug/qman/query_congestion

Description:

Query the state of all 256 Congestion Groups in QMan. This is a read-only file. The output of the file returns the state of all congestion group records. The state of a congestion group is either "in congestion" or "not in congestion". Since CGR are normally not in congestion, only CGR which are in congestion are returned. If no CGR are in congestion, then this is indicated.

For example, if we want to perform a query we would do the following:

# cat query_congestion

Query Congestion Result

All congestion groups not congested.

## 28.10.3.4 /sys/kernel/debug/qman/query_fq_fields

Description:

Query the frame queue programmable fields. This file takes as input the frame queue id to be queried on a subsequent read. The output of this file returns all the frame queue programmable fields. The default frame queue id is 1.

Refer to the appropriate QorIQ SOC Reference Manual for detailed explanation on the return values.

For example, if we determine that our application is using frame queue 482 we could use this file in the following manner:

# echo 482 > query_fq_fields

# cat query_fq_fields

Query FQ Programmable Fields Result fqid 0x1e2

orprws: 0

oa: 0

olws: 0

cgid: 0

fq_ctrl:

Aggressively cache FQ

Don't block active

Context-A stashing

Tail-Drop Enable

dest_channel: 33

dest_wq: 7

ics_cred: 0

td_mant: 128

td_exp: 7

ctx_b: 0x19e

ctx_a: 0x78b59e18

ctx_a_stash_exclusive:

FQ Ctx Stash

Frame Annotation Stash

ctx_a_stash_annotation_cl: 1

ctx_a_stash_data_cl: 2

ctx_a_stash_context_cl: 2

#### 28.10.3.5 /sys/kernel/debug/qman/query_fq_np_fields

Description:

Query the frame queue non programmable fields. This file takes as input the frame queue id to be queried on a subsequent read. The output of this file returns all the frame queue non programmable fields. The default frame queue id is 1.

Refer to the appropriate QorIQ SOC Reference Manual for detailed explanation on the return values.

For example, if we determine that our application is using frame queue 482 we could use this file in the following manner:

# echo 482 > query_fq_np_fields

# cat query_fq_np_fields

Query FQ Non Programmable Fields Result fqid 0x1e2

force eligible pending: no

retirement pending: no

state: Out of Service

fq_link: 0x0

odp_seq: 0

orp_nesn: 0

orp_ea_hseq: 0

orp_ea_tseq: 0

orp_ea_hptr: 0x0

orp_ea_tptr: 0x0

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

#### 28.10.3.6 /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

### 28.10.3.7 /sys/kernel/debug/qman/query_ceetm_ccgr

Description:

Query the configuration and state fields within a CEETM Congestion Group Record that relate to congestion management(CM). This file takes input as the DCP id(most left 8 bits) and CEETM Congestion Group Record ID(most right 24 bits). The output of the file returns the various CCGR fields.

For example, if we want to query ccgr_id 7 of DCP 0, we would do the following:

# echo 0x00000007 > query_ceetm_ccgr

# cat query_ceetm_ccgr

Query CCGID 7

Query CCGR id 7 in DCP 0

wr_parm_g MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0

wr_parm_y MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0

wr_parm_r MA: 0, Mn: 0, SA: 0, Sn: 0, Pn: 0

wr_en_g: 0,

wr_en_y: 0,

we_en_r: 0

cscn_en: 0

cscn_targ_dcp:

cscn_targ_swp:

td_en: 0

cs_thresh_in_TA: 0,

cs_thresh_in_Tn: 0

cs_thresh_out_TA: 0,

cs_thresh_out_Tn: 0

td_thresh_TA: 0,

td_thresh_Tn: 0

mode: byte count

i_cnt: 0

a_cnt: 0

### 28.10.3.8 /sys/kernel/debug/qman/query_wq_lengths

Description:

Query the length of the Work Queues in a particular channel. This file takes as input a specified channel id. The output of this file returns the lengths of the work queues on the specified channel.

For example, if we want to query channel 1 we would do the following:

# echo 1 > query_wq_lengths

# cat query_wq_lengths

Query Result For Channel: 0x1

wq0_len : 0

wq1_len : 0

wq2_len : 0

wq3_len : 0

wq4_len : 0

wq5_len : 0

wq6_len : 0

wq7_len : 0

#### 28.10.3.9 /sys/kernel/debug/qman/fqd/avoid_blocking_[enable | disable]

Description:

Query Avoid_Blocking bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Avoid_Blocking bit mask enabled or disabled.

For example, if we want to find all frame queues with Avoid_Blocking enabled, we would do the following:

```
# cat avoid_blocking_enable
List of fq ids with: Avoid Blocking :enabled
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001de,0x0001df,0x0001e0,0x0001e1,
0x0001ea,0x0001eb,0x0001ec,0x0001ed,0x0001f6,0x0001f7,0x0001f8,0x0001f9,
...
Total FQD with: Avoid Blocking : enabled = 528
Total FQD with: Avoid Blocking : disabled = 32239
```

#### 28.10.3.10 /sys/kernel/debug/qman/fqd/prefer_in_cache_[enable | disable]

Description:

Query Prefer_in_Cache bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Prefer_in_Cache bit mask enabled or disabled.

For example, if we want to find all frame queues with Prefer_in_Cache enabled, we would do the following:

```
# cat prefer_in_cache_enable
List of fq ids with: Prefer in cache :enabled
0x0001ca,0x0001cb,0x0001cc,0x0001cd,0x0001ce,0x0001cf,0x0001d0,0x0001d1,
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001d6,0x0001d7,0x0001d8,0x0001d9,
...
Total FQD with: Prefer in cache : enabled = 560
Total FQD with: Prefer in cache : disabled = 32207
```

#### 28.10.3.11 /sys/kernel/debug/qman/fqd/cge_[enable | disable]

Description:

Query Congestion_Group_Enable bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Congestion_Group_Enable bit mask enabled or disabled.

For example, if we want to find all frame queues with Congestion_Group_Enable disabled, we would do the following:

```
# cat cge_disable
List of fq ids with: Congestion Group Enable :disabled
0x000001,0x000002,0x000003,0x000004,0x000005,0x000006,0x000007,0x000008,
0x000009,0x00000a,0x00000b,0x00000c,0x00000d,0x00000e,0x00000f,0x000010,
...
Total FQD with: Congestion Group Enable : enabled = 0
Total FQD with: Congestion Group Enable : disabled = 32767
```

## 28.10.3.12 /sys/kernel/debug/qman/fqd/cpc_[enable | disable]

Description:

Query CPC_Stash_Enable bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their CPC_Stash_Enable bit mask enabled or disabled.

For example, if we want to find all frame queues with CPC Stash disabled, we would do the following:

```
# cat cpc_disable
List of fq ids with: CPC Stash Enable :disabled
0x000001,0x000002,0x000003,0x000004,0x000005,0x000006,0x000007,0x000008,
0x000009,0x00000a,0x00000b,0x00000c,0x00000d,0x00000e,0x00000f,0x000010,
...
Total FQD with: CPC Stash Enable : enabled = 0
Total FQD with: CPC Stash Enable : disabled = 32767
```

## 28.10.3.13 /sys/kernel/debug/qman/fqd/cred

Description:

Query Intra-Class Scheduling bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma 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
```

## 28.10.3.14 /sys/kernel/debug/qman/fqd/ctx_a_stashing_[enable | disable]

Description:

Query Context_A bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma 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
```

## 28.10.3.15 /sys/kernel/debug/qman/fqd/hold_active_[enable | disable]

Description:

Query Hold_Active bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma 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
```

## 28.10.3.16 /sys/kernel/debug/qman/fqd/orp_[enable | disable]

Description:

Query ORP bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma 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
```

## 28.10.3.17 /sys/kernel/debug/qman/fqd/sfdr_[enable | disable]

Description:

Query Force_SFDR_Allocate bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Force_SFDR_Allocate bit mask enabled or disabled.

For example, if we want to find all frame queues with Force_SFDR_Allocate enabled, we would do the following:

```
# cat sfdr_enable
List of fq ids with: High-priority SFDRs :enabled(1)
Total FQD with: High-priority SFDRs : enabled = 0
Total FQD with: High-priority SFDRs : disabled = 32767
```

### 28.10.3.18 sys/kernel/debug/qman/fqd/state_[active | oos | parked | retired | tentatively_sched | truly_sched]

Description:

Query Frame Queue State in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which are in the specified state: active, oos, parked, retired, tentatively scheduled or truly scheduled.

For example, the following returns all the frame queues in the Tentatively Scheduled state:

```
# cat state_tentatively_sched
List of fq ids in state: Tentatively Scheduled
0x0001ca,0x0001cb,0x0001cc,0x0001cd,0x0001ce,0x0001cf,0x0001d0,0x0001d1,
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001d6,0x0001d7,0x0001d8,0x0001d9,
...
Out of Service count = 32201
Retired count = 0
Tentatively Scheduled count = 566
Truly Scheduled count = 0
Parked count = 0
Active, Active Held or Held Suspended count = 0
```

### 28.10.3.19 /sys/kernel/debug/qman/fqd/tde_[enable | disable]

Description:

Query Tail_Drop_Enable bit in all frame queue descriptors. This is a read only file. The output of this file returns all the frame queue ids, in a comma seperated list, which have their Tail_Drop_Enable bit mask enabled or disabled.

For example, the following returns all the frame queues with Tail_Drop_Enable bit enabled:

```
# cat tde_enable
List of fq ids with: Tail-Drop Enable :enabled(1)
0x0001ca,0x0001cb,0x0001cc,0x0001cd,0x0001ce,0x0001cf,0x0001d0,0x0001d1,
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001d6,0x0001d7,0x0001d8,0x0001d9,
...
Total FQD with: Tail-Drop Enable : enabled = 560
Total FQD with: Tail-Drop Enable : disabled = 32207
```

### 28.10.3.20 /sys/kernel/debug/qman/fqd/wq

Description:

Query Destination Work Queue in all frame queue descriptors. This file takes as input work queue id combined with channel id (destination work queue). The output of this file returns all the frame queues with destination work queue number as specified in the input.

For example, the following returns all the frame queues with their destination work queue number equal to 0x10f:

```
# echo 0x10f > wq
# cat wq
List of fq ids with destination work queue id = 0x10f
0x0001d2,0x0001d3,0x0001d4,0x0001d5,0x0001de,0x0001df,0x0001e0,0x0001e1,
```

```
0x0001ea,0x0001eb,0x0001ec,0x0001ed,0x0001f6,0x0001f7,0x0001f8,0x0001f9,
0x0001fa,0x0001fb,0x0001fd,0x0001fe
Summary of all FQD destination work queue values
Channel: 0x0 WQ: 0x0 WQ_ID: 0x0, count = 32199
Channel: 0x0 WQ: 0x0 WQ_ID: 0x4, count = 1
Channel: 0x0 WQ: 0x3 WQ_ID: 0x7, count = 64
Channel: 0x1 WQ: 0x3 WQ_ID: 0xf, count = 64
Channel: 0x2 WQ: 0x3 WQ_ID: 0x17, count = 64
Channel: 0x3 WQ: 0x3 WQ_ID: 0x1f, count = 64
Channel: 0x4 WQ: 0x3 WQ_ID: 0x27, count = 64
Channel: 0x5 WQ: 0x3 WQ_ID: 0x2f, count = 64
Channel: 0x6 WQ: 0x3 WQ_ID: 0x37, count = 64
Channel: 0x7 WQ: 0x3 WQ_ID: 0x3f, count = 64
Channel: 0x21 WQ: 0x3 WQ_ID: 0x10f, count = 20
Channel: 0x42 WQ: 0x3 WQ_ID: 0x217, count = 8
Channel: 0x45 WQ: 0x0 WQ_ID: 0x228, count = 1
Channel: 0x60 WQ: 0x3 WQ_ID: 0x307, count = 8
Channel: 0x61 WQ: 0x3 WQ_ID: 0x30f, count = 8
Sysfs and Debugfs QMan/BMan interfaces
QMan, BMan API RM, Rev. 0.13
Freescale Semiconductor Freescale 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
```

## 28.10.3.21 /sys/kernel/debug/qman/fqd/summary

Description:

Provides a summary of all the fields in all frame queue descriptors. This is a read only file.

```
# cat summary
Out of Service count = 32201
Retired count = 0
Tentatively Scheduled count = 566
Truly Scheduled count = 0
Parked count = 0
Active, Active Held or Held Suspended count = 0
-----------------------------------------
Prefer in cache count = 560
Hold active in portal count = 0
Avoid Blocking count = 528
High-priority SFDRs count = 0
CPC Stash Enable count = 0
Context-A stashing count = 528
ORP Enable count = 0
Tail-Drop Enable count = 560
```

## 28.10.3.22 /sys/kernel/debug/qman/ccsrmempeek

Description:

Provides access to Queue Manager ccsr memory map. This file takes as input an offset from the QMan CCSR base address. The output of this file returns the 32-bit value of the memory address as specified in the input.

For example, to query the QM IP Block Revision 1 register (which is at offset 0xbf8 from the QMan CCSR base address), we would do the following:

```
# echo 0xbf8 > ccsrmempeek
# cat cccsrmempeek
QMan register offset = 0xbf8
value = 0x0a010101
```

#### 28.10.3.23 /sys/kernel/debug/qman/query_ceetm_xsfdr_in_use

Description:

Query the number of XSFDRs currently in use by the CEETM logic of the DCP portal. This file takes input as the DCP id. The output of the file returns the number of XSFDR in use. Please note this feature is only available in T4/B4 rev2 silicon.

For example, if we want to query XSFDR in use number of DCP 0, we would do the following:

# echo 0 > query_ceetm_xsfdr_in_use

# cat query_ceetm_xsfdr_in_use

DCP0: CEETM_XSFDR_IN_USE number is 0

### 28.10.4 BMan debugfs

#### 28.10.4.1 /sys/kernel/debug/bman

Description:

This directory contains various BMan device debugging attributes.

#### 28.10.4.2 /sys/kernel/debug/bman/query_bp_state

Description:

This file requests a snapshot of the availability and depletion state of each of BMan's buffer pools. This is a read only file.

For example, if we want to perform a query we could use this file in the following manner:

# cat query_bp_state

bp_id free_buffers_avail bp_depleted

0 yes no

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

## 28.11 Error handling and reporting

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

### 28.11.1 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).

## 28.12 Operating system specifics

This chapter captures O/S-specific issues and distinctions, as the rest of the document essentially describes the interfaces in a generalized manner.

## 28.12.1 Portal maintenance

By default, the Linux kernel initializes QMan and BMan portals to perform all processing via interrupt-handling. As such there are no persistent threads or polling requirements in order to use portals in the Linux kernel.

Whereas for USDPAA (linux user space), the default is for all processing to be driven by polling, and support for the use of interrupts is disabled. The applications are required to call qman_poll() and bman_poll() within their run-to-completion loops to ensure that portal processing occurs regularly.

As described in #unique_479 (for BMan) and #unique_480 (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.

## 28.12.2 Callback context

In the Linux kernel, all interrupt-driven portal duties are handled in interrupt context, whereas all other portal duties are invoked from within the qman_poll() and bman_poll() functions, which are invoked by the application.

In USDPAA, even interrupt-driven portal duties are handled in an application context. Interrupts are handled within the kernel and locally disabled, and the presence of such interrupt events is available to the application via the USDPAA file-descriptor representing the portal devices. See Interrupt handling on page 447 for more information. Interrupt-driven portal duties are thus processed when the application calls the qman_thread_irq() and bman_thread_irq() functions, and other portal duties are processed when the application calls qman_poll() and bman_poll().

## 28.12.3 Blocking semantics

Many high-level QMan and BMan API functions provide "WAIT" flags, to allow the API to block as part of its operation.

In the Linux kernel, "WAIT" behavior is implemented by allowing the calling thread to sleep until a given condition is satisfied. The limitation then to using "WAIT" flags is that the caller can not be in atomic context - i. e. not executing within an interrupt handler, tasklet, bottom-half, etc, nor with any spinlocks held. One consequence is that "WAIT" flags can not be used within a callback.

On run-to-completion systems such as USDPAA, "WAIT" behavior is unsupported and unavailable.

QuadSPI Driver User Manual

# Chapter 29
# QuadSPI Driver User Manual

## 29.1 QuadSPI Driver User Manual

### Specifications

Linux-3.12.0.x + u-boot-2014.01

### 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 |

QuadSPI Driver User Manual

### 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.
```

QUICC Engine UCC UART User Manual

# Chapter 30
# QUICC Engine UCC UART User Manual

## 30.1 QUICC Engine UCC UART User Manual

### Description

QE UCC can run UART protocol. P1025RDB supports two RS485 ports via UCC3 and UCC7.

### Dependencies

| | |
|---|---|
| Hardware: | Freescale P1025RDB-PA board and P1025E |
| Software: | Linux 2.6.35 & U-boot v2011.03-dev |

### 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 P1025RDB_config*

before doing the actually build

#### Runtime options

| Env Variable | Env Description | Sub option | Option Description |
|---|---|---|---|
| hwconfig | Hardware configuration for u-boot | qe | Eanble the QE node in DTB. |
| bootargs | Kernel command line argument passed to kernel | | |

### 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 --->`<br>`    Character devices --->` | UCC UART driver |

QUICC Engine UCC UART User Manual

| Kernel Configure Tree View Options | Description |
|---|---|
| `Serial drivers --->`<br>`<*> Freescale QUICC Engine serial port 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_QE | y/m/n | n | UCC UART 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 |
| ucc | serial | enable | Ucc Uart |

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

```
serial2: ucc@2600 {
 device_type = "serial";
 compatible = "ucc_uart";
 status = "disabled";
 reg = <0x2600 0x200>;
 cell-index = <7>;
 port-number = <0>;
 rx-clock-name = "brg6";
 tx-clock-name = "brg6";
 interrupts = <42>;
 interrupt-parent = <&qeic>;
 pio-handle = <&pio3>;
 };
```

### Source Files

The following source file are related the this feature in u-boot.

QUICC Engine UCC UART User Manual

| Source File | Description |
| --- | --- |
| drivers/serial/ucc_uart.c | UCC UART driver file |

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

| Source File | Description |
| --- | --- |
| | |

### 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 uboot startup,set "qe" parameter in hwconfig.
2. After bootup kernel, create the device node manually:

```
QE serial port1: mknod /dev/ttyQE0 c 204 46
QE serial port2: mknod /dev/ttyQE1 c 204 47
```

3. Type the command: getty 115200 /dev/ttyQE0
4. Connect one console cable(RS485 to RS232)to PC COM2 and open a terminal binding with it, and

```
Set the terminal baud rate to 115200.
```

5. You can type any shell command in UCC UART TERMINAL.

### Benchmarking

N/A

### Known Bugs, Limitations, or Technical Issues

N/A

### Supporting Documentation

N/A

QUICC Engine UCC UART User Manual

QUICC Engine Time Division Multiplexing User Manual

# Chapter 31 QUICC Engine Time Division Multiplexing User Manual

## 31.1 QUICC Engine Time Division Multiplexing User Manual

### Linux SDK for QorIQ Processors

#### Description

The Time Division Multiplexing (TDM) driver is implemented by UCC and TSA. It enables UCC1/3/5/7 to work in transparent protocol, connected with pq-mds-t1 card to support T1/E1 function. It can work in normal or loopback mode both for tdm controller and phy.

Based on P1025RDB/P1021RDB and pq-mds-t1 card, connect zarlink le71hr8820g card to pq-mds-t1 card, it can do phone call in full duplex mode.

Based on T1040RDB/LS1021atwr and X-TDM-DS26522 card, connect X-TDM-DS26522 card to TDM Riser slot, it can do phone call in full duplex mode.

#### Dependencies

P1025RDB/P1021RDB and pq-mds-t1 card:

| | |
|---|---|
| Hardware: | Freescale P1025RDB-PA or P1021RDB-PC board |
| Software: | Linux 3.12.19 & U-boot v2014.07 |

T1040RDB/LS1021atwr and X-TDM-DS26522 card:

| | |
|---|---|
| Hardware: | Freescale T1040RDB or LS1021atwr board |
| Software: | Linux 3.0.37 & U-boot v2014.07 |

#### 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 P1025RDB_config or make P1021RDB-PC_config*

*make T1040RDB_config or make ls1021atwr_nor_config*

before doing the actually build

#### Runtime options

QUICC Engine Time Division Multiplexing User Manual

| Env Variable | Env Description | Sub option | Option Description |
|---|---|---|---|
| hwconfig | Hardware configuration for u-boot | qe;tdm | QUICC Engine TDM enabled in DTB |
| bootargs | Kernel command line argument passed to kernel | | |

Kernel Configure Options

Tree View

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

P1025RDB/P1021RDB and pq-mds-t1 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
        [*]   QE GPIO support
    [*] SPI support   --->
          <*>  Freescale MPC8xxx SPI controller
    [*] TDM support  --->
        TDM Device support  --->
            <*> UCC TDM driver for Freescale QE engine
        Line Control Devices  --->
             <*>  Zarlink Slic intialization Module
        TDM test
             <M>   UCC TDM test Module
</pre></td><td>Enable the QE TDM driver and pq-mds-t1 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 driver |
| CONFIG_PQ_MDS_T1 | y/n | n | Enable PQ-MDS-T1 card support |
| CONFIG_SLIC_ZARLINK | y/m/n | n | Enable Zarlink slic card support |
| CONFIG_SPI_MPC8xxx | y/m/n | n | Enable QE SPI driver |
| CONFIG_UCC_TDM_TEST | y/m/n | n | Enable UCC TDM test module |

T1040RDB/LS1021atwr and X-TDM-DS26522 card:

QUICC Engine Time Division Multiplexing User Manual

<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
    [*] TDM support  --->
        TDM Device support  --->
            <*> UCC TDM driver for Freescale QE engine
        Line Control Devices  --->
             <*>  SLIC MAXIM CARD SUPPORT
        TDM test
             <M>   UCC TDM test Module
</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 driver |
| CONFIG_SLIC_MAXIM | y/m/n | n | Enable x-tdm-ds26522 card support |
| CONFIG_UCC_TDM_TEST | y/m/n | n | Enable UCC TDM test module |

### 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 | tdm | enable | QE UCC TDM 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.

P1025RDB/P1021RDB and pq-mds-t1 card:

```
tdma: ucc@2000 {
         compatible = "fsl,ucc-tdm";
         rx-clock-name = "clk3";
         tx-clock-name = "clk4";
         rx-sync-clock = "rsync_pin";
         tx-sync-clock = "tsync_pin";
         tx-timeslot = <0x00fffffe>;
         rx-timeslot = <0x00fffffe>;
         pio-handle = <&pio_tdma>;
         tdm-framer-type = "e1";
         tdm-mode = "normal";
         fsl,tdm-id = <0>;
```

QUICC Engine Time Division Multiplexing User Manual

```
          fsl,siram-entry-id = <0>;
          phy-handle = <&pq_mds_t1>
};
```

T1040RDB/LS1021atwr and X-TDM-DS26522 card:

```
tdma: ucc@2000 {
          compatible = "fsl,ucc-tdm";
          rx-clock-name = "clk8";
          tx-clock-name = "clk9";
          rx-sync-clock = "rsync_pin";
          tx-sync-clock = "tsync_pin";
          tx-timeslot = <0x00fffffe>;
          rx-timeslot = <0x00fffffe>;
          tdm-framer-type = "e1";
          tdm-mode = "normal";
          fsl,tdm-id = <0>;
          fsl,siram-entry-id = <0>;
};
```

### Source Files

The following source file are related the this feature in Linux.

P1025RDB/P1021RDB and pq-mds-t1 card:

| Source File | Description |
|---|---|
| drivers/tdm/device/fsl_ucc_tdm.c | QE UCC TDM driver. |
| drivers/tdm/device/fsl_ucc_tdm.h | QE UCC TDM driver head file. |
| drivers/tdm/line_ctrl/slic_zarlink.c | Zarlink card driver. |
| drivers/tdm/line_ctrl/pq_mds_t1.c | PQ-MDS-T1 card driver |
| Drivers/tdm/test/ucc_tdm_test.c | QE UCC TDM test module. |
| arch/powerpc/boot/dts/p1025rdb_32b.dts | Define the device tree nodes for P1025 TDM |
| arch/powerpc/boot/dts/p1021rdb-pc_32b.dts | Define the device tree nodes for P1021 TDM |

T1040RDB/LS1021atwr and X-TDM-DS26522 card:

| Source File | Description |
|---|---|
| drivers/tdm/device/fsl_ucc_tdm.c | QE UCC TDM driver. |
| drivers/tdm/device/fsl_ucc_tdm.h | QE UCC TDM driver head file. |
| drivers/tdm/line_ctrl/slic_maxim.c | X-TDM-DS26522 card driver. |
| Drivers/tdm/test/ucc_tdm_test.c | QE UCC TDM test module. |
| arch/powerpc/boot/dts/t1040rdb.dts | Define the device tree nodes for T1040RDB TDM |
| arch/powerpc/boot/dts/ls1021a-twr.dts | Define the device tree nodes for ls1021a-twr TDM |

QUICC Engine Time Division Multiplexing User Manual

### 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;tdm" parameter in hwconfig.
2. After bootup kernel, Kernel boot log for tdm:

```
talitos ffe30000.crypto: sha384-talitos
talitos ffe30000.crypto: sha512-talitos
adapter [tdm_ucc_1] registered
adapter [tdm_ucc_3] registered
adapter [tdm_ucc_5] registered
adapter [tdm_ucc_7] registered
pq_mds_t1 ff980000.tdmphy: registed pq-mds-t1 card
IPv4 over IPv4 tunneling driver
GRE over IPv4 tunneling driver
```

3. QE TDM T1/E1 test
   a. Make sure pq-mds-t1 card connected to P1025RDB/P1021RDB board PMC Slot.
   b. To test tdm external ports, please plugin tdm t1/e1 loopback cable in the related port.

      The following is TDM port mapping with PQ-MDS-T1 card:

```
TDM Port                        PQ-MDS-T1 Port
Port A                          CH1;
Port B                          CH4;
Port C                          CH5;
Port D                          CH8.
```

   c. TDM test using E1.

      Use the default dts to test E1 function. Test module can receive ucc_num as parameter. This number should be 1/3/5/7 related to the tdm port. If no parameter input, the test module will use the defaut ucc_num=1.

```
[root@p1025rdb]# insmod ucc_tdm_test.ko
TDM_TEST: Test Module for Freescale Platforms with TDM support
TDM Driver(ID=1)is attached with Adaptertdm_ucc_1(ID = 0) drv_count=1
TDM_TEST module installed
[root@p1021rdb]# 0 1       2       3
4       5       6       7
8       9       a       b
c       d       e       f
10      11      12      13
```

QUICC Engine Time Division Multiplexing User Manual

```
14      15      16      17
18      19      1a      1b
......
6c      6d      6e      6f
70      71      72      73
74      75      76      77
78      79      7a      7b
7c      7d      7e      7f
```

d. TDM test using T1

   In the default dts file, set tdm-framer-type = "t1" for all four ports dts node; And set line-rate = "t1" in tdmphy dts node. Rebuild dts file, and the other steps is same with E1 test.

4. Zarlink card phone call test

   a. Connect Zarlink Le71hr8820G card to PQ-MDS-T1 card socket P14. There are 2 RJ11 phone cable interfaces (SX1/2 – LINE 1/2) on Zarlink tdm voice card. Connect two analog phone sets with these two ports via phone cable.

   b. In the default dts file, set pld:fsl,card-support = "zarlink,le71hr8820g".set tdma:tx-timeslot = <0x000000f0> and tdma:rx-timeslot = <0x000000f0>. Rebuild dts file.

   c. Boot kernel with ucc tdm controller driver and pq-mds-t1 card driver build-in. Insmod zarlink slic card driver module and ucc tdm test module with input parameter fun_num=1.

```

[root@p1025rdb]# insmod slic_zarlink.ko
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 config success
The data read 1 is ff
The data read 2 is 4
The product code read is bc
The device configuration reg 1 is  ff
The device configuration reg 2 is  8a
DEV reg is 82
DEV reg after is 2
Mask reg before setting is 3f bf
Mask reg after setting is ff ff
Read Tx Timeslot for CH1 is 6
Read Tx Timeslot for CH2 is 4
Read Rx Timeslot for CH1 is 4
Read Rx Timeslot for CH2 is 6
Operating Fun for channel 1 is 86
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 21
Operating Fun for channel 2 is a1
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 21
SLIC 1 configuration success
[root@p1025rdb]# insmod ucc_tdm_test.ko fun_num=1
TDM_TEST: Test Module for Freescale Platforms with TDM support
```

QUICC Engine Time Division Multiplexing User Manual

```
TDM Driver(ID=1)is attached with Adaptertdm_ucc_1(ID = 0) drv_count=1
TDM_TEST module installed
root@p2020ds:~#
```

d. Pick up two phone receivers, and talking on the phone.

### Benchmarking

N/A

### Known Bugs, Limitations, or Technical Issues

N/A

### Supporting Documentation

N/A

QUICC Engine Time Division Multiplexing User Manual

Freescale Native SATA Driver User Manual

# Chapter 32
# Freescale Native SATA Driver User Manual

## 32.1 Freescale Native SATA Driver User Manual

### Description

The driver supports Freescale native SATA controller. There are two types of SATA controller. One is PowerPC-based and the other is ARM-based. There is slight difference between them. This manual will take P5020DS board as example for description.

### Specifications

| | |
|---|---|
| Target board: | Freescale MDS/DS/RDB/QDS/TWR board |
| CPU: | Freescale processors |
| Software: | Linux-3.0.x + u-boot-2011.12 |

### Module Loading

SATA driver supports either kernel built-in or module.

### Kernel Configure Tree View Options

1) For PowerPC-based Socs, like P5020, P5040 etc.

| Kernel Configure Tree View Options | Description |
|---|---|
| `Device Drivers--->`<br>` <*> Serial ATA and Parallel ATA drivers  --->`<br>`--- Serial ATA and Parallel ATA drivers`<br>`<*>   Freescale 3.0Gbps SATA`<br>`support` | Enables SATA controller support on PowerPC-based SoCs |

2) For ARM-based Socs, like LS1021, LS2085 etc.

| Kernel Configure Tree View Options | Description |
|---|---|
| `Device Drivers--->`<br>` <*> Serial ATA and Parallel ATA drivers  --->`<br>`--- Serial ATA and Parallel ATA drivers`<br>`<*>   AHCI SATA support`<br>`<*>   Platform AHCI SATA support` | Enables SATA controller support on ARM-based SoCs |

Freescale Native SATA Driver User Manual

### Compile-time Configuration Options

1) For PowerPC-based Socs, like P5020, P5040 etc.

| Option | Values | Default Value | Description |
|---|---|---|---|
| CONFIG_SATA_FSL=y | y/m/n | y | Enables SATA controller |

2) For ARM-based Socs, like LS1021, LS2085 etc.

| Option | Values | Default Value | Description |
|---|---|---|---|
| CONFIG_SATA_AHCI=y | y/m/n | y | Enables SATA controller |
| CONFIG_SATA_AHCI_PLATFORM=y | y/m/n | y | Enables SATA controller |

### Source Files

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

1) For PowerPC-based Socs, like P5020, P5040 etc.

| Source File | Description |
|---|---|
| drivers/ata/sata-fsl.c | SATA controller driver |

2) For ARM-based Socs, like LS1021, LS2085 etc.

| Source File | Description |
|---|---|
| drivers/ata/ahci_platform.c | Platform AHCI SATA support |

### P5020DS Test Procedure

```
Please follow the following steps to use USB in Simics
(1) Boot up the kernel
...
fsl-sata ffe18000.sata: Sata FSL Platform/CSB Driver init
scsi0 : sata_fsl
ata1: SATA max UDMA/133 irq 74
fsl-sata ffe19000.sata: Sata FSL Platform/CSB Driver init
scsi1 : sata_fsl
ata2: SATA max UDMA/133 irq 41
...
(2) The disk will be found by kernel.
...
ata1: Signature Update detected @ 504 msecs
ata2: No Device OR PHYRDY change,Hstatus = 0xa0000000
ata2: SATA link down (SStatus 0 SControl 300)
ata1: SATA link up 1.5 Gbps (SStatus 113 SControl 300)
ata1.00: ATA-8: WDC WD1600AAJS-22WAA0, 58.01D58, max UDMA/133
ata1.00: 312581808 sectors, multi 0: LBA48 NCQ (depth 16/32)
ata1.00: configured for UDMA/133
scsi 0:0:0:0: Direct-Access     ATA      WDC WD1600AAJS-2 58.0 PQ: 0 ANSI: 5
sd 0:0:0:0: [sda] 312581808 512-byte logical blocks: (160 GB/149 GiB)
sd 0:0:0:0: Attached scsi generic sg0 type 0
sd 0:0:0:0: [sda] Write Protect is off
```

Freescale Native SATA Driver User Manual

```
sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't support DPO
 or FUA
 sda: sda1 sda2 sda3 sda4 < sda5 sda6 >
sd 0:0:0:0: [sda] Attached SCSI disk

(3)play with the disk according to the following log.
[root@p5020 root]# fdisk -l /dev/sda
Disk /dev/sda: 160.0 GB, 160041885696 bytes
255 heads, 63 sectors/track, 19457 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

    Device Boot      Start         End      Blocks  Id System
/dev/sda1               1         237     1903671  83 Linux
/dev/sda2             238         480     1951897+ 82 Linux swap
/dev/sda3             481        9852    75280590  83 Linux
/dev/sda4            9853       19457    77152162+  f Win95 Ext'd (LBA)
/dev/sda5            9853       14655    38580066  83 Linux
/dev/sda6           14656       19457    38572033+ 83 Linux
[root@p5020 root]#
[root@p5020 root]# mke2fs /dev/sda1
mke2fs 1.41.4 (27-Jan-2009)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
65280 inodes, 261048 blocks
13052 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=268435456
8 block groups
32768 blocks per group, 32768 fragments per group
8160 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 22 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
[root@p5020 root]#
[root@p5020 root]# mkdir sata
[root@p5020 root]# mount /dev/sda1 sata
[root@p5020 root]# ls sata/
lost+found
[root@p5020 root]# cp /bin/busybox sata/
[root@p5020 root]# umount sata/
[root@p5020 root]# mount /dev/sda1 sata/
[root@p5020 root]# ls sata/
busybox     lost+found
[root@p5020 root]# umount sata/
[root@p5020 root]# mount /dev/sda3 /mnt
[root@p5020 root]# df
Filesystem           1K-blocks      Used Available Use% Mounted on
rootfs               852019676 794801552  13937948  99% /
/dev/root            852019676 794801552  13937948  99% /
tmpfs                  1036480        52   1036428   1% /dev
shm                    1036480         0   1036480   0% /dev/shm
```

Freescale Native SATA Driver User Manual

```
/dev/sda3              74098076   4033092  66300956   6% /mnt
```

### Known Bugs, Limitations, or Technical Issues

1) For PowerPC-based Socs, like P5020, P5040 etc.

- The best value of RX_WATER_MARK for good performance is 0x16, but it is set to 0x10 in driver since some disks cannot work with higher value. The value can be changed at run time like below:

```
echo 22 > /sys/devices/ffe000000.soc/ffe220000.sata/rx_watermark

22 is 0x16, ffe220000 is the register base of first SATA controller (ffe221000 is the
second SATA controller),
after changing it, use below instruction to check:

cat /sys/devices/ffe000000.soc/ffe220000.sata/rx_watermark
```

- The SATA controller has only 32-bit DMA access ability, it cannot access memory space above 4G if there is more than 4G memory in system, then kernel will enable SWIOTLB (which is also know as bounce buffer) to do an extra copy, this will cause performance degradation. So if there is more than 4G memory and need a good performance for SATA, set 'mem=4G' in U-boot bootargs, this will limit the system to use only 4G memory.
- P5040DS board has issue to support Gen1(1.5Gbps) hard drive, use Gen2(3Gbps) hard drive on P5040DS.

2) For ARM-based Socs, like LS1021, LS2085 etc.

- CDROM is not supported due to the silicon limitation
- Gen3 is not supported on LS102X platforms.
- Gen3 SSD is not supported on LS208x platforms.

### Supporting Documentation

- N/A

# Chapter 33
# Security Engine (SEC)

## 33.1 SEC Device Driver User Manual

### Introduction and Terminology

The linux kernel contains two built-in Scatterlist CryptoAPI drivers for the Freescale SEC v2.x, v3.x, v4.x, and v5.x security h/w blocks.

They integrate seamlessly with in-kernel crypto users, such as IPSec, such that any IPSec suite that configures IPSec tunnels with the kernel will automatically use the h/w to do the crypto.

SEC v3.x is backward compatible with SEC v2.x hardware, so one can assume that subsequent SEC v2.x references include SEC v3.x hardware, unless explicity mentioned otherwise.

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 explicity mentioned otherwise.

The name of the s/w driver module for SEC v2.x h/w is 'talitos', after its internal block name.

The name of the s/w driver module for SEC v4.x h/w is 'caam', after its internal block name: Cryptographic Accelerator and Assurance Module.

Table 22:

| SEC h/w version | driver name |
|---|---|
| v2.x | talitos |
| v3.x | talitos |
| v4.x | caam |
| v5.x | caam |

### Module Loading

Talitos Freescale Security Engine device drivers support either kernel built-in or module.

### Kernel Configuration

The designated driver should be configured in the kernel by default for the target platform. If unsure, check CONFIG_CRYPTO_DEV_TALITOS (SEC v2.x) and/or CONFIG_CRYPTO_DEV_FSL_CAAM (SEC v4.x) are set under "Cryptographic API" -> "Hardware crypto devices" in the kernel configuration. The following shows kernel configuration to support the talitos device driver:

| Kernel Configure Tree View Options | Description |
|---|---|
| `Cryptographic API  --->`<br>`     [*]   Hardware crypto devices  --->`<br>`     <*>   Talitos Freescale Security Engine (SEC)` | Enable talitos device driver |

*Table continues on the next page...*

*Table continued from the previous page...*

| Kernel Configure Tree View Options | Description |
|---|---|
| Network support ---><br>  Network option ---><br>    <*> PF_KEY sockets<br>    <*>   IP: AH transformation<br>    <*>   IP: ESP transformation<br>    <*>   IP: IPComp transformation<br>    <*>   IP: IPsec transport mode<br>    <*>   IP: IPsec tunnel mode | IPsec support, of course the TCP/IP networking option should be enabled |

#### Compile-time Configuration Options

| Option | Values | Default Value | Description |
|---|---|---|---|
| | y/n | y | Dependency: MPC85xx silicon family support SEC2.x/3.x |
| CRYPTO_DEV_TALITOS | y/n | y | Talitos Freescale Security Engine (SEC) |

#### CAAM Driver specifics

The CAAM driver module implements and utilizes a job ring operation interface for all crypto API service requests.

The linux kernel implementation does not include support for Data Path mode operation. However, the linux driver automatically sets the enable bit for the SEC hardware's Queue Interface (QI), depending on QI feature availability in the h/w. This enables the h/w to also operate as a DPAA component for use by e.g., USDPAA apps. This behaviour does not conflict with normal in-kernel job ring operation, other than the potential performance-observable effects of internal SEC hardware resource contention, and vice-versa.

Note the CAAM driver has sub-configuration settings, most notably hardware job ring size and interrupt coalescing. They can be used to fine-tune performance for a particular application.

The default configuration options are as follows:

```
<*>   Freescale CAAM-Multicore driver backend
```

```
(9)     Job Ring size
```

```
[ ]     Job Ring interrupt coalescing
```

```
<*>     Register algorithm implementations with the Crypto API
```

```
<*>     Register hash algorithm implementations with Crypto API
```

```
<*>     Register caam device for hwrng API
```

The first item, CRYPTO_DEV_FSL_CAAM, enables the basic Controller Driver, and the Job Queue backend. All suboptions are dependent on this.

The second item allows the user to select the size of the h/w job rings; if requests arrive at the driver enqueue entry point in a bursty nature, and the limit of size of those bursts is known, one can set the greatest burst length to save performance and memory consumption.

The third item allows the user to select the use of the hardware's interrupt coalescing feature. Note that the driver software already performs IRQ coalescing, and zero-loss benchmarks have in fact produced better results with this option turned off.

If selected, two additional options become effective:

- 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.

- CRYPTO_DEV_FSL_CAAM_INTC_TIME_THLD

  Selects the value of the completion timeout threshold in bus clocks/64, 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 last three options are always on, so as to allow the driver to register its algorithm capabilities with the kernel's crypto API. Deselect them only if you do not want crypto API requests to be performed on the SEC; they will be done in software (on the processor core).

Hash algorithms may be individually turned off, since the nature of the application may be such that it prefers software (core) crypto latency due to many small-sized requests.

Random Number Generation (RNG) may be manually turned off in case there is an alternate source of entropy available to the kernel.

### Device Tree Binding

| Property | Type | Status | Description |
|---|---|---|---|
| compatible | String | Required | fsl,secX.Y |

### Source Files

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

| Source File | Description |
|---|---|
| drivers/crypto/talitos.c | Talitos Freescale Security Engine Driver |
| drivers/crypto/caam/ | CAAM Freescale Security Engine Driver |

### Corresponding Device Tree node

```
crypto@30000 {
        compatible = "fsl,sec3.3", "fsl,sec3.1", "fsl,sec3.0",
                     "fsl,sec2.4", "fsl,sec2.2", "fsl,sec2.1",
                     "fsl,sec2.0";
        reg = <0x30000 0x10000>;
        interrupts = <45 2 0 0 58 2 0 0>;
        fsl,num-channels = <4>;
        fsl,channel-fifo-len = <24>;
        fsl,exec-units-mask = <0x97c>;
        fsl,descriptor-types-mask = <0x3a30abf>;
};
```

**NOTE**

See linux/Documentation/devicetree/bindings/crypto/fsl-sec{2,4}.txt for more info.

### How to test the driver

To test the driver, in the kernel configuration menu, under "Cryptographic API" -> "Cryptographic algorithm manager", ensure that run-time self-tests are not disabled, i.e. the "Disable run-time self tests" (CONFIG_CRYPTO_MANAGER_DISABLE_TESTS) entry is not set. This will run standard test vectors against the driver after the driver registers its supported algorithms with the kernel crypto API, usually at boot-time. Then run test on the target system. Below is a snippet extracted from the boot log.

```
....

talitos ffe30000.crypto: hwrng
alg: No test for authenc(hmac(sha1),cbc(aes)) (authenc-hmac-sha1-cbc-aes-talitos)
alg: No test for authenc(hmac(sha1),cbc(des3_ede)) (authenc-hmac-sha1-cbc-3des-talitos)
alg: No test for authenc(hmac(sha256),cbc(aes)) (authenc-hmac-sha256-cbc-aes-talitos)
alg: No test for authenc(hmac(sha256),cbc(des3_ede)) (authenc-hmac-sha256-cbc-3des-
talitos)
alg: No test for authenc(hmac(md5),cbc(aes)) (authenc-hmac-md5-cbc-aes-talitos)
alg: No test for authenc(hmac(md5),cbc(des3_ede)) (authenc-hmac-md5-cbc-3des-talitos)
talitos ffe30000.crypto: fsl,sec3.1 algorithms registered in /proc/crypto

....
```

#### Verifying driver operation and correctness

Other than noting the performance advantages due to the crypto offload, one can also ensure the h/w is doing the crypto by looking for driver messages in dmesg

The driver emits console messages at initialization time:

```
caam ffe300000.crypto: fsl,sec-v4.2 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.

#### Incrementing IRQs in /proc/interrupts

Given a time period when crypto requests are being made, the SEC h/w will fire completion notification interrupts:

```
grep <driver-name> /proc/interrupts
```

where driver-name is one of talitos or caam. 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 the algorithm being exercised is supported by the driver.

#### verifying the 'selftest' fields say 'passed' in /proc/crypto

An entry such as this:

```
name         : cbc(aes)
driver       : cbc-aes-caam
module       : kernel
priority     : 3000
refcnt       : 1
selftest     : passed
type         : ablkcipher
async        : yes
blocksize    : 16
min keysize  : 16
max keysize  : 32
ivsize       : 16
geniv        : eseqiv
```

means the driver has successfully registered support for the algorithm with the kernel crypto API.

Note that although a test vector may not exist for a particular algorithm supported by the driver, the kernel will emit messages saying which algorithms weren't tested, and mark them as 'passed' anyway.

#### Examining the h/w statistics registers in debugfs (CAAM driver only)

The controller driver enables a user level view of performance monitor registers located within the controller's register partition. To enable this view, CONFIG_DEBUG_FS must be enabled in the kernel's configuration. If there is no mount of debugfs performed at bootup time, then a manual mount must be performed in order to view these registers. This normally can be done with a superuser shell command of:

```
mount -t debugfs none /sys/kernel/debug
```

Once done, the user can read controller registers in /sys/kernel/debug/caam/ctl. It should be noted that debugfs will provide a decimal integer view of most accessible registers provided, with the exception of the KEK/TDSK/TKEK registers; those registers are long binary arrays, and should be filtered through a binary dump utility such as hexdump.

Specifically, the CAAM h/w 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 (Peformance Counter Inbound Bytes Decrypted Register)

ib_bytes_validated: - holds contents of PC_IB_VALIDATED (Performance Counter Inbound Bytes Validated Register)

ib_rq_decrypted: - holds contents of PC_IB_DEC_REQ (Performance Counter Inbound Decrypt Requests Register)

kek: - holds contents of JDKEKR (Job Descriptor Key Encryption Key Register)

ob_bytes_encrypted: - holds contents of PC_OB_ENCRYPT (Performance Counter Outbound Bytes Encrypted Register)

ob_bytes_protected: - holds contents of PC_OB_PROTECT (Performance Counter Outbound Bytes Protected Register)

ob_rq_encrypted: - holds contents of PC_OB_ENC_REQ (Performance Counter Outbound Encrypt Requests Register)

rq_dequeued: - holds contents of PC_REQ_DEQ (Performance Counter Requests Dequeued Register)

tdsk: - holds contents of TDKEKR (Trusted Descriptor Key Encryption Key Register)

tkek: - holds contents of TDSKR (Trusted Descriptor Signing Key Register)

See the hardware documentation section "Performance Counter, Fault and Version ID Registers" for more information.

For extended testing process please refer to the Linux IPsec benchmark reproducibility guide.

### Kernel configuration to support talitos device driver

### Algorithms Supported in the linux kernel scatterlist Crypto API

The linux kernel contains various users of the Scatterlist CryptoAPI, including its IPSec implementation, sometimes referred to as the NETKEY stack. The driver, after registering algorithm services with the CryptoAPI, is therefore used to process per-packet symmetric crypto requests and forward them to the SEC hardware.

Since all SEC version hardware processes requests asynchronous to the processor core, the driver registers asynchronous algorithm implementations with the crypto API: ahash, ablkcipher, and aead with CRYPTO_ALG_ASYNC set in .cra_flags.

Different combinations of h/w and driver s/w version support different sets of algorithms, so searching for the driver name in /proc/crypto on the desired target system will ensure the correct report of what algorithms it supports.

### Authenticated Encryption with Associated Data (AEAD) Algorithms

These algorithms are used in applications where the data to be encrypted overlaps, or partially overlaps, the data to be authenticated, as is the case with the IPSec protocol.

These algorithms are implemented in the driver such that the hardware makes a single pass over the input data, and both encryption and authentication data are written out simultaneously. Note that, on decryption, integrity verification is performed in h/w, when available (SEC v2.1 and above).

The AEAD algorithms are mainly for use with IPSec ESP.

At the time of writing, the CAAM driver currently supports offloading the following AEAD algorithms:

authenc(hmac(md5),cbc(aes))

authenc(hmac(sha1),cbc(aes))

authenc(hmac(sha224),cbc(aes))

authenc(hmac(sha256),cbc(aes))

authenc(hmac(sha384),cbc(aes))

authenc(hmac(sha512),cbc(aes))

authenc(hmac(md5),cbc(des3_ede))

authenc(hmac(sha1),cbc(des3_ede))

authenc(hmac(sha224),cbc(des3_ede))

authenc(hmac(sha256),cbc(des3_ede))

authenc(hmac(sha384),cbc(des3_ede))

authenc(hmac(sha512),cbc(des3_ede))

authenc(hmac(md5),cbc(des))

authenc(hmac(sha1),cbc(des))

authenc(hmac(sha224),cbc(des))

authenc(hmac(sha256),cbc(des))

authenc(hmac(sha384),cbc(des))

authenc(hmac(sha512),cbc(des))

i.e., all combinations of AES-CBC, (3)DES-EDE, with MD-5, SHA-1,-224,-256,-384, and -512.

**Cipher Encryption Algorithms**

The CAAM driver currently supports offloading the following encryption algorithms:

cbc(aes)

cbc(des3_ede)

cbc(des)

**Authentication Algorithms**

The CAAM driver's ahash support includes HMAC variants:

hmac(sha1)

hmac(sha224)

hmac(sha256)

hmac(sha384)

hmac(sha512)

hmac(md5)

sha1

sha224

sha256

sha384

sha512

md5

### Random Number Generation

Both talitos and caam drivers support random number generation services via the kernel's built-in hwrng interface when implemented in hardware. To enable:

1. verify that the h/w random device file, e.g., /dev/hwrng or /dev/hwrandom exists. If it doesn't exist, make it with:

   ```
   mknod /dev/hwrng c 10 183
   ```

2. verify /dev/hwrng doesn't block indefinitely and produces random data:

   ```
   rngtest -C 1000 < /dev/hwrng
   ```

3. verify the kernel gets entropy:

   ```
   rngtest -C 1000 < /dev/random
   ```

If it blocks, a kernel entropy supplier daemon, such as rngd, may need to be run. See linux/Documentation/hw_random.txt for more info.

### Using the driver

Once enabled, the driver will forward kernel crypto API requests to the SEC h/w for processing.

### Running IPSec

The IPSec stack built-in to the kernel (usually called NETKEY) will automatically use crypto drivers do offload the crypto to the SEC h/w. Documentation regarding how to set up an IPSec tunnel can be found in the respective open source IPSec suite packages, e.g. strongswan.org, openswan, setkey, etc.

### Running OpenSSL

While not officially supported in the SDK, there are userspace interface implementations that enable offloading OpenSSL requests to the built-in kernel crypto API, and thus the SEC h/w via its respective driver. While the kernel officially supports the AF_ALG socket interface, various third-party cryptodev implementations are also available.

Here are some links to a couple of starting points:

```
http://carnivore.it/2011/04/23/openssl_-_af_alg
http://home.gna.org/cryptodev-linux/
http://ocf-linux.sourceforge.net/
```

### Executing Custom Descriptors

Both talitos and caam have public descriptor submission interfaces, drivers/crypto/talitos.c:talitos_submit() and drivers/crypto/caam/jr.c:caam_jr_enqueue().

### talitos_submit()

*Name*

talitos_submit — submits a descriptor to the device for processing

*Synopsis*

```
int talitos_submit (struct device * dev,
 int ch,
```

```
 struct talitos_desc * desc,
 void (*callback) (struct device *dev, struct talitos_desc *desc, void *context, int
error),
 void * context);
```

*Arguments*

dev: the SEC device to be used

ch: the SEC device channel to be used

desc: the descriptor to be processed by the device

callback: whom to call when processing is complete

context: a handle for use by caller (optional)

*Description*

desc must contain valid dma-mapped (bus physical) address pointers.

callback must check err and feedback in descriptor header for device processing status.

#### caam_jr_enqueue ()

*Name*

caam_jr_enqueue — Enqueue a job descriptor head. Returns 0 if OK, -EBUSY if the queue 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.

Please refer to the source code for example usage.

### Known Bugs, Limitations or Technical Issues

Talitos XOR offloading feature removed temporarily.

### Supporting Documentation

For more information see the *Linux IPSec Benchmark Reproducibility Guide* located in the following SDK directory: sdk_documentation/pdf/Linux_IPSec_benchmark_reproducibility_guide.pdf

# Chapter 34
# User Enablement for Secure Boot - PBL Based Platforms

## 34.1 Preface

This document is the *User Enablement* document for Secure Boot. This includes description of ESBC and how to use Freescale's CST (Code Signing Tool) for creation of ESBC.

Scope: This document explains the method by which user can create ESBC image and sign it using CST.

Applicability: This document is used to identify the details of creation of ESBC image and create a digital signature over it using CST in order to enable the user to carry out above procedure at his end.

Author: A software engineer writes and updates this document.

Readers: This document is intended for the use of internal teams, including system test team who would like to use Secure Boot System or perform testing on that.

It may also be used as a reference by our customers who would like to use our ESBC or use this as reference to develop their own trusted system around ISBC.

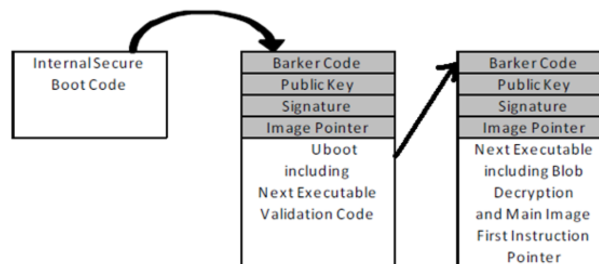
## 34.2 Introduction

### 34.2.1 Purpose

This document is intended for end-users to demonstrate the image validation process. The image validation can be split into stages, where each stage performs a specific function and validates the subsequent stage before passing control to that stage. In the example, the ESBC is Freescale provided reference code referred to as ESBC uboot.

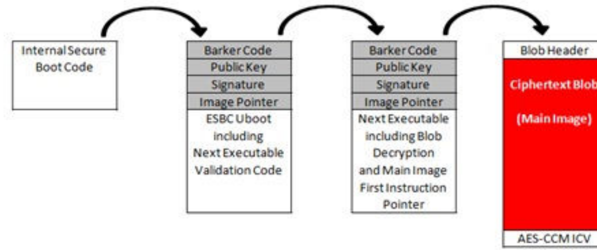
**Chain of Trust** ESBC uboot performs minimal SoC configuration before validating the Next Executable using the same CSF header format as the ISBC used to validate ESBC Uboot. The CSF Header and signature are added to the Next Executable using the Freescale Code Signing Tool.

Figure 43: Chain of Trust




**Chain of Trust with confidentiality** The validated ESBC uboot image is allowed to use the One Time Programmable Master Key to decrypt system secrets. Cryptographic blob mechanism is used to establish Chain of trust with confidentiality.

Figure 44: Chain of Trust with confidentiality




This document provides more details on the secure boot flow, ISBC, ESBC and Freescale Code signing tool.

## 34.3 Secure boot Process

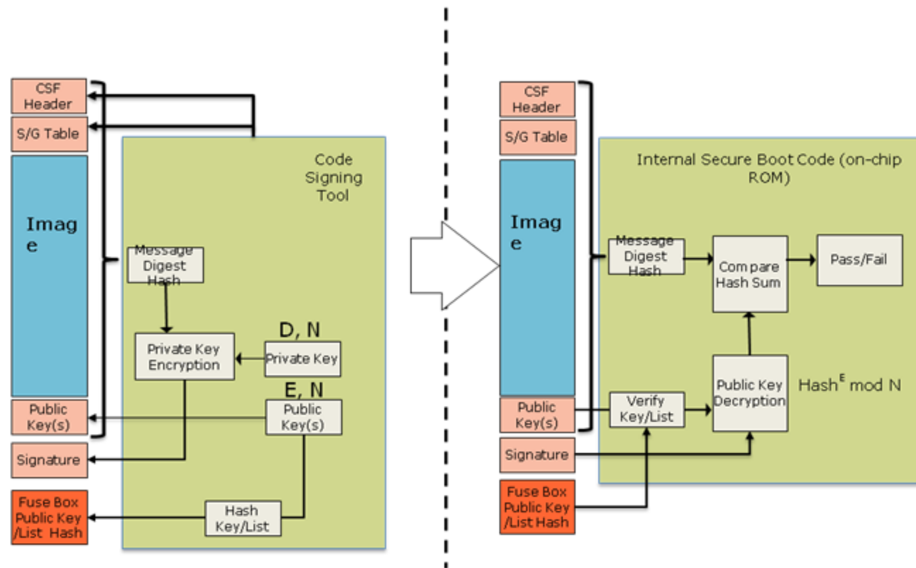
Secure boot process uses a digital signature validation routine already present in INTERNAL BOOT ROM. This routine performs validation using HW bound RSA public key to decrypt the signed hash and compare it to a freshly calculated hash over the same system image. If the comparison passes, the image can 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 45: Secure Boot Process




## 34.4 Pre-Boot Phase

When the processor is powered on, reset control logic blocks all device activity (including scan and debug activity) until fuse values can be accurately sensed. The most important fuse value at this stage of operation is the 'Intent to Secure' (ITS) bit. When an OEM sets ITS, they intend for the system to operate in a secure and trusted manner.

The two main components involved during this process are :

**The security fuse processor (SFP)** has two roles. The first is to physically burn fuses during device provisioning. The second is to use these provisioned values to enforce security policy in the pre-boot phase, and to securely pass provisioned keys and other secret values to other hardware blocks when the system is in a trusted/secure state.

**PreBoot Loader (PBL)** is the micro-sequencer that can simplify system boot by configuring the DDR memory controllers to more optimal settings and copying code and data from low speed memory into DDR. This allows subsequent phases of boot to operate at higher speed. The setting of ITS determines where the PBL is allowed to read and write. The use of the PBL is mandatory when performing secure boot. At a minimum, the PBL must read a command file from a location determined by the Reset Configuration Word (RCW) and perform a store of a value to the ESBC Pointer Register within the SoC. If the PBL doesn't perform this operation (or sets the

ESBC pointer to the wrong value), the ISBC will fail to validate the ESBC. Once the PBL has completed any operations defined by its command file, the PBL is disabled until the next Power on Reset and the Boot Phase begins.

Some example PBI commands used in the demo are given below. The commands are embedded in the RCW's mentioned in the SDK Images required for the demo

NOR SECURE BOOT

- P3/P4/P5

```
#LAW for ESBC
    09000cd0 00000000
    09138000 00000000 (Flush command)
    09000cd4 c0000000
    09138000 00000000  (Flush command)
    09000cd8 81f0001d
    09138000 00000000(FLUSH command)
# Scratch Register
    090e0200 c0b00000
```

- T1/T2/T4/B4

```
#LAW for ESBC
    09000c10 00000000
    09000c14 c0000000
    09000c18 81f0001b
# LAW for CPC/SRAM
    09000d00 00000000
    09000d04 bff00000
    09000d08 81000013
# Scratch Registers
    090e0200 c0b00000
    090e0208 c0c00000
# CPC SRAM
    09010100 00000000
    09010104 bff00009
# CPC Configuration
    09010f00 08000000
    09010000 80000000
```

NAND SECURE BOOT

- P3/P5

```
# SCRATCH REGISTER
    090e0200 bff00000
    09138000 00000000 (Flush Command)
# CPC1 SRAM
    09010000 00200400
    09010100 00000000
    09010104 bff0000b
    09010f00 08000000
    09010000 80000000
    09138000 00000000 (Flush Command)
# LAW for CPC/SRAM
    09000d00 00000000
    09000d04 bff00000
    09000d08 81000013
    09138000 00000000 (Flush Command)
```

```
# Alternate Configuration Space Configuration
    09000010 00000000
    09000014 bf000000
    09000018 81000000
    09138000 00000000 (Flush Command)
# CPC2 Cache
    09110000 80000403
    09110020 2d170008
    09110024 00100008
    09110028 00100008
    0911002c 00100008
    09138000 00000000 (Flush Command)

/* hdr_uboot.out and u-boot.bin must also be loaded on NAND
 * ALT_CONFIG_WRITE command must be used for the same.
 * Starting offset for ALT_CONFIG_WRITE command would be
 * hdr_uboot.out - 0xf00000
 * u-boot.bin    - 0xf40000
 */
```

The ISBC is capable of reading from NOR flash connected to the Local Bus, on-chip memory configured as SRAM, or main memory. Unless the ESBC is stored in NOR flash, the developer is required to create a PBL Image that copies the image to be validated from NVRAM to main memory or internal SRAM prior to writing the SCRATCHRW1 Register and executing the ISBC code.

To assist with the creation of PBL Images (for both normal and Trust systems), Freescale offers a PBL Image Tool.

Note that it is possible for an attacker to modify the board to direct the PBL to the wrong non-volatile memory interface, or change the PBL Image and CSF Header pointer, however this will result in a secure boot failure and the system remaining in an idle loop indefinitely.

## 34.5 ISBC Phase

### 34.5.1 Flow in the ISBC Code

With the PBL disabled and all external masters blocked by the PAMUs, CPU 0 is released from boot hold-off and begins executing instructions from a hardwired location within the Internal BootROM. The instructions inside the Internal BootROM are Freescale developed code known as the Internal Secure Boot Code (ISBC). The ISBC leads CPU 0 to perform the following actions:

1. **Who am I check?** - CPU 0 reads its Processor ID Register, and if it finds any value besides physical CPU 0, the CPU enters a loop. This insures that only CPU 0 executes the ISBC.
2. **Sec_Mon check** - CPU 0 confirms that the Sec_Mon is in the Check state. If not, it writes a 'fail' bit in a Sec_Mon control register, leading to a state transition.
3. **ESBC pointer read** - CPU 0 reads the ESBC Pointer Register, and then reads the word at the indicated address, which is the first word of the Command Sequence File Header which precedes the ESBC itself. If the contents of the word don't match a hard coded preamble value, the ISBC takes this to mean it has not found a valid CSF and cannot proceed. This leads to a fail, as described in #2 above.
4. **CSF parsing and public key check** - If CPU 0 finds a valid CSF header, it parses the CSF header to locate the public key to be used to validate the code. There can be a single public key or a table of 4 public keys present in the header. The Secure Fuse Processor doesn't actually store a public key, it stores a SHA-256 hash of the public key/table of 4 keys. This is done to allow support for up to 4096b keys without an excessively large fuse block. If the hash of the public key fails to match the stored hash, secure boot fails.

5. **Signature validation** - With the validated public key, CPU 0 decrypts the digital signature stored with the CSF header. The ISBC then uses the ESBC lengths and pointer fields in the CSF header to calculate a hash over the code. The ISBC checks that the CSF header is included in the address range to be hashed. Option flags in the CSF header tell the ISBC whether the Freescale Unique ID and the OEM Unique ID (in the Secure Fuse Processor) are included in the hash calculation. Including these IDs allows the image to be bound to a single platform. If the decrypted hash and generated hash don't match, secure boot fails.
6. **ESBC First Instruction Pointer check** - One final check is performed by the ISBC. This check confirms that the First Instruction Pointer in the CSF header falls within the range of the addresses included in the previous hash. If the pointer is valid, the ISBC writes a 'PASS' bit in a Sec_Mon command register, the state machine transitions to 'Trusted', and the OTPMK is made available to the SEC.
7. In case of failure , for Trust v2.0 devices , secondary flag is checked in the CSF header. If set, ISBC reads the CSF header pointer 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.

## 34.5.2 Super Root keys (SRKs) and signing keys

These are RSA public and private key pairs. Private keys are used to sign the images and public keys are used to validate the image during ISBC and ESBC phase.

Public keys are embedded in the header and the hash of srk table is fused in SRKH register of SFP.

These are Hardware Bound Keys, once the hash is fused the public private key pair can't be modified.

Keys of sizes 1k, 2k and 4k are supported in FSL Secure Boot Process.

It is the OEM's responsibility to tightly control access to the RSA private signature key. If this key is ever exposed, attackers will be able to generate alternate images that will pass secure boot.

If this key is ever lost, the OEM will be unable to update the image.

## 34.5.3 Key Revocation

Trust Architecture 2.0 introduces support for revoking the RSA public keys used by the ISBC to verify the ESBC. The RSA public keys used for this purpose are called super root keys.

OEM can use either a single key or a list of upto 4 super root keys in the Trust Arch v2.0 devices.

In the Freescale Code Signing Tool (CST), the OEM defines whether the device uses a single super root key, or offers a list of super root keys. If using a single super root key, a new flag bit in the CSF header will indicate "Key", otherwise the flag will indicate "Key List". Assuming key list, the OEM can populate a list of up to 4 super root keys for trust arch v2.0 onwards platforms . And calculates a SHA-256 hash over the list. This hash is written to the SRKH registers in the SFP.

As part of code signing, the OEM defines which key in the key list is to be used for validating the image. This key number is included as a new field in the CSF header.

During secure boot, the ISBC determines whether a key list is in use. If the key list is valid, the ISBC checks the key number indicated in the CSF header against the revocation fuses in the SFP's OEM Security Policy Register (SFP_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 a super root key is the loss of the associated RSA private key to an attacker. If the attacker has gained access to a legitimate RSA private key, and the attacker can turn on power to the fuse programming circuitry, then the attacker could maliciously revoke keys. To prevent this from being used to permanently disable the system, one super root key does not have an associated revocation fuse.

## 34.5.4 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).

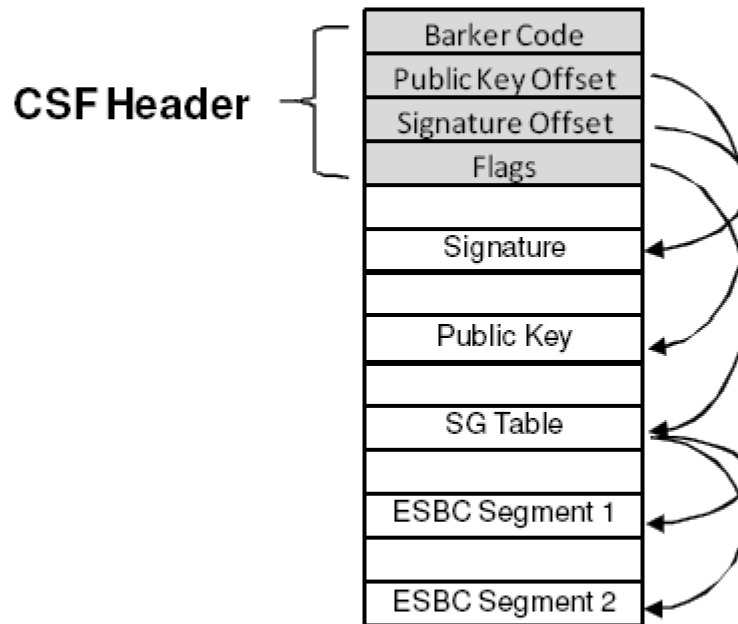
## 34.5.5 ESBC with CSF Header

ESBC is the generic name for the code that the ISBC validates. A few ESBC scenarios are described in later sections.

The figure below provides an example of an ESBC with CSF (Command Sequence File) Header. The CSF Header includes lengths and offset which allow the ISBC to locate the operands used in ESBC image validation, as well as describe the size and location of the ESBC image itself.

Note: CSF Header and ESBC Header may be used synonymously in this and other Freescale Trust Architecture documentation.

Figure 46: ESBC with CSF Header




## 34.6 ESBC Phase

Unlike the ISBC, which is in an internal ROM and therefore unchangeable, the ESBC is Freescale-supplied reference code, and can be changed by OEMs. The remainder of this section is the description of a reasonable secure boot chain of trust based on Freescale's reference software for secure boot. Depending on the requirement, ESBC can be a monolithic image - including uboot, device trees, boot firmware, drivers along with the OS and applications or can be mini-uboot.

Freescale provided ESBC consists of standard u-boot which has been signed using a private key. U-boot reserves a small space for storing environment variables. This space is typically one sector above or below the u-boot and is stored on persistent storage devices like NOR flash if macro CONFIG_ENV_IS_IN_FLASH is used. In case of secure boot, macro CONFIG_ENV_IS_NOWHERE is used and so, environment is compiled in uboot image and is called default environment. This default environment can't be stored on flash devices. User won't be able to edit this environment also as he can't reach to uboot prompt in case of secure boot. There is default boot command for secure boot in this default environment which executes on autoboot.

ESBC validates a file called boot script and on successful validation execute the commands in the boot script.

There are many reasons ESBC could fail to validate Client images or boot script. The error status message along with the code is printed on the u-boot console. For a list of error codes refer ESBC Validation Error Codes.

Users are free to use Freescale ESBC as it is provided or to use it as reference to modify their own secure boot system.

NOTE

On Soc's with ARMv8 core (eg:- LS1043), during ISBC phase in Internal Boot ROM, SMMU (which by default is in by-pass mode) is configured to allow only secure transactions from CAAM.

The security policy w.r.t. SMMU in ESBC phase must be decided by the user/customer. So, currently in ESBC (U-Boot), SMMU is configured back to by-pass mode allowing all transactions (secure as well as non-secure).

## 34.6.1 Boot script

Bootscript is a U-Boot script image which contains u-boot commands. ESBC would validate this boot script before executing commands in it.

NOTE

1. Boot script can have any commands which u-boot supports. No checking on the allowed commands in boot script. Since it is validated image, assumption is that commands in boot script would be correct.
2. If some basic scripting error done in boot script like unknown command, missing arguments, the required usage of that command and core is put in infinite loop.
3. After execution of commands in boot script, if control reaches back in u-boot, error message would be printed on u-boot console and core would be put in spin loop by command esbc_halt.
4. Scatter gather images not supported with validate command.
5. If ITS fuse is blown, any error in verification of the image would result in system reset. The error would be printed on console before system goes for a reset.

### 34.6.1.1 Where to place the boot script?

Freescale's ESBC u-boot expects the boot script to be loaded in flash as specified in Address map used for the demo on page 563. ESBC u-boot code assumes that the public/private key pair used to sign the boot script is same as that was used while signing the u-boot image. If user used different key pair to sign the image, hash of the N and E component of the key pair should be defined in macro:

CONFIG_BOOTSCRIPT_KEY_HASH.

Note - The hash defined should be hex value, 256 bits long.

Both the above macros can be defined or changed in the configuration file secure_boot.h at the following location in u-boot code:

```
u-boot/arch/powerpc//include/asm/fsl_secure_boot.h
```

Two new commands called esbc_validate and esbc_halt have been added in Freescale ESBC u-boot.

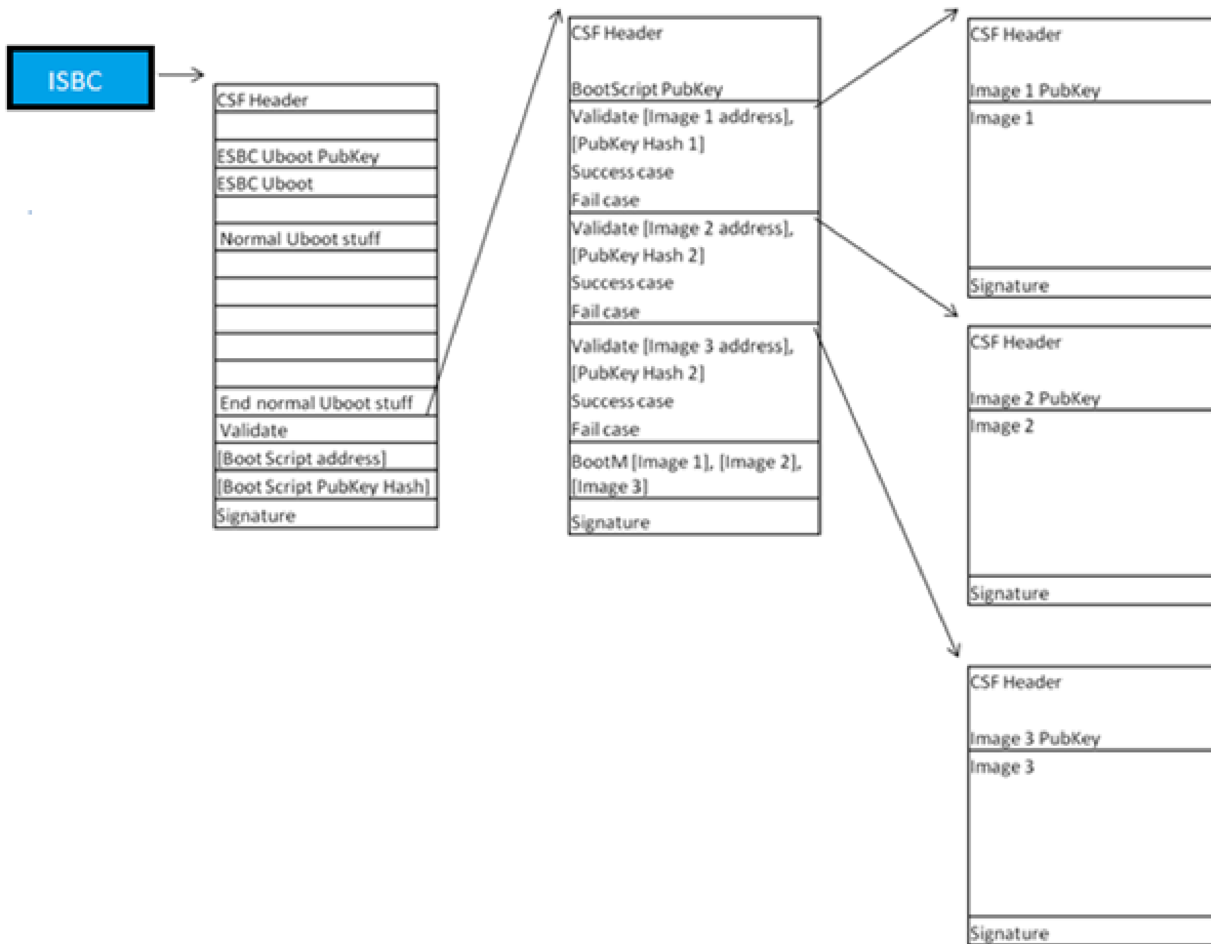
### 34.6.1.2 Chain of Trust

Boot script contains information about the next level of images, e.g. Linux, HV, etc. ESBC validates these images as per their public keys and then executes bootm command to pass-on the control to next image.

Users are free to use Freescale ESBC as it is provided or to use it as reference to modify their own secure boot system.

Figure below shows the Chain of trust established for Validation with this ESBC u-boot.

Figure 47: Secure boot flow (Chain of Trust)




## 34.6.1.2.1 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>
```

### 34.6.1.2.1.1 esbc_validate command

esbc_validate img_hdr [pub_key_hash]

**Input arguments:**

`img_hdr` - Location of CSF Header of the image to be validated

`pub_key_hash` - hash of the public key used to verify the image. This is optional parameter. If not provided, code makes the assumption that the key pair used to sign the image is same as that used with ISBC. So the hash of the key in the header is checked against the hash available in SRK fuse for verification.

**Description:**

The command would do the following:

- Perform CSF header validation on the address passed in the image header. During parsing of the header, image address in stored in an environment variable which is later used in source command in default secure boot command.
- Signature checks on the image

#### 34.6.1.2.1.2 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 uboot. If somehow, control reaches back to uboot (eg. bootm not present in bootscript), core should just spin.

### 34.6.1.3 Chain of Trust with Confidentiality

To establish chain of trust with confidentiality, cryptgraphic blob mechanism can be used. In this chain of trust, validated image is allowed to use the One Time Programmable Master Key to decrypt system secrets.

Two bootscripts are to be used. First encap bootscripts is used which creates a blob of the LINUX images and saves them. After this the system is booted after replacing the encap bootscript with decap bootscript which decapsulates the blobs and boot the LINUX with the images.

Figures below show the Chain of trust with confidentiality (Encapsulation and Decapsulation).

Figure 48: Chain of Trust with Confidentiality (Encapsulation)

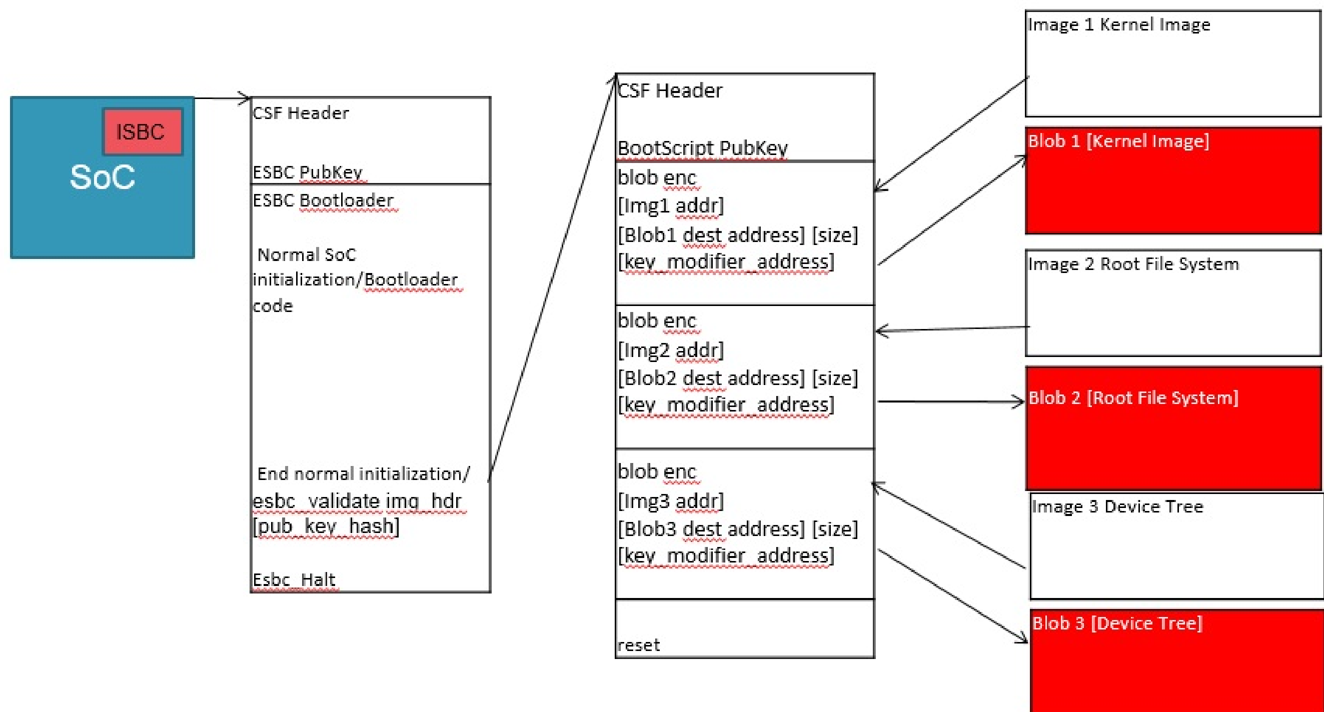
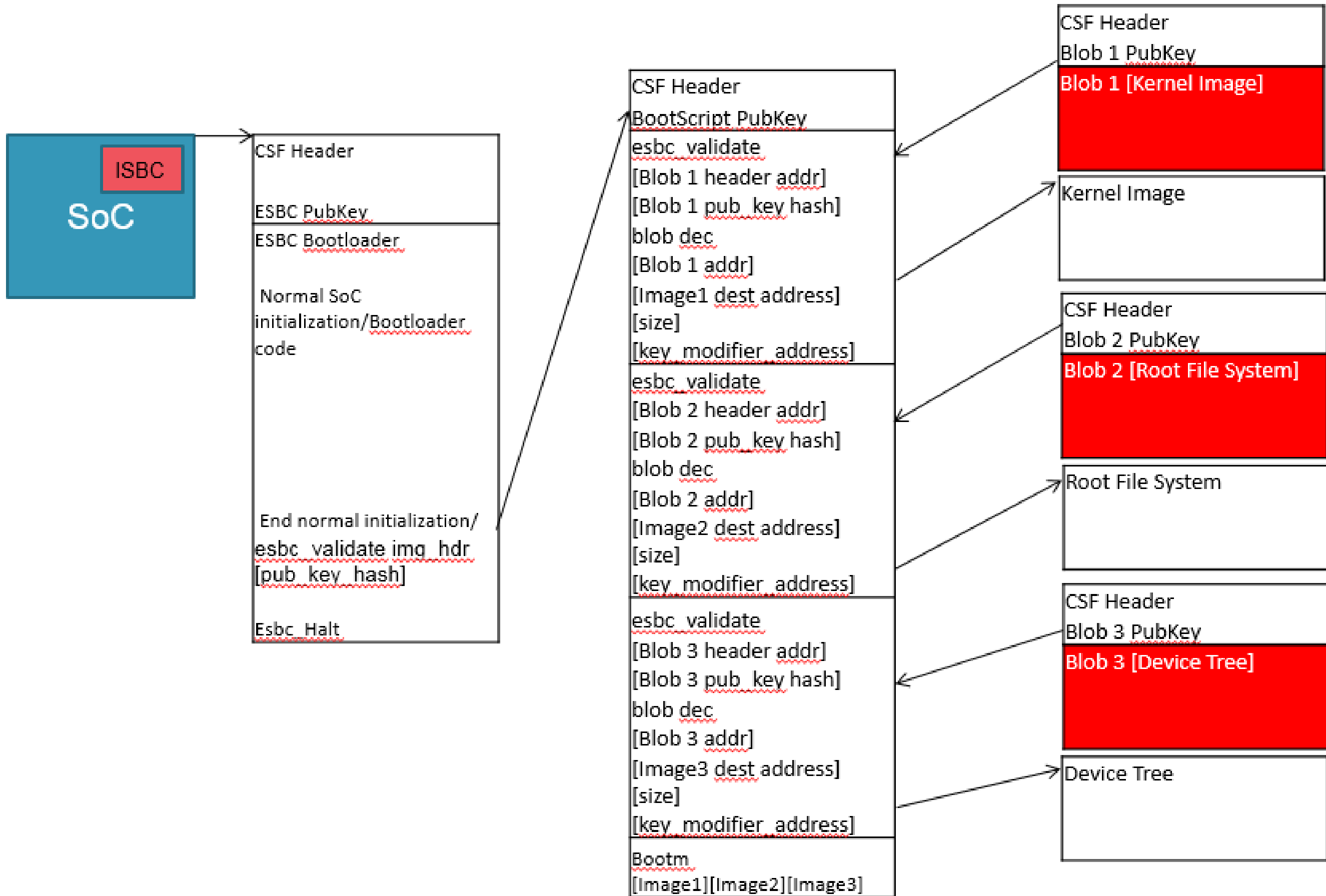


Figure 49: Chain of Trust with Confidentiality (Decapsulation)




## 34.6.1.3.1 Sample Encap Boot Script

A sample encap boot script would look like:

```
...
blob enc <Img1 addr> <Img1 dest addr> <Img1 size> <key_modifier address>
erase <encap Img1 addr> +<encap 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>

...
```

### 34.6.1.3.1.1 blob enc command

blob enc <src location> <dst location> <length> <key_modifier address>

**Input arguments:**

`src location` - Address of the image to be encapsulated

`dst location` - Address where the blob will be created

`length` - Size of the image to be encapsulated

`key_modifier address` - Address where a random number 16 bytes long(key modifier) is placed

**Description:**

The command would do the following:

- Create a cryptographic blob of the image placed at src location and place the blob at dst location.

#### 34.6.1.3.2 Sample Decap Boot Script

A sample decap boot script would look like:

```
...
blob dec <Img1 blob addr> <Img1 dest addr> <expected Img1 size> <key_modifier address>
blob dec <Img2 blob addr> <Img2 dest addr> <expected Img2 size> <key_modifier address>
blob dec <Img3 blob addr> <Img3 dest addr> <expected Img3 size> <key_modifier address>
...
bootm <Img1 dest addr> <Img2 dest addr> <Img3 dest addr>
```

##### 34.6.1.3.2.1 blob dec command

blob dec <src location> <dst location> <length> <key_modifier address>

**Input arguments:**

`src location` - Address of the image blob to be decapsulated

`dst location` - Address where the decapsulated image will be placed

`length` - Expected Size of the image after decapsulation.

`key_modifier address` - Address where key modifier (Same as that used for Encapsulation) is placed

**Description:**

The command would do the following:

- Decapsulate the blob placed at src location and place the decapsulated data of expected size at dst location.

## 34.7 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.

## 34.8 CST Tool

### 34.8.1 Code Signing Tool Walkthrough

**Step-1)**

Yocto installs the cst package at the following location:

tmp/sysroots/x86_64-linux/usr/bin/cst

OR

In the Yocto environment, the user can use below commands to rebuild cst:

1. bitbake cst-native –c cleanall
2. bitbake cst-native

**Step-2)**

```
cd tmp/sysroots/x86_64-linux/usr/bin/cst
```

gen_keys and uni_sign binaries are available in cst.

Note : LD_LIBRARY_PATH should be set to the library path in yocto workspace. <project_folder_path>/tmp/sysroots/x86_64-linux/usr/lib

**Step-3)**

Generate private key public key pair -

```
./gen_keys 1024
```

**NOTE**

- Here, 1024 refers to the size of public key Modulus in bits.
- Other allowed sizes are - 2048 bits, 4096 bits.
- See help -

```
bash-2.05a$ ./gen_keys -h
```

**Step-4)**

Put all the images (limited by number 8) you want to sign using OPENSSL RSA APIs in current directory.

**Step-5)**

Execute the binary uni_cfsign to generate signature over CSF header and ESBC images.

**CSF Header Generation**

Example taken for B4860:

```
$ ./uni_sign input_files/uni_sign/b4860/input_uboot_nor_secure

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




Key Hash :
a9bb28a23c641e13b58c19a7fc48dfd8660a29895ebbc8bd0beba432e04c0785

HEADER file hdr_uboot.out created
```

**The header would look like this:**

```
00000000  68 39 27 81 00 00 02 00  00 00 01 00 00 00 14 00  |h9'.............|
00000010  00 00 00 80 00 00 16 00  00 00 00 01 11 07 f0 00  |................|
```

```
00000020  00 00 00 01 00 00 00 01  11 11 11 11 99 99 99 99  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000200  cc fe 4d fc 20 c1 6d ba  77 42 51 c2 4d c4 5b 45  |..M. .m.wBQ.M.[E|
00000210  41 e2 88 a9 55 d0 49 7b  86 fe 5a 85 89 68 b7 db  |A...U.I{..Z..h..|
00000220  89 ef b7 2d 2a 1f 5b 74  4d 9c 7a c7 54 a9 b0 ff  |...-*.[tM.z.T...|
00000230  cf a6 1c ed 3d f3 de 8d  cc 91 ae 5f 60 b4 88 ab  |....=......_`...|
00000240  a5 70 0b 20 73 30 75 38  5b 1b 51 22 e7 2f fd a6  |.p. s0u8[.Q"./..|
00000250  65 00 07 4a 78 5d 1e ee  81 b8 a6 c4 81 e5 bc be  |e..Jx]..........|
00000260  dc 64 09 c0 07 91 7a 36  ab 7c 0c e0 ab b1 01 bb  |.d....z6.|......|
00000270  de a0 e2 56 65 0a 29 73  67 57 d3 ba 1f 52 7a 5f  |...Ve.)sgW...Rz_|
00000280  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000002f0  00 00 00 00 00 00 00 00  00 00 00 00 00 01 00 01  |................|
00000300  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001400  b3 6e 9e d5 3a 47 c6 44  4e 09 ff 29 0d a5 a1 c3  |.n..:G.DN..)....|
00001410  32 f3 b5 50 c6 42 f0 5b  59 29 f6 7d 57 0d 0a f9  |2..P.B.[Y).}W...|
00001420  22 d6 d8 68 57 85 2a e9  dd 15 18 c1 eb d3 03 d6  |"..hW.*.........|
00001430  8f 79 27 60 fa 4b 8c 1c  3e 7c db e6 3e 72 fd 8d  |.y'`.K..>|..>r..|
00001440  50 25 d9 ee 0f 30 5a 3a  cf 7e d4 3a dc 98 bc c9  |P%...0Z:.~.:....|
00001450  34 b3 8f 13 35 2e 55 1a  f5 92 98 32 71 9c 8d 5b  |4...5.U....2q..[|
00001460  8c f0 80 d2 1c 38 d5 a1  77 07 38 49 7c 7d 01 2f  |.....8..w.8I|}./|
00001470  a1 c4 08 43 f5 af 67 7f  d2 eb b9 e4 84 6c e1 77  |...C..g......l.w|
00001480  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001600  00 08 00 00 00 00 00 08  00 00 40 00 11 00 00 00  |..........@.....|
```

## 34.8.2 KEY GENERATION

### 34.8.2.1 gen_otpmk_drbg

**Introduction**

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, specified in SP800-90A. The entropy source is the Linux /dev/random.

#### 34.8.2.1.1 Features

- Generates random numbers, which can be used if usr defined string not provided, to generate OTPMK value.
- Calculates and Embed the hamming code in the hexadecimal string.

**Usage:**

```
Usage : ./gen_otpmk_drbg <bit_order> [string]
<bit_order> : (1 or 2) OTPMK Bit Ordering Scheme in SFP
    1 : BSC913x, P1010, P3, P4, P5, C29x
    2 : T1, T2, T4, B4, LSx
<string> : 32 byte string
```

In case string is not specified, the utility generates a 32 bytes random number and embeds hamming code in it.

Table 23: Options for gen_keys command

| | |
|---|---|
| bit_order | 1 or 2 only. |
| STRING | 32 BYTE Hexadecimal String. |
| -h, --help | Show this help message and exit |

**Usage Example:**

```
$ ./gen_otpmk_drbg 2 1234567890098765432 1abcd1234567890098765432 1abcd1234567887654321

OTPMK[255:0] is:
133c567810098664c329aacc123cd678910987654321abcd1234567887654321

 NAME     |       BITS      |     VALUE
__________|_______________|____________
OTPMKR 0 | 255-224     |   133c5678
OTPMKR 1 | 223-192     |   10098664
OTPMKR 2 | 191-160     |   c329aacc
OTPMKR 3 | 159-128     |   123cd678
OTPMKR 4 | 127- 96     |   91098765
OTPMKR 5 |  95- 64     |   4321abcd
OTPMKR 6 |  63- 32     |   12345678
OTPMKR 7 |  31-  0     |   87654321
```

```
$ ./gen_otpmk_drbg 1

Input string not provided
Generating a random string
-------------------------------------------
* Hash_DRBG library invoked
* Seed being taken from /dev/random
-------------------------------------------
OTPMK[255:0] is:
150665804fa4dfbea2271767b1649e64ccb35c966a3622e197c91741e73ea5cb

 NAME     |       BITS      |     VALUE
__________|_______________|____________
OTPMKR 0 |  31-  0     |   e73ea5cb
OTPMKR 1 |  63- 32     |   97c91741
OTPMKR 2 |  95- 64     |   6a3622e1
OTPMKR 3 | 127- 96     |   ccb35c96
OTPMKR 4 | 159-128     |   b1649e64
OTPMKR 5 | 191-160     |   a2271767
OTPMKR 6 | 223-192     |   4fa4dfbe
OTPMKR 7 | 255-224     |   15066580
```

## 34.8.2.2 gen_drv_drbg

**Introduction**

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, specified in SP800-90A. The entropy source is the Linux /dev/random.

### 34.8.2.2.1 Features

- Generates random numbers, which can be used if usr defined string not provided, to generate Debug Response value.
- Calculates and Embed the hamming code in the hexadecimal string.

**Usage:**

```
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
```

e.g. ./gen_drv_drbg A1 1111111122222222

In case string is not specified, the utility generates a 8 bytes random number and embeds hamming code in it.

**Table 24: Options for gen_drv command**

| Hamming_algo | A1, A2 or B for the corresponding platform. |
|---|---|
| STRING | 8 BYTE Hexadecimal String |
| -h, --help | Show this help message and exit |

**Usage Example:**

```
$ ./gen_drv_drbg A1 01aa00bbcafecafe

 NAME     |      BITS      |     VALUE
_________|______________|____________
DRV 0    |  31-   0      |    cafecbee
DRV 1    |  63-  32      |    01aa00ba
```

```
$ ./gen_drv_drbg A2

Input string not provided
Generating a random string
-------------------------------------------
* Hash_DRBG library invoked
* Seed being taken from /dev/random
-------------------------------------------
Random Key Genearted is:
5db76f03e7614e55
DRV[63:0] after Hamming Code is:
5db76f02e7614f41
 NAME     |      BITS      |     VALUE
_________|______________|____________
```

```
DRV 0     |     63 - 32    | 5db76f02
DRV 1     |     31 - 0     | e7614f41
```

### 34.8.2.3 gen_keys

**Introduction**

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.

#### 34.8.2.3.1 Features

- The application allows the user to generate 3 sizes keys. The sizes allowed are - 1024 bits, 2048 bits and 4096 bits.
- It generates RSA key pairs in PEM format.
- Keys are generated and stored in the files. User can provide filenames through command line option.

**Usage:**

./gen_keys [OPTION] SIZE

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

Sizes supported -- 1024, 2048, 4096. The generated keys would be in PEM format.

**Table 25: Options for gen_keys command**

| | |
|---|---|
| -p ,--privkey FILE | Output file where generated PRIVATE key in PEM format would be saved. (default = srk.pri) |
| -k ,--pubkey FILE | Output file where generated PUBLIC key in PEM format would be saved. (default = srk.pub) |
| -h, --help | Show this help message and exit |

**Usage Example:**

```
$ ./gen_keys 1024 -k pub_key -p priv_key

=============================================================
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 pub_key
PRIVATE KEY priv_key
```

NOTE
The random number generator used for genarting keys uses Rsa_generate_key function of openssl libraries.

## 34.8.3 CSF Header Generation

uni_sign tool can be used for the following functions :

- CSF header generation along with signature for both ISBC and ESBC phase
- CSF header generation without signature if private key is not provided

Usage :: $./uni_sign [options] <input_file>

<INPUT_FILE> input file provided by the user

[options] :

Table 26: [options] used:

| Field | Field Description |
|---|---|
| --verbose | Generate output header along with displays the header info. |
| --key_ext | Generate ISBC/ESBC output header with ISBC Key Extension. |
| --hash | Print the hash of the public key as per input file(key/srk table). |
| --img_hash | Dump the complete hash in a separate file (DEFAULT file name: hash.out). Header file is generated without signature embedded in it. |
| --sign_app_verify | .Generates the output header file based on input file provided,verifies the SHA hash(fig.2) calculated over output file, with the hash file generated with – img_hash option. The signature provided as input is appended in the output header binary file. HASH FILE and SIGN FILE names should be defined in input file passed. |
| --help | Show the help message and exit. |

### 34.8.3.1 Verbose Mode (--verbose)

Verbose mode can be used to display extra information while creating the header. If selected, along with header creation, the tool will also display information about Output header and SG_TABLE entries..

Usage :: $./uni_sign --verbose <input_file>

### Example

```
$ ./uni_sign --verbose input_uboot_secure

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

Key Hash :
9288723a0253229000a70bcbaa9d3aa1acb70f6369e1e81c9225319d9a364e2a

Image Hash :6c0048c92079b5e44152634a16d2463b808b5fd6ae23e7c42dcd30f49efafa8e
********** HEADER **************
barker:0x68392781
srk_table_offset 200
srk_table_flag(8) : 1
srk_sel(8) : 1
num_srk_entries(16) : 3
psign 1410, length 128
uid_flag 0
sfp_wp(8) : 0
sec_image_flag(8) : 0
uid_flag(16) : 0
psgtable  1400 num_entries 1
img start cfffffc
FSL UID 0
OEM UID 0
sg_flag 1
hkptr bff00000
hksize 10000
********** SG TABLE ************
no of entries  1
entry 0  len 786432 ptr cff40000
SIGFNATURE file sign.out created
HEADER file hdr_uboot.out created
```

## 34.8.3.2 Public Key/ SRK Hash Generation Only (--hash)

The Hash of the Public Key or SRK Table as selected by user in the input file while signing the images needs to be fused in the SFP block. So if user wants to get the value of SRK Hash, this option can be used.

Usage :: $./uni_sign --hash <input_file>

### Example

```
$./uni_sign --hash input_uboot_secure

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

Key Hash :
```

```
478c14568d8f76e822a2d483489a5ed9752c7c453b1fe5351f085a57fae2a30f
```

#### 34.8.3.3 Combined Usage of Various Options

**Key Extension with Image Hash**

Usage :: $./uni_sign --key_ext --img_hash <input_file>

It works same as option --img_hash except it embeds IE key structure also in header file generated. The hash generated also includes hash of the IE key table.

To use this option use input file as described in img_hash section above and provide fields required for IE key usage as highlighted above.

**Key Extension with Image Hash**

Usage :: $./uni_sign --key_ext --sign_app_verify <input_file>

It works same as option --sign_app_verify except it embeds IE key structure also in header file generated.

To use this option use input file as described in sign_app_verify section above and provide fields required for IE key usage as highlighted above

#### 34.8.3.4 Help (--help)

It prints help menu describing various options available.

#### 34.8.3.5 Default Usage

When uni_sign is executed without any option i.e. only providing the input file as the argument, it parses the required fields from the input file and creates the CSF header as described in 5.2 along with the Public Key/ SRK Hash, Digital Signature and SG Table to create a combined binary.

Usage :: $./uni_sign <input_file>

**Example**

```
$ ./uni_sign input_uboot_secure

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

Key Hash :
9288723a0253229000a70bcbaa9d3aa1acb70f6369e1e81c9225319d9a364e2a

HEADER file hdr_uboot.out created
```

##### 34.8.3.5.1 Sample Input File and Output

Sample input file to generate CSF Header is as follows –

```
---------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform - 1010/1040/2041/3041/4080/5020/5040/9131/9132/9164/4240/C290
```

```
PLATFORM=4240
# ESBC Flag. Specify ESBC=0 to sign u-boot and ESBC=1 to sign ESBC images.(default is
0)
ESBC=0
---------------------------------------------------
# Entry Point/Image start address field in the header.[Mandatory]
# (default=ADDRESS of first file specified in images)
ENTRY_POINT=cffffffc
---------------------------------------------------
# Specify the file name of the keys seperated by comma.
# The number of files and key select should lie between 1 and 4 for 1040 and C290.
# For rest of the platforms only one key is required and key select should not be
provided.

# USAGE (for 4080/5020/5040/3041/2041/1010/913x): PRI_KEY = <key1.pri>
# USAGE (for 1040/C290/9164/4240): PRI_KEY = <key1.pri>, <key2.pri>, <key3.pri>,
<key4.pri>

# PRI_KEY (Default private key :srk.pri) - [Optional]
PRI_KEY=srk.pri
# PUB_KEY (Default public key :srk.pub) - [Optional]
PUB_KEY=srk.pub
# Please provide KEY_SELECT(between 1 to 4) (Required for 1040/C290/9164/4240 only) -
[Optional]
KEY_SELECT=
---------------------------------------------------
# Specify SG table address, only for (2041/3041/4080/5020/5040) with ESBC=0 -
[Optional]
SG_TABLE_ADDR=
---------------------------------------------------
# Specify the target where image will be loaded. (Default is NOR_16B) - [Optional]
# Only required for Non-PBL Devices (1010/1040/9131/9132i/C290)
# Select from - NOR_8B/NOR_16B/NAND_8B_512/NAND_8B_2K/NAND_8B_4K/NAND_16B_512/
NAND_16B_2K/NAND_16B_4K/SD/MMC/SPI
IMAGE_TARGET=
---------------------------------------------------
# Specify IMAGE, Max 8 images are possible. DST_ADDR is required only for Non-PBL
Platform. [Mandatory]
# USAGE : IMAGE_NO = {IMAGE_NAME, SRC_ADDR, DST_ADDR}
IMAGE_1={u-boot.bin,cff40000,ffffffff}
IMAGE_2={,,}
IMAGE_3={,,}
IMAGE_4={,,}
IMAGE_5={,,}
IMAGE_6={,,}
IMAGE_7={,,}
IMAGE_8={,,}
---------------------------------------------------
# Specify OEM AND FSL ID to be populated in header. [Optional]
# e.g FSL_UID=11111111
FSL_UID=
OEM_UID=
---------------------------------------------------
# Specify the file names of csf header and sg table. (Default :hdr.out) [Optional]
OUTPUT_HDR_FILENAME=hdr_uboot.out

# Specify the file names of hash file and sign file.
HASH_FILENAME=img_hash.out
INPUT_SIGN_FILENAME=sign.out
```

```
# Specify the signature size.It is mandatory when neither public key nor private key
is specified.
# Signature size would be [0x80 for 1k key, 0x100 for 2k key, and 0x200 for 4k key].
SIGN_SIZE=
---------------------------------------------------
# Specify the output file name of sg table. (Default :sg_table.out). [Optional]
# Please note that OUTPUT SG BIN is only required for 2041/3041/4080/5020/5040 when
ESBC flag is not set.
OUTPUT_SG_BIN=
---------------------------------------------------
# Following fields are Required for 4240/9164/1040/C290 only

# Specify House keeping Area
# Required for 4240/9164/1040/C290 only when ESBC flag is not set. [Mandatory]
HK_AREA_POINTER=bff00000
HK_AREA_SIZE=00010000
---------------------------------------------------
# Following field Required for 4240/9164/1040/C290 only
# Specify Secondary Image Flag. (0 or 1) - [Optional]
# (Default is 0)
SEC_IMAGE=
---------------------------------------------------
```

Table 27: Description of fields.

| Field | Field Description |
|---|---|
| PLATFORM | To identify the platform/SoC for which CF Header needs to be created. |
| ESBC | Don't set this flag when code signing is being performed on the image directly verified by the ISBC. For later images in the chain of trust, set this flag. |
| ENTRY_POINT | Entry Point address / Image start address field in the header. |
| PRI_KEY | Private key filename to be used for signing the image. (File has to be in PEM format) (default = srk.pri generated by gen_keys command) FILE1 [,FILE2, FILE3, FILE4]. Multiple key support for Trust Arch v2.x devices only. |
| PUB_KEY | Public key filename in PEM format. (default = srk.pub generated by gen_keys) FILE1 [,FILE2, FILE3, FILE4]. Multiple key support for Trust Arch v2.x devices only. |
| KEY_SELECT | Specify the key to be used in signature generation when more than one key has been given as input. (Default=1, first key will be selected) |
| IMAGE_1 - IMAGE_8 | Create Entries for SG Table in the format { IMAGE_NAME, SRC_ADDR, DST_ADDR } |

*Table continues on the next page...*

Table 27: Description of fields. (continued)

| Field | Field Description |
| --- | --- |
| OEM_UID | OEM UID to be populated in the header. |
| OEM_UID_1 | OEM UID 1 to be populated in the header. Required Only for ls1 |
| FSL_UID | FSL UID to be populated in header. |
| FSL_UID_1 | FSL UID 1 to be populated in header.Required Only for ls1 |
| HK_AREA_POINTER | House Keeping Area Starting Pointer Required by Sec (Required for Trust Arch v2.x devices only when esbc option is not provided) |
| HKAREA_SIZE | House Keeping Area Size (Required for Trust Arch v2.x devices only when esbc option is not provided) |
| OUTPUT_HDR_FILENAME | Name of the combined header binary to be created by tool |
| SG_TABLE_ADDR | Specify SG_TABLE Address where Scatter Gather table is present for 2041/3041/4080/5020/5040 when ESBC=0. |
| OUTPUT_SG_BIN | Specify the output file name of sg table. |
| IMAGE_TARGET | Specify the target where image will be loaded. Ex:NOR_8B/NOR_16B/NAND_8B_512/ NAND_8B_2K/NAND_8B_4K/ NAND_16B_512/ NAND_16B_2K/NAND_16B_4K/SD/MMC/SPI |
| SEC_IMG | Flag for Secondary Image. Required for Trust Arch v2.x devices only |
| MP_FLAG | Specify Manufacturing Protection Flag. Available for LS1 only. |
| VERBOSE | Specify Verbose option. Contents of header generated will be printed. |

## 34.8.3.6 Key Extension (--key_ext)

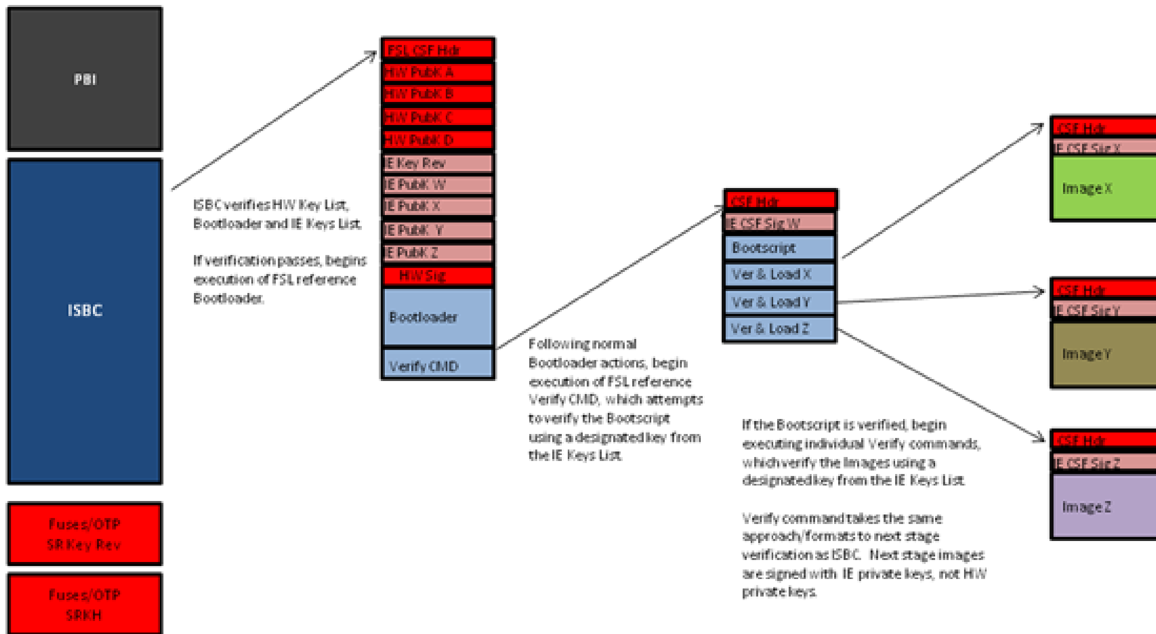
### 34.8.3.6.1 Introduction

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.

### 34.8.3.6.2 How it works

When the --key_ext option is selected, the CST signs the image along with a number of public keys. Logically, the --key_ext will be used when signing Boot 1 (bootloader), so that the bootloader and downstream images in the chain of trust can use keys which aren't directly bound to the silicon via the SRKH. Decoupling the chain of trust from the hardware super root keys minimizes the need to perform hardware key revocation.

Figure 50: Execution and Verification of Images using Key_Ext feature.




NOTE

Next stage images are signed with corresponding pair of Extension private keys list, not HW private keys.

Key Extension feature is applicable only for NOR secure Boot. It is not applicable for RAMBOOT (where data has to be copied onto RAM, eg:- NAND, SD, SPI)

## 34.8.3.6.3 IE Key Structure

Table 28: 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. |

*Table continues on the next page...*

Table 28: IE Key Structure which is embedded in header and placed in memory. (continued)

| Offset | Data Bits [0:31] |
| --- | --- |
| 0x410-0x80f | Key 2 value. |
| - | - |
| - | Key N value |

## 34.8.3.6.4 Sample Input File and Output

This file is same as file described above in <link to 4.1.2> except fields required for IE Key extension highlighted in red.

```
---------------------------------------------------
# Specify the platform. [Mandatory]
# Choose Platform - 1040/2080/2041/3041/4080/5020/5040/4860/4240/LS1
PLATFORM=1040
# ESBC Flag. Specify ESBC=0 to sign u-boot and ESBC=1 to sign ESBC images.(default is
0)
ESBC=0
# ESBC Header address. It contains address where ESBC header is loaded in memory.
ESBC_HDRADDR=c0b00000
---------------------------------------------------
# Entry Point/Image start address field in the header.[Mandatory]
# (default=ADDRESS of first file specified in images)
ENTRY_POINT=cffffffc
---------------------------------------------------
# Specify the file name of the keys seperated by comma.
# The number of files and key select should lie between 1 and 4 for 1040/2080 and C290.
# For rest of the platforms only one key is required and key select should not be
provided.

# USAGE (for 4080/5020/5040/3041/2041/1010/913x): PRI_KEY = <key1.pri>
# USAGE (for 1040/2080/C290/4860/4240): PRI_KEY =
<key1.pri>,<key2.pri>,<key3.pri>,<key4.pri>

# PRI_KEY (Default private key :srk.pri) - [Optional]
PRI_KEY=srk.pri
# PUB_KEY (Default public key :srk.pub) - [Optional]
PUB_KEY=srk.pub
# Please provide KEY_SELECT(between 1 to 4) (Required for 1040/2080/C290/4860/4240
only) - [Optional]
KEY_SELECT=

# Specify the file name of the extension keys seperated by comma.
# USAGE : IE_KEY = <key1.pub>,<key2.pub>,<key3.pub>,<key4.pub>,<key5.pub>
IE_KEY=<iekey1k_1.pub>,<iekey1k_2.pub>,<iekey1k_3.pub>,<iekey2k_1.pub>,<iekey2k_2.pub>,
<iekey2k_3.pub>,<iekey4k_1.pub>,<iekey4k_2.pub>

# Please provide Revoke keys. - [Optional]
# Provide key numbers from available ie keys to be revoked. Max n-1 keys can be
revoked. n is total number of IE keys.
# LSb represents key0 and MSb represents key 31. So total 32 keys are supported.
IE_REVOC=1,7
---------------------------------------------------
```

```
# Specify SG table address, only for (2041/3041/4080/5020/5040) with ESBC=0 -
[Optional]
SG_TABLE_ADDR=
---------------------------------------------------
# Specify the target where image will be loaded. (Default is NOR_16B) - [Optional]
# Only required for Non-PBL Devices (1010/9131/9132/C290)
# Select from - NOR_8B/NOR_16B/NAND_8B_512/NAND_8B_2K/NAND_8B_4K/NAND_16B_512/
NAND_16B_2K/NAND_16B_4K/SD/MMC/SPI
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 29: Description of new fields introduced.

| Field | Field Description |
|---|---|
| ESBC_HDRADDR | ESBC Header address. It contains location of ESBC header in the memory |
| IE_KEY | Extension Public key filenames to be used by further level images. (File has to be in PEM format) FILE1 [,FILE2, FILE3, FILE4]. |
| IE_REVOC | Revoked keys numbers from available ie keys. If a key is compromised then this feature helps to avoid that key usage. Max n-1 keys can be revoked. n is total number of IE keys and less than equal to 32.Ex.[1,3,5] |

OUTPUT

Size of IE Key Structure
Memory address of IE structure

```
0001400: 0000 2028 0000 000f c8b0 1500 ffff ffff
0001410: 000c 0000 0000 000f cff4 0000 ffff ffff
0001420: 0000 0000 0000 0000 0000 0000 0000 0000
0001430: 0000 0000 0000 0000 0000 0000 0000 0000
0001440: 0000 0000 0000 0000 0000 0000 0000 0000
0001450: 0000 0000 0000 0000 0000 0000 0000 0000
0001460: 0000 0000 0000 0000 0000 0000 0000 0000
0001470: 0000 0000 0000 0000 0000 0000 0000 0000
0001480: 0000 0000 0000 0000 0000 0000 0000 0000
0001490: 0000 0000 0000 0000 0000 0000 0000 0000
00014a0: 0000 0000 0000 0000 0000 0000 0000 0000
00014b0: 0000 0000 0000 0000 0000 0000 0000 0000
00014c0: 0000 0000 0000 0000 0000 0000 0000 0000
00014d0: 0000 0000 0000 0000 0000 0000 0000 0000
00014e0: 0000 0000 0000 0000 0000 0000 0000 0000
00014f0: 0000 0000 0000 0000 0000 0000 0000 0000
0001500: 0000 0001 0000 0008 0000 0100 b277 ef63
0001510: bde1 50c5 29a7 d2ab abe7 52d5 3fcb 8c00
0001520: 02b7 cc0b bdc8 dbba f966 6b9d a9c2 d8e2
0001530: e05d 2313 99f8 4b1a 9198 aa76 68d1 b452
0001540: 9b23 6d46 5ac0 4554 cf01 6b40 827d 12ac
0001550: 9b7f 9d25 13a6 c5ca 8f8c af58 d29b 865f
0001560: 0969 33cc d5b0 d90a f5ee 170c e896 3b1d
0001570: 086e 9f45 31f5 c843 4038 2137 c37f 4fab
0001580: 9e78 f8e8 f7f8 22f5 5759 deab 0000 0000
```

Highlighted fields shows IE structure is embedded in the CSF header.

### 34.8.3.6.5 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 30: 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.

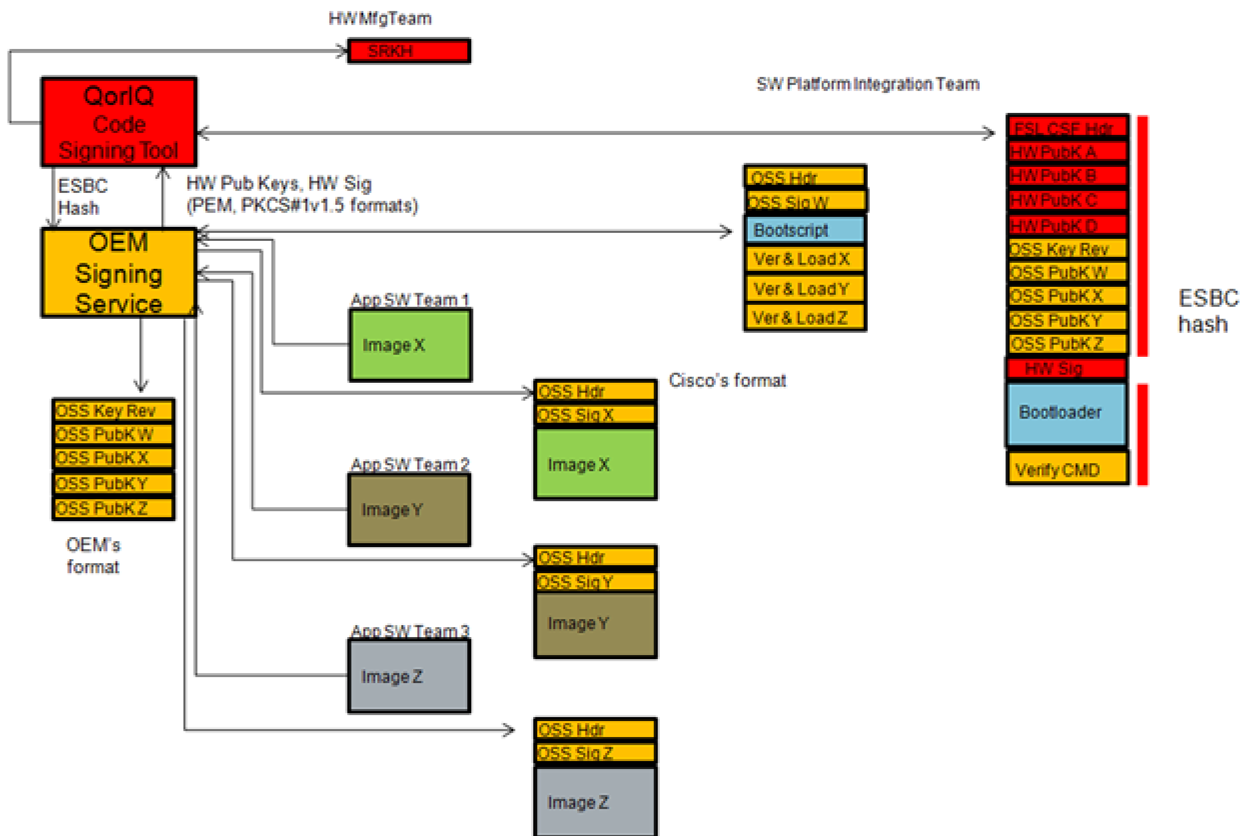
## 34.8.3.7 Image Hash Generation (--img_hash)

### 34.8.3.7.1 Introduction

The -img_hash generation feature provides OEMs with the ability to perform code signing in a secure environment which does not run the FSL Code Signing Tool.

When used in conjunction with the IE Key List feature, the user generates the IE key list and the list of hardware public keys (those bound to the silicon with the SRKH), and passes them to the CST for inclusion in the ESBC image hash calculation. The CST generates the appropriate CSF header, S/G table, and key lists, then calculates and exports the SHA256 hash. The OEM then RSA encrypts the hash with one of the private keys associated with the public key provided to verify the signature.

The signature, which must be in PKCS#1v1.5 format, is then appended to the ESBC. See section 4.7 for more information on appending.



#### 34.8.3.7.2 Features

- Generates hash file in binary format which contains SHA256 hash of CSF header along with keys(SRK table, IE keys), SG table and its entries.
- Generates output header binary file based on the fields specified in input file.
- Output header binary file doesn't contain signature.
- Provides flexibility to manually append signature at the end of output header file. User's can use their own custom tool to generate the signature. The signature offset chosen in the header is such that the signature can be appended at the end of the header file.
- This option does not require private key to be provided. But the corresponding Public key from the public/ private key pair must be provided to calculate correct SHA256 hash.

Usage Example:

```
./uni_sign --img_hash input_uboot_nor_secure

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

Key Hash :
f1f18e7eaeb28cee50a30f5ba5d270b3e71b2c7c8382507ca8e7e4c110547eb2

HASH file hash.out created
HEADER file esbc_hdr.out created
```

### 34.8.3.8 Import Signature (--sign_app_verify)

#### 34.8.3.8.1 Introduction

This feature helps to embed the signature in the output header binary file which is calculated over the hash exported using the --img_hash option.

- Uses same input file as being used for –img_hash option.
- Generates output header binary file as specified in input file.
- It requires hash file through which signature is calculated, signature file which contains calculated signature and input file.
- It verifies the hash by comparing the hash provided in hash file and hash calculated on current header generated using input file.
- It embeds the signature in output file copied from signature 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.

#### 34.8.3.8.2 Example

```
$ ./uni_sign --sign_app_verify input_uboot_nor_secure

================================================================
This product includes software developed by the OpenSSL Project
```

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

Key Hash :
f1f18e7eaeb28cee50a30f5ba5d270b3e71b2c7c8382507ca8e7e4c110547eb2

HEADER file esbc_hdr.out created
```

## 34.8.3.9 Signature Calculation (--gen_sign)

### 34.8.3.9.1 Introduction

This feature is provided for the user to calculate signature for a given hash using CST Tool. In this case we would seek only hash file and priv key file from the user as input. And 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

Option can be:

--sign_file SIGN_FILE Provide 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. SIGN_FILE default value is sign.out

HASH_FILE: name of hash file containing hash over signature needs to be calculated.

PRIV_KEY_FILE: name of key file containing private key.

### 34.8.3.9.2 Example

```
$ ./gen_sign img_hash.out 1k_1.pri
HEADER file sign.out created
```

## 34.8.3.10 Signature Embedding (--sign_embed)

### 34.8.3.10.1 Introduction

This feature embeds signature in the header file generated using img_hash option which generates header but doesn't embed signature in the header. This option opens header file and copy signature at signature offset in the header.

The steps would be::

1. Open hdr.out file.
2. Read sign_offset from hdr.out file.
3. Read sign.out file.
4. Append signature in the hdr.out file at an offset of sign_offset.

User can generate complete header in three steps independent of each other followed in the same order. Uni_sign with img_hash followed by gen_sign which in turn followed by sign_embed.

Usage :: ./sign_embed [option] HDR_FILE SIGN_FILE

--trust_arch TRUST_ARCH TRUST_ARCH is the value of trust arch of the platform.

--hdr_file HDR_OUT HDR_OUT is the output file generated, its default value is hdr_new.out. HDR_OUT is generated embedding signature from SIGN_FILE in HDR_FILE generated using --img_hash option.

HDR_FILE: name of header file in which signature needs to be embed.

SIGN_FILE: name of sign file containing signature which needs to be embed

#### 34.8.3.10.2 Example

```
./sign_embed hdr_uboot.out sign.out --hdr_file hdr1.out --trust_arch 1
HEADER file hdr1.out created
```

## 34.9 Product execution

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

### 34.9.1 Getting started

The example below demonstrates the secure-boot flow with all the images loaded in NOR Flash.

Steps in the demo would be:

1. ISBC code would validate the ESBC code.
2. On successful validation, ESBC code would run, which would then validate the boot script.
3. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to validate next level images, i.e rootfs, linux 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.

NOTE

For PowerPC SoC's, ISBC expects the code to be validated i.e. ESBC code to be within 0 - 3.5G address map. In the demo we map the flash to address 0xc0000000 for the ISBC code to validate ESBC in NOR Flash using PBI commands. Once the control reaches the ESBC code the earlier mapping of flash is removed and flash is mapped to address 0xe0000000.

For useful commands of U-Boot and CCS, refer Useful U-Boot and CCS Commands on page 568

#### 34.9.1.1 Environment for Secure Boot

There are 2 ways in which secure boot can be initiated:

- Set SB_EN bit in RCW to 1.
- Programming the ITS fuse.

In a manufacturing environment, it is recommended that all fuses be programmed at once, including the ITS and OEM Section Write Protect bits. In a prototyping environment, it may be preferable to leave ITS and Write Protect unprogrammed (relying on RCW to initiate secure boot) until the developer has confidence in the secure boot process.

Two different RCW's are provided for the demo purpose:

1. The RCW which has SB_EN bit set as 0 (sben0) and can be used when ITS = 1 i.e user wants to initiate secure boot flow using fuse.

2. The RCW which has the SB_EN bit set as 1 (sben1)and can be used when user wants to initiate secure boot using RCW.

#### 34.9.1.2 SDK Images required for the demo

Given below are the images required for the demo which are built with Yocto as part of the SDK:

1. RCW with PBI commands
2. ESBC (U-Boot)
3. uImage (Linux Image) *
4. rootfs Image *
5. Device tree *

Please refer to User Manual QorIQ DPAA SDK for detailed description on how to run Yocto Build. Once the build process finishes, all the binaries would be present at the following location:

*build_<platform>_release/tmp/deploy/images*

The images will be created with the following names:

| | |
|---|---|
| u-boot-<platform>.bin | U-Boot binary image for Secure Boot |
| uImage-<platform>.bin | kernel image that can be loaded with U-Boot |
| fsl-image-core-<platform>.ext2.gz.u-boot | ramdisk filesystem image that can be loaded with U-Boot |
| uImage-<platform>.dtb | device tree binary(dtb) for kernel bootup |

RCW files will be present in the path *build_<platform>_release/tmp/deploy/images/rcw*

NOTE

* Some platforms like LS1043 have support for LINUX boot using a single kernel FIT image instead of 3 separate images (uImage, rootfs and DTB)

### 34.9.2 Chain of Trust

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

Steps in the demo would be:

1. ISBC code would validate the ESBC code.
2. On successful validation, ESBC code would run, which would then validate the boot script.
3. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to validate next level images, i.e rootfs, linux 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.

#### 34.9.2.1 Other images required for the demo

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

1. CSF Header of the ESBC u-boot image
2. CSF Header of the uImage *
3. CSF Header of the rootfs image *
4. CSF Header of the device tree *

5. Boot Script
6. CSF Header of the boot script

The following section describes how to create the CSF headers and boot script.

NOTE

* Some platforms like LS1043 have support for LINUX boot using a single kernel FIT image instead of 3 separate images (uImage, rootfs and DTB). For these platforms a single CSF Header is required.

### 34.9.2.2 Boot Script and Signing the images

User can sign all the images with same public/private key pair or can use different key pairs to sign the images. Section below describes both the processes.

CST tool used for signing the images is provided as a package with yocto and is built for host. It can be run from your host machine.

Install path for CST binaries in yocto:

```
tmp/sysroots/x86_64-linux/usr/bin/cst/
```

CST uses openssl libraries, version 0.9.8.

In the Yocto environment, the user needs to use below commands to rebuild cst:

1. bitbake cst-native –c cleanall
2. bitbake cst-native
3. Modify the CST source code if needed.
4. bitbake cst-native –c cleanall
5. bitbake cst-native –c patch

Note: after step5, CST binary will be put to build_<platform_release/tmp/sysroots/x86_64-linux/usr/bin/cst/ directory.

Table 31: Platforms supported in SDK for Secure Boot

| Soc | <platform> supported in sdk | <platform> supported in CST |
|---|---|---|
| B4860 | b4860qds | b4860 |
| P2041 | p2041rdb | p3_p4_p5 |
| P3041 | p3041ds | p3_p4_p5 |
| P4080 | p4080ds | p3_p4_p5 |
| P5020 | p5020ds | p3_p4_p5 |
| P5040 | p5040ds | p3_p4_p5 |
| T1024 | t1024rdb | t1_t2_t4 |
| T104x | t1040rdb, t1042rdb | t1_t2_t4 |

*Table continues on the next page...*


Non-disclosure Agreement Required.

Table 31: Platforms supported in SDK for Secure Boot (continued)

| Soc | <platform> supported in sdk | <platform> supported in CST |
|---|---|---|
| T2080 | t2080qds, t2080rdb | t1_t2_t4 |
| T4240 | t4240qds | t1_t2_t4 |
| LS1021 | ls1021aqds | ls1 |
| LS1021 | ls1021atwr | ls1 |
| LS1043 | ls1043ardb | ls1043 |
| LS1043 | ls1043aqds | ls1043 |

NOTE

Some platforms like LS1043 have support for LINUX boot using a single kernel FIT image instead of 3 separate images (uImage, rootfs and DTB).

For such platforms, in CST as well instead of 3 different input files for signing uImage, rootfs and dtb, there is a single input file named **input_kernel_secure** which can be used to sign the single kernel FIT image.

This applies to all subsequent sections below.

## 34.9.2.2.1 Signing the images using same key pair

CSF header needs to be generated for all the images. More details on the commands provided by CST can be found in Section .

1. Generate the key pair to be used for signing the image

   ```
   ./gen_keys 1024
   ```

   Key pair - public key file - srk.pub and private key in srk.priv would be generated.

2. Obtain hash string of the key pair generated to be programmed in SFP

   ```
   ./uni_sign --hash input_files/uni_sign/<platform>/input_uboot_secure
   ```

   This would provide you the 256 bit hash in form of string of the key pair generated in the previous step. The hash has to be programmed in the SRK hash Fuse.

3. Create CSF header for u-boot Image.

   ```
   ./uni_sign input_files/uni_sign/<platform>/input_uboot_secure
   ```

   The input fields are specified in input_uboot_secure file. Please ensure that the filename mentioned in the input_uboot_secure is same as copied in the cst directory.

4. Create CSF header for Linux uImage

   **./uni_sign input_files/uni_sign/<platform>/input_uimage_secure**

   uImage.bin would be validated form u-boot. The flash address used here is according to the address map of u-boot. Please ensure that filename mentioned in the input_uimage_secure is same as copied in the cst directory.

5. Create CSF header for rootfs

   ```
   ./uni_sign input_files/uni_sign/<platform>/input_rootfs_secure
   ```

   ```
   Please make sure that filename mentioned in the input_rootfs_secure is same as copied in
   the cst directory
   ```

6. Create CSF Header for hardware device tree

   `./uni_sign input_files/uni_sign/<platform>/input_dtb_secure`

   `Please make sure that filename mentioned in the input_dtb_secure is same as copied in the cst directory`

7. Create Boot script

   Bootscript is a U-Boot script image. Steps to create bootscript are given below :

   a. Create a text file bootscript.txt with following commands.

   ```
   esbc_validate <uImage CSF Header address>

   esbc_validate <dtb CSF Header address>

   esbc_validate <rootfs CSF Header address>

   bootm <uImage Address> <rootfs address> <dtb address>
   ```

   b. Then you will have to use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

   powerpc arch:: `/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d bootscript.txt bootscript`

   arm arch:: `/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A arm -T script -a 0 -e 0x40 -d bootscript.txt bootscript`

8. Generate CSF hdr for the boot script

   ./uni_sign input_files/uni_sign/<platform>/input_bootscript_secure

   The fields can be changed in the input files for the images based on the requirement.

## 34.9.2.2.2 Signing the images using different key pair

If boot script is also signed with a different key, remember to define the macro "CONFIG_BOOTSCRIPT_KEY_HASH" with the hash of the key used to sign the boot script in file *arch/powerpc/asm/include/fsl_secure_boot.h. ESBC u-boot would have to be recompiled if any change in this file is made*.

1. Generate the key pair to be used for signing the image

   `./gen_keys 1024 -p u-boot.priv -k u-boot.pub`

   Key pair - public key file - u-boot.pub and private key in u-boot.priv would be generated.

2. Obtain hash string of the key pair generated to be programmed in SFP

   `./uni_sign --hash` **`input_files/uni_sign/<platform>/input_uboot_secure`**

   This would provide you the 256 bit hash in form of string of the key pair generated in the previous step. The hash has to be programmed in the SRK hash Fuse.

3. Create CSF header for u-boot Image.

   Open `input_files/uni_sign/<platform>/input_uboot_secure and change PRI_KEY and PUB_KEY to u-noot.priv and u-boot.pub respectively and run the following command.`

   ./uni_sign input_files/uni_sign/<platform>/input_uboot_secure

4. Create CSF header for Linux uImage using different key pair.

Repeat step 1 to generate another key pair.

`./gen_keys 1024 -p lnx.priv -k lnx.pub`

Open `input_files/uni_sign/<platform>/input_uimage_secure` and change `PRI_KEY` and `PUB_KEY` to `lnx.priv` and `lnx.pub` respectively and run the following command. **`./uni_sign input_files/uni_sign/<platform>/input_uimage_secure`**

Remember the "Key Hash" printed as it would be required in esbc_validate command in boot script. Say the hash of the key is <lnx_key_hash>

5. Create CSF header for rootfs

`./gen_keys 1024 -p rootfs.priv -k rootfs.pub`

Open `input_files/uni_sign/<platform>/input_rootfs_secure` and change `PRI_KEY` and `PUB_KEY` to `rootfs.priv` and `rootfs.pub` respectively and run the following command.

**`./uni_sign input_files/uni_sign/<platform>/input_rootfs_secure`**

Remember the "Key Hash" printed as it would be required in esbc_validate command in boot script. Say the hash of the key is <rootfs_key_hash>

6. Create CSF Header for hardware device tree

`./gen_keys 1024 -p dtb.priv -k dtb.pub`

Open `input_files/uni_sign/<platform>/input_dtb_secure` and change `PRI_KEY` and `PUB_KEY` to `dtb.priv` and `dtb.pub` respectively and run the following command. **`./uni_sign input_files/uni_sign/<platform>/input_dtb_secure`**

Remember the "Key Hash" printed as it would be required in esbc_validate command in boot script. Say the hash of the key is <dtb_key_hash>

7. Write Boot script

Bootscript is a U-Boot script image. Steps to create bootscript are given below :

a. Create a text file bootscript.txt with following commands.

```
esbc_validate <uImage CSF Header address>  <lnx_key_hash>
esbc_validate <dtb CSF Header address>  <dtb_key_hash>
esbc_validate <rootfs CSF Header address>  <rootfs_key_hahs>
bootm <uImage Address> <rootfs address> <dtb address>
```

NOTE

Hashes would be the 256 bit string hash. These are the hashes of the key used to sign the respective images.

b. Generate header over bootscript.txt which will be consumed by uboot command source

powerpc arch :: tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d bootscript.txt bootscript

arm arch :: tmp/sysroots/x86_64-linux/usr/bin/mkimage -A arm -T script -a 0 -e 0x40 -d bootscript.txt bootscript

8. Generate CSF hdr for the boot script

`./gen_keys 1024 -p bs.priv -k bs.pub`

Open `input_files/uni_sign/<platform>/input_dtb_secure` and change `PRI_KEY` and `PUB_KEY` to `bs.priv` and `bs.pub` respectively and run the following command.

```
./uni_sign input_files/uni_sign/<platform>/input_dtb_secure
```

### 34.9.2.3 Running secure boot (Chain of Trust)

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

      Program the ITS fuse. Use RCW with SB_EN=0

      Or
   b. **Flow B**

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

      **Note: For P3/P4/P5, if ITS fuse is blown, then ITF fuse must also be blown. (The value of ITS and ITF fuse must be identical.)**
2. Blow other required fuses on the board. (OTPMK and SRK hash[4]) For more details regarding fuse blowing, CCS and Boot Hold Off, refer to Platform reference manual and Trust Architecture User Guide.

   NOTE

   SRK hash in the fuse should be same as the hash of the key pair being used to sign the ESBC u-boot. Step 2 of #unique_564

   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 (Address map used for the demo on page 563).
   a. **Flow A** - All the images would have to be flashed at the current bank addresses. Once ITS fuse is blown, the control would automatically shift to ISBC on power on.
   b. If you are using **Flow B**, you can use alternate bank for demo purpose. This would mean flashing the images on alternate bank addresses from Bank0 and then switching to Bank4.
4. Give a power on cycle to the board.
   a. For **Flow A** and **Flow B** (*Secure boot Images flashed on default Bank*)
      - On power on, ISBC code would get control, validate the ESBC image.
      - ESBC image would further validate the signed linux, rootfs and dtb images
      - Linux would come up
   b. **Flow B** (*Secure boot Images flashed on alternate Bank*)
      - On power on cycle, u-boot prompt on bank 0 would come up.
      - On switching to alternate bank, the secure boot flow as mentioned above would execute.

---

[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. 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.

## 34.9.3 Chain of Trust with Confidentiality

This section presents the steps needed to be followed in order to execute Chain of Trust with confidentiality.

The demo would be divided into two parts:

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

Steps in the demo would be:

**Step 1: Creating blobs**

1. ISBC code would validate the ESBC code.
2. On successful validation, ESBC code would run, which would then validate the boot script.
3. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to encapsullate next level images, i.e 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 will be stored.

$km - It is the address where the key modifier is stored. The modifier is required and used as key for cryptographic operation. Key modifier should be 16 bytes long.

**Step 2: Decrypting blob and booting**

1. ISBC code would validate the ESBC code.
2. On successful validation, ESBC code would run, which would then validate the boot script.
3. On successful validation of boot script, commands in boot script would be executed.
4. The boot script contains commands to decapsulate/decrypt next level images, i.e rootfs, linux 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 - It is the address where the key modifier is stored. The modifier is required and used as key for cryptographic operation. Key modifier should be 16 bytes long. It should be same as passed while encapsulating the image.

### 34.9.3.1 Other images required for the demo

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

1. Encap Boot script
2. Decap Boot script
3. CSF header for ESBC u-boot Image

4. CSF Header of the encap boot script
5. CSF Header of the decap boot script

The following section describes how to create the CSF headers and boot script.

### 34.9.3.2 Encap Bootscript

1. Create a bootscript_en.txt file with following commands:

```
blob enc <uImage address> 0x10000000 <uImage size> <key_modifier address>

erase <encapsulated uImage address> +<encapsulated uImage size>
cp.b 0x10000000 <encapsulated uImage address> <encapsulated uImage size>

blob enc <rootfs address> 0x20000000 <rootfs size> <key_modifier address>

erase <encapsulated rootfs address> +<encapsulated rootfs size>
cp.b 0x20000000 <encapsulated rootfs address>  <encapsulated rootfs size>

blob enc <dtb address> 0x1000000 <dtb size> <key_modifier address>

erase <encapsulated dtb address> +<encapsulated dtb size>
cp.b 0x1000000 <encapsulated dtb address>  <encapsulated dtb size>
```

   For the addresses to load images refer Section Address map used for the demo on page 563

2. Use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

```
/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d
bootscript_en.txt bootscript_encap
```

### 34.9.3.3 Decap Bootscript

1. Create a bootscript_de.txt file with following commands:

```
blob dec <encapsulated uImage address> 0x10000000 <uImage size + 0x30> <key_modifier
address>

blob dec <encapsulated rootfs address> 0x20000000 <rootfs size + 0x30> <key_modifier
address>

blob dec <encapsulated dtb address> 0x1000000 <dtb size + 0x30> <key_modifier address>

bootm 0x10000000 0x20000000 0x1000000
```

   For the addresses to load images refer Section Address map used for the demo on page 563

   The script decapsulates/decrypts the blob created by earlier boot script and boots using them.

NOTE

0x30(48 bytes) should be added in the size of encapsulated images while decapsulating them. Always 48B are added at the end of the encapsulated image which needs to be added while providing the size of image to be decapsulated in blob dec command.

2. Use the mkimage tool to convert this text file into a U-Boot image (using the image type script)

```
/tmp/sysroots/x86_64-linux/usr/bin/mkimage -A ppc -T script -a 0 -e 0x40 -d
bootscript_de.txt bootscript_decap
```

#### 34.9.3.4 Creating CSF Headers

- CSF Header for ESBC

  Use the command given below to generate the hdr for u-boot binary.

  ./uni_sign input_files/uni_sign/<platform>/<input file for uboot>

  Please change the binary name as per your uboot binary in “IMAGE_1”.

- CSF Header for bootscript_encap and bootscript_decap

  Use the command given below to generate the headers for bootscripts

  ./uni_sign input_files/uni_sign/<platform>/<input file for bootscript>

  Please change the binary name as per your bootscript in “IMAGE_1”.

#### 34.9.3.5 Running secure boot (Chain of Trust with Confidentiality)

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

      Program the ITS fuse. Use RCW with SB_EN=0

      Or

   b. Flow B

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

      **Note: For P3/P4/P5, if ITS fuse is blown, then ITF fuse must also be blown. (The value of ITS and ITF fuse must be identical.)**

2. Blow other required fuses on the board. (OTPMK and SRK hash[5]) 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.

[5] Blowing of **OTPMK**is essential to run secure boot for both Production (Flow A) and Prototyping/ Development (Flow B).

For **SRK Hash**,in Development Mode (Flow B), there is a workaround to avoid blowing fuses. For this use RCW with BOOT_HO = 1. This will put the core in Boot Hold off stage. Then a CCS can be connected via JTAG.

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

3. Flash all the generated images at locations as described in the address map (Address map used for the demo on page 563)
   - Uboot binary
   - CSF Header of uboot
   - Linux uImage
   - Rootfs
   - Device tree
   - bootscript_encap
   - CSF Header for bootscript_encap

   a. **Flow A** - All the images would have to be flashed at the current bank addresses. Once ITS fuse is blown, the control would automatically shift to ISBC on power on.
   b. If you are using **Flow B**, you can use alternate bank for demo purpose. This would mean flashing the images on alternate bank addresses from Bank0 and then switching to Bank4.
4. Give a power on cycle to the board.
   a. For **Flow A** and **Flow B** (*Secure boot Images flashed on default Bank*)
      - On power on, ISBC code would get control, validate the ESBC image.
      - ESBC image would further validate the bootscript.
      - Bootscript would encapsulate the Linux, rootfs and device tree and store the blobs at the desired locations.
   b. **Flow B** (*Secure boot Images flashed on alternate Bank*)
      - On power on cycle, u-boot prompt on bank 0 would come up.
      - On switching to alternate bank, the secure boot flow as mentioned above would execute.
5. Give a power on cycle to the board.
6. Replace the encap bootscript and its CSF header with decap bootscript and the CSF header of the decap bootscript respectively.
7. Give a power on cycle to the board.
   a. For **Flow A** and **Flow B** (*Secure boot Images flashed on default Bank*)
      - On power on, ISBC code would get control, validate the ESBC image.
      - ESBC image would further validate the bootscript.
      - Bootscript would decapsulate the Linux, rootfs and device tree bloband store them on DDR
      - Bootm commnd in bootscript would execute on successful decapsulation
      - Linux prompt would come up. .
   b. **Flow B** (*Secure boot Images flashed on alternate Bank*)
      - On power on cycle, u-boot prompt on bank 0 would come up.
      - On switching to alternate bank, the secure boot flow as mentioned above would execute.

### 34.9.4 NAND Secure Boot (Chain of Trust)

This section presents the steps and images needed for running Secure Boot Chain of Trust from NAND on **P3/P5**.

The procedure for running Secure boot from NAND is same as Secure Boot from NAND. The only difference is that in case of NOR, image is not required to be copied from NOR while in case of NAND, images have to be copied from NAND to SRAM/DDR before validation.

Images Required for Demo

1. PBL.bin

   The PBL.bin is generated using QCVS Tool. It creates the RCW along with PBI commands. ESBC (U-boot) and CSF Header for U-Boot are added using ACS_WRITE PBI commands. (For details/screenshots refer Using QCVS Tool (Secure Boot From NAND) on page 573)
2. uImage (Linux Image)
3. rootfs
4. dtb (Device Tree)
5. CSF Header of the uImage
6. CSF Header of the rootfs image
7. CSF Header of the device tree
8. Boot Script
9. CSF Header of the Boot Script

   Boot Script

   The sample bootscript.txt would have the following commands:

```
# Read uImage & Header
nand read <uImage DDR> <uImage NAND> <uImage size>
nand read <uImage Header DDR> <uImage Header NAND> <uImage Header size>

# Read rootfs & Header
nand read <rootfs DDR> <rootfs NAND> <rootfs size>
nand read <rootfs Header DDR> <rootfs Header NAND> <rootfs Header size>

# Read dtb & Header
nand read <dtb DDR> <dtb NAND> <dtb size>
nand read <dtb Header DDR> <dtb Header NAND> <dtb Header size>

# Validate and Boot
esbc_validate <uImage Header DDR>
esbc_validate <DTB Header DDR>
esbc_validate <rootfs Header DDR>
bootm <uImage DDR> <rootfs DDR> <dtb DDR>
```

Image Signing

The image signing process will remain same as in case of NOR #unique_573

<platform> will be p3_p4_p5/nand

NOTE

ISBC Key Extension Feature is not applicable for Secure Boot from NAND.

### 34.9.4.1 Running Secure Boot Chain of Trust (from NAND)

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

      Program the ITS fuse. Use RCW with SB_EN=0

      Or
   b. Flow B

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

      **Note: For P3/P5, if ITS fuse is blown, then ITF fuse must also be blown. (The value of ITS and ITF fuse must be identical.)**
2. Blow other required fuses on the board. (OTPMK and SRK hash[6]) For more details regarding fuse blowing, CCS and Boot Hold Off, refer to Platform reference manual and Trust Architecture User Guide.

> NOTE
>
> SRK hash in the fuse should be same as the hash of the key pair being used to sign the ESBC u-boot. Step 2 of #unique_564
>
> For testing purpose, the SRK Hash can be written in the mirror registers.
>
> gen_otpmk_drbg utility in cst can be used to generate otpmk key.

3. Flash all the generated images on NAND Flash at locations as described in the address map (Address map used for the demo on page 563).
4. Switch to NAND Boot.
   a. FLOW A

      Change the Switch Settings to change the RCW_SRC to NAND and power on the board.
   b. FLOW B

      Power on the board to bring up Non-Secure U-Boot on NOR and from U-Boot promt issue the following command.

```
mw.b 0xffdf0020 0x48;mw.b 0xffdf0021 0x78;mw.b 0xffdf002c 0x90;mw.b 0xffdf002d
0xf0;mw.b 0xffdf0010 0; mw.b 0xffdf0010 1
```

- The PBL would configure CPC as SRAM, update the SCRATCH register and copy the Header and U-boot (ESBC) on CPC configured as SRAM.
- ISBC code would get control, validate the ESBC image.
- ESBC image would further copy the Boot Script Header and Boot Script from NAND to DDR, validate the boot script and execute it.
- The Boot Script has commands to copy the linux images and their respective headers from NAND to DDR, validate the signed linux, rootfs and dtb images.

---

[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. 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.

- Linux would be booted.

## 34.9.5 NAND Secure Boot (Chain of Trust with Confidentiality)

This section presents the steps and images needed for running Secure Boot Chain of Trust with Confidentiality from NAND on **P3/P5**

The procedure for running Secure boot from NAND is same as Secure Boot from NAND. The only difference is that in case of NOR, image is not required to be copied from NOR while in case of NAND, images have to be copied from NAND to SRAM/DDR before validation.

Images Required for Demo

1. PBL.bin

   The PBL.bin is generated using QCVS Tool. It creates the RCW along with PBI commands. ESBC (U-boot) and CSF Header for U-Boot are added using ACS_WRITE PBI commands. (For details/screenshots refer Using QCVS Tool (Secure Boot From NAND) on page 573)
2. uImage (Linux Image)
3. rootfs
4. dtb (Device Tree)
5. Encap Boot Script
6. CSF Header of the Encap Boot Script
7. Decap Boot Script
8. CSH Header for Decap Boot Script

   Encap Boot Script

```
# uImage
nand read <uImage DDR> <uImage NAND> <uImage size>
esbc_blob_encap <uImage DDR> 0x10000000 <uImage size>
0x112233445566778899000aabbccddeeff
nand erase <encapsulated uImage NAND> <encapsulated uImage size>
nand write 0x10000000 <encapsulated uImage NAND> <encapsulated uImage size>

# rootfs
nand read <rootfs DDR> <rootfs NAND> <rootfs size>
esbc_blob_encap <rootfs DDR> 0x10000000 <rootfs size>
0x112233445566778899000aabbccddeeff
nand erase <encapsulated rootfs NAND> <encapsulated rootfs size>
nand write 0x10000000 <encapsulated rootfs NAND> <encapsulated rootfs size>

# dtb
nand read <dtb DDR> <dtb NAND> <dtb size>
esbc_blob_encap <dtb DDR> 0x10000000 <dtb size> 0x112233445566778899000aabbccddeeff
nand erase <encapsulated dtb NAND> <encapsulated dtb size>
nand write 0x10000000 <encapsulated dtb NAND> <encapsulated dtb size>
```

   Decap Boot Script

```
nand read <encapsulated uImage DDR> <encapsulated uImage NAND> <encapsulated uImage
size>
esbc_blob_decap <encapsulated uImage DDR> 0x10000000 <uImage size>
0x112233445566778899000aabbccddeeff

nand read <encapsulated rootfs DDR> <encapsulated rootfs NAND> <encapsulated rootfs
```

```
size>
esbc_blob_decap <encapsulated rootfs DDR> 0x20000000 <rootfs size>
0x11223344556677889900aabbccddeeff

nand read <encapsulated dtb DDR> <encapsulated dtb NAND> <encapsulated dtb size>
esbc_blob_decap <encapsulated dtb DDR> 0x1000000 <dtb size>
0x11223344556677889900aabbccddeeff

bootm 0x10000000 0x20000000 0x1000000
```

Image Signing

The image signing process will remain same as in case of NOR #unique_573

<platform> will be p3_p4_p5/nand

### 34.9.5.1 Running Secure Boot Chain of Trust with Confidentiality (from NAND)

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

      Program the ITS fuse. Use RCW with SB_EN=0

      Or
   b. Flow B

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

      **Note: For P3/P5, if ITS fuse is blown, then ITF fuse must also be blown. (The value of ITS and ITF fuse must be identical.)**
2. Blow other required fuses on the board. (OTPMK and SRK hash[7]) For more details regarding fuse blowing, CCS and Boot Hold Off, refer to Platform reference manual and Trust Architecture User Guide.

NOTE

SRK hash in the fuse should be same as the hash of the key pair being used to sign the ESBC u-boot. Step 2 of #unique_564

For testing purpose, the SRK Hash can be written in the mirror registers.

gen_otpmk_drbg utility in cst can be used to generate otpmk key.

3. Flash all the generated images on NAND Flash at locations as described in the address map (Address map used for the demo on page 563).
   a. PBL.bin
   b. LINUX Images (uImage, dtb, rootfs)
   c. CSF Header for bootscript_encap

[7] 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.

    d. bootscript_encap
4. Switch to NAND Boot.
    a. **FLOW A**

       Change the Switch Settings to change the RCW_SRC to NAND and power on the board.
    b. **FLOW B**

       Power on the board to bring up Non-Secure U-Boot on NOR and from U-Boot promt issue the following command.

```
mw.b 0xffdf0020 0x48;mw.b 0xffdf0021 0x78;mw.b 0xffdf002c 0x90;mw.b 0xffdf002d
0xf0;mw.b 0xffdf0010 0; mw.b 0xffdf0010 1
```

- The PBL would configure CPC as SRAM, update the SCRATCH register and copy the Header and U-boot (ESBC) on CPC configured as SRAM.
- ISBC code would get control, validate the ESBC image.
- ESBC image would further copy the Boot Script Header and Boot Script from NAND to DDR, validate the boot script and execute it.
- Bootscript would encapsulate the Linux, rootfs and device tree and store the blobs at the desired locations.

5. Revert the switch settings to earlier RCW_SRC and power on the board. Replace the encap bootscript and its CSF header with decap bootscript and the CSF header of the decap bootscript respectively.
6. Switch to NAND Boot.
    a. **FLOW A**

       Change the Switch Settings to change the RCW_SRC to NAND and power on the board.
    b. **FLOW B**

       Power on the board to bring up Non-Secure U-Boot on NOR and from U-Boot promt issue the following command.

```
mw.b 0xffdf0020 0x48;mw.b 0xffdf0021 0x78;mw.b 0xffdf002c 0x90;mw.b 0xffdf002d
0xf0;mw.b 0xffdf0010 0; mw.b 0xffdf0010 1
```

- The PBL would configure CPC as SRAM, update the SCRATCH register and copy the Header and U-boot (ESBC) on CPC configured as SRAM.
- ISBC code would get control, validate the ESBC image.
- ESBC image would further copy the Boot Script Header and Boot Script from NAND to DDR, validate the boot script and execute it.
- Bootscript would copy the Linux, rootfs and device tree blobs on DDR and then decapsulate them on DDR.
- Bootm commnd in bootscript would execute on successful decapsulation.
- Linux prompt would come up.

## 34.10 Troubleshooting

Table 32: 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.<br>• If OTMPK fuse is correct (see Step 1), check the SCRATCHRW2 register for errors. Refer to Section for error codes.<br>• If **Error code = 0** then check the Security Monitor state in HPSR register of Sec Mon.<br>**Sec Mon in Check State (0x9)**<br>If ITS fuse = 1, then it means ISBC code has reset the board. This may be due to the following reasons:<br>Hash of the public key used to sign the ESBC u-boot doesn't match with the value in SRK Hash Fuse<br>Or<br>Signature verification of the image failed<br>**Sec Mon in Trusted State (0xd) or Non Secure State (0xb)**<br>Check the entry point field in the ESBC header. It should be 0xcffffffc for the demo described in Section 4.<br>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. Check Step 1 in #unique_578. You would reach this stage if ITS = 0 and you are using rcw where sben0 is present in its name. |
| 3 | u-boot hangs or board resets | Some validation failure occurred in ESBC u-boot. Error code and description would be printed on u-boot console. Refer to for more details on errors. |

## 34.11 CSF Header Data Structure Definition

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

P3/P4/P5 Platforms

Figure 51: CSF Header for P3/P4/P5 (ISBC and ESBC Phase)

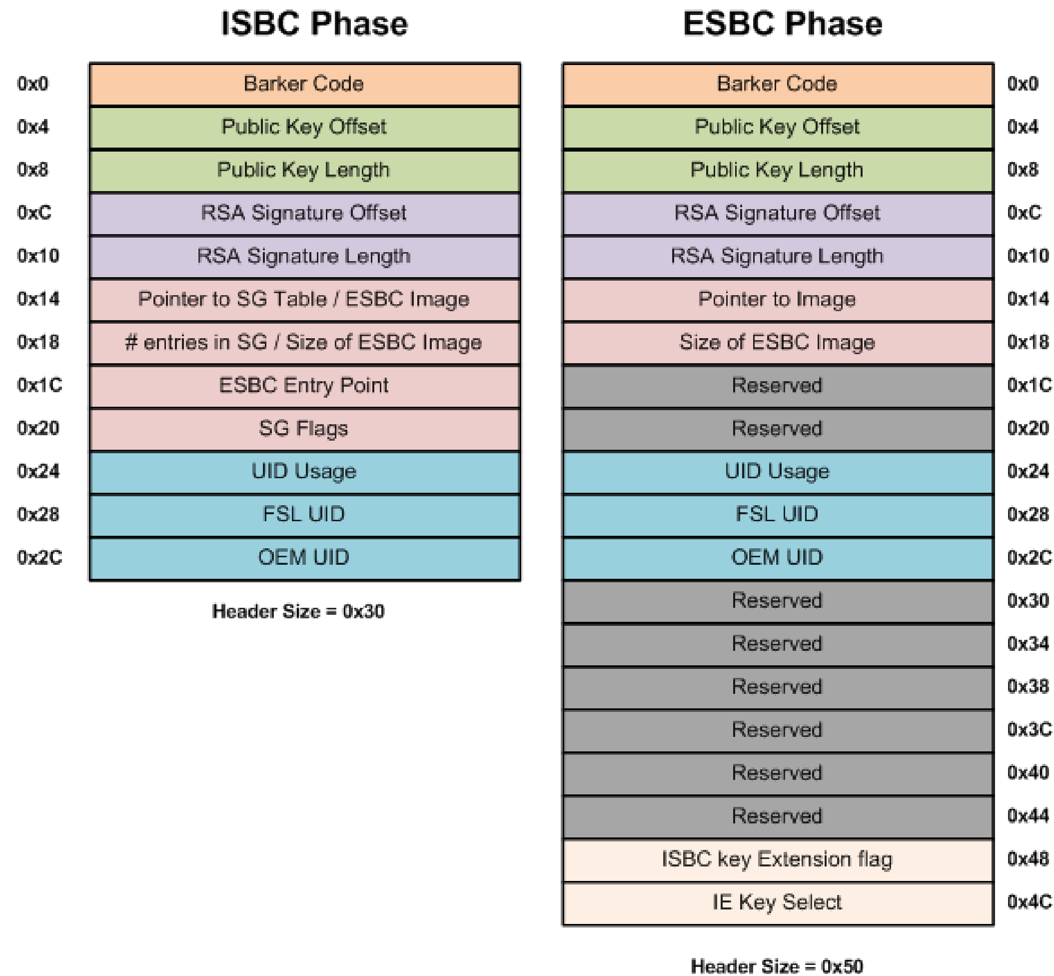



Table 33: CSF Header Format (P3/P4/P5 Platforms)

| Offset | Data Bits [0:31] |
|---|---|
| 0x00-0x03 | Barker code.<br>This location should contain the value: 0x68392781. The ISBC code searches for this Barker code. If the value in this location does not match the Barker code, the ISBC stops execution and reports error. |
| 0x04-0x07 | Public key offset.<br>This location contains an offset in bytes of the public key from the start of CSF header. Using this offset and the public key length, the public key is read. |
| 0x08-0x0b | Public key length in bytes.<br>(Value populated here should be twice of Modulus size). Supported sizes are 256, 512 or 1024 bytes (2 * 1024, 2 * 2048 , 2 * 4096 bits). |

*Table continues on the next page...*

Table 33: CSF Header Format (P3/P4/P5 Platforms) (continued)

| Offset | Data Bits [0:31] |
|---|---|
| 0x0c-0x0f | **RSA Signature offset.**<br>This location contains an offset(in bytes) of the RSA signature from the start of CSF header. Using this offset and the Signature length, the RSA signature is read. The RSA signature is calculated over CSF Header, Scatter Gather table and ESBC images. |
| 0x10-0x13 | **RSA Signature length** in bytes. |
| 0x14-0x17 | For ISBC Phase:<br>Based on the Scatter Gather flag in CSF header, this location can either be treated as **Pointer to Scatter Gather table or the address of ESBC image**.<br>For ESBC Phase:<br>This location is treated as **address of image**(linux/bootscript/rootfs/dtb) to be validated. |
| 0x18-0x1b | For ISBC Phase:<br>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.<br>For ESBC Phase: Size of image to be validated |
| 0x1c-0x1f | For ISBC Phase:<br>**ESBC entry point**. ISBC transfers control to this location upon successful validation of ESBC image(s).<br>For ESBC Phase: Reserved. |
| 0x20-0x23 | For ISBC Phase:<br>**Scatter Gather flag.**<br>0x0000 - 0x14-0x17 is a pointer to the ESBC image<br>0x0001 - 0x14-0x17 is a pointer to a scatter/gather table.<br>For ESBC Phase: Reserved |
| 0x24-0x27 | **Unique ID Usage.**<br>UIDs present in the CSF Header are compared to the corresponding UIDs in the SFP, and are included in the ESBC validation.<br>0x0000 - No UIDs are present in CSF header<br>0x0001 - FSL_UID and OEM_UID are present in CSF header<br>0x0002 - Only OEM_UID present in CSF header<br>0x0004 - Only FSL_UID present in CSF header |

*Table continues on the next page...*

Table 33: CSF Header Format (P3/P4/P5 Platforms) (continued)

| Offset | Data Bits [0:31] |
|---|---|
| 0x28-0x2b | **Freescale unique ID.**<br>A unique 32 bit value, which is specific to Freescale. This value is compared with the FSL ID in Secure Fuse Processor 's FSL-ID registers |
| 0x2c-0x2f | **OEM unique ID.**<br>A unique 32 bit value, which is specific to OEM. This value is compared with the OEM ID in Secure Fuse Processor 's OEM-ID registers |
| 0x30-0x47 | For ISBC Phase: Not Applicable<br>For ESBC Phase: Reserved |
| 0x48-0x4b | For ISBC Phase: Not Applicable<br>For ESBC Phase:<br>**ISBC key Extension flag.**<br>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<br>For ESBC Phase:<br>**IE Key Select.**<br>Key Number to be used from the IE Key Table if IE flag is set. |

Table 34: Scatter Gather Table Format (P3/P4/P5 Platforms)

| Offset | Data Bits [0:31] |
|---|---|
| 0x00-0x03 | Length in bytes of the first segment of the ESBC image. |
| 0x04-0x07 | Pointer to first segment of ESBC image. |
| 0x08-0x0b | Length in bytes of the second segment of the ESBC image. |
| 0x0c-0x0f | Pointer to second segment of ESBC image. |
| 0xww-0xxx | Length in bytes of the nth segment of the ESBC image. |
| 0xyy-0xzz | Pointer to nth segment of ESBC image |

Table 35: Signature (P3/P4/P5 Platforms)

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

Table 36: Public key (P3/P4/P5 Platforms)

| Offset | Data Bits [0:31] |
|---|---|
| 0x00-size | Public Key Value. The hash of this public key is compared with the hash stored in Secure Fuse Processor SRKH registers. |

B4/T1/T2/T4 Platforms

Figure 52: CSF Header for B4/T1/T2/T4 (ISBC and ESBC Phase)

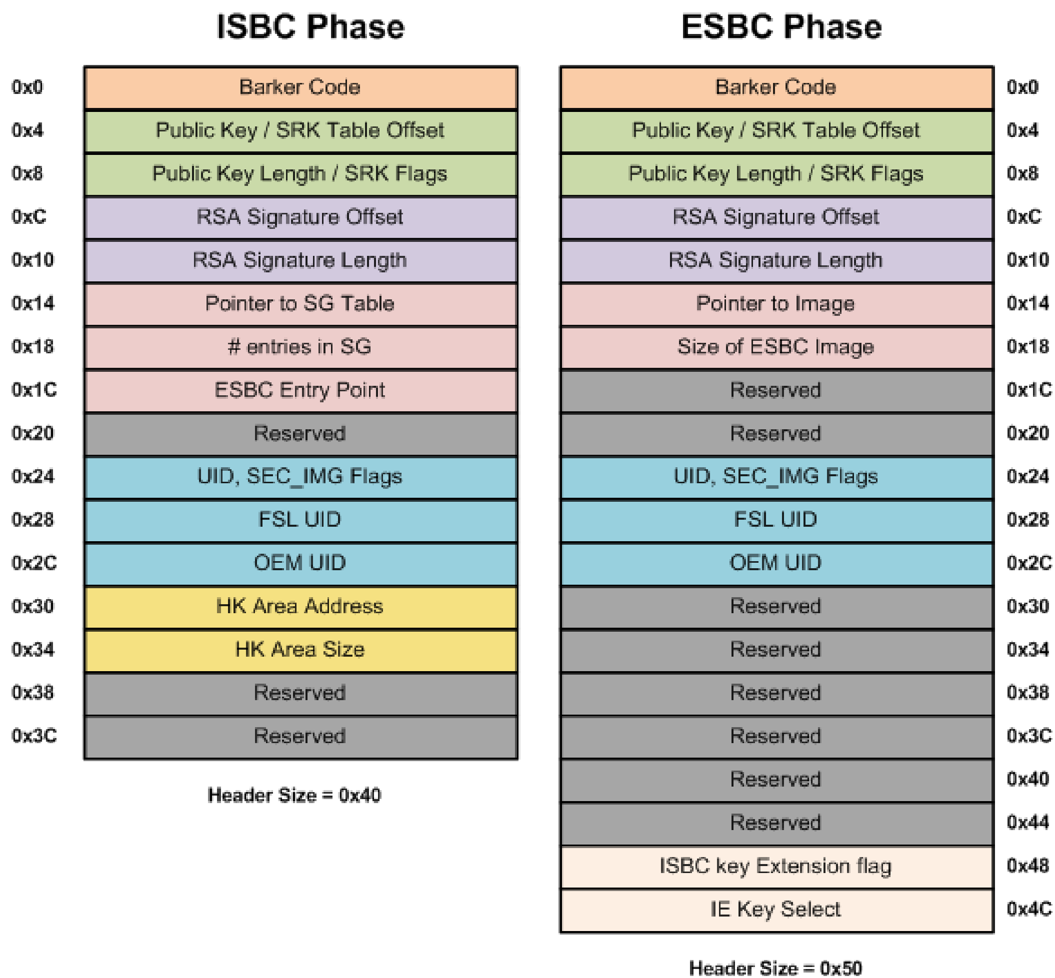


Table 37: CSF Header Format (B4/T1/T2/T4 Platforms)

| Offset | Data Bits [0:31] |
|---|---|
| 0x00-0x03 | **Barker code.**<br>This location should contain the value: 0x68392781. The ISBC code searches for this Barker code. If the value in this location does not match the Barker code, the ISBC stops execution and reports error. |
| 0x04-0x07 | If the srk_table_flag is not set :<br>• **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.<br>If srk_table_flag is set:<br>• **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.**<br>This flag indicates whether hash burnt in srk fuse is of a single key or of srk table. |
| 0x09-0x0b | If the srk_table_flag is not set :<br>• **Public key length**: This location contains the length of the public key in bytes.<br>If srk_table_flag is set:<br>• 0x09 – **Key Number from srk table** which is to be used for verification.<br>• 0x0a-0x0b – **Number of entries in srk table.** Minimum number of entries in table = 1, Maximum = 4. |
| 0x0c-0x0f | **RSA Signature offset.**<br>This location contains an offset(in bytes) of the RSA signature from the start of CSF header. Using this offset and the Signature length, the RSA signature is read. The RSA signature is calculated over CSF Header, Scatter Gather table and ESBC images. |
| 0x10-0x13 | **RSA Signature length** in bytes. |
| 0x14-0x17 | For ISBC Phase:<br>**SG Table offset**<br>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.<br>For ESBC Phase:<br>Address of the image to be validated. |

*Table continues on the next page...*

Table 37: CSF Header Format (B4/T1/T2/T4 Platforms) (continued)

| Offset | Data Bits [0:31] |
|---|---|
| 0x18-0x1b | For ISBC Phase:<br>**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.).<br>For ESBC Phase:<br>**Size of Image** to be validated |
| 0x1c-0x1f | For ISBC Phase:<br>**ESBC entry point.** ISBC transfers control to this location upon successful validation of ESBC image(s).<br>For ESBC Phase: Reserved |
| 0x20-0x23 | **Reserved** .(Earlier this field was SG Flag. SG flag is always assumed to be 1 in unified implementation.) |
| 0x24 | For ISBC Phase: Reserved<br>For ESBC Phase: Reserved |
| 0x25 | For ISBC Phase:<br>**Secondary Image flag**<br>Indicates if user has a secondary image available in case of failures in validating primary iamge.Valid in case of primary Images's Header.<br>For ESBC Phase:Reserved |
| 0x26-0x27 | **Unique ID Usage**<br>This location contains a flag which specifies one of these possibilities<br>• 0x00 - No UID's present<br>• 0x01 - FSL UID and OEM UID are present<br>• 0x02 - Only FSL UID is present<br>• 0x04 - Only OEM UID is present |
| 0x28-0x2b | **Freescale unique ID**.<br>A unique 32 bit value, which is specific to Freescale. This value is compared with the FSL ID in Secure Fuse Processor 's FSL-ID registers |
| 0x2c-0x2f | **OEM unique ID**.<br>A unique 32 bit value, which is specific to OEM. This value is compared with the OEM ID in Secure Fuse Processor 's OEM-ID registers |

*Table continues on the next page...*

Table 37: CSF Header Format (B4/T1/T2/T4 Platforms) (continued)

| Offset | Data Bits [0:31] |
|---|---|
| 0x30-0x33 | For ISBC Phase:<br>**Housekeeping area address**<br>This is address of start of a memory which can be accessed by devices on SOC bus. (DDR, L3 cache configured as SRAM ). The area should have been pre-configured by user through PBL commands or configuration header<br>For ESBC Phase: Reserved |
| 0x34-0x37 | For ISBC Phase:<br>**Size of the housekeeping area**<br>Size of the pre-configured memory which can be used by Boot Rom Code.<br>For ESBC Phase: Reserved |
| 0x38-0x3f | Reserved |
| 0x40-0x47 | For ISBC Phase: Not Applicable<br>For ESBC Phase: Reserved |
| 0x48-0x4b | For ISBC Phase: Not Applicable<br>For ESBC Phase:<br>**ISBC key Extension flag**<br>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<br>For ESBC Phase:<br>**IE Key Select**<br>Key Number to be used from the IE Key Table if IE flag is set. |

Table 38: Scatter Gather Table Format (B4/T1/T2/T4 Platforms)

| Offset | Data Bits [0:31] |
|---|---|
| 0x00-0x03 | Length. This location specifies the length in bytes of the ESBC image 1. |
| 0x04-0x07 | Target where the ESBC Image 1 can be found. This field is ignored in case of PBL based SOC’s. |

*Table continues on the next page...*

Table 38: Scatter Gather Table Format (B4/T1/T2/T4 Platforms) (continued)

| Offset | Data Bits [0:31] |
|---|---|
| 0x08-0x0b | Source Address of ESBC Image 1<br>NOTE<br>For high capacity SD/MMC cards(>2G), the address is in block address format |
| 0x0c-0x0f | Destination Address of ESBC Image 1<br>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<br>NOTE<br>For high capacity SD/MMC cards(>2G), the address is in block address format |
| 0x1c-0x1f | Destination Address of ESBC Image 2<br>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 39: Signature (B4/T1/T2/T4 Platforms)

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

Table 40: Public key (B4/T1/T2/T4 Platforms)

| Offset | Data Bits [0:31] |
|---|---|
| 0x00-size | Public Key Value. The hash of this public key is compared with the hash stored in Secure Fuse Processor SRKH registers. |

Table 41: SRK Table (B4/T1/T2/T4 Platforms)

| Offset | Data Bits [0:31] |
|---|---|
| 0x00-0x03 | Key 1 length |
| 0x04-0x403 | Key 1 value. (Remaining bytes will be padded with zero) |
| 0x404-0x407 | Key 2 length |
| 0x408-0x807 | Key 2 value. (Remaining bytes will be padded with zero) |
| 0x808-0x80b | Key 3 length |
| 0x80c-0xb0b | Key 3 value. (Remaining bytes will be padded with zero) |
| 0xb0c-0xb0f | Key 4 length |
| 0xb10-0xe10 | Key 4 value. (Remaining bytes will be padded with zero) |

## LS1 Platform

Figure 53: CSF Header for LS1 (ISBC and ESBC Phase)

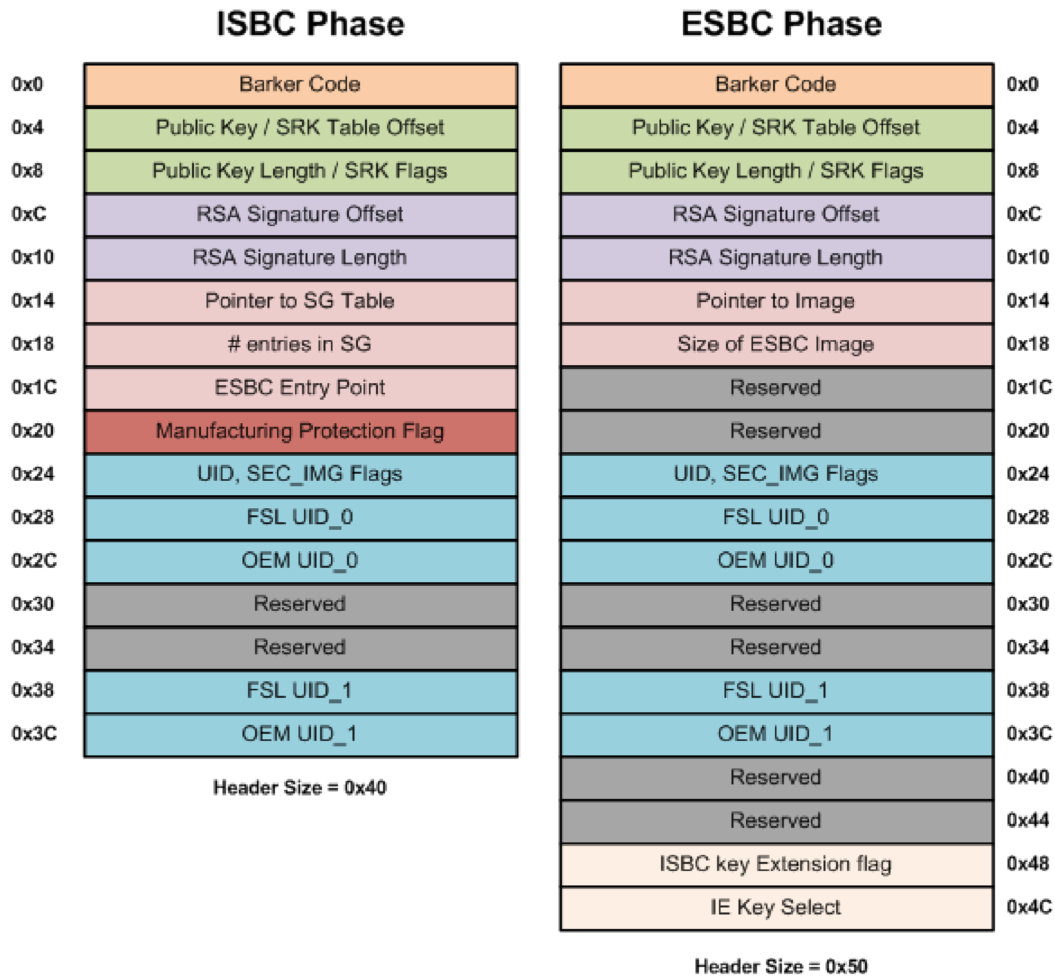



Table 42: CSF Header Format (LS1021A Platform)

| Offset | Data Bits [0:31] |
|---|---|
| 0x00-0x03 | Barker code.<br>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. |

*Table continues on the next page...*

Table 42: CSF Header Format (LS1021A Platform) (continued)

| Offset | Data Bits [0:31] |
|---|---|
| 0x07-0x04 | If the srk_table_flag is not set :<br>• **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.<br>If srk_table_flag is set:<br>• **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.**<br>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 :<br>• **0x0b-0x9 -- Public key length**: This location contains the length of the public key in bytes.<br>If srk_table_flag is set:<br>• **0x09 – Key Number from srk table** which is to be used for verification.<br>• **0x0b-0x0a – Number of entries in srk table.** Minimum number of entries in table = 1, Maximum = 4. |
| 0x0f-0x0c | **RSA Signature offset.**<br>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**:<br>**SG Table offset**<br>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.<br>**For ESBC Phase**:<br>Address of the image to be validated. |
| 0x1b-0x18 | **For ISBC Phase**:<br>**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.).<br>**For ESBC Phase**<br>**Size of image** to be validated |

*Table continues on the next page...*

Table 42: CSF Header Format (LS1021A Platform) (continued)

| Offset | Data Bits [0:31] |
|---|---|
| 0x1f-0x1c | **For ISBC Phase**:<br>**ESBC entry point.**<br>ISBC transfers control to this location upon successful validation of ESBC image(s).<br>**For ESBC Phase**: Reserved |
| 0x21-0x20 | **Manufacturing Protection Flag**<br>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<br>For ESBC Phase: Reserved |
| 0x25 | **For ISBC Phase**<br>**Secondary Image flag**<br>Indicates if user has a secondary image available in case of failures in validating primary iamge.Valid in case of primary Images's Header.<br>**For ESBC Phase**:Reserved |
| 0x27-0x26 | **Unique ID Usage**<br>This location contains a flag which specifies one of these possibilities<br>• 0x00 - No UID's present<br>• 0x01 - FSL UID and OEM UID are present<br>• 0x02 - Only FSL UID is present<br>• 0x04 - Only OEM UID is present |
| 0x2b-0x28 | **Freescale unique ID 0**<br>Upper 32 bits of a unique 64 bit value, which is specific to Freescale. This value is compared with the FSL ID 1 in Secure Fuse Processor 's FSL-ID registers |
| 0x2f-0x2c | **OEM unique ID 0**<br>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 |

*Table continues on the next page...*

Table 42: CSF Header Format (LS1021A Platform) (continued)

| Offset | Data Bits [0:31] |
|---|---|
| 0x3b-0x38 | **Freescale unique ID 1**<br>Lower 32 bits of a unique 64 bit value, which is specific to Freescale. This value is compared with the FSL ID 1 in Secure Fuse Processor 's FSL-ID registers |
| 0x3f-0x3c | **OEM unique ID 1**<br>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<br>For ESBC Phase: Reserved |
| 0x48-0x4b | **For ISBC Phase**: Not Applicable<br>**For ESBC Phase**:<br>**ISBC key Extension flag**<br>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<br>**For ESBC Phase**:<br>**IE Key Select**<br>Key Number to be used from the IE Key Table if IE flag is set. |

Table 43: Scatter Gather Table Format (LS1021A 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<br>**NOTE**<br>For high capacity SD/MMC cards(>2G), the address is in block address format |
| 0x0c-0x0f | Destination Address of ESBC Image 1<br>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. |

*Table continues on the next page...*

Table 43: Scatter Gather Table Format (LS1021A Platform) (continued)

| Offset | Data Bits [0:31] |
|---|---|
| 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<br>NOTE<br>For high capacity SD/MMC cards(>2G), the address is in block address format |
| 0x1c-0x1f | Destination Address of ESBC Image 2<br>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 44: Signature (LS1021A Platform)

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

Table 45: Public key (LS1021A 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 46: SRK Table (LS1021A 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) |

*Table continues on the next page...*

Table 46: SRK Table (LS1021A Platform) (continued)

| Offset | Data Bits [0:31] |
|---|---|
| 0xb0c-0xb0f | Key 4 length |
| 0xb10-0xe10 | Key 4 value. (Remaining bytes will be padded with zero) |

## 34.12 ISBC Validation Error Codes

P3/P4/P5 platforms

Table 47: ISBC Validation Failures (P3/P4/P5 platforms)

| Value | Code | Definition |
|---|---|---|
| 0x1 | CPUID_NO_MATCH | ISBC is not running on CPU0 |
| 0x2 | ESBC_HDR_LOC | ESBC header location is not in 3.5G space |
| 0x4 | ESBC_HEADER_BARKER | Barker code in the header is incorrect. |
| 0x8 | ESBC_HEADER_KEY_LEN | Length of public key in header is not one of the supported values. |
| 0x10 | ESBC_HEADER_SIGN_LEN | Length of RSA signature in header is not one of the supported values. |
| 0x20 | ESBC_HEADER_KEY_LEN_NOT_TWICE_SIG_LEN | Public key is not twice the length of the RSA signature |
| 0x40 | ESBC_HEADER_SG_TABLE_ADDR_NULL | SG table/ESBC image address (0x14-0x17 in CSF Header) is null |
| 0x80 | ESBC_HEADER_SG_TABLE_ADDR_NOT_IN_3_5G | SG table/ESBC image address (0x14-0x17 in CSF Header) is beyond 3.5G |
| 0x100 | ESBC_HEADER_KEY_MOD_1 | Most significant bit of modulus in header is zero. |
| 0x200 | ESBC_HEADER_KEY_MOD_2 | Modulus in header is even number |
| 0x400 | ESBC_HEADER_SIG_KEY_MOD | Signature value is greater than modulus in header |
| 0x800 | ESBC_HEADER_SG_ENTRIES_NUL | SG Table contains zero entries |

*Table continues on the next page...*

Table 47: ISBC Validation Failures (P3/P4/P5 platforms) (continued)

| Value | Code | Definition |
|---|---|---|
| 0x1000 | ESBC_HEADER_SG_ENTRIES_NOT_IN_3_5G | Address in SG entry in not in 3.5G |
| 0x2000 | ESBC_HEADER_SG_ESBC_EP | ESBC entry point in header not within ESBC address range |
| 0x4000 | HASH_COMPARE_KEY | Super Root Key Hash Comparison failure. Mismatch in the hash of the public key as present in the header with the value in the SRK HASH fuse. |
| 0x8000 | HASH_COMPARE_EM | RSA signature check failure. Signature provided by you in the header doesn't match with the signature of the ESBC image generated by ISBC. The ESBC image loaded by you may be different than the image used while generating the signature(using CST) |
| 0x10000 | SSM_CHECKSTS | SEC_MON State Machine not in CHECK state at start of ISBC. Some Security violation could have occurred. Details can be found in P4080 Reference Manual. |
| 0x20000 | SSM_TRUSTSTS | SEC_MON State Machine not in TRUSTED state at end of ISBC. |
| 0x40000 | FSL_UID | FSL_UID in ESBC Header did not match the FSL_UID in SFP |
| 0x80000 | OEM_UID | OEM_UID in ESBC Header did not match the OEM_UID in SFP |
| 0x100000 | BAD_ADDRESS | A Data / Instruction TLB Exception occurred during ISBC execution |
| 0x200000 | MISC | E500mc exception other than TLB |
| 0x400000 | ESBC_HEADER_SG_ENTIRES_BAD | SG Table too large (too many entries) |

NOTE

For error codes 0x2 - 0x2000 i,e errors in the ESBC Header, check the value of that particular field by dumping the header.

B4/T1/T2/T4/LS1021A 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 48: Core Exceptions (LS1021A platform)

| Value | Code | Definition |
|---|---|---|
| 0x1 | ERROR_UNDEFINED_INSTRUCTIO N | 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 49: Core Exceptions (B4/T1/T2/T4 platforms)

| Value | Code | Definition |
|---|---|---|
| 0x1 | ERROR_MACHINECHECK | Machine check Exception |
| 0x2 | ERROR_DSI | DSI Exception |
| 0x3 | ERROR_ISI | ISI Exception |
| 0x4 | ERROR_CRITICAL | Critical Exception |
| 0x5 | ERROR_ALIGN | Alignment Exception |
| 0x6 | ERROR_PROG | Program Exception |
| 0x13 | ERROR_DATA_TLB | Data TLB Miss |
| 0x14 | ERROR_INST_TLB | Instruction TLb Miss |
| 0x20 | ERROR_MISC | Any other exception |

Table 50: System State Failures (B4/T1/T2/T4/LS1021A 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 51: General Header Checking Failures (B4/T1/T2/T4/LS1021A 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 |
| 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 |
| 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 52: Key/Signature/UID related errors (B4/T1/T2/T4/LS1021A platforms)

| Value | Code | Definition |
| --- | --- | --- |
| 0x320 | ERROR_ESBC_HEADER_KEY_LEN | Length of public key in header is not one of the supported values. |

*Table continues on the next page...*

Table 52: Key/Signature/UID related errors (B4/T1/T2/T4/LS1021A platforms) (continued)

| Value | Code | Definition |
|---|---|---|
| 0x321 | ERROR_ESBC_HEADER_KEY_LEN_ NOT_TWICE_SIG_LEN | Public key is not twice the length of the RSA signature |
| 0x322 | ERROR_ESBC_HEADER_KEY_MOD_1 | Most significant bit of modulus in header is zero. |
| 0x323 | ERROR_ESBC_HEADER_KEY_MOD_2 | Modulus in header is even number |
| 0x324 | ERROR_ESBC_HEADER_SIG_KEY_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) |
| 0x32c | ERROR_KEY_NOT_IN_3_5G | Key is not in 3.5G boundary |

Table 53: Verification Failures (B4/T1/T2/T4/LS1021A 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 54: SEC/PAMU Failures (B4/T1/T2/T4/LS1021A 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 |
| 0x800 | ERROR_PAMU | Error while programming PAACT/SPAACT tables in PAMU |

## 34.13 ESBC Validation Error Codes

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

Table 55: 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 |

*Table continues on the next page...*

Table 55: ESBC Validation Failures (continued)

| Value | Code | Definition |
|---|---|---|
| 0x10000 | ERROR_ESBC_CLIENT_HEADER_S G | No SG support |
| 0x20000 | ERROR_ESBC_WRONG_CMD | Failure in command/Unknown command/Wrong arguments of boot script. |
| 0x40000 | ERROR_ESBC_MISSING_BOOTM | Bootm command missing from boot script. |

## 34.14 Address map used for the demo

The addresses below are effective addresses as mapped by u-boot.

P3/P4/P5/T1/T2/T4 platforms

Table 56: Memory map for P3/P4/P5/T1_T2_T4 platforms

| Address(N OR vBank 0) | Address (NOR Alternate Bank)[8] | Definition (Chain of Trust) | Definition (Chain of Trust with Confidentiality) | Size Reserved (KB) |
|---|---|---|---|---|
| E8000000 | EC000000 | RCW | RCW | 128 |
| E8020000 | EC020000 | uImage | uImage | 8064 |
| E8800000 | EC800000 | DTB | DTB | 1024 |
| E8900000 | EC900000 | | | 1024 |
| E8A00000 | ECA00000 | Bootscript | Bootscript | 1024 |
| E8B00000 | ECB00000 | ESBC U-Boot HEADER | ESBC U-Boot HEADER | 1024 |
| E8C00000 | ECC00000 | | | 2048 |
| E8E00000 | ECE00000 | Bootscript Header | Bootscript Header | 1024 |
| E8F00000 | ECF00000 | | | 1024 |
| E9000000 | ED000000 | uImage Heaer | | 1024 |
| E9100000 | ED100000 | DTB Header | | 1024 |

*Table continues on the next page...*

[8] Address to be used for loading the images in case of working in Development mode with Non-Secure Boot images on Bank 0.

Table 56: Memory map for P3/P4/P5/T1_T2_T4 platforms (continued)

| Address(NOR vBank 0) | Address (NOR Alternate Bank)[8] | Definition (Chain of Trust) | Definition (Chain of Trust with Confidentiality) | Size Reserved (KB) |
|---|---|---|---|---|
| E9200000 | ED200000 | Rootfs Header | | 1024 |
| E9300000 | ED300000 | Rootfs | Rootfs | 29696 |
| EC020000 | E8020000 | | uImage (ENCAPSULATED) | 8064 |
| EC800000 | E8800000 | | DTB (ENCAPSULATED) | 1024 |
| ED300000 | E9300000 | | Rootfs (ENCAPSULATED) | 29696 |
| EFF20000 | EBF20000 | u-boot env | u-boot env (current bank) | 128 |
| EFF40000 | EBF40000 | u-boot | u-boot | 768 |

B4 platforms

Table 57: Memory map for B4 platforms

| Address(NOR vBank 0) | Address (NOR Alternate Bank)[9] | Definition (Chain of Trust) | Definition (Chain of Trust with Confidentiality) | Size Reserved (KB) |
|---|---|---|---|---|
| EC000000 | EE000000 | RCW | RCW | 128 |
| EC020000 | EE020000 | uImage | uImage | 7040 |
| EC700000 | EE700000 | Bootscript | Bootscript | 1024 |
| EC800000 | EE800000 | DTB | DTB | 1024 |
| EC900000 | EE900000 | | | 1024 |

*Table continues on the next page...*

[8] Address to be used for loading the images in case of working in Development mode with Non-Secure Boot images on Bank 0.
[9] Address to be used for loading the images in case of working in Development mode with Non-Secure Boot images on Bank 0.

Table 57: Memory map for B4 platforms (continued)

| Address(NOR vBank 0) | Address (NOR Alternate Bank)[9] | Definition (Chain of Trust) | Definition (Chain of Trust with Confidentiality) | Size Reserved (KB) |
|---|---|---|---|---|
| ECA00000 | EEA00000 | | | 1024 |
| ECB00000 | EEB00000 | ESBC U-Boot HEADER | ESBC U-Boot HEADER | 1024 |
| ECC00000 | EEC00000 | Bootscript Header | Bootscript Header | 1024 |
| ECD00000 | EED00000 | uImage Heaer | | 1024 |
| ECE00000 | EEE00000 | Rootfs Header | | 1024 |
| ECF00000 | EEF00000 | DTB Header | | 1024 |
| ED300000 | EF300000 | | DTB (Encapsulated) | 1024 |
| ED500000 | EF500000 | | uImage (Encapsulated ) | 10496 |
| EE020000 | EC020000 | | rootfs/ rootfs (ENCAPSULATED) | 31232 |
| EFF20000 | EDF20000 | u-boot env | u-boot env | 128 |
| EFF40000 | EDF40000 | u-boot | u-boot | 768 |

NOTE
To use 512KB u-boot, change u-boot address from xxx40000 to xxx80000.

LS1020

Table 58: Memory map for LS1 platforms

| Address NOR(vBank 0) | Address (NOR Alternate Bank)[10] | Definition (Chain of Trust) | Size Reserved(KB) |
|---|---|---|---|
| 60000000 | 64000000 | RCW | 128 |

*Table continues on the next page...*

[9] Address to be used for loading the images in case of working in Development mode with Non-Secure Boot images on Bank 0.
[10] Address to be used for loading the images in case of working in Development mode with Non-Secure Boot images on Bank 0.

Table 58: Memory map for LS1 platforms (continued)

| Address NOR(vBank 0) | Address (NOR Alternate Bank)[10] | Definition (Chain of Trust) | Size Reserved(KB) |
|---|---|---|---|
| 60020000 | 64020000 | DTB | 256 |
| 60060000 | 64060000 | Bootscript | 128 |
| 60080000 | 64080000 | ESBC U-Boot HEADER | 128 |
| 600A0000 | 640A0000 | Bootscript Header | 128 |
| 60200000 | 64200000 | uImage | 8192 |
| 60A00000 | 64A00000 | Rootfs | 54272 |
| 63F00000 | 67F00000 | uImage Header | 128 |
| 63F20000 | 67F20000 | DTB Header | 128 |
| 63F40000 | 67F40000 | rootfs Header | 128 |

LS1043

Table 59: Memory map for LS1043 platforms

| Address NOR(vBank 0) | Address (NOR Alternate Bank)[11] | Definition (Chain of Trust) | Size Reserved (KB) |
|---|---|---|---|
| 60000000 | 64000000 | RCW | 128 |
| 60060000 | 64060000 | Bootscript | 128 |
| 60080000 | 64080000 | ESBC U-Boot HEADER | 128 |
| 600A0000 | 640A0000 | Bootscript Header | 128 |
| 60A00000* | 64A00000* | Kernel FIT Image | 54272 |
| 63F40000 | 64F40000 | kernel Header | 128 |

[10] Address to be used for loading the images in case of working in Development mode with Non-Secure Boot images on Bank 0.
[11] Address to be used for loading the images in case of working in Development mode with Non-Secure Boot images on Bank 0.

NOTE

* For LS1043 Bootscript, kernel image must be copied to DDR address 0x81000000 before issuing esbc_validate command.

The Boot script for LS1043 will be

```
# Copy the Kernel Image from Flash to DDR
cp.b 0x60A00000 0x81000000 0x1000000

# Validate the Kernel Image (The header has Image address as
0x81000000)
esbc_validate 0x63F40000

# Boot the validated Kernel FIT Image.
bootm $img_addr
```

NOTE

In LS1043, there is an issue currently with 'source <bootscript>' command.

As a workaround, the U-Boot code has been modified and the commands for bootscript mentioned above are part of U-Boot. So there is no need to place bootscript till the issue is fixed. The user can just place the Kernel image and its corresponding header on flash.

## P3/P5 NAND SECURE BOOT

Table 60: Memory Map for P3/P5 NAND SECURE BOOT

| Description (Chain of Trust) | Description (Chain of Trust with Confidentiality) | Address on NAND | Address on DDR (Image copied to from NAND) | Size |
|---|---|---|---|---|
| PBL.bin (RCW, PBI, U-Boot, U-boot Header) | PBL.bin (RCW, PBI, U-Boot, U-boot Header) | 0x00000000 | -- | 0xD0000 |
| Boot Script Header | Boot Script Header | 0x00800000 | 0x00010000 | 0x2000 |
| Boot Script | Boot Script | 0x00802000 | 0x00012000 | 0x2000 |
| uImage Header | | 0x00804000 | 0x00014000 | 0x2000 |
| dtb Header | | 0x00806000 | 0x00016000 | 0x2000 |
| rootfs Header | | 0x00808000 | 0x00018000 | 0x2000 |
| uImage | uImage | 0x06500000 | 0x01000000 | 0x410000 |
| dtb | dtb | 0x06b00000 | 0x00c00000 | 0x9000 |
| rootfs | rootfs | 0x02000000 | 0x02000000 | 0x2000000 |
| | uImage (Encapsulated) | 0x06500000 | 0x10000000 | 0x410000 |
| | dtb (Encapsulated) | 0x06b00000 | 0x10000000 | 0x9000 |
| | rootfs (Encapsulated) | 0x02000000 | 0x10000000 | 0x2000000 |

## 34.15 Useful U-Boot and CCS Commands

This section contains some useful commands for loading images via U-Boot (As per memory map defined in this document) and CCS commands to load the SRK Hash in shadow registers and get the core out of boot hold off in Development mode.

NOTE

The CCS commands to connect and configure config chain might change with Board/ Silicon Revision. Kindly refer the board manual/CCS Guide in case of issues with CCS commands.**The commands provided below are for reference only.** These will assist users in writing to SFP mirror registers and getting the core out of Boot Hold Off.

NOTE

For permanently blowing the values (OTPMK, SRK HASH etc.) in SFP, refer to the details in **Trust Arch. User Guide.** A brief summary of the steps is described below:

1. Ensure that PROG_SFP/POVDD signal is correctly asserted. This is usually controlled via a switch or a jumper. (Refer Board schematic/manual for the same)
2. Write the required fuse values to the SFP mirror registers.
3. To permamnetly blow the fuses, write to 'PROGFB' in SFP Instruction Register (SFP_INGR).

   Make sure that the values written in SFP mirror registers are correct before blowing the fuses as once blown, the fuse values cannot be changed.

   - To check if OTPMK is blown or not on the Silicon, check the bit 'OTPMK_ZERO' in the SECMON_HPSR register. If the bit is set, it means OTPMK is zero i.e. OTPMK needs to be blown.
   - The OTPMK value can be generated using 'gen_otpmk_drbg' tool provided in CST.
   - After writing the values in SFP mirror registers, check the hamming error register. Ensure that the value is 0x0 i.e. no error is reported.
     - Hamming error is reported in SecMon_HP Status Register (HPSR) for P3/P4/P5.
     - For other SoC's, it is reported in SFP Secret Value Hamming Error Status Register (SFP_SVHESR).

T1/T2/T4

```
protect off all;
setenv dir <tftp_path>
tftp 1000000 $dir/rcw.bin; erase EC000000 +$filesize; cp.b 1000000 EC000000 $filesize;
tftp 1000000 $dir/hdr_uboot.out;erase ECB00000 +$filesize; cp.b 1000000
ECB00000 $filesize;
tftp 1000000 $dir/u-boot.bin;erase EBF40000 +$filesize; cp.b 1000000 EBF40000 $filesize ;

tftp 1000000 $dir/hdr_bs.out;erase 0xECE00000 +$filesize;cp.b 1000000
0xECE00000 $filesize;
tftp 1000000 $dir/hdr_linux.out;erase 0xED000000 +$filesize;cp.b 1000000
0xED000000 $filesize;
tftp 1000000 $dir/hdr_rootfs.out;erase 0xED200000 +$filesize;cp.b 1000000
0xED200000 $filesize;
tftp 1000000 $dir/hdr_dtb.out;erase 0xED100000 +$filesize;cp.b 1000000
0xED100000 $filesize;
```

```
tftp 1000000 $dir/rootfs;erase 0xED300000 +$filesize;cp.b 1000000 0xED300000 $filesize;
tftp 1000000 $dir/bootscript;erase 0xECA00000 +$filesize;cp.b 1000000
0xECA00000 $filesize;
tftp 1000000 $dir/uImage.bin;erase 0xEC020000 +$filesize;cp.b 1000000
0xEC020000 $filesize;
tftp 1000000 $dir/uImage.dtb;erase 0xEC800000 +$filesize;cp.b 1000000
0xEC800000 $filesize;
```

```
# Connect to CCS and configure Config Chain
ccs::config_chain {t4240 j2i2cs}
display ccs::get_config_chain

#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem  0 0xfe314014 4 0 1
ccs::display_mem  0 0xfe0e0200 4 0 4

#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem 0 0xfe0e823c 4 0 <SRKH1>
ccs::write_mem 0 0xfe0e8240 4 0 <SRKH2>
ccs::write_mem 0 0xfe0e8244 4 0 <SRKH3>
ccs::write_mem 0 0xfe0e8248 4 0 <SRKH4>
ccs::write_mem 0 0xfe0e824c 4 0 <SRKH5>
ccs::write_mem 0 0xfe0e8250 4 0 <SRKH6>
ccs::write_mem 0 0xfe0e8254 4 0 <SRKH7>
ccs::write_mem 0 0xfe0e8258 4 0 <SRKH8>

#Get the Core Out of Boot Hold-Off
ccs::write_mem 0 0xfe0e00e4 4 0 0x00000001
```

B4

```
protect off all;
setenv dir <tftp_path>
tftp 1000000 $dir/rcw.bin; erase EE000000 +$filesize; cp.b 1000000 EE000000 $filesize;
tftp 1000000 $dir/hdr_uboot.out;erase EEB00000 +$filesize; cp.b 1000000
EEB00000 $filesize;
tftp 1000000 $dir/u-boot.bin;erase EDF40000 +$filesize; cp.b 1000000 EDF40000 $filesize ;

tftp 1000000 $dir/hdr_bs.out;erase 0xEEC00000 +$filesize;cp.b 1000000
0xEEC00000 $filesize;
tftp 1000000 $dir/hdr_linux.out;erase 0xEED00000 +$filesize;cp.b 1000000
0xEED00000 $filesize;
tftp 1000000 $dir/hdr_rootfs.out;erase 0xEEE00000 +$filesize;cp.b 1000000
0xEEE00000 $filesize;
tftp 1000000 $dir/hdr_dtb.out;erase 0xEEF00000 +$filesize;cp.b 1000000
0xEEF00000 $filesize;

tftp 1000000 $dir/rootfs;erase 0xEC020000 +$filesize;cp.b 1000000 0xEC020000 $filesize;
tftp 1000000 $dir/bootscript;erase 0xEE700000 +$filesize;cp.b 1000000
0xEE700000 $filesize;
tftp 1000000 $dir/uImage.bin;erase 0xEE020000 +$filesize;cp.b 1000000
0xEE020000 $filesize;
tftp 1000000 $dir/uImage.dtb;erase 0xEE800000 +$filesize;cp.b 1000000
0xEE800000 $filesize;
```

```
# Connect to CCS and configure Config Chain
ccs::config_chain {b4860 j2i2cs}
```

```
display ccs::get_config_chain

#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem  0 0xfe314014 4 0 1
ccs::display_mem  0 0xfe0e0200 4 0 4

#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem 0 0xfe0e823c 4 0 <SRKH1>
ccs::write_mem 0 0xfe0e8240 4 0 <SRKH2>
ccs::write_mem 0 0xfe0e8244 4 0 <SRKH3>
ccs::write_mem 0 0xfe0e8248 4 0 <SRKH4>
ccs::write_mem 0 0xfe0e824c 4 0 <SRKH5>
ccs::write_mem 0 0xfe0e8250 4 0 <SRKH6>
ccs::write_mem 0 0xfe0e8254 4 0 <SRKH7>
ccs::write_mem 0 0xfe0e8258 4 0 <SRKH8>

#Get the Core Out of Boot Hold-Off
ccs::write_mem 0 0xfe0e00e4 4 0 0x00000001
```

## P3/P4/P5

```
protect off all;
setenv dir <tftp_path>
tftp 1000000 $dir/rcw.bin; erase EC000000 +$filesize; cp.b 1000000 EC000000 $filesize;
tftp 1000000 $dir/hdr_uboot.out;erase ECB00000 +$filesize; cp.b 1000000
ECB00000 $filesize;
tftp 1000000 $dir/u-boot.bin;erase EBF40000 +$filesize; cp.b 1000000 EBF40000 $filesize ;

tftp 1000000 $dir/hdr_bs.out;erase 0xECE00000 +$filesize;cp.b 1000000
0xECE00000 $filesize;
tftp 1000000 $dir/hdr_linux.out;erase 0xED000000 +$filesize;cp.b 1000000
0xED000000 $filesize;
tftp 1000000 $dir/hdr_rootfs.out;erase 0xED200000 +$filesize;cp.b 1000000
0xED200000 $filesize;
tftp 1000000 $dir/hdr_dtb.out;erase 0xED100000 +$filesize;cp.b 1000000
0xED100000 $filesize;

tftp 1000000 $dir/rootfs;erase 0xED300000 +$filesize;cp.b 1000000 0xED300000 $filesize;
tftp 1000000 $dir/bootscript;erase 0xECA00000 +$filesize;cp.b 1000000
0xECA00000 $filesize;
tftp 1000000 $dir/uImage.bin;erase 0xEC020000 +$filesize;cp.b 1000000
0xEC020000 $filesize;
tftp 1000000 $dir/uImage.dtb;erase 0xEC800000 +$filesize;cp.b 1000000
0xEC800000 $filesize;
```

```
# Connect to CCS and configure Config Chain
ccs::config_chain p3040
display ccs::get_config_chain

#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem  0 0xfe314014 4 0 1
ccs::display_mem  0 0xfe0e0200 4 0 4

#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem 0 0xfe0e807c 4 0 <SRKH1>
ccs::write_mem 0 0xfe0e8080 4 0 <SRKH2>
ccs::write_mem 0 0xfe0e8084 4 0 <SRKH3>
ccs::write_mem 0 0xfe0e8088 4 0 <SRKH4>
ccs::write_mem 0 0xfe0e808c 4 0 <SRKH5>
```

```
ccs::write_mem 0 0xfe0e8090 4 0 <SRKH6>
ccs::write_mem 0 0xfe0e8094 4 0 <SRKH7>
ccs::write_mem 0 0xfe0e8098 4 0 <SRKH8>

#Get the Core Out of Boot Hold-Off
ccs::write_mem 0 0xfe0e00e4 4 0 0x00000001
```

LS1020

```
protect off all;
setenv path <tftp_path>
tftp 80000000 $path/rcw.bin;erase 64000000 +$filesize;cp.b 80000000 64000000 $filesize;
tftp 80000000 $path/hdr_uboot.out;erase 64080000 +$filesize;cp.b 80000000
64080000 $filesize;
tftp 80000000 $path/u-boot.bin;erase 64100000 +$filesize;cp.b 80000000
64100000 $filesize;

tftp 80000000 $path/hdr_bs.out;erase 640A0000 +$filesize;cp.b 80000000
640A0000 $filesize;
tftp 80000000 $path/bootscript;erase 64060000 +$filesize;cp.b 80000000
64060000 $filesize;

tftp 80000000 $path/hdr_dtb.out;erase 67F20000 +$filesize;cp.b 80000000
67F20000 $filesize;
tftp 80000000 $path/uImage.dtb;erase 64020000 +$filesize;cp.b 80000000
64020000 $filesize;

tftp 80000000 $path/hdr_linux.out;erase 67F00000 +$filesize;cp.b 80000000
67F00000 $filesize;
tftp 80000000 $path/uImage.bin;erase 64200000 +$filesize;cp.b 80000000
64200000 $filesize;

tftp 80000000 $path/hdr_rootfs.out;erase 67F40000 +$filesize;cp.b 80000000
67F40000 $filesize;
tftp 80000000 $path/rootfs;erase 64a00000 +$filesize;cp.b 80000000 64a00000 $filesize;
```

```
# Connect to CCS and configure Config Chain
ccs::config_server 0 10000
ccs::config_chain {ls1020a dap sap2}
display ccs::get_config_chain

#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem  <dap chain pos> 0x1e90014 4 0 4
ccs::display_mem  <dap chain pos> 0x1ee0200 4 0 4

#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem <dap chain pos> 0x1e80254 4 0 <SRKH1>
ccs::write_mem <dap chain pos> 0x1e80258 4 0 <SRKH2>
ccs::write_mem <dap chain pos> 0x1e8025c 4 0 <SRKH3>
ccs::write_mem <dap chain pos> 0x1e80260 4 0 <SRKH4>
ccs::write_mem <dap chain pos> 0x1e80264 4 0 <SRKH5>
ccs::write_mem <dap chain pos> 0x1e80268 4 0 <SRKH6>
ccs::write_mem <dap chain pos> 0x1e8026c 4 0 <SRKH7>
ccs::write_mem <dap chain pos> 0x1e80270 4 0 <SRKH8>

#Get the Core Out of Boot Hold-Off
ccs::write_mem <dap chain pos> 0x1ee00e4 4 0 0x00000001
```

LS1043

```
protect off all;
setenv path <tftp_path>
tftp 80000000 $path/rcw.bin;erase 64000000 +$filesize;cp.b 80000000 64000000 $filesize;
tftp 80000000 $path/hdr_uboot.out;erase 64080000 +$filesize;cp.b 80000000
64080000 $filesize;
tftp 80000000 $path/u-boot.bin;erase 64100000 +$filesize;cp.b 80000000
64100000 $filesize;

tftp 80000000 $path/hdr_bs.out;erase 640A0000 +$filesize;cp.b 80000000
640A0000 $filesize;
tftp 80000000 $path/bootscript;erase 64060000 +$filesize;cp.b 80000000
64060000 $filesize;

tftp 80000000 $path/hdr_kernel.out;erase 67F40000 +$filesize;cp.b 80000000
67F40000 $filesize;
tftp 80000000 $path/kernel.itb;erase 64a00000 +$filesize;cp.b 80000000
64a00000 $filesize;
```

```
# Connect to CCS and configure Config Chain
ccs::config_server 0 10000
ccs::config_chain {ls1043a dap sap2}
display ccs::get_config_chain

#Check Initial SNVS State and Value in SCRATCH Registers
ccs::display_mem  <dap chain pos> 0x1e90014 4 0 4
ccs::display_mem  <dap chain pos> 0x1ee0200 4 0 4

#Wrie the SRK Hash Value in Mirror Registers
ccs::write_mem <dap chain pos> 0x1e80254 4 0 <SRKH1>
ccs::write_mem <dap chain pos> 0x1e80258 4 0 <SRKH2>
ccs::write_mem <dap chain pos> 0x1e8025c 4 0 <SRKH3>
ccs::write_mem <dap chain pos> 0x1e80260 4 0 <SRKH4>
ccs::write_mem <dap chain pos> 0x1e80264 4 0 <SRKH5>
ccs::write_mem <dap chain pos> 0x1e80268 4 0 <SRKH6>
ccs::write_mem <dap chain pos> 0x1e8026c 4 0 <SRKH7>
ccs::write_mem <dap chain pos> 0x1e80270 4 0 <SRKH8>

#Get the Core Out of Boot Hold-Off
ccs::write_mem <dap chain pos> 0x1ee00e4 4 0 0x00000001
```

## 34.16 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 | |

*Table continues on the next page...*

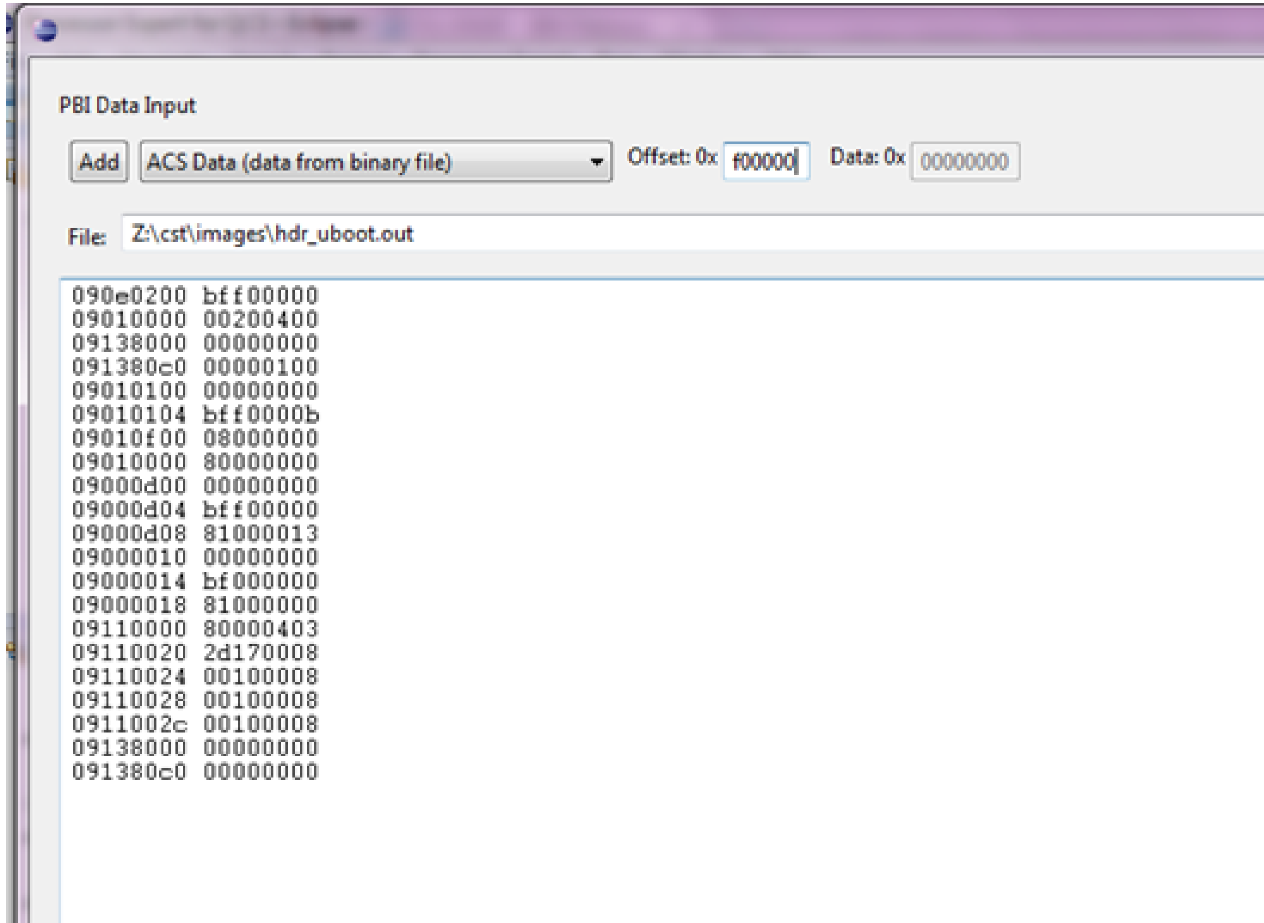
*Table continued from the previous page...*

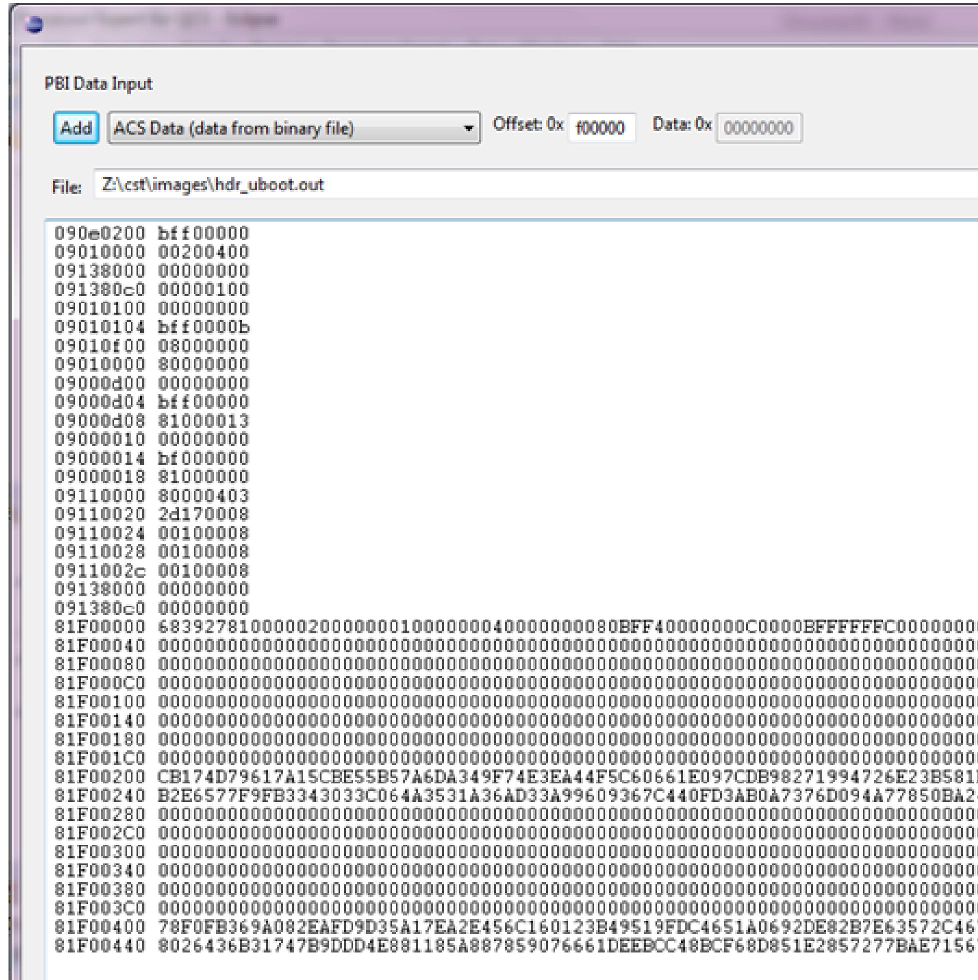
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| P4080 rev1 | 1.0 | 1.0 | 1.5 V | A | None/ Simulation | 1 | SNVS | SecMon_HP Status (HPSR) |
| P4080 rev2 | 1.0 | 1.1 | 1.5 V | B | None/ Simulation | 1 | SNVS | |
| P4080 rev3 | 1.0 | 2.2 | 1.5 V | B | None/ Simulation | 1 | SNVS | |
| P3041 | 1.1 | 2.0 | 1.5 V | B | None/ Simulation | 1 | SNVS | |
| P5020 | 1.1 | 2.0 | 1.5 V | B | None/ Simulation | 1 | SNVS | |
| P5040 | 1.1 | 2.2 | 1.5 V | B | None/ Simulation | 1 | SNVS | |
| P5021 | 1.1 | 2.2 | 1.5 V | B | None/ Simulation | 1 | SNVS | |
| T4240 rev1 | 2.0 | 3.1 | 1.89 V | A | SFP | 2 | SFP | SFP Secret Value Hamming Error Status Register (SFP_SVHESR) |
| T4240 rev2 | 2.0 | 3.2 | 1.89 V | A | SFP | 2 | SFP | |
| B4860 rev1 | 2.0 | 3.1 | 1.89 V | A | SFP | 2 | SFP | |
| B4860 rev2 | 2.0 | 3.2 | 1.89 V | A | SFP | 2 | SFP | |
| T2080 | 2.0 | 3.2 | 1.89 V | A | SFP | 2 | SFP | |
| T1040 | 2.0 | 3.2 | 1.89 V | A | SFP | 2 | SFP | |
| T1023 | 2.0 | 3.2 | 1.89 V | A | SFP | 2 | SFP | |
| LS1020A | 2.1 | 3.3 | 1.89 V | A | SFP | 2 | SFP | |
| LS1043A | 2.1 | 3.3 | 1.89 V | A | SFP | 2 | SFP | |

## 34.17 Using QCVS Tool (Secure Boot From NAND)

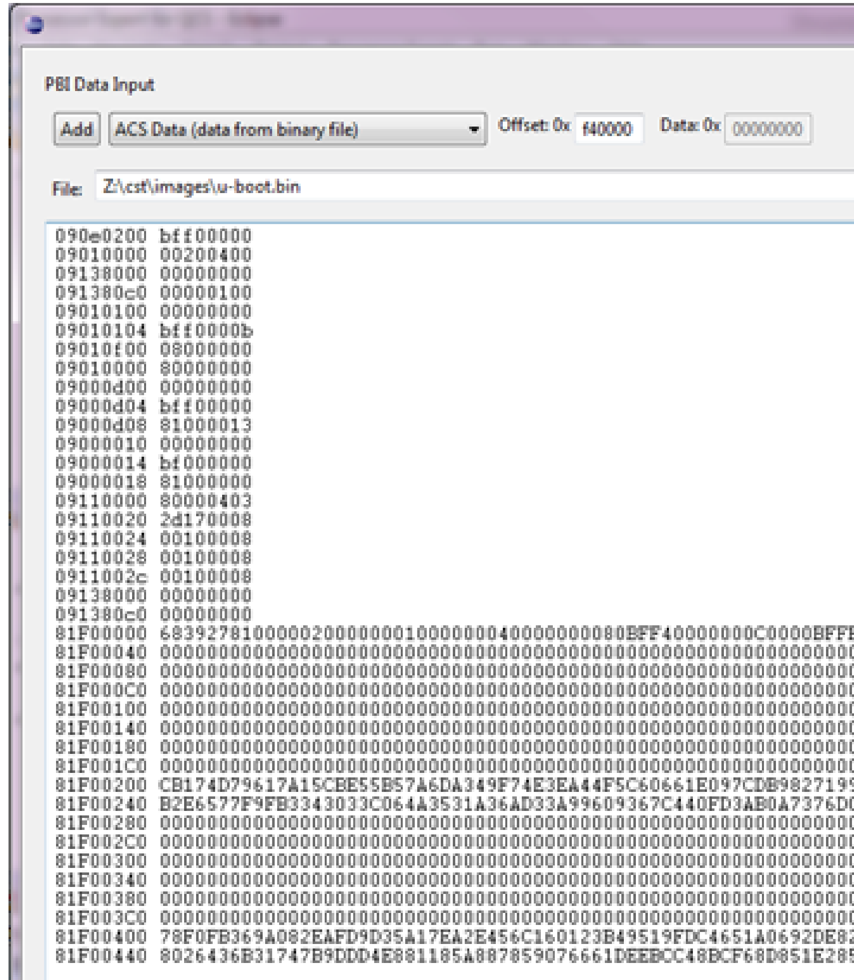
Use Freescale's QCVS tool for adding the hdr_uboot.out and u-boot.bin in terms of ALT_CONFIG_WRITE PBI commands at required addresses. The below screenshots describe the usage of QCVS Tool.

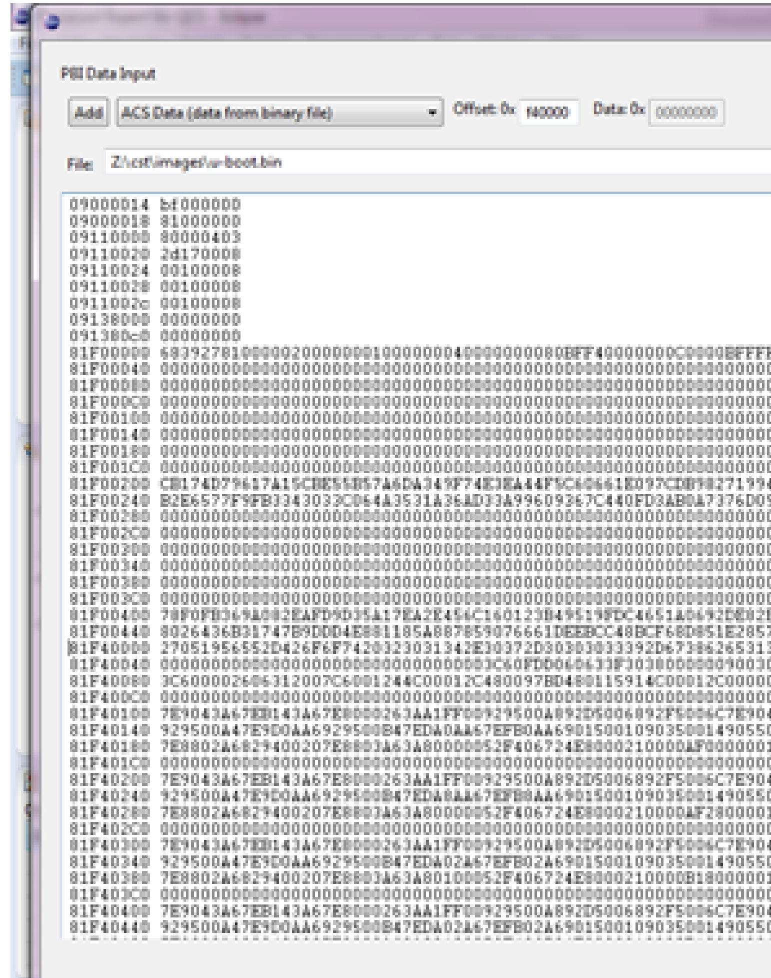
1. Import rcw.bin from SDK in QCVS Tool.
2. Add ACS Data hdr_uboot.out @ 0xF00000.


PBI Data Input
Add
ACS Data (data from binary file)
Offset: 0x
f00000
Data: 0x
00000000
File:
Z:\cst\images\hdr_uboot.out
090e0200 bff00000
09010000 00200400
09138000 00000000
091380c0 00000100
09010100 00000000
09010104 bff0000b
09010f00 08000000
09010000 80000000
09000d00 00000000
09000d04 bff00000
09000d08 81000013
09000010 00000000
09000014 bf000000
09000018 81000000
09110000 80000403
09110020 2d170008
09110024 00100008
09110028 00100008
0911002c 00100008
09138000 00000000
091380c0 00000000

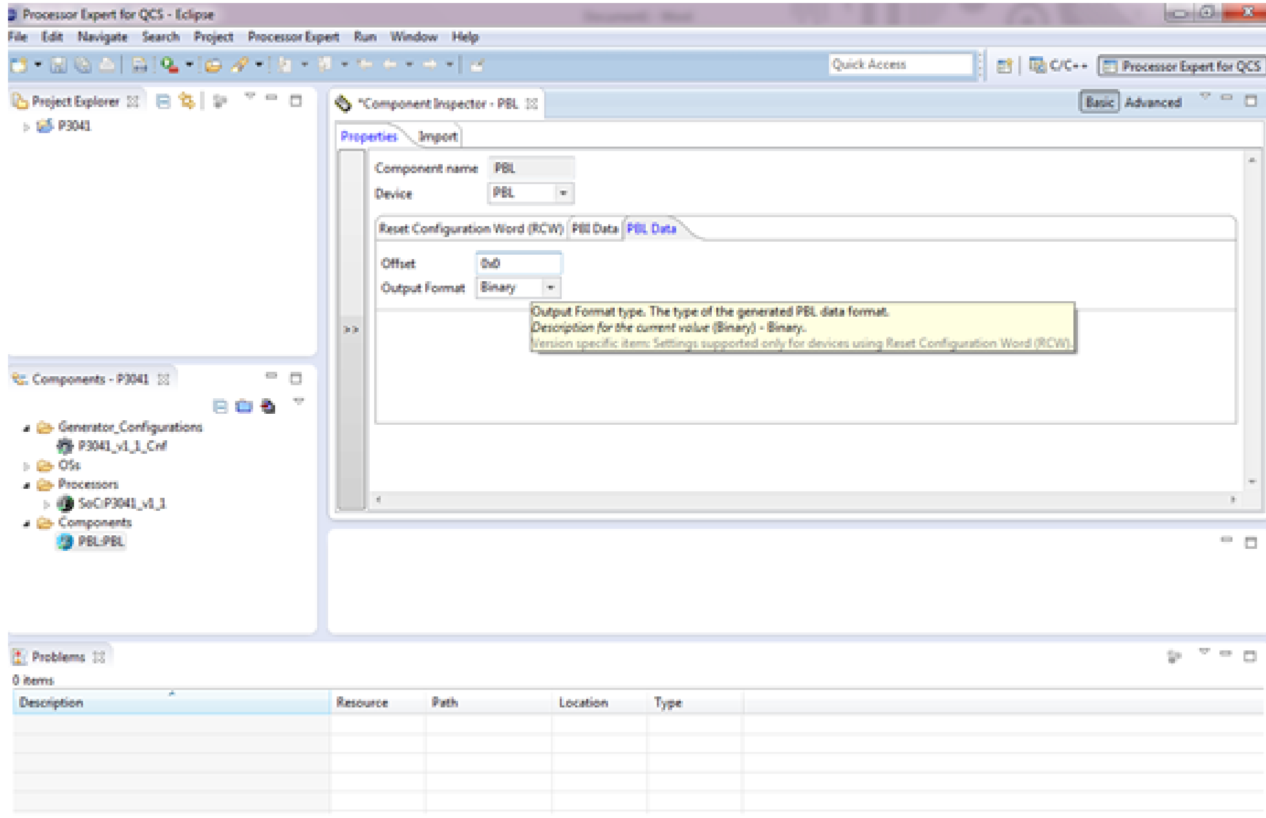



3. Add ACS Data u-boot.bin @0xF40000.

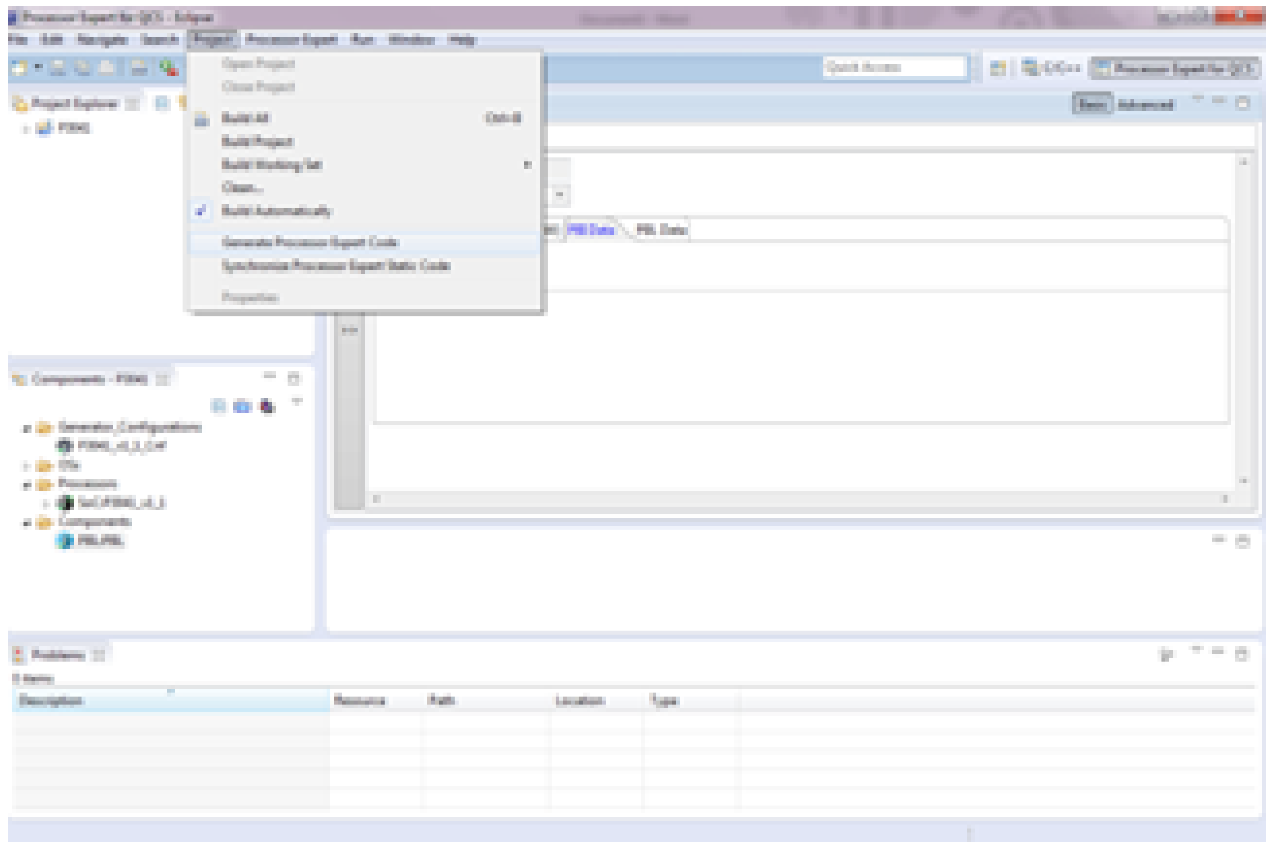

PBI Data Input
Add
ACS Data (data from binary file)
Offset: 0x f40000
Data: 0x 00000000
File: Z:\cst\images\u-boot.bin
090e0200 bff00000
09010000 00200400
09138000 00000000
091380c0 00000100
09010100 00000000
09010104 bff0000b
09010f00 08000000
09010000 80000000
09000d00 00000000
09000d04 bff00000
09000d08 81000013
09000010 00000000
09000014 bf000000
09000018 81000000
09110000 80000403
09110020 2d170008
09110024 00100008
09110028 00100008
0911002c 00100008
09138000 00000000
091380c0 00000000




4. Make Sure that the output format is Binary.




5. Generate Processor Expert Code to get PBL.bin.



.

# Chapter 35
# Universal Serial Bus Interfaces

## 35.1 USB 3.0 Host/Peripheral Linux Driver User Manual

### Description

#### Main features of xHCI controller

- Supports operation as a standalone USB xHCI host controller
- USB dual-role operation and can be configured as host or device
- Super-speed (5 GT/s), High-speed (480 Mbps), full-speed (12 Mbps), and low-speed (1.5 Mbps) operations
- Supports operation as a standalone single port USB
- Supports eight programmable, bidirectional USB endpoints
- OTG (On-The-Go) 2.0 compliant, which includes both device and host capability.

#### Modes of Operation

- Host Mode: SS/HS/FS/LS
- Device Mode: SS/HS/FS
- OTG: HS/FS/LS

NOTE

**Super-speed operation is not supported when OTG is enabled**

NOTE

This document explains working of **HS Host and HS Device** in Linux

#### Specifications

| Hardware: | on-board xhci usb controller |
|---|---|
| Software: | 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--->` | Enables USB host controller support |

*Table continues on the next page...*

*Table continued from the previous page...*

<table>
<tr><th>Kernel Configure Tree View Options</th><th>Description</th></tr>
<tr><td><pre>USB support --->

[*] Support for Host-side USB</pre></td><td></td></tr>
<tr><td><pre>Device Drivers--->

 USB support --->
 <*>     xHCI HCD (USB 3.0) support</pre></td><td>Enables XHCI Host Controller Driver and transaction translator</td></tr>
<tr><td><pre>Device Drivers--->

 USB support --->
 <*>     USB Mass Storage support

 [ ]       USB Mass Storage verbose debug</pre></td><td>Enable support for USB mass storage devices. This is the driver needed for USB flash devices, and memory sticks</td></tr>
<tr><td><pre>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)</pre></td><td>Note: Required only for USB Gadget/Peripheral Support<ul><li>Enable driver for peripheral/ device controller</li><li>Enable Ethernet Gadget Client driver</li><li>Enable Mass Storage Client driver</li></ul></td></tr>
<tr><td><pre>Device Drivers--->

 <*>   DesignWare USB3 DRD Core Support</pre></td><td>Enable XHCI DRD Core Support</td></tr>
</table>

### Compile-time Configuration Options

| Option | Values | Default Value | Description |
|---|---|---|---|
| CONFIG_USB | y/m/n | y | Enables USB host controller |

*Table continues on the next page...*

*Table continued from the previous page...*

| Option | Values | Default Value | Description |
|---|---|---|---|
| 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 |

### Source Files

The driver source is maintained in the Linux kernel source tree in below files

*Table continued from the previous page...*

| Source File | Description |
|---|---|
| | |
| drivers/usb/host/xhci-* | xhci platform driver |
| drivers/usb/gadget/mass_storage.c | USB Mass Storage |
| drivers/usb/gadget/ether.c | Ethernet gadget driver |

### Device Tree Binding for Host

```
usb@3100000 {
               compatible = "snps,dwc3";
               reg = <0x0 0x3100000 0x0 0x10000>;
               interrupts = <GIC_SPI 93 IRQ_TYPE_LEVEL_HIGH>;
              dr_mode = "host;
            };
```

### Device Tree Binding for Peripheral

```
usb@3100000 {
               compatible = "snps,dwc3";
               reg = <0x0 0x3100000 0x0 0x10000>;
               interrupts = <GIC_SPI 93 IRQ_TYPE_LEVEL_HIGH>;
              dr_mode = "peripheral;
               maximum-speed = "super-speed";
          };
```

### Host Testing

```
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
```

```
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.
```

```
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 /$
```

```
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.

```
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
```

```
# cat /proc/device-tree/soc/usb\@3100000/dwc3/dr_mode
peripheralroot@ls1021aqds:~#
```

```
fs/configfs/configfs.ko
driver/usb/gadget/libcomposite.ko
driver/usb/gadget/g_mass_storage.ko
driver/usb/gadget/u_rndis.ko
driver/usb/gadget/u_ether.ko
driver/usb/gadget/usb_f_ecm.ko
driver/usb/gadget/usb_f_ecm_subset.ko
driver/usb/gadget/usb_f_rndis.ko
driver/usb/gadget/g_ether.ko
```

### Mass Storage Gadget

```
To use ramdisk as a backing store use the following

root@ls1021aqds:/home# mkdir /mnt/ramdrive
```

```
root@ls1021a~# mount -t tmpfs tmpfs /mnt/ramdrive -o size=600M
roo~# dd if=/dev/zero of=/mnt/ramdrive/vfat-file bs=1M count=500
root@ls1021aqds:/home# mke2fs -F /mnt/ramdrive/vfat-file
root@ls1021aqds:/home# insmod configfs.ko
root@ls1021aqds:/home# insmod libcomposite.ko
ro~# 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
```

### Ethernet Gadget

```
To use Ethernet gadget use the following

root@ls1021aqds:/home# insmod configfs.ko
root@ls1021aqds:/home# insmod libcomposite.ko
root@ls1021aqds:/home# insmod u_ether.ko
root@ls1021aqds:/home# insmod u_rndis.ko
root@ls1021aqds:/home# insmod usb_f_ecm.ko
root@ls1021aqds:/home# insmod usb_f_ecm_subset.ko
root@ls1021aqds:/home# insmod usb_f_rndis.ko
root@ls1021aqds:/home# insmod g_ether.ko
```

We will get below messages

```
[   28.692611] using random self ethernet address
[   28.697156] using random host ethernet address
[   28.694271] usb0: HOST MAC 82:96:69:7e:a5:7d
[   28.698928] usb0: MAC 72:00:a5:80:2b:e8
[   28.692586] using random self ethernet address
[   28.697080] using random host ethernet address
[   28.691368] g_ether gadget: Ethernet Gadget, version: Memorial Day 2008
[   28.698028] g_ether gadget: g_ether ready
```

Make sure USB0 ethernet interface is available after this

```
root@ls1021aqds:/home# ifconfig -a
can0      Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
          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@ls1021aqds:/home# ifconfig usb0 10.232.1.11
root@ls1021aqds:/home#
root@ls1021aqds:/home#
```

```
root@ls1021aqds:/home# ping usb   10.232.1.10
PING 10.232.1.10 (10.232.1.10): 56 data bytes
64 bytes from 10.232.1.10: seq=0 ttl=128 time=5.294 ms
64 bytes from 10.232.1.10: seq=1 ttl=128 time=6.101 ms
64 bytes from 10.232.1.10: seq=2 ttl=128 time=4.170 ms
64 bytes from 10.232.1.10: seq=3 ttl=128 time=4.233 ms
```

### Known Bugs, Limitations, or Technical Issues

- Some issue with Pen drives from Kingston/Transcend. This have noticed some patches floating in open-source for these issues, and also found that open-source USB community trying to fix.
- Linux allow only one peripheral at one time. Please make sure When DWC3 set as Peripheral the other should not be set in same mode.
- Erratum:A-009116 (Frame length of USB3 controller for USB2.0 and USB3.0 operation is incorrect) impacts some socs like LS1020A/LS1021A because of which some USB2.0 and USB3.0 devices may not work properly, and hence, a sw workaround is needed. This sw workaround involves programing following registers of XHCI controller as: GFLADJ[5:0] = 20H and GFLADJ[7] = 1. This is already done via u-boot and linux codebase.

### Supporting Documentation

· N/A

# Chapter 36 USDPAA User Guide

## 36.1 Introduction

The Freescale Data Path Acceleration Architecture comprises a set of hardware components which are integrated via a hardware queue manager and use a common hardware buffer manager. Software accesses the DPAA via hardware components called "software portals". These directly provide queue and buffer manager operations such as enqueues, dequeues, buffer allocations, and buffer releases and indirectly provide access to all of the other DPAA hardware components via the queue manager.

This document describes the User Space Datapath Acceleration Architecture (USDPAA) software. USDPAA is a software framework that permits Linux user space applications to directly access the DPAA queue and buffer manager software portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

### 36.1.1 Intended audience

This document is intended for software developers and system architects who work with the USDPAA framework.

The material is technical in nature. The reader is assumed to be familiar with

- General Linux software development, operation, and configuration-- for Power architecture devices in particular.
- The concept of device drivers and their role in operating systems.
- Linux network administration and use.
- The Freescale Linux SDK for DPAA-based SoCs.
- At least broadly, the DPAA hardware blocks.

The P4080 was the first Freescale SoC to incorporate the DPAA. As such, it is used in many examples. However, USDPAA is intended to apply in the same manner to all DPAA-based SoCs.

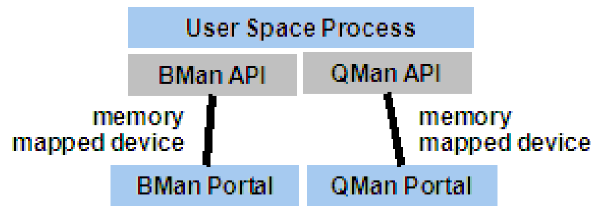
### 36.1.2 USDPAA overview

As mentioned above, USDPAA is an environment that allows the development of Linux user space applications that have direct access to software portals for the DPAA buffer manager (BMan) and the DPAA queue manager (QMan).

Software portals are memory mapped hardware components. The items within them (message ring, dequeue ring, etc.) have specific addresses in the SoC's physical address map just like regular memory does. USDPAA works by permitting the physical address ranges that correspond to software portals to be mapped into the virtual (technically "effective" in the Power architecture nomenclature) address space of user space processes.Thus, the user space application can use normal load and store instructions to perform operations on the portals.

Portals must be accessed using correct instruction sequences, and also the memory range for the portals must be mapped using the correct cache attributes. For this reason, the USDPAA software includes both the user space driver for the portals and also an access and control API packaged as a user space library.

Figure 54: C-Language APIs access memory-mapped devices




The USDPAA framework assumes the context of a single SMP Linux instance on an SoC such as the P4080 . There is no dependence on the Freescale Embedded Hypervisor, and this document assumes that it is not used.

Specifically, the SMP Linux instance may contain many user spaces processes. These processes can be (and normally are) multi-threaded using the pthreads facility of Linux. Define a "USDPAA thread" as a thread that has been allocated a BMan and a QMan software portal for its use via direct access. A "USDPAA process" is a Linux user space process that contains at least one USDPAA thread.

Clearly, the number of USDPAA threads is limited by the number of software portals that the particular SoC provides. The P4080 provides 10 QMan and 10 BMan software portals. The available portals must be divided into two sets: portals for use by the Linux kernel and portals for use by USDPAA threads.

## 36.1.3 USDPAA and legacy Linux software

This section describes the relationship between USDPAA and various types of legacy software.

### 36.1.3.1 Legacy user space applications

USDPAA provides user space processes access to DPAA functionality via user space device drivers and device-oriented C-language APIs layered upon them. Legacy applications must be modified in order to use these APIs. USDPAA's primary goal is to provide these APIs.

As an example, it is possible to offload cryptographic operations to the DPAA Security Engine (SEC) using USDPAA drivers, but this does not imply that all applications that use a cryptography-related legacy Linux facility are automatically accelerated. Software someplace (in user space) must explicitly use the USDPAA functions.

It is possible that some legacy user space facilities could be accelerated in a manner that automatically applies to legacy user space applications if means other than or in addition to USDPAA are used. This topic is, however, beyond the scope of the document.

### 36.1.3.2 Relationship to conventional kernel-based drivers

Conventional device drivers reside in the Linux kernel and are accessed via system calls such as read and write. These often (but not always) involve copying data between the application's address space and the kernel's address space. These types of drivers are desirable in many situations.

The existence of USDPAA neither implies nor excludes the existence of conventional drivers. It also implies nothing about whether or not conventional drivers copy data. Mostly, the topic of conventional drivers is beyond the scope of this document. Details will vary among DPAA IP blocks. A few points follow:

- SEC has a job queue interface that can be used concurrently with its QMan interface. The DPAA SDK contains kernel code that does this.
- As figure #unique_592 shows, The DPAA's Buffer Manager and Queue Manager in-kernel portal drivers support access by multiple entities such as the DPAA kernel ethernet driver. It is possible to create custom conventional kernel drivers that layer on top of the in-kernel portal drivers. These can provide traditional kernel-mediated device access.

## 36.2 USDPAA assumptions and use cases

The primary purpose of USDPAA is application performance. User space drivers can provide substantial performance improvement over traditional kernel-mediated access to devices.

- No system calls are needed to do I/O.
- This implies no need to switch into and out of the kernel's execution context.
- User space applications directly access data buffers, thus providing zero copy I/O in all cases.

Because the goal is performance, USDPAA provides simple but low-level access to I/O functionality. It does not employ complex and costly abstractions. Applications deal directly with QMan and BMan via their software APIs. These form a thin but helpful layer between the application and the hardware.

### 36.2.1 Assumptions

The USDPAA software makes certain assumptions and supports multiple use-cases. These are detailed in the sections that follow.

#### 36.2.1.1 General assumptions

- USDPAA threads should be made affine to a core. This is due to support for stashing to per-core caches. (Nothing breaks if this is not the case, but performance will be sub-optimal.)
- Every USDPAA thread has its own BMan and QMan software portals. No other thread or entity accesses these portals. Within the drivers, these are stored as thread-local variables and the locking assumptions depend on them being thread-private.
- USDPAA threads may make use of arbitrary Linux system calls. For example, they can do file I/O.
- USDPAA threads are compatible with general Linux services such as debuggers like gdb.
- Threading is via standard Linux pthreads and the standard Linux GNU C library.

### 36.2.2 Use cases

USDPAA is intended to support multiple use-cases, and the supported use-case set will grow with time.

#### 36.2.2.1 Run-to-completion

The full run-to-completion use case is defined by the following characteristics:

- The number of USDPAA threads per core must not exceed the number of QMan and BMan portals assigned to that core (and reserved for USDPAA use) within the device-tree. Note however that testing with more than one thread per core has been minimal. Not all cores need have a USDPAA thread, however.
- Cores hosting USDPAA threads may be configured to run nothing in user space other than that core's USDPAA thread. For example the kernel parameter "isolcpus" could be used - see CPU isolation on page 600 for more on this topic.
- The /proc/irq mechanism may be used to limit interrupts for miscellaneous peripherals to non-isolated cores. Again, USDPAA does not require this architecturally, but it is often done.
- In run-to-completion, USDPAA threads poll their portals. Polling generally implies that the USDPAA threads will always be running or ready to run. It would be unusual to have other threads scheduled on the same core when USDPAA threads are in this "non-interactive" mode.

- Often, one thinks of the USDPAA threads as "workers" able to process any form of "work" delivered via QMan messages. In this model, the QMan scheduler can be used as a general "work" scheduler rather than relying on the Linux scheduler for this purpose.

#### 36.2.2.2 Interrupt-driven

USDPAA supports an interrupt-driven model that allows benefits such as cooperating with other threads on the same core, potential power-saving by not polling all of the time, and so on. This model introduces more operating system overheads and thus trades performance and latency for these benefits.

- The user space drivers for BMan and QMan software portals are implemented using a Linux character device. It permits USDPAA threads to await interrupts from software portals by doing file operations (like "select()") using a file descriptor associated with the user space device. See section QMan and BMan drivers and C API. on page 595 for more details.
- USDPAA threads process dequeued frames from portals (as much as they like) after an interrupt indicates that data are available. When dequeue processing is complete, the thread re-enables the interrupt.
- This interrupt mechanism allows USDPAA threads to sleep awaiting I/O. Thus, other threads can execute while a particular USDPAA thread sleeps. In this model, it is normal it have multiple threads, USDPAA or otherwise, execute on the same core.
- This use-case is compatible with SCHED_FIFO Linux scheduling. USDPAA is independent of scheduling policy. It is also independent of real-time policy and patches such as PREEMPT_RT.

Application developers may chose the model, run-to-completion or interrupt-driven that best fits their need. The PPAC-based example applications in USDPAA (eg. reflector) implement a hybrid of both modes, where an interrupt-driven mode is used whenever the packet-processing has been idle for a short period of time, and switching back to run-to-completion once processing resumes.

## 36.3 USDPAA components

USDPAA consists of several components which may used in the context of standard SMP Linux on the Freescale family of DPAA-based SoCs. These components are described below.

### 36.3.1 Device-tree handling

Many configuration and resource details are defined within the "device-tree" used to boot Linux. The kernel itself uses this data structure to determine the system's devices and attributes, and USDPAA applications are dependent on the same information. As will be seen, the QMan and BMan portals that are available to USDPAA (and their attributes such as channel IDs) are declared within the device-tree, as are the ethernet interfaces and related attributes from the FMan side of things.

The USDPAA "of" driver[12] parses the device-tree information from the procfs filesystem exported by the Linux kernel (as found at /proc/device-tree/) and constructs an internal data-structure representation of the tree, including interrelationships between device-nodes (eg. portals and CPUs, pool-channels and portals, etc). This information is then used by driver and application-configuration layers where required to obtain device information.

#### 36.3.1.1 Device-tree initialization requirements

In the current "of" driver implementation, it is the application's responsibility to initialize this driver layer prior to initializing or using any other driver layers that may be dependent on it. This is achieved by calling the of_init()

[12] "OF" refers to Open Firmware, the specification behind device-trees, a subject beyond the scope of this document.

API. To satisfy leak-checkers or to support the use of persistent/reusable processes, an application can also undo all allocations and driver state by calling of_finish().

The "Queue Manager, Buffer Manager API Reference Manual" cover these APIs in more detail, if the above description (and the <usdpaa/of.h> header) are insufficient.

## 36.3.2 QMan and BMan drivers and C API.

The Queue Manager (QMan) and Buffer Manager (BMan) drivers are the heart of USDPAA. The two drivers are structurally similar in the broadest sense. This discussion will focus on QMan, but the material also applies to BMan.

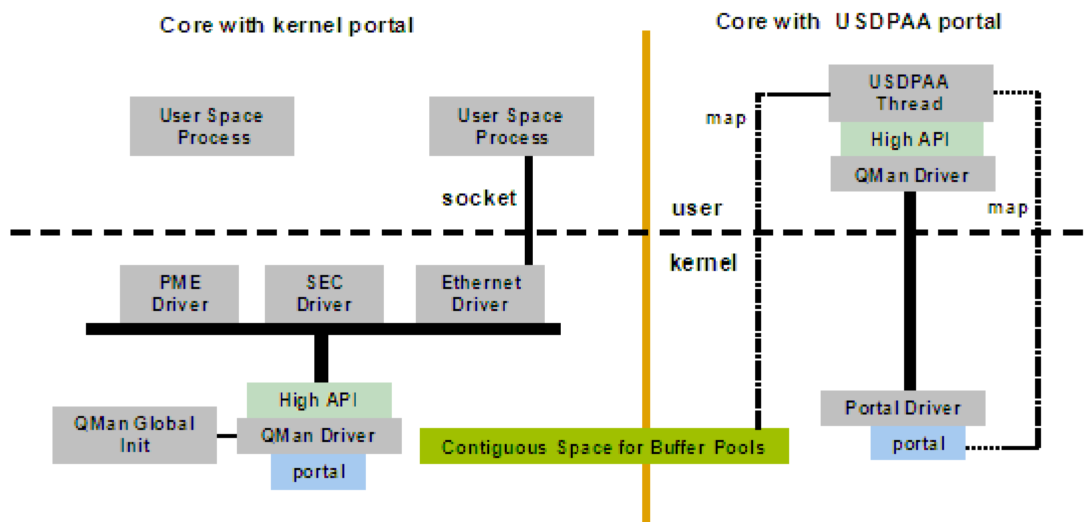
### 36.3.2.1 QMan driver overview

The QMan driver serves two roles:

1. Initialization and management of all aspects of QMan that are not per-portal, i.e. are global.
2. Initialization and management of per software portal QMan operations, i.e. operations that are local to the system component to which the portal is dedicated.

It is reasonable to think of the QMan driver as being split into parts: a QMan global driver and QMan software portal driver. There is only one instance of the global driver per SoC. It is always a kernel-level driver. There is an instance of the software portal driver for every QMan software portal that exists physically on the SoC. For example, P4080 provides 10 QMan software portals. Thus, there are 10 instances of the portal driver available.

Figure 55: QMan Driver Architecture




As the figure above shows, the software portals (and hence their driver instances) may be dedicated for kernel or or user space use. Essentially the same QMan portal API is presented for both the kernel and the user space instances. Indeed, the same software is used in user space and the kernel.

The kernel driver instances have existing "users" in the kernel. For example, the ethernet driver, the PME driver and (though not in the current release) the SEC driver all use the QMan API to share access to the same physical portal on a core. They all enqueue and queue on the same portal using the API associated with the portal's driver instance. Kernel developers are free to add additional users in the form of additional kernel-level components that share the same portal.

As an aside, the SEC is a special case because it has two interfaces that software can use: the job queue interface, and the QMan interface. With current versions of the DPAA SDK, kernel software uses the job queue interface. User space processes can use the SEC via QMan using their portals.

User space processes access QMan using software portals that are dedicated to user space. To a great extent the QMan API abstracts it, but this access is implemented using standard Linux character devices for user space drivers. It provides /dev entries for devices to be accessed from user space. A user space process opens such a device and requests that the device be mapped directly into the process's address space. The assumption is that the physical device is a conventional memory mapped peripheral - as QMan software portals are. At this point, the process can access the device hardware via normal load and store instructions.

QMan's hardware design requires that portals be mapped with the correct caching attributes for each part of the portal. In addition, careful instruction sequences are needed to ensure that portal operations are carried out correctly. This is due to the hardware's cache stashing support, which provides low-latency access to portals and data dequeued from portals.

The QMan API mentioned above abstracts this and provides software with a convenient method to perform portal operations such as enqueues and dequeues. The QMan API in user space is a library that is layered on top of the USDPAA character device infrastructure. For the user space driver instances, there is

- In user space:
  - The QMan API
  - Direct access to the portal hardware
- In the kernel:
  - The character device code to establish mappings
  - Handle interrupts

This C-callable API is documented in the "Queue Manager, Buffer Manager API Reference Manual" that is distributed with the DPAA SDK software.

One difference in usage of portals in user space versus the kernel is that kernel portals are expected to be persistent. They are brought into service and remain in service so upper layer kernel services such as the ethernet driver can assume that they are there. User space portals are, in contrast, dedicated to an application thread and need to be in service only as long as that thread is running. Thus, the time during which the portal is in service is determined by the application and all uses of the portal are determined by the application.

## 36.3.2.2 QMan portals and the Linux device-tree

Since QMan software portals can be dedicated to either the Linux kernel or user space, there must be a mechanism to indicate how each portal will be used.

Linux device trees describe physical resources that are available to Linux and provide information that allows Linux to select drivers for those devices. As such, there are entries in the device tree for all QMan software portals.

The device tree property "fsl,usdpaa-portal" indicates that a given portal is to be made available to user space; the absence of this property implies that the portal will be used only within the Linux kernel.

A listing of two QMan software portal device tree nodes follows. The first portal is used by the kernel. The second is made available to user space.

```
                qportal0: qman-portal@0 {
                        cell-index = <0x0>;
                        compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";
                        reg = <0x0 0x4000 0x100000 0x1000>;
                        cpu-handle = <&cpu0>;
                        interrupts = <104 0x2 0 0>;
                        fsl,qman-channel-id = <0x0>;
                        fsl,qman-pool-channels = <&qpool1 &qpool2 &qpool3>;
                };
                qportal1: qman-portal@4000 {
```

```
                cell-index = <0x1>;
                compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";
                fsl,usdpaa-portal;
                reg = <0x4000 0x4000 0x101000 0x1000>;
                cpu-handle = <&cpu1>;
                interrupts = <106 0x2 0 0>;
                fsl,qman-channel-id = <0x1>;
                fsl,qman-pool-channels = <&qpool4 &qpool5 &qpool6
                                          &qpool7 &qpool8 &qpool9
                                          &qpool10 &qpool11 &qpool12
                                          &qpool13 &qpool14 &qpool15>;
        };
```

There is no mechanism provided to determine what, if anything, user space software will do with a given user space portal. Initially, these portals are present but not initialized. Each user space portal has a /dev entry. It is a USDPAA driver decision as to which portals a given process or thread will use. A portal is placed into use when a user space process or thread requests it via the QMan API.

## 36.3.2.3 Note on the current implementation

In the current implementation, portals are bound to cores via the cpu-handle in the portal device tree node. Most commonly, a thread will make itself affine to its portal's core, because stashing for the portal will be targeting that core's cache. Functionality will not break if the thread executes on a different core, because coherency will be maintained, but at the expense of sub-optimal performance (and increased contention between the cores).

Expect more flexibility in this respect in later releases of the QMan software.

## 36.3.2.4 Portal initialization requirements

As described in Device-tree initialization requirements on page 594, the USDPAA "of" driver must be initialised prior to trying to initialize or use QMan or BMan. Once this is done, a pthread that wishes to be initialized with access to a QMan portal should call the qman_thread_init() API, which is defined in the <usdpaa/fsl_usd.h> header along with all other USDPAA-specific QMan/BMan APIs.

Please consult the "Queue Manager, Buffer Manager API reference" for more information.

## 36.3.2.5 Buffer Manager (BMan)

The BMan driver and its software support is very similar to QMan's. BMan software portals may be used in the kernel or in user space just like QMan portals. Just as with QMan, the same BMan software API is available in both the kernel and in user space. This API also is defined in the "Queue Manager, Buffer Manager API Reference Manual."

## 36.3.2.6 Raw Portal APIs

Raw portal APIs are available to allow USDPAA process to allocate QMan and BMan portals on behalf of another processor. The portals allocated by these APIs are not configured, it is the responsibility of the allocator to do any configuration of the portal that may be needed. The APIs are as follows:

- int qman_allocate_raw_portal(struct usdpaa_raw_portal *portal) - Allocates an unconfigured QMan Portal
- int qman_free_raw_portal(struct usdpaa_raw_portal *portal) - Releases a raw QMan portal that was previously allocated
- int bman_allocate_raw_portal(struct usdpaa_raw_portal *portal) - Allocates an unconfigured BMan portal
- int bman_free_raw_portal(struct usdpaa_raw_portal *portal) - Releases a raw BMan portal that was previously allocated

For detailed information on then usdpaa_raw_portal struct refer to the fsl_usd.h header file.

## 36.3.3 DMA memory management

The Freescale DPAA hardware provides several peripherals such as FMan, SEC, and PME that read and write memory directly using DMA. Buffers used for peripherals' DMA must be allocated from memory with special properties:

- The memory must be physically contiguous since the peripheral does not access memory via the core's MMU (or any page-mode I/O MMU).
- The memory must be addressable by the peripheral.
- The memory must not be swapped out by Linux while the device is accessing it.
- For user space drivers, there must be an efficient mechanism to convert the physical addresses used by the peripheral hardware to and from the effective addresses used in a user space process or thread's address space.
- For BMan and DPAA usage, it is often convenient for software to allocate memory from physically contiguous regions that are quite large.
- It is desirable to use the Power core's TLB1 mechanism to map large physically contiguous memory regions of this sort. This increases performance by reducing the number of MMU-related interrupts that must be processed. A single TLB0 entry can map only a single 4K page. A single TLB1 entry, in contrast, can map a large (but variable-sized) page. One TLB1 entry can do the work of many TLB0 entries in circumstances such as this one.

Memory that meets the criteria above is called DMA memory. Drivers for all DMA-capable devices must use DMA memory for buffers. This is true for both conventional in-kernel drivers and user space drivers. It is a hardware implication of peripherals' relationships to cores and MMUs.

### 36.3.3.1 Current USDPAA solution

The USDPAA Linux kernel reserves a contiguous region of memory very early in the kernel boot process for use as DMA memory. This is done prior to full initialization of the kernel's memory-management subsystem that subsequently inhibits such allocations. This memory reservation is of a size and alignment that is hard-coded into the kernel via the Kconfig option "CONFIG_FSL_USDPAA_SHMEM". Within the kernel's "make menuconfig" interface, this can be found under "Device Drivers" -> "Misc devices" -> "Freescale USDPAA shared memory driver". The default is for a 64MB memory reservation.

This same USDPAA kernel driver exposes the reserved memory range via a "/dev/fsl_usdpaa_shmem" character device, which allows the USDPAA application to mmap() the physically-contiguous memory range to a corresponding contiguous range in its virtual address space, thus facilitating trivial virtual/physical address conversions. Additionally, a hook is placed into the memory-management code to "catch" page faults within this memory range and ensure that they are resolved by a single TLB1 mapping that spans the entire memory reservation (rather than the usual 4KB TLB0 mappings). That is, once the USDPAA application has accessed any address within the DMA memory range, no more page faults should occur for any other access within the entire region. Given that USDPAA datapaths of less than 200 processor cycles per packet have been demonstrated, for traffic rates that can consume over 1MB of buffer space in less than a millisecond, we consider that incurring the overhead of page-fault handlers in the kernel when iterating across thousands of pages of memory would likely be too high a price to pay for high performance applications.

The 'fsl_usdpaa_shmem' driver does not automatically constrain the mmap() alignment, so due to the alignment requirements of TLB1 entries, the USDPAA application has to propose virtual base addresses to the kernel rather than letting it allocate them. This will be fixed in a future release. However all of this is encapsulated within the USDPAA "dma_mem" driver. This driver is initialized via the dma_mem_setup() API, which handles the loading and mapping of the DMA memory region. Furthermore, a hard-coded subset of the DMA memory is reserved for buffer pool usage in the USDPAA demo applications, and the rest of it is exposed for general purpose DMA-able memory allocation via the dma_mem_memalign() and dma_mem_free() APIs.

**Implementation Note**

The USDPAA QMan and BMan APIs always refer to buffers by physical address, as these are what are passed to and from DMA-enabled devices within the Datapath Architecture. So ultimately, any mechanism can be used to provide "DMA memory" to a USDPAA application so long as it allows the application easy access to the memory and an efficient mechanism for converting to and from physical addresses.

Performance considerations will probably also require that page-faults be minimized or eliminated in performance-critical scenarios. The USDPAA "dma_mem" driver (and underlying character device and TLB1 kernel hook) is just one way to achieve this. In future USDPAA releases, it is likely that this will be deprecated in favour of a HugeTLB-based mechanism. In particular, this would help eliminate a limitation of the current USDPAA release, that prevents more than one application instance running at once-- the DMA memory region can only be safely mapped into one process at a time.

#### 36.3.3.2 DMA memory API

See the <usdpaa/dma_mem.h> header for a simple user space DMA memory API.

- dma_mem_memalign - dynamic allocation of DMA memory from within the large physically contiguous region.
- dma_mem_free - free allocated memory.
- dma_mem_ptov - convert physical to virtual (effective) address within user space process address space.
- dma_mem_vtop - convert virtual (effective) address to physical address.

Expect this API to be expanded in future software releases.

### 36.3.4 Network configuration

The USDPAA QMan and BMan drivers do not, in and of themselves, dictate which resources such as frame queues or buffer pools are used. In some cases, they can be dynamically allocated. In other cases, the specific resource is determined by a factor that is external to the USDPAA application itself.

The most common examples relate to the frame manager (FMan) and are discussed in section Relationship to SDK Linux ethernet subsystem on page 601.

The PPAC-based USDPAA applications ("reflector" and "ipfwd") use the usdpaa_netcfg_acquire() APIto determine the specific resources needed for a network interface. This configuration information is collected from several external sources:

- FMC policy file
- FMC configuration file
- the device-tree

#### 36.3.4.1 Network configuration initialisation requirements

As described in #unique_616, the USDPAA "of" driver must be initialised prior to network configuration being extracted from it. Once this is done, the usdpaa_netcfg_acquire() API, as defined in the <usdpaa/usdpaa_netcfg.h> header, can be used to extract the network configuration. In addition to initialisation of the USDPAA "of" driver layer, this API also requires the path to the two FMC XML files described above (policy and configuration), which are passed as parameters.

To satisfy leak-checkers or to support the use of persistent/reusable processes, an application can also undo all allocations and state by calling usdpaa_netcfg_release(), passing the "info" structure previously returned from usdpaa_netcfg_acquire() as a parameter.

## 36.3.5 CPU isolation

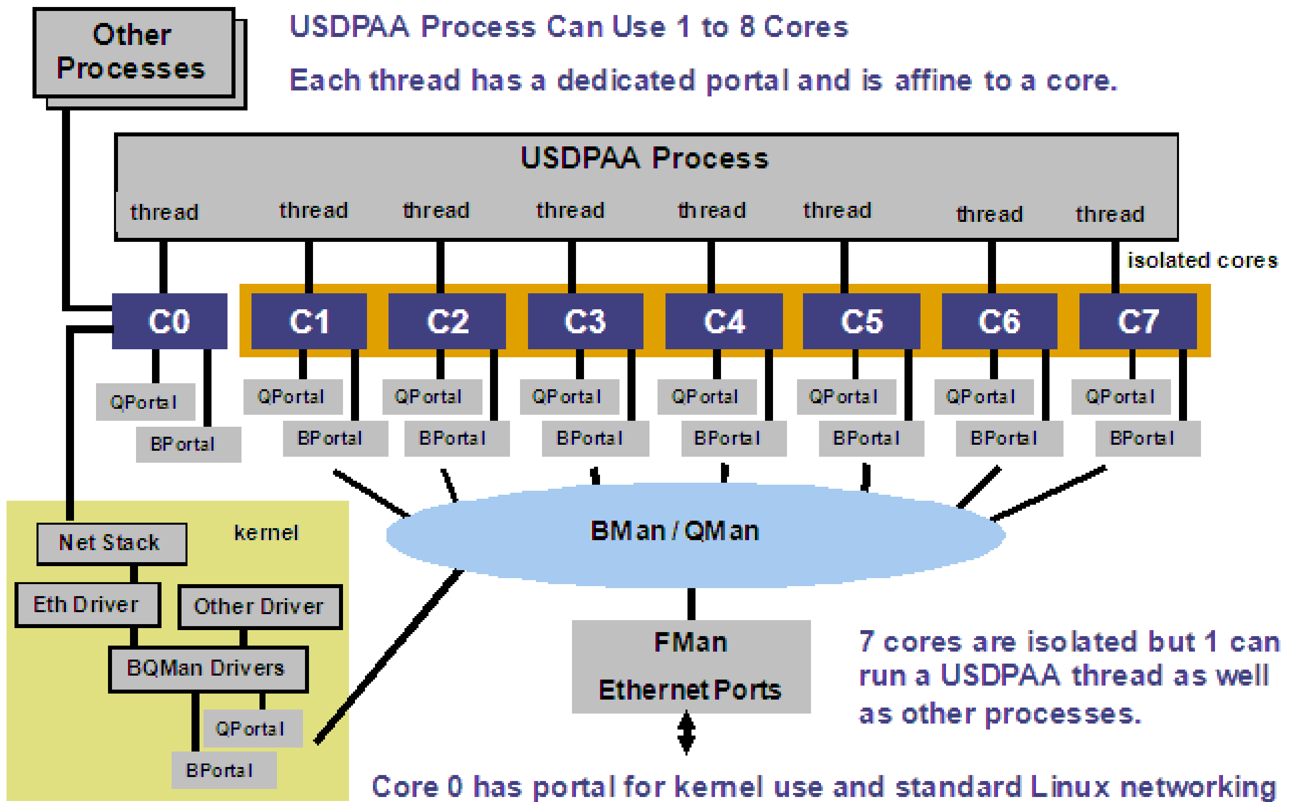
USDPAA provides no non-standard Linux features to provide CPU isolation, but the topic is important enough to merit some discussion. Standard Linux features may be use to achieve CPU isolation.

Especially in the run-to-completion use case, it can make sense to dedicate an entire core (which Linux would call a "cpu") to a single USDPAA process or thread. In other words, one wishes to reduce or eliminate any other software use of that particular core.

First, consider user space processes and threads. Linux provides a helpful kernel parameter called "isolcpus". This parameter indicates to the Linux kernel which cores should not have any user space process or thread scheduled onto them by default. The only way that a user space process or thread can execute on an isolated core is by explicit request.

For example, booting the kernel with "isolcpus=1-7" will isolate cores 1 through 7 as shown in the figure below. Note that this isolation is not architecturally required for functionality.

Figure 56: Portal allocation, affinity, and core-isolation




The "isolcpus" kernel parameter is documented (along with others) in the kernel source documentation directory, Documentation/kernel-parameters.txt.

The most convenient way to cause a USDPAA thread to be scheduled on an isolated to core is to use the function pthread_setaffinity_np(). This is a standard Linux pthreads function and "man pthread_setaffinity_np" should give documentation on it. (The "_np", for "non-portable", simply implies that it is not necessarily available in other pthreads implementations on other operating systems.)

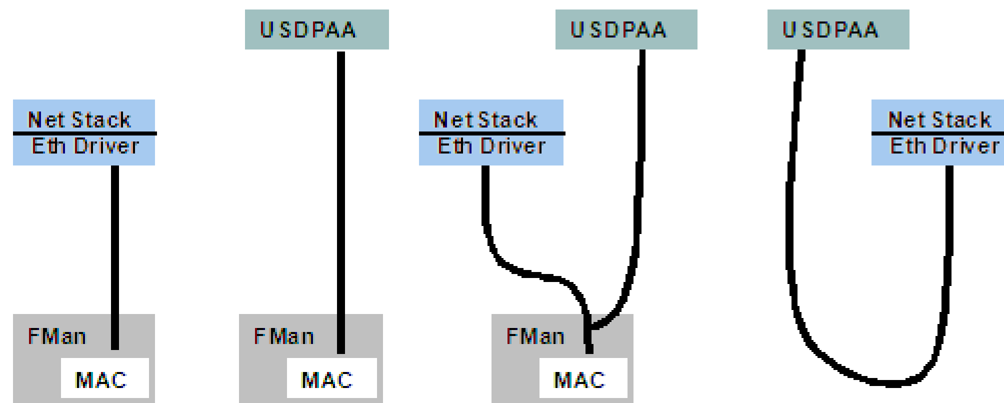
The USDPAA example applications also provide an example of this function's use. It is used not only to allow the USDPAA thread to be executed on an isolated core, it makes the thread affine to that core. Recall that affinity is recommended when using the user space QMan and BMan drivers.

Isolation is also a kernel topic. Users who wish to maximize isolation should use the standard Linux interrupt core binding mechanism, /proc/irq to bind interrupts to cores other than those that are isolated. Of course, interrupts associated with portals bound to cores should be bound to that core.

## 36.4 Relationship to SDK Linux ethernet subsystem

The DPAA SDK FMan and Linux kernel ethernet driver software is defined as external to the USDPAA software for this release. However, USDPAA, the ethernet driver in the kernel, and the FMan software are all interrelated. The figure below shows four use cases involving the three components.

Figure 57: USDPAA, FMan, ethernet use cases




1. QMan connects FMan MAC only to the kernel ethernet driver, which exchanges frames with the Linux network stack.
2. QMan connects an FMan MAC only to a USDPAA application.
3. QMan connects an FMan MAC to both the kernel driver and to a USDPAA application. On ingress, FMan selects the destination for each frame and enqueues it onto a particular frame queue accordingly. Different frame queues make the connections between the MAC and the ethernet driver and the MAC and the USDPAA application. This use case can be generalized by assuming multiple USDPAA applications, multiple ethernet driver instances, or both.
4. QMan connects an ethernet driver to a USDPAA application. FMan is not involved. Note that it is possible to accomplish this via the standard Linux facility TUN/TAP rather than using QMan.

The current release of USDPAA demonstrates cases 1 and 2.

### 36.4.1 Selecting ethernet interfaces for USDPAA

As will be discussed below, the ethernet driver is always involved in configuring FMan. This is true even for the second use case above in which the ethernet driver does not directly use FMan. Instead, the ethernet driver is creating an interface for another entity to use. Historically in the SDK that other entity was the LWE. In the current release, that other entity is a USDPAA application. The FMan and ethernet subsystem is actually unchanged from the standard DPAA SDK.

It is ethernet-related Linux device tree entries that determine the use case. This is documented in the "P4080 DPAA Device Bindings" distributed with the DPAA SDK. The sub-topic is "Data-Path Acceleration Assist".

The following device tree snippet shows a Linux private interface and also an interface used privately by USDPAA.

```
ethernet@0 {
   compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
   fsl,bman-buffer-pools = <&bp7 &bp8 &bp9>;
   fsl,qman-channel = <&qpool4>;
   fsl,qman-frame-queues-rx = <0x50 1 0x51 1>;
   fsl,qman-frame-queues-tx = <0x70 1 0x71 1>;
   fsl,fman-mac = <&enet0>;
};
ethernet@1 {
   compatible = "fsl,p4080-dpa-ethernet", "fsl,dpa-ethernet";
   fsl,qman-channel = <&qpool1>;
   fsl,fman-mac = <&enet1>;
};
```

The first ethernet is used by USDPAA. The second is used by the Linux ethernet driver.

Table 61: P4080DS ethernet interfaces

| Deice Tree Name | U-boot Name | U-Boot MAC Environment Variable | Linux Name (udev) | SerDes 0xe Physical Position |
|---|---|---|---|---|
| ethernet@0 | FM1@DTSEC 1 | ethaddr | fm1-gb0 | not used |
| ethernet@1 | FM1@DTSEC 2 | eth1addr | fm1-gb1 | Motherboad RGMII |
| ethernet@2 | FM1@DTSEC 3 | eth2addr | fm1-gb2 | not used |
| ethernet@3 | FM1@DTSEC 4 | eth3addr | fm1-gb3 | not used |
| ethernet@4 | FM1@TGEC1 | eth4addr | fm1-10g | slot 5 XAUI |
| ethernet@5 | FM2@DTSEC 1 | eth5addr | fm2-gb0 | not used |
| ethernet@6 | FM2@DTSEC 2 | eth6addr | fm2-gb1 | not used |
| ethernet@7 | FM2@DTSEC 3 | eth7addr | fm2-gb2 | slot 3 SGMII 2nd closest to motherboard |
| ethernet@8 | FM2@DTSEC 4 | eth8addr | fm2-gb3 | slot 3 SGMII closest to motherboard |
| ethernet@9 | FM2@TGEC1 | eth9addr | fm2-10g | slot 4 XAUI |

The DPAA SDK uses different names for different physical ethernet interfaces in different contexts. This is due to long-standing decisions on naming conventions, many not made by Freescale. It can be confusing. See table

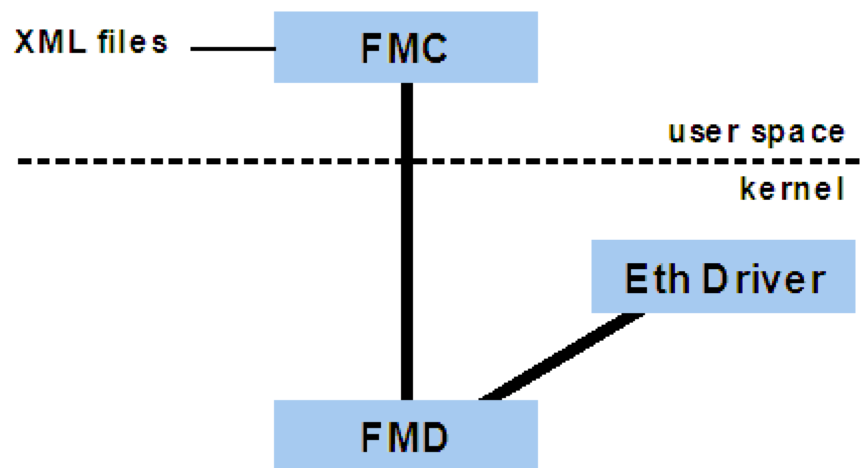
above for a list of all of the ethernet interfaces on the P4080 and their names in all of the important contexts. For P4080, no single SerDes protocol number provides access to all of the ethernets at the same time. This is due to pin multiplexing reasons.

The example defaults assume the use of the SerDes 0xe because it gives a total of 23 Gbps of ethernet connectivity on the P4080DS. The table shows which ethernet interfaces are used in SerDes 0xe.

### 36.4.2 FMC, FMD, and the ethernet Driver

The Frame Manager (FMan) software in the DPAA SDK is divided into a user space component (FMC) and a kernel component (FMD). The latter is the FMan driver in the conventional sense. FMD provides a capable API to support FMan configuration.

Figure 58: FMC, FMD, and ethernet




By itself, the ethernet driver uses FMD to configure only very simple FMan use cases. Basically, it configures only the default and error frame queues for ingress and egress. Doing more requires running the "fmc" user space application.

This application reads xml files as input and can establish more complex use-cases such as parsing incoming frames and hashing them into a set of frame queues for distribution to multiple cores (meaning USDPAA threads in the context of this document).

Various USDPAA example applications provide XML files and assume that fmc will be run. This usage is described in the application user guides.

## 36.5 Supported hardware platforms

This USDPAA release is tested in the following environments:

### 36.5.1 P4080DS

- P4080 revision 2 SoC. (Revision 1 is not supported.)
- The main test configuration
  - SGMII card in slot 3 with two interfaces used.
  - XAUI card in slot 4 (but the reflector example will work without it)
  - XAUI card in slot 5 (but the reflector example will work without it)

- SerDes protocol 0xe is the main test case. It provides
  - 1 x 1G ethernet port (via RGMII) used by the Linux kernel network stack.
  - 2 x 1G ethernet ports (via SGMII) used by USDPAA.
  - 2 x 10G ethernet ports (via XAUI) used by USDPAA.
- Testing is done with P4080 high bin clocking:
  - 1500 MHz core, 800 MHz platform, 1300 MHz DDR rate, 600 MHz FMan and PME
- Memory, 4 GB (but 8 GB should work also).
- Linux 8-way SMP.
- All caches (core-private L1/L2 and shared L3) are enabled. No locking or cache partitioning is used.

## 36.5.2 P3041DS

- SerDes protocol 0x36 (RR_HXAPNSP_0x36) is the main test case. It provides
  - 2 x 1G ethernet port (via RGMII) used by USDPAA.
  - 3 x 1G ethernet ports (via SGMII) used by USDPAA.
  - 2 x 10G ethernet ports (via XAUI) used by USDPAA.
- no interfaces used/available for the Linux kernel network stack.
- Testing is done with the rcw_15g_1500mhz:
  - 1500 MHz core, 750 MHz platform, 1333 MHz DDR rate, 583 MHz FMan and 375 MHz PME

## 36.5.3 P5020DS

- SerDes protocol 0x36 (RR_HXAPNSP_0x36) is the main test case. It provides
  - 2 x 1G ethernet ports (via RGMII) used by USDPAA.
  - 3 x 1G ethernet ports (via SGMII) used by USDPAA.
  - 1 x 10G ethernet port (via XAUI) used by USDPAA.
  - no interfaces used/available for the Linux kernel network stack.
- Testing is done with the rcw_15g_2000mhz:
  - 2000 MHz core, 800 MHz platform, 1333 MHz DDR rate, 600 MHz FMan and 400 MHz PME

## 36.5.4 P5040DS

- SerDes protocol 0x02 (RR_XXSNSpP_0x02) is the main test case. It provides
  - 2 x 1G ethernet ports (via RGMII) used by USDPAA.
  - 4 x 1G ethernet ports (via SGMII) used by USDPAA.
  - 2 x 10G ethernet port (via XAUI) used by USDPAA.
  - no interfaces used/available for the Linux kernel network stack.
- Testing is done with the rcw_26g_2267mhz.bin:
  - 2267 MHz core, 800 MHz platform, 1600 MHz DDR rate, 600 MHz FMan and 400 MHz

### 36.5.5 P2041RDB

- SerDes protocol 0x09 (RR_PX_0x09) is the main test case. It provides
  - 2 x 1G ethernet ports (via RGMII) used by USDPAA.
  - 2 x 1G ethernet ports (via SGMII) used by USDPAA.
  - 1 x 10G ethernet port (via XAUI) used by USDPAA.
  - no interfaces used/available for the Linux kernel network stack.
- Testing is done with the rcw_14g_1500mhz:
  - 1500 MHz core, 750 MHz platform, 1333 MHz DDR rate, 583 MHz FMan and 400 MHz PME

### 36.5.6 B4860QDS

- SerDes protocol 2a and 49 (N_NNSS_0x2A_0x49) is the main test case. It provides
  - 4 x 1G ethernet port (via SGMII) used by USDPAA.
  - 1 x 1G ethernet port (via SGMII) used/available for the Linux kernel network stack.
- Testing is done with the rcw_5sgmii_1600mhz.bin:
  - 1600 MHz core, 666.667 MHz platform, 800 MHz DDR rate, 666.667 MHz FMan and 333.333 MHz QMan.

### 36.5.7 T4240QDS

- SerDes protocol RR_XXXXPRPR_1_1_6_6 is the main test case. It provides
  - 2 x 1G ethernet ports (via SRIO) used by USDPAA.
  - 4 x 10G ethernet port (via XAUI) used by USDPAA.
  - 2 x 1G ethernet ports (via PCIe) used/available for the Linux kernel network stack.
- Testing is done with the rcw_1_1_6_6_1666MHz.bin:
  - 1666.667 MHz core, 666.667 MHz platform, 800 MHz DDR rate, 400 MHz FMan and 333.333 MHz PME

## 36.6 Example applications

USDPAA will be distributed with a set of example applications that is expected to grow over time.

The current set is

- A packet reflector application, "reflector".
- An IP forwarding performance demonstration, "ipfwd".
- A cryptographic accelerator example, "simple_crypto".
- A pattern-matching accelerator example, "pme_loopback_test".
- An IPFwd application based upon Longest Prefix Match methodology, "lpm_ipfwd".
- An application to route IPv4 packets after performing encryption/decryption, "IPSecfwd".
- A non-PPAC based stand-alone application, "hello_reflector".

Each example has its own user manual.

# 36.7 USDPAA installation and execution

The USDPAA software is contained within the SDK release, and should be compiled and installed by default by following the general instructions accompanying the release. Those instructions will not be reproduced here in their entirety. All that should be required to bring up and run the USDPAA environment, compared to booting the "standard" Linux system from the SDK, is to use the USDPAA-specific device tree. Eg. on the P4080DS system, one should use "uImage-p4080ds-usdpaa.dtb" instead of "uImage-p4080ds.dtb". Other than that, please follow the steps prescribed in the SDK installation notes. The following sections reproduce some of these steps (for the P4080DS case only), but primarily to ensure that USDPAA-relevant information is correctly identified.

## 36.7.1 Files needed to boot Linux on the P4080DS system

To run the USDPAA software, one must first boot Linux with the correct files;

- R_PPSXX_0xe/rcw_2sgmii_1500mhz.bin

  Reset configuration word (rcw) that selects the SerDes Protocol, suitable to program into the P4080DS NOR flash.

- u-boot-P4080DS.bin

  U-Boot bootloader image suitable to program into the P4080DS NOR flash.

- fsl_fman_ucode.bin

  FMan microcode for appropriate P4080 silicon suitable to program into the P4080DS NOR flash.

- uImage-p4080ds.bin

  Linux kernel supporting USDPAA.

- uImage-p4080ds-usdpaa.dtb

  Device tree file configured for USDPAA usage.

- fsl-image-core-p4080ds.ext2.gz.u-boot

  File system (used in RAM disk) that contains the needed user space files including USDPAA example application binaries. Linux must be booted using this file system.

All six of the files listed above are needed to run USDPAA. The first three must be programmed into the P4080DS NOR flash. The last three may be programmed into the NOR flash, but also may be loaded into RAM by u-boot using the tftp protocol.

U-boot is also capable of loading files into RAM via tftp and then programming them into the NOR flash. In all cases, you must have access to a tftp server, ideally on your Linux development host.

Copy the six files listed above to a directory from which they can be accessed via your tftp server. U-boot on the P4080DS must use tftp to access them. Details of installing and configuring a tftp server on your development host are specific to your host Linux distribution.

## 36.7.2 About U-Boot and network interfaces

U-boot is a bootloader. It is very flexible. but its main job is to do quite a bit of system configuration and then to load an operating system image (mainly Linux) into RAM and transfer control to it.

One of the simplest ways of transferring images (and other files) to the P4080DS running u-boot is to use tftp. For this to work, you must configure a network interface in u-boot.

Specifically, we assume that the 1 Gbps ethernet interface on the P4080DS motherboard will be used, and that the RCW file used causes this interface to correspond to the 2nd DTSEC in the P4080's first FMan instance.

At this point, we list for reference the text that U-boot should print when it runs. Note that there can be some variation in what U-boot prints.

```
U-Boot 2011.06-rc1-00037-g2bc0243 (May 27 2011 - 11:01:49)

CPU0:  P4080E, Version: 2.0, (0x82080020)
Core:  E500MC, Version: 2.0, (0x80230020)
Clock Configuration:
       CPU0:1499.985 MHz, CPU1:1499.985 MHz, CPU2:1499.985 MHz, CPU3:1499.985 MHz,
       CPU4:1499.985 MHz, CPU5:1499.985 MHz, CPU6:1499.985 MHz, CPU7:1499.985 MHz,
       CCB:799.992 MHz,
       DDR:649.994 MHz (1299.987 MT/s data rate) (Asynchronous), LBC:99.999 MHz
       FMAN1: 599.994 MHz
       FMAN2: 599.994 MHz
       PME:   599.994 MHz
L1:    D-cache 32 kB enabled
       I-cache 32 kB enabled
Board: P4080DS, Sys ID: 0x17, Sys Ver: 0x01, FPGA Ver: 0x0b, vBank: 4
36-bit Addressing
Reset Configuration Word (RCW):
       00000000: 105a0000 00000000 1e1e181e 0000cccc
       00000010: 3842440c 3c3c2000 fe800000 e1000000
       00000020: 00000000 00000000 00000000 008b6000
       00000030: 00000000 00000000 00000000 00000000
SERDES Reference Clocks: Bank1=100MHz Bank2=125MHz Bank3=125MHz
I2C:   ready
SPI:   ready
DRAM:  Initializing....using SPD
Detected UDIMM EBJ21EE8BAFA-DJ-E
Detected UDIMM EBJ21EE8BAFA-DJ-E
CS2 is disabled.
CS3 is disabled.
CS2 is disabled.
CS3 is disabled.
2 GiB left unmapped
    DDR: 4 GiB (DDR3, 64-bit, CL=9, ECC on)
       DDR Controller Interleaving Mode: cache line
       DDR Chip-Select Interleaving Mode: CS0+CS1
Testing 0x00000000 - 0x7fffffff
Testing 0x80000000 - 0xffffffff
Remap DDR 2 GiB left unmapped

POST memory PASSED
Flash: 128 MiB
L2:    128 KB enabled
Corenet Platform Cache: 2048 KB enabled
SRIO1: disabled
SRIO2: disabled
MMC:  FSL_ESDHC: 0
EEPROM: Invalid ID (ff ff ff ff)
PCIe1: Root Complex, x1, regs @ 0xfe200000
  01:00.0     - 1095:3132 - Mass storage controller
PCIe1: Bus 00 - 01
PCIe2: disabled
PCIe3: Root Complex, no link, regs @ 0xfe202000
PCIe3: Bus 02 - 02
In:    serial
Out:   serial
Err:   serial
```

```
Net:   Fman1: Uploading microcode version 101.8.0
FM1@DTSEC2 connected to Vitesse VSC8244
FM1@TGEC1 connected to Teranetics TN2020
Fman2: Uploading microcode version 101.8.0
FM2@DTSEC3 connected to Vitesse VSC8234
FM2@DTSEC4 connected to Vitesse VSC8234
FM2@TGEC1 connected to Teranetics TN2020
FM1@DTSEC2, FM1@TGEC1, FM2@DTSEC3, FM2@DTSEC4, FM2@TGEC1
```

In the above text, notice the final line;

```
FM1@DTSEC2, FM1@TGEC1, FM2@DTSEC3, FM2@DTSEC4, FM2@TGEC1
```

It lists the available network interfaces, which is determined by the RCW file and the SerDes protocol that it selects. This document assumes that SerDes 0xe is used. It provides the interfaces above. They will be used as follows.

- FM1@DTSEC2, U-boot MAC "eth1addr"

  This is the 1 GBps ethernet interface on the P4080DS motherboard. It will be used by U-Boot to transfer images and also is available in Linux as a standard ethernet interface. The "ifconfig" command will show it as "fm1-gb1" because the "gb's" are counted from zero.

- FM1@TGEC1 (slot 5 XAUI), U-boot MAC "eth4addr"
- FM2@DTSEC3 (slot 3 SGMII, 2nd closest to motherboard), U-boot MAC "eth7addr"
- FM2@DTSEC4 (slot 3 SGMII, closest to motherboard), U-boot MAC "eth8addr"
- FM2@TGEC1 (slot 4 XAUI), U-boot MAC "eth9addr"

  These other 4 interfaces, 2 x 1 GB and 2 x 10 GB, are dedicated to user space access via USDPAA.

This provides the USDPAA application with 22 Gbps of full-duplex ethernet capacity, assuming that an SGMII card is in P4080DS slot 3, and 10G XAUI cards are in slots 4 and 5. The USDPAA "reflector" application will function if the XAUI cards are not present, but only two GB of capacity will be available.

The P4080 has many ethernet interfaces. See #unique_631 for a complete summary of the names of these interfaces in different software contexts and for a statement of their use by the SerDes 0xe protocol on the P4080DS.

Returning to u-boot, selection of the network interface and networking parameters is done via u-boot environment variables. These are set using the "setenv" command that can be entered at the u-boot prompt. The network parameters include MAC addresses for the interfaces. Set them for all ten of the P4080 interfaces even though only a subset will be used.

Here is a complete list of the needed setenv commands. You will have to adjust the IP addresses and netmask to match your network. The addresses shown below are examples only.

```
setenv ethact FM1@DTSEC2
setenv ethaddr 00:04:9F:77:4E:00
setenv eth1addr 00:04:9F:77:4E:01
setenv eth2addr 00:04:9F:77:4E:02
setenv eth3addr 00:04:9F:77:4E:03
setenv eth4addr 00:04:9F:77:4E:04
setenv eth5addr 00:04:9F:77:4E:05
setenv eth6addr 00:04:9F:77:4E:06
setenv eth7addr 00:04:9F:77:4E:07
setenv eth8addr 00:04:9F:77:4E:08
setenv eth9addr 00:04:9F:77:4E:09
setenv ipaddr 10.82.146.151
setenv serverip 10.82.146.150
```

```
setenv gatewayip 10.82.146.150
setenv netmask 255.255.255.0
saveenv
```

In summary, U-boot will use the motherboard ethernet and will give it IP address 10.82.146.151. The tftp server has IP address 10.82.146.150. The "saveenv" saves all environment variable values into flash so they are retained after a reboot.

After entering the values above, you can test the U-boot network connection via ping and a trial tftp transfer. If this does not work, check the variables and network cables. This must work. Here is a text capture showing a successful test. You may need to adjust the path usdpaa/u-boot.bin per your tftp server configuration.

```
=> ping $serverip
Using FM1@DTSEC2 device
host 10.82.146.150 is alive
=> tftpboot 01000000 usdpaa/u-boot.bin
Using FM1@DTSEC2 device
TFTP from server 10.82.146.150; our IP address is 10.82.146.151
Filename 'usdpaa/u-boot.bin'.
Load address: 0x1000000
Loading: ####################################
done
Bytes transferred = 524288 (80000 hex)
```

## 36.7.3 P4080DS NOR flash banks

The P4080DS board has a feature that uses address swizzling to make it appear that the NOR flash is divided into multiple parts-- this document will assume two. The parts are called "bank 0" and "bank 4".

When you power-on or reset, u-boot will boot from bank 0. U-boot in bank 0 can program images into bank 4. Then, you can enter "pixis altbank" from the bank 0 u-boot prompt to boot into bank 4.

It is very wise to leave the bank 0 images alone and simply use them to program images into bank 4. This is to ensure that you always have working images in bank 0. This document will assume that you have u-boot flashed in bank 0. Use it only to program bank 4.

Look at the u-boot output in #unique_633. The line

```
Board: P4080DS, Sys ID: 0x17, Sys Ver: 0x01, FPGA Ver: 0x0b, vBank: 4
```

shows the bank from which u-boot was booted. In this case it was bank 4 (per vBank).

## 36.7.4 Programming the P4080DS NOR flash bank 4

First, boot from bank 0 by doing a reset or power-on. Check that you see "vBank: 0" since this is very important. Then, set network parameters using the u-boot environment variables described in section #unique_633.

The following U-boot commands will flash all six of the needed files into bank 4. Remember that you may have to adjust the paths in the tftpboot commands per your tftp server.

```
# BE BOOTED FROM BANK 0; WE WILL FLASH THE ALT BANK, WHICH WILL BE BANK 4

# rcw altbank
tftpboot 0x01000000 usdpaa/rcw_2sgmii_1500mhz.bin
protect off 0xec000000 +$filesize && erase 0xec000000 +$filesize && cp.b 0x01000000
0xec000000
$filesize
```

```
# u-boot altbank
tftpboot 0x01000000 usdpaa/u-boot.bin
protect off 0xebf80000 +$filesize && erase 0xebf80000 +$filesize && cp.b 0x01000000
0xebf80000
$filesize

# Delete altbank the u-boot env-- use default
protect off 0xebf60000 +0x20000 && erase 0xebf60000 +0x20000

# FMan u-code altbank
tftpboot 0x01000000 usdpaa/fsl_fman_ucode.bin
protect off 0xeb000000 +$filesize && erase 0xeb000000 +$filesize && cp.b 0x01000000
0xeb000000
$filesize

# device tree alt bank
tftpboot 0x01000000 usdpaa/p4080ds-usdpaa.dtb
protect off 0xec800000 +$filesize && erase 0xec800000 +$filesize && cp.b 0x01000000
0xec800000
$filesize

# kernel altbank
tftpboot 0x01000000 usdpaa/uImage
protect off 0xec020000 +$filesize && erase 0xec020000 +$filesize && cp.b 0x01000000
0xec020000
$filesize

# rootfs altbank
tftpboot 0x01000000 usdpaa/initramfs.cpio.gz.uboot
protect off 0xed300000 +$filesize && erase 0xed300000 +$filesize && cp.b 0x01000000
0xed300000
$filesize
```

## 36.7.5 Boot into bank 4 and set more variables

Next, enter the command "pixis altbank" to boot into bank 4 and press "any key" to stop the boot. Check that u-boot prints "vBank: 4". It is important that it does. If not, there is probably a mistake in the previous steps.

At this point, you must enter all of the networking u-boot environment variables (see section #unique_633) again and also set variable bootcmd. The latter is shown below along with a "saveenv" to save the values.

```
setenv bootcmd setenv bootargs root=/dev/ram rw console=ttyS0,115200 usdpaa_mem=256M
bportals=s0-1 qportals=s0-1 \; bootm 0xe8020000 0xe9300000 0xe8800000
saveenv
```

It is important that you type the backslash (\) because semicolon (;) is a command separator in u-boot. Here is a "printenv bootcmd" in a larger font and with line wrap showing the correct value. Note that the printenv does not show the backslash.

```
=> printenv bootcmd
bootcmd=setenv bootargs root=/dev/ram rw console=ttyS0,115200 usdpaa_mem=256M
bportals=s0-1 qportals=s0-1 ; bootm 0xe8020000 0xe9300000 0xe8800000
```

U-boot has an environment variable "bootdelay" that controls the number of seconds u-boot counts down before automatically running command "boot". If you prefer to run "boot" manually, set bootdelay to -1. This will cause u-boot to go directly to a command prompt. You can set bootdelay to whatever you want.

```
setenv bootdelay -1
saveenv
```

### 36.7.6 Environment variable hwconfig and optical 10G

At this point , enter the u-boot command "printenv hwconfig" as shown below.

```
=> printenv hwconfig
hwconfig=fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1
```

If you see the value above, and you plan to use the 10G copper interfaces (or no 10G at all), then all is well. You may skip to the next section.

If you plan to use optical 10G interfaces, you must add information to hwconfig. This is important. The optical interface will operate erratically without it. It is easiest to do this using the u-boot "editenv" command. Whatever you type will be appended to hwconfig. The text you need to append is

```
;fsl_fm2_xaui_phy:xfi;fsl_fm1_xaui_phy:xfi
```

The leading semicolon is needed. Do another "printenv hwconfig" to ensure that the value is correct and then a saveenv as shown below.

```
=> printenv hwconfig
hwconfig=fsl_ddr:ctlr_intlv=cacheline,bank_intlv=cs0_cs1;fsl_fm2_xaui_phy:xfi;fsl_fm1_x
aui_ph
y:xfi
=> saveenv
```

### 36.7.7 Booting Linux

At this point, you need to reset once again into bank 4 (u-boot command "pixis altbank") and run the u-boot "boot" command, either manually or by letting it happen automatically after the count-down.

Linux will boot and give you a login prompt.

Login as user "root" with password "root".

An "ifconfig -a" should show only one FMan (fm) ethernet interface, fm1-gb1. This is the P4080DS motherboard 1G ethernet interface. You can set its IP address and use it as an ordinary Linux network interface if you wish.

If you cat /root/SOURCE_THIS, you will see the commands needed to run the "reflector" example application. This is the first application you should examine. Please see its user manual for more information.

```
[root@p4080 /root]# cat /root/SOURCE_THIS
cd /usr/etc
fmc -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst.xml -a
reflector
```

### 36.7.8 Using tftp for the kernel, device-tree, and file-system

It can be slow to reflash the kernel, device tree, and file system every time you run ltib to change a USDPAA application (see section Known limitations on page 613). As an alternative, you can use the following u-boot commands (from bank 4) to boot Linux.

```
tftpboot 01000000 usdpaa/uImage
tftpboot 02000000 usdpaa/p4080ds-usdpaa.dtb
tftpboot 02100000 pusdpaa/initramfs.cpio.gz.uboot
boot
```

However, in this case the addresses in the "bootm" run by bootcmd need to be different. So, set bootcmd as follows.

```
setenv bootcmd setenv bootargs root=/dev/ram rw console=ttyS0,115200 usdpaa_mem=256M
bportals=s0-1 qportals=s0-1 \; bootm 01000000 02100000 02000000
saveenv
```

## 36.8 Using configurations other than SerDes 0xe

Most Freescale testing, examples, and discussion of examples in documentation assumes the use of SerDes 0xe and 22 Gbps of ethernet connectivity to USDPAA in the form of 2 x 1 Gbps + 2 x 10 Gbps.

It is possible to use USDPAA in other configurations and some examples will be summarized below.

See the document "Freescale DPAA SDK <version>: Selecting Ethernet Interfaces" for background information. It is distributed with the Freescale DPAA SDK.

### 36.8.1 SGMII (4 x 1 Gbps) card and one XAUI (10 Gbps) card

Goal (using Linux names for network interfaces, see Selecting ethernet interfaces for USDPAA on page 601):

- fm1-gb1 used by Linux kernel
- fm2-gb2 used by USDPAA
- fm2-gb3 used by USDPAA
- fm2-10g used by USDPAA

Method:

- SGMII card goes in P4080DS slot 3.
- XAUI card goes in P4080DS slot 4.
- Continue to use SerDes 0xe and RCW file R_PPSXX_0xe/rcw_2sgmii_1500mhz.bin in the P4080DS NOR flash.
- Boot into bank 4 (assuming you are using bank 4 for USDPAA).
- Add text ";serdes:fsl_srds_lpd_b3=0xf" to u-boot environment variable hwconfig, saveenv, and reset the system again into bank 4. This disables fm1-10g. The leading ";" separates the information added to hwconfig from what was already there.

- Delete references to fm1-10g from the fmc configuration file you will use. For example, delete the following text from it:

```
<engine name="fm0">
          <port type="10G" number="0" policy="hash_ipv4_policy_4"/>
</engine>
```

Run the USDPAA application.

### 36.8.2 SGMII (4 x 1 Gbps) card and no XAUI (10 Gbps) card

Goal (using Linux names for network interfaces, see table Selecting ethernet interfaces for USDPAA on page 601):

- fm1-gb1 used by Linux kernel
- fm2-gb0 used by USDPAA
- fm2-gb1 used by USDPAA
- fm2-gb2 used by USDPAA
- fm2-gb3 used by USDPAA

Method:

- SGMII card goes in P4080DS slot 3.
- Use SerDes 0x10 and RCW file: R_PPSXN_0x10/**rcw_5g_1500mhz.bin** in the P4080DS NOR flash. This is a different RCW file than the one discussed above.
- Boot into bank 4 (assuming you are using bank 4 for USDPAA).
- Add text ";serdes:fsl_srds_lpd_b2=0xf" to u-boot environment variable hwconfig, saveenv, and reset the system again into bank 4. This disables fm2-10g. The leading ";" separates the information added to hwconfig from what was already there. Note that fm1-10g is never available in SerDes 0x10.
- Delete references to any of the 10 Gbps interfaces and ensure references to all USDPAA 1 Gbps interfaces are present in the fmc configuration file. For example, it should like the following after you edit it:

```
<cfgdata>
    <config>
        <engine name="fm1">
                <port type="1G" number="0" policy="hash_ipv4_policy_5"/>
                <port type="1G" number="1" policy="hash_ipv4_policy_6"/>
                <port type="1G" number="2" policy="hash_ipv4_policy_7"/>
                <port type="1G" number="3" policy="hash_ipv4_policy_8"/>
        </engine>
    </config>
</cfgdata>
```

Run the USDPAA application.

## 36.9 Known limitations

- Interrupts for QMan and BMan portals used in user-space by USDPAA threads are not necessarily affine to the CPU to which the portal is assigned (ie. to the portal where stashing is performed and where the threads are advised to be affine). This is a current limitation of the UIO interface in the kernel which does not give the QMan/BMan drivers explict control over interrupt affinity. However given that interrupts are generally

used to implement sleeping/blocking semantics (eg. when idle), this is not expected to have a significant impact. As a workaround, if the need arises, the user can manually override the interrupt-affinity via the procfs controls available at /proc/irq.

- Only 1 Gbps full-duplex operation is supported on 1 Gbps ethernet links. It is also true that 10 Gbps links may only be used at 10 Gbps, but in this case the reason is a P4080DS board-level hardware limitation.
- Present release does not permit working with all four SGMII 1 Gbps ethernet ports and a XAUI 10 Gbps ethernet port at the same time. This would appear possible using SerDes 0x10, but it is not due to FMan buffer size constraints and support for jumbo frames. This is, at heart, a P4080 SoC hardware limitation, but future releases will provide greater flexibility.

## 36.10 Document history

Table 62: Document history

| Rev | Notes |
|---|---|
| 1.0 | Initial version. |
| 1.1 | FQID detail and more. |
| 1.2 | Description of proof-of-concept added. |
| 1.3 | Add TX FQIDs and discussion of application timing |
| 1.4 | Documentation of configuration files. |
| 1.5 | Instructions on running on the P4080DS. This version describes the first release of the proof-of-concept. |
| 1.6 | Corrected reserving 512 bytes in buffers. |
| 1.7 | Updated performance table for case when FMan automatically releases buffers. |
| 1.8 | Updated for 10G XAUI + 4x1G SGMII support and some optimizations. |
| 1.9 | Changed document title. Updated for phase 0 release. |
| 1.10 | Updated for USDPAA phase 1 release. |
| 1.11 | Late updates for phase 1 release including advice on non-22Gbps runs. |
| 1.12 | Add discussion of use cases beyond SerDes 0xe and 22 Gbps of ethernet connectivity for USDPAA. |
| 1.13 | Fixed minor typographical errors and made other small improvements. |
| 1.14 | Updated for SDK v1.0. |

*Table continues on the next page...*

Table 62: Document history (continued)

| Rev | Notes |
|---|---|
| 1.15 | Updated for SDK v1.3. |
| 1.16 | Added T4 & B4 platforms for SDK v1.3.1. |

# Chapter 37
# USDPAA Multiple Processor Support User Guide

## 37.1 USDPAA Multiple Process Support

Describes the modifications done in the QorIQ SDK1.2 to provide USDPAA multiple process support.

The changes introduced in the SDK V1.2 release and are described in this document as a set of changes between SDK V1.1 and the SDK V1.2 releases. Concurrently, a set of interim releases (IRn) are being developed to provide DPAA Offload support in the SDK. The IR2 release integrates support for multiple USDPAA processes. As such, the delta between SDK V1.1 and SDK V1.2 described in this document also applies as the delta between IR1 and IR2 releases.

## 37.2 USDPAA User/Kernel Device Interface

For SDK 1.1 most USDPAA resources were hard-coded into the USDPAA applications, there being only a single USDPAA process. For SDK 1.2 each process opens the “/dev/fsl_usdpaa” device once and uses this process driver for all that process's resource management.

#### SDK 1.1 QMan/BMan resources

Prior to the SDK 1.2 release, a simplifying assumption of a single process, together with assumptions about unused resources in the device tree led to hard-coded resources. This was the case for buffer pools, congestion groups, pool channels and frame queues.

#### SDK 1.1 DMA Memory

The only USDPAA resource that was kernel-managed was the mapping of DMA memory, via the

```
/dev/fsl_usdpaa_shmem
```

device. This also had the simplifying assumption that the entire memory region reserved by the kernel would always be mapped in its entirety by the unique USDPAA process, meaning that the functions in the USDPAA “dma_mem” API did not need any parameter to specify which DMA region was implied as there was only one region.

#### SDK 2.1 Process Driver

The “fsl_usdpaa_shmem” device has been renamed to simply “/dev/fsl_usdpaa” and is also referred to as the “process” device (within USDPAA, this is handled via the “process” driver), because the intention is for each process to open this device once and to use it for all that process's resource management.

#### SDK 2.1 QMan and BMan Resources

As of the 1.2 release, the process device supports ioctl() commands for (de)allocating resources to (and from) the kernel. So multiple USDPAA processes as well as any datapath logic in the kernel will all be using the same allocator for resources.

#### SDK 2.1 DMA Memory

The DMA mapping of the device remains but is enhanced to allow multiple regions and sub-regions to be allocated and mapped out of the total memory reservation. These regions are mapped independently and can be allocated by USDPAA apps, via the “dma_mem” driver, on the fly. The allocated regions have the same size and alignment limitations that come from the use of TLB1 entries to map them for fault-less access during

datapath operations, but the kernel management of these regions is maximally optimal. Ie. any combination of regions that can conceivably fit within the total memory reservation will always be obtainable, independent of the order in which the USDPAA processes request the allocation of those regions.

Other features of the allocation and mapping of DMA regions:

- They can be process-private (“unnamed”) or shareable (“named”). Named regions can be mapped by multiple processes.
- For shared regions, the process driver in the kernel provides a sleep-based locking scheme that the USDPAA dma_mem driver uses to synchronise buffer allocations within a sub-region across multiple processes.

### SDK 2.1 Resource Tracking

Because USDPAA processes open the /dev/fsl_usdpaa device and perform all resource management through that file-descriptor, the kernel device driver can track what resources are allocated and deallocated by those process. When such a process exits (intentionally or otherwise) the file-descriptor is cleaned up and this allows the device driver to check which resources had not been explicitly released by the application. If there are unreleased frame queues, buffer pools, pool channels or congestion groups, the kernel driver will issue leak warnings to the kernel log (and/or the serial console).Examples:

```
USDPAA process leaking 10 FQIDs
USDPAA process leaking 4 QPOOLs
```

Leaked resources are not automatically returned to the allocators, because the current drivers do not yet support automatic clean up and recovery of resources that are left in an undefined (and possibly volatile) state.

Even when applications explicitly deallocate resources back to the kernel-managed allocators (subsequently described here in the "Multi-process PPAC Applications" topic), there is no protection against applications that fail to first put those resources back into their expected “power-on” states. As such, an application that does not correctly clean up resources can, for the current version of the SDK (1.2), pollute the allocators with resources that will be allocated out to other users and lead to undefined results.

## 37.3 USDPAA Resource Management

Describes QMan and BMan resource availability and how to declare these resources at system initialization.

### QMan and BMan Portals

In SDK 1.1, QMan and BMan portals were declared in the device-tree with properties that pre-determined whether they were for use in the kernel or USDPAA (the latter were marked by the “fsl,usdpaa-portal” property) as well as a pre-determined CPU-affinity (the “cpu-handle” property links to a CPU node).

For SDK 1.2 and later versions, portals are declared in the device-tree simply as hardware resources, with no specification of what the portals will be used for nor which CPU they will be used from. The kernel parses all portals into an internal list, from(/to) which they can be (de)allocated as required.

The QMan driver will, by default, try to allocate a distinct portal for each core and initialise it for kernel use. This behaviour can be influenced by the use of the “qportals” boot argument, to use fewer portals and share them between cores. The mechanism by which a portal can be shared involves cores that do not have their own portals being “slaves” to a core that does have its own portal. Portal processing (interrupts, polling, changing dequeue masks, [etc]) is still performed only on the core to which the portal has been assigned, but software running on slave cores can perform software-initiated commands (enqueues, management commands, [etc]) on the shared portal due.

Once the kernel has initialised portals for its own use, it will allocate and export all the remaining portals as UIO devices for USDPAA use. When a USDPAA application thread initialises a portal for use, the opening of the UIO device triggers kernel logic to configure the portal as affine to the CPU the USDPAA thread is executing on.

### QMan Frame Queues (FQs) in SDK 1.1

The allocation of FQs was not coordinated between user-space and kernel-space. Kernel-space FQ allocations would, by default, acquire FQIDs from buffer pool zero, which was statically seeded (via device-tree entries) with a range of values from 0x100 (255) to 0x1ff (511) inclusive, though this behaviour could be overridden by the presence of a “fsl,fqid-range” node in the device-tree which would bypass the buffer pool allocator and instead implemented a software allocator using the given range.

User-space FQ allocations on the other hand always used a software allocator implementation whose range was hard-coded (in source code) to be from 0x200 (512) to 0x3ff (1023).

### QMan FQs in SDK 1.2

In kernal-space, support for using buffer pool zero as a special case for FQ allocations has been entirely removed. Now, FQ allocations are only possible if the device-tree contains a “fsl,fqid-range” node. There are device-tree include files (arch/powerpc/boot/dts/fsl/qoriq-dpaa-res*.dtsi) that declare default allocation ranges.

User-space FQ allocations are now always handled by using the resource allocation ioctl() commands in the USDPAA “process” driver. Ie. user-space FQ allocations are sourced from the allocator residing in kernel-space, and so multiple user-space processes and kernel code are using a common allocator.

### QMan Congestion Group Records (CGRs) for SDK 1.1

There was no facility at all for providing allocation of CGRs, and indeed there was no knowledge on the part of kernel or user-space as to how many CGRs were physically present in the hardware – any CGR-dependent software would simply be making its own assumptions about what CGRIDs were safe to use relative to the hardware and any other CGR-dependent software.

### QMan CGRs for SDK 1.2

The kernel now implements a CGRID allocator which is seed by a “fsl,cgrid-range” node in the device-tree. The user-space has the same API, and its allocations are routed in the kernel via the “process” driver in the same way as was mentioned for frame queues.

### QMan Pool Channels in SDK 1.1

There was no facility for providing allocations of pool-channels, however the device-tree did represent how many pool-channels were present in hardware (this was primarily to allow network devices to be statically configured to use particular pool-channels via device-tree linkage). The kernel-space driver could then enforce its knowledge of what pool-channels were available, but did not coordinate the resource so independent entities of software would need to avoid conflicts via its own means. The user-space driver also used the device-tree to determine the physically-available pool-channels, and provided no coordination of the resources.

### QMan Pool Channels in SDK 1.2

The use of device-tree linkage between network nodes and pool-channels in previous versions is gone. The kernel network driver has been adjusted to dynamically allocate its pool-channel instead. As with the other resource types, the USDPAA driver now has the same pool-channel allocation API as the kernel, with user-space allocation operations going via the “process” driver to be handled by the allocator in the kernel.

### BMan Buffer Pools in SDK 1.1

The kernel used to determine the number of buffer pools available by looking at the SoC version. Some buffer pools would be represented by device-tree nodes in order to support device-tree linkage with network nodes and in doing so could optionally be seed with ranges of values specified by node properties. The kernel would, by default, implement a BPID allocator that would automatically include all physically available buffer pools that were not explicitly mentioned in device-tree nodes (ie. device-tree nodes acted as reservations against being allocated). An optional “fsl,bpool-range” node could be used to override this behaviour, implementing a software allocator in the same way as “fsl,fqid-range” does.

### BMan Buffer Pools in SDK 1.2

If the device-tree contains a “fsl,bpid-range” node (previously named “fsl,bpool-range”). There are device-tree include files

```
arch/powerpc/boot/dts/fsl/qoriq-dpaa-res*.dtsi
```

that declare default allocation ranges. User-space FQ allocations are now always handled by using the resource allocation ioctl() commands in the USDPAA “process” driver. Ie. user-space FQ allocations are sourced from the allocator residing in kernel-space, and so multiple user-space processes and kernel code are using a common allocator.

**Changes for FMan Resources**

The only change to FMan resource management caused by the SDK 1.2 changes for USDPAA multi-process support is the removal of statically-assigned pool channels for ethernet interfaces. The “dpaa_eth” kernel driver now dynamically allocates a pool-channel during initialisation, and uses it for all the network interfaces it instantiates.

**USDPAA DMA Memory for SDK 1.1**

The kernel driver used an early-boot hook to reserve a memory region of a hard-coded size (configurable at the expense of a kernel recompile), and the USDPAA “dma_mem” driver would open the “/dev/fsl_usdpaa_shmem” device on behalf of the unique application process and mmap() all the of reserved memory. It was not possible to map less memory than that, it was not possible to create named/shared mappings, and so this was one of the reasons it was not possible to run multiple USDPAA processes.

**USDPAA DMA Memory for SDK 1.2**

The kernel driver now requires a boot-argument (“usdpaa_mem=<size>[,<num_tlb1>”) to trigger the reservation of USDPAA memory early during the kernel boot, otherwise no memory is reserved. The reservation of multiple TLB1 indices for use by USDPAA is also possible by passing a comma-separated argument. Using multiple TLB1 indices allows simultaneous mappings from USDPAA processes to DMA regions without any risk of fault handling overheads (note that if two processes map the same shared region, that requires 2 TLB1 indices). See section 2.2.3 for information on the device that exposes this memory to mapping from USDPAA applications.

## 37.4 BMan and QMan API

SDK 1.2 USDPAA processes performs resource management through the /dev/fsl_usdpaa file-descriptor .

**BMan Modifications - fsl_bman.h**

The changes described in this section apply to the BMan API in kernel-space and user-space unless otherwise specified. For more information on specific APIs (eg. the “partial” parameter to allocation functions or API return values) please see the comments in the header file that declares the interface (or consult the API reference manual).

**Removal of “recovery” API**

bman_recovery_cleanup_bpid() and bman_recovery_exit() have been removed, as they were never more than non-functional stubs and were conflicting with ongoing development.

**New BPID allocation API**

```
int bman_alloc_bpid_range(u32 *result, u32 count, u32 align, int partial);
static inline int bman_alloc_bpid(u32 *result)
{
       int ret = bman_alloc_bpid_range(result, 1, 0, 0);
       return (ret > 0) ? 0 : ret;
}
```

```
void bman_release_bpid_range(u32 bpid, unsigned int count);
static inline void bman_release_bpid(u32 bpid)
{
       bman_release_bpid_range(bpid, 1);
}
```

#### QMan Modifications - fsl_qman.h

The changes described here apply to the QMan API in kernel-space and user-space unless otherwise specified.

#### Removal of "recovery" API for SDK 1.2

qman_recovery_cleanup_fq() and qman_recovery_exit() have been removed, as they were never more than non-functional stubs and were conflicting with ongoing development.

#### Removal of Buffer-Pool Based FQ Allocator API

qm_fq_new(), qm_fq_free(), and the QM_FQ_FREE_* flags have been removed, as they were rendered unnecessary and awkward, in particular as they used to support “wait” options that were useful when deallocating FQIDs to a buffer pool but have no sane interpretation with the software-implemented allocator.

#### Removal of 'struct qman_portal_config::has_stashing' for SDK 1.2

Stashing can now be assumed as enabled in all environments (kernel-space, user-space, with Linux running natively, under the topaz hypervisor, or under KVM), so support for stashing-disabled operation has been removed for the sake of optimisation.

#### Removal of 'struct qman_fq_cb::dc_ern' for SDK 1.2

The callbacks associated with a frame queue object no longer support a DC_ERN handler (as these never worked properly because the concept is fundamentally unworkable). The lowest level at which DC_ERN messages can meaningfully be handled is at the portal level.

#### New API for handling DC_ERNs in SDK 1.2

It is possible to register a handler for DC_ERN messages with the portal affine to the running CPU, or as a global fallback for any portals that don't have their own handler.

```
void qman_set_dc_ern(qman_cb_dc_ern handler, int affine);
```

#### Removal of “NULL FQ” API in SDK 1.2

qman_get_null_cb(), qman_set_null_cb(), and QMAN_INITFQ_FLAG_NULL flag have been removed, as this functionality was considered marginal, had no known use-case, and was conflicting with ongoing development.

#### New API to Obtain Portal Channel for SDK 1.2

As portals are dynamically allocated, initialised, and assigned to CPUs during boot up, it became necessary for a user to be able to determine the channel ID for the portal associated with a given CPU, eg. in order to schedule frame queues such that dequeues would be handled on that CPU core.

```
enum qm_channel qman_affine_channel(int cpu);
```

#### New Pool-Channel Allocation API for SDK 1.2

For QMan pool-channel allocation:

```
int qman_alloc_pool_range(u32 *result, u32 count, u32 align, int partial);
static inline int qman_alloc_pool(u32 *result)
{
       int ret = qman_alloc_pool_range(result, 1, 0, 0);
       return (ret > 0) ? 0 : ret;
}
```

```
void qman_release_pool_range(u32 id, unsigned int count);
static inline void qman_release_pool(u32 id)
{
	qman_release_pool_range(id, 1);
}
```

**New QMan CGR allocation API for SDK 1.2**

```
int qman_alloc_cgrid_range(u32 *result, u32 count, u32 align, int partial);
static inline int qman_alloc_cgrid(u32 *result)
{
	int ret = qman_alloc_cgrid_range(result, 1, 0, 0);
	return (ret > 0) ? 0 : ret;
}

void qman_release_cgrid_range(u32 id, unsigned int count);
static inline void qman_release_cgrid(u32 id)
{
	qman_release_cgrid_range(id, 1);
}
```

## 37.5 USDPAA Thread and Global API

The API changes described apply to user-space (USDPAA).

**fsl_d.h -- Thread Initialization simplified**

The SDK 1.1 Thread initialization was verbose:

```
int qman_thread_init(int cpu, int recovery_mode);   /* remove for SDK 1.2 */
int bman_thread_init(int cpu, int recovery_mode);   /* remove for SDK 1.2 */
int qman_thread_init(void);
int bman_thread_init(void);
```

The SDK 1.2 Thread initialization is compact:

```
int qman_thread_init(void);
int bman_thread_init(void);
```

Recovery support (which was non-functional) has been removed so 'recovery_mode' is no longer a parameter. As for the 'cpu' parameter, portals are now dynamically bound to CPUs, so these functions will allocate any unused portal and it will be automatically bound to the CPU on which the caller is executing.

**fsl_d.h -- Global Initialization**

The SDK 1.1 Global initialization was verbose:

```
int qman_global_init(int recovery_mode);           /* remove for SDK 1.2 */
int bman_global_init(int recovery_mode);           /* remove for SDK 1.2 */
int qman_global_init(void);
int bman_global_init(void);
```

The SDK 1.2 Global initialization is compact:

```
int qman_global_init(void);
int bman_global_init(void);
```

As before, the 'recovery_mode' parameter is removed because the non-functional recovery interfaces have been removed.

## 37.6 USDPAA DMA API

The API changes described apply to DMA user-space (USDPAA).

#### dma_mem.h

The key change to this interface is that there can now be more than one DMA region available to each USDPAA process, so there is support for creating multiple such mappings, and as such most of the functions require that the DMA region be supplied as a parameter where there was no such need before.

#### Setup, or creation of DMA maps

The dma_mem driver used to initialise in a parameterless manner, creating the unique DMA map via dma_mem_setup(void), which has been removed. Instead, maps are created by the application using the following interfaces:

```
struct dma_mem;
#define DMA_MAP_FLAG_SHARED   0x01
#define DMA_MAP_FLAG_ALLOC    0x08
#define DMA_MAP_FLAG_NEW      0x02
#define DMA_MAP_FLAG_LAZY     0x04
#define DMA_MAP_FLAG_READONLY 0x10
struct dma_mem *dma_mem_create(uint32_t flags, const char *map_name,
                               size_t len);
```

The significance of the flags is described in more detail within the dma_mem.h header. To summarise, the SHARED flag creates a mapping to a new or existing DMA region that can be mapped by multiple USDPAA processes ('map_name' is the identifier for the region), otherwise a new region and mapping created that remains private to the process. If NEW is not specified the DMA region must already exist, whereas if NEW is specified the region must not already exist unless LAZY is also specified. LAZY refers to “lazy initialisation”, meaning that multiple processes can independently issue the same API call with the same name and specifying both the NEW and LAZY flags, with the result being that the named region will be allocated only once (by whichever process “wins the race”) and mapped into all the requesting processes.

If ALLOC is specified, then all processes that map the same region can use the dma_mem_memalign() and dma_mem_free() interfaces to allocate blocks from the region in a coordinated way. Without ALLOC, the region is created “raw”, meaning the user manipulates the entire region directly without any allocator functionality provided by the dma_mem driver.

#### Raw Memory Regions

If a DMA region and mapping is created with the RAW flag, it can then be accessed via;

```
void *dma_mem_raw(struct dma_mem *map, size_t *len);
```

#### Memory allocation

These functions are similar to those in SDK 1.1, with the exception that they require a parameter to indicate which DMA map to use. For SDK 1.1:

```
void *dma_mem_memalign(size_t boundary, size_t size);    /* SDK 1.1 version */
void dma_mem_free(void *ptr, size_t size);               /* SDK 1.1 version */
```

Additional parameter for SDK 1.2:

```
void *dma_mem_memalign(struct dma_mem *map, size_t boundary, size_t size);
void dma_mem_free(struct dma_mem *map, void *ptr);
```

### Distinguishing DMA regions

```
struct dma_mem *dma_mem_findv(void *v);
struct dma_mem *dma_mem_findp(dma_addr_t p);
```

### Physical/virtual address conversion

As with memory allocation, these functions require a parameter to indicate which DMA map to use, but are otherwise similar to those in SDK 1.1. In the case where the DMA map is not known, use the functions mentioned in “Distinguishing DMA regions” first. Note, these functions are actually implemented as inlines with some nasty details involving casts that should be ignored, this is simply because these routines are performance critical in packet-processing processing;

```
static inline void *dma_mem_ptov(struct dma_mem *map, dma_addr_t p) { … }
static inline dma_addr_t dma_mem_vtop(struct dma_mem *map, void *v) { … }
```

### Legacy interfaces, “dma_mem_generic”

In order to facilitate porting of legacy applications to the new dma_mem API, the following mechanism is provided. A global variable within the dma_mem driver, dma_mem_generic, is NULL by default but can be set by the application once it has created a DMA mapping. From that point on, it could use the following __dma_mem_*() functions, which do not require a DMA map parameter.

```
extern struct dma_mem *dma_mem_generic;
static inline void *__dma_mem_ptov(dma_addr_t p)
{
       return dma_mem_ptov(dma_mem_generic, p);
}
static inline dma_addr_t __dma_mem_vtop(void *v)
{
       return dma_mem_vtop(dma_mem_generic, v);
}
static inline void *__dma_mem_memalign(size_t boundary, size_t size)
{
       return dma_mem_memalign(dma_mem_generic, boundary, size);
}
static inline void __dma_mem_free(void *ptr)
{
       return dma_mem_free(dma_mem_generic, ptr);
}
```

That is, in order to port a legacy application (which worked on the assumption of there being a unique, canonical DMA mapping for all DMA operations), it should suffice to;

1. During application initialisation, create a default DMA map using dma_mem_create(), and assign that to dma_mem_generic,
2. Change all the legacy dma_mem_*() calls in the application to __dma_mem_*().

## 37.7 USDPAA netcfg.h

The API changes described apply to user-space (USDPAA).

For SDK 1.1 and previous versions:

```
struct usdpaa_netcfg_info {
       uint8_t num_cgrids;             /* SDK 1.1 and previous */
       uint32_t *cgrids;               /* SDK 1.1 and previous */
       uint8_t num_pool_channels;      /* SDK 1.1 and previous */
       enum qm_channel *pool_channels; /* SDK 1.1 and previous */
       uint8_t num_ethports;           /* Number of ports */
       [...]
```

For SDK 1.2 and subsequent versions:

```
struct usdpaa_netcfg_info {
       uint8_t num_ethports;           /* Number of ports */
       [...]
```

'struct usdpaa_netcfg_info' no longer specifies CGR and pool-channel resources to applications. There are now allocators in the QMan API for both these resource types, and they no longer need to come from the network configuration. Note that these resources were previously coming from hard-coded work-arounds if present at all.

See the "USDPAA Resource Management Modifications for SDK 1.2" topic for details on CGRs and QMan Pool Channels.

## 37.8 Kernel configuration

Kconfig settings for the fsl_qbman driver have changed.

SDK 1.2 Kconfig changes:

- CONFIG_FSL_DPA_HAVE_IRQ is removed, IRQ support is always enabled.
- CONFIG_FSL_BMAN_PORTAL is removed, support for BMan portals is always enabled.
- CONFIG_FSL_QMAN_PORTAL is removed, support for QMan portals is always enabled.
- CONFIG_FSL_QMAN_PORTAL_DISABLEAUTO_DCA is removed, QMan portals are always enabled for DCA consumption of DQRR (dequeue response ring) entries.
- CONFIG_FSL_QMAN_NULL_FQ_DEMUX is removed, see the BMan and QMan API Modifications topic, Removal of “NULL FQ” API heading.
- CONFIG_FSL_QMAN_DQRR_PREFETCHING is removed, the driver is now optimised to always assume stashing is always enabled, so support for pre-fetching is removed. This removes a run-time check from the critical path.

## 37.9 Device Tree (Excluding QMan/BMan Resource Ranges)

Device tree changes, excluding the QMan/BMan resource ranges.

**QMan/BMan portals**

Portals no longer have “fsl,usdpaa-portal” or “cpu-handle” properties.

```
        qportal1: qman-portal@4000 {
                cell-index = <0x1>;
                compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";
                reg = <0x4000 0x4000 0x101000 0x1000>;
                interrupts = <106 0x2 0 0>;
                fsl,qman-channel-id = <0x1>;
                cpu-handle = <&cpu1>;

        };
```

**BPID 0 FQ-allocator**

No buffer pool device tree node for BPID 0, because we no longer support that legacy mechanism for FQID allocation.

**QMan Pool channels**

Pool channel nodes have been removed and replaced by “pool channel range” nodes.

**BMan buffer pools**

Buffer pool nodes have not been removed, because they are still linked to by network-related nodes. However they are now ignored by the fsl_qbman driver and so no longer contain the “fsl,bpool-cfg” property type. Eventually, network configuration will obtain buffer pools dynamically, at which point there should be no more need for individual buffer pol nodes in the device-tree. (There is no current plan for when this deprecation will occur.)

**Ethernet Interfaces**

Ethernet nodes no longer have “fsl,qman-channel” properties linking them to pool channels.

```
        ethernet@2 {
                compatible = "fsl,p4080-dpa-ethernet", "fsl,dpa-ethernet";
                fsl,fman-mac = <&enet2>;
```

## 37.10 Device Tree (QMan/BMan Resource Ranges)

These device tree resource properties share a common format.

**QMan and BMan Portals**

Portals no longer have “fsl,usdpaa-portal” or “cpu-handle” properties. For SDK 1.1 and previous:

```
qportal1: qman-portal@4000 {
             cell-index = <0x1>;
             compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";
             reg = <0x4000 0x4000 0x101000 0x1000>;
             interrupts = <106 0x2 0 0>;
             fsl,qman-channel-id = <0x1>;
```

```
            fsl,usdpaa-portal;                                  /* SDK 1.1 only */
            cpu-handle = <&cpu1>;
            fsl,qman-pool-channels = <&qpool4 &qpool5 &qpool6
                                     &qpool7 &qpool8 &qpool9
                                     &qpool10 &qpool11 &qpool12
                                     &qpool13 &qpool14 &qpool15>; /*SDK 1.1 only */
      };
```

For SDK 1.2 and later versions:

```
qportal1: qman-portal@4000 {
            cell-index = <0x1>;
            compatible = "fsl,p4080-qman-portal", "fsl,qman-portal";
            reg = <0x4000 0x4000 0x101000 0x1000>;
            interrupts = <106 0x2 0 0>;
            fsl,qman-channel-id = <0x1>;
            cpu-handle = <&cpu1>;
      };
```

### BPID 0 FQ-allocator

No buffer pool device tree node for BPID 0, because we no longer support that legacy mechanism for FQID allocation. See section 3.2.

### QMan Pool Channels

Pool channel nodes have been removed (and replaced by “pool channel range” nodes, see the corresponding item below). See section 3.4.

### BMan Buffer Pools

Buffer pool nodes have not been removed, because they are still linked to by network-related nodes. However they are now ignored by the fsl_qbman driver and so no longer contain the “fsl,bpool-cfg” property type. Eventually, network configuration will obtain buffer pools dynamically, at which point there should be no more need for individual buffer pol nodes in the device-tree. There is no precise plan for when this deprecation will occur though.

### QMan and BMan Resource Ranges (Allocation)

The following resource properties share a common format, with a 2-tuple specifying a base+count pair. Eg. if the pool-channel range property specifies “<0x21 0xf>”, that corresponds to a range of pool channel IDs ranging from 33 (0x21) to 47 (0x21+0xf-1), inclusive.

### FQID

“FQID range” nodes have been added to specify FQs that are available for dynamic allocation. These do not yet include all the FQs that are available in the system, because there are still some legacy requirements for pre-configured FQIDs that have not been updated to use dynamic allocation. These nodes were previously supported but were not enabled by the default device trees, whereas they are now used in all cases, via the including of arch/powerpc/boot/dts/fsl/qoriq-dpaa-res*.dtsi files (which did not exist in SDK 1.1) as follows:

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

### CGRID Dynamic Allocation

“CGRID range” nodes have been added to specify CGRs that are available for dynamic allocation. Example:

```
         qman-cgrids@0 {
                compatible = "fsl,cgrid-range";
                fsl,cgrid-range = <0 256>;
            };
```

#### Pool Channel

“Pool channel range” nodes have been added to specify pool channels that are available for dynamic allocation. Example:

```
        qman-pools@0 {
                compatible = "fsl,pool-channel-range";
                fsl,pool-channel-range = <0x21 0xf>;
        };
```

#### BPID

“BPID range” nodes have been added to specify buffer pools that are available for dynamic allocation. Example:

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

## 37.11 USDPAA Boot Arguments

Without this boot-argument, no memory is reserved by the “fsl_usdpaa” driver and so no memory will be available for creating DMA mappings in USDPAA applications.

#### usdpaa_mem

The usdpaa_mem argument is not set by default by u-boot, device-trees or any other resource. The user must add it explicitly to boot commands in order to be able to run USDPAA applications. The size can be expressed using the standard suffixes for memory size notation (“256M”, “1G”, etc).

A second (and optional) argument allows multiple TLB1 indices to be reserved for mapping regions within the memory reservation. Without this extra argument, the default behaviour is to reserve only 1 TLB1 entry. Ie. “usdpaa_mem=64M,1” is equivalent to “usdpaa_mem=64M”. Note that the handling of page faults for software access to these resources will be satisfied by using the reserved TLB1 entries in a round-robin fashion, so if there are more mappings between user-space processes and DMA regions than there are TLB1 entries, and all of those mappings are being actively used at run-time, then performance will degenerate.

Since single TLB1 entries can only map power of 4 memory sizes a set of 1 or more TLB1 entries is required in to map a DMA region into each process. The USDPAA driver will use the minimum number of TLB1 entries possible to map each DMA region.

If a single process maps to two DMA regions, that would require two sets of TLB1 entries. But the same is true if two distinct processes map to one private DMA region each. So the number of TLB1 entries required is really the number of pairings between user-space processes and distinct DMA regions, or “the number of mappings”, to put it another way. The total number of TLB1 entries available depends on the SoC version, so one should check this when determining how many TLB1 entries to dedicate to USDPAA.

Note that a typical kernel would internally only use 3 or so TLB1 entries, and all the remaining entries are normally reserved by the “hugetlb” driver for a similar kind of large-page fault-handling algorithm as that implemented for

USDPAA – indeed, the reason USDPAA does not just use “hugetlb” directly is that it has no mechanisms to support user-space obtaining physical addresses and performing DMA.

One must also determine whether HugeTLB support is required. This decision determines the number of TLB1 entries for USDPAA use and, in turn, how many processes and active DMA mappings used. The USDPAA could use most of the TLB1 entries if HugeTLB support is not used.

**"qportals" and "bportal;s"**

By default, the kernel will attempt to allocate portals for each online core. If however the “qportals” (for QMan portals) and/or “bportals” (for BMan portals) boot-arguments are specified, this behaviour will be overridden. These boot-arguments specify cores that should be assigned portals to them, with the implication being that cores that are not specified will need to “slave” off the cores that are assigned.

Table 63: boot-argument: "qportals=1,s3-4"

| Core | Portal | Association |
|---|---|---|
| 0 | B | slave |
| 1 | A | affine unshared |
| 2 | C | slave |
| 3 | B | affine shared |
| 4 | C | affine shared |
| 5 | B | slave |
| 7 | B | slave |

# 37.12 USDPAA Virtualisation and Partitioning

In a partitioned system, it is necessary to assign each partition non-conflicting subsets of the hardware resources.

All the resources mentioned in section 3 can and should be divided up, with each instance of Linux receiving distinct portals and other dynamically-allocated resources via their respective device-trees.

Some use-cases may intentionally share resources between partitions, in which coordination of the corresponding resource IDs is up to the application. The resources provided to the partitions by the device-tree are inherently managed by the drivers themselves, and these must be mutually-exclusive because the drivers in the distinct partitions have no innate coordination.

# 37.13 Multi-process PPAC Applications

A description of how the networking applications are affected by, and have been adapted for, multi-process support.

The USDPAA toolkit provides a template called “PPAC” (Packet Processing Application Core) for simple networking applications, together with some applications called “PPAM”s (Packet Processing Application Module) that are written on top of this template. These networking applications notably include “reflector” and “ipfwd”.

Running multiple distinct PPAM application processes (or “instances” as they are sometimes described) requires that each process have its own dedicated set of FMAN interfaces, buffer pools, and cores.

### FMan Interfaces

By default, a PPAM application will automatically configure and use all available FMan interfaces, unless a specific set of interfaces is specified via the “-i” option. For example:

```
<app1> -i fm1-10g,fm2-10g
<app2> -i fm2-gb2,fm2-gb3
```

The names of FMAN interfaces that can be used with “-i” option are as follows.

Table 64: FMan "-i" Options

| FMAN1 | Fman2 |
|---|---|
| fm1-gb0 | fm2-gb0 |
| fm1-gb1 | fm2-gb1 |
| fm1-gb2 | fm2-gb2 |
| fm1-gb32 | fm2-gb3 |
| fm1-gb4 | fm2-gb4 |
| fm1-10g | fm2-10g |

As per SERDES protocol, a set of FMAN interfaces can be chosen to run with an application.

### Buffer pool restrictions

The device tree specifies the ethernet nodes such that each has a set of buffer pools for FMan to use when receiving frames. One restriction of the PPAC multi-process support is that the buffer pools used by FMan interfaces in one process must not also be used by any FMan interfaces in any other process. As such the device tree may need to be adjusted to ensure that any reuse of a buffer pool by more than one FMan interface must only occur when those FMan interfaces will always belong to the same process. For example, if one application wants to use fm1-10g and fm2-10g and another application is going to use fm2-gb2 and fm2-gb3, then corresponding ethernet nodes might look like:

```
ethernet@4 {
                compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
                    fsl,bman-buffer-pools = <&bp9>;
                    fsl,qman-frame-queues-rx = <0x5a 1 0x5b 1>;
                    fsl,qman-frame-queues-tx = <0x7a 1 0x7b 1>;
            };

ethernet@9 {
                compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
                    fsl,bman-buffer-pools = <&bp9>;
                    fsl,qman-frame-queues-rx = <0x66 1 0x67 1>;
                    fsl,qman-frame-queues-tx = <0x86 1 0x87 1>;
            };
ethernet@7 {
                compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
                    fsl,bman-buffer-pools = <&bp8>;
```

```
                        fsl,qman-frame-queues-rx = <0x60 1 0x61 1>;
                        fsl,qman-frame-queues-tx = <0x80 1 0x81 1>;
                };
ethernet@8 {
                    compatible = "fsl,p4080-dpa-ethernet-init", "fsl,dpa-ethernet-init";
                        fsl,bman-buffer-pools = <&bp8>;
                        fsl,qman-frame-queues-rx = <0x62 1 0x63 1>;
                        fsl,qman-frame-queues-tx = <0x82 1 0x83 1>;
                };
```

### Seeding buffer pools

Each PPAC-based application process will initialise the (usually 3) buffer pools used by each FMan interface that belongs to it. (If a buffer pool is used by more than one interface, it will only be initialised once.) By default the number of buffers to allocate for the triplet of pools used by an FMan interface is 0 for the first two pools and 1728 for the third. The default allocation triplet can be overridden via the "–b" option. For example, to continue the earlier example of a two-process scenario, and to have each process allocate 1600 buffers for the first pool used by any network interface and 0 for the second and third pools, they would use the following arguments :

```
<app1> -b 1600:0:0 -i fm1-10g,fm2-10g
<app2> -b 1600:0:0 -i fm2-gb2,fm2-gb3
```

### Cores

By default, each PPAC-based application process will start a single thread running on core 1. Additional threads can be started and stopped on arbitrary cores using the interactive CLI, but the first/primary thread can not be removed without ending the process. So for multi-process scenarios, it is better for each application instance to specify a core or set of cores as part of the command line. For example, to split 8 cores in half between our two hypothetical application processes;

```
<app1> 0..3 -b 1600:0:0 -i fm1-10g,fm2-10g
<app2> 4..7 -b 1600:0:0 -i fm2-gb2,fm2-gb3
```

## 37.14 Limitations

The use of dynamic allocation of resources does not address the appropriate state of an allocation.

A resource that is deallocated by one user can subsequently be allocated by another and it will be in the same state it was left, for better or worse. As such, stability and integrity of (and between) datapath applications is entirely a matter of cooperation. Future releases will implement measures to quiesce and recover such resources to make the system more robust in the face of individual application failures.

For the reasons explained above, most types of resources can be leaked by USDPAA applications that exit (or crash) before deallocating them. Although an application can explicitly deallocate a resource in a bad state, the risk of a resource being in a bad state when an application exits without having deallocated it is considered too great – so it is leaked rather than allowing it lead to undefined behaviour in future uses.

An exception to the last comment is for QMan and BMan portals, which due to them being UIO-based means they are implicitly "deallocated" when a process exits. In a future release, portals will likely no longer be UIO-based, and in any case, they will likely be "cleaned up" on process-exit.

# Chapter 38
# USDPAA Reflector and PPAC User Guide SDK

## 38.1 Introduction

The User Space Datapath Acceleration Architecture (USDPAA) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document describes the "reflector" application and the PPAC abstraction on which it is built, both contained in the USDPAA package. This application serves as a reference example for working with the USDPAA interface, as well as providing a benchmark for USDPAA system performance. More elaborate PPAC-based applications (such as "ipfwd") have their own user guide documents, but do not repeat the PPAC discussion found here.

Furthermore, a stripped-down "hello_reflector" application exists within USDPAA that is implemented without using the PPAC abstraction. This application is missing many of the features of PPAC, but is intended to provide a usable comparison between PPAC-based and stand-alone applications, eg. for customers prototyping with PPAC and then looking to develop stand-alone production code. As "hello_reflector" is very simplistic, the source code is self-documenting and it will not be discussed in any detail here. A short guide to executing the "hello_reflector" application will be provided at the end of this document.

A variant of hello_reflector called hello_reflector "short circuit" is also provided which basically tests the sanity of the hardware path for the packet flow by this way without any processing by the core on the received packets.

### 38.1.1 Intended audience

This document is intended for software developers and system architects who work with the USDPAA framework.

The material is technical in nature. The reader is assumed to be familiar with:

- General Linux software development, operation, and configuration for Power architecture devices in particular.
- Familiarity with the concept of device drivers and their role in operating systems.
- Linux network administration and use.
- The Freescale Linux SDK for DPAA-based SoCs.
- At least broadly, the DPAA hardware blocks.
- The USDPAA User Guide document (this document covers only the reflector application).
- Linux UIO (User space I/O) driver infrastructure (USDPAA drivers for QMan/BMan portals use the standard UIO mechanisms for memory-mapping and interrupt-handling).

The P4080 was the first Freescale SoC to incorporate the DPAA. As such, it is used in many examples. However, USDPAA is intended to apply in the same manner to all DPAA-based SoCs.

### 38.1.2 Change history

Table 65: Change History

| Version | Updates |
|---|---|
| 1.0 | Creation of reflector UG for phase1 release, based on the general UG prepared for the phase0 release.<br>- tidy up of text<br>- describe phase1-specific additions; PPAC/PPAM, IRQ mode, etc. |
| 1.1 | Add section on testing reflector using a Linux PC. |
| 1.2 | Fix minor typographical errors. |
| 1.3 | Updates for v1.0 release.<br>- split PPAC-common material from reflector specifics<br>- corrected and updated buffer pool descriptions |
| 1.4 | Updates for v1.1 release<br>- document "hello_reflector"<br>- document cmd-line args and environment-variables<br>- update sample console output<br>- update buffer definitions |

## 38.2 Overview of reflector

The reflector application is a simple packet-processing USDPAA application. The main purpose of USDPAA's user space drivers (as opposed to more conventional kernel-mediated device access) is raw I/O performance, and the reflector application focuses on this. Reflector is basically all I/O - it receives and sends frames, but does very little processing on them. It demonstrates the I/O capabilities of the DPAA hardware and corresponding user-space drivers.

## 38.3 Overview of PPAC

The source code to reflector has been reorganized into two parts; the "PPAC" (Packet-Processing Application Core) and a "PPAM" (Packet-Processing Application Module). The idea behind this is that many packet-processing applications (particularly any variations on the theme of network forwarding) would likely only differ from reflector in the way they make their "forwarding decision". The essential difference is the logic that looks at the packet and determines what to do with it. The PPAM portion implements this application specific logic. On the other hand, the PPAC component represents the common infrastructure to support such PPAMs; initializing devices, handling flow-control, implementing a CLI (Command-Line Interface), managing threads and buffers, [etc]. For USDPAA demonstration applications, the use of a common PPAC component allows for easier code development and maintenance and a uniform application look and feel.

Users are not obliged to use PPAC for their USDPAA application development. Moreover users could choose to implement PPAC-based applications (eg. for quick prototyping) and then later port their work to a stand-alone product. A stand-alone version of reflector, called "hello_reflector", has been provided to illustrate this principle. Please note that the emphasis in "hello_reflector" is on simplicity, so it does not (re)implement stand-alone equivalents of all the features and flexibility of the PPAC environment.

Additionally, if the user only wants to first check the sanity of the hardware path for the packet flow and later wants to send the packets to the core for the processing, he can first run hello_reflector in "short circuit" mode and if the traffic is received back on the hardware path, he can now run hello_reflector and so involve the core in the packet processing.

In "short circuit" hello reflector, packets are dequeued from the interface and received on Rx FQs. These packets are then sent out from the QMAN channel on which Tx FQs are scheduled.

## 38.4 PPAC details

However at an implementation level, there is a limit to the degree of abstraction one can obtain. E.g. for 64 byte packets, using between 1 and 4 CPUs on a p4080 DS, reflector processing averages out to approximately 170 CPU cycles per-packet. Adding just one extra level of indirection to that processing path can add enough overhead to make a visible performance difference to the application (one such ad-hoc experiment showed an additional overhead of ~20 cycles, causing a 12% performance degradation). For this reason, the PPAC/PPAM interface is implemented by a strategic use of inlining. The resulting PPAC implementation and interface is organized in such a way that the compiler is able to inline the fast-path code of PPAC and PPAM together, as though they were written as a single (or "flat") application. Indeed, the PPAM version of reflector, despite the PPAC/PPAM split, has no performance degradation relative to earlier non-PPAM versions.

### 38.4.1 IRQ mode for sleeping when idle

A new feature of PPAC (and thus all PPAM applications like reflector and "ipfwd") is support for interrupts and sleeping. As USDPAA's QMan and BMan drivers use the standard Linux UIO subsystem, the behavior of interrupt-handling is in keeping with UIO semantics. Note that the qman_irqsource_*() and bman_irqsource_*() APIs must be used prior to entering a blocking read(), select(), poll(), [etc] to configure the relevant QMan/BMan portals to report activity via interrupt. Failure to do so may cause the application to block indefinitely waiting for interrupts that won't occur because the portals are configured for polling. Similarly, when leaving IRQ/blocking mode to run in polling mode (and in particular to post-process the events that caused interrupts), one must again use the "irqsource" APIs, this time to configure the portals to be processed by polling APIs rather than interrupts.

For an illustration of PPAC's use of IRQ mode, see apps/ppac/main.c, specifically the core loop of the worker_fn() function. This loop migrates from polling mode to a blocking "IRQ mode" built around select() whenever reflector has looped a certain number of times without any forward progress. If input traffic is significantly below processing capacity, then it should be possible to observe reflector going in to (and out of) blocking select()s, even if the traffic is never completely stopped. In such cases, any latency associated with sleeping and scheduling is absorbed by system buffering long enough for reflector to wake up and resume polling mode (and catch up on the buffered backlog). This mechanism allows reflector to share CPUs with other tasks more effectively (and consume less power) provided the throughputs are low or busty enough to make this acceptable.

### 38.4.2 Buffers

When starting up, PPAC applications seed the 3 buffer pools that the Fman is configured to use for Rx processing. To improvement restartability, the pools are first drained of any stale contents, after which they are seeded using allocations from the "/dev/fsl_usdpaa_shmem" DMA device, which is described in the USDPAA User Guide.

The three buffer pools initialized and used by PPAC (and the default USDPAA configuration of Fman) have the following attributes.

Table 66: Buffer Pool attributes

| Pool ID | Buffer Size (used by Fman) | Number of Buffers |
|---|---|---|
| 7 | 320 | 0 |
| 8 | 704 | 0 |
| 9 | 1728 | 0x2000 |

When processing IPv4 frames, buffers are allocated by FMan on Rx and then released by FMan after Tx.

The source code specifies the buffer pool requirements in include/internal/conf.h:

```
#define DMA_MEM_BP1_BPID        7
#define DMA_MEM_BP1_SIZE        320
#define DMA_MEM_BP1_NUM         0 /* 0*320==0 (0MB) */
#define DMA_MEM_BP2_BPID        8
#define DMA_MEM_BP2_SIZE        704
#define DMA_MEM_BP2_NUM         0 /* 0*704==0 (0MB) */
#define DMA_MEM_BP3_BPID        9
#define DMA_MEM_BP3_SIZE        1728
#define DMA_MEM_BP3_NUM         0x2000 /* 0x2000*1728==13.5MB) */
#define DMA_MEM_BPOOL \
          (DMA_MEM_BP1_SIZE * DMA_MEM_BP1_NUM + \
          DMA_MEM_BP2_SIZE * DMA_MEM_BP2_NUM + \
          DMA_MEM_BP3_SIZE * DMA_MEM_BP3_NUM) /* (13.5MB) */
```

And the application implements the seeding of these buffers according to the following structure in apps/ppac/main.c:

```
/* Seed buffer pools according to the configuration symbols */
const struct ppac_bpool_static {
          int bpid;
          unsigned int num;
          unsigned int sz;
} ppac_bpool_static[] = {
          { DMA_MEM_BP1_BPID, DMA_MEM_BP1_NUM, DMA_MEM_BP1_SIZE},
          { DMA_MEM_BP2_BPID, DMA_MEM_BP2_NUM, DMA_MEM_BP2_SIZE},
          { DMA_MEM_BP3_BPID, DMA_MEM_BP3_NUM, DMA_MEM_BP3_SIZE},
          { -1, 0, 0 }
};
```

## 38.4.3 Compile-time configuration

PPAC-based application are compiled using a certain set of options that are currently defined in the header located at apps/include/ppac.h. The following describes the most useful options for modification if alternative application behaviour is desired;

### 38.4.3.1 Order preservation

Order preservation is a functionality of the QMan software portal interface that allows processing of Rx FQs across multiple portals to retain order when transmitted corresponding Tx FQs. The technique only applies to

frames dequeued from a given Rx FQ that are all transmitted out the same Tx FQ (which is the case for "reflector", and also the case for "ipfwd" when frames are from the same flow). The mechanism requires two QMan features, "HOLDACTIVE" and "enqueue DCA". The former ensures that a FQ that has been dequeued to a software portal from should remain bound to that portal until all the corresponding DQRR entries have been consumed. The latter ensures that a DQRR entry is consumed by QMan itself once it has dispatched the corresponding enqueue (Tx) command. Together, for any given Rx/Tx FQ pair, the processing via pool-channels and multiple CPUs does not allow frame processing to get out of order. (For more information on this feature, consult the QMan/ BMan API Guide.)

Use of "HOLDACTIVE" is mutually exclusive with another QMan option "AVOIDBLOCK", which is selected by default in PPAC. So to enable order-preservation, one must change the settings from;

```
#undef PPAC_2FWD_HOLDACTIVE
#undef PPAC_2FWD_ORDER_PRESERVATION
#define PPAC_2FWD_AVOIDBLOCK
```

to;

```
#define PPAC_2FWD_HOLDACTIVE
#define PPAC_2FWD_ORDER_PRESERVATION
#undef PPAC_2FWD_AVOIDBLOCK
```

## 38.4.3.2 Monitoring Rx/Tx fill-levels via CGR

If CGR-based monitoring is enabled, then two Congestion Group Records will be configured, with all Rx FQs for all interfaces being subscribed to one, and all Tx FQs being subscribed to the other. The CGRs are not configured to perform any flow-control (ie. no tail-drop nor WRED options are enabled), so this simply allows the user to monitor the overall fill-level of frame queues in the system, in particularly to determine whether build-up is occuring before or after the software-processing phase. The thresholds used for the CGRs are determined from the two constants also defined in ppac.h, PPAC_CGR_RX_PERFQ_THRESH and PPAC_CGR_TX_PERFQ_THRESH. These constants, combined with the number of FQs, define the thresholds for CGR congestion-entry. Qman in turn defines the CGR exit-threshold to be 7/8 the entry-threshold (to provide hysteresis), and so these combined entry/exit events will be logged to stdout independently for the Rx and Tx CGRs.

An extra command, "cli", becomes available in the CLI when this feature is compiled in, which will query and display all the fields of both CGRs. Note however that this option introduces extra latency, and more critically, extra contention within the system. This is because Qman must lock the CGR for each enqueue and dequeue event that relates to it, meaning that the forwarding of a single packet through the system requires 2 lock/unlock pairs for each CGR (thus 4 lock/unlock pairs). A small but noticable performance degradation should be expected when running in this mode. (Real-world use of CGRs would not subscribe all Rx/Tx FQs from all interfaces to a single CGR, so this scalability issue should not be a concern for production software usage of CGR-based congestion management.)

To enable this feature, change;

```
#undef PPAC_CGR
```

to;

```
#define PPAC_CGR
```

## 38.4.3.3 Other settings

Many other settings used by PPAC (and thus PPAC-based apps) are defined in ppac.h but they are less intended for ad-hoc manipulation than those mentioned above. However a curious user may wish to examine some of

these, and perhaps search out their usage within the source-code, in order to see how they are used and explore some of the driver interfaces and application design in this way.

## 38.5 Running reflector

The following comments about executing reflector apply to all PPAM applications - PPAMs may of course define new command-line (and environment-variable) behaviour, but the following behaviour is all generic to PPAC (reflector does not implement any extensions of its own).

After logging in to the p4080 DS environment as "root", the SOURCE_THIS file in the "/root" directory can be used to simplify running "fmc" (to configure FMan for the required network device configuration) and starting up reflector, as can be seen from the following:

```
login: root
Password:
[root@p4080 root]# cat /root/SOURCE_THIS
cd /usr/etc
fmc -c usdpaa_config_p4_serdes_0xe.xml -p usdpaa_policy_hash_ipv4.xml -a
reflector
[root@p4080 root]# source /root/SOURCE_THIS
Found /fsl,dpaa/dpa-fman0-oh@1, Tx Channel = 46, FMAN = 0, Port ID = 1
Found /fsl,dpaa/ethernet@4, Tx Channel = 40, FMAN = 0, Port ID = 0
Found /fsl,dpaa/ethernet@7, Tx Channel = 63, FMAN = 1, Port ID = 2
Found /fsl,dpaa/ethernet@8, Tx Channel = 64, FMAN = 1, Port ID = 3
Found /fsl,dpaa/ethernet@9, Tx Channel = 60, FMAN = 1, Port ID = 0
Qman: FQID allocator includes range 512:128
Bman: BPID allocator includes range 56:8
Configuring for 4 network interfaces and 4 pool channels
FSL dma_mem device mapped (phys=0xe0000000,virt=0x70000000,sz=0x1000000)
Thread uid:0 alive (on cpu 1)
Release 0 bufs to BPID 7
Release 0 bufs to BPID 8
Release 8192 bufs to BPID 9
reflector starting
Thread uid:0 alive (on cpu 1)
reflector>
```

Reflector starts up with a single thread running on CPU 1 by default, with all the network interfaces enabled. The CLI (Command-Line Interface) allows you to add and remove additional threads to enable the use of multiple CPUs, with the only restriction being that the primary thread on CPU 1 can not be removed (except by shutting down the application).

Reflector can alternatively be started as a single thread on a different CPU by executing it with the desired CPU as its sole argument:

```
[root@p4080 root]# reflector 5
```

Alternatively, multiple threads can be started by specifying a range of CPUs:

```
[root@p4080 root]# reflector 3..7
```

By default, reflector is compiled to load the XML configuration and PCD files passed to the "fmc" tool in the afore-mentioned SOURCE_THIS script. Indeed, it is important that reflector always load the same configuration as is passed to "fmc". If "fmc" is run with different XML inputs, eg. when running on another board than p4080ds and/ or when using a different SERDES configuration, then reflector must be instructed to load non-default XML files to match. This can be achieved either by setting the DEF_PCD_PATH and DEF_CFG_PATH environment

variables, or by using short or long command-line arguments: Eg. the following 3 examples achieve the same thing:

```
[root@p4080 root]# reflector -c my_cfg.xml -p my_pcd.xml

[root@p4080 root]# reflector --fm-config my_cfg.xml --fm-pcd my_pcd.xml

[root@p4080 root]# export DEF_CFG_PATH=my_cfg.xml
[root@p4080 root]# export DEF_PCD_PATH=my_pcd.xml
[root@p4080 root]# reflector
```

Note also that PPAC implements a CLI for all PPAM applications, which means that it expects a console to be present on 'stdin'. If the reflector process is backgrounded, or run as a daemon, then the process will pause waiting for it to receive control of console input. If the user wishes to launch reflector or any other PPAM without the use of the console, they should instruct it ignore input and not run the CLI by passing the "-n" or "--non-interactive" command-line arguments:

```
[root@p4080 root]# reflector --non-interactive &
```

## 38.6 PPAC (and reflector) CLI commands

The following commands are illustrated in the context of reflector (carrying on from the session started in the previous section), but the commands are common to all PPAC-based applications.

To add a thread on a single CPU (e.g. CPU 2):

```
reflector> add 2
```

To add threads on a range of CPUs:

```
reflector> add 3..6
```

To list the threads (this also queries each thread, verifying that they aren't blocked):

```
reflector> list
Thread uid:0 alive (on cpu 1)
Thread uid:1 alive (on cpu 2)
Thread uid:2 alive (on cpu 3)
Thread uid:3 alive (on cpu 4)
Thread uid:4 alive (on cpu 5)
Thread uid:5 alive (on cpu 6)
```

To remove a thread by its UID:

```
reflector> rm uid:2
Thread uid:2 killed (cpu 3)
```

To remove a thread running on a given CPU:

```
reflector> rm 4
Thread uid:3 killed (cpu 4)
```

If more than one thread is running on a given CPU (only possible if the device-tree has been updated to reserve multiple portals for USDPAA on the same CPU), then this removes the first thread found running on the named CPU. A range of CPUs can likewise be specified:

```
reflector> rm 5..6
Thread uid:4 killed (cpu 5)
Thread uid:5 killed (cpu 6)
```

To perform a controlled shutdown of reflector (this includes disabling the network ports):

```
reflector> quit
```

## 38.7 Running hello_reflector

The stand-alone "hello_reflector" application, which is not PPAC-based, can be executed in exactly the same environment as the PPAC-based "reflector". Instead of executing "reflector", one executes "hello_reflector" instead:

```
[root@p4080 root]# hello_reflector
```

Hello_reflector always starts up on CPU 0, and by default it only starts a single thread. Multiple threads can be started by providing the "-n" argument, in which case that many CPUs will be used starting with CPU 0 and counting upwards.

```
[root@p4080 root]# hello_reflector -n 5
```

Hello_reflector does not implement a CLI, so the only way to gracefully shut it down is to enter a Ctrl-C on the controlling console, or equivalently issue a SIGINT signal to the hello_reflector process.

Alternative configuration and PCD files can be provided to hello_reflector by using the "-c" and "-p" arguments respectively. These should always match the XML files that are passed to "fmc":

```
[root@p4080 root]# hello_reflector -p my_pcd.xml -c my_cfg.xml
```

## 38.8 Running hello_reflector (short circuit)

Issue following command to run hello_reflector in "short circuit" mode:

```
[root@p4080 root]# hello_reflector -c /usr/etc/
usdpaa_config_p4_serdes_0xe.xml -p /usr/etc/usdpaa_policy_hash_ipv4.xml -n 1 -sc
```

To test sc mode, stop the process (ctrl + Z) and feed the traffic to FMAN port (using spirent), you will still see the rx traffic due to the fact that the destination channel for RX queues is tx_channel.

Follow these steps precisely :

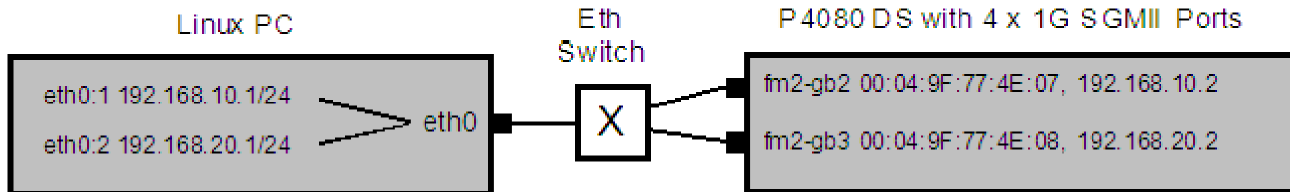
1. Hang up process by ctrl+z, this will avoid using cpu and will not impact traffic flow.

2. To quit process, you have to recover process by 'fg %1' then press 'ctrl+c' to quit.

## 38.9 Testing reflector

Functional testing of the "reflector" application is possible by connecting any subset of the P4080 USDPAA network interfaces to a conventional computer, assumed to be a Linux PC using an ethernet switch as shown in the figure below.

Figure 59: Testing Reflector




The figure assumes that the default USDPAA SerDes 0xe reset configuration word (RCW) is used. This configuration provides 2 x 10 Gbps and 2 x 1 Gbps ethernet interfaces to the USDPAA application. The test case in the figure assumes that only the 2 x 1 Gbps interfaces will be used. These are the two interfaces on the SGMII riser card that are closest to the P4080DS motherboard. This test will work even if 10 Gbps XAUI riser cards are not fitted in the P4080DS.

See the main USDPAA User Guide for more information on network interfaces.

To perform the test, boot the P4080 and run the example application as described in section Running reflector on page 638.

The IP and ethernet MAC addresses used below are examples. You can change them as long as you are consistent. The most important thing is to be sure of the MAC addresses on the P4080DS board. Again, see the main USDPAA User Guide.

On the Linux PC, create eth0:1 through eth0:3 via, for example:

```
sudo ifconfig eth0:1 192.168.10.1 netmask 255.255.255.0
sudo ifconfig eth0:2 192.168.20.1 netmask 255.255.255.0
```

The reflector application does not respond to ARP requests, so static ARP entries for all of the P4080 USDPAA interfaces must be created on the Linux PC:

```
sudo arp -s 192.168.10.2 00:04:9F:77:4E:07
sudo arp -s 192.168.20.2 00:04:9F:77:4E:08
```

Then, from the Linux PC ping one of the P4080 interfaces, e.g. "ping 192.168.10.2". This causes the following sequence:

1. Linux PC sends ICMP echo request to the switch.
2. At least for the first ping, the switch will flood the request to all P4080 ports. It will be dropped by all but the correct one.
3. The P4080 receives the frame, swaps the IP and MAC addresses, and sends the frame back out the MAC it came in on.
4. The Linux PC receives the frame, an ICMP echo request.
5. The Linux PC responds to the request.

6. The response is received by the P4080 which swaps the IP and MAC addresses and sends the frame back out on the MAC it came in on.
7. The Linux receives the response.
8. The original ping is now complete. The Linux PC just "pinged itself via the P4080".

It is possible to also test with different packet types using this same technique. For example, have a telnet server running on the Linux PC and then do "telnet 192.168.10.2" from the Linux PC. The Linux PC will telnet to itself via the P4080. One can also use ssh in the same way.

To test performance, it is best to used dedicated ethernet traffic generation equipment such as a Spirent Test Center. Such equipment can inject packets into the P4080 at known and very high rates and measure the rate of the packets that egress from the P4080.

# Chapter 39
# Freescale USDPAA IPFWD User Manual Rev. 1.2

## 39.1 About this Book

The User Space Datapath Acceleration Architecture (USDPAA) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document provides the following:

- A summary of the USDPAA IPFwd application
- Execution steps for USDPAA IPFwd application from the Freescale yocto package on the P4080DS/ P3041DS/P5020DS/T4240QDS/B4860QDS/B4420QDS.

Conventions

This document uses the following conventions:

Courier is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.

## 39.2 Introduction

### 39.2.1 Purpose

This document describes the USDPAA IPv4 forwarding application. This application documents the USDPAA IPv4 demonstration applications forwarding flow and its performance measured on a P4080DS.

## 39.3 Overview

The USDPAA IPv4 forwarding (IPFwd) application is a multi-threaded application that routes IPv4 packets from one ethernet interface to another on all QorIQ platforms. The routing is done based on the source IP address and destination IP address in the frame. Any combination of the cores can run a USDPAA IPv4 Forwarding application thread in USDPAA SDK v1.1.

IPFwd packet-processing:

· Receives ethernet frames on all USDPAA ethernet interfaces.

· Discards (and deallocates buffers for) all non-IPv4 frames.

· For IPv4 frames processing takes place as defined in section Overview of packet flow: on page 644

### 39.3.1 USDPAA IPv4 forwarding application flow

The IPFwd application has two main phases. There is an initial configuration phase and a subsequent packet processing phase.

The configuration phase executes when the application starts. Application threads are created and global initialization of resources is done which is the part of PPAC (see USDPAA PPAC User Guide for more details).

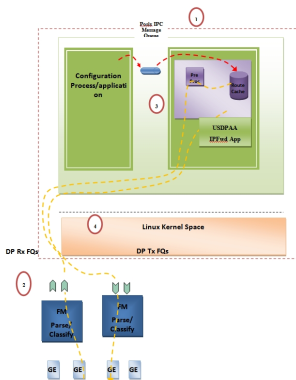
The configuration phase also includes PPAM (i.e. IPFwd) related initialization. Once the configuration phase is completed the IPFwd application moves to the packet processing phase. This application provides a command-line interface to enable users to add and remove routing table and ARP cache entries at any given time. For each user input, the appropriate information is communicated to the IPFwd application via standard Posix IPC. Messages are placed onto the message queue till they are received by the IPfwd application. Note that the IPFwd application does not dynamically resolve ARP – missing ARP entries will result in the application dropping the packet.

In the packet processing phase, the loop migrates from polling mode to a blocking “IRQ mode” whenever IPFwd has looped a certain number of times without any forward progress. For more information on IRQ mode please refer to “USDPAA PPAC User Guide”. In polling mode – the application constantly looks for data to process on its dedicated QMan portal . Network traffic is classified and distributed by the FMan to frame queues based on source and destination IP address in the packet. There is an associated handler that processes the packets arriving on each frame queue.

#### 39.3.1.1 Overview of packet flow:

1. Route table entries are populated using the IPFwd configuration commands at any given time.
2. Packet is received by the FMan, which uses 2-tuple (Src IP & Dest IP) to hash the packet to a Rx frame queue.
3. The classified packet is presented to the USDPAA IPFwd application thread running on one of the cores - the distribution is based on the portal and FQ setup done by the application during its portal and frame queue initialization. The packet is subjected to IPv4 Forwarding route lookup.
4. Based on the packet destination, the application transmits the packet by enqueueing it on a frame queue destined for the appropriate Tx interface.

Figure 60: Packet flow for USDPAA IPv4 Forwarding




## 39.4 Overview of PPAC

The source code to IPFwd has been reorganized into two parts; the “PPAC” (Packet-Processing Application Core) and a “PPAM” (Packet-Processing Application Module). The PPAM portion implements the IPFWD application specific logic of processing the packet to forward it. On the other hand, the PPAC component represents the common infrastructure to support PPAM; initializing devices, handling flow-control, implementing a CLI (Command-Line Interface), managing threads and buffers. PPAC details can be found in the document “USDPAA PPAC User Guide”.

## 39.5 IPFwd related PPAC Details

### 39.5.1 Compile-time configuration

PPAC-based application are compiled using a certain set of options that are currently defined in the header located at apps/include/ppac.h. The following describes the most useful options for modification if alternative application behaviour is desired.

#### 39.5.1.1 Order Preservation in IPFWD

This section describes how user can enable Order Preservation in IPFWD application. By default Order Preservation is disabled in IPFWD application and in order to enable it the user will have to re-compile the binary by making following changes to the source code.

In file, usdpaa/apps/include/ppac.h you can find these two lines.

/* Application options */

#undef PPAC_2FWD_HOLDACTIVE /* Process each FQ on one portal at a time */

#undef PPAC_2FWD_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */

Change the above to

#define PPAC_HOLDACTIVE /* Process each FQ on one portal at a time */

#define PPAC_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */

And then compile usdpaa once again. Now run the IPFWD application with Order Preservation.

#### 39.5.1.2 Order Restoration in IPFWD

Oder restoration is the functionality of QMan software portal interface which restores the relative temporal order of a flow of frames (sequence of frames) to that observed before transmitting to the destination Frame Queue and Order Definition Point (ODP) takes note of the correct order of packets before start processing by using the sequence number. Use of “HOLDACTIVE” is mutually exclusive with another QMan option “AVOIDBLOCK”, which is selected by default in PPAC. To enable order-restoration, the user will have to re-compile the binary by making following changes to the source code.

/* Application options */

#undef PPAC_2FWD_HOLDACTIVE /* Process each FQ on one portal at a time */

#undef PPAC_2FWD_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */

#undef PPAC_2FWD_ORDER_RESTORATION /* Use ORP */

#define PPAC_2FWD_AVOIDBLOCK /* No full-DQRR blocking of FQs */

Change the above to

#undef PPAC_HOLDACTIVE /* Process each FQ on one portal at a time */

#undef PPAC_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */

#define PPAC_ORDER_RESTORATION /* Use ORP */

#define PPAC_AVOIDBLOCK /* No full-DQRR blocking of FQs */

And then compile usdpaa once again. Now run the IPFWD application with Order Restoration. Implementation note: Order restoration has been implemented such that each PCD Frame queue has a corresponding ORP

(order restoration point) frame queue associated with it. Each ORP is configured with default window settings as seen below

#define PPAC_ORP_WINDOW_SIZE 7 /* 0->32, 1->64, 2->128, ... 7->4096 */

#define PPAC_ORP_AUTO_ADVANCE 1 /* boolean */

#define PPAC_ORP_ACCEPT_LATE 3 /* 0->no, 3->yes (for 1 & 2->see RM) */

Here the ORP window size is set to be 4K, auto advance window size as 4K and accept late arrival window size as 8K. This ensures that no traffic is getting dropped but are always accepted below and at Zero loss throughput. Beyond zero loss throughput, as usual packets would be dropped and thus you can see mis-ordering.

ORP FQ descriptor attributes settings:

· Prefer in cache

· No “ HOLDACTIVE ”

· No “ AVOIDBLOC K”

· ORP enabled

Assumption: To see the effect of Order Restoration in IPFwd application the user must use separate streamblocks as a source of traffic. If not done so, mis-ordering would be seen.

Key observation: It has been observed in IPFwd application that use of “HOLDACTIVE” with traffic generated using separate streamblocks, all the packets are IN sequence. Therefore, it is recommended that if user wants to see the real effect of Order restoration in IPFwd application he should use “AVOIDBLOCK” with “RESTORATION” and not “HOLDACTIVE” with “RESTORATION”

## 39.5.1.3 Monitoring Rx/Tx fill-levels and flow-control via CGR

If CGR-based monitoring is enabled, then two Congestion Group Records will be configured, with all Rx FQs for all interfaces being subscribed to one, and all Tx FQs being subscribed to the other. This simply allows the user to monitor the overall fill-level of frame queues in the system, in particular to determine whether build-up is occuring before or after the software-processing phase. Refer to section “ Monitoring Rx/Tx fill-levels via CGR ” of USDPAA PPAC User Guide for more details. To enable this feature, in ppac.h change;

#undef PPAC_CGR

to;

#define PPAC_CGR

The CGRs can also be configured to perform flow-control using Congestion state tail drop by setting CSTD_EN bits . Each congestion group record can be configured to track either byte counts or frame counts in all frame queues in the Congestion Group. When the threshold set for each CGR is exceeded, the CS bit is set in the CGR, and the congestion group is said to have entered congestion. At this point the incoming frames are marked for discard and QMAN will generate enqueue rejections to the producer. When the group’s I_BCNT returns below the threshold (minus approximately 1/8 of the threshold to provide hysteresis), the CS bit is cleared, and the congestion group’s state exits congestion. To enable tail drop, in ppac.h change;

#undef PPAC_CGR /* Track rx and tx fill-levels via CGR */

#undef PPAC_CSTD /* CGR tail-drop */

#undef PPAC_CSCN /* Log CGR state-change notifications */

to

#define PPAC_CGR /* Track rx and tx fill-levels via CGR */

#define PPAC_CSTD /* CGR tail-drop */

#undef PPAC_CSCN /* Log CGR state-change notifications */

And then compile usdpaa once again. Now run the IPFWD application with CGR tail drop enabled. To test this feature PPAC CLI provides a command "cgr" which will query and display all the fields of both CGRs. On pumping the traffic to IPFWD application at full line rate, t he instantaneous group byte count value I_BCNT(Instantaneous frame/byte count) must be maintained lesser than the CGR threshold set for each congestion group. Here is one such cgr command output:

> cgr

Rx CGR ID: 10, selected fields;

cscn_en: 0

cscn_targ: 0x00800000

cstd_en: 1

cs: 0

cs_thresh: 0x00_0000_1000

mode: 1

i_bcnt: 0x00_0000_0e1e

a_bcnt: 0x00_0000_0e1e

Tx CGR ID: 11, selected fields;

cscn_en: 0

cscn_targ: 0x00800000

cstd_en: 1

cs: 0

cs_thresh: 0x00_0000_0200

mode: 1

i_bcnt: 0x00_0000_0002

a_bcnt: 0x00_0000_0004

On the other hand if this feature of Congestion Group tail drop is disabled in IPFWD application I_BCNT is never maintained below CGR threshold value with traffic at full line-rate. This can be checked by compiling the IPFWD application with;

#define PPAC_CGR /* Track rx and tx fill-levels via CGR */

#undef PPAC_CSTD /* CGR tail-drop */

#undef PPAC_CSCN /* Log CGR state-change notifications */

Now on pumping traffic at full line-rate, atleast one of the CGRs must go into congestion state and its I_BCNT should be above CGR threshold value. Here is the sample output:

> cgr

Rx CGR ID: 10, selected fields;

cscn_en: 1

cscn_targ: 0x00800000

cstd_en: 0

cs: 0

cs_thresh: 0x00_0000_1000

mode: 1

i_bcnt: 0x00_0000_0006

a_bcnt: 0x00_0000_0004

Tx CGR ID: 11, selected fields;

cscn_en: 1

cscn_targ: 0x00800000

cstd_en: 0

cs: 1

cs_thresh: 0x00_0000_0200

mode: 1

i_bcnt: 0x00_0000_5fb7

a_bcnt: 0x00_0000_5fb8

Thus, the above test showcases the flow control achieved by enabling congestion group tail drop in IPFWD application.

# 39.6 PPAM related compile time configuration

## 39.6.1 One million route support

By default IPfwd application can work only with 1K routes. The user may want to run it for higher number of routes, so there is an option available in the header located at apps/ipfwd/include/app_common.h. To enable one million route support, the user will have to re-compile the binary by making following changes to the source code.

#undef ONE_MILLION_ROUTE_SUPPORT

Change the above line to

#define ONE_MILLION_ROUTE_SUPPORT

And then compile usdpaa once again. Now run the IPFWD application for one million routes. There is a sample shell script available at /usr/bin/ipfwd_20G_1Mroutes.sh in the rootfs. It creates one million routes using the 2 x 10G interfaces. It can assign ip addresses to the interfaces, add an ARP entry and assumes the netmask to be 255.255.255.0. The following table summarizes the setting done by this script.

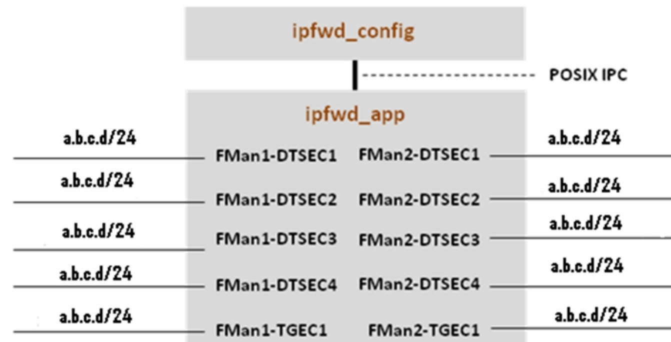
| Port ID | Source IP Address | Destination IP Address (for each src IP addr) | Default Gateway IP Address |
|---|---|---|---|
| FM1:TGEC1 | 192.168.24.2 to 192.168.24.255 | 192.168.29.2 .. 255 | 192.168.29.2 |
| | 192.168.25.2 to 192.168.25.255 | 192.168.29.2 .. 255 | 192.168.29.2 |

*Table continues on the next page...*

*Table continued from the previous page...*

| | | | |
|---|---|---|---|
| | 192.168.26.2 to 192.168.26.255 | 192.168.29.2 .. 255 | 192.168.29.2 |
| | 192.168.27.2 to 192.168.27.255 | 192.168.29.2 .. 255 | 192.168.29.2 |
| | 192.168.28.2 to 192.168.28.255 | 192.168.29.2 .. 255 | 192.168.29.2 |
| | 192.168.1.2 to 192.168.1.255 | 192.168.29.2 .. 255 | 192.168.29.2 |
| | 192.168.18.2 to 192.168.18.255 | 192.168.29.2 .. 255 | 192.168.29.2 |
| | 192.168.30.2 to 192.168.30.255 | 192.168.29.2 .. 255 | 192.168.29.2 |
| | 192.168.31.2 to 192.168.31.91 | 192.168.29.2 .. 91 | 192.168.29.2 |
| FM2:TGEC2 | 192.168.29.2 to 192.168.29.255 | 192.168.24.2 .. 255 | 192.168.24.2 |
| | 192.168.2.2 to 192.168.2.255 | 192.168.24.2 .. 255 | 192.168.24.2 |
| | 192.168.3.2 to 192.168.3.255 | 192.168.24.2 .. 255 | 192.168.24.2 |
| | 192.168.4.2 to 192.168.4.255 | 192.168.24.2 .. 255 | 192.168.24.2 |
| | 192.168.5.2 to 192.168.5.255 | 192.168.24.2 .. 255 | 192.168.24.2 |
| | 192.168.6.2 to 192.168.6.255 | 192.168.24.2 .. 255 | 192.168.24.2 |
| | 192.168.17.2 to 192.168.17.255 | 192.168.24.2 .. 255 | 192.168.24.2 |
| | 192.168.19.2 to 192.168.19.255 | 192.168.24.2 .. 255 | 192.168.24.2 |
| | 192.168.20.2 to 192.168.20.91 | 192.168.24.2 .. 91 | 192.168.24.2 |

# 39.7 IPFWD Application Suite




The figure above shows the structure of the IPFWD USDPAA application suite. Its purpose is to forward IPv4 packets between the interfaces shown. No IP address is assigned to any of the interfaces by default. Using ipfwd_config command as mentioned in section Assign IP address to interfaces on page 668 IP address can be assigned to all these interfaces. Each interface has a fixed netmask shown in the figure. The notation "/24" refers to a netmask of 255.255.255.0. The MAC addresses of these interfaces are determined by u-boot environment variables ethaddr, eth1addr, eth2addr, etc.

The ipfwd application will respond to ARP requests on these interfaces. However, the application will not generate ARP requests to determine the destination MAC address of forwarded packets. An ipfwd_config command should be used to set these MAC addresses.

It is important to understand that the application can use only a subset of these interfaces at any one time. This is due to pin multiplexing rules on the P4080 SoC. The set of available interfaces is determined by a number of factors:

1. The interface set made available by the selected SerDes Protocol and u-boot variable "hwconfig". See the SDK document "Freescale DPAA SDK X.Y: Selecting Ethernet Interfaces". This document is distributed with the DPAA SDK.
2. The Linux device tree determines which subset of the available interfaces is for use by USDPAA applications. See the USDPAA User Guide for more information.
3. Finally, the fmc configuration file passed by command line argument to ipfwd_app determines the subset of these interfaces that the application will attempt to initialize.

In the default SerDes 0xe example, the interfaces used by ipfwd_app are:

- FMan1-TGEC1
- FMan2-DTSEC3
- FMan2-DTSEC4
- FMan2-TGEC1

Running the ipfwd application suite involves four steps:

1. Run the fmc application to configure the FMan hardware instances.
2. Run ipfwd_app
3. Run ipfwd_config repeatedly to add routes to ipfwd's route cache.
4. Run "ipfwd_config -O" to start application processing.

Specific examples showing these steps are provided in other sections of this document.

# 39.8 Possible configuration scenario for IPFWD

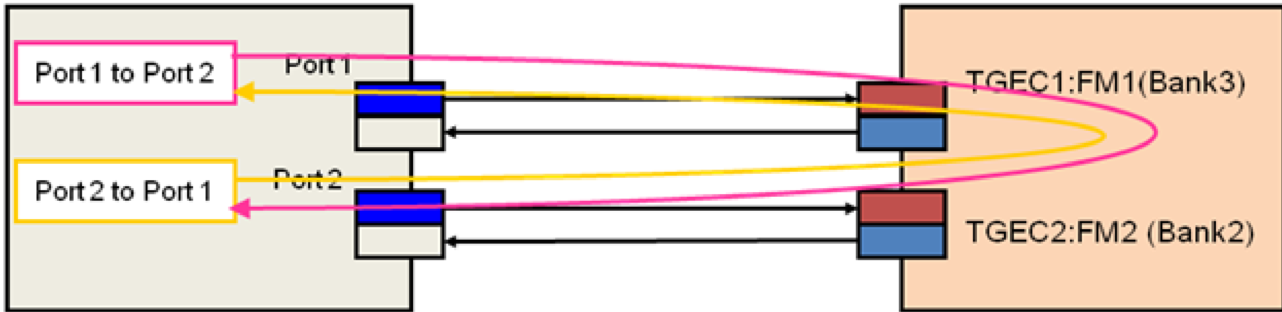
IPfwd application can run in different configuration scenario. The table below shows the configuration files and corresponding sample shell script existing in the repository.

| xml file | Sample shell script | Number of routes (as per sample shell script) |
|---|---|---|
| usdpaa_config_p4_serdes_0xe.xml (2x10G+2x1G) | ipfwd_20G.sh | 1012 routes (2x10G) |
| | ipfwd_22G.sh | 1022 routes (2x10G+2x1G) |
| usdpaa_config_p3_p5_serdes_0x36.xml (5x1G+10G) | ipfwd_15G.sh | 1000 routes (5x1G+10G) |

Figure 1: usdpaa_config_p4_serdes_0xe.xml

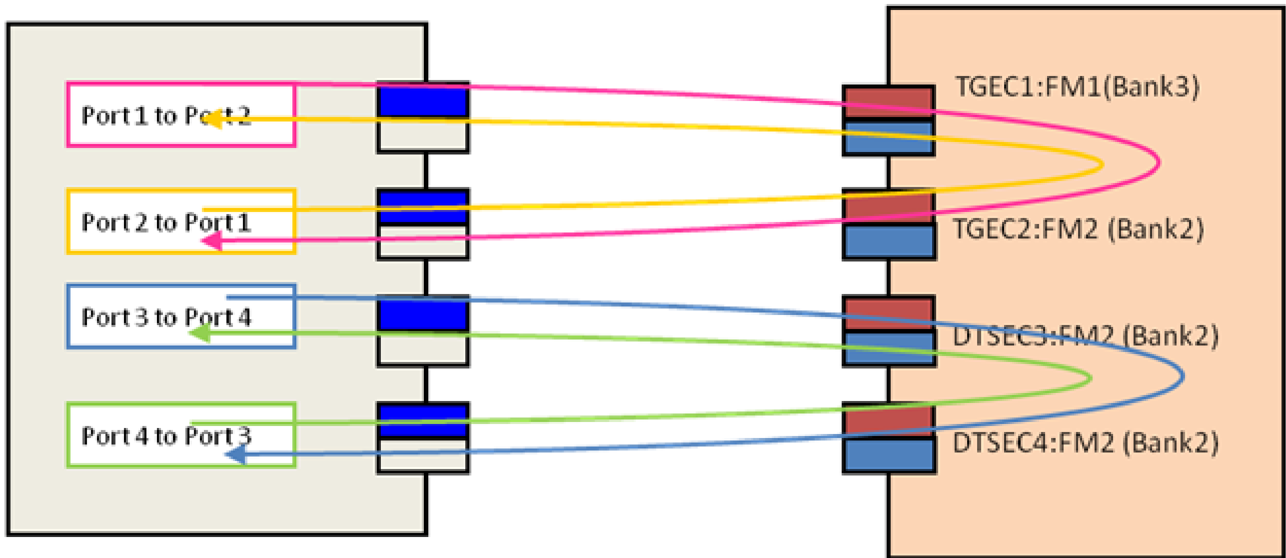
The figure above shows the ethernet interfaces available as per configuration file " usdpaa_config_p4_serdes_0xe.xml". It contains the 1 RGMII port (FMAN1, DTSEC 2), 2 SGMII ports ( FMan2, DTSECs 3- 4 ) and 2 XAUI ports (FMAN1, TGEC1 and FMAN2-TGEC2). It is the user's wish to use any combination of ports available with the configuration file. The above table shows the sample shell scripts that user can use for this configuration. If user uses 2x10G, following is the flow configuration.

Test Center <-> P4080




**Port 1 to Port 2 (506 flows):**
**Src: 192.168.60.2 - 192.168.60.23**
**Dst: 192.168.160.2 - 192.168.160.24**
**Port 2 to Port 1 (506 flows):**
**Src: 192.168.160.2 - 192.168.160.23**
**Dst: 192.168.60.2 - 192.168.60.24**

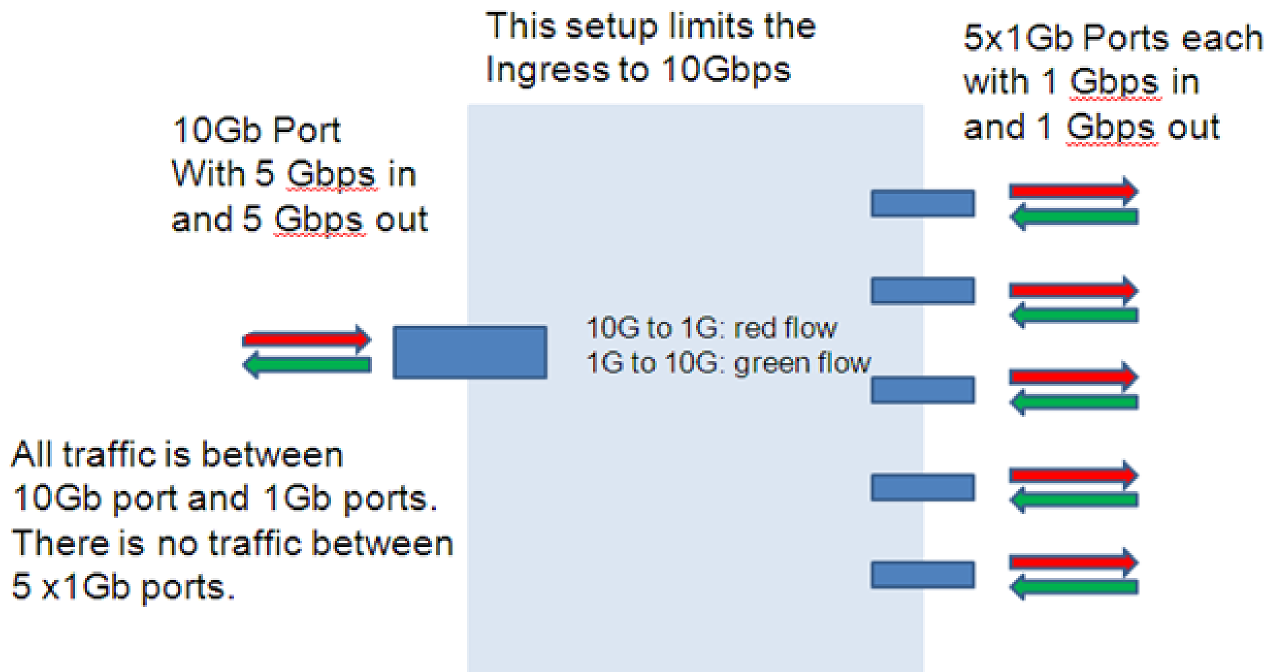
Figure 2 : Flow Configuration for 2x10G on P4080DS



```
Port 1 to Port 2 (462 flows):
Source: 192.168.60.2 – 192.168.60.22
Destination: 192.168.160.2 – 192.168.160.23
Port 2 to Port 1 (462 flows) :
Source: 192.168.160.2 – 192.168.160.22
Destination: 192.168.60.2 – 192.168.60.23
Port 3 to Port 4 (46 flows):
Source: 192.168.130.2 – 192.168.130.22
Destination: 192.168.140.2 – 192.168.140.23
Port 2 to Port 1 (46 flows) :
Source: 192.168.140.2 – 192.168.140.22
Destination: 192.168.130.2 – 192.168.130.23
```

Figure 3 : configuration for 2x10G and 2x1G on P4080

For running IPFwd on P3041DS/P5020DS, the user can use “ usdpaa_config_p3_p5_serdes_0x36.xml”. Following is the flow configuration.



Left: 10G port (port6: fm1-10g)
Right: top-to-bottom: 1G port (port1-port6)

Figure 4 : Flow Configuration for 10G on P5020DS and P3041DS

```
Port 1 to Port 6 (100 flows):
                Src: 192.168.10.2 - 192.168.10.11
                Dst: 192.168.60.2 - 192.168.60.11

Port 2 to Port 6 (100 flows):
                Src: 192.168.20.2 - 192.168.20.11
                Dst: 192.168.60.2 - 192.168.60.11

Port 3 to Port 6 (100 flows):
                Src: 192.168.30.2 - 192.168.30.11
                Dst: 192.168.60.2 - 192.168.60.11

Port 4 to Port 6 (100 flows):
                Src: 192.168.40.2 - 192.168.40.11
                Dst: 192.168.60.2 - 192.168.60.11

Port 5 to Port 6 (100 flows):
                Src: 192.168.50.2 - 192.168.50.11
                Dst: 192.168.60.2 - 192.168.60.11
```

```
Port 6 to Port 1 (100 flows):
                Src: 192.168.60.2 - 192.168.60.11
                Dst: 192.168.10.2 - 192.168.10.11

Port 6 to Port 2 (100 flows):
                Src: 192.168.60.2 - 192.168.60.11
                Dst: 192.168.20.2 - 192.168.20.11

Port 6 to Port 3 (100 flows):
                Src: 192.168.60.2 - 192.168.60.11
                Dst: 192.168.30.2 - 192.168.30.11

Port 6 to Port 4 (100 flows):
                Src: 192.168.60.2 - 192.168.60.11
                Dst: 192.168.40.2 - 192.168.40.11

Port 6 to Port 5 (100 flows):
                Src: 192.168.60.2 - 192.168.60.11
                Dst: 192.168.50.2 - 192.168.50.11
```

### How to run if user has no XAUI but only SGMII card?

Assume user does not have a XAUI card and wants to run IPFwd using only the SGMII ports as shown below.

This configuration contains the 1 RGMII port (FMAN1, DTSEC 2), 4 SGMII ports ( FMan2, DTSECs 1- 4 ). The user can modify the configuration file and sample shell scripts as per requirement. The configuration file would contain the following

```
<cfgdata>
<config>
<engine name="fm1">
<port type="1G" number="0" policy="hash_ipsec_src_dst_spi_policy6"/>
<port type="1G" number="1" policy="hash_ipsec_src_dst_spi_policy7"/>
<port type="1G" number="2" policy="hash_ipsec_src_dst_spi_policy8"/>
<port type="1G" number="3" policy="hash_ipsec_src_dst_spi_policy9"/>
</engine>
</config>
</cfgdata>
```

User can create a new shell script which would make routes between the 4x1G. Here is the content of the script.

```
net_pair_routes()
{
for net in $1 $2
    do
ipfwd_config -P $pid -B -s 192.168.$net.2 -c $3 \
-d 192.168.$(expr $1 + $2 - $net).2 -n $4 -g \
192.168.$(expr $1 + $2 - $net).2
done
}
```

ipfwd_config -P $pid -F -a 192.168.110.1 -i 6

ipfwd_config -P $pid -F -a 192.168.120.1 -i 7

ipfwd_config -P $pid -F -a 192.168.130.1 -i 8

ipfwd_config -P $pid -F -a 192.168.140.1 -i 9

ipfwd_config -P $pid -G -s 192.168.110.2  -m 02:00:c0:a8:6e:02 -r true

ipfwd_config -P $pid -G -s 192.168.120.2 -m 02:00:c0:a8:78:02 -r true

ipfwd_config -P $pid -G -s 192.168.130.2 -m 02:00:c0:a8:82:02 -r true

ipfwd_config -P $pid -G -s 192.168.140.2 -m 02:00:c0:a8:8c:02 -r true

# 1024

net_pair_routes 110 120  16  16   # 2 *  16 *  16 =  512

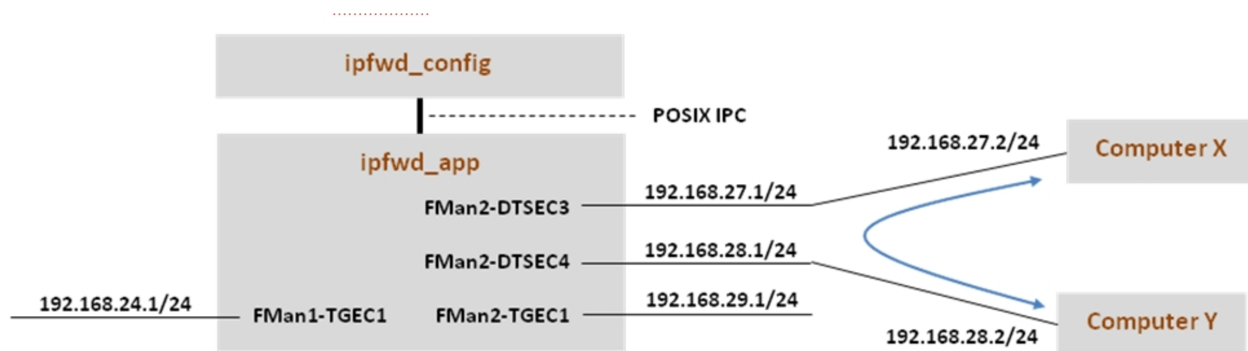
net_pair_routes 130 140 16 16   # 2 * 16 * 16 = 512

echo IPFwd Route Creation completed

Now traffic can be run as per the routes created.

## 39.9 Using Two Computers to Test the IPFWD Application Suite

This section describes how to configure the IPFWD application suite to forward packets between two ordinary computers as shown in the following figure. It is assumed that the two 1 Gbps Ethernet interfaces provided by SerDes 0xe are used.




Assign IP addresses to all the interfaces. The IP addresses used here for Computer X and Computer Y are examples. Their MAC addresses must be known as they will be needed in the commands below.

Keep in mind that ipfwd_app has no default IP address assigned to the interfaces. This means that you must first assign IP addresses to the interfaces and then use IP addresses for Computer X and Computer Y that are on the subnets shown in the figure. Please be careful with the network parameters. Any mistake will prevent packets from flowing from end to end.

Follow these steps on Computer X:

1. Set the ip address for eth0 interface

ifconfig eth0 192.168.27.2 up

Set the default gateway for subnet 192.168.27.1

route add default gw 192.168.27.1 eth0

2. Add arp for gw

arp –s 192.168.27.1 “MAC address for 192.168.27.1 on P4080”

Follow these steps on Computer Y:

1. Set the ip address for eth0 interface

ifconfig eth0 192.168.28.2 up

Set the default gateway for subnet 192.168.28.1

route add default gw 192.168.28.1 eth0

2. Add arp for gw

arp –s 192.168.28.1 “MAC address for 192.168.28.1 on P4080”

The commands to perform on P4080 are:

# Boot the P4080 and login as root.

*Assign IP address to fm1-gb1*

ifconfig fm1-gb1 <IPADD> up

cd /usr/etc

fmc -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_32_fq.xml -a

# Assume use of cores 1 - 7 ipfwd_app 1..7

# Now ssh to P4080 linux on other terminal

ssh root@<IPADD>

give IP address as assigned to fm1-gb1 in the beginning

# Now assign ip address to the interfaces

# First run command to check what all enabled interfaces are available

Note: check pid from application print "Message queue to send: /mq_snd_2536"

ipfwd_config -P 2536 -E -a true

Interface number: 11

PortID=1:5 is FMan interface node

with MAC Address

02:00:c0:a8:65:fe

Interface number: 9

PortID=1:3 is FMan interface node

with MAC Address

02:00:c0:a8:5b:fe

Interface number: 8

PortID=1:2 is FMan interface node

with MAC Address

02:00:c0:a8:51:fe

Interface number: 5

PortID=0:5 is FMan interface node

with MAC Address

02:00:c0:a8:33:fe

Are all the Enabled Interfaces

# Assign IP address to the interfaces.Use the same interface number displayed as an ouput on giving the above command

ipfwd_config -P 2536 -F -a 192.168.24.1 -i 5

ipfwd_config -P 2536 -F -a 192.168.28.1 -i 9

ipfwd_config -P 2536 -F -a 192.168.27.1 -i 8

ipfwd_config -P 2536 -F -a 192.168.29.1 -i 11

# Now enter routes and MAC addresses. Format of a MAC address is

# aa:bb:cc:dd:ee:ff where the letters are hexadecimal digits.

ipfwd_config -P 2536 -B -s 192.168.27.2 -d 192.168.28.2 -g 192.168.28.2

ipfwd_config -P 2536 -G -s 192.168.28.2 -m COMPUTER_Y_MAC_ADDRESS -r true

ipfwd_config -P 2536 -B -s 192.168.28.2 -d 192.168.27.2 -g 192.168.27.2

ipfwd_config -P 2536 -G -s 192.168.27.2 -m COMPUTER_X_MAC_ADDRESS -r true

# Computer X and Computer Y need to be told to route via the P4080.

# On Computer X (assuming it runs Linux), enter this command as root:

route add -net 192.168.28.0 netmask 255.255.255.0 gw 192.168.27.1

# On Computer Y (assuming it runs Linux), enter this command as root:

route add -net 192.168.27.0 netmask 255.255.255.0 gw 192.168.28.1

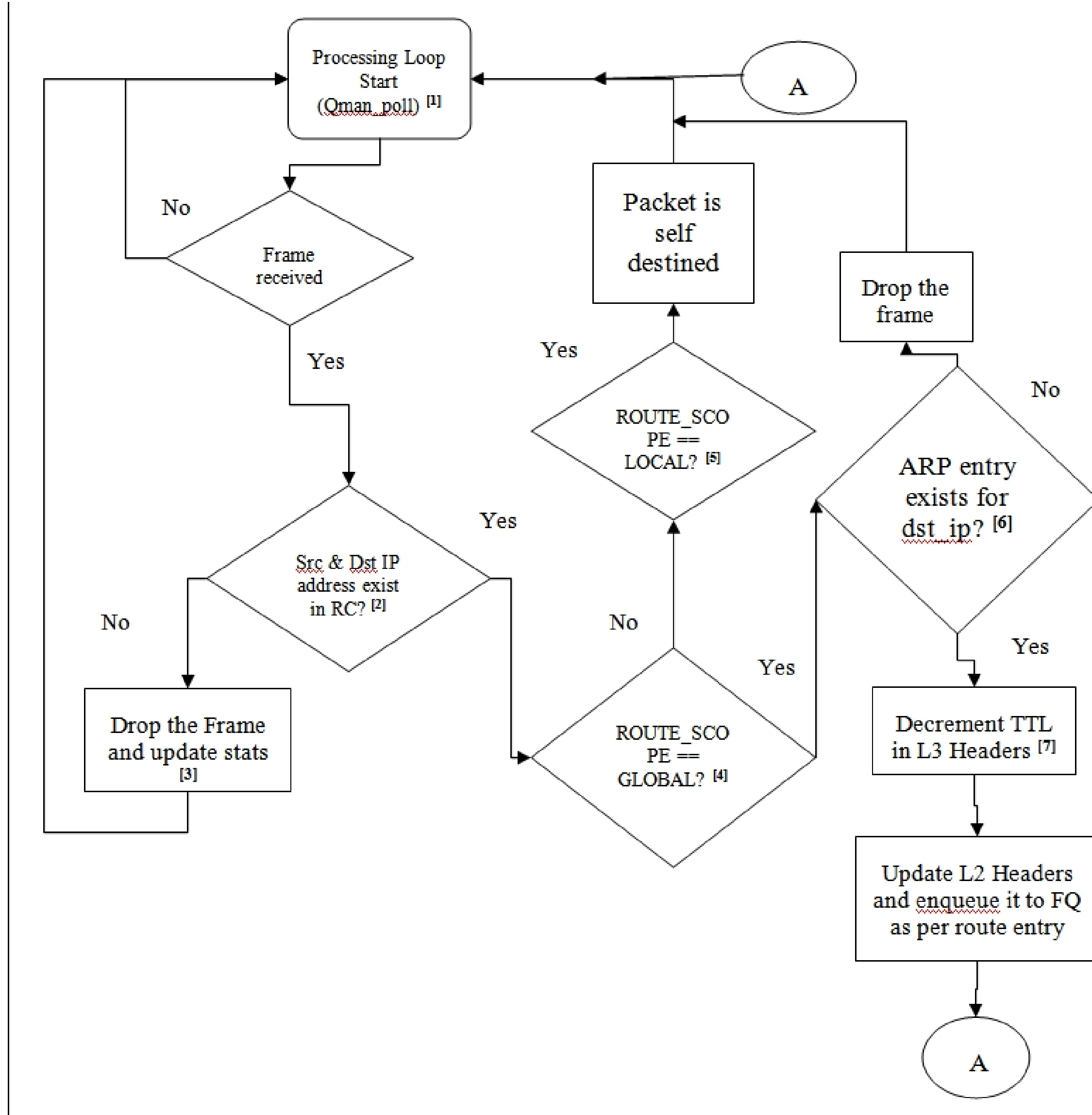
# Now, traffic can pass between Computer X and Computer Y. For example, on Computer X

# enter:

ping 192.168.28.2

## 39.10 Flowchart for packet processing

This topic provides a flowchart for packet processing.




### 39.10.1 Description of Flow chart

1. All of the threads enter the processing loop where they call Qman_poll till a frame is received.
2. If a frame is received, the application checks whether the source and destination IP address exist in the Route cache. For this it calls rc_entry_fast_lookup(), which does the fast route look up by using the hash value provided by the FMan. The hash value is indexed into the route table. The IPFWD application does not change its route cache in response to seeing the first packet in a flow. Instead the route cache is set only by commands, as mentioned in section IPv4 forward application Configuration command on page 666.
3. If the route entry for this frame is not present in the Route cache, the frame is dropped and statistics are updated. It then continues with Qman_poll.

4. If the route entry for this frame exists in the Route cache, it checks for the scope of the frame. If the scope is GLOBAL, the frame is sent for forwarding.
5. If the scope is not GLOBAL but LOCAL, the frame is self-destined. It then continues with Qman_poll.
6. If the frame is to be forwarded, check if ARP entry exists in ARP table for destination IP address. IPFWD application will not dynamically resolve the ARP. So if sending packets to IPFWD using a regular computer, the user will have to create static ARP entries.
7. If ARP entry exists, TTL is decremented in L3 header.
8. Finally, the L2 header is updated, which includes changing the dst MAC address in L2 header. The frame is then enqueued to the TX FQ.

## 39.10.2 Running IPv4 forwarding on P4080DS board

The instructions below describe how to run IPFWD. Traffic should only be directed to the P4080DS once the application is running and configuration via the ipfwd_config application is completed.

- On linux prompt, assign IP address to fm1-gb1

$ ifconfig fm1-gb1 <IPADD> up

- Configure FMan PCD using fmc with the XML files in /usr/etc:

$ cd /usr/etc

*To setup the FMan to distribute traffic to 32 ingress frame queues per port:*

$ fmc -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_32_fq.xml -a

*To setup the FMan to distribute traffic to 1024 ingress frame queues per port:*

$ fmc -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_1024_fq.xml -a

- Run IPFWD application

*The main IPFwd application binary is called* ipfwd_app *. The application can run on multiple cores as specified by the first parameter to the application. To run it to handle traffic distributed over 32 ingress frame queues per port:*

$ ipfwd_app <m..n>

By default ipfwd_app uses us_config_serdes_0xe.xml and us_policy_hash_ipv4_src_dst_32_fq.xml files.

*To run for scenario 1024 ingress frame queues per port*

$ ipfwd_app <m..n> -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_1024_fq.xml

**IPFWD application command syntax:**

[root@p4080 etc]# ipfwd_app --usage

Usage: ipfwd_app [-n?V] [-c FILE] [-d SIZE] [-i FILE] [-p FILE] [--fm-config=FILE]

[--non-interactive] [--fm-pcd=FILE] [--cpu-range] [--help]

[--usage] [--version] [cpu-range]

IPFWD application run command:

[root@p4080 root]# cd /usr/etc

<_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_32_fq.xml -a

If the user wants to use all interfaces :

[root@p4080 etc]# ipfwd_app 1..7

If the user wants to use ONLY selective interfaces :

[root@p4080 etc]# ipfwd_app 1..7 -i fm1-10g,fm2-gb2

[1] 5363

[root@p4080 etc]# Found /fsl,dpaa/ethernet@9

Found /fsl,dpaa/ethernet@8

Found /fsl,dpaa/ethernet@7

Found /fsl,dpaa/ethernet@4

Qman: FQID allocator includes range 512:128

Bman: BPID allocator includes range 56:8

Configuring for 4 network interfaces and 4 pool channels

FSL dma_mem device mapped (phys=0xf8000000,virt=0x70000000,sz=0x4000000)

Thread uid:0 alive (on cpu 1)

Release 16384 bufs to BPID 7

Release 4096 bufs to BPID 8

Release 4096 bufs to BPID 9

ipfwd_app starting

Message queue to send: /mq_snd_2536

Message queue to receive: /mq_rcv_2536

Thread uid:1 alive (on cpu 2)

Thread uid:2 alive (on cpu 3)

Thread uid:3 alive (on cpu 4)

Thread uid:4 alive (on cpu 5)

Thread uid:5 alive (on cpu 6)

Thread uid:6 alive (on cpu 7)

If, in the run application command, cpu-range is given i.e. "ipfwd_app *<m..n>*" IPFWD application starts threads on cpu-range m..n. The main thread (by default on CPU 1) then does global initialization needed by the application, including starting other application threads.

If, on the other hand, run application command is given without any cpu-range i.e. "ipfwd_app" IPFWD application starts up with a single thread running on CPU 1 by default, which does global initialization needed by the application and enables all the network interfaces.

The CLI (Command-Line Interface) allows you to add and remove additional threads to enable the use of multiple CPUs, with the only restriction being that the primary thread on CPU 1 cannot be removed (except by shutting down the application).

To add a thread on a single CPU (e.g. CPU 2):

> add 2

To add threads on a range of CPUs:

> add 3..6

To list the threads (this also queries each thread, verifying that they aren't blocked):

> list

Thread alive on cpu 1

Thread alive on cpu 2

Thread alive on cpu 3

Thread alive on cpu 4

Thread alive on cpu 5

Thread alive on cpu 6

To enable all interfaces

> macs on

To disable all interfaces

> macs off

To perform a controlled shutdown of ipfwd (this includes disabling the network ports):

> quit

- Once the application starts, it can receive the configuration commands. Run application configuration script.

For creating route entries, the binary *ipfwd_config* is run. This binary processes the configuration request from the user (using the command line) and populates the configuration via Linux standard posix IPC messages to the IPFwd application.

The shell script mentioned below contains sample commands to add route entries. Detailed description of all ipfwd_config commands is provided in section IPv4 forward application Configuration command on page 666.

SSH to p4080 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to fm1-gb1 in the beginning)

*Run the shell script:*

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipfwd_20G.sh "pid"

$ ipfwd_20G.sh 2536

There are two example shell scripts available and those are "ipfwd_20G.sh" and "ipfwd_22G.sh". ipfwd_20G.sh creates routes for only the 2 x 10G interfaces and ipfwd_22G.sh creates routes for 2 x 10G and 2 x 1G interfaces. They can assign IP addresses to the interfaces, add an ARP entry and assumes the netmask to be 255.255.255.0. The following table summarizes the settings done by these scripts. Check sectionPossible configuration scenario for IPFWD on page 651 for more details.

For the IPFwd application to forward traffic successfully, traffic destined for the P4080DS ports must have the appropriate source and destination addresses.

Console messages are printed for each entry added to the routing table. Once all configuration is completed, the application moves to the packet processing phase - the message "Application Started successfully" is printed on the console. At this point, traffic can be sent to the IPFWD application, which would do its processing on the cpu-range specified by the user on the application command-line.

### 39.10.3 Running IPv4 forwarding on P3041/P5020 board

The instructions below describe how to run IPFWD on P3041/P5020. Traffic should only be directed to P3041/P5020 once the application is running and configuration via the ipfwd_config application is completed. · On linux prompt, assign IP address to eth0

$ ifconfig eth0 <IPADD> up

Running IPFWD

· Configure FMan PCD using fmc with the XML files in /usr/etc:

$ cd /usr/etc

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

$ fmc -c usdpaa_config_p3_p5_serdes_0x36.xml -p usdpaa_policy_hash_ipv4.xml -a

· Run IPFWD application

If the user wants to use all interfaces :

$ ipfwd_app <m..n> -c usdpaa_config_p3_p5_serdes_0x36.xml -p usdpaa_policy_hash_ipv4.xml

If the user wants to use ONLY selective interfaces :

$ipfwd_app m..n -c usdpaa_config_p3_p5_serdes_0x36.xml -p usdpaa_policy_hash_ipv4.xml -i fm1-10g,fm1-gb4

For P3041, m..n can be 0..3. For P5020, m..n can be 0..1.

SSH to p4080 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipfwd_15G.sh "pid"

$ ipfwd_15G.sh 2536

There is an example shell script available named as ipfwd_15G.sh creates routes for only the 5 x 1G and 1x10G interfaces. It can assign ip addresses to the interfaces, add an ARP entry and assumes the netmask to be 255.255.255.0. Check section Possible configuration scenario for IPFWD on page 651 for more details. Now traffic can be run as per the routes created.

### 39.10.4 Running IPv4 forwarding on T4240 board

The instructions below describe how to run IPFWD on T4240. Traffic should only be directed to T4240 once the application is running and configuration via the ipfwd_config application is completed. · On linux prompt, assign IP address to eth0

$ ifconfig eth0 <IPADD> up

Running IPFWD

· Configure FMan PCD using fmc with the XML files in /usr/etc:

$ cd /usr/etc

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

$ fmc -c usdpaa_config_t4_serdes_1_1_6_6.xml -p usdpaa_policy_hash_ipv4.xml -a

· Run IPFWD application

$ipfwd_app m..n -c usdpaa_config_t4_serdes_1_1_6_6.xml -p usdpaa_policy_hash_ipv4.xml -d 0x10000000

For T4240, m..n can be 0..23.

SSH to t4240 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipfwd_20G.sh "pid"

$ ipfwd_20G.sh 2536

There is an example shell script available named as ipfwd_20G.sh creates routes for only the 2x10G interfaces. It can assign ip addresses to the interfaces, add an ARP entry and assumes the netmask to be 255.255.255.0. Now traffic can be run as per the routes created.

## 39.10.5 Running IPv4 forwarding on B4860 board

The instructions below describe how to run IPFWD on B4860. Traffic should only be directed to B4860 once the application is running and configuration via the ipfwd_config application is completed. · On linux prompt, assign IP address to eth0

$ ifconfig eth0 <IPADD> up

Running IPFWD

· Configure FMan PCD using fmc with the XML files in /usr/etc:

$ cd /usr/etc

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

$ fmc -c usdpaa_config_b4_serdes_0x2a_0x98.xml -p usdpaa_policy_hash_ipv4.xml -a

· Run IPFWD application

$ipfwd_app m..n -c usdpaa_config_b4_serdes_0x2a_0x98.xml -p usdpaa_policy_hash_ipv4.xml

For B4860, m..n can be 0..7.

SSH to b4860 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipfwd_20G.sh "pid"

$ ipfwd_20G.sh 2536

There is an example shell script available named as ipfwd_20G.sh creates routes for only the 2x10G interfaces. It can assign ip addresses to the interfaces, add an ARP entry and assumes the netmask to be 255.255.255.0. Now traffic can be run as per the routes created.

### 39.10.6 USDPAA IP Fwd performance gap between 6 core and 8 core

USDPAA IPfwd performance for 8 core is less than 6 core on e6500 series. USDPAA IP Fwd application need to run using "-s" option to bridge the gap between two configuration.

### 39.10.7 PPAC (and IPFwd) CLI commands

The following commands are illustrated in the context of IPFwd, but the commands are common to all PPAC-based applications.

To add a thread on a single CPU (e.g. CPU 2):

> add 2

To add threads on a range of CPUs:

> add 3..6

To list the threads (this also queries each thread, verifying that they aren’t blocked):

> list

Thread uid:0 alive (on cpu 1)

Thread uid:1 alive (on cpu 2)

Thread uid:2 alive (on cpu 3)

Thread uid:3 alive (on cpu 4)

Thread uid:4 alive (on cpu 5)

Thread uid:5 alive (on cpu 6)

Thread uid:6 alive (on cpu 7)

To remove a thread by its UID:

> rm uid:2

Thread uid:2 killed (cpu 3)

To remove a thread running on a given CPU:

> rm 5 Thread uid:4 killed (cpu 5)

To enable all interfaces:

> macs on

To disable all interfaces:

> macs off

To perform a controlled shutdown of ipfwd (this includes disabling the network ports):

> quit

To query cgr

> cgr

Rx CGR ID: 10, selected fields;

cscn_en: 0

cscn_targ: 0x00800000

cstd_en: 1

cs: 0

cs_thresh: 0x00_0000_1000

mode: 1

i_bcnt: 0x00_0000_0e1e

a_bcnt: 0x00_0000_0e1e

Tx CGR ID: 11, selected fields;

cscn_en: 0

cscn_targ: 0x00800000

cstd_en: 1

cs: 0

cs_thresh: 0x00_0000_0200

mode: 1

i_bcnt: 0x00_0000_0002

a_bcnt: 0x00_0000_0004

# 39.11 IPv4 forward application Configuration command

## 39.11.1 Syntax

The syntax is as follows:

$ [root@p4080 bin]# ipfwd_config --help

Usage: ipfwd_config [OPTION...]

-B, --routeadd=TYPE adding a route

-C, --routedel=TYPE deleting a route

-E, --showintf=TYPE show interfaces

-F, --intfconf=TYPE change intf config

-G, --arpadd=TYPE adding a arp entry

-H, --arpdel=TYPE deleting a arp entry

-O, --Start/ Go=TYPE Start the processing of packets

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

### 39.11.1.1 Command to show all enabled interfaces and their interface numbers

The command to show all the enabled interfaces while running IPv4 forward is as follows:

Ipfwd_config -P pid -E -a true

Table 67: Field Description (show all enabled interfaces)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -a | Show all the interfaces | Yes | true |

Command to show all enabled interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# Ipfwd_config -P 2536 -E -a true

Interface number: 11

PortID=1:5 is FMan interface node

with MAC Address

02:00:c0:a8:65:fe

Interface number: 9

PortID=1:3 is FMan interface node

with MAC Address

02:00:c0:a8:5b:fe

Interface number: 8

PortID=1:2 is FMan interface node

with MAC Address

02:00:c0:a8:51:fe

Interface number: 5

PortID=0:5 is FMan interface node

with MAC Address

02:00:c0:a8:33:fe

Are all the Enabled Interfaces [root@p4080 bin]#

### 39.11.1.2 Help for show all enabled interfaces command

The command to obtain help for show all enabled interfaces command is as follows:

ipfwd_config -E -help

Help for show all enabled interfaces

[root@p4080 etc]# ipfwd_config -E --help

Usage: -E [OPTION...]

-a, --a=ALL All interfaces

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

#### 39.11.1.3 Assign IP address to interfaces

The command to assign IP address while running IPv4 forward is as follows:

ipfwd_config -P pid -F -a 192.168.60.1 -i <Interface number>

NOTE

Note: The interface number to be used here must be one of the numbers that got displayed as the output of "show all enabled interfaces command" in section Command to show all enabled interfaces and their interface numbers on page 666.

Table 68: Field description (assign IP address to interfaces)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -a | IP Address | Yes | a.b.c.d |
| -i | Interface number | Yes | 0-11<br>(Choose this number from "show all enabled interfaces" command output) |

Assign IP address to interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

ipfwd_config -P 2536 -F -a 192.168.60.1 -i 5

IPADDR assigned = 0xc0a83c01 to interface num 5

Intf Configuration Changed successfully

#### 39.11.1.4 Help for assign IP address to interfaces

The command to obtain help for assign IP address to interfaces command is as follows:

ipfwd_config -F --help

Help for assign IP address to interfaces

[root@p4080 etc]# ipfwd_config -F --help

Usage: -F [OPTION...]

-a, --a=IPADDR IP Address

-i, --i=IFNAME If Name

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

#### 39.11.1.5 Adding a Route Entry

The command to add a route while running IPv4 forward is as follows:

ipfwd_config -P pid -B -s a.b.c.d -d b.c.d.e -g c.d.e.f

Table 69: Field Description (Adding a Route Entry)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -s | Source IP Address | Yes | a.b.c.d |
| -d | Destination IP Address | Yes | a.b.c.d |
| -g | Gateway IP Address | Yes | a.b.c.d |

Adding a Route Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# ipfwd_config -P 2536 -B -s 192.168.29.2 -d 192.168.24.2 -g 192.168.24.2

Route Entry Added successfully

[root@p4080 bin]#

#### 39.11.1.6 Help for Route Entry Addition

The command to obtain help for route entry addition is as follows:

ipfwd_config -B --help

Help for Adding a Route Entry

[root@p4080 bin]# ipfwd_config -B --help

Usage: -B [OPTION]

-d, --d=DESTIP Destination IP

-f, --f=FLOWID Flow ID - (0 - 1024) {Default: 0}

-g, --g=GWIP Gateway IP

-s, --s=SRCIP Source IP

-t, --t=TOS Type of Service - (0 - 256) {Default: 0}

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

#### 39.11.1.7 Deleting a Route Entry

The command to delete a route while running IPv4 forward is as follows:

ipfwd_config -P pid -C -s a.b.c.d -d b.c.d.e

Table 70: Field Description (Deleting a Route Entry)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -s | Source IP Address | Yes | a.b.c.d |
| -d | Destination IP Address | Yes | a.b.c.d |

Deleting a Route Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# ipfwd_config -p 2536 -C -s 192.168.29.2 -d 192.168.24.2 Route Entry Deleted successfully

#### 39.11.1.8 Help for Deleting a Route Entry

The command to obtain help for route entry deletion is as follows:

ipfwd_config -C --help

Help for Deleting a Route Entry

[root@p4080 bin]# ipfwd_config -C --help

Usage: -C [OPTION...]

-d, --d=DESTIP Destination IP

-s, --s=SRCIP Source IP

-t, --t=TOS Type of Service - (0 - 256) {Default: 0}

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

#### 39.11.1.9 Starting the Application Processing

The command to start the application processing phase is as follows:

ipfwd_config -O

Starting the Application Processing

[root@p4080 bin]# ipfwd_config -O

Application Started successfully

### 39.11.1.10 Adding an ARP Entry

The command to add an ARP entry while running IPv4 forward is as follows:

ipfwd_config -P pid -G -s a.b.c.d -m aa:bb:cc:dd:ee [-r true]

**Table 71: Field Description (Adding an ARP Entry)**

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -s | Gateway IP Address | Yes | a.b.c.d |
| -m | Mac Address | Yes | aa:bb:cc:dd:ee |
| -r | Replace existing entry | No | true/ false {Default: false} |

Adding an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# ipfwd_config -P 2536 -G -s 192.168.24.2 -m 02:00:c0:a8:33:fd -r true

ARP Entry Added successfully

### 39.11.1.11 Help for ARP Entry Addition

The command to obtain help for ARP entry addition is as follows:

ipfwd_config -G --help

Help for Adding an ARP Entry

[root@p4080 etc]# ipfwd_config -G --help

Usage: -G [OPTION...]

-m, --m=MACADDR MAC Address

-r, --r=Replace Replace Exiting Entry - true/ false {Default:

false}

-s, --s=IPADDR IP Address

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

### 39.11.1.12 Deleting an ARP Entry

The command to delete an ARP while running the IPv4 forward is as follows:

ipfwd_config -P pid -H -s a.b.c.d

Table 72: Field Description (Deleting an ARP Entry)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -s | Gateway IP Address | Yes | a.b.c.d |

Deleting an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# ipfwd_config -P 2536 -H -s 192.168.24.2

Arp Entry Deleted successfully

[root@p4080 bin]#

#### 39.11.1.13 Help for Deleting an ARP Entry

The command to obtain help for ARP entry deletion is as follows:

ipfwd_config -H --help

Help for Deleting an ARP Entry

[root@p4080 etc]# ipfwd_config -H --help

Usage: -H [OPTION...]

-s, --s=IPADDR IP Address

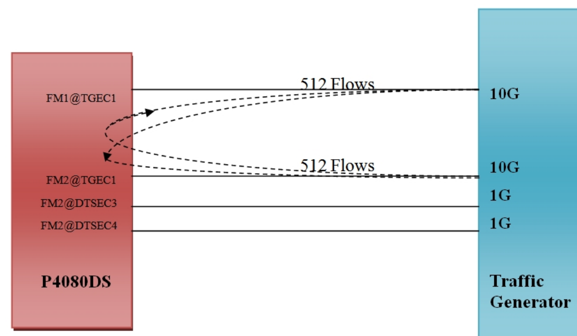
-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

## 39.12 Traffic Generation



## 39.13 References

1. USDPAA PPAC User Guide
2. QMan/BMan API Guide

# Chapter 40 Freescale USDPAA IPSecfwd User Manual

## 40.1 Introduction

The User Space Data Path Acceleration Architecture (USDPAA) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document provides the following:

- A summary of the USDPAA IPSecfwd application.
- Execution steps for the IPSecfwd application.

### 40.1.1 Purpose

This document describes the USDPAA IPSec forwarding application. This application documents the USDPAA IPSecfwd demonstration applications forwarding flow.

### 40.1.2 Change History

Table 73: Change History

| Version | Updates |
| --- | --- |
| 1.1 | Creation of IPSecfwd UG for phase v1.0 release.<br>- PPAC/PPAM overview<br>- Basic application flow<br>- Commands to use<br>- Testing IPSecfwd application |
| 1.2 | Addition of t4/b4 sections |

## 40.2 USDPAA IPSecfwd application

The USDPAA IPSec forwarding (IPSecfwd) application is a multi-threaded application that routes IPv4 packets from one Ethernet interface to another after performing encryption/decryption if required on P4080/P3041/P5020/T4240/B4860 systems. The configuration done for the system and the type of the packet is used to determine the type of operation to be performed on the packet. The routing is done based on the source IP address and destination IP address in the frame.

Any combination of the 8 cores on the P4080 can run a USDPAA IPSec Forwarding application thread in USDPAA SDK v1.0.

## 40.2.1 Application Overview

The IPSecfwd application can route IPv4 traffic directly over interfaces as well as over the IPSec tunnel between two network nodes connected over different subnets with the assistance of the IPSec protocol security provided by SEC4.0 block. The application works as an extension of the IPv4 forwarding application with encryption/ decryption in addition to route lookup and packet forwarding. The IPSec processing tunnel table contains information on frame queues toward SEC4.0 for the IPSec protocol processing of the packet before it is sent out on the egress interface. IPsec can use an extended sequence number (ESN) optionally that is authenticated but not transmitted. If an ESN is found in the PDB, it is given to the authentication CHA in the last. The ESN is incremented whenever the Seq Num rolls over.

The IPSecfwd application has two main phases. There is an initial configuration phase and a subsequent packet processing phase.

The configuration phase executes when the application starts. Application threads are created and global initialization of resources is done which is the part of PPAC (see USDPAA PPAC User Guide for more details). The configuration phase also includes PPAM (i.e. IPSecfwd) related initialization. Once the configuration phase is completed the IPSecfwd application moves to the packet processing phase.

The application provides a command-line interface to enable users to add and remove routing table, ARP cache entries and SA entries at any given time. For each user input, the appropriate information is communicated to the IPSecfwd application via standard Posix IPC. Messages are placed onto the message queue till they are received by the IPSecfwd application. Note that the IPSecfwd application does not dynamically resolve ARP - missing ARP entries will result in the application dropping the packet.

In the packet processing phase, the loop migrates from polling mode to a blocking ìIRQ modeî whenever IPSecfwd has looped a certain number of times without any forward progress. For more information on IRQ mode please refer to ìUSDPAA PPAC User Guideî. In polling mode - the application constantly looks for data to process on its dedicated QMan portal. Network traffic is classified and distributed by the FMan to frame queues based on source and destination IP address in the packet. There is an associated handler that processes the packets arriving on each frame queue.

## 40.2.2 Packet Flow

The IPSecfwd application is capable of routing packets with the security processing support from the P4080's SEC4.0 engine. After all the initialization and configurations, each core enters the polling loop and polls for packets from FMan(PPAC), or SEC4.0 (PPAM). The loop dequeues packets from the frame queue associated to pool channels shared by the application cores.

Once a frame is dequeued, it is sent for processing based on callback handler in contextB of the frame queue response ring entry. When the frame is ingressed from FMAN, then the callback handler is called which performs the route lookup and tunnel lookup; and then sends the packet to SEC4.0 block/FMan. For the frame returning from SEC4.0, the encap callback handler is called for a frame coming after encryption or decap callback handler is called for a frame coming after decryption.

The steps for the packet flow for ingress from FMan are as follows:

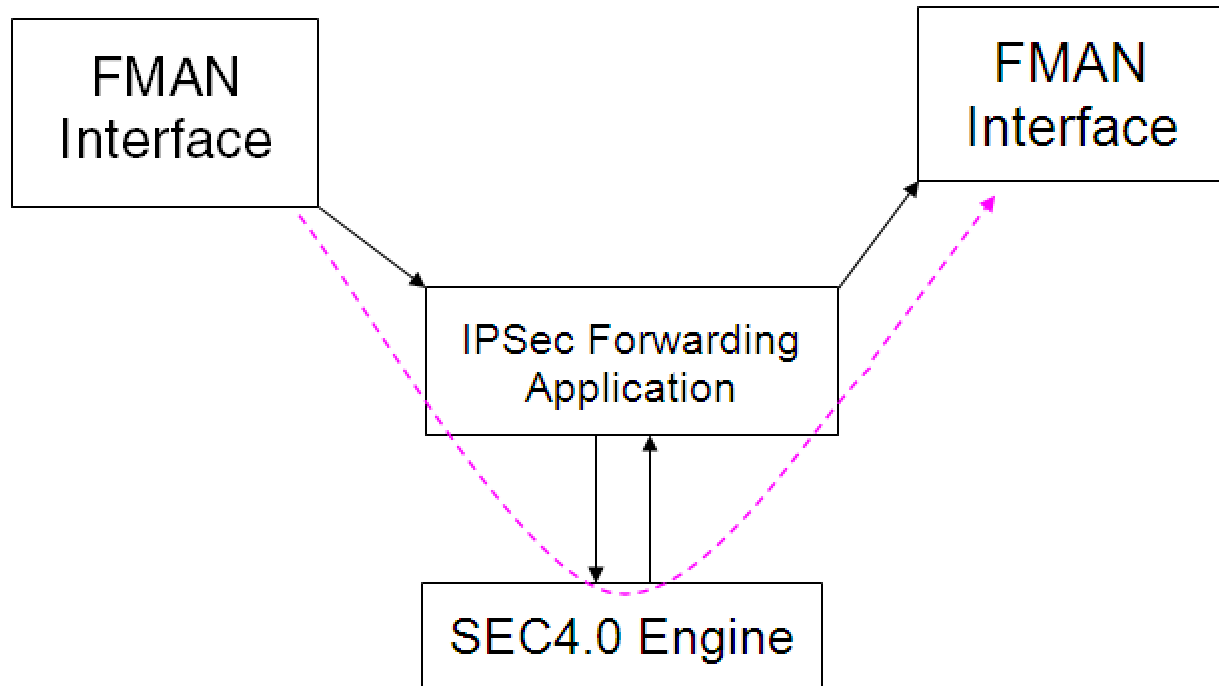
1. Once the packet is dequeued from FMan, DQRR callback handler associated with Fman is called.
2. After a basic sanity check of the packet (packet header is correct, packet checksum is correct etc.), the packet is checked for the ESP protocol.
    a. If the packet is ESP, a direct tunnel lookup happens based on FMan's PCD hash value and the frame is sent to SEC4.0 engine
    b. If the packet is non-ESP, a route lookup based on FMan's PCD happens. If there is an SA entry, it is sent to SEC4.0 block after updating annotations in frame with next level route destination.
    c. If the packet is non-ESP and the route is normal, the packet is sent to FMan interface.

   d. If no route is found, the packet is discarded.

3. When the packet is dequeued from SEC4.0 Engine, the DQRR callback handler associated with SEC4.0 frame queues is called. Encap callback handler or Decap callback handler will be called from based on the callback handler in contextB of DQRR ring entry.
4. The packet is now taken directly to the FMan interface as the destination field was already filled while sending the packet to the SEC4.0 Engine.

A typical packet flow in a simulated P4080 environment is shown below.

Figure 61: IPSecfwd Application basic packet flow




## 40.2.3 Overview of IPSecfwd packet processing

IPSecfwd Steady state processing or Forwarding consists of two parts

- Outbound processing: when un-encrypted packets are received by the system and tunneled using IPSecfwd application.
- Inbound processing: when encrypted packets are received by the systems and are decrypted by the system.

### 40.2.3.1 Outbound processing:

1. *Pre-processing:*
   a. Packet is received by the FM which uses 2-tuple (Src IP, Dest IP) to hash the packets to different Core Rx Queues.
   b. This packet is picked up by the core and first subjected to IPv4 Forwarding lookup.
   c. If the action is to do IPSec processing, then it looks up for a corresponding entry in the SADB, which has the SA information and Queue-ids for the SA used by SEC4.0.
2. *Crypto-processing:*
   a. SEC4.0 dequeues the job from egress Queue toward SEC4.0.

b. The SEC4.0 encrypts, authenticates and adds the ESP header and outer IP header to the packet. It then enqueues the processed packet to the Ingress Queue from SEC4.0.

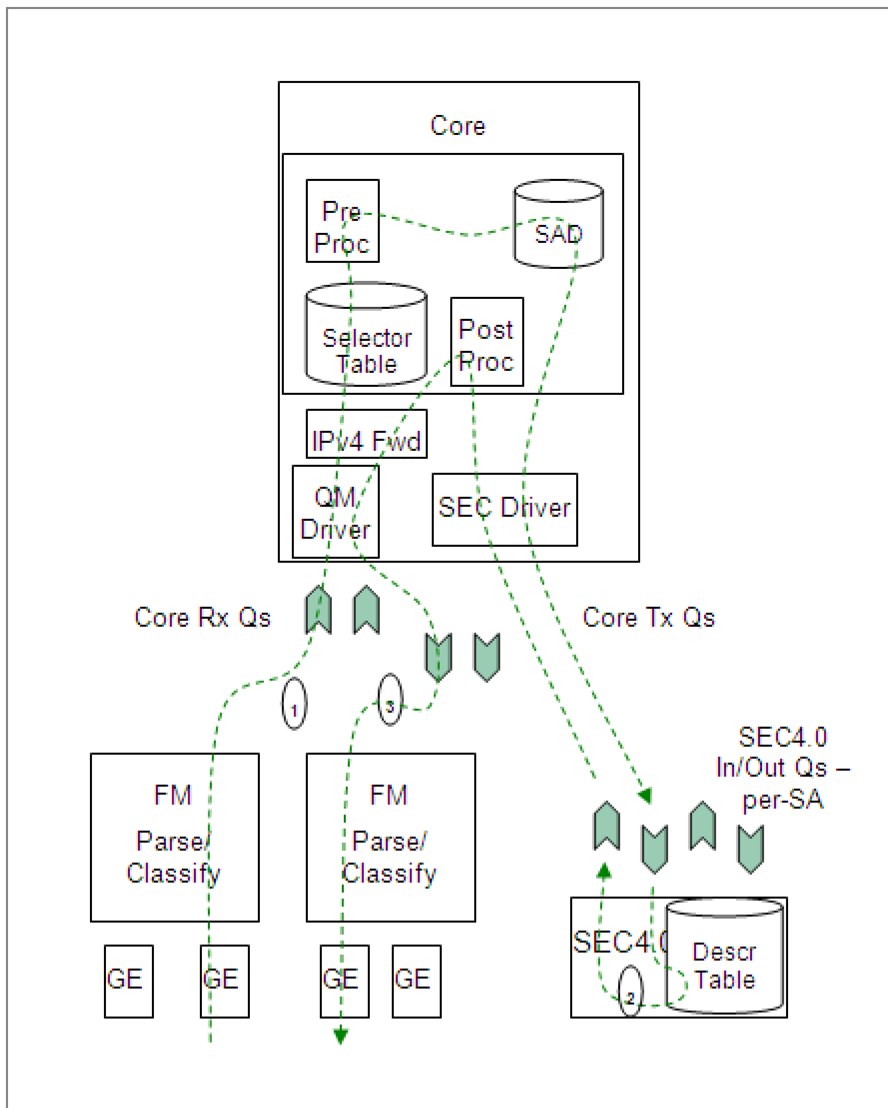
3. *Post-processing*:

a. The core now dequeues the processed packet from the SEC4.0 on Out FQ and sends it to the IPv4 Forwarding module (since there is a new Outer-IP header).

b. IPv4 Forwarding module does a lookup and transmits the packet from the egress interface.

c. If the SEC4.0 result indicates an error in the Crypto processing of the packet, it is discarded and statistics (packet, octets, and errors - per SA / Interface) are updated.

Figure 62: IPSecfwd Outbound processing




## 40.2.3.2 Inbound Processing:

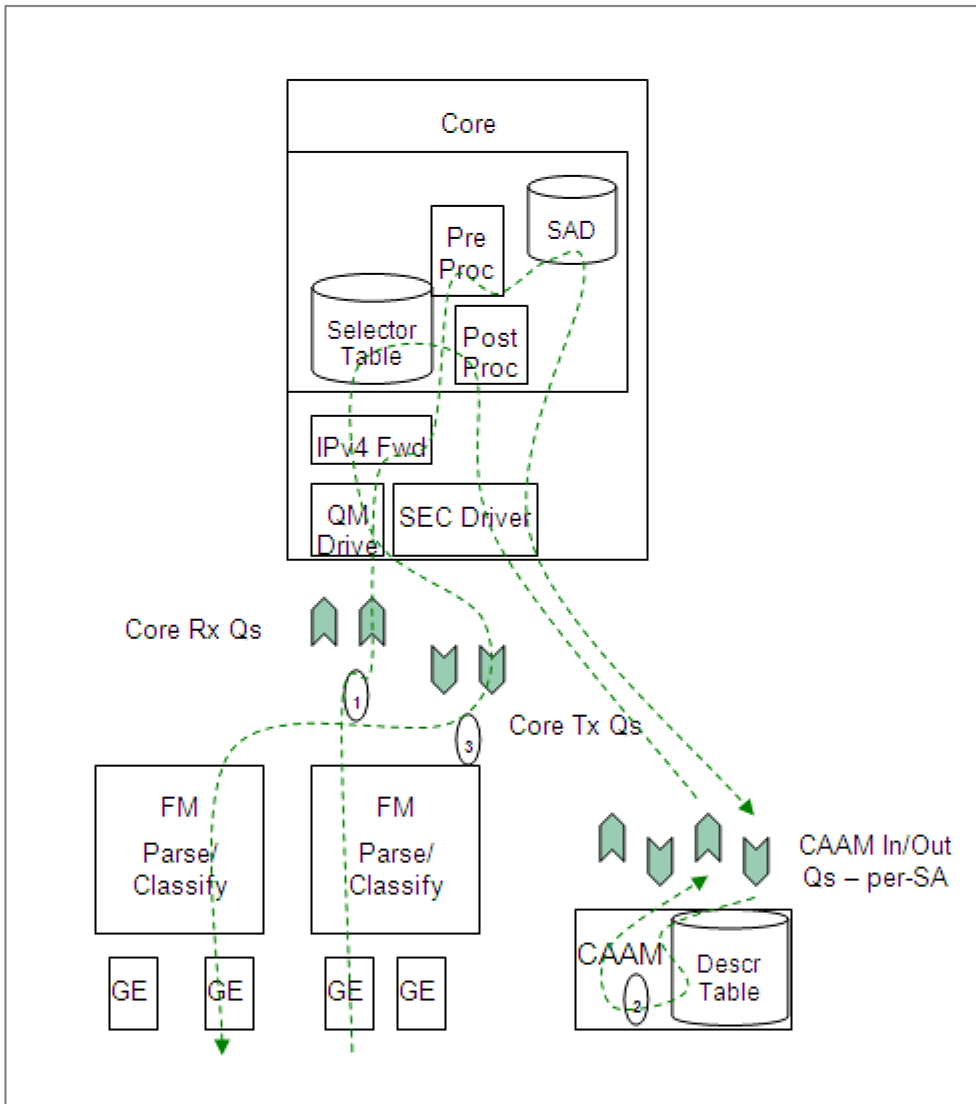
1. Pre-processing:

a. The IPSec tunneled packet is received by the FM, which uses a 3-tuple (Dest-IP, Src-IP, SPI) to hash the packet to different Core Rx Queues.

b. The packet is picked up and undergoes IPv4 Forwarding lookup. If packet is self-destined, and protocol is ESP, it is sent to IPSec Forwarding module for processing.
c. The SA info in SADB yields a SEC4.0-In Queue for the SA. The packet is sent to that SEC4.0 on its ingress FQ.

2. Crypto-processing:
   a. The SEC4.0 decrypts, authenticates and enqueues the processed packet to the SEC4.0-Out Queue found in its input FQs contextB.
3. Post-processing:
   a. The core de-queues the processed packet from the SEC4.0-Out Queue, and subjects the packet to a Selector-table lookup.
   b. The packet is handed over to IPv4 Forwarding module, which does a route cache lookup and transmits the packet from the egress interface.
   c. If the SEC4.0 result indicates an error in the Crypto processing of the packet, it is discarded and statistics are updated.

Figure 63: IPSecfwd Inbound processing



### 40.2.4 Flow chart for IpSecfwd packet processing

Figure 64: Flow chart for IpSecfwd packet processing

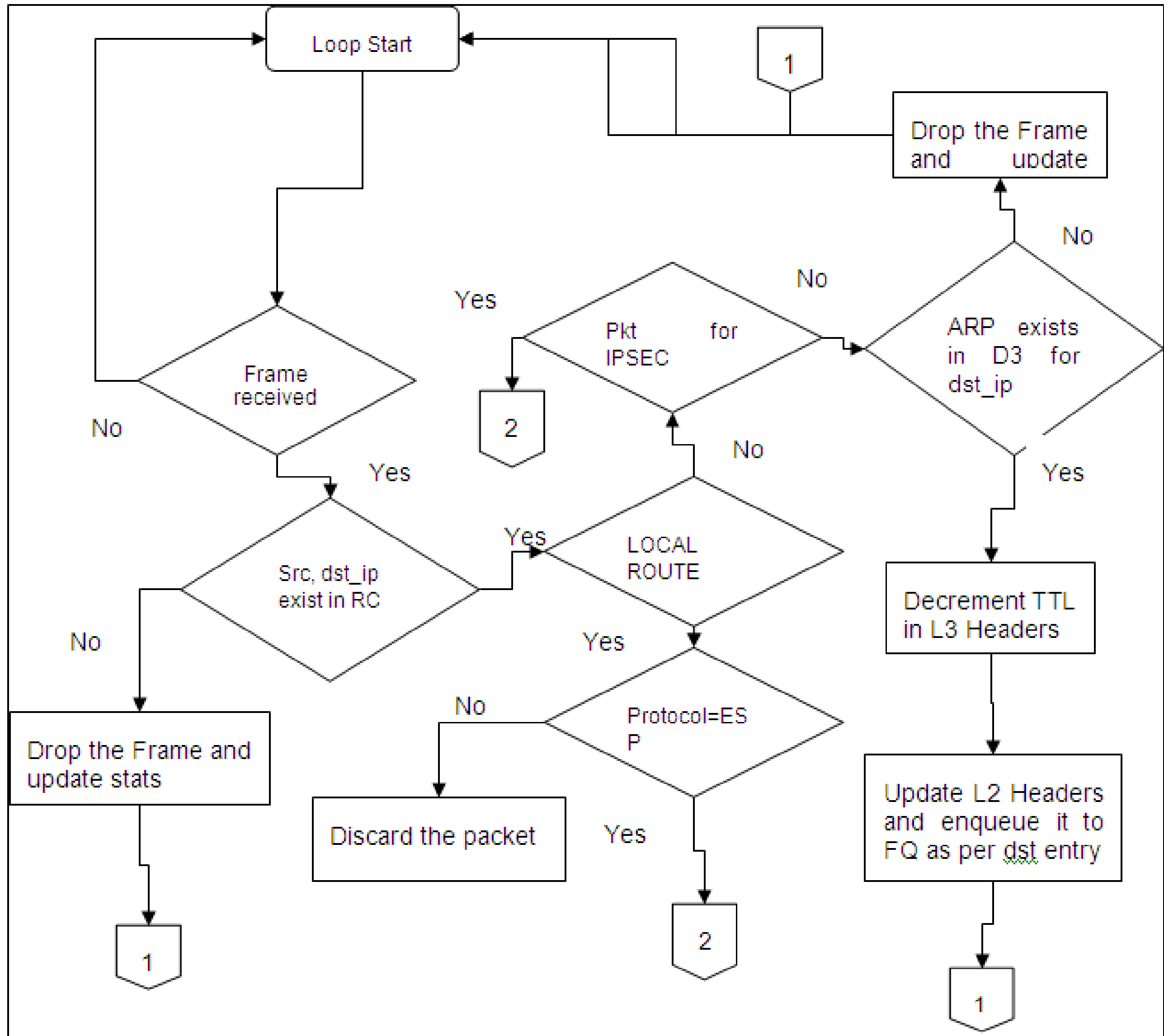
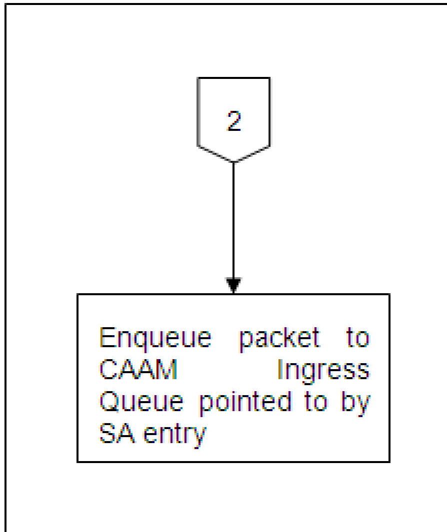


Figure 65: IPSecfwd packet processing




## 40.3 Overview of PPAC

The source code to IPSecfwd has been reorganized into two parts; the "PPAC" (Packet-Processing Application Core) and a "PPAM" (Packet-Processing Application Module). The PPAM portion implements the IPSecfwd application specific logic of processing the packet and forwarding it. On the other hand, the PPAC component represents the common infrastructure to support PPAM; initializing devices, handling flow-control, implementing a CLI (Command-Line Interface), managing threads and buffers. PPAC details can be found in the document "USDPAA PPAC User Guide".

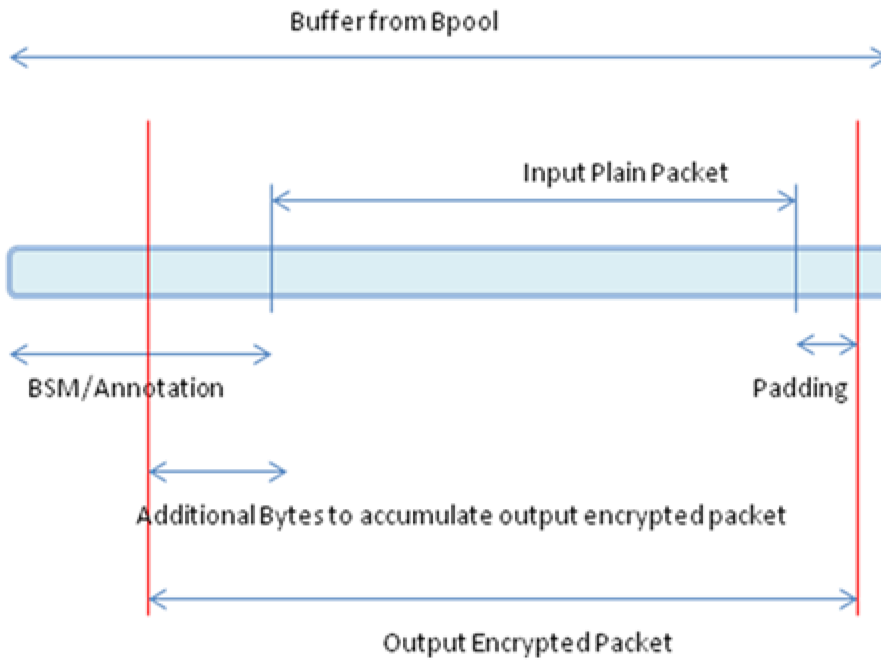
## 40.4 IPSecfwd related PPAM Details

### 40.4.1 In-Place Encryption/Decryption

IPSecfwd application uses in-place Encryption/Decryption when packet is sent to SEC4.0 block. In-place Encryption/Decryption is supported in IPSecfwd application by using the same output buffer as the input buffer.

In case of Encryption the size of the output packet is more than the size of input packet due to addition of tunnel header, padding of extra bytes etc. The FMam is configured to acquire the buffer from BMan which is large enough to accommodate the output packet after encryption.

Figure 66: IPSecfwd In-place Encryption




In case of Decryption the output packet size is smaller than the Input packet size. So the output plain packet in case of Decryption can easily be accommodated in the buffer acquired by FMan for storing the Input packet.

Figure 67: IPSecfwd In-place Decryption

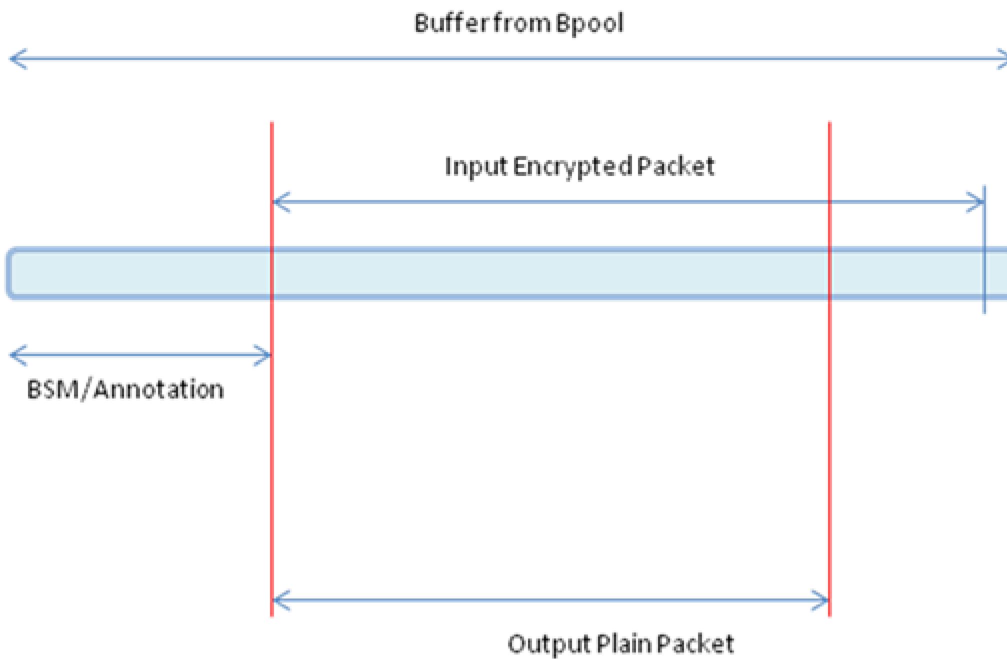
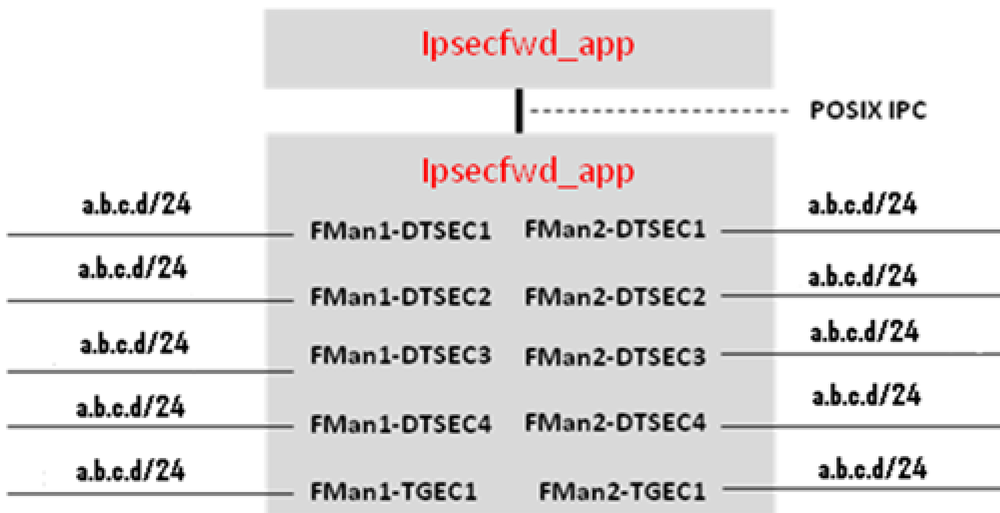



Figure 6: IPSecfwd In-place Decryption

# 40.5 Secfwd application suite

The figure below shows the structure of the IPSecfwd USDPAA application suite. Its purpose is to encrypt/decrypt and forward IPv4 packets between the interfaces shown. No IP address is assigned to any of the interfaces by default. Using ipsecfwd_config command as mentioned in section 5.3.1.3 IP address can be assigned to all these interfaces. Each interface has a fixed netmask shown in the figure. The notation "/24" refers to a netmask of 255.255.255.0. The MAC addresses of these interfaces are determined by u-boot environment variables ethaddr, eth1addr, eth2addr, etc.

Figure 68: IPSecfwd application suite




The ipsecfwd application will respond to ARP requests on these interfaces. However, the application will not generate ARP requests to determine the destination MAC address of forwarded packets. An ipsecfwd_config command should be used to set these MAC addresses.

It is important to understand that the application can use only a subset of these interfaces at any one time. This is due to pin multiplexing rules on the P4080 SoC. The set of available interfaces is determined by a number of factors:

1. The interface set made available by the selected SerDes Protocol and u-boot variable "hwconfig". See the SDK document "Freescale DPAA SDK X.Y: Selecting Ethernet Interfaces". This document is distributed with the DPAA SDK.
2. The Linux device tree determines which subset of the available interfaces is for use by USDPAA applications. See the USDPAA User Guide for more information.
3. Finally, the fmc configuration file passed by command line argument to ipsecfwd_app determines the subset of these interfaces that the application will attempt to initialize.

In the default SerDes 0xe example, the interfaces used by ipsecfwd_app are:

- FMan1-TGEC1
- FMan2-DTSEC3
- FMan2-DTSEC4
- FMan2-TGEC1

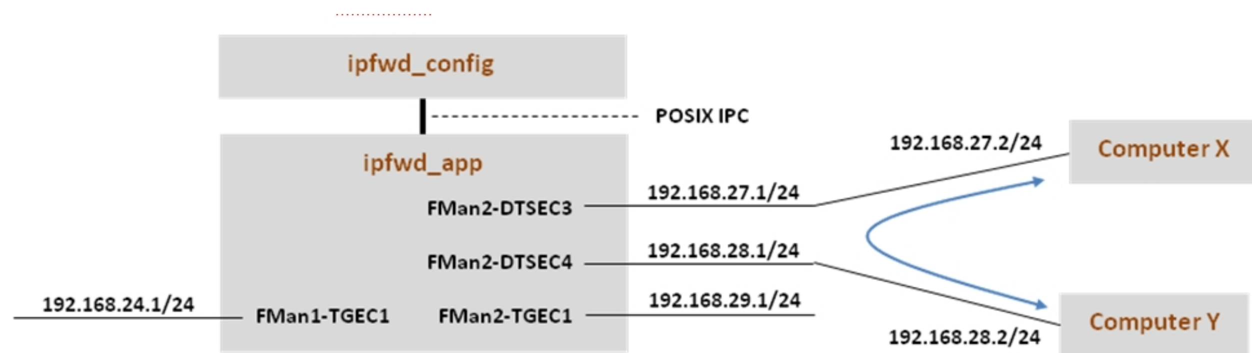
Running the ipsecfwd application suite involves four steps:

1. Run the fmc application to configure the FMan hardware instances.
2. Run ipsecfwd_app
3. Run ipsecfwd_config repeatedly to add SA Entries/routes.

Specific examples showing these steps are provided in other sections of this document.

## 40.5.1 Using Two Computers to Test the IPFWD Application Suite

This section describes how to configure the IPFWD application suite to forward packets between two ordinary computers as shown in the following figure. It is assumed that the two 1 Gbps Ethernet interfaces provided by SerDes 0xe are used.




Assign IP addresses to all the interfaces. The IP addresses used here for Computer X and Computer Y are examples. Their MAC addresses must be known as they will be needed in the commands below.

Keep in mind that ipfwd_app has no default IP address assigned to the interfaces. This means that you must first assign IP addresses to the interfaces and then use IP addresses for Computer X and Computer Y that are on the subnets shown in the figure. Please be careful with the network parameters. Any mistake will prevent packets from flowing from end to end.

Follow these steps on Computer X:

1. Set the ip address for eth0 interface

ifconfig eth0 192.168.27.2 up

Set the default gateway for subnet 192.168.27.1

route add default gw 192.168.27.1 eth0

2. Add arp for gw

arp –s 192.168.27.1 “MAC address for 192.168.27.1 on P4080”

Follow these steps on Computer Y:

1. Set the ip address for eth0 interface

ifconfig eth0 192.168.28.2 up

Set the default gateway for subnet 192.168.28.1

route add default gw 192.168.28.1 eth0

2. Add arp for gw

arp –s 192.168.28.1 “MAC address for 192.168.28.1 on P4080”

The commands to perform on P4080 are:

# Boot the P4080 and login as root.

*Assign IP address to fm1-gb1*

ifconfig fm1-gb1 <IPADD> up

cd /usr/etc

fmc -c us_config_serdes_0xe.xml -p us_policy_hash_ipv4_src_dst_32_fq.xml -a

# Assume use of cores 1 - 7 ipfwd_app 1..7

# Now ssh to P4080 linux on other terminal

ssh root@<IPADD>

give IP address as assigned to fm1-gb1 in the beginning

# Now assign ip address to the interfaces

# First run command to check what all enabled interfaces are available

Note: check pid from application print "Message queue to send: /mq_snd_2536"

ipfwd_config -P 2536 -E -a true

Interface number: 11

PortID=1:5 is FMan interface node

with MAC Address

02:00:c0:a8:65:fe

Interface number: 9

PortID=1:3 is FMan interface node

with MAC Address

02:00:c0:a8:5b:fe

Interface number: 8

PortID=1:2 is FMan interface node

with MAC Address

02:00:c0:a8:51:fe

Interface number: 5

PortID=0:5 is FMan interface node

with MAC Address

02:00:c0:a8:33:fe

Are all the Enabled Interfaces

# Assign IP address to the interfaces.Use the same interface number displayed as an ouput on giving the above command

ipfwd_config -P 2536 -F -a 192.168.24.1 -i 5

ipfwd_config -P 2536 -F -a 192.168.28.1 -i 9

ipfwd_config -P 2536 -F -a 192.168.27.1 -i 8

ipfwd_config -P 2536 -F -a 192.168.29.1 -i 11

# Now enter routes and MAC addresses. Format of a MAC address is

# aa:bb:cc:dd:ee:ff where the letters are hexadecimal digits.

ipfwd_config -P 2536 -B -s 192.168.27.2 -d 192.168.28.2 -g 192.168.28.2

ipfwd_config -P 2536 -G -s 192.168.28.2 -m COMPUTER_Y_MAC_ADDRESS -r true

ipfwd_config -P 2536 -B -s 192.168.28.2 -d 192.168.27.2 -g 192.168.27.2

ipfwd_config -P 2536 -G -s 192.168.27.2 -m COMPUTER_X_MAC_ADDRESS -r true

# Computer X and Computer Y need to be told to route via the P4080.

# On Computer X (assuming it runs Linux), enter this command as root:

route add -net 192.168.28.0 netmask 255.255.255.0 gw 192.168.27.1

# On Computer Y (assuming it runs Linux), enter this command as root:

route add -net 192.168.27.0 netmask 255.255.255.0 gw 192.168.28.1

# Now, traffic can pass between Computer X and Computer Y. For example, on Computer X

# enter:

ping 192.168.28.2

## 40.5.2 Running IPSecfwd on P4080DS board

The instructions below describe how to run IPSecfwd. Traffic should only be directed to the P4080DS once the application is running and configuration via the ipfsecwd_config application is completed.

•On linux prompt, assign IP address to fm1-gb1

```
$ ifconfig fm1-gb1 <IPADD> up
```

•Configure FMan PCD using fmc with the XML files in /usr/etc:

```
    $ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
    $ fmc -c usdpaa_config_serdes_0xe.xml -p usdpaa_policy_hash_ipv4.xml -a
```

•Run IPSecfwd application

The main IPSecfwd application binary is called **ipsecfwd_app**. The application can run on multiple cores as specified by the first parameter to the application. To run it to handle traffic distributed over 32 ingress frame queues per port:

```
$ ipsecfwd_app <m..n>
```

By default ipsecfwd_app uses usdpaa_config_serdes_0xe.xml and usdpaa_policy_hash_ipv4.xml files.

**IPSECFWD application command syntax:**

```
[root@p4080 etc]# ipsecfwd_app --usage
Usage: ipsecfwd_app [-n?V] [-c FILE] [-p FILE] [--fm-config=FILE]
```

```
            [--non-interactive] [--fm-pcd=FILE] [--cpu-range] [--help]
            [--usage] [--version] [cpu-range]
```

IPSECFWD application run command:

```
[root@p4080 root]# cd /usr/etc
<_config_serdes_0xe.xml -p usdpaa_policy_hash_ipv4.xml -a
[root@p4080 etc]# ipsecfwd_app 1..7
[1] 5363
ipsecfwd_app starting
Message queue to send: /mq_snd_2536
Message queue to receive: /mq_rcv_2536
```

If in the run application command, cpu-range is given i.e. ìipsecfwd_app <m..n>î IPSecfwd application starts threads on cpu-range m..n. The main thread (by default on CPU1), then does global initialization needed by the application, including starting other application threads.

If on the other hand run application command is given without any cpu-range i.e. ìipsecfwd_appî IPSecfwd application starts up with a single thread running on CPU 1 by default, which does global initialization needed by the application and enables all the network interfaces.

•Once the application starts it can receive the configuration commands. Run application configuration script

For creating SA entries, the binary **ipsecfwd_config** is run. This binary processes the configuration request from the user (using the command line) and populates the configuration via Linux standard posix IPC messages to the IPSecfwd application.

The shell script mentioned below contains sample commands to add SA entries.

SSH to p4080 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to fm1-gb1 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipsecfwd_20G.sh "pid"

```
    $ ipsecfwd_enc_20G.sh 2536 or
    $ ipsecfwd_dec_20G.sh 2536
```

One of the example shell scripts available is ìipsecfwd_enc_20G.shî which creates SA Entries for encryption for only the 2 x 10G interfaces. They can assign IP addresses to the interfaces, add SA and ARP entries and assumes the netmask to be 255.255.255.0. The following table summarizes the settings done by this script.

**Table 74: ipsecfwd_enc_20G.sh**

| Port ID | Source IP Address | Destination IP Address (for each src IP addr) | Source tunnel address | Destination tunnel address | Default Gateway IP Address |
|---|---|---|---|---|---|
| 4 | 192.168.60.2 to 192.168.60.24 | 192.168.160.2 .. 24 | 192.168.60.2 | 192.168.60.1 | 192.168.160.2 |
| 9 | 192.168.160.2 to 192.168.160.24 | 192.168.60.2 .. 24 | 192.168.160.2 | 192.168.160.1 | 192.168.60.2 |

For the IPSecfwd application to send out traffic successfully, traffic destined for the P4080DS ports must have the appropriate source and destination addresses.

Console messages are printed for each entry added to the SA/routing table. Once all configuration is completed, the application moves to the packet processing phase - the message "Application Started successfully" is printed on the console. At this point, traffic can be sent to the IPSecfwd application which would do its processing on the cpu-range specified by the user on the application command-line.

## 40.5.3 Running IPSec forwarding on P3041/P5020 board

The instructions below describe how to run *IPSecfwd* on P3041/P5020. Traffic should only be directed to P3041/P5020 once the application is running and configuration via the *ipsecfwd_config* application is completed.

•On linux prompt, assign IP address to eth0

```
$ ifconfig eth0 <IPADD> up
```

•Configure FMan PCD *using fmc* with the XML files in /usr/etc:

```
    $ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
    $ fmc -c usdpaa_config_p3_p5_serdes_0x36.xml -p usdpaa_policy_hash_ipv4.xml -a
```

•Run IPSecfwd application

```
    $ ipsecfwd_app <m..n> -c usdpaa_config_p3_p5_serdes_0x36.xml -p
usdpaa_policy_hash_ipv4.xml
```

For P3041, m..n can be 0..3. For P5020, m..n can be 0..1.

SSH to p3041/p5020 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipsecfwd_mix_15G.sh "pid"

```
    $  ipsecfwd_mix_15G.sh 2536
```

This is an example shell script available which creates SA Entries for the 5 x 1G and 1x10G interfaces for back to back configuration.

## 40.5.4 Running IPSec forwarding on T4240 board

The instructions below describe how to run *IPSecfwd* on T4240. Traffic should only be directed to T4240 once the application is running and configuration via the *ipsecfwd_config* application is completed.

•On linux prompt, assign IP address to eth0

```
$ ifconfig eth0 <IPADD> up
```

•Configure FMan PCD *using fmc* with the XML files in /usr/etc:

```
    $ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
    $ fmc -c usdpaa_config_t4_serdes_1_1_6_6.xml -p usdpaa_policy_hash_ipv4.xml -a
```

•Run IPSecfwd application

```
    $ ipsecfwd_app <m..n> -c usdpaa_config_t4_serdes_1_1_6_6.xml -p 
usdpaa_policy_hash_ipv4.xml -d 0x10000000 -b 0:0:1024
```

For T4240, m..n can be 0..23.

SSH to t4240 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipsecfwd_mix_20G.sh "pid"

```
    $  ipsecfwd_mix_20G.sh 2536
```

This is an example shell script available which creates SA Entries for the 2x10G interfaces for back to back configuration.

## 40.5.5 Running IPSec forwarding on B4860 board

The instructions below describe how to run *IPSecfwd* on B4860. Traffic should only be directed to B4860 once the application is running and configuration via the *ipsecfwd_config* application is completed.

•On linux prompt, assign IP address to eth0

```
$ ifconfig eth0 <IPADD> up
```

•Configure FMan PCD *using fmc* with the XML files in /usr/etc:

```
    $ cd /usr/etc
```

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

```
    $ fmc -c usdpaa_config_b4_serdes_0x2a_0x98.xml -p usdpaa_policy_hash_ipv4.xml -a
```

•Run IPSecfwd application

```
    $ ipsecfwd_app <m..n> -c usdpaa_config_b4_serdes_0x2a_0x98.xml -p 
usdpaa_policy_hash_ipv4.xml
```

For B4860, m..n can be 0..7.

SSH to b4860 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)

Run the shell script:

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

ipsecfwd_mix_20G.sh "pid"

```
    $  ipsecfwd_mix_20G.sh 2536
```

This is an example shell script available which creates SA Entries for the 2x10G interfaces for back to back configuration.

## 40.5.6 PPAC (and IPSecfwd) CLI commands

The following commands are illustrated in the context of IPSecfwd, but the commands are common to all PPAC-based applications.

The CLI (Command-Line Interface) allows you to add and remove additional threads to enable the use of multiple CPUs, with the only restriction being that the primary thread on CPU 1 cannot be removed (except by shutting down the application).

To add a thread on a single CPU (e.g. CPU 2):

> add 2

To add threads on a range of CPUs:

> add 3..6

To list the threads (this also queries each thread, verifying that they aren't blocked):

> list

Thread alive on cpu 1

Thread alive on cpu 2

Thread alive on cpu 3

Thread alive on cpu 4

Thread alive on cpu 5

Thread alive on cpu 6

To enable all interfaces

> macs on

To disable all interfaces

> macs off

To perform a controlled shutdown of ipsecfwd (this includes disabling the network ports):

> quit

## 40.5.7 IPSecfwd application Configuration command

### 40.5.7.1 Syntax

The syntax is as follows:

```
$ [root@p4080 bin]# ipsecfwd_config --help
Usage: ipsecfwd_config [OPTION...]
```

```
  -A  --SAadd=TYPE            adding an SA entry
  -B, --routeadd=TYPE         adding a route
  -C, --routedel=TYPE         deleting a route
  -D  --SAdel=TYPE            deleting an SA entry
  -E, --showintf=TYPE         show interfaces
  -F, --intfconf=TYPE         change intf config
  -G, --arpadd=TYPE           adding a arp entry
  -H, --arpdel=TYPE           deleting a arp entry
  -?, --help                  Give this help list
      --usage                 Give a short usage message
  -V, --version               Print program version
```

##### 40.5.7.1.1 Command to show all enabled interfaces and their interface numbers

The command to show all the enabled interfaces while running IPv4 forward is as follows:

Ipfwd_config -P pid -E -a true

Table 75: Field Description (show all enabled interfaces)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -a | Show all the interfaces | Yes | true |

Command to show all enabled interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# Ipfwd_config -P 2536 -E -a true

Interface number: 11

PortID=1:5 is FMan interface node

with MAC Address

02:00:c0:a8:65:fe

Interface number: 9

PortID=1:3 is FMan interface node

with MAC Address

02:00:c0:a8:5b:fe

Interface number: 8

PortID=1:2 is FMan interface node

with MAC Address

02:00:c0:a8:51:fe

Interface number: 5

PortID=0:5 is FMan interface node

with MAC Address

02:00:c0:a8:33:fe

Are all the Enabled Interfaces [root@p4080 bin]#

#### 40.5.7.1.2 Help for show all enabled interfaces command

The command to obtain help for show all enabled interfaces command is as follows:

```
ipsecfwd_config -E -help
Example 2. Help for show all enabled interfaces
 [root@p4080 etc]# ipsecfwd_config -E --help
Usage: -E [OPTION...]
  -a, --a=ALL                  All interfaces
  -?, --help                   Give this help list
      --usage                  Give a short usage message
  -V, --version                Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

#### 40.5.7.1.3 Assign IP address to interfaces

The command to assign IP address while running IPSecfwd is as follows:

```
ipsecfwd_config -P pid  -F -a 192.168.60.1 -i <Interface number>
```

Note: The interface number to be used here must be one of the numbers that got displayed as the output of "show all enabled interfaces command" in section 2.8.1.1.

Table 76: Field description (assign IP address to interfaces)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -a | IP Address | Yes | a.b.c.d |
| -i | Interface number | Yes | 0-11<br>(Choose this number from "show all enabled interfaces" command output) |

Example 3. Assign IP address to interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
ipsecfwd_config -P 2536 -F -a 192.168.60.1 -i 5
IPADDR assigned = 0xc0a83c01 to interface num 5
Intf Configuration Changed successfully
```

#### 40.5.7.1.4 Help for assign IP address to interfaces

The command to obtain help for assign IP address to interfaces command is as follows:

```
ipsecfwd_config -F --help
```

Example 4. Help for assign IP address to interfaces

```
[root@p4080 etc]# ipsecfwd_config -F --help
Usage: -F [OPTION...]
  -a, --a=IPADDR               IP Address
  -i, --i=IFNAME               If Name
```

```
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

#### 40.5.7.1.5 Adding an SA Entry

The command to add a SA while running IPSecfwd application is as follows:

```
ipsecfwd_config  -P pid  -A -a "AH SA configuration!" -e "encryption key" -s a.b.c.d -
d b.c.d.e -g
c.d.e.f -G a.d.d.a -i 0 -r dir
```

Example 5. Adding an SA Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipsecfwd_config  -P 2536  -A -a "AH SA configuration!" -e "This is
128 bits" -s
192.168.10.2 -d 192.168.60.2 -g 192.168.61.254 -G 192.168.60.99 -i 0 -r in
```

SA Entry Added successfully

```
[root@p4080 bin]#
```

For the purpose of using ESN (Extended Sequence Number) feature, user is provided two optional parameters. One is -x, which is intended to tell if user wants to use ESN option or not. Other is -v, which the user can configure with some starting sequence number for the packets. For example :

```
[root@p4080 bin]# ipsecfwd_config  -P 2536  -A -a "AH SA configuration!" -e "This is
128 bits" -s
192.168.10.2 -d 192.168.60.2 -g 192.168.61.254 -G 192.168.60.99 -i 0 -r in -x 1 -v
4294967294
```

Table 77: Field Description (Adding an SA Entry)

| Parameter | Description | Mandatory | Format/Value |
|---|---|---|---|
| -s | Source IP Address | Yes | a.b.c.d |
| -d | Destination IP Address | Yes | a.b.c.d |
| -g | Left tunnel IP Address | Yes | a.b.c.d |
| -G | Right Tunnel IP Address | Yes | a.b.c.d |

*Table continues on the next page...*

Table 77: Field Description (Adding an SA Entry) (continued)

| | | | |
|---|---|---|---|
| -D | Default Gateway | No | a.b.c.d |
| -a | Authentication Key for HMAC-SHA1 | No (Default key is taken if not supplied) | "string" of length 20 bytes (default = 0x2122_2324_2526_2728_292a_2b2c_2d2e_2f30_3132_3334; |
| -e | Encryption Key for AES-CBC | No (Default key is taken if not supplied) | "string" of length 16 bytes. (default = 0x0102_0304_0506_0708_090a_0b0c_0d0e_0f10 |
| -i | SPI | Yes | Unsigned int |
| -r | Direction (encryption/ decryption) | Yes | in (decryption) or out (encryption) |
| -x | ESN option | No | Unsigned int |
| -v | Sequence number | No | Unsigned int |

## 40.5.7.1.6 Help for SA Entry Addition

The command to help add a SA while running IPSecfwd application is as follows:

```
ipsecfwd_config -A --help
```

Example 6. Help for Adding an SA Entry

```
[root@p4080 bin]#]# ipsecfwd_config -A --help
Usage: -A [OPTION...]
  -a, --a=AKEY                Authentication Key
  -d, --d=DESTIP              Destination IP
  -D, --dg=DEFGW              Default Gateway
  -e, --e=EKEY                Encryption Key
  -g, --ss=SRCGW              Source Gateway IP
  -G, --sd=SRCGW              Destination Gateway IP
  -i, --spi=SPI               SPI - 32 bit unsigned int
  -p, --p=PROTO               IPsec Proto type - ESP(0) {Default: 0}
  -r, --dir=DIR               DIR- in/ out
  -s, --s=SRCIP               Source IP
  -t, --t=ETYPE               Encryption Type - AES-CBC(0), 3DES-CBC(1)
                              {Default: 0}
  -v, --seq_num=SEQNUM        Sequence Number
  -x, --is_esn=ESN            Extended Sequence Number support
  -y, --y=ATYPE               Authentication Type - HMAC-SHA1(0) {Default: 0}
  -?, --help                  Give this help list
      --usage                 Give a short usage message
  -V, --version               Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

```
[root@p4080 bin]#
```

#### 40.5.7.1.7 Deleting an SA Entry

The command to delete an SA while running IPSecfwd is as follows:

```
ipsecfwd_config  -P pid  -D -s 192.168.10.2 -d 192.168.60.2 -g 192.168.61.254 -G
192.168.60.99 -i 0
```

Example 7. Deleting an SA Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
root@p4080 bin]# ipsecfwd_config -P 2536 -D -s 192.168.10.2 -d 192.168.60.2 -g
192.168.61.254 -G
192.168.60.99 -i 0
```

SA Entry Deleted successfully

```
[root@p4080 bin]#
```

Table 78: Field Description (Deleting an SA Entry)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -s | Source IP Address | Yes | a.b.c.d |
| -d | Destination IP Address | Yes | a.b.c.d |
| -g | Left tunnel IP Address | Yes | a.b.c.d |
| -G | Right Tunnel IP Address | Yes | a.b.c.d |
| -i | SPI | Yes | Unsigned int |

#### 40.5.7.1.8 1.4.1.8 Help for Deleting an SA Entry

The command to obtain help for SA entry deletion is as follows:

```
ipsecfwd_config -D --help
```

Example 8. Help for Deleting an SA Entry

```
[root@p4080 bin]# ipsecfwd_config -D --help
Usage: -D [OPTION...]
  -d, --d=DESTIP             Destination IP
  -g, --ss=SRCGW             Source gateway IP
  -G, --sd=DESTGW            Destination gateway IP
  -i, --spi=SPI              SPI
  -s, --s=SRCIP              Source IP
  -?, --help                 Give this help list
```

```
      --usage                   Give a short usage message
  -V, --version                 Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

```
[root@p4080 bin]#
```

### 40.5.7.1.9 Adding a Route Entry

The command to add a route while running IPSecfwd is as follows:

```
ipsecfwd_config -P pid -B -s a.b.c.d -d b.c.d.e -g c.d.e.f
```

Table 79: Field Description (Adding a Route Entry)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -s | Source IP Address | Yes | a.b.c.d |
| -d | Destination IP Address | Yes | a.b.c.d |
| -g | Gateway IP Address | Yes | a.b.c.d |

Example 9. Adding a Route Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipsecfwd_config  -P 2536 -B -s  192.168.29.2 -d  192.168.24.2 -g
192.168.24.2
```

Route Entry Added successfully

```
[root@p4080 bin]#
```

### 40.5.7.1.10 Help for Route Entry Addition

The command to obtain help for route entry addition is as follows:

```
ipsecfwd_config -B --help
```

Example 10. Help for Adding a Route Entry

```
[root@p4080 bin]# ipsecfwd_config -B --help
Usage: -B [OPTION]
  -d, --d=DESTIP                Destination IP
  -f, --f=FLOWID                Flow ID - (0 - 1024) {Default: 0}
  -g, --g=GWIP                  Gateway IP
  -s, --s=SRCIP                 Source IP
  -t, --t=TOS                   Type of Service - (0 - 256) {Default: 0}
  -?, --help                    Give this help list
```

```
      --usage                 Give a short usage message
  -V, --version               Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

## 40.5.7.1.11 Deleting a Route Entry

The command to delete a route while running IPSecfwd is as follows:

```
ipsecfwd_config  -P pid  -C -s a.b.c.d -d b.c.d.e
```

Table 80: Field Description (Deleting a Route Entry)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -s | Source IP Address | Yes | a.b.c.d |
| -d | Destination IP Address | Yes | a.b.c.d |

Example 11. Deleting a Route Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipsecfwd_config -P 2536 -C -s  192.168.29.2 -d  192.168.24.2 Route
Entry Deleted
successfully
```

## 40.5.7.1.12 Help for Deleting a Route Entry

The command to obtain help for route entry deletion is as follows:

```
ipsecfwd_config -C --help
```

Example 12. Help for Deleting a Route Entry

```
[root@p4080 bin]# ipsecfwd_config -C --help
Usage: -C [OPTION...]
  -d, --d=DESTIP              Destination IP
  -s, --s=SRCIP               Source IP
  -t, --t=TOS                 Type of Service - (0 - 256) {Default: 0}
  -?, --help                  Give this help list
      --usage                 Give a short usage message
  -V, --version               Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

#### 40.5.7.1.13 Adding an ARP Entry

The command to add an ARP entry while running IPSecfwd is as follows:

```
ipsecfwd_config  -P pid  -G -s a.b.c.d -m aa:bb:cc:dd:ee [-r true]
```

Table 81: Field Description (Adding an ARP Entry)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -s | Gateway IP Address | Yes | a.b.c.d |
| -m | Mac Address | Yes | aa:bb:cc:dd:ee |
| -r | Replace existing entry | No | true/ false {Default: false} |

Example 14. Adding an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipsecfwd_config -P 2536  -G -s  192.168.24.2 -m 02:00:c0:a8:33:fd -r
true
```

ARP Entry Added successfully

#### 40.5.7.1.14 Help for ARP Entry Addition

The command to obtain help for ARP entry addition is as follows:

```
ipsecfwd_config -G --help
```

Example 15. Help for Adding an ARP Entry

```
[root@p4080 etc]# ipsecfwd_config -G --help
Usage: -G [OPTION...]
  -m, --m=MACADDR            MAC Address
  -r, --r=Replace            Replace Exiting Entry - true/ false {Default:
                             false}
  -s, --s=IPADDR             IP Address
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

#### 40.5.7.1.15 Deleting an ARP Entry

The command to delete an ARP while running the IPSecfwd is as follows:

```
ipsecfwd_config   -P pid   -H -s a.b.c.d
```

Table 82: Field Description (Deleting an ARP Entry)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -s | Gateway IP Address | Yes | a.b.c.d |

Example 16. Deleting an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

```
[root@p4080 bin]# ipsecfwd_config -P 2536 -H -s  192.168.24.2
Arp Entry Deleted successfully
[root@p4080 bin]#
```

### 40.5.7.1.16 Help for Deleting an ARP Entry

The command to obtain help for ARP entry deletion is as follows:

```
ipsecfwd_config -H --help
```

Example 17. Help for Deleting an ARP Entry

```
[root@p4080 etc]# ipsecfwd_config -H --help
Usage: -H [OPTION...]
-s, --s=IPADDR             IP Address
  -?, --help                 Give this help list
      --usage                Give a short usage message
  -V, --version              Print program version
```

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

### 40.5.7.1.17 Adding a high bandwidth tunnel

A high bandwidth tunnel is same as normal tunnel except that it has capability to carry high volume of traffic. The purpose of this feature is to provide high throughput when only single IPSec tunnel is created. For non-high bandwidth tunnel only a single core can process a tunnel's packet at any point of time. High bandwidth tunnel option allows multiple cores to process in parallel the packets of a single tunnel.

To create a high bandwidth tunnel "-b 1" should be appended to command for creating a new security association as shown below:

```
ipsecfwd_config  -P pid  -b 1
```

Table 83: Field Description (Create a high bandwidth tunnel)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -b | High bandwidth tunnel enable | No | 1/0 [enable/ disable (default: false)] |

When a tunnel is in high bandwidth mode, it should show a higher throughput

## 40.6 References

1. USDPAA PPAC User Guide
2. QMan/BMan API Guide

## 40.7 Revision History

Document revision history.

Table 84: Revision history

| Version | Author | Description |
|---|---|---|
| 1.0 | Nipun Gupta | Initial Draft |

# Chapter 41
# Freescale Simple Crypto User Manual

## 41.1 Introduction

**About this Document**

The User Space Data Path Acceleration Architecture (USDPAA) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document provides the following:

- A summary of the USDPAA "simple crypto" application.
- Execution steps for the "simple crypto" application.

**Conventions**

This document uses the following conventions:

`Courier` is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.

## 41.2 USDPAA Simple Crypto Application

### 41.2.1 Overview

USDPAA Crypto Application is a multi-threaded Linux User space process. This application exercises the interface to the SEC4.0 engine for raw mode encryption, decryption, and authentication of data. Using the P4080 DPAA, the application generates traffic for the SEC4.0 engine and also processes the output from the SEC4.0 engine. The application also generates performance data for its SEC4.0 interactions.

The applications' threads are created using the standard pthreads library. In Linux User Space, any of the 8 cores can be configured to run a USDPAA Crypto application thread. A dedicated QMan software portal is assigned to a USDPAA application thread and each thread is affined to a core. Each core has its own dedicated Frame Queues to interact with the SEC4.0 block. Traffic is directed to the SEC4.0 via frame descriptor enqueues onto QMan frame queues, which are configured to deliver the frames to SEC4.0 engine for processing.

The SEC4.0 engine processes packets on the basis of commands passed in the form of a shared descriptor. A pointer to the shared descriptor is passed to the SEC4.0 in the ingress (towards the security engine) frame queue descriptors' *contextA* field. The egress FQID is used by SEC4.0 to return output to the application - this is passed in the *contextB* field of the ingress frame queue descriptor. Different SEC4.0 shared descriptors are created for different operation like encryption and decryption.

### 41.2.2 Parameters to the application

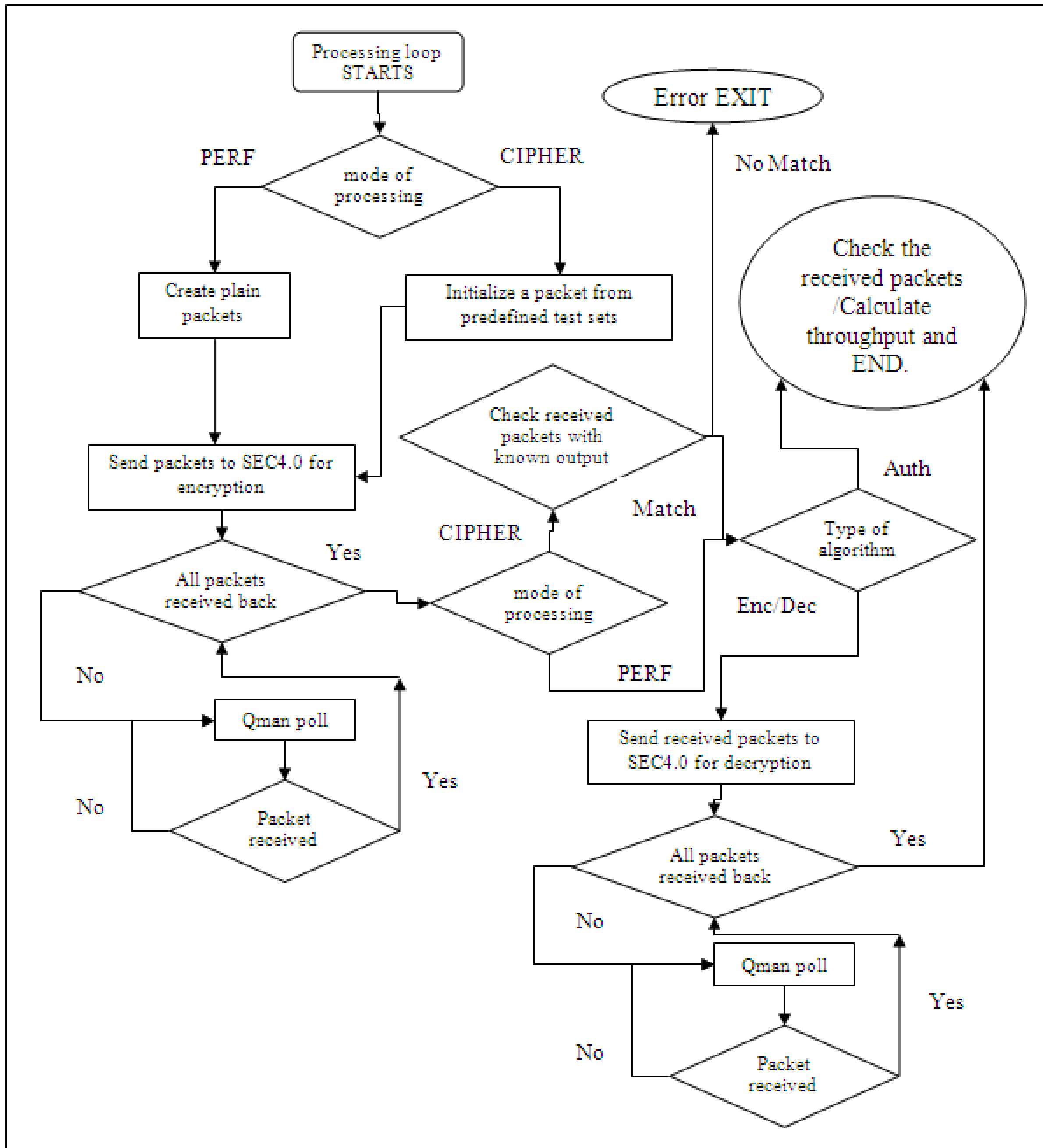
The crypto application supports various runtime parameters. These configurable parameters are passed to the crypto application from the command line while running the application.

1. Mode: Mode can be PERF or CIPHER

   a. **PERF** Mode: In PERF or Performance mode the application calculates the throughput of SEC4.0 processing of data (including enqueue and dequeue operations to and from SEC4.0 block).
   b. **CIPHER** mode: CIPHER mode allows a test run to demonstrate the SEC4.0 throughput and also compares ciphertext generated by SEC4.0 with ciphertext of a standard test vector.
2. Algorithm: This argument specifies the Algorithm to perform on the data. It is passed to SEC4.0 block using a Shared Descriptor. The algorithm choices are documented in Section Simple Crypto command syntax on page 706.
3. Number of Iterations: This parameter specifies the number of iterations of data to be looped through the SEC4.0 engine in a test run.
4. Number of buffers: It specifies the total number of buffers to send to SEC4.0 block from each core in one encryption/decryption or authentication iteration. These buffers are distributed among each core.
5. Size of the buffer: This argument is used only with PERF mode. It specifies the size of the each input buffers sent to SEC4.0.
6. Test set number: This argument is used only with CIPHER mode. It specifies the predefined test set to be used as the data set to send to SEC4.0. There are various test sets (with varying size and data) hardcoded in the application. These are documented in the Section Simple Crypto command syntax on page 706.
7. Number of cores: This is an optional parameter. By default the application uses all the active cores for the processing of the data sets. By specifying the number of cores, the user can limit the application threads to a specific number of cores.

## 41.2.3 Packet Flow

Figure 69: Packet Flow




After initializing the dedicated ingress and egress frame queues for a SEC4.0 operation, the application creates compound frame descriptors (FD's) and fills plain text or pre-defined "cipher" text in input buffers for SEC processing on the basis of the specified mode of operation. For PERF mode, this is plain text; and in the case of CIPHER mode it is pre-defined "cipher" text. The number of FD's equals to the total number of buffers. The FD's, which contain input and output buffer pointers, are first distributed amon the cores and then enqueued to the ingress FQ's. The application thread then polls for output packets from the SEC4.0 egress frame queue until all the packets are received back after being encrypted. As the application uses dedicated Frame Queues between cores and SEC4.0, each core receives the packets which it has enqueued.

In case of CIPHER mode the application checks the received encrypted packets with the already known result.

Lastly, the application checks for the algorithm type. If the algorithm is authentication, then it calculates the throughput in millions of bits per second (Mbps) and exits. If the algorithm type is encryption/decryption, the application sends the same packet (which it received from SEC4.0 after encryption) back to SEC4.0 block for decryption. It does this by changing the pointers of output buffers to point to input buffers and vice-versa in FD's and enqueues these to the SEC4.0's ingress FQ's. It then polls the packet from the SEC4.0 egress frame queue until all the encrypted packets are received back after decryption. The application checks if the packet received after decryption is same as the original plain/cipher text (as a packet after encryption followed by decryption should be the same as the original packet). It then calculates the throughput in Mbps and exits.

## 41.2.4 Throughput calculation

The application measures the CPU cycles just before enqueing the first packet on the FQ and just after receiving the last packet after processing from SEC4.0 for each iteration. The difference between these is the 'delta_cycles' which is accumulated over all the iterations.

Throughput of the application is reported in millions of bits per second (Mbps).

Throughput calculation involves the following parameters.

- 'l' is the number of iterations the application runs for in a test run
- 'n' is the total number of buffers
- 's' is the size of buffer
- 'cpu_freq' is the CPU frequency in MHz

The cycles per frame equals:

cycles_per_frame = (delta_cycles) /( l*n);

Throughput in Mbps equals:

Throughput = (cpu_freq * bits_per_byte *s) / (cycles_per_frame);

= (cpu_freq * 8 *s) / (cycles_per_frame);

## 41.2.5 Running Simple Crypto Application on board

1. On the Linux prompt on USDPAA, run the application by typing the following command:

   ```
   simple_crypto -m <mode> -s <size> -n <num_buffer> -o <algo> -l <num_iterations> -t
   <test_set> [-c <num_cores>]
   ```

   Refer to section Simple Crypto command syntax on page 706 for command syntax.

2. Upon successful completion, the application shows the following message on the USDPAA boot core's console. In case of a failure, a failure message is displayed

   ```
   INFO: SEC4.0 test PASSED
   ```

   Also upon successful completion, the application reports SEC4.0 raw algorithm's throughput on boot core's console.

## 41.2.6 Simple Crypto command syntax

The command syntax is as follows:

[root@p4080 root]# simple_crypto --help

Usage: simple_crypto [OPTION...]

-c, --ncpus=CPUS

OPTIONAL PARAMETER

Number of cpus to work for the Application (1-8)(OPTIONAL)

-l, --itrnum=ITERATIONS

Number of iteration to repeat

-m, --mode=TEST MODE

test mode: specify one of the following

1 for perf

2 for cipher

Only the following two combinations are valid. All options are mandatory:

-m 1 -s <buf_size> -n <buf_num_per_core>

-o <algo> -l <itr_num>

or

-m 2 -t <test_set> -n <buf_num_per_core>

-o <algo> -l <itr_num>

-n, --bufnum=TOTAL BUFFERS

Number of buffers per core (1-6400)

-o, --algo=ALGORITHM

Cryptographic operation to be performed by SEC4.0

Specify one of the following:

1 for AES_CBC

2 for TDES_CBC

3 for SNOW_F8

4 for SNOW_F9

5 for KASUMI_F8

6 for KASUMI_F9

7 for CRC

8 for HMAC_SHA1

9 for SNOW_F8_F9(only with PERF mode)

-s, --bufsize=BUFFER SIZE

OPTION IS VALID ONLY IN PERF MODE

Buffer size (64, 128 ...upto 6400)

-t, --testset=TEST SET

OPTION IS VALID ONLY IN CIPHER MODE

provide following test set number:

AES_CBC: 1-4

TDES_CBC: 1-2

SNOW_F8: 1-5

SNOW_F9: 1-5

KASUMI_F8: 1-5

KASUMI_F9: 1-5

CRC: 1-5

HMAC_SHA1: 1-2

SNOW_F8_F9: 1

-?, --help Give this help list

--usage Give a short usage message

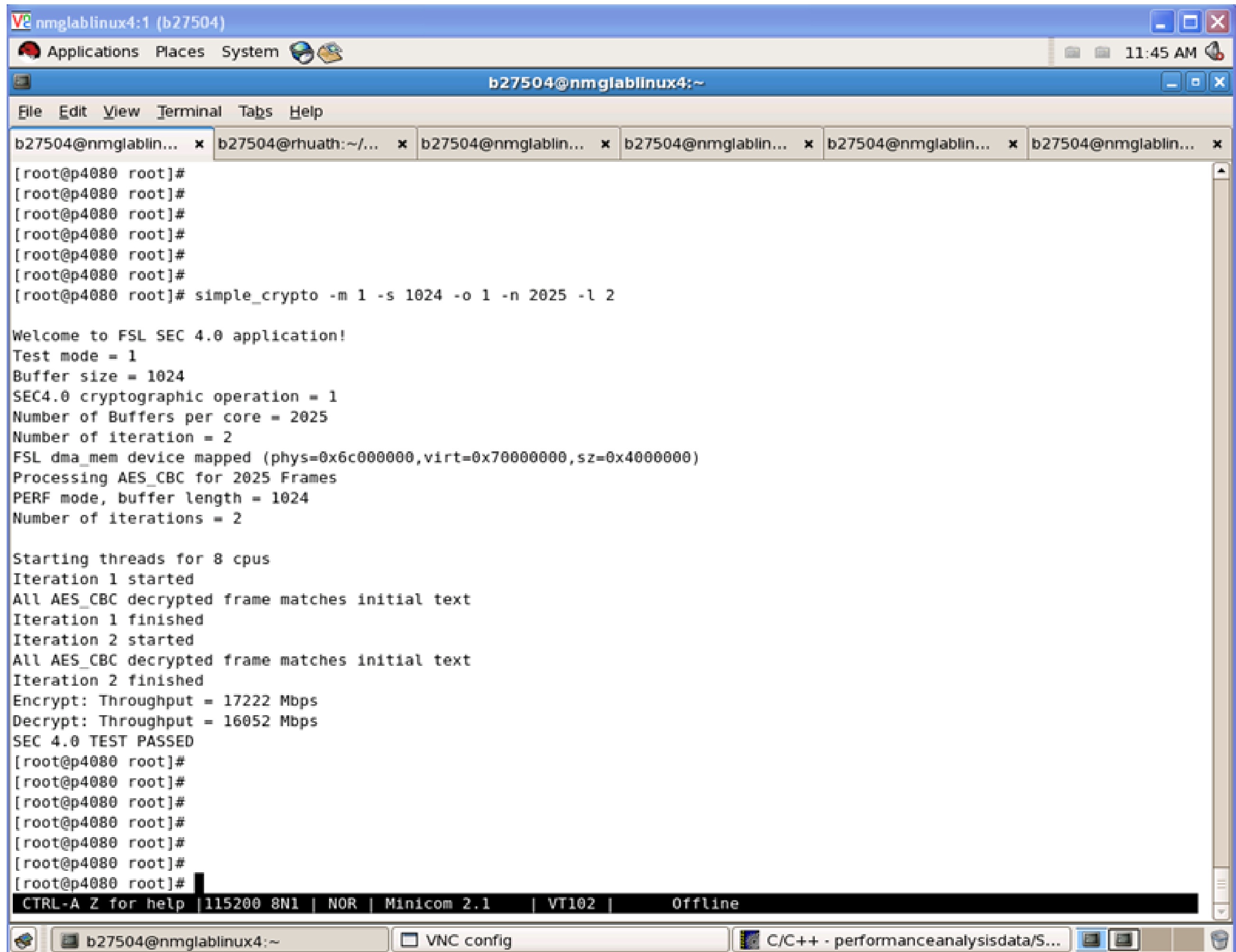
Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

Note: For For p3041 the ncpus(-c) varies from 1-4 and for p5020 it varies from 1-2.

### 41.2.7 Snapshot of Simple Crypto output

The figure below shows a snapshot of simple crypto application output.

Figure 70: Snapshots of Simple Crypto Application



# Chapter 42
# Freescale Simple Proto User Manual

## 42.1 Introduction

**About this Document**

The USDPAA Simple Proto application demonstrates the usage of security coprocessor's capabilities in handling traffic in security protocols context

This document provides the following:

- A summary of the USDPAA "simple proto" application.
- Execution steps for running "simple proto" application.

**Conventions**

This document uses the following conventions:

`Courier` is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.

## 42.2 USDPAA Simple Proto Application

## 42.3 Overview

USDPAA Simple Proto application demonstrates usage of SEC engine protocols' integrity and confidentiality algorithms. Using the DPAA framework, the application generates traffic and enqueues it to SEC engine, process the ouput and generate performance data for its SEC interactions.

This is a multi-threaded Linux User Space application. Threads are created using pthreads library, each application thread has an assigned QMan software portal and is affined to a core. Each core has its own dedicated Frame Queues to interact with SEC. Traffic is injected to SEC via frame descriptors enqueued onto QMan frame queues.

The SEC engine processes packets on the basis of commands passed in the form of a shared descriptor. A pointer to the shared descriptor is passed to the SEC in the ingress frame queue descriptors' *contextA* field. The egress FQID is used by SEC to return output to the application - this is passed in the *contextB* field of the ingress frame queue descriptor. Different SEC shared descriptors are created for different protocols' operation.

## 42.4 Parameters to the application

The simple_proto application supports various runtime parameters. These configurable parameters are passed to the application from the command line when running the application.

1. Mode: Mode can be PERF or CIPHER

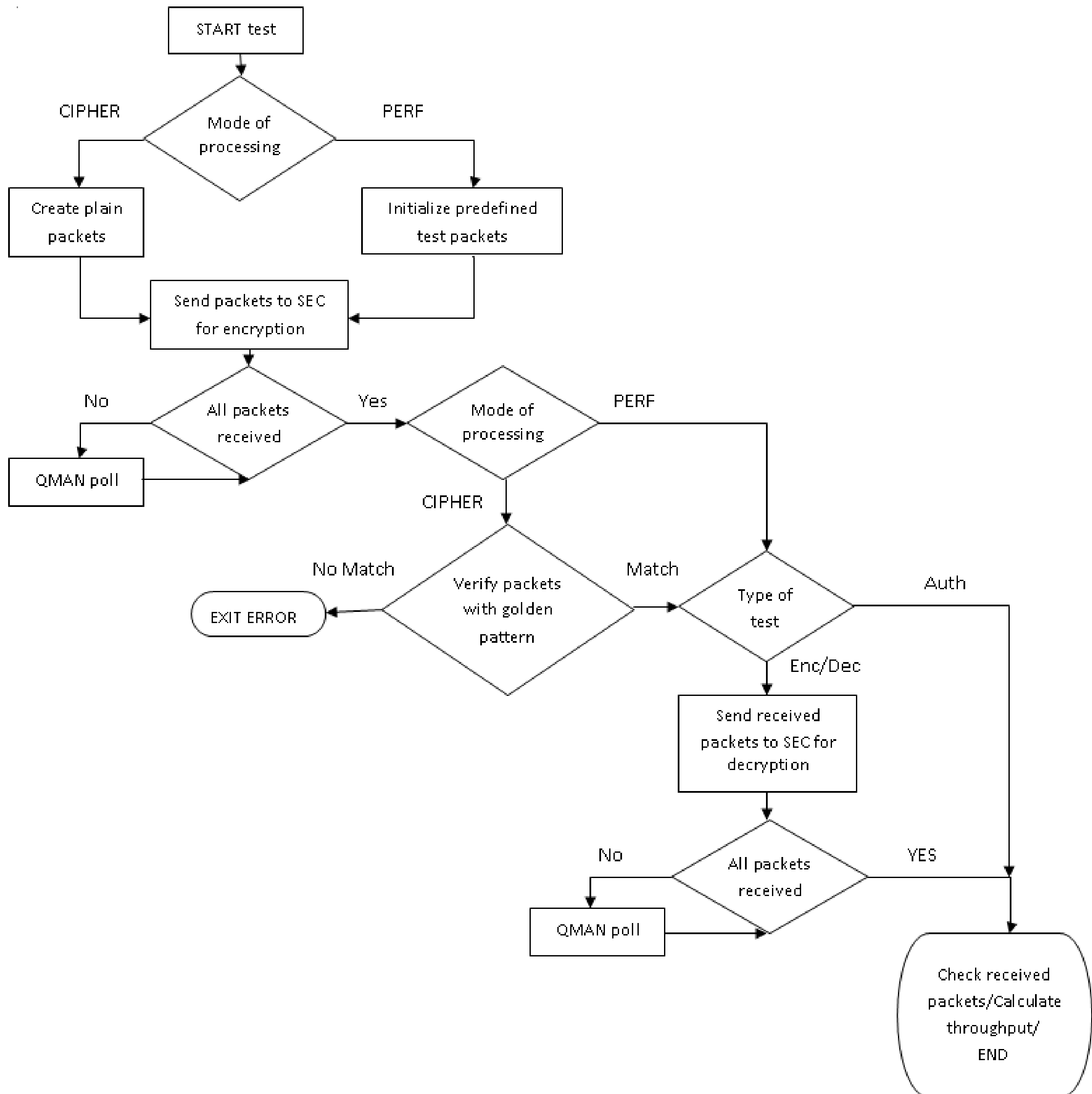
a. **PERF** Mode: In PERF or Performance mode the application calculates the throughput of SEC processing of data (including enqueue and dequeue operations to and from SEC block). Also, output frames from decapsulation are compared against input frames to encapsulation.

b. **CIPHER** mode: CIPHER mode allows a test run to demonstrate the SEC throughput and also compares ciphertext generated by SEC with ciphertext of a standard test vector.

2. Protocol: This argument specifies the protocol to be tested. It is passed to SEC block using a Shared Descriptor. The protocol choices are documented in Section Simple Proto command syntax on page 715. Each protocol has its own set of mandatory and/or optional parameters, which are explained in MACSec protocol options on page 716, WiMAX protocol options on page 716, PDCP protocol options on page 717 and MBMS Protocol Options on page 720
3. Number of Iterations: This parameter specifies the number of iterations of data to be looped through the SEC engine in a test run.
4. Number of buffers: This specifies the total number of buffers to send to SEC block from each core in one encryption/decryption or authentication iteration. These buffers are distributed among each core.
5. Size of the buffer: This argument is used only with PERF mode. It specifies the size of the each input buffers sent to SEC.
6. Test set number: This argument is used only with CIPHER mode. It specifies the predefined test set to be used as the data set to send to SEC. There are various test sets (with varying size and data) hardcoded in the application. These are documented in the Section Simple Proto command syntax on page 715.
7. Number of cores: This is an optional parameter. By default the application uses all the active cores for the processing of the data sets. By specifying the number of cores, the user can limit the application threads to a specific number of cores.
8. SEC Era: This is an optional parameter. This specifies the SEC era hardware block revision for which SEC descriptors will be generated. **By default, the application runs with default era set to value 2.** By specifying the SEC era, the user can set the right value for the targeted platform to test. For example, SEC era on the following platforms is:
   - 2 for P4080 TO2
   - 3 for P3041, P5020
   - 4 for P4080 TO3
   - 5 for P5040, B4860
   - 6 for T4240, T2080 and T1040
   - 7 for LS1021

## 42.5 Packet Flow

Figure 71: Packet Flow




After initializing the dedicated ingress and egress frame queues for a SEC operation, the application creates compound frame descriptors (FD's) and prepare input buffers for SEC processing based on specified mode of operation; for PERF mode, buffers are filled in with incrementing pattern plain text (for WiMAX encapsulation, SEC expects GMH aware input frames: GMH Header Type bit set to zero, the CRC indicatr bit set to one if CRC is included in the PDU, and the GMH LEN field updated accordingly to the input plain data length), and in the case of CIPHER mode, with golden pattern data. The number of FD's equals to the total number of buffers and

are first distributed among the cores and then enqueued to the ingress FQ's. The application thread then polls for output packets from the SEC egress frame queue until all the packets are received back after being encapsulated. As the application uses dedicated Frame Queues between cores and SEC, each core receives the packets which it has enqueued. In CIPHER test mode, the application checks the received encapsulated packets against a golden pattern.

The next step is the application to verify the type of test. If the test set is unidirectional , then it calculates the throughput in millions of bits per second (Mbps) and exits. If the test verifies both encapsulation/decapsulation, the application sends the packet which it received from SEC after encapsulation back to SEC block for decapsulation. It does this by interchanging the pointers to output and input buffers in FD's and enqueues these to the SEC's ingress FQ's. It then polls the packet from the SEC egress frame queue until all the encapsulated packets are received back after decapsulation. The application checks if the packet received after decapsulation is the same as the original plain/cipher text (as a packet after encapsulation followed by decapsulation should be the same as the original packet). It then calculates the throughput in Mbps and exits.

## 42.6 Throughput calculation

The application measures the CPU cycles just before enqueuing the first packet on the FQ and just after receiving the last packet after processing from SEC for each iteration. The difference between these is the 'delta_cycles' which is accumulated over all the iterations.

Throughput of the application is reported in millions of bits per second (Mbps).

Throughput calculation involves the following parameters.

- 'I' is the number of iterations the application runs for in a test run
- 'n' is the total number of buffers
- 's' is the size of buffer
- 'cpu_freq' is the CPU frequency in MHz

The cycles per frame equals:

cycles_per_frame = (delta_cycles) /( I*n);

Throughput in Mbps equals:

Throughput = (cpu_freq * bits_per_byte *s) / (cycles_per_frame);

= (cpu_freq * 8 *s) / (cycles_per_frame);

## 42.7 Running Simple Proto Application on board

1. On the Linux prompt on USDPAA, run the application by typing the following command:

   ```
   simple_proto -m <mode> -s <size> -n <num_buffer> -p <protocol> -l <num_iterations> -t
   <test_set> [-c <num_cores> -e <sec_era>]
   ```

   Refer to section Simple Proto command syntax on page 715 for command syntax.

2. Upon successful completion, the application shows the following message on the USDPAA boot core's console:

   ```
   INFO: SEC4.0 test PASSED
   ```

   Also upon successful completion, the application reports SEC4.0 raw algorithm's throughput on boot core's console.

In case of failure, a failure message is displayed on console.

## 42.8 Simple Proto command syntax

The command syntax is as follows:

```
root@p4080ds:~# simple_proto --help
Usage: simple_proto [OPTION...]

  -c, --ncpus=CPUS            OPTIONAL PARAMETER

                              Number of cpus to work for the application(1-8)

  -e, --sec_era=ERA           OPTIONAL PARAMETER

                              SEC Era version on the targeted platform(2-5)

  -l, --itrnum=ITERATIONS     Number of iterations to repeat

  -m, --mode=TEST MODE        Test mode:
                                 1 for perf
                                 2 for cipher

                              Following two combinations are valid only and all
                              options are mandatory:
                              -m 1 -s <buf_size> -n <buf_num_per_core> -p
                              <proto> -l <itr_num>
                              -m 2 -t <test_set> -n <buf_num_per_core> -p
                              <proto> -l <itr_num>

  -n, --bufnum=TOTAL BUFFERS Total number of buffers (1-6400). Both of Buffer
                              size and buffer number cannot be greater than 3200
                              at the same time.

  -p, --proto=PROTOCOL        Cryptographic operation to perform by SEC:
                               1 for MACsec
                               2 for WiMAX
                               3 for PDCP
                               4 for SRTP
                               5 for WiFi
                               6 for RSA
                               7 for TLS
                               8 for IPsec
                               9 for MBMS


  -s, --bufsize=BUFSIZE       OPTION IS VALID ONLY IN PERF MODE

                              Buffer size (64, 128 ... up to 6400). Note: Both
                              of Buffer size and buffer number cannot be greater
                              than 3200 at the same time.
                              The WiMAX frame size, including the FCS if
                              present, must be shorter than 2048 bytes.

  -t, --testset=TEST SET      OPTION IS VALID ONLY IN CIPHER MODE

  -?, --help                  Give this help list
```

```
       --usage                Give a short usage message

Mandatory or optional arguments to long options are also mandatory or optional
for any corresponding short options.
```

NOTE

Depending on the hardware platform, the ncpus(-c) varies as follows: for p3041 between 1-4, for p5020 between 1-2, for B4860 between 1-8 and for T4240 between 1-24.

NOTE

The valid test set numbers are the following, per each protocol:

1. MACSec - 1 .. 5
2. WiMAX - 1 .. 4
3. PDCP - 1
4. SRTP - 1
5. WiFi - 1 .. 2
6. RSA - 1 .. 2
7. TLS - 1
8. IPsec - 1
9. MBMS:
    a. MBMS PDU Type 0 - 1 .. 2
    b. MBMS PDU Type 1 - 1 .. 3
    c. MBMS PDU Type 3 - 1 .. 3

# 42.9 MACSec protocol options

For MACSec processing, the simple_proto application understands the following parameters (apart from the mandatory & optional ones specified in Simple Proto command syntax on page 715):

- -o, --algo : this is an optional parameter that, when set, allows the user to choose the cipher type. The cipher type can be GCM or GMAC; by default, the MACSec protocol will use GCM processing.

# 42.10 WiMAX protocol options

WiMAX processing is available only if SEC Era if equal or grater than 4.

For WiMAX processing, the simple_proto application understands the following parameters (apart from the mandatory & optional ones specified in Simple Proto command syntax on page 715):

- -a, --ofdma : this is an optional parameter that, when set, it enables OFDMA processing for WiMAX. By default, the WiMAX protocol offload does OFDM processing;
- -f, --fcs : this is an optional parameter that, when set, instructs the WiMAX protocol offload to compute the FCS over the input frame, making it longer by 4 bytes;

- -w, --ar_len=ARWIN : another optional parameter that enables the anti-replay mechanism in WiMAX protocol processing. This parameter also sets the anti-replay window length, which cannot exceed 64 frames.

## 42.11 PDCP protocol options

Packet Data Convergence Protocol (abbrev. PDCP) is one of the layers of the Radio Traffic Stack in UMTS and performs IP header compression and decompression, transfer of user data and maintenance of sequence numbers for Radio Bearers which are configured for lossless serving radio network subsystem (SRNS) relocation.

In simple_proto application, the following protocol sub-sets are tested & supported:

1. PDCP Control Plane;
2. PDCP User Plane;
3. PDCP Short MAC.

The PDCP ciphering & integrity algorithm combinations supported by simple_proto application are the following:

1. PDCP Control Plane:
    a. NULL encryption & NULL integrity (EEA0/EIA0)
    b. NULL encryption & SNOW f9 integrity (EEA0/EIA1)
    c. NULL encryption & AES CMAC integrity (EEA0/EIA2)
    d. NULL encryption & ZUC integrity (EEA0/EIA3)*
    e. SNOW f8 encryption & NULL integrity (EEA1/EIA0)
    f. SNOW f8 encryption & SNOW f9 integrity (EEA1/EIA1)
    g. SNOW f8 encryption & AES CMAC integrity (EEA1/EIA2)
    h. SNOW f8 encryption & ZUC integrity (EEA1/EIA3)*
    i. AES CTR encryption & NULL integrity (EEA2/EIA0)
    j. AES CTR encryption & SNOW f9 integrity (EEA2/EIA1)
    k. AES CTR encryption & AES CMAC integrity (EEA2/EIA2)
    l. AES CTR encryption & ZUC integrity (EEA2/EIA3)*
    m. ZUC encryption & NULL integrity (EEA3/EIA0)
    n. ZUC encryption & SNOW f9 integrity (EEA3/EIA1)
    o. ZUC encryption & AES CMAC integrity (EEA3/EIA2)
    p. ZUC encryption & ZUC integrity (EEA3/EIA3)*
2. PDCP User Plane:
    a. NULL encryption (EEA0)
    b. SNOW f8 encryption (EEA1)
    c. AES CTR encryption (EEA2)
    d. ZUC encryption (EEA3)*
3. PDCP Short MAC:

a. NULL integrity (EIA0)

b. SNOW f9 integrity (EIA1)

c. AES CMAC integrity (EIA2)

d. ZUC integrity (EIA3)*

NOTE

Starred combinations above are available only for platforms with SEC ERA greater than 4 (for instance P5040/B4860R1&R2/T4240/etc.). Attempting to run these combinations on platforms without the proper SEC ERA version will result in a SEC error.

NOTE

For the following combinations used for decapsulating PDCP PDUs, the SEC will return an error code similar to 0x3000XX0a if the last 4 bytes of the decapsulated frame (the ICV) are not set to the value of {0x00, 0x00, 0x00, 0x00}

1. EEA1/EIA0
2. EEA2/EIA0
3. EEA3/EIA0

The following parameters can be provided to the simple_proto application (besides the optional & mandatory parameters specified in Simple Proto command syntax on page 715):

| Parameter | Explanation | Valid for Control Plane? | Valid for User Plane? | Valid for Short MAC? |
| --- | --- | --- | --- | --- |
| -d, --direction | Selects the downlink direction for the inserted PDU; by default, the direction of thePDU is assumed to be uplink. | Yes, optional | Yes, optional | No |
| -i, --integrity | Selects the integrity algorithm to be used for processing the PDU | Yes, mandatory | No | Yes, mandatory |
| -r, --cipher | Selects the ciphering algorithm to be used for processing the PDU | Yes, mandatory | Yes, mandatory | No |
| -v, --hfn_ov | Enables the HFN value used for processing the PDU to be specified by the user. | Yes, optional | Yes, optional | No |
| -x, --snlen | Select the User Plane PDUs sequence number length. Three values are permitted:<br>0 = 12 bit Sequence Number PDU<br>1 = 7 bit Sequence Number PDU<br>2 = 15 bit Sequence Number PDU | No | Yes, optional | No |
| -y, --type | Selects the way the input PDU is to be treated:<br>0 = Control Plane<br>1 = User Plane<br>2 = Short MAC | Yes, mandatory | Yes, mandatory | Yes, mandatory |

## 42.12 RSA operations options

For RSA processing, the simple_proto application understands the following parameter (apart from the mandatory & optional ones specified in Simple Proto command syntax on page 715):

- -b, --form : this is an optional parameter that, when set, allows the user to choose one of the three RSA Decrypt Private Key formats:

  1 = Form 1 (default)

  2 = Form 2

  3 = Form 3

## 42.13 TLS protocol options

Transport Layer Security (TLS) and its predecessor, Secure Sockets Layer (SSL), are cryptographic protocols designed to provide communication security over the Internet.

simple_proto application tests and supports the TLS10 security protocol.

The following parameters can be provided to the simple_proto application (besides the optional & mandatory parameters specified in Simple Proto command syntax on page 715):

| Parameter | Explanation | Optional / Mandatory |
|---|---|---|
| -j, --cipher | Selects the ciphering algorithm to be used for processing the PDU:<br>0 = AES-CBC | mandatory |
| -k, --integrity | Selects the integrity algorithm to be used for processing the PDU:<br>0 = HMAC-SHA1 | mandatory |
| -g, --version | Select the SSL protocol version to run:<br>0 = SSL30 (not supported)<br>1 = TLS10<br>2 = TLS11 (not supported)<br>3 = TLS12 (not supported)<br>4 = DTLS10 (not supported) | mandatory |

## 42.14 IPsec protocol options

Internet Protocol Security (IPsec) is a protocol suite for securing Internet Protocol (IP) communications by authenticating and encrypting each IP packet of a communication session.

simple_proto application tests and supports 3des & hmac-md5-96 IPsec ESP Tunnel mode.

The following parameters can be provided to the simple_proto application (besides the optional & mandatory parameters specified in Simple Proto command syntax on page 715):

| Parameter | Explanation | Optional / Mandatory |
|---|---|---|
| -h, --cipher | Selects the ciphering algorithm to be used for processing the PDU:<br>0 = 3DES | mandatory |
| -q, --integrity | Selects the integrity algorithm to be used for processing the PDU:<br>0 = HMAC_MD5_96 | mandatory |

## 42.15 MBMS Protocol Options

MBMS SYNC protocol is defined in 3GPP TS 25.446 - MBMS synchronisation protocol (SYNC)

The MBMS Synchronisation protocol (SYNC) is located in the User plane of the Radio Network layer over the Iu interface: the Iu UP protocol layer.

The SYNC protocol for UTRAN is used to convey userdata associated to MBMS Radio Access Bearers.

The simple_proto application supports checking for the CRC validity of the following MBMS PDU Types:

1. MBMS PDU Type 0
2. MBMS PDU Type 1
3. MBMS PDU Type 3

The following table summarizes the behavior of the MBMS SYNC processing:

| MBMS SYNC PDU | Default action | Header CRC fail action | Payload CRC fail action(s) |
|---|---|---|---|
| Type 0 | Copy PDU | Drop PDU | N/A |
| Type 1 | Copy PDU | Drop PDU | 1. Update Payload CRC in PDU's header<br>2. Copy Header only |
| Type 3 | Copy PDU | Drop PDU | 1. Update Payload CRC in PDU's header<br>2. Copy Header only |

The following table lists SEC return codes used for signaling the different actions the SEC takes in order to process the MBMS SYNC PDUs:

| Processing result | SEC status/command |
|---|---|
| PDU Header & Payload CRC OK | 0x0000_0000 |
| Wrong PDU Header CRC | 0x3000_XXAA |
| Wrong PDU Payload CRC | 0x3000_XXAB |

NOTE

The "XX" in the above rows is an internal offset used by SEC and can be safely masked out when checking the SEC status.

The following parameters can be provided to the simple_proto application (besides the optional & mandatory parameters specified in Simple Proto command syntax on page 715

| Parameter | Explanation | Valid values |
|---|---|---|
| -z, --type | Selects the MBMS PDU Type to be processed | • 0 - MBMS PDU Type 0<br>• 1 - MBMS PDU Type 1<br>• 3 - MBMS PDU Type 2 |

# Chapter 43
# SEC Descriptor construction library (DCL)

## 43.1 SEC Descriptor Construction Library (DCL)

A description of the SEC descriptor construction library (DCL) as a library within USDPAA.

The following is a description of the SEC descriptor construction library (DCL) as a library within USDPAA:

- DCL Description on page 723, provides a brief overview of the DCL
- DCL Packaging on page 723, describes how the DCL is packaged
- DCL Files on page 723, lists the supported files
- DCL Functional Description on page 724, describes the three layers of the DCL:
  - Lower layers-Command Generator on page 724, and Descriptor Disassembler on page 724
  - Upper layer Upper-Tier DCL Functions - Descriptor Constructors on page 738

## 43.2 DCL Description

The Descriptor Construction Library (DCL) provides a collection of simple functions capable of building SEC4.x descriptors for a wide range of purposes, either as a standalone library or integrated within a driver subsystem. Applications may use all, part, or none of the DCL's functions, or they may use DCL as a simple reference for their own construction functions.

The DCL package will evolve over time. Additional protocol support and descriptor utilities will be added in future software (SW) releases, as new applications for SEC 4.x are developed.

## 43.3 DCL Packaging

DCL is packaged with and used only by USDPAA sample applications.

Types in DCL are defined using POSIX conventions for the sake of external portability.

## 43.4 DCL Files

The following files are supported

- `cmdgen.c`-Descriptor command generator.
- `disasm.c`-Descriptor disassembler.
- `jobdesc.c`-Job descriptor constructors.
- `protoshared.c`-Shared/protocol descriptor constructors.
- `dcl.h`-Definitions for all published DCL functions.

# 43.5 DCL Functional Description

DCL consists of two layers, decomposed into three subsystems:

- A lower tier, comprising the following:
  - Command generator (described in Command Generator on page 724), colloquially referred to as "cmdgen".
  - Descriptor disassembler (described in Descriptor Disassembler on page 724)
- An upper tier, composed of descriptor constructors (described in Upper-Tier DCL Descriptor Constructors on page 725). This is dependent on the command generator in the lower tier.

# 43.6 Command Generator

The Command Generator is the lowest level of DCL functionality. Each function within it is capable of generating a single command/instruction in a SEC4.x descriptor and increments a "next in" pointer to the next descriptor word following the generated command.

Applications may use the command generator independently of any other DCL functionality.

In general, creation of any application needing to generate a descriptor on an individual command basis like this starts by building the first commands (or PDB data) past the header, until the body of the descriptor is complete. At this point, the full size of the descriptor in known; the application can then fill in the header using this size.

# 43.7 Descriptor Disassembler

The Descriptor Disassembler is meant to be a simple debug tool that can display the content of a constructed descriptor for the user to see in a simple "disassembled" representation. It is intended for developers to use as a visualization aid during development, or as a debug tool, in which descriptor content can be displayed on-the-fly in a human-decipherable form. It does not perform consistency checking, or otherwise identify problem areas in poorly formed descriptors.

The disassembler is a simple C function, and can be packaged in the library with the balance of DCL functions so that it may be linked into a higher-level application.

An example of a disassembled IPSec CBC decapsulation shared descriptor:

```
shrdesc: stidx=8 len=20 share-always
     (pdb): [00] 0x00340001 0x00000000 0x00000000 0x00000000
     (pdb): [04] 0x00000000 0x00000000 0x00000000
       key: len=20 class2->keyreg inline
            [00] 0x000e0f00 0x0d0f0a00 0x0d0f0a00 0x0d0f0a00
            [04] 0x0d0f0a00
       key: len=16 class1->keyreg inline
            [00] 0x00e0f0a0 0x00d0f0a0 0x00e0f0a0 0x00d0f0a0
 operation: type=decap-pcl ipsec aes-cbc hmac-sha1-96
```

An example of a disassembled IPSec CBC encapsulation shared descriptor:

```
shrdesc: stidx=23 len=35 share-always
     (pdb): [00] 0x0000000d 0x00000000 0x00000000 0x00000000
```

```
      (pdb): [04] 0x00000000 0x00000000 0x00000000 0x00000000
      (pdb): [08] 0x00000034 0x34001045 0x00402512 0xd2860640
      (pdb): [12] 0x7746430a 0xc046430a 0x160022d0 0x5d891888
      (pdb): [16] 0x9cee1912 0xbc211080 0x00000898 0x0a080101
      (pdb): [20] 0x22759aa6 0xdb143f08
        key: len=20 class2->keyreg inline
             [00] 0x000e0f00 0x0d0f0a00 0x0d0f0a00 0x0d0f0a00
             [04] 0x0d0f0a00
        key: len=16 class1->keyreg inline
             [00] 0x00e0f0a0 0x00d0f0a0 0x00e0f0a0 0x00d0f0a0
  operation: type=encap-pcl ipsec aes-cbc hmac-sha1-96
```

## 43.8 Upper-Tier DCL Descriptor Constructors

A higher level of functionality is provided through complex descriptor constructors. These constructors are single-purpose functions capable of generating complete descriptors from user specifications. These constructors fit into two categories, one for job descriptors and another for shared descriptors generally targeted to protocol processing.

These constructors are by no means the "definitive" reference to all possible descriptor permutations, nor are they meant to work with any specific application. Instead, they are meant to be general-purpose examples of descriptor construction. It is expected that, over time, this library will grow to accommodate a wide range of examples.

All constructor functions are dependent on the underlying command generator.

## 43.9 API reference

### 43.9.1 API reference command generator

#### 43.9.1.1 cmd_insert_shared_hdr()

`cmd_insert_shared_hdr()`: Insert a shared descriptor header into a descriptor

```
u_int32_t *cmd_insert_shared_hdr(u_int32_t *descwd,
                                 u_int8_t startidx,
                                 u_int8_t desclen,
                                 enum ctxsave ctxsave,
                                 enum shrst share);
```

Inputs:

- `descwd`-pointer to target descriptor word to hold this command. Note that this should always be the first word of a descriptor.
- `startidx`-index to continuation of descriptor data, normally the first descriptor word past a PDB. This tells DECO what to skip over.
- `desclen`-length of descriptor in words, including header.
- `ctxsave`-Saved or erases context when a descriptor is self-shared
  - `CTX_SAVE` = context saved between iterations

- CTX_ERASE = context is erased
- share-Share state of this descriptor:
  - SHR_NEVER = Never share. Fetching is repeated for each processing pass.
  - SHR_WAIT = Share once processing starts.
  - SHR_SERIAL = Share once completed.
  - SHR_ALWAYS = Always share (except keys)

Returns:

Pointer to next incremental descriptor word past the header just constructed. If an error occurred, returns 0.

NOTE

Headers should normally be constructed as the final operation in the descriptor construction, because the start index and overall descriptor length will likely not be known until construction is complete. For this reason, there is little use to the "incremental pointer" convention. The exception is probably in the construction of simple descriptors where the size is easily known early in the construction process.

## 43.9.1.2 cmd_insert_hdr()

cmd_insert_hdr(): Insert a standard descriptor header into a descriptor

```
u_int32_t *cmd_insert_hdr(u_int32_t *descwd,
                           u_int8_t startidx,
                           u_int8_t desclen,
                           enum shrst share,
                           enum shrnext sharenext,
                           enum execorder reverse,
                           enum mktrust mktrusted);
```

Inputs:

- descwd-pointer to target descriptor word to hold this command. Note that this should always be the first word of a descriptor.
- startidx-index to continuation of descriptor data, or if sharenext = SHRNXT_SHARED, then specifies the size of the associated shared descriptor referenced in the following instruction.
- desclen-length of descriptor in words, including header.
- share-Share state for this descriptor:
  - SHR_NEVER-Never share. Fetching is repeated for each processing pass.
  - SHR_WAIT-Share once processing starts.
  - SHR_SERIAL-Share once completed.
  - SHR_ALWAYS-Always share (except keys)
  - SHR_DEFER-Use the referenced sharedesc to determine sharing intent
- sharenext-Control state of shared descriptor processing
  - SHRNXT_SHARED-This is a job descriptor consisting of a header and a pointer to a shared descriptor only.
  - SHRNXT_LENGTH-This is a detailed job descriptor, thus desclen refers to the full length of this descriptor.
- reverse-Reverse execution order between this job descriptor, and an associated shared descriptor:

  - ORDER_REVERSE-execute this descriptor before the shared descriptor referenced.
  - ORDER_FORWARD-execute the shared descriptor, then this descriptor.
- mktrusted-DESC_SIGN-sign this descriptor prior to execution
  - DESC_STD -leave descriptor non-trusted

#### 43.9.1.3 cmd_insert_key()

cmd_insert_key(): Insert a key command into a descriptor

```
u_int32_t *cmd_insert_key(u_int32_t           *descwd,
              u_int8_t                        *key,
              u_int32_t                       keylen,
              enum ref_type                   sgref,
              enum key_dest                   dest,
              enum key_cover                  cover,
              enum item_inline                imm,
              enum item_purpose purpose);
```

Inputs:

- descwd-pointer to target descriptor word to hold this command
- key-pointer to key data as an array of bytes.
- keylen-pointer to key size, expressed in bits.
- sgref-pointer is actual data, or a scatter-gather list representing the key:
  - PTR_DIRECT-points to data
  - PTR_SGLIST-points to SEC4.x-specific scatter gather table. Cannot use if imm = ITEM_INLINE.
- dest-target destination in SEC4.x to receive the key. This may be:
  - KEYDST_KEYREG-Key register in the CHA selected by an OPERATION command.
  - KEYDST_PK_E-The 'e' register in the public key block
  - KEYDST_MD_SPLIT-Message digest IPAD/OPAD direct load.
- cover-Key was encrypted, and must be decrypted during the load. If trusted descriptor, use TDEK, else use JDEK to decrypt.
  - KEY_CLEAR-key is cleartext, no decryption needed
  - KEY_COVERED-key is ciphertext, decrypt.
- imm-Key can either be referenced, or loaded into the descriptor immediately following the command for improved performance.
  - ITEM_REFERENCE-a pointer follows the command.
  - ITEM_INLINE-key data follows the command, padded out to a descriptor word boundary.
- purpose-Sends the key to the class 1 or 2 CHA as selected by an OPERATION command. If dest is KEYDST_PK_E, this must be ITEM_CLASS1.

Returns:


Non-disclosure Agreement Required.

If successful, returns a pointer to the target word incremented past the newly-inserted command (including item pointer or inlined data). Effectively, this becomes a pointer to the next word to receive a new command in this descriptor. If error, returns 0

### 43.9.1.4 cmd_insert_seq_key()

`cmd_insert_key()`: Insert a key command into a descriptor using a sequence

```
u_int32_t *cmd_insert_key(u_int32_t           *descwd,
                          u_int32_t           keylen,
                          enum ref_type       sgref,
                          enum key_dest       dest,
                          enum key_cover      cover,
                          enum item_inline    imm,
                          enum item_purpose purpose);
```

Inputs:

- `descwd`-pointer to target descriptor word to hold this command
- `keylen`-pointer to key size, expressed in bits.
- `sgref`-pointer is actual data, or a scatter-gather list representing the key:
  - `PTR_DIRECT`-points to data
  - `PTR_SGLIST`-points to SEC4.x-specific scatter gather table. Cannot use if `imm` = `ITEM_INLINE`.
- `dest`-target destination in SEC4.x to receive the key. This may be:
  - `KEYDST_KEYREG`-Key register in the CHA selected by an `OPERATION` command.
  - `KEYDST_PK_E`-The 'e' register in the public key block
  - `KEYDST_MD_SPLIT`-Message digest IPAD/OPAD direct load.
- `cover`-Key was encrypted, and must be decrypted during the load. If trusted descriptor, use TDEK, else use JDEK to decrypt.
  - `KEY_CLEAR`-key is cleartext, no decryption needed
  - `KEY_COVERED`-key is ciphertext, decrypt.
- `imm`-Key can either be referenced, or loaded into the descriptor immediately following the command for improved performance.
  - `ITEM_REFERENCE`-a pointer follows the command.
  - `ITEM_INLINE`-key data follows the command, padded out to a descriptor word boundary.
- `purpose`-Sends the key to the class 1 or 2 CHA as selected by an `OPERATION` command. If dest is `KEYDST_PK_E`, this must be `ITEM_CLASS1`.

Returns:

If successful, returns a pointer to the target word incremented past the newly-inserted command (including item pointer or inlined data). Effectively, this becomes a pointer to the next word to receive a new command in this descriptor. If error, returns 0

#### 43.9.1.5 cmd_insert_proto_op_ipsec()

cmd_insert_proto_op_ipsec()-Insert an IPSec protocol operation command into a descriptor.

```
u_int32_t *cmd_insert_proto_op_ipsec(u_int32_t      *descwd,
                                     u_int8_t       cipheralg,
                                     u_int8_t       authalg,
                                     enum protdir   dir);
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instrction. For an OPERATION instruction, this is normally the final word of a single descriptor.
- cipheralg-blockcipher selection for this protocol descriptor. This should be one of CIPHER_TYPE_IPSEC_.
- authalg-authentication selection for this protocol descriptor. This should be one of AUTH_TYPE_IPSEC_.
- dir-Select DIR_ENCAP for encapsulation, or DIR_DECAP for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

#### 43.9.1.6 cmd_insert_proto_op_wimax()

cmd_insert_proto_op_wimax()-Insert an 802.16 WiMAX protocol OPERATION instruction into a descriptor. These can only operate as AES-CCM.

```
u_int32_t *cmd_insert_proto_op_wimax(u_int32_t      *descwd,
                                     u_int8_t       mode,
                                     enum protdir   dir);
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction. For an OPERATION instruction within the scope of a protocol descriptor, this is normally the final word of a single descriptor.
- mode-a nonzero value selects OFDMa, else assume OFDM operation.
- dir-Select DIR_ENCAP for encapsulation, or DIR_DECAP for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

#### 43.9.1.7 cmd_insert_proto_op_wifi()

cmd_insert_proto_op_wifi()-Insert an 802.11 WiFi protocol OPERATION command into a descriptor.

```
u_int32_t *cmd_insert_proto_op_wifi(u_int32_t    *descwd,
                                    enum protdir  dir);
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction. For an OPERATION instruction within the scope of a protocol descriptor, this is normally the final word of a single descriptor.
- dir-Select DIR_ENCAP for encapsulation, or DIR_DECAP for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

### 43.9.1.8 cmd_insert_proto_op_macsec()

cmd_insert_proto_op_macsec()-Insert an MacSec protocol OPERATION instruction into a descriptor.

```
u_int32_t *cmd_insert_proto_op_macsec(u_int32_t    *descwd,
                       enum protdir                dir);
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction. For an OPERATION instruction within the scope of a protocol descriptor, this is normally the final word of a single descriptor.
- dir-Select DIR_ENCAP for encapsulation, or DIR_DECAP for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

### 43.9.1.9 cmd_insert_proto_op_unidir()

cmd_insert_proto_op_unidir()-Insert a unidirectional protocol OPERATION instruction into a descriptor.

```
u_int32_t *cmd_insert_proto_op_unidir(u_int32_t *descwd, u_int32_t protid,
                                      u_int32_t protinfo);
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction. For an OPERATION instruction within the scope of a protocol descriptor, this is normally the final word of a single descriptor.
- protid -Select any PROTID field needed for a unidirectional protocol descriptor from OP_PCLID_.
- dir-Select DIR_ENCAP for encapsulation, or DIR_DECAP for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

### 43.9.1.10 cmd_insert_alg_op()

cmd_insert_alg_op()-Insert a simple algorithm OPERATION instruction into a descriptor.

```
u_int32_t *cmd_insert_alg_op(u_int32_t *descwd, u_int32_t optype,
                             u_int32_t algtype, u_int32_t algmode,
                             enum mdstatesel mdstate, enum icvsel icv,
                             enum algdir dir);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `optype`-specify as class 1 or 2 using `OP_TYPE_CLASSx_ALG`.
- `algtype`-cipher selection, specify one of `ALG_TYPE_`.
- `algmode`-cipher mode selection, specify one of `ALG_MODE_`. Some combinations are ORable depending on application.
- `mdstate`-if a message digest is being processed, selects the processing state. May be one of `MDSTATE_UPDATE`, `MDSTATE_INIT`, `MDSTATE_FINAL`, or `MDSTATE_COMPLETE`.
- `icv`-if processing a message digest, or a cipher with an inclusing authentication function, then `ICV_CHECK_ON` selects an inline signature comparison on the computed result.
- `protid` -Select any PROTID field needed for a unidirectional protocol descriptor from `OP_PCLID_`.
- `dir`-Select `DIR_ENCAP` for encapsulation, or `DIR_DECAP` for decapsulation operations.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

## 43.9.1.11 cmd_insert_pkha_op()

`cmd_insert_pkha_op()`-Insert a PKHA-algorithm OPERATION instruction into a descriptor.

`u_int32_t *cmd_insert_pkha_op(u_int32_t *descwd, u_int32_t pkmode);`

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `pkmode`-mode selection bits, an OR of `OP_ALG_PKMODE_` from one of the 3 possible PKHA sets (clear memory, modular arithmetic, copy memory).

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

## 43.9.1.12 cmd_insert_seq_in_ptr()

`cmd_insert_seq_in_ptr()`: Insert an `SEQ IN PTR` command into a descriptor

```
int *cmd_insert_seq_in_ptr(u_int32_t  *descwd,
                           u_int32_t  *ptr,
                           u_int32_t  len,
                           enum ref_type sgref);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this command. For an `OPERATION` command, this is normally the final word of a single descriptor.
- `ptr`-bus address pointing to the input data buffer
- `len`-input length
- `sgref`-pointer is actual data, or a scatter-gather list representing the key:
  - `PTR_DIRECT`-points to data

- `PTR_SGLIST`-points to SEC4.x-specific scatter gather table.

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

### 43.9.1.13 cmd_insert_seq_out_ptr()

`cmd_insert_seq_out_ptr()`: Insert an `SEQ OUT PTR` command into a descriptor

```
int *cmd_insert_seq_out_ptr(u_int32_t *descwd,
                            u_int32_t *ptr,
                            u_int32_t len,
                            enum ref_type sgref);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this command. For an `OPERATION` command, this is normally the final word of a single descriptor.
- `ptr`-bus address pointing to the output data buffer
- `len`-output length
- `sgref`-pointer is actual data, or a scatter-gather list representing the key:
  - `PTR_DIRECT`-points to data
  - `PTR_SGLIST`-points to SEC4.x-specific scatter gather table.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

### 43.9.1.14 cmd_insert_load()

`cmd_insert_load()`: Insert a `LOAD` instruction into a descriptor:

```
u_int32_t *cmd_insert_load(u_int32_t *descwd, void *data,
                                  u_int32_t class_access, u_int32_t sgflag,
                                  u_int32_t dest, u_int8_t offset,
                                  u_int8_t len, enum item_inline imm)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `data`-pointer to data to be loaded.
- `class_access`
  - `LDST_CLASS_IND_CCB`-access class-independent objects in CCB
  - `LDST_CLASS_1_CCB` -access class 1 objects in CCB
  - `LDST_CLASS_2_CCB` -access class 2 objects in CCB
  - `LDST_CLASS_DECO` -access DECO objects
- `sgflag`-specify `LDST_SGF` if data reference points to a scatter/gather list representing the data.
- `dest` - internal destination for the LOAD. Should be one of `LDST_SRCDST_`.

- `offset` - starting point for writing in the destination.
- `len` - length of data in bytes.
- `imm` - if specified as `ITEM_INLINE`, data is inlined into the descriptor immediately following the LOAD instruction.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

## 43.9.1.15 cmd_insert_seq_load()

`cmd_insert_seq_load()`: Insert a `SEQ LOAD` instruction into a descriptor:

```
        int *cmd_insert_seq_load(u_int32_t *descwd,
unsigned int class_access,
int variable_len_flag,
unsigned char dest,
unsigned char offset,
unsigned char len);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `class_access`
  - `LDST_CLASS_IND_CCB`-access class-independent objects in CCB
  - `LDST_CLASS_1_CCB` -access class 1 objects in CCB
  - `LDST_CLASS_2_CCB` -access class 2 objects in CCB
  - `LDST_CLASS_DECO` -access DECO objects
- `variable_len_flag`-use the variable input sequence length
  - `dest`-destination
  - `offset`-the start point for writing in the destination
  - `len`-length of data in bytes

Returns:

Pointer to next incremental descriptor word past the command just constructed. If an error occurred, returns 0.

## 43.9.1.16 cmd_insert_fifo_load()

`cmd_insert_fifo_load()`: Insert a FIFO LOAD instruction into a descriptor

```
u_int32_t *cmd_insert_fifo_load(u_int32_t *descwd, void *data, u_int32_t len,
                                          u_int32_t class_access, u_int32_t sgflag,
                                          u_int32_t imm, u_int32_t ext, u_int32_t type)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `data` - pointer to data to be loaded.
- `len` - length of load data in bytes.

- class_access
  - LDST_CLASS_IND_CCB-access class-independent objects in CCB
  - LDST_CLASS_1_CCB-access class 1 objects in CCB
  - LDST_CLASS_2_CCB-access class 2 objects in CCB
  - LDST_CLASS_DECO-access DECO objects
- sgflag-data points to a scatter/gather list representing the data to be loaded.
- imm - specify FIFOLDST_IMM if fdata is to be included immediately following this instruction.
- ext - if length needs to be >16 bits, specify FIFOLDST_EXT to inlclude the extended length in a word following the instruction.
- type-FIFO input data type specified as FIFOLD_TYPE_.r

Returns:

Pointer to next incremental descriptor word past the instruction just constructed. If an error occurred, returns 0.

## 43.9.1.17 cmd_insert_seq_fifo_load()

cmd_insert_seq_fifo_load(): Insert a SEQ FIFO LOAD instruction into a descriptor

```
u_int32_t *cmd_insert_seq_fifo_load(u_int32_t *descwd, u_int32_t class_access,
                                     u_int32_t variable_len_flag,
                                     u_int32_t data_type, u_int32_t len)
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction.
- class_access
  - LDST_CLASS_IND_CCB-access class-independent objects in CCB
  - LDST_CLASS_1_CCB-access class 1 objects in CCB
  - LDST_CLASS_2_CCB-access class 2 objects in CCB
  - LDST_CLASS_DECO-access DECO objects
- variable_len_flag-use the variable input sequence length
- data_type-FIFO input data type (FIFOLD_TYPE_* in desc.h)
- len-input data length

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

## 43.9.1.18 cmd_insert_store()

cmd_insert_store(): Insert a STORE instruction into a descriptor

```
u_int32_t *cmd_insert_store(u_int32_t *descwd, void *data,
                             u_int32_t class_access, u_int32_t sg_flag,
                             u_int32_t src, u_int8_t offset,
                             u_int8_t len, enum item_inline imm)
```

Inputs:

- descwd-pointer to target descriptor word intended to hold this instruction.
- data - pointer to the data store location.
- class_access
- LDST_CLASS_IND_CCB-access class-independent objects in CCB
- LDST_CLASS_1_CCB-access class 1 objects in CCB
- LDST_CLASS_2_CCB-access class 2 objects in CCB
- LDST_CLASS_DECO-access DECO objects
- sgflag-if LDST_SGF, the data pointer references a scatter/gather list describing the buffer to receive the stored data.
- src - data souce specification, one of LDST_SRCDST_
- offset-offset into source to begin store operation.
- len-store length in bytes.
- imm - if LDST_IMM, then the data to be stored follows the instruction in the descriptor.

Returns:

1. Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

### 43.9.1.19 cmd_insert_seq_store()

`cmd_insert_seq_store()`: Insert a SEQ STORE instruction into a descriptor

```
u_int32_t *cmd_insert_seq_store(u_int32_t *descwd, u_int32_t class_access,
                                 u_int32_t variable_len_flag, u_int32_t src,
                                 u_int8_t offset, u_int8_t len);
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `class_access`
  - `LDST_CLASS_IND_CCB`-access class-independent objects in CCB
  - `LDST_CLASS_1_CCB`-access class 1 objects in CCB
  - `LDST_CLASS_2_CCB`-access class 2 objects in CCB
  - `LDST_CLASS_DECO`-access DECO objects
- `variable_len_flag`-if `LDST_VLF`, uses the variable sequence output length.
- `src` - data souce specification, one of `LDST_SRCDST_`
- `offset`-offset into source to begin store operation.
- `len`-store length in bytes.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

## 43.9.1.20 cmd_insert_fifo_store()

`cmd_insert_fifo_store()`: Insert a FIFO STORE instruction into a descriptor

```
u_int32_t *cmd_insert_fifo_store(u_int32_t *descwd, void *data, u_int32_t len,
                                           u_int32_t class_access, u_int32_t sgflag,
                                           u_int32_t imm, u_int32_t ext, u_int32_t
type)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `data` - pointer to data to be stored from FIFO.
- `len` - length of data to store.
- `class_access`
  - `LDST_CLASS_IND_CCB`-access class-independent objects in CCB
  - `LDST_CLASS_1_CCB`-access class 1 objects in CCB
  - `LDST_CLASS_2_CCB`-access class 2 objects in CCB
  - `LDST_CLASS_DECO`-access DECO objects
- `sgflag`-if `FIFOLDST_SGF`, data points to a scatter/gather list describing the buffer to be used for the store.
- `imm` - if `FIFOLDST_IMM`, store data is to be inlined into the descriptor itself, immediately following the generated instructuion.
- `ext`-if `FIFOLDST_EXT`, length exceeds 16 bits, and therefore cannot be included in the instruction itself. Write the extended length out to a word following the instruction.
- `type`-FIFO input type, an OR combination of `FIFOST_TYPE_` type and last/flush bits for class1 and 2.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

## 43.9.1.21 cmd_insert_seq_fifo_store()

`cmd_insert_seq_fifo_store()`: Insert a SEQ FIFO STORE instruction into a descriptor

```
u_int32_t *cmd_insert_seq_fifo_store(u_int32_t *descwd, u_int32_t class_access,
                                               u_int32_t variable_len_flag,
                                               u_int32_t out_type, u_int32_t len)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `len` - length of data to store.
- `class_access`
  - `LDST_CLASS_IND_CCB`-access class-independent objects in CCB
  - `LDST_CLASS_1_CCB`-access class 1 objects in CCB
  - `LDST_CLASS_2_CCB`-access class 2 objects in CCB
  - `LDST_CLASS_DECO`-access DECO objects
- `sgflag`-if `FIFOLDST_SGF`, data points to a scatter/gather list describing the buffer to be used for the store.

- `imm` - if `FIFOLDST_IMM`, store data is to be inlined into the descriptor itself, immediately following the generated instructuion.
- `ext`-if `FIFOLDST_EXT`, length exceeds 16 bits, and therefore cannot be included in the instruction itself. Write the extended length out to a word following the instruction.
- `type`-FIFO input type, an OR combination of `FIFOST_TYPE_` type and last/flush bits for class1 and 2.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

## 43.9.1.22 cmd_insert_jump()

`cmd_insert_jump()`: Insert a JUMP instruction into a descriptor

```
u_int32_t *cmd_insert_jump(u_int32_t *descwd, u_int32_t jtype,
                                  u_int32_t class, u_int32_t test, u_int32_t cond,
                                  int8_t offset, u_int32_t *jmpdesc)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `jtype` - type of jump to perform, one of `JUMP_TYPE_`
- `class`
  - `CLASS_NONE` -jump is not a checkpoint
  - `CLASS_1` -jump is a checkpoing for a class 1 operation
  - `CLASS_1` -jump is a checkpoint for a class 2 operation
  - `CLASS_BOTH` -jump is a checkpoint for both classes
- `test` -selects how to assess the conditional test, one of `JUMP_TEST_`
- `cond` - OR combination of conditions to test, based on the test type selected in `test`. May be a combination of `JUMP_COND_`. Note that the JSL bit is factored into the definitions for `JUMP_COND_`, and therefore there are two possible combinational sets.
- `offset` -relative offset of descriptor words to jump to if `JUMP_TYPE_LOCAL` is selected. May be a positive or negative offset.
- `jmpdesc` -address of descriptor to jump to is `JUMP_TYPE_NONLOCAL` is selected.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

## 43.9.1.23 cmd_insert_math()

`cmd_insert_math()`: Insert a MATH instruction into a descriptor

```
u_int32_t *cmd_insert_math(u_int32_t *descwd, u_int32_t func,
                                   u_int32_t src0, u_int32_t src1,
                                   u_int32_t dest, u_int32_t len,
                                   u_int32_t flagupd, u_int32_t stall,
                                   u_int32_t immediate, u_int32_t *data)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.

- `func` - math functuon to perform, one of `MATH_FUN_`.
- `src0` -first source operand, one of `MATH_SRC0_`.
- `src1` - second source operand, one of `MATH_SRC1_`. Note differences between what can be selected between SRC0 and SRC1.
- `dest` - destination operand for the result, one of `MATH_DEST_`.
- `flagupd` - specify `MATH_NFU` if the flags should not be updated.
- `stall` -specify `MATH_STL` to cause the instruction to consume an extra clock cycle.
- `immediate` -specify `MATH_IFB` to use 4 bytes of immediate data when the length needs to remain as 8.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

#### 43.9.1.24 cmd_insert_move()

`cmd_insert_move()`: Insert a MOVE instruction into a descriptor

```
u_int32_t *cmd_insert_move(u_int32_t *descwd, u_int32_t waitcomp,
                                  u_int32_t src, u_int32_t dst, u_int8_t offset,
                                  u_int8_t length)
```

Inputs:

- `descwd`-pointer to target descriptor word intended to hold this instruction.
- `waitcomp` - specify `MOVE_WAITCOMP` if intending to stall execution until the move completes.
- `src` -data source, one of `MOVE_SRC_`.
- `dest` - destination, one of `MOVE_DEST_`.
- `offset` - offset into source for move.
- `length` -length of data to move.

Returns:

Pointer to next incremental descriptor word past the instruction just inserted. If an error occurred, returns 0.

### 43.9.2 Upper-Tier DCL Functions - Descriptor Constructors

A higher level of functionality is provided through complex descriptor constructors. These constructors are single-purpose functions capable of generating complete descriptors from user specifications. These constructors fit into two categories, one for job descriptors and another for shared descriptors generally targeted to protocol processing.

These constructors are by no means the “definitive” reference to all possible descriptor permutations, nor are they meant to work with any specific application. Instead, they are meant to be general-purpose examples of descriptor construction. It is expected that, over time, this library will grow to accommodate a wide range of examples.

All constructor functions are dependent on the underlying command generator.

### 43.9.2.1 Job descriptor constructors

#### 43.9.2.1.1 cnstr_seq_jobdesc()

`cnstr_seq_jobdesc()`: Construct simple sequence job descriptor

```
int cnstr_seq_jobdesc(u_int32_t *jobdesc, unsigned short *jobdescsz,
    u_int32_t *shrdesc, unsigned short shrdescsize,
    unsigned char *inbuf, unsigned long insize,
    unsigned char *outbuf, unsigned long outsize);
```

Inputs:

- `jobdesc`-pointer to buffer in which to build descriptor in
- `jobdescsz`-max size of descriptor build buffer
- `shrdesc`-pointer to associated shared descriptor holding session contex
- `shrdescsz`-size of associated shared descriptor
- `inbuf`-pointer to input frame
- `insize`-size of input frame
- `outbuf`-pointer to output frame
- `outsize`-size of output frame

Constructs a simple job descriptor, emulating QI-level frame processing behavior useful at the job queue level. Besides a target descriptor output, this constructor depends on three references.

1. A pointer to a shared descriptor to do the work. This is normally assumed to be some sort of a protocol-level shared descriptor.
2. A pointer to a packet/frame for input data
3. A pointer to a packet/frame for output data

The constructed descriptor is a simple reverse-order-execution descriptor, and has no provisions for other content specifications.

#### 43.9.2.1.2 cnstr_jobdesc_blkcipher_cbc()

Construct a job descriptor capable of performing a CBC blockcipher operation:

```
int cnstr_jobdesc_blkcipher_cbc(u_int32_t *descbuf, u_int16_t *bufsz,
                                u_int8_t *data_in, u_int8_t *data_out,
                                u_int32_t datasz,
                                u_int8_t *key, u_int32_t keylen,
                                u_int8_t *iv, u_int32_t ivlen,
                                enum algdir dir, u_int32_t cipher,
                                u_int8_t clear);
```

Inputs:

- `descbuf` - Pointer to DMA-able buffer for descriptor construction.

- `bufsz` - Size of constructed descriptor (as output)
- `data_in` - Pointer to input message
- `data_out` - Pointer to output message
- `datasz` - Size of input/output messages
- `key` - Pointer to cipher key
- `keylen` - Size of cipher key
- `iv` - Pointer to cipher IV
- `ivlen` - Size of cipher IV
- `dir` - Direction of cipher operation, select DIR_ENCRYPT or DIR_DECRYPT
- `cipher` - Blockcipher algorithmselection chosen from OP_ALG_ALGSEL_.
- `clear` - Clear descriptor buffer before construction

Returns: -1 on construction error, 0 if construction succeeded.

## 43.9.2.1.3 cnstr_jobdesc_hmac()

Construct a job descriptor capable of performing an HMAC operation:

```
int32_t cnstr_jobdesc_hmac(u_int32_t *descbuf, u_int16_t *bufsize,
                                  u_int8_t *msg, u_int32_t msgsz, u_int8_t *digest,
                                  u_int8_t *key, u_int32_t cipher, u_int8_t *icv,
                                  u_int8_t clear);
```

Inputs:

- `descbuf` - descriptor buffer
- `bufsize` - limit/returned descriptor buffer size
- `msg` - pointer to message being processed
- `msgsz` - size of message in bytes
- `digest` - output buffer for digest (size derived from cipher)
- `key` - key data (size derived from cipher)
- `cipher` - OP_ALG_ALGSEL_MD5/SHA1-512
- `icv` - HMAC comparison for ICV, NULL if no check desired
- `clear` - clear buffer before writing

Returns: -1 on construction error, 0 if construction succeeded.

## 43.9.2.1.4 cnstr_jobdesc_mdsplitkey()

Generate an MDHA "split key" from HMAC key content. A split key is a precomputed IPAD/OPAD pair; MDHA can save cycles during sequential packet processing on a flow by using the precomputed pair directly, and thus saving the pad generation step for each packet.

Generally, the split key is generated at flow setup time as a control-plane activity, thus, this step is performed as a job descriptor.

```
int cnstr_jobdesc_mdsplitkey(u_int32_t *descbuf, u_int16_t *bufsize,
                                u_int8_t *key, u_int32_t cipher,
                                u_int8_t *padbuf);
```

Inputs:

- `descbuf` - pointer to buffer to hold constructed descriptor
- `bufsize` - pointer to size of descriptor once constructed
- `key` - pointer to HMAC key to generate pad pair from. Key size is determined by cipher selection. Note that SHA224/384 pairs are not truncated to the digest size:

Table 85:

| | Key size | Split key size | Buffer size |
|---|---|---|---|
| OP_ALG_ALGSEL_MD5 | 16 | 32 | 32 |
| OP_ALG_ALGSEL_SHA1 | 20 | 40 | 48 |
| OP_ALG_ALGSEL_SHA224 | 28 | 64 | 64 |
| OP_ALG_ALGSEL_SHA256 | 32 | 64 | 64 |
| OP_ALG_ALGSEL_SHA384 | 48 | 128 | 128 |
| OP_ALG_ALGSEL_SHA512 | 64 | 128 | 128 |

- `cipher` - HMAC algorithm selection, one of OP_ALG_ALGSEL_
- `padbuf` - buffer to store generated ipad/opad. Should be 2x the untruncated HMAC keysize for the chosen cipher rounded up to the nearest 16-byte boundary (where 16 bytes = an AES blocksize). See table under "`key`" above.

Returns: -1 on construction error, 0 if construction succeeded.

## 43.9.2.1.5 cnstr_jobdesc_aes_gcm()

Construct a job descriptor capable of performing an AES-GCM operation:

```
int cnstr_jobdesc_aes_gcm(u_int32_t *descbuf, u_int16_t *bufsize,
                               u_int8_t *key, u_int32_t keylen, u_int8_t *ctx,
                               enum mdstatesel mdstate, enum icvsel icv, enum algdir
dir,
                               u_int8_t *in, u_int8_t *out, u_int16_t size, u_int8_t
*mac);
```

Inputs:

- `descbuf` - pointer to buffer that will hold constructed descriptor
- `bufsiz` - pointer to size of descriptor once constructed
- `key` - pointer to AES key
- `keylen` - AES key length

- `ctx` - points to GCM context block. This is a concatenation of: MAC (128 bits), Yi (128 bits), Y0 (128 bits), IV (64 bits), and text bitsize (64 bits). See the AESA section of the blockguide for more information.
- `mdstate` - select `MDSTATE_UPDATE`, `MDSTATE_INIT`, or `MDSTATE_FINAL` if a partial MAC operation is desired, else select `MDSTATE_COMPLETE`.
- `icv` - select `ICV_CHECK_ON` if a MAC compare is requested.
- `dir` - select `DIR_ENCRYPT` or `DIR_DECRYPT` as needed for cipher operation
- `in` - Pointer to input text buffer
- `out` - Pointer to output data text
- `size` - Size of data to be processed
- `mac` - Pointer to output MAC. This can point to the head of context if an updated MAC is required for subsequent operations.

Returns: -1 on construction error, 0 if construction succeeded.

## 43.9.2.1.6 cnstr_jobdesc_kasumi_f8()

Construct a job descriptor capable of performing a Kasumi f8 (confidentiality) operation:

```
int cnstr_jobdesc_kasumi_f8(u_int32_t *descbuf, u_int16_t *bufsz,
                               u_int8_t *key, u_int32_t keylen,
                               enum algdir dir, u_int32_t *ctx,
                               u_int8_t *in, u_int8_t *out, u_int16_t size);
```

Inputs:

- `descbuf` - pointer to buffer that will hold constructed descriptor
- `bufsiz` - pointer to size of descriptor once constructed
- `key` - pointer to KFHA cipher key
- `keylen` - cipher key length
- `dir` - select `DIR_ENCRYPT` or `DIR_DECRYPT` as needed
- `ctx` - points to preformatted f8 context block, containing the 32-bit count (word 0), bearer (word 1 bits 7:16), and cb (word 1 bits 17:31). Refer to the KFHA section of the block guide for more detail.
- `in` - Pointer to input data text
- `out` - Pointer to output data text
- `size` - Size of the data to be processed

Returns: -1 on construction error, 0 if construction succeeded.

## 43.9.2.1.7 cnstr_jobdesc_kasumi_f9()

Construct a job descriptor capable of performing a Kasumi f9 (authentication) operation:

```
int cnstr_jobdesc_kasumi_f9(u_int32_t *descbuf, u_int16_t *bufsz,
                               u_int8_t *key, u_int32_t keylen,
                               enum algdir dir, u_int32_t *ctx,
                               u_int8_t *in, u_int16_t size, u_int_t *mac);
```

Inputs:

- `descbuf` - pointer to buffer that will hold constructed descriptor

- `bufsiz` - pointer to size of descriptor once constructed
- `key` - pointer to cipher key
- `keylen` - size of cipher key
- `dir` - select `DIR_ENCRYPT` or `DIR_DECRYPT` as required
- `ctx` - points to preformatted f8 context block, containing 32-bit count (word 0), bearer (word 1 bits 0:5), direction (word 1 bit 6), ca (word 1 bits 7:16), cb (word 1 bits 17:31), fresh (word 2), and the ICV input (word 3). Refer to the KFHA section of the block guide for more detail
- `out` - pointer to input data
- `out_siz` - size of input data
- `mac` - pointer to output MAC

Returns: -1 on construction error, 0 if construction succeeded.

#### 43.9.2.1.8 cnstr_jobdesc_pkha_rsaexp()

Construct a job descriptor capable of performing an RSA exponentiation operation:

```
int cnstr_jobdesc_pkha_rsaexp(u_int32_t *descbuf, u_int16_t *bufsz,
                                     struct pk_in_params *pkin,
                                     u_int8_t *out, u_int32_t out_siz,
                                     u_int8_t clear);
```

Inputs:

- `descbuf` - pointer to buffer to hold descriptor
- `bufsiz` - pointer to size of written descriptor
- `pkin` - Values of A, B, E, and N
- `out` - Encrypted output
- `out_siz` - size of buffer for encrypted output
- `clear` - nonzero if descriptor buffer space is to be cleared before construction

Returns: -1 on construction error, 0 if construction succeeded.

#### 43.9.2.1.9 cnstr_jobdesc_dsaverify()

Construct a job descriptor capable of performing DSA signature verification:

```
int cnstr_jobdesc_dsaverify(u_int32_t *descbuf, u_int16_t *bufsz,
                                   struct dsa_pdb *dsadata, u_int8_t *msg,
                                   u_int32_t msg_sz, u_int8_t clear);
```

Inputs:

- `descbuf` - pointer to descriptor buffer for construction
- `bufsz` - pointer to size of descriptor constructed (output)
- `dsadata` - pointer to DSA parameters
- `msg` - pointer to input message for verfication
- `msg_sz` - size of message to verify
- `clear` - clear buffer before writing descriptor

Returns: -1 on construction error, 0 if construction succeeded.

### 43.9.2.2 Protocol/shared descriptor constructors

These constructors build a full protocol-level shared descriptor used for semi-autonomous processing of secured traffic through SEC4.x. Such descriptors function as single-pass processors (integrating cipher and authentication functions into a single logical step) with the added factor of performing protocol-level packet manipulation in the same step in the packet-handling process, by maintaining protocol-level connection state information within the descriptor itself.

#### 43.9.2.2.1 cnstr_pcl_shdesc_ipsec_cbc_decap()

Note: this function is deprecated in 2.0, and will be removed in a future release. Use `cnstr_shdsc_ipsec_decap()` instead.

`cnstr_pcl_shdesc_ipsec_cbc_decap()`: Shared protocol-level descriptor for IPSec CBC decapsulation. This function can create a descriptor capable of either tunnel or transport mode processing.

```
int32_t cnstr_pcl_shdsc_ipsec_cbc_decap(u_int32_t           *descbuf,
                                         u_int16_t          *bufsize,
                                         struct pdbcont     *pdb,
                                         struct cipherparams *cipherdata,
                                         struct authparams  *authdata,
                                         u_int8_t            clear);
```

Inputs:

- `descbuf`-Points to a buffer to construct the descriptor in. All SEC4.x descriptors are built of an array of up to sixty-three 32-bit words. If the caller wishes to construct a descriptor directly in the executable buffer, then that buffer must be hardware DMA-able, and physically contiguous.
- `bufsize`-Points to an unsigned 16-bit word with the maximum length of the buffer to hold the descriptor. This will be written back to with the actual size of the descriptor once constructed. (Note: bounds checking not yet implemented).
- `pdb`-Points to a block of data (struct pdbcont) used to describe the content if the Protocol Data Block to be maintained inside the descriptor. PDB content is protocol and mode specific:
  - `pdb.opthdrlen` = Size of inbound header to skip over.
  - `pdb.transmode` = `PDB_TUNNEL/PDB_TRANSPORT` for tunnel or transport handling for the next header.
  - `pdb.pclvers` = `PDB_IPV4/PDB_IPV6` as appropriate for this connection.
  - `pdb.seq.esn` = `PDB_NO_ESN` unless extended sequence numbers are to be supported, then `PDB_INCLUDE_ESN`.
  - `pdb.seq/antirplysz` = `PDB_ANTIRPLY_NONE` if no antireplay window is to be maintained in the PDB. Otherwise may be `PDB_ANTIRPLY_32` for a 32-entry window, or `PDB_ANTIRPLY_64` for a 64-entry window.
- `cipherdata`-Points to a block of data used to describe the cipher information for encryption/decryption of packet content:
  - `algtype`-one of `CIPHER_TYPE_IPSEC_xxx`
  - `key`-pointer to the cipher key data
  - `keydata`-size of the key data in bits
- `authdata`-Points to a block of data used to describe the authentication information for validating the authenticity of the packet source.

    - `algtype`-one of `AUTH_TYPE_IPSEC_xxx`
    - `key`-pointer to the HMAC key data
    - `keydata`-size of the key data in bits
- `clear`-If nonzero, buffer is cleared before writing

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

## 43.9.2.2.2 cnstr_pcl_shdesc_ipsec_cbc_encap()

Note: this function is deprecated in 2.0, and will be removed in a future release. Use `cnstr_shdsc_ipsec_encap()` instead.

`cnstr_pcl_shdesc_ipsec_cbc_encap()`: Shared protocol-level descriptor for IPSec CBC encapsulation. This function can construct a descriptor for either transport or tunnel mode operation.

```
int32_t cnstr_pcl_shdsc_ipsec_cbc_encap(u_int32_t  *descbuf,
                                       u_int16_t           *bufsize,
                                       struct pdbcont      *pdb,
                                       struct cipherparams  *cipherdata,
                                       struct authparams    *authdata,
                                       u_int8_t clear);
```

Inputs:

- `descbuf`-Points to a buffer to construct the descriptor in. All SEC4.x descriptors are built of an array of up to sixty-three 32-bit words. If the caller wishes to construct a descriptor directly in the executable buffer, then that buffer must be hardware DMA-able, and physically contiguous.
- `bufsize`-Points to an unsigned 16-bit word with the maximum length of the buffer to hold the descriptor. This will be written back to with the actual size of the descriptor once constructed. (Note: bounds checking not yet implemented).
- `pdb`-Points to a block of data (struct pdbcont) used to describe the content if the Protocol Data Block to be maintained inside the descriptor. PDB content is protocol and mode specific:
    - `pdbinfo.opthdrlen` = Size of outbound IP header to be prepended to output.
    - `pdbinfo.opthdr` = Pointer to the IP header to be prepended to the output, of size opthdrlen.
    - `pdbinfo.transmode` = `PDB_TUNNEL/PDB_TRANSPORT` for tunnel/transport handling for the next header.
    - `pdbinfo.pclvers` = `PDB_IPV4/PDB_IPV6` as appropriate for this connection.
    - `pdbinfo.seq.esn` = `PDB_NO_ESN` unless extended sequence numbers are to be supported, then `PDB_INCLUDE_ESN`.
    - `pdbinfo.ivsrc` = `PDB_IV_FROM_PDB` if the IV is to be maintained in the PDB, else `PDB_IV_FROM_RNG` if the IV is to be generated internally by SEC4.x's random number generator.
- `cipherdata`-Points to a block of data used to describe the cipher information for encryption/decryption of packet content:
    - `algtype`-one of `CIPHER_TYPE_IPSEC_xxx`
    - `key`-pointer to the cipher key data
    - `keydata`-size of the key data in bits

- authdata-Points to a block of data used to describe the authentication information for validating the authenticity of the packet source.
  - algtype-one of AUTH_TYPE_IPSEC_xxx
  - key-pointer to the HMAC key data
  - keydata-size of the key data in bits
- clear-If nonzero, buffer is cleared before writing

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

### 43.9.2.2.3 cnstr_shdsc_ipsec_encap()

Construct a shared protocol-level descriptor capable of performing IPSec ESP packet encapsulation:

```
int32_t cnstr_shdsc_ipsec_encap(u_int32_t *descbuf, u_int16_t *bufsize,
                                struct ipsec_encap_pdb *pdb, u_int8_t *opthdr,
                                struct cipherparams *cipherdata,
                                struct authparams *authdata);
```

Inputs:

- descbuf - Pointer to buffer used for descriptor construction
- bufsize - Pointer to size to be written back upon completion
- pdb - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the encapsulation PDB, and it's cipher-dependent unioned substructure.
- opthdr - Pointer to the optional header meant to be prepended to an encapsulated frame. Size of the optional header is defined in pdb.opt_hdr_len.
- cipherdata - Pointer to blockcipher transform definitions
- authdata - Pointer to authentication transform definitions. Note that an MDHA split key is to be used with this descriptor (potentially constructed using cnstr_jobdesc_mdsplitkey()), and so the size of the uncovered split key is to be specified here, not the size of the encrypted split key buffer. See the description of cnstr_jobdesc_mdsplitkey() for a detailed discussion of split key lengths versus buffer sizes

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

### 43.9.2.2.4 cnstr_shdsc_ipsec_decap()

Construct a shared protocol-level descriptor capable of performing IPSec ESP packet decapsulation:

```
int32_t cnstr_shdsc_ipsec_decap(u_int32_t *descbuf, u_int16_t *bufsize,
                                struct ipsec_encap_pdb *pdb,
                                struct cipherparams *cipherdata,
                                struct authparams *authdata);
```

Inputs:

- 
  ```
  descbuf
  ```

  - Pointer to buffer used for descriptor construction

- ```
  bufsize
  ```
  - Pointer to size to be written back upon completion
- ```
  pdb
  ```
  - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the decapsulation PDB, and it's cipher-dependent unioned substructure.
- ```
  cipherdata
  ```
  - Pointer to blockcipher transform definitions
- ```
  authdata
  ```
  - Pointer to authentication transform definitions. Note that an MDHA split key is to be used with this descriptor (potentially constructed using
  ```
  cnstr_jobdesc_mdsplitkey()
  ```
  ), and so the size of the uncovered split key is to be specified here, not the size of the encrypted split key buffer. See the description of
  ```
  cnstr_jobdesc_mdsplitkey()
  ```
  for a detailed discussion of split key lengths versus buffer sizes

Returns:

1. -1 if the descriptor creation failed for any reason, zero if creation succeeded.

#### 43.9.2.2.5 cnstr_shdsc_wifi_encap()

Construct a shared protocol-level descriptor capable of performing IEEE 802.11i WiFi packet encapsulation:

```
int32_t cnstr_shdsc_wifi_encap(u_int32_t *descbuf, u_int16_t *bufsize,
                               struct wifi_encap_pdb *pdb,
                               struct cipherparams *cipherdata);
```

Inputs:

- ```
  descbuf
  ```
  - Pointer to buffer used for descriptor construction
- ```
  bufsize
  ```
  - Pointer to size to be written back upon completion
- ```
  pdb
  ```
  - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the encapsulation PDB.
  ```

- ```
  cipherdata
  ```
  - Pointer to blockcipher transform definitions

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

## 43.9.2.2.6 cnstr_shdsc_wifi_decap()

Construct a shared protocol-level descriptor capable of performing IEEE 802.11i WiFi packet decapsulation:

```
int32_t cnstr_shdsc_wifi_decap(u_int32_t *descbuf, u_int16_t *bufsize,
                               struct wifi_decap_pdb *pdb,
                               struct cipherparams *cipherdata);
```

Inputs:

- ```
  descbuf
  ```
  - Pointer to buffer used for descriptor construction
- ```
  bufsize
  ```
  - Pointer to size to be written back upon completion
- ```
  pdb
  ```
  - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the decapsulation PDB.
- ```
  cipherdata
  ```
  - Pointer to blockcipher transform definitions

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

## 43.9.2.2.7 cnstr_shdsc_wimax_encap()

Construct a shared protocol-level descriptor capable of performing IEEE 802.16 WiMAX message encapsulation:

```
int32_t cnstr_shdsc_wimax_encap(u_int32_t *descbuf, u_int16_t *bufsize,
                                struct wimax_encap_pdb *pdb,
                                struct cipherparams *cipherdata,
                                u_int8_t mode);
```

Inputs:

- ```
  descbuf
  ```
  - Pointer to buffer used for descriptor construction
- ```
  bufsize
  ```
  ```

- Pointer to size value to be written back upon completion

- `pdb`

  - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the encapsulation PDB.

- `cipherdata`

  - Pointer to cipher parameters. Only

  `key`

  and

  `keylen`

  are used for this descriptor.

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

#### 43.9.2.2.8 cnstr_shdsc_wimax_decap()

Construct a shared protocol-level descriptor capable of performing IEEE 802.16 WiMAX message decapsulation:

```
int32_t cnstr_shdsc_wimax_decap(u_int32_t *descbuf, u_int16_t *bufsize,
                                struct wimax_decap_pdb *pdb,
                                struct cipherparams *cipherdata,
                                u_int8_t mode);
```

Inputs:

- `descbuf`

  - Pointer to buffer used for descriptor construction

- `bufsize`

  - Pointer to size value to be written back upon completion

- `pdb`

  - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the decapsulation PDB.

- `cipherdata`

  - Pointer to cipher parameters. Only

  `key`

and

```
keylen
```

are used for this descriptor.

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

#### 43.9.2.2.9 cnstr_shdsc_macsec_encap()

Construct a shared protocol-level descriptor capable of performing IEEE 802.1AE MACsec message encapsulation:

```
int32_t cnstr_shdsc_macsec_encap(u_int32_t *descbuf, u_int16_t *bufsize,
                                          struct macsec_encap_pdb *pdb,
                                          struct cipherparams *cipherdata);
```

- ```
  descbuf
  ```
  - Pointer to buffer used for descriptor construction
- ```
  bufsize
  ```
  - Pointer to size value to be written back upon completion
- ```
  pdb
  ```
  - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the encapsulation PDB.
- ```
  cipherdata
  ```
  - Pointer to cipher parameters. Only
  ```
  key
  ```
  and
  ```
  keylen
  ```
  are used for this descriptor.

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

#### 43.9.2.2.10 cnstr_shdsc_macsec_decap()

Construct a shared protocol-level descriptor capable of performing IEEE 802.1AE MACsec message decapsulation:

```
int32_t cnstr_shdsc_macsec_decap(u_int32_t *descbuf, u_int16_t *bufsize,
                                           struct macsec_decap_pdb *pdb,
                                           struct cipherparams *cipherdata);
```

Inputs:

- `descbuf`

  - Pointer to buffer used for descriptor construction
- `bufsize`

  - Pointer to size value to be written back upon completion
- `pdb`

  - Pointer to the PDB to be used with this descriptor. This structure will be copied inline to the descriptor under construction. No error checking of the PDB content shall be made. Refer to the block guide for a detailed discussion of the content of the encapsulation PDB.
- `cipherdata`

  - Pointer to cipher parameters. Only

  `key`

  and

  `keylen`

  are used for this descriptor.

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

## 43.9.2.2.11 cnstr_shdsc_snow_f8()

Construct a shared descriptor capable of performing SNOW UEA2 confidentiality message processing:

```
int32_t cnstr_shdsc_snow_f8(u_int32_t *descbuf, u_int16_t *bufsize,
                                   u_int8_t *key, u_int32_t keylen,
                                   enum algdir dir, u_int32_t count,
                                   u_int8_t bearer, u_int8_t direction);
```

Inputs:

- `descbuf`

  - pointer to descriptor-under-construction buffer
- `bufsize`

  - points to size to be updated at completion
- `key`

  - cipher key
- `keylen`

- size of key in bits

- `dir`

  - cipher direction (DIR_ENCRYPT/DIR_DECRYPT)

- `count`

  - UEA2 count value (32 bits)

- `bearer`

  - UEA2 bearer ID (5 bits)

- `direction`

  - UEA2 direction (1 bit)

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

#### 43.9.2.2.12 cnstr_shdsc_snow_f9()

Construct a shared descriptor capable of performing SNOW UIA2 message authentication:

```
int32_t cnstr_shdsc_snow_f9(u_int32_t *descbuf, u_int16_t *bufsize,
                                   u_int8_t *key, u_int32_t keylen,
                                   enum algdir dir, u_int32_t count,
                                   u_int32_t fresh, u_int8_t direction);
```

Inputs:

- `descbuf`

  - Pointer to buffer for descriptor construction

- `bufsize`

  - Pointer to descriptor size to be updated upon completion

- `key`

  - Cipher key

- `keylen`

  - Size of cipher key

- `dir`

  - Cipher direction (

  `DIR_ENCRYPT/DIR_DECRYPT`

  )

- `count`

  - UEA2 count value (32 bits)
- `fresh`

  - UEA2 fresh value ID (32 bits)
- `direction`

  - UEA2 direction (1 bit)
- `clear`

  - Nonzero if descriptor buffer clear requested

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

## 43.9.2.2.13 cnstr_shdsc_kasumi_f8()

Construct a shared descriptor capable of performing Kasumi f8 confidentiality message processing:

```
int32_t cnstr_shdsc_snow_f8(u_int32_t *descbuf, u_int16_t *bufsize,
                                   u_int8_t *key, u_int32_t keylen,
                                   enum algdir dir, u_int32_t count,
                                   u_int8_t bearer, u_int8_t direction);
```

Inputs:

- `descbuf`

  - pointer to descriptor-under-construction buffer
- `bufsize`

  - points to size to be updated at completion
- `key`

  - cipher key
- `keylen`

  - size of key in bits
- `dir`

  - cipher direction (

  `DIR_ENCRYPT/DIR_DECRYPT`

  )
- `count`

-f8count value (32 bits)

- `bearer`

  - f8 bearer ID (5 bits)

- `direction`

  - f8 direction (1 bit)

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

## 43.9.2.2.14 cnstr_shdsc_kasumi_f9()

Construct a shared descriptor capable of performing Kasumi f9 message authentication:

```
int32_t cnstr_shdsc_snow_f9(u_int32_t *descbuf, u_int16_t *bufsize,
                                   u_int8_t *key, u_int32_t keylen,
                                   enum algdir dir, u_int32_t count,
                                   u_int32_t fresh, u_int8_t direction);
```

Inputs:

- descbuf - Pointer to buffer for descriptor construction
- bufsize - Pointer to descriptor size to be updated upon completion
- key - cipher key
- keylen - size of cipher key
- dir - cipher direction (DIR_ENCRYPT/DIR_DECRYPT)
- count - f9 count value (32 bits)
- fresh - f9 fresh value ID (32 bits)
- direction - f9 direction (1 bit)

Returns:

1. -1 if the descriptor creation failed for any reason, zero if creation succeeded.

## 43.9.2.2.15 cnstr_shdsc_cbc_blkcipher()

Construct a shared descriptor capable of performing CBC blockcipher confidentiality processing:

```
int32_t cnstr_shdsc_cbc_blkcipher(u_int32_t *descbuf, u_int16_t *bufsize,
                                        u_int8_t *key, u_int32_t keylen,
                                        u_int8_t *iv, u_int32_t ivlen,
                                        enum algdir dir, u_int32_t cipher,
                                        u_int8_t clear);
```

Inputs:

- `descbuf`

  - Pointer to buffer for descriptor construction

- `bufsize`

  - Pointer to descriptor size to be updated upon completion
- `key`

  - Pointer to cipher key
- `keylen`

  - Size of cipher key
- `iv`

  - Pointer to IV data
- `ivsize`

  - Size of IV
- `dir`

  - Cipher direction (

  `DIR_ENCRYPT/DIR_DECRYPT)`
- `cipher`

  - Cipher selection (

  `OP_ALG_ALGSEL_AES/DES/3DES)`
- `clear`

  - Nonzero if descriptor buffer clear is requested

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

#### 43.9.2.2.16 cnstr_shdsc_hmac()

Construct a shared descriptor capable of performing hashed message authentication processing:

```
int32_t cnstr_shdsc_hmac(u_int32_t *descbuf, u_int16_t *bufsize,
                               u_int8_t *key, u_int32_t cipher, u_int8_t *icv,
                               u_int8_t clear);
```

Inputs:

- `descbuf`

  - Pointer to buffer for descriptor construction
- `bufsize`

- Pointer to size of descriptor to be updated upon

- `key`

  - Pointer to key data. Note that key length will be automatically selected based on the HMAC cipher chosen.

- `cipher`

  - HMAC cipher selection, one of

```
OP_ALG_ALGSEL_MD5/SHA1/SHA224/SHA256/SHA384/SHA512
```

- `icv`

  - HMAC comparison for ICV, NULL if no check desired

- `clear`

  - Nonzero if descriptor buffer clear is requested

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

## 43.9.2.2.17 cnstr_pcl_shdsc_3gpp_rlc_decap()

Note: this is a P4080 tapeout #1 operation, to be replaced in a future release.

Construct a shared descriptor capable of performing 3GPP RLC message decapsulation operations:

```
int32_t cnstr_pcl_shdsc_3gpp_rlc_decap(u_int32_t *descbuf, u_int16_t *bufsize,
                                       u_int8_t *key, u_int32_t keysz,
                                       u_int32_t count, u_int32_t bearer,
                                       u_int32_t direction,
                                       u_int16_t payload_sz, u_int8_t clear);
```

Inputs:

- `descbuf`

  - Pointer to buffer for descriptor construction

- `bufsize`

  - Pointer to size of descriptor to be updated upon completion

- `key`

  - Pointer to f8 cipher key

- `keysz`

  - Size of cipher key

- `count`

- f8 count value

- `bearer`

  - f8 bearer value

- `direction`

  - f8 direction value

- `payload_sz`

  - Size of payload to be processed (descriptor generated does not use VLF).

- `clear`

  - clear descriptor buffer before construction

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

### 43.9.2.2.18 cnstr_pcl_shdsc_3gpp_rlc_encap()

Note: this is a P4080 tapeout #1 operation, to be replaced in a future release.

Construct a shared descriptor capable of performing 3GPP RLC message encapsulation operations:

```
int32_t cnstr_pcl_shdsc_3gpp_rlc_encap(u_int32_t *descbuf, u_int16_t *bufsize,
                                       u_int8_t *key, u_int32_t keysz,
                                       u_int32_t count, u_int32_t bearer,
                                       u_int32_t direction,
                                       u_int16_t payload_sz);
```

Inputs:

- `descbuf`

  - Pointer to buffer for descriptor construction

- `bufsize`

  - Pointer to size of descriptor to be updated upon completion

- `key`

  - Pointer to f8 cipher key

- `keysz`

  - Size of cipher key

- `count`

  - f8 count value

- `bearer`

- f8 bearer value

- ```
  direction
  ```

  - f8 direction value

- ```
  payload_sz
  ```

  - Size of payload to be processed (descriptor generated does not use VLF).

Returns:

-1 if the descriptor creation failed for any reason, zero if creation succeeded.

## 43.9.3 Disassembler

### 43.9.3.1 caam_desc_disasm()

```
caam_desc_disasm()
```

-Top-level descriptor disassembler

```
        void caam_desc_disasm(u_int32_t *desc, u_int32_t opts);
```

Inputs:

- ```
  desc
  ```

  -points to the descriptor to disassemble. First command must be a header, or shared header, and the overall size to disassemble is determined by the header. Does not handle a QI preheader as its first command, and cannot yet follow links in a list of descriptors.

- ```
  opts
  ```

  - selects options to add to the disassembled output:

  ```
  DISASM_SHOW_OFFSETS
  ```

  - shows the index/offset of each instruction in the descriptor preceding the textual disassembly. This is useful for visualizing flow control changes in a descriptor, since any offset to a specific instruction in the disassembly will be displayed both as a relative number of instructions, and as the offset of the specific instruction.

  ```
  DISASM_SHOW_RAW
  ```

  - shows the hexadecimal value of each instruction before the displayed value of the instruction itself.

Runtime Assembler Library Reference

# Chapter 44
# Runtime Assembler Library Reference

## 44.1 Runtime Assembler Library Reference

Use the Runtime Assembler Library to write SEC descriptorsThis reference describes the structure, concept, functionality, and high level API.

Runtime Assembler Library Reference

# Chapter 45
# USDPAA IPFwd Longest Prefix Match User Manual

## 45.1 Freescale P4080/P5020/P3041 USDPAA IPFwd Longest Prefix Match User Manual

### 45.1.1 Introduction

This user manual describes USDPAA IPFwd based upon Longest Prefix Match methodology. This IPFwd application is different from the other route cache based IPFwd. The User Space Datapath Acceleration Architecture (USDPAA) is a software framework that permits Linux user space applications to directly access DPAA queue and buffer manager portals in a high performance manner. The applications can then use the portals to access other DPAA hardware components such as the security coprocessor and the frame manager.

This document provides the following:

- A summary of the USDPAA LPM IPFwd application
- A summary of usage of Shared MAC
- A summary of usage of MAC-less interface
- Execution steps for USDPAA LPM IPFwd application from the Freescale SDK package on the P4080 DS/ P3041DS/P5020DS

This document describes the USDPAA application which demonstrates:

1. IP Forwarding application using Longest prefix match as route decision algorithm
2. MAC less communication between USDPAA application and kernel
3. Sharing of same physical Ethernet port using shared-mac mode

### 45.1.2 Overview

The USDPAA LPM based IPv4 forwarding application is a multi-threaded application that routes IPv4 packets from one ethernet interface to another on all QorIQ platforms. The LPM routing algorithm uses the prefix for destination ip address to do the route look up. Any combination of the cores can run a LPM based USDPAA IPv4 Forwarding application thread. The packets that reach the USDPAA application can be forwarded to destination IP address using LPM route algorithm.

LPM IPFwd packet-processing:

· Receives ethernet frames on ethernet interfaces.

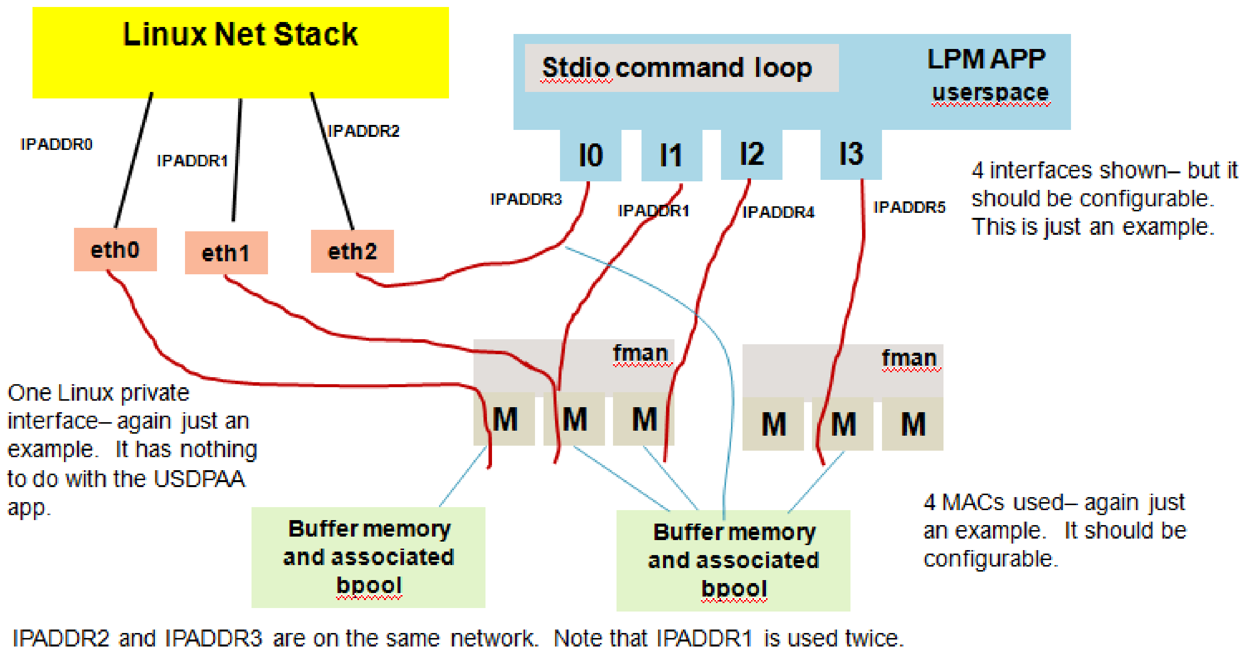
· On the basis of defined PCD rules in FMAN’s PCD table, IPv4 traffic would be sent to USDPAA application through PCD Frame queue range.

· For IPv4 frames, processing takes place as defined in section Overview of packet flow: on page 768

This application also demonstrates the following features:

. how the traffic coming from a common MAC port can be split in between kernel and USDPAA application on the basis of defined PCD rules. The packets that reach the USDPAA application can be forwarded to destination IP address using LPM route algorithm.

. ping between linux and USDPAA using MAC-less interface.



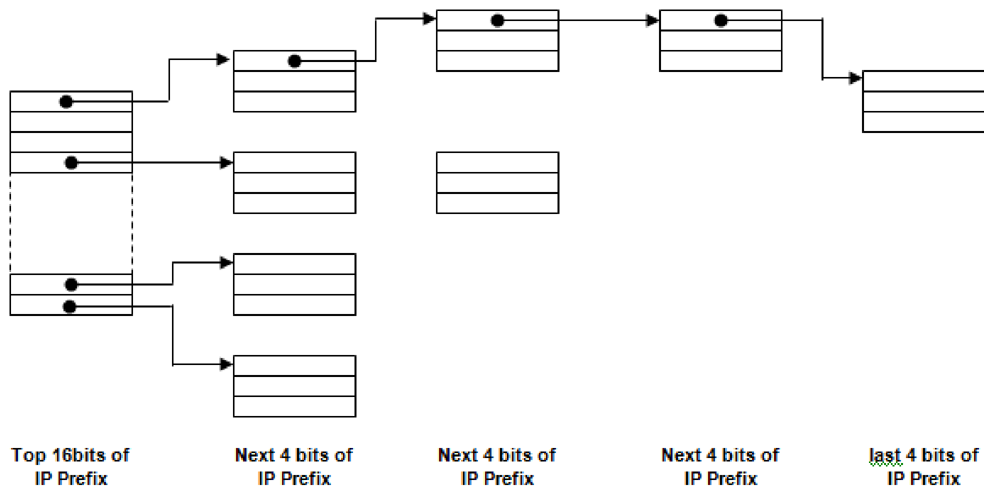

## 45.1.3 How is it different from existing Route cache based IPFwd?

This USDPAA application uses Longest Prefix Match algorithm for doing route lookup by using prefix for destination IP address in contrast to the existing route cache based IPFwd which takes route decision based upon hash results calculated by FMAN using source IP address and destination IP address in the frame.

The user will have to use new commands for route addition in LPM table (check the command section)

## 45.1.4 Longest Prefix Match algorithm

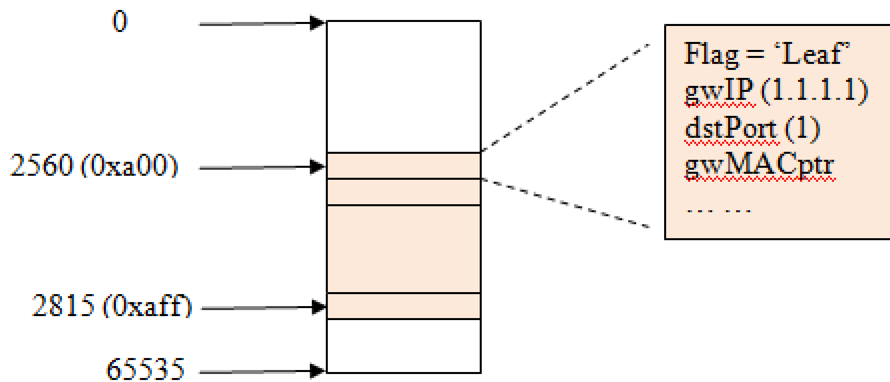
Instead of using traditional Radix-Trie algorithm, here we choose to use one simpler LPM algorithm. It uses 5 level tables: First level table – 65536 entries array, indexed by the top 16 bits of IP address. Second level table – 32 entries array, indexed by bit12~15 of IP address. Third level table – 32 entries array, indexed by bit8~11 of IP address. Fourth level table – 32 entries array, indexed by bit4~7 of IP address. Fifth level table – 32 entries array, indexed by bit0~3 of IP address. The 2nd level to 5th level tables are only created when its first level table entry has valid value. See below figure:




At init time, only the first level (i.e. top16 bits) array is created which contains 65536 null entries. While adding route entries to the FIB table, the 2nd level to 5th level arrays will be created accordingly. This is a typical ASIC design algorithm of LPM which is fast and simple to search while costs far more memory. The worst case is to index and compare 5 times when searching an IP address, but it's still fast enough.

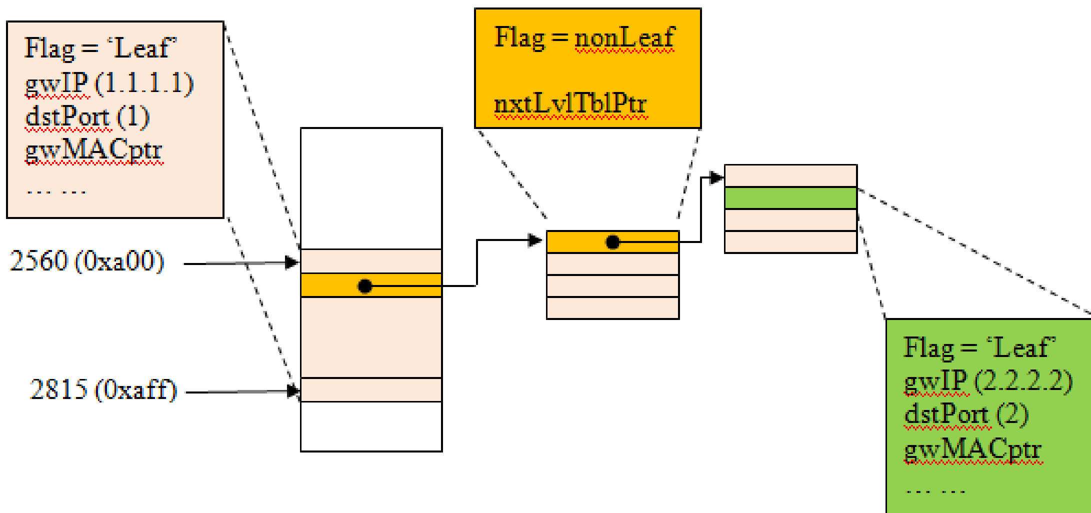
Examples:

1) Add one ClassA route 10.0.0.0/8 to the route table. (gateway is 1.1.1.1, destination port is 1) The first 16bit of 10.0.0.0 (0x0a000000) is 0x0a00 (2560). And the mask is 8 bit which is smaller than the 1st level bit-length (16b), so below entries (from 0x0a00 to 0x0aff) will be created in the FIB table:




From entry No.2560 to No.2815 (total 255 entries) are filled with same content (flag, gwIP, dstPort, ptr …). Now if a packet with DIP of 10.1.1.1 comes in, its first 16 bit value is 0x0a01 (2561). So, the No.2561 entry of the 1st level table will be checked. If it's a 'leaf' node (now it is), then the best-match is found. And the packet will be forwarded to the 'dstPort' after replacing the SMAC and DMAC. And, any DIP of 10.x.x.x will all be forwarded to port 1 with gwIP 1.1.1.1 based on above table.

2) Now, a new route 10.1.1.0/24 is added to the FIB table. The table will be like this:




The new route will overwrite the No.2561 (0x0a01) entry with 'flag' of 'nonLeaf' and a pointer to 'next-level-table'. Now a 16-entry-block memory will be allocated as the 'next-level-table' because the 2nd level is 4bit indexed. And the base address of this new block will be set to the 'nxtLvlTblPtr' of No.2561. Because the next 4 bit of the new route is 0 (bit16 to bit19 of 0x0a010100), so the 1st entry in the 2nd level table is used as another 'nonLeaf' entry. While all the other entries in the 2nd level table should be filled with the same value of its 'parent' route (10.0.0.0/8). The netmask is 24bit which is larger than 16+4, so the 3rd level table should also be allocated (16 entries). And the next 4bit of the new route is 1 (bit20 to bit23 of 0x0a010100), so the 2nd entry will be used for the new route. And because the netmask (24bit) is no-larger-than 16+4+4, so this entry will be the 'Leaf' entry of this new route (see above figure in green). And the according values (gwIP, dstPort, etc.) will be filled in that entry. Now a frame with DIP of 10.1.1.100 comes. There will have 3 lookups to get the final result:

• Index with first 16bit of DIP, whose value is 0x0a01. 'Non-leaf' means to continue the next-level lookup.

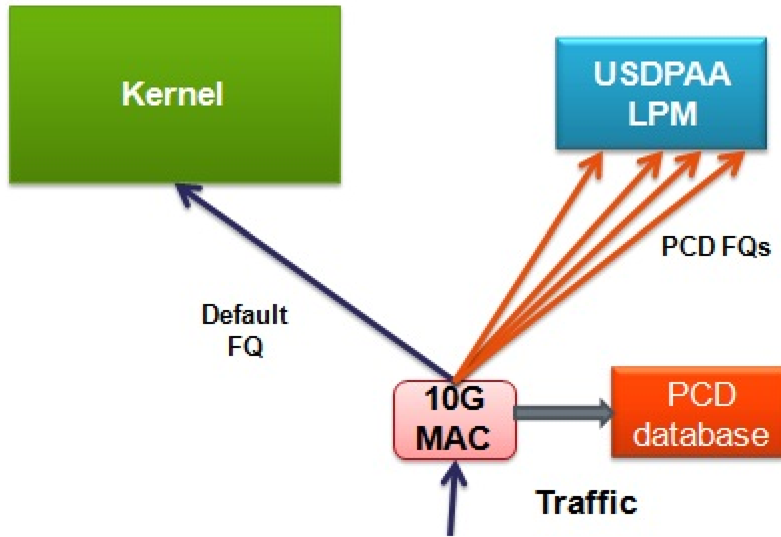
• Index with the next 4bit of DIP, whose value is 0. Then 'Non-leaf' again.

• Index with the next 4bit of DIP, whose value is 1. Then the 'leaf' node is found and the lookup reaches an end.

Now a frame with DIP of 10.1.192.10 comes. You can see it will find the 'leaf' node in the 2nd level table and get the route of net-address 10.0.0.0/8. And a frame with DIP of 10.1.10.10 will find its 'leaf' node in the 3rd level table and also get the route of net-address 10.0.0.0/8 as we expected. The multi-branch trie algorithm provides a very fast way of route-lookup but a relatively complicated way of route-add/deletion.

## 45.1.5 Shared MAC Overview

The kernel and USDPAA should be able to receive traffic of their interest from a shared Ethernet port. On the basis of defined PCD rules in FMAN's PCD table, IPv4 traffic ( for non-owned IP addresses) would be sent to USDPAA application through PCD Frame queue range and rest of the traffic ( i.e. ICMP etc) would be sent to kernel through default Frame queue.




## 45.1.6 How to run shared MAC interface ?

1. Make sure to use the corresponding dtb to "p4080ds-usdpaa-shared-interfaces.dts" file from kernel source. In this example dts file ethernet@9 depicts shared MAC node.

   **NOTE**
   For p3-p5, use p3041ds-usdpaa-shared-interfaces.dts and p5020ds-usdpaa-shared-interfaces.dts respectively. ethernet@5 depicts shared MAC node.

   **NOTE**
   For B4, use b4860qds-usdpaa-shared-interfaces.dts where ethernet@9 depicts shared MAC node. And for T4, use t4240qds-usdpaa-shared-interfaces.dts where ethernet@15 depicts shared MAC node.

2. When linux comes up, check the output of "ifconfig -a".

   root@p4080ds:~# ifconfig -a

   fm2-10g Link encap:Ethernet HWaddr 00:e0:0c:00:96:09

   inet6 addr: fe80::2e0:cff:fe00:9609/64 Scope:Link

   UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

   RX packets:0 errors:0 dropped:0 overruns:0 frame:0

   TX packets:5 errors:1 dropped:0 overruns:0 carrier:0

   collisions:0 txqueuelen:1000

   RX bytes:0 (0.0 B) TX bytes:378 (378.0 B)

   Memory:fe5f0000-fe5f0fff

3. Run fmc

   $cd /usr/etc

   fmc -c usdpaa_config_p4_serdes_0xe.xml -p usdpaa_policy_hash_shared_mac_ipv4.xml -a

   For p3-p5 use this command:

fmc -c usdpaa_config_p3_p5_serdes_0x36_shared_mac.xml -p usdpaa_policy_hash_shared_mac_ipv4.xml -a

For B4 use this command:

fmc -c usdpaa_config_b4_serdes_0x2a_0x98.xml –p usdpaa_policy_hash_shared_mac_ipv4.xml –a

For T4 use this command:

fmc -c usdpaa_config_t4_serdes_1_1_6_6.xml –p usdpaa_policy_hash_shared_mac_ipv4.xml –a

4. Run lpm-ipfwd application

   $lpm_ipfwd_app -d 0x10000000 -b 1600:1600:1600 -i fm2-10g

   For p3-p5

   $lpm_ipfwd_app -d 0x10000000 -b 1600:1600:1600 -i fm1-10g

   For B4

   $lpm_ipfwd_app -d 0x10000000 -b 1600:1600:1600 –c usdpaa_config_b4_serdes_0x2a_0x98.xml –p usdpaa_policy_hash_shared_mac_ipv4.xml -i fm1-mac10

   For T4

   $lpm_ipfwd_app -d 0x10000000 -b 1600:1600:1600 –c usdpaa_config_t4_serdes_1_1_6_6.xml –p usdpaa_policy_hash_shared_mac_ipv4.xml -i fm2-mac10

5. Assign ip address to USDPAA side as well as kernel side as per script "usdpaa_policy_hash_shared_mac_ipv4.xml".

   In this script, the coarse classification "ip_dest_clsf" corresponding to this port uses "192.168.44.3" for USDPAA and "192.168.44.4" for kernel.

   **NOTE**

   Check ip address for p3-p5 as per scripts usdpaa_config_p3_p5_serdes_0x36_shared_mac.xml and usdpaa_policy_hash_shared_mac_ipv4.xml.

   **NOTE**

   For B4/T4, use “192.168.44.3” for USDPAA and “192.168.44.4” for kernel as defined in usdpaa_policy_hash_shared_mac_ipv4.xml.

   ssh to p4080 and give these commands:

   $lpm_ipfwd_config -E -a true

   output:

   FMAN Interface number: 11

   , PortID=1:5 is FMan interface node with MAC Address 00:e0:0c:00:96:09

   **NOTE**

   Check pid from application print "Message queue to send: /mq_snd_2536"

   $ lpm_ipfwd_config -P 2536 -F -a 192.168.44.1 -i 11

   $ lpm_ipfwd_config -P 2536 -G -s 192.168.44.3 -m 02:00:c0:a8:a0:02 -r true

   $ lpm_ipfwd_config -P 2536 -B -c 1 -d 192.168.44.3 -n 16 -g 192.168.44.3

   $ifconfig fm2-10g 192.168.44.4

6. Now run traffic on fm2-10g using the above ip addresses and can see traffic splitting amongst kernel and USDPAA.

## 45.1.7 MAC-less use case

This section describes how MAC-less interface is being used in this application. The user space configuration commands and USDPAA application can communicate with each other through the management interface which can be a MAC interface (SGMII, RGMII etc) or MAC-less interface. In this application we are just using MAC-less interface. The user can do such a communication with USDPAA LPM application by using lpm_ipfwd_config binary. For command reference, please check section IPv4 forward application Configuration command on page 666.

## 45.1.8 How to ping MAC-less interface ?

. Make sure to use the corresponding dtb to "p4080ds-usdpaa-shared-interfaces.dts" file from kernel source. In this example dts file ethernet@10 depicts MAC-less node.

. NOTE: For B4, use b4860qds-usdpaa-shared-interfaces.dts and for T4, use t4240qds-usdpaa-shared-interfaces.dts. In both cases, ethernet@16 is the macless node.

. When linux comes up, check the output of "ifconfig -a". The interface containing similar mac address as given under ethernet@10 node in device tree, is the MAC-less interface.

root@p4080ds:~# ifconfig -a

eth0 Link encap:Ethernet HWaddr 00:15:17:1e:22:9e

UP BROADCAST MULTICAST MTU:1500 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

Interrupt:42 Memory:40000000-40020000

eth3 Link encap:Ethernet HWaddr 00:11:22:33:44:55

BROADCAST MULTICAST MTU:1500 Metric:1

RX packets:0 errors:0 dropped:0 overruns:0 frame:0

TX packets:0 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:1000

RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

The above output shows that "eth3" is the MAC-less node.

. Run lpm-ipfwd application

$cd /usr/etc

$lpm_ipfwd_app -d 0x10000000 -b 1600:1600:1600 -i eth3:[66-22-33-44-55-66]

The mac address "66-22-33-44-55-66" is the mac that will be assigned as source mac address to USDPAA side of MAC-less interface

. Assign ip address to USDPAA side as well as kernel side.

ssh to p4080 and give these commands:

$lpm_ipfwd_config -E -a true

Output:

MACLESS Interface:

name : eth3

$lpm_ipfwd_config -F -a 192.168.55.6 -n eth3

$ifconfig eth3 192.168.55.2

$ping 192.168.55.6

PING 192.168.55.6 (192.168.55.6): 56 data bytes

64 bytes from 192.168.55.6: icmp_seq=0 ttl=64 time=12.413 ms

64 bytes from 192.168.55.6: icmp_seq=1 ttl=64 time=12.431 ms

## 45.1.9 USDPAA LPM based IPv4 forwarding application flow

The LPM based IPFwd application has two main phases. There is an initial configuration phase and a subsequent packet processing phase.

The configuration phase executes when the application starts. Application threads are created and global initialization of resources is done which is the part of PPAC (see USDPAA PPAC User Guide for more details). The configuration phase also includes PPAM (i.e. IPFwd) related initialization. Once the configuration phase is completed the IPFwd application moves to the packet processing phase. This application provides a command-line interface to enable users to add and remove routing table and ARP cache entries. For each user input, the appropriate information is communicated to the IPFwd application via standard Posix IPC. Messages are placed onto the message queue till they are received by the IPfwd application. Note that this application does not dynamically resolve ARP – missing ARP entries will result in the application dropping the packet.

In the packet processing phase, the loop migrates from polling mode to a blocking “IRQ mode” whenever LPM IPFwd has looped a certain number of times without any forward progress. For more information on IRQ mode please refer to “USDPAA PPAC User Guide”. In polling mode – the application constantly looks for data to process on its dedicated QMan portal . Network traffic is classified and distributed by the FMan to frame queues based on source and destination IP address in the packet. There is an associated handler that processes the packets arriving on each frame queue.

## 45.1.10 Overview of packet flow:

1. Static Route table entries are populated using the user space configuration commands.
2. If a packet received by the FMan is an IPv4 packet it uses 2-tuple (Src IP & Dest IP) to hash the packet to a Rx frame queue. Otherwise if packet is ICMP etc it would be sent through default Frame queue where the buffer is freed.
3. The packet enqueued to PCD FQ to reach USDPAA LPM based IPFwd application thread running on one of the cores is subjected to LPM based route lookup.
4. All of the threads enter the processing loop where they call Qman_poll till a frame is received.
5. If a frame is received, the application checks whether the destination IP address exist in the FIB table. For this it calls ip_route_lookup(), which does the route look up using LPM algorithm. The LPM IPFWD application does not change its FIB table in response to seeing the first packet in a flow. Instead the FIB table is set only by commands, as mentioned in section Syntax on page 666.
6. If the route entry for this frame is not present in the FIB table, the frame is dropped. It then continues with Qman_poll.
7. If the route entry for this frame exists in the FIB table, the frame is sent for forwarding.

8. If the frame is to be forwarded, it is checked if ARP entry exists in ARP table for destination IP address. LPM IPFWD application will not dynamically resolve the ARP. So if sending packets to forward using a regular computer, the user will have to create static ARP entries. On host computer, ARP table can be updated by sending the ARP request. LPM IPFWD will respond to external ARP requests
9. If ARP entry exists, TTL is decremented in L3 header.
10. Finally, the L2 header is updated, which includes changing the dst MAC address in L2 header. The frame is then enqueued to the TX FQ.

## 45.1.11 Overview of PPAC

The source code to LPM-IPFwd has been reorganized into two parts; the “PPAC” (Packet-Processing Application Core) and a “PPAM” (Packet-Processing Application Module). The PPAM portion implements the LPM-IPFWD application specific logic of processing the packet using longest prefix match and forward it. On the other hand, the PPAC component represents the common infrastructure to support PPAM; initializing devices, handling flow-control, implementing a CLI (Command-Line Interface), managing threads and buffers. PPAC details can be found in the document “USDPAA PPAC User Guide”.

## 45.1.12 Compile-time configuration

PPAC-based application are compiled using a certain set of options that are currently defined in the header located at apps/include/ppac.h. The following describes the most useful options for modification if alternative application behaviour is desired.

## 45.1.13 Order Preservation in LPM-IPFWD

This section describes how user can enable Order Preservation in LPM-IPFWD application. By default Order Preservation is disabled in LPM-IPFWD application and in order to enable it the user will have to re-compile the binary by making following changes to the source code.

In file, usdpaa/apps/include/ppac.h you can find these two lines.

/* Application options */

#undef PPAC_2FWD_HOLDACTIVE /* Process each FQ on one portal at a time */

#undef PPAC_2FWD_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */

Change the above to

#define PPAC_HOLDACTIVE /* Process each FQ on one portal at a time */

#define PPAC_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */

And then compile usdpaa once again. Now run the LPM-IPFWD application with Order Preservation.

## 45.1.14 Order Restoration in LPM IPFWD

Order restoration is the functionality of QMan software portal interface which restores the relative temporal order of a flow of frames (sequence of frames) to that observed before transmitting to the destination Frame Queue and Order Definition Point (ODP) takes note of the correct order of packets before start processing by using the sequence number. Use of “HOLDACTIVE” is mutually exclusive with another QMan option “AVOIDBLOCK”, which is selected by default in PPAC. To enable order-restoration, the user will have to re-compile the binary by making following changes to the source code.

/* Application options */

#undef PPAC_2FWD_HOLDACTIVE /* Process each FQ on one portal at a time */

#undef PPAC_2FWD_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */

#undef PPAC_2FWD_ORDER_RESTORATION /* Use ORP */

#define PPAC_2FWD_AVOIDBLOCK /* No full-DQRR blocking of FQs */

Change the above to

#undef PPAC_HOLDACTIVE /* Process each FQ on one portal at a time */

#undef PPAC_ORDER_PRESERVATION /* HOLDACTIVE + enqueue-DCAs */

#define PPAC_ORDER_RESTORATION /* Use ORP */

#define PPAC_AVOIDBLOCK /* No full-DQRR blocking of FQs */

And then compile usdpaa once again. Now run the LPM-IPFWD application with Order Restoration. Implementation note: Order restoration has been implemented such that each PCD Frame queue has a corresponding ORP (order restoration point) frame queue associated with it. Each ORP is configured with default window settings as seen below

#define PPAC_ORP_WINDOW_SIZE 7 /* 0->32, 1->64, 2->128, ... 7->4096 */

#define PPAC_ORP_AUTO_ADVANCE 1 /* boolean */

#define PPAC_ORP_ACCEPT_LATE 3 /* 0->no, 3->yes (for 1 & 2->see RM) */

Here the ORP window size is set to be 4K, auto advance window size as 4K and accept late arrival window size as 8K. This ensures that no traffic is getting dropped but are always accepted below and at Zero loss throughput. Beyond zero loss throughput, as usual packets would be dropped and thus you can see mis-ordering.

ORP FQ descriptor attributes settings:

· Prefer in cache

· No “ HOLDACTIVE ”

· No “ AVOIDBLOC K”

· ORP enabled

Assumption: To see the effect of Order Restoration in LPM-IPFwd application the user must use separate streamblocks as a source of traffic. If not done so, mis-ordering would be seen.

Key observation: It has been observed in LPM-IPFwd application that use of “HOLDACTIVE” with traffic generated using separate streamblocks, all the packets are IN sequence. Therefore, it is recommended that if user wants to see the real effect of Order restoration in LPM-IPFwd application he should use “AVOIDBLOCK” with “RESTORATION” and not “HOLDACTIVE” with “RESTORATION”

## 45.1.15 Monitoring Rx/Tx fill-levels and flow-control via CGR

If CGR-based monitoring is enabled, then two Congestion Group Records will be configured, with all Rx FQs for all interfaces being subscribed to one, and all Tx FQs being subscribed to the other. This simply allows the user to monitor the overall fill-level of frame queues in the system, in particular to determine whether build-up is occuring before or after the software-processing phase. Refer to section “ Monitoring Rx/Tx fill-levels via CGR ” of USDPAA PPAC User Guide for more details. To enable this feature, in ppac.h change;

#undef PPAC_CGR

to;

#define PPAC_CGR

The CGRs can also be configured to perform flow-control using Congestion state tail drop by setting CSTD_EN bits . Each congestion group record can be configured to track either byte counts or frame counts in all frame

queues in the Congestion Group. When the threshold set for each CGR is exceeded, the CS bit is set in the CGR, and the congestion group is said to have entered congestion. At this point the incoming frames are marked for discard and QMAN will generate enqueue rejections to the producer. When the group's I_BCNT returns below the threshold (minus approximately 1/8 of the threshold to provide hysteresis), the CS bit is cleared, and the congestion group's state exits congestion. To enable tail drop, in ppac.h change;

```
#undef PPAC_CGR            /* Track rx and tx fill-levels via CGR */
#undef PPAC_CSTD           /* CGR tail-drop */
#undef PPAC_CSCN           /* Log CGR state-change notifications */
```

to

```
#define PPAC_CGR           /* Track rx and tx fill-levels via CGR */
#define PPAC_CSTD          /* CGR tail-drop */
#undef PPAC_CSCN           /* Log CGR state-change notifications */
```

And then compile usdpaa once again. Now run the IPFWD application with CGR tail drop enabled. To test this feature PPAC CLI provides a command "cgr" which will query and display all the fields of both CGRs. On pumping the traffic to IPFWD application at full line rate, t he instantaneous group byte count value I_BCNT(Instantaneous frame/byte count) must be maintained lesser than the CGR threshold set for each congestion group. Here is one such cgr command output:

```
> cgr
Rx CGR ID: 10, selected fields;
cscn_en: 0
cscn_targ: 0x00800000
cstd_en: 1
cs: 0
cs_thresh: 0x00_0000_1000
mode: 1
i_bcnt: 0x00_0000_0e1e
a_bcnt: 0x00_0000_0e1e
Tx CGR ID: 11, selected fields;
cscn_en: 0
cscn_targ: 0x00800000
cstd_en: 1
cs: 0
cs_thresh: 0x00_0000_0200
 mode: 1
i_bcnt: 0x00_0000_0002
a_bcnt: 0x00_0000_0004
```

On the other hand if this feature of Congestion Group tail drop is disabled in IPFWD application I_BCNT is never maintained below CGR threshold value with traffic at full line-rate. This can be checked by compiling the IPFWD application with;

```
#define PPAC_CGR           /* Track rx and tx fill-levels via CGR */
```

#undef PPAC_CSTD /* CGR tail-drop */

#undef PPAC_CSCN /* Log CGR state-change notifications */

Now on pumping traffic at full line-rate, atleast one of the CGRs must go into congestion state and its I_BCNT should be above CGR threshold value. Here is the sample output:

> cgr

Rx CGR ID: 10, selected fields;

cscn_en: 1

cscn_targ: 0x00800000

cstd_en: 0

cs: 0

cs_thresh: 0x00_0000_1000

mode: 1

i_bcnt: 0x00_0000_0006

a_bcnt: 0x00_0000_0004

Tx CGR ID: 11, selected fields;

cscn_en: 1

cscn_targ: 0x00800000

cstd_en: 0

cs: 1

cs_thresh: 0x00_0000_0200

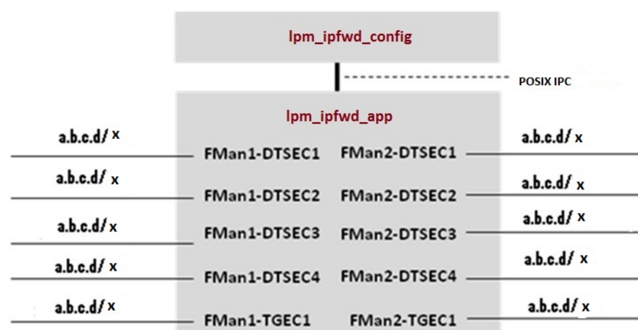
mode: 1

i_bcnt: 0x00_0000_5fb7

a_bcnt: 0x00_0000_5fb8

Thus, the above test showcases the flow control achieved by enabling congestion group tail drop in IPFWD application.

## 45.1.16 LPM IPFWD Application Suite




The figure above shows the structure of the LPM IPFWD USDPAA application suite. Its purpose is to forward IPv4 packets between the interfaces shown. No IP address is assigned to any of the interfaces by default. Using ipfwd_config command as mentioned in section Assign IP address to interfaces on page 787 IP address can

be assigned to all these interfaces. Each interface can have a variable netmask. The MAC addresses of these interfaces are determined by u-boot environment variables ethaddr, eth1addr, eth2addr, etc.

The ipfwd application will respond to ARP requests on these interfaces. However, the application will not generate ARP requests to determine the destination MAC address of forwarded packets. An ipfwd_config command should be used to set these MAC addresses. On host side, ARP table can be updated by sending ARP requests to this application.

It is important to understand that the application can use only a subset of these interfaces at any one time. This is due to pin multiplexing rules on the P4080 SoC. The set of available interfaces is determined by a number of factors:

1. The interface set made available by the selected SerDes Protocol and u-boot variable "hwconfig". See the SDK document "Freescale DPAA SDK X.Y: Selecting Ethernet Interfaces". This document is distributed with the DPAA SDK.
2. The Linux device tree determines which subset of the available interfaces is for use by USDPAA applications. See the USDPAA User Guide for more information.
3. Finally, the fmc configuration file passed by command line argument to lpm_ipfwd_app determines the subset of these interfaces that the application will attempt to initialize.

In the default SerDes 0xe example, the interfaces used by lpm_ipfwd_app are:

- FMan1-TGEC1
- FMan2-DTSEC3
- FMan2-DTSEC4
- FMan2-TGEC1

Running the lpm ipfwd application suite involves four steps:

1. Run lpm_ipfwd_app
2. Run lpm_ipfwd_config repeatedly to add routes to LPM FIB table.
3. Run the fmc application to configure the FMan hardware instances.

Specific examples showing these steps are provided in other sections of this document.

## 45.1.17 Possible configuration scenario for LPM based IPFWD

LPM based IPfwd application can run in different configuration scenario. The table below shows the configuration files and corresponding sample shell script existing in the repository.

| xml file | Sample shell script | Number of routes (as per sample shell script) | Netmask (as per sample shell script) |
|---|---|---|---|
| usdpaa_config_p4_serdes_0xe.xml (2x10G+2x1G) | lpm_ipfwd_20G.sh | 1024 routes (2x10G) | 16 |
| | lpm_ipfwd_22G.sh | 1024 routes (2x10G +2x1G) | 16 |
| | lpm_ipfwd_20G_1Mroutes.sh | 1M/256 (4K) routes | 24 |
| usdpaa_config_p2_p3_p5_14g.xml(4x1G+10G) | lpm_ipfwd_14G.sh | 1020 routes (4x1G+10G) | 16 |

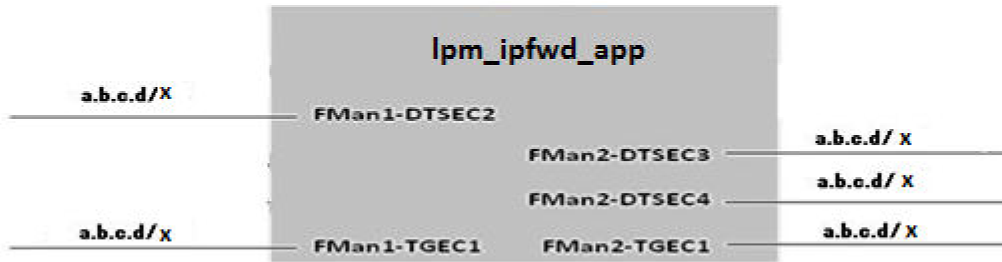



Figure 1: usdpaa_config_p4_serdes_0xe.xml

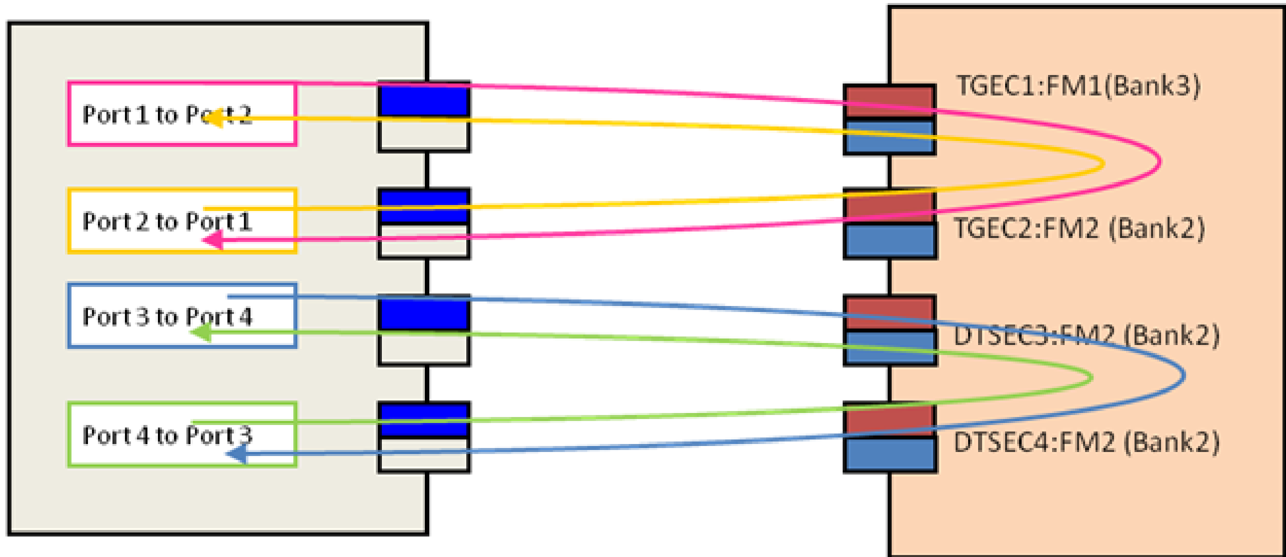
The figure above shows the ethernet interfaces available as per configuration file " usdpaa_config_p4_serdes_0xe.xml". It contains the 1 RGMII port (FMAN1, DTSEC 2), 2 SGMII ports ( FMan2, DTSECs 3- 4 ) and 2 XAUI ports (FMAN1, TGEC1 and FMAN2-TGEC2). It is the user's wish to use any combination of ports available with the configuration file. The above table shows the sample shell scripts that user can use for this configuration. If user uses 2x10G, following is the flow configuration.

Test Center <-> P4080

**Port 2 to Port 1 (512 flows) :**
**Dst : 190.0.24.2 - 190.255.24.2**
**Dst : 191.0.24.2 - 191.255.24.2**
**Gw : 192.168.60.2**

**Port 1 to Port 2 (512 flows) :**
**Dst : 192.0.24.2 - 192.255.24.2**
**Dst : 193.0.24.2 - 193.255.24.2**
**Gw : 192.168.160.2**

Figure 2 : Flow Configuration for 2x10G on P4080DS




**Port 2 to Port 1 (256 flows):**
**Dst : 190.0.24.2 - 190.255.24.2**
**Gw : 192.168.60.2**

**Port 1 to Port 2 (256 flows):**
**Dst : 193.0.24.2 - 193.255.24.2**
**Gw : 192.168.160.2**

**Port 4 to Port 3 (256 flows):**
**Dst : 191.0.24.2 - 191.255.24.2**
**Gw : 192.168.130.2**

**Port 3 to Port 4 (256 flows):**
**Dst : 192.0.24.2 - 192.255.24.2**
**Gw : 192.168.140.2**

Figure 3 : configuration for 2x10G and 2x1G on P4080

For running LPM IPFwd on P3041DS/P5020DS, the user can use " usdpaa_config_p2_p3_p5_14g.xml". Following is the flow configuration.

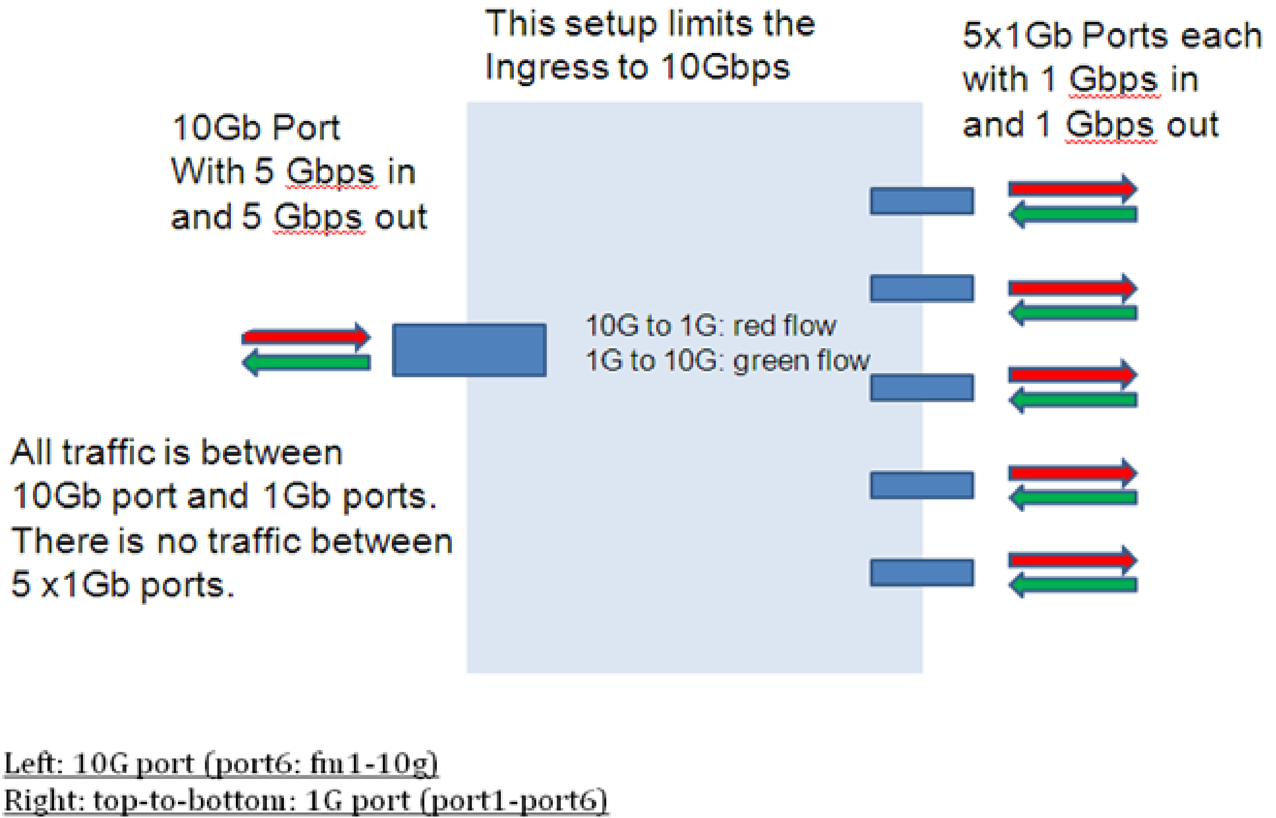



Figure 4 : Flow Configuration for 10G on P5020DS and P3041DS

Port 1 to Port 6 (100 flows):
Dst: 195.0.24.2 – 195.99.24.2
Gw: 192.168.60.2

Port 6 to Port 1 (100 flows):
Dst: 190.0.24.2 – 190.99.24.2
Gw: 192.168.10.2

Port 2 to Port 6 (100 flows):
Dst: 196.0.24.2 – 196.99.24.2
Gw: 192.168.60.2

Port 6 to Port 2 (100 flows):
Dst: 191.0.24.2 – 191.99.24.2
Gw: 192.168.20.2

Port 3 to Port 6 (100 flows):
Dst: 197.0.24.2 – 197.99.24.2
Gw: 192.168.60.2

Port 6 to Port 3 (100 flows):
Dst: 192.0.24.2 – 192.99.24.2
Gw: 192.168.30.2

Port 4 to Port 6 (100 flows):
Dst: 198.0.24.2 – 198.99.24.2
Gw: 192.168.60.2

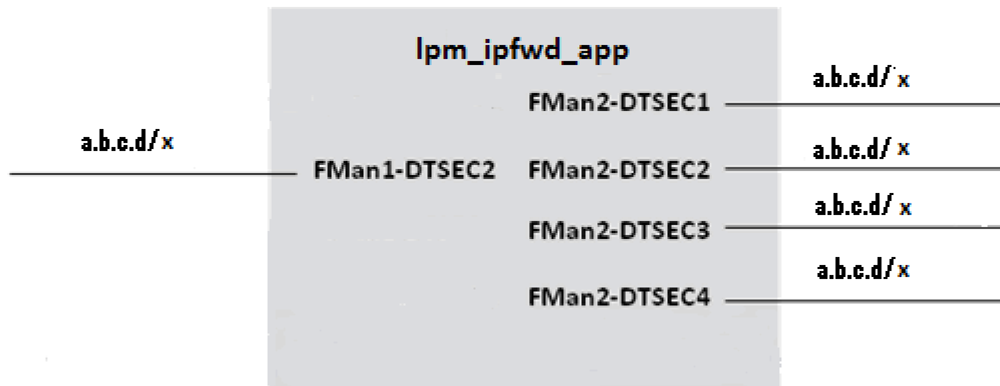
Port 6 to Port 4 (100 flows):
Dst: 193.0.24.2 – 193.99.24.2
Gw: 192.168.40.2

Port 5 to Port 6 (100 flows):
Dst: 199.0.24.2 – 199.99.24.2
Gw: 192.168.60.2

Port 6 to Port 5 (100 flows):
Dst: 194.0.24.2 – 194.99.24.2
Gw: 192.168.50.2

How to run if user has no XAUI but only SGMII riser card?

Assume user does not have a XAUI riser card and wants to run LPM IPFwd using only the SGMII ports as shown below.




This configuration contains the 1 RGMII port (FMAN1, DTSEC 2), 4 SGMII ports ( FMan2, DTSECs 1- 4 ). The user can modify the configuration file and sample shell scripts as per requirement. The configuration file would contain the following

<cfgdata>

<config>

<engine name="fm1">

<port type="1G" number="0" policy="hash_ipsec_src_dst_spi_policy6"/>

<port type="1G" number="1" policy="hash_ipsec_src_dst_spi_policy7"/>

<port type="1G" number="2" policy="hash_ipsec_src_dst_spi_policy8"/>

<port type="1G" number="3" policy="hash_ipsec_src_dst_spi_policy9"/>

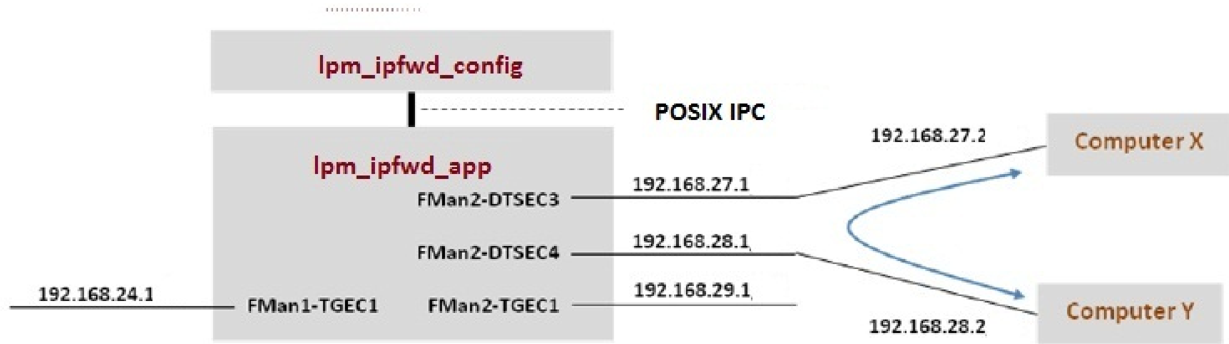
```
</engine>
</config>
</cfgdata>
```

User can create a new shell script which would make routes between the 4x1G. Here is the content of the script.

```
net_pair_routes()
{
net=0
while [ "$net" -le $5 ]
do
lpm_ipfwd_config -P $pid -B -c $2 -d $1.$net.24.2 -n $3 -g \
192.168.$4.2
net=`expr $net + 1`
done
}
lpm_ipfwd_config -P $pid -F -a 192.168.110.1 -i 6
lpm_ipfwd_config -P $pid -F -a 192.168.120.1 -i 7
lpm_ipfwd_config -P $pid -F -a 192.168.130.1 -i 8
lpm_ipfwd_config -P $pid -F -a 192.168.140.1 -i 9
lpm_ipfwd_config -P $pid -G -s 192.168.110.2 -m 02:00:c0:a8:6e:02 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.120.2 -m 02:00:c0:a8:78:02 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.130.2 -m 02:00:c0:a8:82:02 -r true
lpm_ipfwd_config -P $pid -G -s 192.168.140.2 -m 02:00:c0:a8:8c:02 -r true
# 1024
net_pair_routes 190 1 16 110 255 # 256
net_pair_routes 191 1 16 120 255 # 256
net_pair_routes 192 1 16 130 255 # 256
net_pair_routes 193 1 16 140 255 # 256
```

## 45.1.18 Using Two Computers to Test the IPFWD Application Suite

This section describes how to configure the IPFWD application suite to forward packets between two ordinary computers as shown in the following figure. It is assumed that the two 1 Gbps Ethernet interfaces provided by SerDes 0xe are used.




Assign IP addresses to all the interfaces. The IP addresses used here for Computer X and Computer Y are examples. Their MAC addresses must be known as they will be needed in the commands below.

Keep in mind that ipfwd_app has no default IP address assigned to the interfaces. This means that you must first assign IP addresses to the interfaces and then use IP addresses for Computer X and Computer Y that are on the subnets shown in the figure. Please be careful with the network parameters. Any mistake will prevent packets from flowing from end to end.

Follow these steps on Computer X:

1. Set the ip address for eth0 interface

ifconfig eth0 192.168.27.2 up

Set the default gateway for subnet 192.168.27.1

route add default gw 192.168.27.1 eth0

2. No need to add arp on host side. The application can handle external arp requests

Follow these steps on Computer Y:

1. Set the ip address for eth0 interface

ifconfig eth0 192.168.28.2 up

Set the default gateway for subnet 192.168.28.1

route add default gw 192.168.28.1 eth0

2. No need to add arp on host side. The application can handle external arp requests

The commands to perform on P4080 are:

# Boot the P4080 and login as root.

*Assign IP address to fm1-gb1*

ifconfig fm1-gb1 <IPADD> up

# Assume use of cores 1 - 7 lpm_ipfwd_app 1..7 -d 0x10000000 -b 0:0:1728 -i fm1-10g,fm2-gb2,fm2-gb3,fm2-10g

# Now ssh to P4080 linux on other terminal

ssh root@<IPADD>

give IP address as assigned to fm1-gb1 in the beginning

# Now assign ip address to the interfaces

# First run command to check what all enabled interfaces are available

Note: check pid from application print "Message queue to send: /mq_snd_2536"

lpm_ipfwd_config -P 2536 -E -a true

Interface number: 11

PortID=1:5 is FMan interface node

with MAC Address

02:00:c0:a8:65:fe

Interface number: 9

PortID=1:3 is FMan interface node

with MAC Address

02:00:c0:a8:5b:fe

Interface number: 8

PortID=1:2 is FMan interface node

with MAC Address

02:00:c0:a8:51:fe

Interface number: 5

PortID=0:5 is FMan interface node

with MAC Address

02:00:c0:a8:33:fe

Are all the Enabled Interfaces

# Assign IP address to the interfaces.Use the same interface number displayed as an ouput on giving the above command

lpm_ipfwd_config -P 2536 -F -a 192.168.24.1 -i 5

lpm_ipfwd_config -P 2536 -F -a 192.168.28.1 -i 9

lpm_ipfwd_config -P 2536 -F -a 192.168.27.1 -i 8

lpm_ipfwd_config -P 2536 -F -a 192.168.29.1 -i 11

# Now enter routes and MAC addresses. Format of a MAC address is

# aa:bb:cc:dd:ee:ff where the letters are hexadecimal digits.

lpm_ipfwd_config -P 2536 -B -d 192.168.28.2 -g 192.168.28.2 -n 24 -c 1

lpm_ipfwd_config -P 2536 -G -s 192.168.28.2 -m COMPUTER_Y_MAC_ADDRESS -r true

lpm_ipfwd_config -P 2536 -B -d 192.168.27.2 -g 192.168.27.2 -n 24 -c 1

lpm_ipfwd_config -P 2536 -G -s 192.168.27.2 -m COMPUTER_X_MAC_ADDRESS -r true

# Run fmc command cd /usr/etc

fmc -c usdpaa_config_p4_serdes_0xe.xml -p usdpaa_policy_hash_lpm_ipv4.xml -a

# Now, traffic can pass between Computer X and Computer Y. For example, on Computer X

# enter:

ping 192.168.28.2

## 45.1.19 Running LPM IPv4 forwarding on P4080DS board

The instructions below describe how to run LPM-IPFWD. Traffic should only be directed to the P4080DS once the application is running and configuration via the ipfwd_config application is completed.

- On linux prompt, assign IP address to fm1-gb1

$ ifconfig fm1-gb1 <IPADD> up

Now run the FMC command

$ cd /usr/etc

*To setup the FMan to distribute traffic to 32 ingress frame queues per port:*

$ fmc -c usdpaa_config_p4_serdes_0xe.xml -p usdpaa_policy_hash_lpm_ipv4.xml -a

- Run LPM-IPFWD application

*The main LPM-IPFwd application binary is called* lpm_ipfwd_app *. The application can run on multiple cores as specified by the first parameter to the application. To run it to handle traffic distributed over 32 ingress frame queues per port:*

$ lpm_ipfwd_app <m..n>

By default lpm_ipfwd_app uses usdpaa_config_p4_serdes_0xe.xml and usdpaa_policy_hash_lpm_ipv4.xml files.

**LPM-IPFWD application command syntax:**

[root@p4080 etc]# lpm_ipfwd_app --usage

Usage: lpm_ipfwd_app [-n?V] [-b x:y:z] [-c FILE] [-d SIZE] [-i FILE] [-p FILE]

[--buffers=x:y:z] [--fm-config=FILE] [--dma-mem=SIZE]

[--fm-interfaces=FILE] [--non-interactive] [--fm-pcd=FILE]

[--cpu-range] [--help] [--usage] [--version] [cpu-range]

LPM-IPFWD application run command:

[root@p4080 root]# cd /usr/etc

[root@p4080 etc]# lpm_ipfwd_app 1..7 -d 0x4000000 -b 0:0:1728 -i fm1-10g,fm2-gb2,fm2-gb3,fm2-10g

To use dma region size as 64M in USDPAA for lpm-ipfwd application, make sure to set the same or greater size in bootargs. E.g. usdpaa_mem=256M

[1] 5363

[root@p4080 etc]# Found /fsl,dpaa/dpa-fman0-oh@1, Tx Channel = 46, FMAN = 0, Port ID = 1

Found /fsl,dpaa/ethernet@4, Tx Channel = 40, FMAN = 0, Port ID = 0

Found /fsl,dpaa/ethernet@7, Tx Channel = 63, FMAN = 1, Port ID = 2

Found /fsl,dpaa/ethernet@8, Tx Channel = 64, FMAN = 1, Port ID = 3

Found /fsl,dpaa/ethernet@9, Tx Channel = 60, FMAN = 1, Port ID = 0

Configuring for 4 network interfaces

Allocated DMA region size 0x10000000

lpm_ipfwd_app starting

IPv4 FIB table init now... Done!

Message queue to send: /mq_snd_2536

Message queue to receive: /mq_rcv_2536

Thread uid:0 alive (on cpu 1)

Release 0 bufs to BPID 7

Release 0 bufs to BPID 8

Release 1600 bufs to BPID 9

Thread uid:1 alive (on cpu 2)

Thread uid:2 alive (on cpu 3)

Thread uid:3 alive (on cpu 4)

Thread uid:4 alive (on cpu 5)

Thread uid:5 alive (on cpu 6)

Thread uid:6 alive (on cpu 7)

If, in the run application command, cpu-range is given i.e. "lpm_ipfwd_app *<m..n>*" LPM-IPFWD application starts threads on cpu-range m..n. The main thread (by default on CPU 1) then does global initialization needed by the application, including starting other application threads.

If, on the other hand, run application command is given without any cpu-range i.e. "lpm_ipfwd_app" LPM-IPFWD application starts up with a single thread running on CPU 1 by default, which does global initialization needed by the application and enables all the network interfaces.

The CLI (Command-Line Interface) allows you to add and remove additional threads to enable the use of multiple CPUs, with the only restriction being that the primary thread on CPU 1 cannot be removed (except by shutting down the application).

To add a thread on a single CPU (e.g. CPU 2):

> add 2

To add threads on a range of CPUs:

> add 3..6

To list the threads (this also queries each thread, verifying that they aren't blocked):

> list

Thread alive on cpu 1

Thread alive on cpu 2

Thread alive on cpu 3

Thread alive on cpu 4

Thread alive on cpu 5

Thread alive on cpu 6

To enable all interfaces

> macs on

To disable all interfaces

> macs off

To perform a controlled shutdown of ipfwd (this includes disabling the network ports):

> quit

- Once the application starts, it can receive the configuration commands. Run application configuration script.

For creating route entries, the binary *lpm_ipfwd_config* is run. This binary processes the configuration request from the user (using the command line) and populates the configuration via Linux standard posix IPC messages to the LPM-IPFwd application.

The shell script mentioned below contains sample commands to add route entries. Detailed description of all lpm_ipfwd_config commands is provided in section Syntax on page 666.

SSH to p4080 linux on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to fm1-gb1 in the beginning)

*Run the shell script:*

The shell script needs pid as input (process id of the application to hook up with)

pid can be read from the application prints "Message queue to send: /mq_snd_2536 "

lpm_ipfwd_20G.sh "pid"

$ lpm_ipfwd_ 20G.sh 2536

There are example shell scripts available. They can assign IP addresses to the interfaces, add an ARP entry and can use variable netmask while creating route entries. The following table summarizes the settings done by these scripts. Check sectionPossible configuration scenario for LPM based IPFWD on page 773 for more details.

For the LPM-IPFwd application to forward traffic successfully, traffic destined for the P4080DS ports must have the appropriate destination IP addresses.

Console messages are printed for each entry added to the routing table. Once all configuration is completed, the application moves to the packet processing phase - the message "LPM-IPFwd Route Creation completed" is printed on the console.

At this point, traffic can be sent to the ethernet interfaces, IPv4 packets would be processed by the LPM application on the cpu-range specified by the user on the application command-line.

## 45.1.20 Running LPM IPv4 forwarding on P3041/P5020 board

The instructions below describe how to run LPM-IPFWD on P3041/P5020. Traffic should only be directed to P3041/P5020 once the application is running and configuration via the lpm_ipfwd_config application is completed. On linux prompt, assign IP address to eth0

$ ifconfig eth0 <IPADD> up

Configure FMan PCD using fmc with the XML files in /usr/etc:

$ cd /usr/etc

To setup the FMan to distribute traffic to 32 ingress frame queues per port:

$ fmc -c usdpaa_config_p2_p3_p5_14g.xml -p usdpaa_policy_hash_lpm_ipv4.xml -a

Run lpm-IPFWD

$ lpm_ipfwd_app <m..n> -c usdpaa_config_p2_p3_p5_14g.xml -p usdpaa_policy_hash_lpm_ipv4.xml -d 0x4000000 -b 0:0:1728 -i fm1-gb0, fm1-gb1,fm1-gb3,fm1-gb4,fm1-10g

For P3041, m..n can be 0..3. For P5020, m..n can be 0..1.

SSH to board (linux) on another terminal:

$ ssh root@<IPADD> (give the IP address as assigned to eth0 in the beginning)

Run the shell script:

$ lpm_ipfwd_14G.sh

There is an example shell script available named as lpm_ipfwd_14G.sh creates routes for only the 4 x 1G and 1x10G interfaces. It can assign ip addresses to the interfaces, add an ARP entry . Check section Possible configuration scenario for LPM based IPFWD on page 773 for more details. Now traffic can be run as per the routes created.

## 45.1.21 USDPAA LPM IP Fwd performance gap between 6 core and 8 core

USDPAA LPM IPfwd performance for 8 core is less than 6 core on e6500 series. USDPAA LPM IP Fwd application need to run using "-s" option to bridge the gap between two configuration.

## 45.1.22 PPAC (and IPFwd) CLI commands

The following commands are illustrated in the context of IPFwd, but the commands are common to all PPAC-based applications.

To add a thread on a single CPU (e.g. CPU 2):

> add 2

To add threads on a range of CPUs:

> add 3..6

To list the threads (this also queries each thread, verifying that they aren’t blocked):

> list

Thread uid:0 alive (on cpu 1)

Thread uid:1 alive (on cpu 2)

Thread uid:2 alive (on cpu 3)

Thread uid:3 alive (on cpu 4)

Thread uid:4 alive (on cpu 5)

Thread uid:5 alive (on cpu 6)

Thread uid:6 alive (on cpu 7)

To remove a thread by its UID:

> rm uid:2

Thread uid:2 killed (cpu 3)

To remove a thread running on a given CPU:

> rm 5 Thread uid:4 killed (cpu 5)

To enable all interfaces:

> macs on

To disable all interfaces:

> macs off

To perform a controlled shutdown of ipfwd (this includes disabling the network ports):

```
> quit
```

To query cgr

```
> cgr
Rx CGR ID: 10, selected fields;
      cscn_en: 0
    cscn_targ: 0x00800000
      cstd_en: 1
           cs: 0
    cs_thresh: 0x00_0000_1000
         mode: 1
       i_bcnt: 0x00_0000_0e1e
       a_bcnt: 0x00_0000_0e1e
Tx CGR ID: 11, selected fields;
      cscn_en: 0
    cscn_targ: 0x00800000
      cstd_en: 1
           cs: 0
    cs_thresh: 0x00_0000_0200
         mode: 1
       i_bcnt: 0x00_0000_0002
       a_bcnt: 0x00_0000_0004
```

## 45.1.23 Syntax

The syntax is as follows:

```
$ [root@p4080 bin]# ipfwd_config --help
Usage: ipfwd_config [OPTION...]

-B, --routeadd=TYPE adding a route
-C, --routedel=TYPE deleting a route
-E, --showintf=TYPE show interfaces
-F, --intfconf=TYPE change intf config
-G, --arpadd=TYPE adding a arp entry
-H, --arpdel=TYPE deleting a arp entry
-?, --help Give this help list
--usage Give a short usage message
-V, --version Print program version
```

### 45.1.24 Command to show all enabled interfaces and their interface numbers

The command to show all the enabled interfaces while running IPv4 forward is as follows:

Ipfwd_config -P pid -E -a true

Table 86: Field Description (show all enabled interfaces)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -a | Show all the interfaces | Yes | true |

Command to show all enabled interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# Ipfwd_config -P 2536 -E -a true

Interface number: 11

PortID=1:5 is FMan interface node

with MAC Address

02:00:c0:a8:65:fe

Interface number: 9

PortID=1:3 is FMan interface node

with MAC Address

02:00:c0:a8:5b:fe

Interface number: 8

PortID=1:2 is FMan interface node

with MAC Address

02:00:c0:a8:51:fe

Interface number: 5

PortID=0:5 is FMan interface node

with MAC Address

02:00:c0:a8:33:fe

Are all the Enabled Interfaces [root@p4080 bin]#

### 45.1.25 Help for show all enabled interfaces command

The command to obtain help for show all enabled interfaces command is as follows:

lpm_ipfwd_config -E -help

Help for show all enabled interfaces

[root@p4080 etc]# lpm_ipfwd_config -E --help

Usage: -E [OPTION...]

-a, --a=ALL All interfaces

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

### 45.1.26 Assign IP address to interfaces

The command to assign IP address to shared or private interfaces while running IPv4 forward is as follows:

lpm_ipfwd_config -P pid -F -a 192.168.60.1 -i <Interface number>

The command to assign IP address to MAC-less interfaces while running IPv4 forward is as follows:

lpm_ipfwd_config -P pid -F -a 192.168.60.1 -n <MAC-less interface name>

NOTE

Note: The interface name(MAC-less) or interface number to be used here must be one of the names/numbers that got displayed as the output of "show all enabled interfaces command" in section Command to show all enabled interfaces and their interface numbers on page 666.

Table 87: Field description (assign IP address to shared or private MAC interfaces)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -a | IP Address | Yes | a.b.c.d |
| -i | Interface number | Yes | 0-11<br>(Choose this number from "show all enabled interfaces" command output) |

Table 88: Field description (assign IP address to MAC-less interfaces)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -a | IP Address | Yes | a.b.c.d |
| -n | MAC-less Interface name | Yes | eth3<br>(Choose this name in case of MAC-less from "show all enabled interfaces" command output) |

Assign IP address to interfaces

Note: check pid from application print "Message queue to send: /mq_snd_2536"

. Command to assign IP address to private or shared MAC interfaces

lpm_ipfwd_config -P 2536 -F -a 192.168.60.1 -i 5

IPADDR assigned = 0xc0a83c01 to interface num 5


Non-disclosure Agreement Required.

Intf Configuration Changed successfully

. Command to assign IP address to MAC-less interface

lpm_ipfwd_config -P 2536 -F -a 192.168.55.6 -n eth3

IPADDR assigned = 0xc0a88506 to MACLESS intf eth3

Intf Configuration Changed successfully

## 45.1.27 Help for assign IP address to interfaces

The command to obtain help for assign IP address to interfaces command is as follows:

lpm_ipfwd_config -F --help

Help for assign IP address to interfaces

[root@p4080 etc]# lpm_ipfwd_config -F --help

Usage: -F [OPTION...]

-a, --a=IPADDR IP Address

-i, --i=IFNUM If Number

-n, --n=IFNAME MACLESS Interface Name

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

## 45.1.28 Adding a Route Entry

The command to add a route while running IPv4 forward is as follows:

lpm_ipfwd_config -P pid -B -d b.c.d.e -g c.d.e.f -n maskbits -c numentry

Table 89: Field Description (Adding a Route Entry)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -d | Destination IP Address | Yes | a.b.c.d |
| -g | Gateway IP Address | Yes | a.b.c.d |
| -n | netmask length to be used by LPM | Yes | upto 32 bits |
| -c | number of fib entries | Yes | 1- valid count |

Adding a Route Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# lpm_ipfwd_config -P 2536 -B -d 192.168.24.2 -g 192.168.24.2 -n 24 -c 1024

Route Entry Added successfully

[root@p4080 bin]#

## 45.1.29 Help for Route Entry Addition

The command to obtain help for route entry addition is as follows:

lpm_ipfwd_config -B --help

Help for Adding a Route Entry

[root@p4080 bin]# lpm_ipfwd_config -B --help

Usage: -B [OPTION]

-d, --d=DESTIP Destination IP

-g, --g=GWIP Gateway IP

-n, --n=MASKBITS Netmask length

-c, --c=NUMENTRY Number of fib entries

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

## 45.1.30 Deleting a Route Entry

NOT IMPLEMENTED

## 45.1.31 Help for Deleting a Route Entry

Delete route command is not implemented

## 45.1.32 Adding an ARP Entry

The command to add an ARP entry while running IPv4 forward is as follows:

lpm_ipfwd_config -P pid -G -s a.b.c.d -m aa:bb:cc:dd:ee [-r true]

Table 90: Field Description (Adding an ARP Entry)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|

*Table continues on the next page...*

Table 90: Field Description (Adding an ARP Entry) (continued)

| | | | |
|---|---|---|---|
| -s | Gateway IP Address | Yes | a.b.c.d |
| -m | Mac Address | Yes | aa:bb:cc:dd:ee |
| -r | Replace existing entry | No | true/ false {Default: false} |

Adding an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# lpm_ipfwd_config -P 2536 -G -s 192.168.24.2 -m 02:00:c0:a8:33:fd -r true

ARP Entry Added successfully

## 45.1.33 Help for ARP Entry Addition

The command to obtain help for ARP entry addition is as follows:

lpm-ipfwd_config -G --help

Help for Adding an ARP Entry

[root@p4080 etc]# lpm_ipfwd_config -G --help

Usage: -G [OPTION...]

-m, --m=MACADDR MAC Address

-r, --r=Replace Replace Exiting Entry - true/ false {Default:

false}

-s, --s=IPADDR IP Address

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

## 45.1.34 Deleting an ARP Entry

The command to delete an ARP while running the IPv4 forward is as follows:

lpm_ipfwd_config -P pid -H -s a.b.c.d

Table 91: Field Description (Deleting an ARP Entry)

| Parameter | Description | Mandatory | Format/ Value |
|---|---|---|---|
| -s | Gateway IP Address | Yes | a.b.c.d |

Deleting an ARP Entry

Note: check pid from application print "Message queue to send: /mq_snd_2536"

[root@p4080 bin]# lpm_ipfwd_config -P 2536 -H -s 192.168.24.2

Arp Entry Deleted successfully

[root@p4080 bin]#

## 45.1.35 Help for Deleting an ARP Entry

The command to obtain help for ARP entry deletion is as follows:

lpm_ipfwd_config -H --help

Help for Deleting an ARP Entry

[root@p4080 etc]# lpm_ipfwd_config -H --help

Usage: -H [OPTION...]

-s, --s=IPADDR IP Address

-?, --help Give this help list

--usage Give a short usage message

-V, --version Print program version

Mandatory or optional arguments to long options are also mandatory or optional for any corresponding short options.

## 45.1.36 References

1. USDPAA PPAC User Guide
2. QMan/BMan API Guide

# Chapter 46
# Watchdog Timers

## 46.1 Watchdog Device Driver User Manual

**Description**
Watchdog driver description here.

**Specifications**

| Hardware: | watchdog timer |
|---|---|
| Operating system: | Linux 2.6.35+ |

**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 --->`<br>`[*] Watchdog Timer Support --->`<br>`[*] Disable watchdog shutdown on close`<br>`[*] PowerPC Book-E Watchdog Timer` | PowerPC Book-E Watchdog Timer |

**Compile-time Configuration Options**

| Option | Values | Default Value | Description |
|---|---|---|---|
| CONFIG_BOOKE_WDT | y/n | y | PowerPC Book-E Watchdog Timer |

### Source Files

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

| Source File | Description |
|---|---|
| drivers/char/watchdog/booke_wdt.c | PowerPC Book-E Watchdog Timer |

### User Space Application

The following applications will be used during functional or performance testing. Please refer to the SDK UM document for the detailed build procedure.

| Command Name | Description | Package Name |
|---|---|---|
| watch | watchdog is a daemon for watchdog feeding | watchdog |

### Verification in Linux

· set nfs rootfs

```
build a rootfs image which includes watchdog daemon.
```

· et booting parameter

```
on the u-boot prompt, set following parameter



set nfsargs "setenv bootargs wdt_period=35 root=/dev/nfs rw nfsroot=$serverip:
$rootpath ip=$ipaddr:$serverip:$gatewayip:$netmask:$hostname:$netdev:off



console=$consoledev,$baudrate $othbootargs"



set nfsboot "run nfsargs;tftp $loadaddr $bootfile;tftp $fdtaddr $fdtfile;bootm
$loadaddr - $fdtaddr"



run nfsboot



Note: wdt_period is watchdog timeout period, set it with proper value depending on
your board bus frequency.

Also wdt_period is inversely proportional to watchdog expiry time ie. Higher the
wdt_period, lower the watchdog expiry time.

So if we increase wdt_period to high, watchdog will expiry early.
```

· check watchdog feeding operation

```
after system boots up, check the screen output, if you see

...

PowerPC Book-E Watchdog Timer Enabled (wdt_period=35)

...



it means watchdog module loads successfully



login in system, run command "watchdog /dev/watchdog"



root@p1020rdb:~# watchdog /dev/watchdog

root@p1020rdb:~# ps -ae | grep watchdog

 3285 ?        00:00:00 watchdog

root@p1020rdb:~#
```

```
wait for some minutes, if system is still alive, watchdog feeding is OK
```

· check watchdog reboot operation

```
run command "killall"




root@p1020rdb:~# killall -9 watchdog


root@p1020rdb:~#


root@p1020rdb:~# ps -ae | grep watchdog


root@p1020rdb:~#


root@p1020rdb:~# PowerPC Book-E Watchdog Exception




wait for some seconds, if system reboots, watchdog reboot operation is OK
```

## Known Bugs, Limitations, or Technical Issues

```
 · On the T4240RDB board, if you will use watchdog, please disable the following menu
configuration in kernel

Location:

      |--> Device
Drivers

                |--> Hardware Monitoring support (HWMON [=n])
   Or they are conflicting with each other.
```

## Supporting Documentation

· N/A

# Chapter 47
# Open Data Plane (ODP) User Guide

## 47.1 Introduction

This document provides information about the usage of ODP Sample Applications built on QORIQ DPAA1 ODP implementation. User can experience the main functionalities of OpenDataPlane (ODP) with these examples and can also learn how to use the ODP API from source code of these applications. The document explores the following target applications:

- ODP generator sample application
- ODP pktio sample application
- ODP ipsec sample application
- ODP packet classify sample application
- ODP timer sample application
- ODP L3 Forwarding sample application

See Appendix A of the DPAA1 ODP Simple Applications User Guide for list of supported processors.

Before using any of the sample application in this document, you need to build and install the SDK using yocto.

More information about ODP introduction and architecture are available on Linaro.

### 47.1.1 Intended Audience

This document is intended for software developers and architects who want to develop ODP applications on QorIQ DPAA1 based platforms. The document assumes that users are familiar with Linux-based software development, and also with the ODP concepts and APIs.

### 47.1.2 Definitions and Acronyms

| | |
|---|---|
| BMan | Buffer Manager |
| DPAA | Data path acceleration architecture |
| FMan | Frame Manager |
| QMan | Queue Manager |
| USDPAA | User Space Data Path Acceleration Architecture |

## 47.2 Test Setup

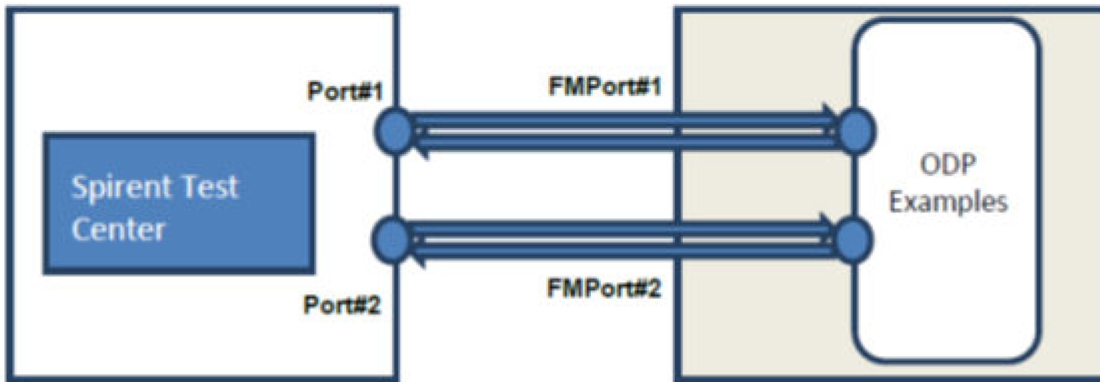



Table 92: Test Setup

| Ports | LS1043A RDB |
| --- | --- |
| FMPort#1 | fm0-mac3 |
| FMPort#2 | fm0-mac4 |

### Note

When running any application make sure that the board was booted with the following args in uboot:

```
setenv othbootargs "isolcpus=1-3 usdpaa_mem=256M bportals=s0 qportals=s0
fsl_fm_max_frm=9000".
```

## 47.3 ODP generator sample application (odp_generator)

### Overview

This application verifies the following DPAA ODP components, by generating own traffic or by receiving traffic from external source:

- Pktio
- Packet
- Queue
- Scheduler
- Timer
- Pool

### Test Configuration

The application will be configured using its available command line options:

```
Usage: %s OPTIONS
  E.g. %s -I eth1 -r

OpenDataPlane example application.

  Work mode:
    1.send udp packets
      odp_generator -I eth0 --srcmac fe:0f:97:c9:e0:44  --dstmac 32:cb:9b:27:2f:1a --
srcip 192.168.0.1 --dstip 192.168.0.2 -m u
    2.receive udp packets
      odp_generator -I eth0 -m r
    3.work likes ping
      odp_generator -I eth0 --srcmac fe:0f:97:c9:e0:44  --dstmac 32:cb:9b:27:2f:1a --
srcip 192.168.0.1 --dstip 192.168.0.2 -m p

Mandatory OPTIONS:
  -I, --interface Eth interfaces (comma-separated, no spaces)
  -a, --srcmac src mac address
  -b, --dstmac dst mac address
  -c, --srcip src ip address
  -d, --dstip dst ip address
  -s, --packetsize payload length of the packets
  -m, --mode work mode: send udp(u), receive(r), send icmp(p)
  -n, --count the number of packets to be send
  -t, --timeout only for ping mode, wait ICMP reply timeout seconds
  -i, --interval wait interval ms between sending each packet
                 default is 1000ms. 0 for flood mode

Optional OPTIONS
  -h, --help       Display help and exit.
 environment variables: ODP_PKTIO_DISABLE_SOCKET_MMAP
                        ODP_PKTIO_DISABLE_SOCKET_MMSG
                        ODP_PKTIO_DISABLE_SOCKET_BASIC
 can be used to advanced pkt I/O selection for linux-generic
```

### Test environment

This test uses the hardware setup described in paragraph 2 with the following connections:

| Ports | LS1043A RDB |
|---|---|
| FMPort#1 | fm0-mac3 |
| | |

The ODP application needs to be provided with the following resources:

- test config xml file (defines the fman and ports used by the application)
  - `odp_test_config_ls1043.xml`
- pcd xml file
  - /usr/etc/odp_pktio_pcd.xml

### Running the ODP generator application

In order to run this application one needs to use the following commands:

```
export DEF_CFG_PATH=<path_to_test_config_file>
export DEF_PCD_PATH=<path_to_pcd_file>


#./odp_generator <options>
```

For additional details on options, see the command line options mentioned above.

### Test description

Depending on the options given in command line, the application may only receive traffic from the test center – and this will be seen in the application console, without any reply from the application, or may send packets to the traffic source (when it runs in one of the two modes – ping or udp)

As a reference packet generator mode will be chosen. The command line for running this mode is:

```
#./odp_generator -I <traffic port> -m r
```

A number of 10000 packets will be sent using the test center.

### Packet format

| Eth source | Eth destination | Ip source | Ip destination | Proto |
|---|---|---|---|---|
| any | Input interface MAC | any | any | IPv4/UDP |

The output of the application will be:

```
[02] created mode: RECEIVE
  [01] created mode: RECEIVE
  [03] created mode: RECEIVE
  [04] created mode: RECEIVE
  [06] created mode: RECEIVE
  [05] created mode: RECEIVE
  [07] created mode: RECEIVE
  [08] created mode: RECEIVE
  [07] receive Packet proto:IP id 32965 UDP payload 18
  [05] receive Packet proto:IP id 32966 UDP payload 18
  [06] receive Packet proto:IP id 32967 UDP payload 18
  [03] receive Packet proto:IP id 32968 UDP payload 18
  [03] receive Packet proto:IP id 32969 UDP payload 18
  [03] receive Packet proto:IP id 32970 UDP payload 18
  [03] receive Packet proto:IP id 32971 UDP payload 18
  [03] receive Packet proto:IP id 32972 UDP payload 18
  [03] receive Packet proto:IP id 32973 UDP payload 18

[…]
```

Note that the leftmost number will vary depending on the worker thread that received a given packet.

## 47.4 ODP pktio sample application (odp_pktio)

### Overview

This application verifies the following DPAA ODP components, by receiving traffic from external source and sending back the received packets:

- Pktio
- Packet
- Queue
- Scheduler
- Pool

### Test configuration

The application will be configured using its available command line options:

```
Usage: %s OPTIONS
          E.g. %s -i eth1,eth2,eth3 -m 0

        OpenDataPlane example application.

        Mandatory OPTIONS:
          -i, --interface Eth interfaces (comma-separated, no spaces)

        Optional OPTIONS
          -c, --count <number> CPU count.
          -m, --mode      0: Receive and send directly (no queues)
                          1: Receive and send via queues.
                          2: Receive via scheduler, send via queues.
          -h, --help           Display help and exit.
         environment variables: ODP_PKTIO_DISABLE_SOCKET_MMAP
                                ODP_PKTIO_DISABLE_SOCKET_MMSG
                                ODP_PKTIO_DISABLE_SOCKET_BASIC
         can be used to advanced pkt I/O selection for linux-generic
```

### Test Environment

This test uses the hardware setup described in paragraph 2 with the following connections:

| Ports | LS1043A RDB |
|---|---|
| FMPort#1 | fm0-mac3 |
| FMPort#2 | fm0-mac4 |

The ODP pktio application needs to be provided with the following resources:

- test config xml file (defines the fman and ports used by the application)
  - `odp_test_config_ls1043.xml`
- pcd xml file
  - `/usr/etc/odp_pktio_pcd.xml`

### Running the ODP pktio application

In order to run this application one needs to use the following commands:

```
export DEF_CFG_PATH=<path_to_test_config_file>
export DEF_PCD_PATH=<path_to_pcd_file>


#./odp_pktio <options>
```

For additional options details, see the command line options mentioned above.

### Test description

The application will receive traffic from the test center – and this will be seen in the application console, on one interface or on both interfaces. Once a packet is received, its destination Ethernet addresses will be swapped (and so the IP addresses) and the packet will be sent back on the port it came, to the packet originator (test center).

As a reference packet scheduler mode will be chosen. The command line for running this mode is:

```
#./ odp_pktio  -i <traffic_port1> -m 2
```

A number of 100000 packets will be sent on one port, from the test center.

### Packet format

| Eth source | Eth destination | Ip source | Ip destination | Proto |
| --- | --- | --- | --- | --- |
| any | Input interface MAC | any | any | Ipv4/UDP |

The application will send back 100000 packets to test center, each packet having its Ethernet and IP addresses swapped.

The output will be:

```
 […]

  [03] looked up pktio:01, queue mode (ATOMIC queues)
        default pktio01-INPUT queue:2
  [07] looked up pktio:01, queue mode (ATOMIC queues)
        default pktio01-INPUT queue:2
  [08] looked up pktio:01, queue mode (ATOMIC queues)
        default pktio01-INPUT queue:2
[02] pkt_cnt:100001

[…]
```

## 47.5 ODP ipsec sample applications (odp_ipsec, odp_ipsec_proto)

### Overview

These applications verify the following DPAA ODP components, by authenticating and encrypting/decrypting traffic from external source and forwarding the encrypted/decrypted packets:

- Pktio
- Packet
- Queue
- Scheduler
- Crypto

In addition, the odp_ipsec_proto application demonstrates IPSec ESP protocol offload advantages over standard algorithm-oriented crypto API.

### Test configuration

The applications will be configured using its available command line options:

```
Usage: %s OPTIONS
          E.g. %s -i eth1,eth2,eth3 -m 0

        OpenDataPlane example application.

        Mandatory OPTIONS:
         -i, --interface Eth interfaces (comma-separated, no spaces)
         -m, --mode    0: SYNC
                       1: ASYNC_IN_PLACE
                       2: ASYNC_NEW_BUFFER
                 Default: 0: SYNC api mode

        Routing / IPSec OPTIONS:
         -r, --route SubNet:Intf:NextHopMAC
         -p, --policy SrcSubNet:DstSubNet:(in|out):(ah|esp|both)
         -e, --esp SrcIP:DstIP:(3des|null):SPI:Key192
         -a, --ah SrcIP:DstIP:(md5|null):SPI:Key128

          Where: NextHopMAC is raw hex/dot notation, i.e. 03.BA.44.9A.CE.02
                 IP is decimal/dot notation, i.e. 192.168.1.1
                 SubNet is decimal/dot/slash notation, i.e 192.168.0.0/16
                 SPI is raw hex, 32 bits
                 KeyXXX is raw hex, XXX bits long

          Examples:
             -r 192.168.222.0/24:p8p1:08.00.27.F5.8B.DB
             -p 192.168.111.0/24:192.168.222.0/24:out:esp
             -e 192.168.111.2:192.168.222.2:3des:
201:656c8523255ccc23a66c1917aa0cf30991fce83532a4b224
             -a 192.168.111.2:192.168.222.2:md5:201:a731649644c5dee92cbd9c2e7e188ee6

        Optional OPTIONS
          -c, --count <number> CPU count.
```

```
          -h, --help           Display help and exit.
         environment variables: ODP_PKTIO_DISABLE_SOCKET_MMAP
                                ODP_PKTIO_DISABLE_SOCKET_MMSG
                                ODP_PKTIO_DISABLE_SOCKET_BASIC
         can be used to advanced pkt I/O selection for linux-generic
                                ODP_IPSEC_USE_POLL_QUEUES
         to enable use of poll queues instead of scheduled (default)
                                ODP_IPSEC_STREAM_VERIFY_MDEQ
         to enable use of multiple dequeue for queue draining during
         stream verification instead of single dequeue (default)
```

### Test Environment

This test uses the hardware setup described in paragraph 2 with the following connections:

| Ports | LS1043A RDB |
|---|---|
| FMPort#1 | fm0-mac3 |
| FMPort#2 | fm0-mac4 |

The ODP ipsec applications need to be provided with the following resources:

- test config xml file (defines the fman and ports used by the application)
  - `odp_test_config_ls1043.xml`
- pcd xml file
  - /usr/etc/odp_ipsec_pcd.xml

### Running the ODP ipsec applications

In order to run this application one needs to use the following commands:

```
export DEF_CFG_PATH=<path_to_test_config_file>
export DEF_PCD_PATH=<path_to_pcd_file>


#./odp_ipsec <options>
```

For additional options details, see the command line options mentioned above.

### Test description

The applications will authenticate and encrypt/decrypt the received frames on one port and then will forward the processed frames on the other port.

As a reference the applications will be run with the following command line options:

```
#./ odp_ipsec -i <interface1>,<interface2>\
-r 192.168.111.2/32: <interface1>,<:00.10.18.BA.E4.80 \
-r 192.168.222.2/32: <interface2>,<:A0.36.9F.19.FE.15 \
-p 192.168.111.0/24:192.168.222.0/24:out:both \
-e 192.168.111.2:192.168.222.2:\
3des:201:656c8523255ccc23a66c1917aa0cf30991fce83532a4b224 \
-a 192.168.111.2:192.168.222.2:md5:200:a731649644c5dee92cbd9c2e7e188ee6 \
-p 192.168.222.0/24:192.168.111.0/24:in:both \
-e 192.168.222.2:192.168.111.2:\
3des:201:656c8523255ccc23a66c1917aa0cf30991fce83532a4b224 \
```

```
-a 192.168.222.2:192.168.111.2:md5:200:a731649644c5dee92cbd9c2e7e188ee6 \
-m 1 -c 2
```

NOTE

- The odp_ipsec application doesn’t support the tunnel mode.
- The odp_ipsec_proto application supports only the ipsec-esp policy, so there is no need to specify it. It will be run with the same command line options. It supports only the ipsec-esp policy
- <interface1>, <interface2> depend on DUT.In the code snipet from above <interface1> is the outbound interface – traffic will be encrypted. <interface2> will be the inbound interface – traffic will be authenticated and decrypted.

A number of 100000 packets will be sent on the outbound port, from the test center.

### Packet format

| Eth source | Eth destination | Ip source | Ip destination | Proto |
|---|---|---|---|---|
| any | Input interface MAC | 192.168.111.02 | 192.168.222.02 | Ipv4/UDP |

The application will send back 100000 packets to test center on the inbound port. Packets will be encrypted.

## 47.6 ODP ‘packet classify’ sample application (odp_pkt_classify)

### Overview

This application verifies the following DPAA ODP components, by classifying traffic from external source and sending back the traffic on the received port:

- Pktio
- Packet
- Queue
- Scheduler
- Classification

### Test configuration

The application will be configured using its available command line options:

```
Usage: %s OPTIONS
          E.g. %s -i eth1,eth2,eth3

        OpenDataPlane example application.

        Mandatory OPTIONS:
          -i, --interface Eth interfaces (comma-separated, no spaces)

        Optional OPTIONS
          -c, --count <number> CPU count.
```

```
          -h, --help           Display help and exit.
         environment variables: ODP_PKTIO_DISABLE_SOCKET_MMAP
                                ODP_PKTIO_DISABLE_SOCKET_MMSG
                                ODP_PKTIO_DISABLE_SOCKET_BASIC
         can be used to advanced pkt I/O selection for linux-generic
```

### Test Environment

This test uses the hardware setup described in paragraph 2 with the following connections:

| Ports | LS1043A RDB |
|---|---|
| FMPort#1 | fm0-mac3 |
| FMPort#2 | fm0-mac4 |

The ODP ipsec application needs to be provided with the following resources:

- test config xml file (defines the fman and ports used by the application)
  - `odp_test_config_ls1043.xml`
- pcd xml file
  - `/usr/etc/odp_pktio_pcd.xml`

### Running the ODP packet classify application

In order to run this application one needs to use the following commands:

```
export DEF_CFG_PATH=<path_to_test_config_file>
export DEF_PCD_PATH=<path_to_pcd_file>


#./odp_pkt_classify <options>
```

For additional options details, see the command line options mentioned above.

### Test description

The application will receive traffic from the test center – and this will be seen in the application console, on one interface or on both interfaces. Once a packet is received, it will be classified based on the traffic selectors defined in *odp_pkt_classify* source file and enqueued to the corresponding destination queue. The packet will the be changed - destination Ethernet addresses will be swapped (and so the IP addresses) and sent back on the port it came, to the packet originator (test center).

As a reference the application will be run with the following command line options:

```
#./ odp_pk_classify  -i <traffic_port1>
```

A number of 100000 packets will be sent on one port, from the test center.

### Packet format

| Eth source | Eth destination | Ip source | Ip destination | UDP sport | UDP dport |
|---|---|---|---|---|---|
| any | Input interface MAC | 192.168.100.5 | any | any | any |
| any | Input interface MAC | 192.168.100.10 | any | any | any |
| any | Input interface MAC | any | any | 2152 | any |

The application will send back 100000 packets to test center, each packet having its Ethernet and IP addresses swapped.

The output will be:

```
  […]

   [03] looked up pktio:01, queue mode (ATOMIC queues)
          default pktio01-INPUT queue:2
   [07] looked up pktio:01, queue mode (ATOMIC queues)
          default pktio01-INPUT queue:2
   [08] looked up pktio:01, queue mode (ATOMIC queues)
          default pktio01-INPUT queue:2
 [02] pkt_cnt:100001

 […]
```

# 47.7 ODP timer sample application (odp_timer_test)

### Overview

This application verifies the following DPAA ODP components, by creating a number of odp worker threads each worker thread with timer resolution and period:

- Pktio
- Queue
- Scheduler
- Timer
- Pool

### Test configuration

The application will be configured using its available command line options:

```
  -c, --count <number>    CPU count);
  -r, --resolution <us>   timeout resolution in usec
  -m, --min <us>          minimum timeout in usec
  -x, --max <us>          maximum timeout in usec
  -p, --period <us>       timeout period in usec
  -t, --timeouts <count>  timeout repeat count
  -h, --help              this help
```

### Test environment

There is no requirement to connect the DUT with an external traffic source for this application.

The ODP timer test application needs to be provided with the following resources:

- test config xml file
  - `odp_test_config_ls1043.xml`
- pcd xml file
  - `/usr/etc/odp_pktio_pcd.xml`

### Running the ODP timer application

In order to run this application one needs to use the following commands:

```
export DEF_CFG_PATH=<path_to_test_config_file>
export DEF_PCD_PATH=<path_to_pcd_file>


#./odp_timer_test <options>
```

For additional options details, see the command line options mentioned above.

### Test description

The application create a timer for each worker thread. The timer will have the resolution in micro second given from command line (or default one if it is not provided) and a timeout period which will give the total ticks per thread after which a timeout event is generated.

As a reference the application will be run on one core (only one thread will be created). The command line for running in this mode is:

```
#./odp_timer_test -c 1
```

The output will be:

```
[…]
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 1400
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 1500
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 1600
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 1700
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 1800
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 1900
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 2000
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 2100
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 2200
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 2300
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 2400
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 2500
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 2600
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 2700
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 2800
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 2900
odp_timer_test.c:162:test_abs_timeouts():  [1] timeout, tick 3000
Thread 1 exits
odp_init.c:212:odp_term_local():odp_term_local
ODP timer test complete
```

## 47.8 ODP L3 forwarding sample application (odp_l3_forwarding)

### Overview

This application verifies the following DPAA ODP components, by creating a number of odp worker threads. Each thread forwards the received packets on the interface defined in the routing table.The following components are verified:

- Pktio
- Queue
- Scheduler
- Pool

### Test configuration

The application will be configured using its available command line options:

```
Usage: %s OPTIONS
                 E.g. %s -i eth1,eth2,eth3 -m 0

                 OpenDataPlane example application.

                 Mandatory OPTIONS:
                -i, --interface Eth interfaces:next_hop_mac (comma-separated, no spaces)

                 Optional OPTIONS
                   -c, --count <number> CPU count.
                   -h, --help           Display help and exit.
                 environment variables: ODP_PKTIO_DISABLE_SOCKET_MMAP
                                         ODP_PKTIO_DISABLE_SOCKET_MMSG
                                         ODP_PKTIO_DISABLE_SOCKET_BASIC
                 can be used to advanced pkt I/O selection for linux-generic
```

### Test environment

This test uses the hardware setup described in Test Setup with the following connections:

| Ports | LS1043A RDB |
|---|---|
| Traffic port | fm0-mac3 |
| Traffic port | fm0-mac4 |

The odp_ l3_forwarding application needs to be provided with the following resources:

- test config xml file (defines the fman and ports used by the application)
  - odp_test_config_p4.xml
  - odp_test_config_ls1043.xml
- pcd xml file.
  - `/usr/etc/odp_pktio_pcd.xml`

### Running the ODP L3 forwarding application

In order to run this application one needs to use the following commands:

```
export DEF_CFG_PATH=<path_to_test_config_file>
export DEF_PCD_PATH=<path_to_pcd_file>


#./odp_l3_frowarding <options>
```

For additional options details, see the command line options mentioned above.

### Test Description

At startup the application will display a routing table based on the interfaces received in command line. When a packet is received on one of the input interfaces, it will be routed and forwarded according to the routing table information. The packet will be changed - destination Ethernet address will be updated according to received command line options, the TTL (Hop Limit) will be decremented and the checksum will be recalculated. Finally the packet will be transmited on the outgoing interface.

Example input command:

```
./odp_l3_forwarding -i fm1-mac1: 22.03.fa.cc.ba.cf,fm1-mac2: 8a.bb.cf.9d.55.b2
```

For this command, the output is:

```
IPv4 Routing Table:
IP source       IP dest.        Proto.  S Port  D Port  Interface
101.0.0.0       100.10.0.1      0          0       0       fm1-mac1
201.0.0.0       200.20.0.1      TCP     8       9       fm1-mac2
111.0.0.0       100.30.0.1      TCP     0       0       fm1-mac1
211.0.0.0       200.40.0.1      TCP     0       0       fm1-mac2

IPv6 Routing Table:
IP S                                                          IP
D                                                           Proto    S Port  D Port  If
fe90:0000:0000:0000:021e:67ff:fe00:0000    fe90:0000:0000:0000:021b:21ff:fe91:3801
TCP     1       2       fm1-mac1
fe90:0000:0000:0000:021e:67ff:fe00:0000    fe90:0000:0000:0000:021b:21ff:fe91:3802
TCP     2       3       fm1-mac2
fe90:0000:0000:0000:021e:67ff:fe00:0000    fe90:0000:0000:0000:021b:21ff:fe91:3803
UDP     10      20      fm1-mac1
fe90:0000:0000:0000:021e:67ff:fe00:0000    fe90:0000:0000:0000:021b:21ff:fe91:3804
TCP     4       5       fm1-mac2
```

## 47.9 Appendix A. References

PktIO - ODP PktIO API Design (multiple queues)

Buffers - ODP Buffer Management API Design

Classification - ODP Classification API Design

Crypto - ODP Crypto API Design

Packet - ODP Packet Managmenet API Design

Queues & Scheduler - ODP Queue and Scheduler API Design

Timers - ODP Timer API Design

Supported Platforms:

1. LS1043A RDB

## 47.10 Release Notes

### Conventions

This document uses the following conventions:

`Courier` Is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.

### Building Standalone DPAA1 ODP

1. Compile Linux, usdpaa, fmlib
   - Compile Linux (LS1043)

```
export SDK_ROOT=${sdk_ls1043}
source ${usdpaa_path}/standalone-env -m ls1043ardb

cd ${linux_dir}
export ARCH=arm64
export CROSS_COMPILE=aarch64-fsl-linux-
make ls1043a_defconfig
yes "" | make oldconfig
make
```

   - Compile usdpaa (LS1043)

```
export SDK_ROOT=${sdk_ls1043}
source ${usdpaa_path}/standalone-env -m ls1043ardb
cd ${usdpaa_path}
make
```

   - Compile fmlib (LS1043)

```
export SDK_ROOT=${sdk_ls1043}
source ${usdpaa_path}/standalone-env -m ls1043ardb

cd ${fmlib_path}
make KERNEL_SRC=${linux_dir} libfm-arm64a53.a
```

2. Download and compile CUnit-2.1-3(optional)
3. Compile ODP
   - (LS1043A):

```
cd ${odp_root}
    ./bootstrap
export SDK_ROOT=${sdk_ls1043}
    source ${usdpaa_path}/standalone-env -m p4080ds
```

```
#e.g ${usdpaa_path} is usually ${sdk_ls1043}/usdpaa

./configure --with-platform=linux-qoriq --host=arm-fsl-linux --with-usdpaa-path=$
{usdpaa_path} --with-fmlib-path=${fmlib_path}  [--with-cunit-path=$
{cunit_install_path}] --disable-shared
```

Compile with debug support (optional)

```
make "CFLAGS=-O0 -g"
```

Compile with no asserts and optimization (optional)

```
make "CFLAGS=-O3 -DNDEBUG=1"
```

4. Deploy ODP applications to `{$rootfs}/usr/bin/`

   - applications can be found in `{$odp_root}/example`

   **NOTE**

   ODP can be deployed running the following command:

   ```
   make install DESTDIR=<deploy_path>
   ```

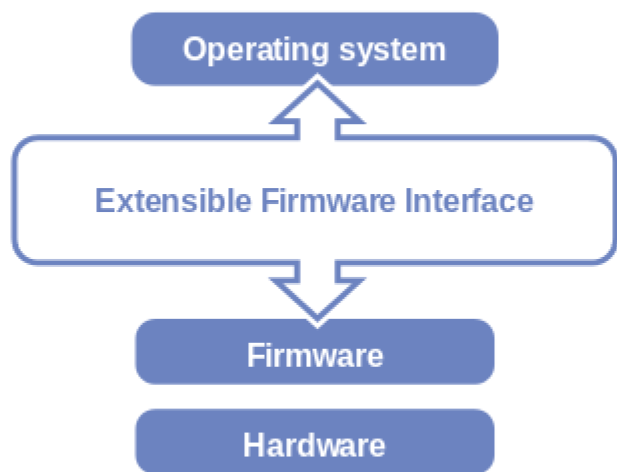
5. Deploy configurations xml files to `{$rootfs}/usr/etc/`

   - configs xml files can be found in `platform/linux-qoriq/pcd`

# Chapter 48
# UEFI

## 48.1 Introduction

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 implementatons 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. Similarly, the specification defines the set of interfaces and structures that the OS may use in booting. How either the firmware developer chooses to implement the required elements or the OS developer chooses to make use of those interfaces and structures is an implementation decision left for the developer.

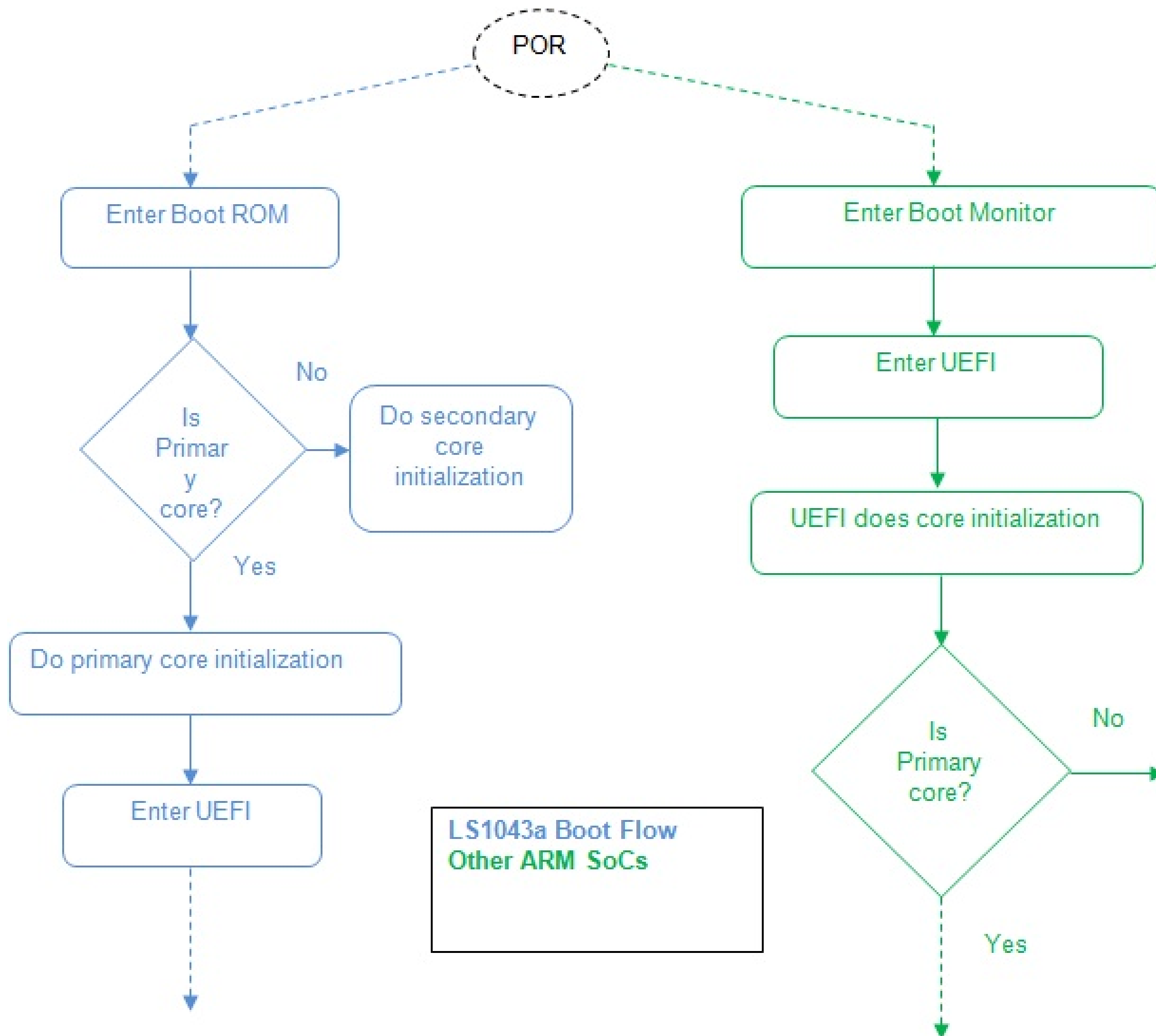
The intent of this specification is to define a way for the OS and platform firmware to communicate only information necessary to support the OS boot process. This is accomplished through a formal and complete abstract specification of the software-visible interface presented to the OS by the platform and firmware.

Using this formal definition, a shrink-wrap OS intended to run on platforms compatible with supported processor specifications will be able to boot on a variety of system designs without further platform or OS customization. The definition will also allow for platform innovation to introduce new features and functionality that enhance platform capability without requiring new code to be written in the OS boot sequence.

The specification is applicable to a full range of hardware platforms from mobile systems to servers. The specification provides a core set of services along with a selection of protocol interfaces. The latest version of UEFI specifications (UEFI specification 2.4) is available here:http://www.uefi.org/sites/default/files/resources/UEFI_2.4.pdf

### 48.1.1 UEFI Boot Flow on LS1043A

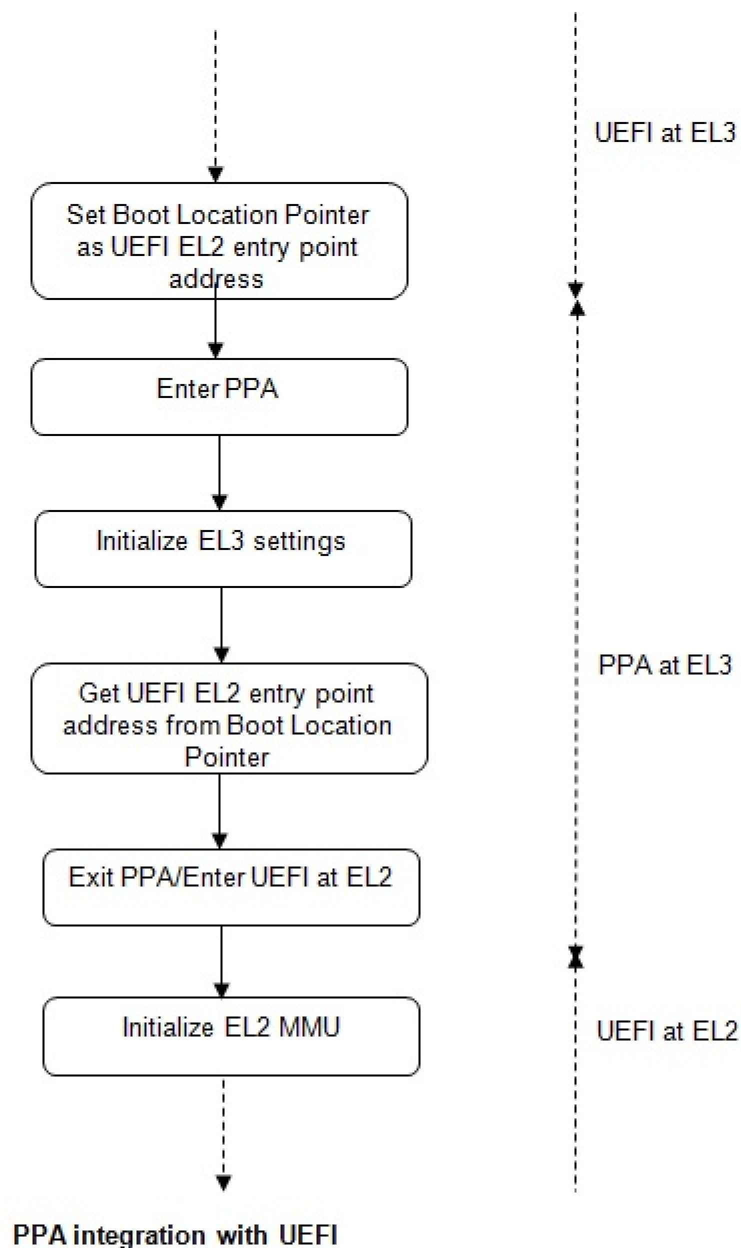
This section describes the UEFI boot flow on LS1043A platform.The LS1043a UEFI boot flow departs from other ARM SoCs boot flow, as diagrammatically presented in figure below. Note that for LS1043a, primary and secondary core boot flows diverge from the BootROM itself, while in case of other ARM SoCs the primary and secondary boot flows diverge in UEFI. So essentially, secondary cores never execute UEFI in case of LS1043a.




**UEFI Boot flow comparison b/w LS1043a and other ARM SoCs**

LS1043A UEFI starts execution at **EL3** exception level and then branches to PPA. PPA initializes the EL3 vector table and performs other EL3 settings, and then it returns the control to UEFI at EL2. Before the UEFI EL3 code transfers control to PPA it sets the Boot Location Pointer to UEFI EL2 entry point address. The PPA reads the Boot Location Pointer to get the UEFI EL2 return address and branches to that address. This is shown in the figure below.

Figure 72: PPA Integration with UEFI




## 48.1.2 Primary Protected Application (PPA)

Primary Protected Application (PPA), has the following characteristics:

- Is loaded into the secure side of an ARM core early in the boot process
- Remains resident after boot
- Provides secure services for boot sw and runtime sw
- Contains a secure monitor, which controls access to/from the secure world
- Implements services behind a std abstract interface (published by ARM)

- Has the secure world exception vectors and handlers
- Is the focal point for implementing the Platform Security Policy

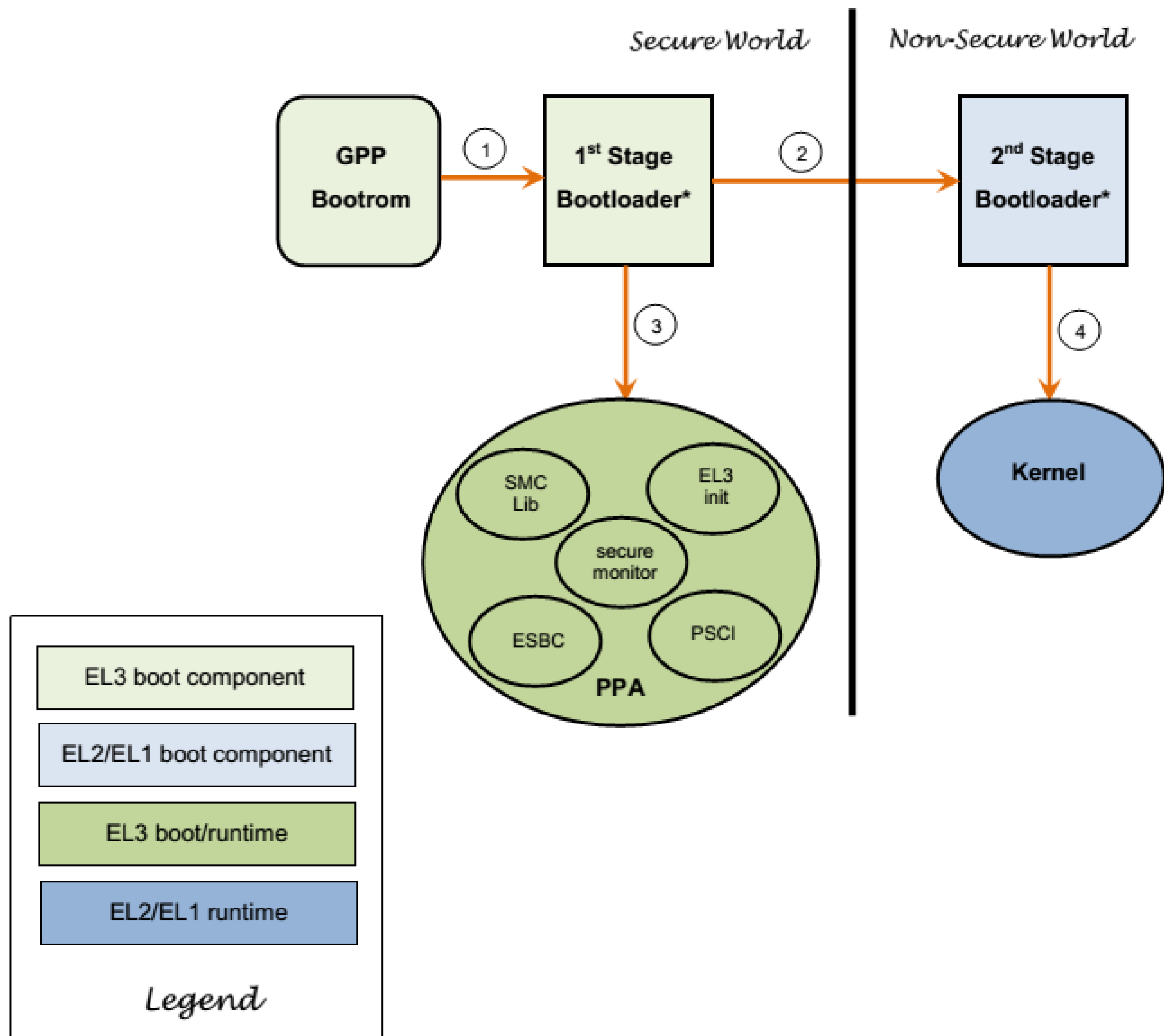
There are a number of compelling reasons for having a resident secure services layer:

1. The secure services layer is first-and-foremost a focal point for implementation of a Platform Security Policy
2. ARM cores come out of reset executing in the secure world.
3. The non-secure world needs an agent to perform tasks in the secure world
4. The PSCI interface, which is a subset of the SMC interface, is now required by ARM
5. Streamlines bootloaders, making it easier to support multiple bootloaders
6. The PPA is the foundation upon which a deeper TrustZone sw stack can be built.

### PPA Component Load Sequence

1. Bootrom loads/validates 1 st stage bootloader.
2. 1st stage bootloader loads/validates 2nd stage bootloader.
3. 1st stage bootloader loads/validates PPA.
4. 2nd stage bootloader loads/validates kernel.

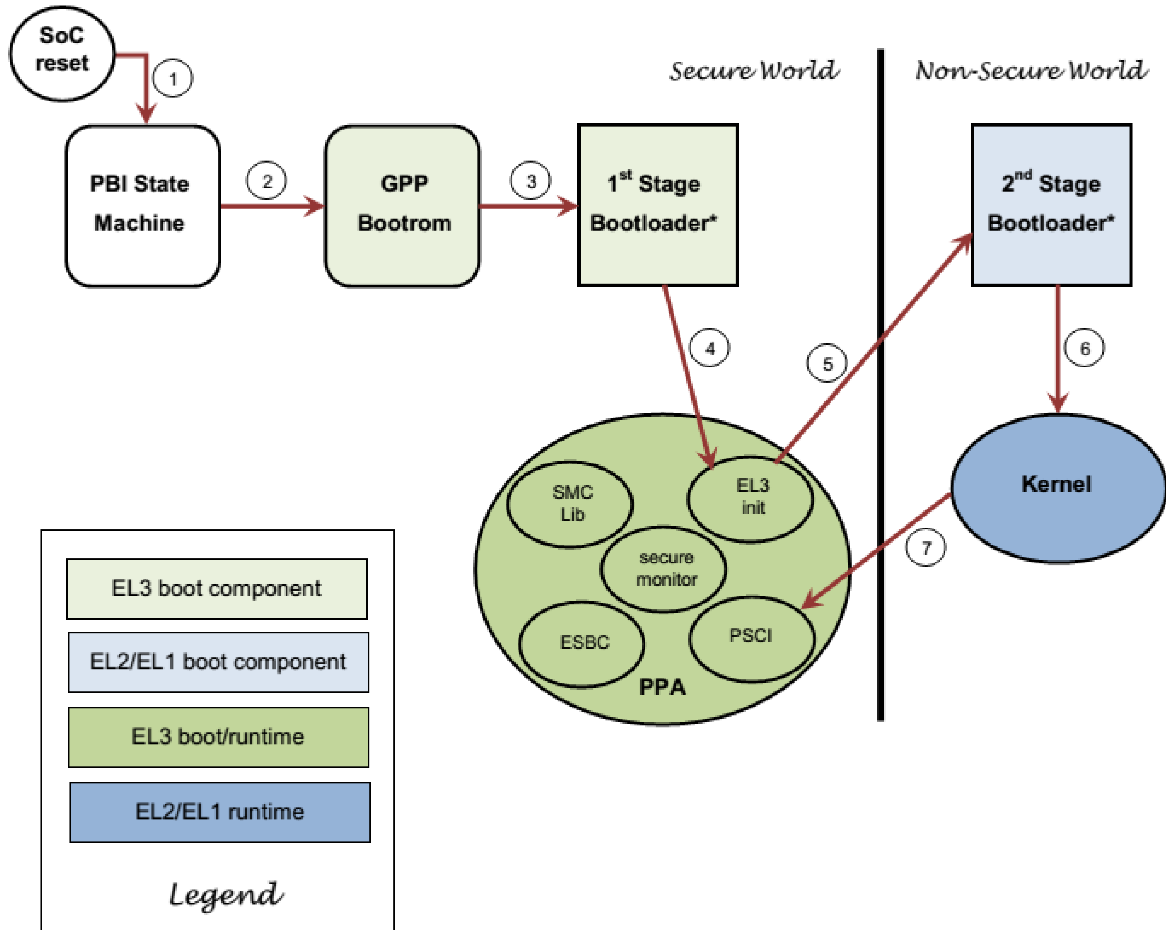
Figure 73: PPA boot flow




Boot Execution Order

1. Execution begins in the PBI State Machine when the SoC comes out of reset
2. After PBI, execution starts with bootcore in bootrom.
3. After PBI, execution starts with bootcore in GPP bootrom
4. Bootcore in 1 st stage bootloader branches to EL3 init code in PPA
5. When bootcore completes EL3 init, it branches to 2nd stage bootloader in EL2/EL1.
6. Bootcore in 2nd stage bootloader branches to Linux kernel in EL1
7. Kernel calls PSCI (cpu_on) to release secondary cores

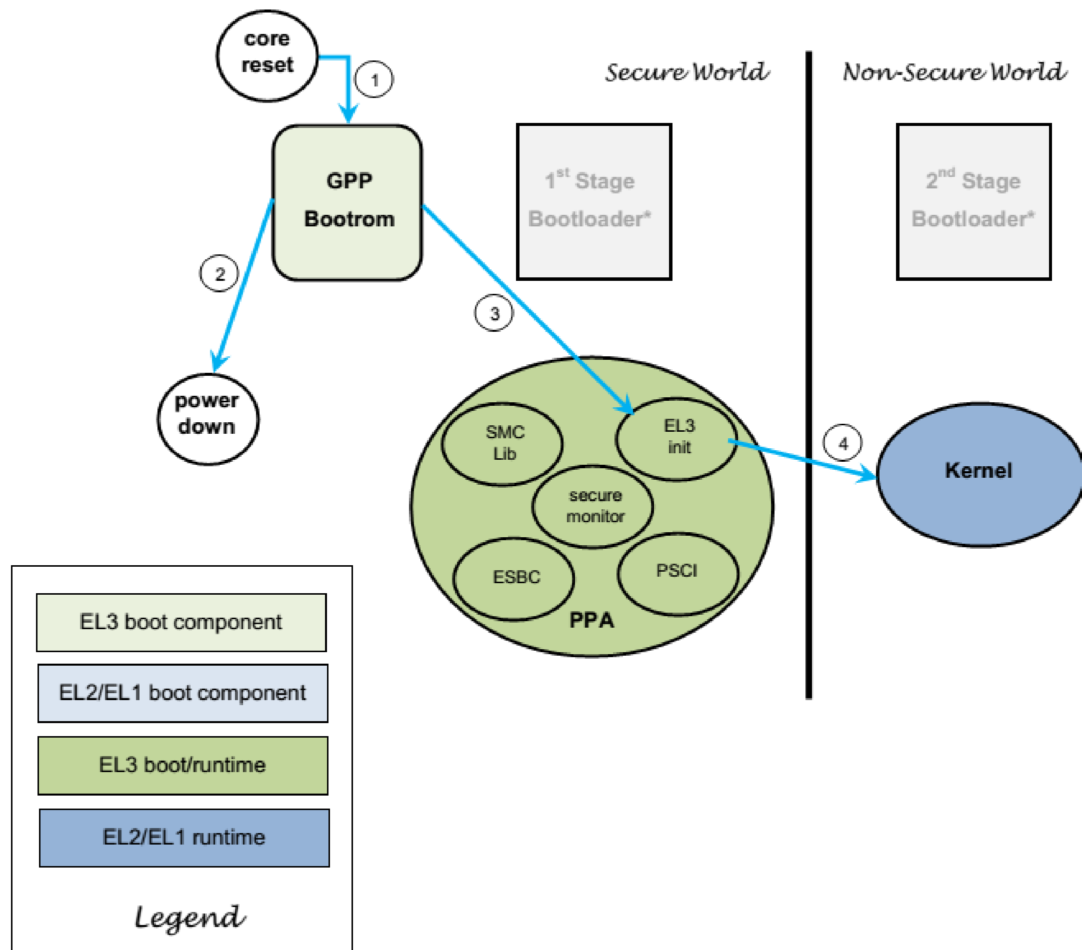
Figure 74: PPA Boot Execution Order




### Secondary Core Execution Path

1. Execution starts in the GPP bootrom when secondary core released from reset.
2. If core is marked to be disabled, core enters power-down sequence in bootrom.
3. Cores not disabled branch to EL3 init code in PPA.
4. Upon completion of EL3 init, cores branch to start address at EL1 in kernel

Figure 75: Secondary Core Execution Path




## 48.1.3 How to Compile UEFI Image in Yocto

To compile uefi in yocto for LS1043A, run the following steps:

1. `$ cd <yocto_install_path>/`
2. `$ source ./poky/fsl-setup-poky -m ls1043ardb`
3. `$ bitbake uefi-ls1043a`

NOTE

The UEFI build depends on FatPkg. FatPkg source code must been downloaded.

- add LS1043aRdbPkgFatXipBoot in `meta-fsl-arm/conf/machine/ls1043ardb.conf`

  `"UEFI_MACHINES ?= "LS1043aRdbPkgXipBoot LS1043aRdbPkgNonXipBoot LS1043aRdbPkgFatXipBoot"`
- `$ cd tmp/work/ls1043ardb-fsl-linux/uefi-ls1043a/git-r0/git`
- `$ git clone git://git.code.sf.net/p/tianocore/edk2-FatPkg`
- `$ cd edk2-FatPkg`
- `$ $ git reset --hard 8ff136aaa3fff82d81514fd3091961ec4a63c873`
- `$ mv edk2-FatPkg FatPkg`
- `$ cd FatPkg`
- `$ cp <yocto_install_patch>/meta-fsl-networking/recipes-bsp/uefi/uefi-ls1043a/0001-FatPkg-Added-changes-for-running-SCT.patch ./`
- `$ cp <yocto_install_patch>/meta-fsl-networking/recipes-bsp/uefi/uefi-ls1043a/0002-FatPkg-Enhanced-code-to-find-out-location-of-FAT-tab.patch ./`
- `$ git apply 0001-FatPkg-Added-changes-for-running-SCT.patch`
- `$ git apply 0002-FatPkg-Enhanced-code-to-find-out-location-of-FAT-tab.patch`

NOTE

you can find XIP, NONXIP and FATXIP in `<yocto_install_path>/ tmp/deploy/images/ls1043ardb/`

4. `$ bitbake ppa`

### 48.1.4 SCT Overview

The UEFI Self-Certification Test (SCT) II is a toolset for platform firmware developers to validate UEFI implementations on IA32, X64, and ARM platforms for compliance to the UEFI Specification. The toolset features a Test Harness for executing built-in EFI Compliance Tests, as well as for integrating user-defined tests that were developed using the UEFI SCT open source code.

1. Build SCT

   Run following command to compile SCT:

   ```
   $ ./SctPkg/build.sh AARCH64 ARMGCC DEBUG
   ```

   On successful compilation of SCT package, you will see following prints:

   ```
   cp: cannot stat `/proj/nmgsw_be/users/b46476/code/UEFI/ls1043a-uefi/Build/UefiSct/
   DEBUG_ARMGCC/AARCH64/SctPkg/TestInfrastructure/SCT/Framework/ENTS/Eftp/Eftp/DEBUG/
   *.pdb': No such file or directory

   make: [/proj/nmgsw_be/users/b46476/code/UEFI/ls1043a-uefi/Build/UefiSct/DEBUG_ARMGCC/
   AARCH64/SctPkg/TestInfrastructure/SCT/Framework/ENTS/Eftp/Eftp/DEBUG/Eftp.efi] Error
   1 (ignored)
   ```

```
- Done -

Build end time: 09:49:04, Aug.04 2015

Build total time: 00:02:50



Generating SCT binary

The SCT binary SctPackageARM is located at /proj/nmgsw_be/users/b46476/code/UEFI/
ls1043a-uefi/Build/UefiSct/DEBUG_ARMGCC
```

2. Copy SCT binary on SD card

   Format your SD card with FAT32 using diskpart utility (steps are mentioned at bottom of page)

   Copy SCT binaries to SD card:

   ```
   $ cp -r Build/UefiSct/DEBUG_ARMGCC/SctPackageAARCH64/* PATH_TO_SD_CARD
   ```

   SctPackage AARCH64 contains:

   - `InstallSctArm.efi`
   - `AARCH64`
   - `SctStartup.nsh`

   Rename AARCH64 to SCT

3. Run SCT:
   a. To execute SCT, insert your SD card in LS1043ARDB board.
   b. Compile UEFI using following command:

      ```
      $ ./build.sh RELEASE FATXIP
      ```

   c. Flash UEFI image `(/Build/LS1043aRdb/RELEASE_GCC48/FV/LS1043ARDB_EFI.fd)` on LS1043ARDB and boot with this image.
   d. Follow steps to select filesystem on LS1043aRDB board (mentioned above)
   e. Run SCT on FS0 using command, `"sct - a":`

## 48.1.5 How to Compile Linux image in yocto

To compile Linux in yocto for LS1043A for use with UEFI, run the following steps

1. `$ cd <yocto_install_path>/`
2. Update the `KERNEL_DEFCONFIG` variable in `meta-fsl-arm/conf/machine/<machine>.conf`
   - `KERNEL_DEFCONFIG ?= "${S}/arch/arm64/configs/ls1043a_uefi_defconfig "`
3. `Use kernel-ls1043a-uefi-rdb.its instead of kernel-rdb.its in meta-fsl-networking/images/fsl-image-kernelitb/`
   - `$ cp kernel-ls1043a-uefi-rdb.its meta-fsl-networking/images/fsl-image-kernelitb/kernel-rdb.its`

NOTE
you can find kernel-ls1043a-uefi-rdb.its in linux source code

4. Modify fsl-image-kernelitb.bb in meta-fsl-networking/images recipes
   - `sed -i -e "s,./arch/arm64/boot/dts/freescale/fsl-ls1043a-uefi-rdb.dtb,${DTB_IMAGE}," ${S}/kernel.its`
   - `sed -i -e "s,./arch/arm64/boot/Image,${KERNEL_IMAGE}," ${S}/kernel.its`
5. Rebuild images:
   - `$ cd <yocto_install_path>/build_<machine>_release`
   - `$ bitbake -c cleansstate fsl-image-kernelitb`
   - `$ bitbake fsl-image-kernelitb`

# 48.2 LS1043A UEFI Hardware

## 48.2.1 Switch Settings

The RDB has user selectable switches for evaluating different boot options for the LS1043A device. Table below lists the default switch settings and the description of these settings.

Table 93: Default Clock Frequency

| ARM CPU Core | Platform | DDR rate |
|---|---|---|
| 1000 MHZ | 400 MHZ | 1600 MT/S |

Table 94: SWITCH Initialization

| | |
|---|---|
| SW3[1-8]: | 0b'10010011 |
| SW4[1-8]: | 0b'00010010 |
| SW5[1-8]: | 0b'10100000 |

Table 95: SWITCH Setting

| Boot source | SWITCH |
|---|---|
| NOR(bank0) | SW4[1-8] +SW5[1] = 0b'00010010_1 SW5[4-6]= 0b'000 |
| NOR(bank4) | SW4[1-8] +SW5[1] = 0b'00010010_1 SW5[4-6]= 0b'001 |
| NAND | SW4[1-8] +SW5[1] = 0b'10000011_0 |
| SD | SW4[1-8] +SW5[1] = 0b'00100000_0 |

RCW used with fields details:

LS1043AQDS RCW for SerDes Protocol 0x1455

15G configuration -- 2 RGMII + 1 QSGMII + 1 XFI

Frequencies:

- Sys Clock: 100 MHz
- DDR_Refclock: 100 MHz

- Core -- 1400 MHz (Mul 14)
- Platform -- 300 MHz (Mul 3)
- DDR -- 800 MHz (Mul 8)
- FMan -- 500 MHz (CGA2 /2)
- XFI -- 156.25 MHz (10.3125G)
- QSGMII -- 100 MHz (5G)
- PCIE -- 100 MHz (5G)
- eSDHC -- 1000 MHz (CGA2 /1)

Serdes Lanes vs Slot information

- A XFI : Slot 1
- B QSGMII : Slot 1
- C PCIe2 : Slot 2
- D PCIe3 : Slot 3

**Serdes configuration**

SRDS_PRTCL_S1 : 0x1455

**SRDS_PLL_REF_CLK_SEL_S1 :**

SerDes 1, PLL1 : 1 - 156.25MHz for XFI

SerDes 1, PLL2 : 0 - 100MHz for QSGMII and PCIe

SRDS_DIV_PEX : 00 Can train up to a max rate of 5G

**DDR clock:**

DDR_REFCLK_SEL : 0 - DDRCLK pin provides the reference clock to the DDR PLL

## 48.2.2 RCW (Reset Configuration Word) and Ethernet Interfaces

The following RCW binary for use on the ls1043ardb:

- RR_FQPP_1455/rcw_1500.bin
- RR_FQPP_1455/rcw_1500_qetdm.bin
- RR_FQPP_1455/rcw_1500_sben.bin

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

```
ab_cdef_g
```

Table 96: ls1043ardb Directories Naming Convention Legend

| Slot | Convention |
|---|---|
| a[What is available for DTSEC3] | 'R' indicates RGMII1@DTSEC3 is supported<br>'N' if not available/not used |
| b[What is available for DTSEC4] | 'R' indicates RGMII2@DTSEC4 is supported<br>'N' if not available/not used |
| c | 'F' indicates XFI is available<br>'N' is NULL, not available/not used |
| d | 'Q' indicates QSGMII is available<br>'N' is NULL, not available/not used |
| e | 'P' indicates PCIe@slot1 is supported<br>'N' is NULL, not available/not used |
| f | 'P'includes PCIe@slot2 is supported<br>'N' is NULL, not available/not used |
| g | Hex value of serdes1 protocol value |

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 97: ls1043ardb Files Naming Convention Legend

| Code | Convention |
|---|---|
| frequency | Core frequency(MHZ) |

*Table continues on the next page...*

Table 97: ls1043ardb Files Naming Convention Legend (continued)

| | | |
|---|---|---|
| specialsetting | bootmode | SD/NAND/NOR and so on |
| | special support | nand: Nand boot<br>sben: Secure 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 Of 1500MHz with Nor
boot.
```

Default rcw:

```
• RR_FQPP_1455/rcw_1500.bin[2 RGMII, 1 XFI, 1 QSGMII, 2 PCIE]
```

## 48.2.3 LS1043A NOR Flash memory map for UEFI Bootloader

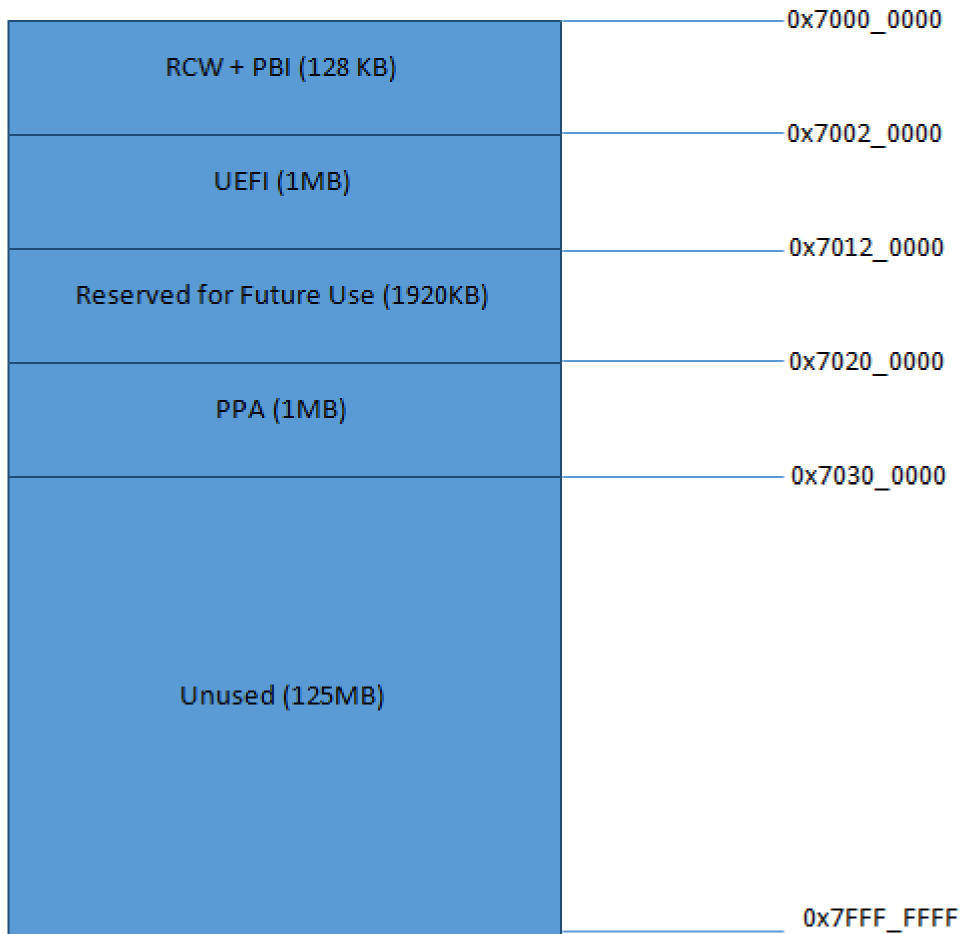
| S.No | Start Address | End Address | Field Name | Size |
|---|---|---|---|---|
| 1 | 0x0_6000_0000 | 0x0_600F_FFFF | RCW + PBI | 1 MB |
| 2 | 0x0_6010_0000 | 0x0_601F_FFFF | U-Boot | 1 MB |
| 3 | 0x0_6020_0000 | 0x0_602F_FFFF | U-Boot Env | 1 MB |
| 4 | 0x0_6030_0000 | 0x0_603F_FFFF | FMan ucode | 1 MB |
| 5 | 0x0_6040_0000 | 0x0_604F_FFFF | UEFI | 1 MB |
| 6 | 0x0_6050_0000 | 0x0_605F_FFFF | PPA | 1 MB |
| 7 | 0x0_6060_0000 | 0x0_606F_FFFF | QE firmware | 1 MB |
| 8 | 0x0_6070_0000 | 0x0_60EF_FFFF | Reserved | 8 MB |
| 9 | 0x0_60F0_0000 | 0x0_60FF_FFFF | PHY firmware | 1 MB |
| 10 | 0x0_6100_0000 | 0x0_610F_FFFF | CORTINA PHY firmware | 1 MB |
| 11 | 0x0_6110_0000 | 0x0_638F_FFFF | FIT Image | 40 MB |
| 12 | 0x0_6390_0000 | 0x0_63FF_FFFF | Unused | 7 MB |
| 13 | 0x0_6400_0000 | 0x0_640F_FFFF | RCW + PBI | 1 MB |
| 14 | 0x0_6410_0000 | 0x0_641F_FFFF | U-Boot | 1 MB |
| 15 | 0x0_6420_0000 | 0x0_642F_FFFF | U-Boot Env | 1 MB |
| 16 | 0x0_6430_0000 | 0x0_643F_FFFF | FMan ucode | 1 MB |

*Table continues on the next page...*

*Table continued from the previous page...*

| | | | | |
|---|---|---|---|---|
| 17 | 0x0_6440_0000 | 0x0_644F_FFFF | UEFI | 1 MB |
| 18 | 0x0_6450_0000 | 0x0_645F_FFFF | PPA | 1 MB |
| 19 | 0x0_6460_0000 | 0x0_646F_FFFF | QE firmware | 1 MB |
| 20 | 0x0_6470_0000 | 0x0_64EF_FFFF | Reserved | 8 MB |
| 21 | 0x0_64F0_0000 | 0x0_64FF_FFFF | PHY firmware | 1 MB |
| 22 | 0x0_6500_0000 | 0x0_650F_FFFF | CORTINA PHY firmware | 1 MB |
| 23 | 0x0_6510_0000 | 0x0_678F_FFFF | FIT Image | 40 MB |
| 24 | 0x0_6790_0000 | 0x0_67FF_FFFF | Unused | 7 MB |

### 48.2.3.1 LS1043A NAND Flash memory map for UEFI Bootloader



# 48.3 Booting UEFI

## 48.3.1 Booting to UEFI prompt from NOR flash:

**Booting to UEFI prompt on LS1043ARDB Board:**

1. Boot to U-Boot prompt from NOR flash bank 0 on LS1043ARDB.
   - Setup serial port connection on host machine, to capture logs from the target LS1043a RDB board..
   - Reset the board to boot u-boot on bank 0, assuming that there is a valid u-boot image flashed on the primary bank 0.

```
U-Boot 2015.01QorIQ-SDK-V1.7+g7063f8a (Jun 18 2015 - 22:03:08)

Clock Configuration:
       CPU0(A53):1500 MHz  CPU1(A53):1500 MHz  CPU2(A53):1500 MHz
       CPU3(A53):1500 MHz
       Bus:      400  MHz  DDR:      1600 MHz
Reset Configuration Word (RCW):
       00000000: 0810000f 0c000000 00000000 00000000
       00000010: 14550002 80004012 e0025000 61002000
       00000020: 00000000 00000000 00000000 00038800
       00000030: 00000000 00001100 00000096 00000001
Board: LS1043ARDB, boot from vBank 0
CPLD:  V1.4
PCBA:  V2.0
SERDES Reference Clocks:
SD1_CLK1 = 156.25MHZ, SD1_CLK2 = 100.00MHZ
I2C:   ready
DRAM:  2 GiB (DDR4, 32-bit, CL=12, ECC off)
Waking secondary cores to start from fff2a000
All (4) cores are up.
Using SERDES1 Protocol: 5205 (0x1455)
fman_port_enet_if:71: port(FM1_DTSEC3) is OK
fman_port_enet_if:77: port(FM1_DTSEC4) is OK
Flash: 128 MiB
NAND:  512 MiB
MMC:   FSL_SDHC: 0
EEPROM: Invalid ID (00 5a 5a 5a)
PCIe1: disabled
PCIe2: Root Complex no link, regs @ 0x3500000
PCIe3: Root Complex x1 gen1, regs @ 0x3600000
     01:00.0    - 8086:10d3 - Network controller
PCIe3: Bus 00 - 01
In:    serial
Out:   serial
Err:   serial
Net:   Fman1: Uploading microcode version 106.4.15
e1000: 68:05:ca:0f:23:a5
       FM1@DTSEC1, FM1@DTSEC2, FM1@DTSEC3, FM1@DTSEC4, FM1@DTSEC5, FM1@DTSEC6, F
M1@TGEC1, e1000#0 [PRIME]
Warning: e1000#0 MAC addresses don't match:
Address in SROM is         68:05:ca:0f:23:a5
Address in environment is  68:05:ca:12:91:44

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

2. Copy Images to NOR flash alternate bank using U-Boot commands.

   - sete uefi 'tftp 80000000 LS1043ARDB_EFI.fd; erase 0x64400000 0x644FFFFF ; cp.b 80000000 0x64400000 $filesize'
   - sete rcw 'tftp 80000000 rcw_uefi_1500.bin; erase 0x64000000 0x640FFFFF ; cp.b 80000000 0x64000000 $filesize;'
   - sete ppa 'tftp 80000000 ppa.itb; erase 0x64500000 0x645FFFFF ; cp.b 80000000 0x64500000 $filesize;'

- sete linux 'tftp 80000000 kernel.itb; erase 0x65100000 0x0678FFFFF ; cp.b 80000000 0x65100000 $filesize;'
- run uefi
- run ppa
- run rcw
- run linux

**NOTE**

The host machine is assumed to be having tftp server running, with the relevant files in place. The rcw, uefi, linux and ppa images can also be found at compass link shared above.

3. Reset RDB to boot from NOR flash bank 4.

```
=> cpld reset altbank
```

**XIP Build Boot logs:**

```
=> cpld reset altbank
UEFI firmware (version  built at 02:21:14 on Aug  5 2015)
Clock Configuration:CPU0(A53):1400 MHz  CPU1(A53):1400 MHz  CPU2(A53):1400 MHz
CPU3(A53):1400 MHz
       Bus:      300  MHz  DDR:      1600 MHz
Reset Configuration Word (RCW):
       00000000: 0610000E 0A000000 00000000 00000000
       00000010: 14550002 80004002 E0025000 C1002400
       00000020: 00000000 00000000 00000000 00038800
       00000030: 00000000 00001100 00000096 00000001
Cfg->SdhcClk 1000000000
Not MMC Card
Device: FSL_SDXC
Manufacturer ID: 3
OEM: 5344
Name: SU16G
Card: 3
Product revision: 0.5
Product Serial No. : 561D4200
Manufacturing date : 13:1999
Tran Speed: 25000000
Rd Block Len: 512
SD Version 2.0
High Capacity: Yes
Capacity: 14.8 GiB
Bus Width: 1-Bit
Valid Chip Addresses :
0x8 0x4C 0x52 0x53 0x69
EEPROM0: 0x55 0x55 0x55  0x55  0xAA 0xAA 0xAA 0xAA  0x0  0x0
I2c Test Result: PASS
Cs0Bnds = 0x7F
Cs1Bnds = 0x0
Cs2Bnds = 0x0
Cs3Bnds = 0x0
Cs0Config = 0x80010322
Cs1Config = 0x0
Cs2Config = 0x0
Cs3Config = 0x0
Cs0Config2 = 0x0
Cs1Config2 = 0x0
Cs2Config2 = 0x0
Cs3Config2 = 0x0
TimingCfg3 = 0x20C1000
TimingCfg0 = 0xD0550018
TimingCfg1 = 0xC2C68C42
TimingCfg2 = 0x48C114
SdramCfg = 0xC50C000C
SdramCfg2 = 0x401000
SdramMode = 0x1010214
SdramMode2 = 0x0
SdramMdCntl = 0x600041F
SdramInterval = 0x18600618
SdramDataInit = 0x0
SdramClkCntl = 0x2000000
InitAddr = 0x0
InitExtAddr = 0x0
TimingCfg4 = 0x2
TimingCfg5 = 0x4401400
TimingCfg6 = 0x0
TimingCfg7 = 0x13300000
DdrZqCntl = 0x8A090705
DdrWrlvlCntl = 0xC655F606
DdrSrCntr = 0x0
DdrSdramRcw1 = 0x0
```

```
DqMap1 = 0x0
DqMap2 = 0x0
DqMap3 = 0x0
DdrDsr1 = 0x0
DdrDsr2 = 0x0
DdrCdr1 = 0x80040000
DdrCdr2 = 0xA181
IpRev1 = 0x20501
IpRev2 = 0x200
Eor = 0x0
Mtcr = 0x0
Mtp1 = 0x0
Mtp2 = 0x0
Mtp3 = 0x0
Mtp4 = 0x0
Mtp5 = 0x0
Mtp6 = 0x0
Mtp7 = 0x0
Mtp8 = 0x0
Mtp9 = 0x0
Mtp10 = 0x0
DataErrInjectHi = 0x0
DataErrInjectLo = 0x0
EccErrInject = 0x0
CaptureDataHi = 0x0
CaptureDataLo = 0x0
CaptureEcc = 0x0
ErrDetect = 0x0
ErrDisable = 0x0
ErrIntEn = 0x0
CaptureAttributes = 0x0
CaptureAddress = 0x0
CaptureExtAddress = 0x0
ErrSbe = 0x0

NorFlashBlockIoWriteBlocks(MediaId=0x0, Lba=32, BufferSize=0x20000 bytes (-53816
9344 kB), BufferPtr @ 0xFFC08B00)
NorFlashBlockIoReadBlocks(MediaId=0x0, Lba=32, BufferSize=0x20000 bytes (-538169
344 kB), BufferPtr @ 0xFFC00A88)
Nor Test Result: PASS
Nand Test Result: PASS
The default boot selection will start in  10 seconds
[1] Linux FIT from NOR
        - MemoryMapped(0x0,0x61100000,0x63000000)
        - Arguments: console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,0x21c
0500,115200
        - LoaderType: Linux kernel with FDT support, using FIT image
[2] Shell
[3] Boot Manager
Start: 2
UEFI Interactive Shell v2.1
EDK II
UEFI v2.40 (LS1043a Simulator EFI Aug  5 2015 02:23:02, 0x00000000)
Mapping table
      FS0: Alias(s):F2:;BLK0:
          VenHw(1F15DA3C-37FF-4070-B471-BB4AF12A724A)
      FS1: Alias(s):F3:;BLK1:
          VenHw(4D00EF14-C4E0-426B-81B7-30A00A14AAD6)
      FS2: Alias(s):F5:;BLK2:
          VenHw(B6F44CC0-9E45-11DF-BE21-0002A5D5C51B)
Press ESC in 5 seconds to skip startup.nsh or any other key to continue.Press ES
C in 4 seconds to skip startup.nsh or any other key to continue.Press ESC in 3 s
econds to skip startup.nsh or any other key to continue.
Shell>
```

Linux Boot Logs:

```
NorFlashBlockIoWriteBlocks(MediaId=0x0, Lba=32, BufferSize=0x20000 bytes (-53816
9344 kB), BufferPtr @ 0xFFC08B00)
NorFlashBlockIoReadBlocks(MediaId=0x0, Lba=32, BufferSize=0x20000 bytes (-538169
344 kB), BufferPtr @ 0xFFC00A88)
Nor Test Result: PASS
Nand Test Result: PASS
The default boot selection will start in  10 seconds
[1] Linux FIT from NOR
        - MemoryMapped(0x0,0x61100000,0x63000000)
        - Arguments: console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,0x21c
0500,115200
        - LoaderType: Linux kernel with FDT support, using FIT image
[2] Shell
[3] Boot Manager
Start: 1
Loading linux at 0x80080000
Loading initrd at 0x86F79000
ConvertPages: Incompatible memory types
Loading FDT at 0x86F72000
Booting Linux on physical CPU 0x0
Initializing cgroup subsys cpu
Linux version 3.19.3+ (root@b45370) (gcc version 4.8.3 20140401 (prerelease) (cr
osstool-NG linaro-1.13.1-4.8-2014.04 - Linaro GCC 4.8-2014.04) ) #1 SMP PREEMPT
Tue Aug 4 14:52:57 IST 2015
CPU: AArch64 Processor [410fd034] revision 4
Detected VIPT I-cache on CPU0
Early serial console at MMIO 0x21c0500 (options '115200')
bootconsole [uart0] enabled
efi: Getting EFI parameters from FDT:
efi: UEFI not found.
cma: Reserved 16 MiB at 0x00000000fb000000
psci: probing for conduit method from DT.
psci: PSCIv0.2 detected in firmware.
psci: Using standard PSCI v0.2 function IDs
PERCPU: Embedded 13 pages/cpu @ffffffc07ff95000 s13376 r8192 d31680 u53248
Built 1 zonelists in Zone order, mobility grouping on.  Total pages: 516096
Kernel command line: console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,0x21c
0500,115200
log_buf_len individual max cpu contribution: 4096 bytes
log_buf_len total cpu_extra contributions: 12288 bytes
log_buf_len min size: 16384 bytes
log_buf_len: 32768 bytes
early log buf free: 14720(89%)
PID hash table entries: 4096 (order: 3, 32768 bytes)
Dentry cache hash table entries: 262144 (order: 9, 2097152 bytes)
Inode-cache hash table entries: 131072 (order: 8, 1048576 bytes)
Memory: 1965552K/2097152K available (5309K kernel code, 424K rwdata, 1888K rodat
a, 260K init, 189K bss, 115216K reserved, 16384K cma-reserved)
Virtual kernel memory layout:
    vmalloc : 0xffffff8000000000 - 0xffffffbdbfff0000   (   246 GB)
    vmemmap : 0xffffffbdc0000000 - 0xffffffbfc0000000   (     8 GB maximum)
              0xffffffbdc2000000 - 0xffffffbdc4000000   (    32 MB actual)
    PCI I/O : 0xffffffbffa000000 - 0xffffffbffb000000   (    16 MB)
    fixed   : 0xffffffbffbdfd000 - 0xffffffbffbdff000   (     8 KB)
    modules : 0xffffffbffc000000 - 0xffffffc000000000   (    64 MB)
    memory  : 0xffffffc000000000 - 0xffffffc080000000   (  2048 MB)
      .init : 0xffffffc00078a000 - 0xffffffc0007cb000   (   260 KB)
      .text : 0xffffffc000080000 - 0xffffffc000789534   (  7206 KB)
      .data : 0xffffffc0007d3000 - 0xffffffc00083d200   (   425 KB)
SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=4, Nodes=1
Preemptible hierarchical RCU implementation.
        RCU restricting CPUs from NR_CPUS=64 to nr_cpu_ids=4.
RCU: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=4
NR_IRQS:64 nr_irqs:64 0
```

```
Architected cp15 timer(s) running at 25.00MHz (phys).
sched_clock: 56 bits at 25MHz, resolution 40ns, wraps every 2748779069440ns
Console: colour dummy device 80x25
Calibrating delay loop (skipped), value calculated using timer frequency.. 50.00
 BogoMIPS (lpj=250000)
pid_max: default: 32768 minimum: 301
Security Framework initialized
Mount-cache hash table entries: 4096 (order: 3, 32768 bytes)
Mountpoint-cache hash table entries: 4096 (order: 3, 32768 bytes)
Initializing cgroup subsys memory
Initializing cgroup subsys hugetlb
hw perfevents: enabled with arm/armv8-pmuv3 PMU driver, 7 counters available
EFI services will not be available.
CPU1: Booted secondary processor
Detected VIPT I-cache on CPU1
CPU2: Booted secondary processor
Detected VIPT I-cache on CPU2
CPU3: Booted secondary processor
Detected VIPT I-cache on CPU3
Brought up 4 CPUs
SMP: Total of 4 processors activated.
devtmpfs: initialized
DMI not present or invalid.
NET: Registered protocol family 16
vdso: 2 pages (1 code @ ffffffc0007d9000, 1 data @ ffffffc0007d8000)
hw-breakpoint: found 6 breakpoint and 4 watchpoint registers.
software IO TLB [mem 0xf8800000-0xf8c00000] (4MB) mapped at [ffffffc078800000-ff
ffffc078bfffff]
DMA: preallocated 256 KiB pool for atomic allocations
Serial: AMBA PL011 UART driver
irq: no irq domain found for /uqe@2400000/qeic@80 !
irq: no irq domain found for /uqe@2400000/qeic@80 !
vgaarb: loaded
SCSI subsystem initialized
usbcore: registered new interface driver usbfs
usbcore: registered new interface driver hub
usbcore: registered new device driver usb
i2c i2c-0: IMX I2C adapter registered
i2c i2c-0: using dma0chan16 (tx) and dma0chan17 (rx) for DMA transfers
pps_core: LinuxPPS API ver. 1 registered
pps_core: Software ver. 5.3.6 - Copyright 2005-2007 Rodolfo Giometti <giometti@l
inux.it>
PTP clock support registered
fsl-ifc 1530000.ifc: Freescale Integrated Flash Controller
fsl-ifc 1530000.ifc: IFC version 1.4, 8 banks
Switched to clocksource arch_sys_counter
NET: Registered protocol family 2
TCP established hash table entries: 16384 (order: 5, 131072 bytes)
TCP bind hash table entries: 16384 (order: 6, 262144 bytes)
TCP: Hash tables configured (established 16384 bind 16384)
TCP: reno registered
UDP hash table entries: 1024 (order: 3, 32768 bytes)
UDP-Lite hash table entries: 1024 (order: 3, 32768 bytes)
NET: Registered protocol family 1
RPC: Registered named UNIX socket transport module.
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
RPC: Registered tcp NFSv4.1 backchannel transport module.
Trying to unpack rootfs image as initramfs...
rootfs image is not initramfs (no cpio magic); looks like an initrd
Freeing initrd memory: 16920K (ffffffc006f79000 - ffffffc007fff000)
kvm [1]: HYP mode not available
futex hash table entries: 1024 (order: 4, 65536 bytes)
audit: initializing netlink subsys (disabled)
```

```
INIT: Entering runlevel: 5
Configuring network interfaces... udhcpc (v1.22.1) started
Sending discover...
e1000e: eth0 NIC Link is Up 100 Mbps Full Duplex, Flow Control: Rx/Tx
e1000e 0001:01:00.0 eth0: 10/100 speed: disabling TSO
Sending discover...
Sending select for 192.168.0.214...
Lease of 192.168.0.214 obtained, lease time 3600
/etc/udhcpc.d/50default: Adding DNS 192.168.1.1
done.
Starting system message bus: dbus.
Starting Dropbear SSH server: Generating key, this may take a while...
Public key portion is:
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCkt6GsZQ7sUHJsJnJ1uX9QenFDn77FISpSRI3byylc
MKhXbsEvYtC4N+luDkDrZO8jwB4gVHhQ1Eez+ZxJVU702JJV1rNFdrp/xs40bn7CvaiYUqEyy266cM7y
+bUNn2l4mZ4noW/TX03OSfIOFPhZdcH4oH3ulcA2WDxnzUP/hThhGUelJefqadVQKQgBqPZLDoYcJNDr
VqmKIDR/ISphAD8z5eS4dje4ZwIqTG2hYJM38pGtgwmqz3XSzlF3zntpw5ZQ/PKRxoVd0smLGu4SOc5J
wJvOL/jNdTF3cxE055eCy4erEqsCEn3CBNUZPin8YvTkUO8sHGQU9/3AvzLL root@ls1043ardb
Fingerprint: md5 1d:d0:7e:91:e1:4d:a6:df:b1:aa:89:67:4b:cc:38:5f
dropbear.
Starting network benchmark server: netserver.
Starting system log daemon...0
Starting kernel log daemon...0
Stopping Bootlog daemon: bootlogd.

Poky (Yocto Project Reference Distro) 1.6.1 ls1043ardb /dev/ttyS0

ls1043ardb login: root
root@ls1043ardb:~# random: nonblocking pool is initialized

root@ls1043ardb:~# cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  1:          0          0          0          0       GIC  29  arch_timer
  2:      45564        614        857        723       GIC  30  arch_timer
 14:          0          0          0          0       GIC 148  MSI1
 15:          0          0          0          0       GIC 158  MSI2
 16:        246          0          0          0       GIC 192  MSI3
 17:          0          0          0          0       GIC  75  fsl-ifc
 23:          7          0          0          0       GIC  96  2100000.dspi
 24:         51          0          0          0       GIC  88  2180000.i2c
 25:        228          0          0          0       GIC  86  serial
 28:          0          0          0          0       GIC 118  29d0000.ftm0
 30:          0          0          0          0       GIC 135  eDMA
 31:          1          0          0          0       GIC  92  xhci-hcd:usb1
 32:          0          0          0          0       GIC  93  xhci-hcd:usb3
 33:          0          0          0          0       GIC  95  xhci-hcd:usb5
 34:          0          0          0          0       GIC 101  3200000.sata
 61:        241          0          0          0      MSI3   0  eth0-rx-0
 62:          3          0          0          0      MSI3   1  eth0-tx-0
 63:          2          0          0          0      MSI3   2  eth0
 64:          2          0          0          0       GIC 103  1710000.jr
 65:          0          0          0          0       GIC 104  1720000.jr
 66:          0          0          0          0       GIC 105  1730000.jr
 67:          0          0          0          0       GIC 106  1740000.jr
IPI0:       924       1493        922        673       Rescheduling interrupts
IPI1:         3          8          8         10       Function call interrupts
IPI2:         0          2          0          1       Single function call inte
rrupts
IPI3:         0          0          0          0       CPU stop interrupts
IPI4:         0          0          0          0       Timer broadcast interrupt
s
IPI5:         0          0          0          0       IRQ work interrupts
Err:          0
root@ls1043ardb:~# DdrSdramRcw1
```

## 48.3.2 Booting to UEFI prompt via NAND flash

1. Boot to U-Boot prompt from NAND flash.

2. Erase NAND flash chip.

```
=> nand erase.chip
```

3. Copy boot images to NAND flash

```
=> tftp 80000000 LS1043ARDBPI_EFI.pbl;nand write 80000000 0 20000
=> tftp 80000000 LS1043ARDB_EFI.fd;nand write 80000000 20000 a0000
```

4. Copy PPA FIT image to NAND flash

```
=> tftp 80000000 ppa.itb;nand write 80000000 200000 20000
```

5. Switch the board off. Change switch settings to the following and switch the board on.

   SW4[1:8] = 10000011

   SW5[1] = 0

NONXIP NAND boot logs :

```
UEFI primary boot firmware (built at 16:49:49 on Aug  4 2015)

UEFI firmware (version  built at 14:37:25 on Aug  4 2015)
Clock Configuration:CPU0(A53):1500 MHz  CPU1(A53):1500 MHz
       Bus:      400  MHz  DDR:      1600 MHz
Reset Configuration Word (RCW):
       00000000: 0810000F 0C000000 00000000 00000000
       00000010: 14550002 80004012 E0106000 61002000
       00000020: 00000000 00000000 00000000 00038800
       00000030: 00000000 00001100 00000096 00000001
Cfg->SdhcClk 1200000000
Not MMC Card
Device: FSL_SDXC
Manufacturer ID: 74
OEM: 4A45
Name: SDC
Card: 2
Product revision: 0.0
Product Serial No. : 0012AE00
Manufacturing date : 11:2004
Tran Speed: 25000000
Rd Block Len: 512
SD Version 2.0
High Capacity: Yes
Capacity: 3.7 GiB
Bus Width: 1-Bit
Valid Chip Addresses :
0x0 0x8 0x40 0x4C 0x52 0x53 0x68 0x69
```

```
The default boot selection will start in  10 seconds
[1] Linux FIT from NOR
        - MemoryMapped(0x0,0x65100000,0x67000000)
        - Arguments: console=ttyS0,115200 root=/dev/ram0 earlycon=uart8250,0x21c
0500,115200
        - LoaderType: Linux kernel with FDT support, using FIT image
[2] Shell
[3] Boot Manager
Start: 2
UEFI Interactive Shell v2.0
EDK II
UEFI v2.40 (LS1043a Simulator EFI Aug  4 2015 14:38:25, 0x00000000)
Mapping table
      FS0: Alias(s):F2:;BLK0:
          VenHw(4D00EF14-C4E0-426B-81B7-30A00A14AAD6)
      FS1: Alias(s):F4:;BLK1:
          VenHw(B6F44CC0-9E45-11DF-BE21-0002A5D5C51B)
Press ESC in 5 seconds to skip startup.nsh or any other key to continue.
Shell>
```

**NOTE**

NAND boot may get stuck at the first reset after writing UEFI and PPA images on to NAND flash. This is a known issue and under debug. A workaround for this is to reset the board in NOR boot mode only just after flashing NAND boot images. Then make the switch settings to switch to NAND boot mode and reset again.

## 48.4 LS1043A UEFI Software

### 48.4.1 FAT32 Filesystem

The EDK II UEFI FAT32 File System Driver is based on Microsoft's FAT32 File System Driver Specification (http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx).

#### Downloading EDK2 FAT Driver

The EDK II Fat Driver is separate from the EDK II project only because the terms of use of the code are unique, requiring to keep it in a separate code repository (the terms are basically that code developed using the FAT32 Specification must be associated with EFI). The EDK II Fat Driver is structured as a Package that fits into the EDK II's overall build and packaging system.

You can download the EDK2 FAT driver from this link: https://github.com/tianocore/edk2-FatPkg, using the steps mentioned below.

**NOTE**

The latest FAT driver changes are not supported with this release, so the user must use the following commit as the HEAD:

```
commit 8ff136aaa3fff82d81514fd3091961ec4a63c873
Author: Ruiyu Ni <ruiyu.ni@intel.com>
Date:   Wed Jan 28 08:58:38 2015 +0000

    EFI_FILE_PROTOCOL spec conformance bug fix.

    1. Write() should return Unsupported instead of WriteProtected when operating above a directory in read-only media.
    2. SetInfo() should return Unsupported instead of WriteProtected when operating above a directory using a undefined GUID

    Contributed-under: TianoCore Contribution Agreement 1.0
    Signed-off-by: Ruiyu Ni <ruiyu.ni@intel.com>
    Reviewed-by: Eric Jin <eric.jin@intel.com>
    Reviewed-by: Feng Tian <feng.tian@intel.com>

    git-svn-id: https://svn.code.sf.net/p/edk2-fatdriver2/code/trunk/FatPkg@93 efd2e655-3735-4b16-a529-16bfa2dd702b
```

```
$ git clone git://git.code.sf.net/p/tianocore/edk2-FatPkg
$ cd edk2-FatPkg
$ git reset --hard 8ff136aaa3fff82d81514fd3091961ec4a63c873
```

Now add this FatPkg to the overal EDK2 UEFI source tree

```
$ cp -rf edk2-FatPkg <Path-to-UEFI-Base-Folder>/FatPkg
```

### Building FAT32 UEFI Target

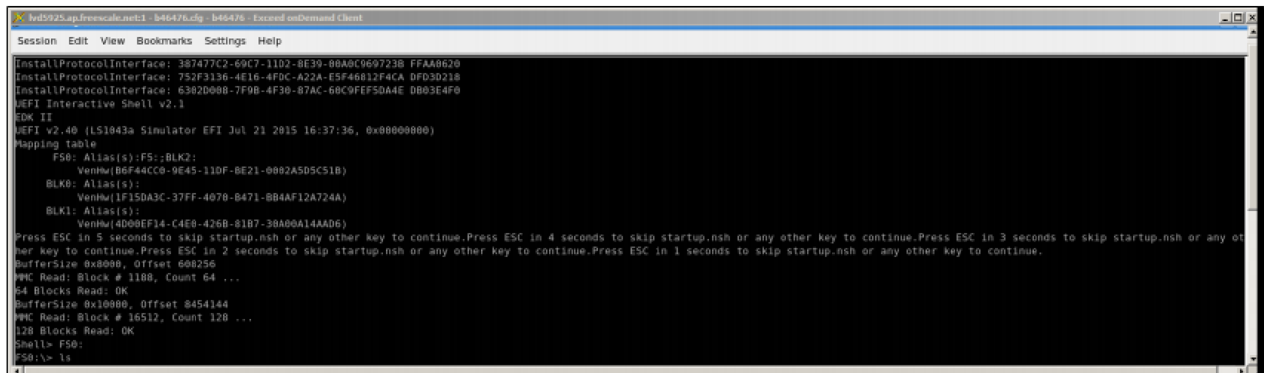
Please see the "How to Compile UEFI Image in Yocto" section for details.

### Testing the FAT32 filesystem on a SD card

The FATXIP UEFI build does not test NAND, NOR, DSPI and displays their test results as failed since they are not supported in this build.

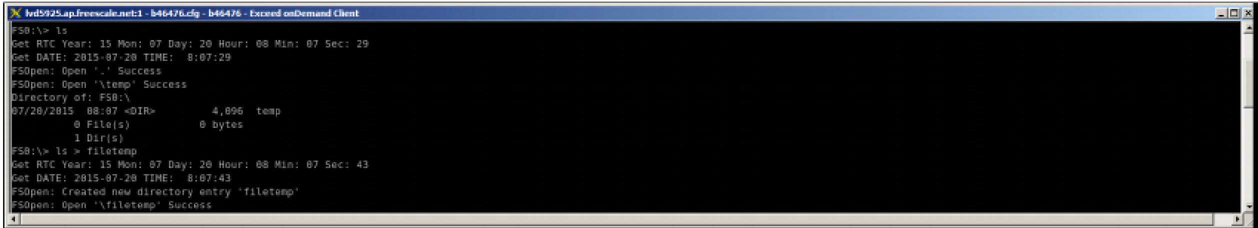
The steps to test Fat file system are as captured in the snapshots below:

1. Go to UEFI Shell.
2. Select File System.

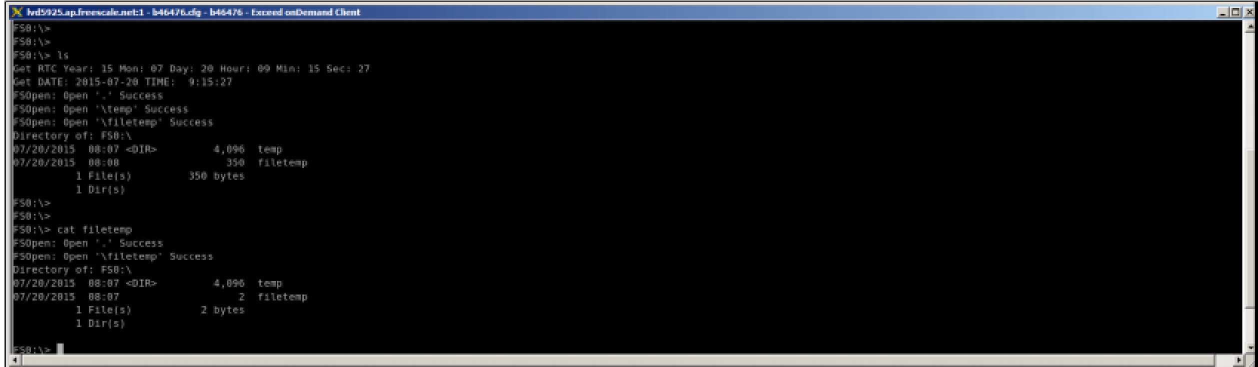



3. Make a new directory.

4. Create a new file.




5. Dump the contents of the new file.




### Miscelleneous - Install FAT32 file system on SD card

Use the DISKPART utililty on windows command prompt to format a SD card with FAT32 file system. Use the following snapshot forreference.

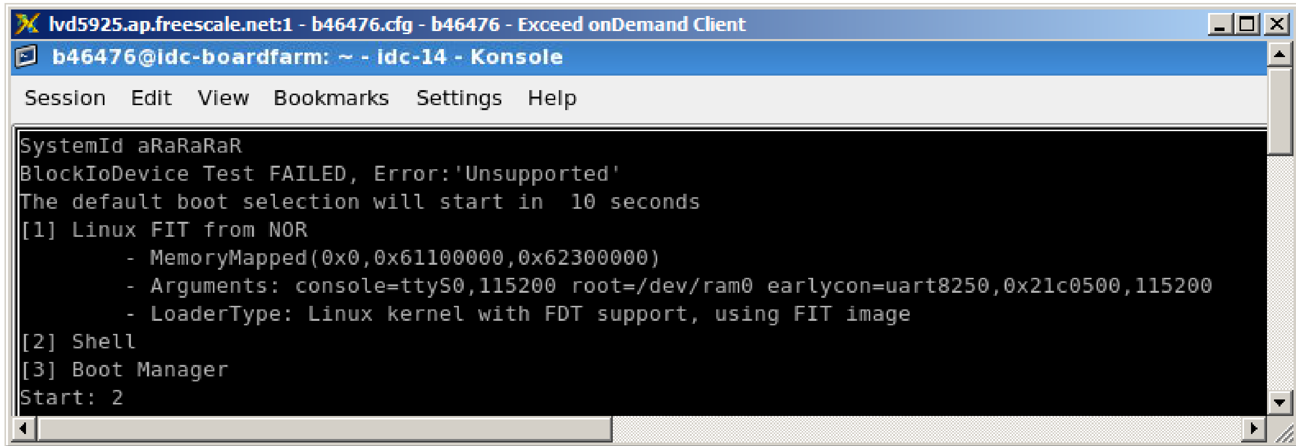
Figure 76: Using DiskPart Utility on Windows

```
C:\Windows\system32\diskpart.exe

DISKPART> list disk

  Disk ###  Status         Size     Free     Dyn  Gpt
  --------  -------------  -------  -------  ---  ---
  Disk 0    Online          298 GB      0 B
  Disk 1    Online         3781 MB      0 B

DISKPART> select disk 1

Disk 1 is now the selected disk.

DISKPART> list partition

  Partition ###  Type              Size     Offset
  -------------  ----------------  -------  -------
  Partition 1    Primary           3781 MB    64 KB

DISKPART> clean

DiskPart succeeded in cleaning the disk.

DISKPART> create partition primary

DiskPart succeeded in creating the specified partition.

DISKPART> format fs=fat32 quick

  100 percent completed

DiskPart successfully formatted the volume.

DISKPART> active

DiskPart marked the current partition as active.

DISKPART> list partition

  Partition ###  Type              Size     Offset
  -------------  ----------------  -------  -------
* Partition 1    Primary           3781 MB    64 KB

DISKPART>
```

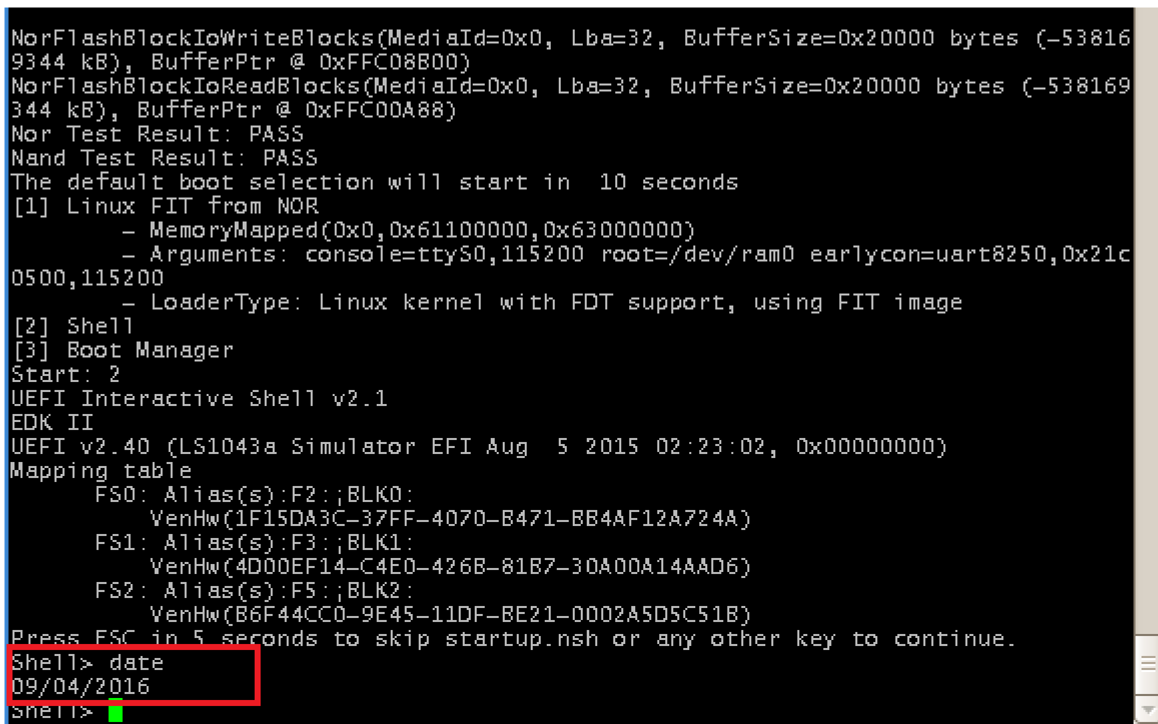
## 48.4.2 Using commands on the UEFI Shell:

The screenshot below showcases how to go to the UEFI shell before executing commands


Non-disclosure Agreement Required.




The screenshot below showcases how to use the UEFI shell to execute commands like **date**




**Resetting board to VBANK0 using reset on the UEFI Shell.**

The screenshot below showcases how to use the UEFI shell to execute **reset** command:

```
lvd5925.ap.freescale.net:1 - b46476.cfg - b46476 - Exceed onDemand Client
b46476@idc-boardfarm: ~ - idc-14 - Konsole
Session  Edit  View  Bookmarks  Settings  Help
Shell>
Shell>
Shell> reset
ASSERT /proj/nmgsw_be/users/b46476/code/ls1043a-uefi/LS1043aRdbPkg/Library/ResetSystemLib/ResetSystemLib.c(147): ((BOOLEAN)(0==1))


U-Boot 2015.01QorIQ-SDK-V1.7+g8a1f578 (Jun 09 2015 - 02:38:26)

Clock Configuration:
       CPU0(A53):1500 MHz  CPU1(A53):1500 MHz  CPU2(A53):1500 MHz
       CPU3(A53):1500 MHz
       Bus:      400  MHz  DDR:      1600 MHz
Reset Configuration Word (RCW):
       00000000: 0810000f 0c000000 00000000 00000000
       00000010: 14550002 80004002 e0025000 61002000
       00000020: 00000000 00000000 00000000 00038800
       00000030: 00000000 00001100 00000096 00000001
Board: LS1043ARDB, boot from vBank 0
CPLD:  V1.4
```

## 48.4.3 Platform Configuration Database (PCDs) Used in LS1043A UEFI Implementation

UEFI platforms support abstract configuration details associated with their platform and behavior. These configuration settings can be classified as two types:

1. Build-time generated platform settings.
2. Run-time generation platform settings.

LS1043A UEFI implementation does not support the Run-time PCDs, so we will talk here about the Build-time PCDs only.

A Build-time Platform Configuration Database (PCD) entry is a setting which is established during the time that the platform BIOS/Boot-loader is built.

The PCD entries are translated to a variable or macro that is auto-generatedby the build tool in each module's autogen.h and autogen.c files.

### PCD Rules

1. A PCD entry is defined in a DEC file.
2. Each module must include any PCD entries used in the code in the INF file for that module or driver.
3. The platform DSC can assign a static value to a PCD entry.

### Fixed PCDs

- **PcdsFeatureFlag**: “Feature Flag” PCD entries are Boolean values that enable or disable afeature flag. E.g.:

```
gArmPlatformTokenSpaceGuid.PcdTzc380Initialize|TRUE
```

- **PcdsFixedAtBuild**: “Fixed at Build” PCD entries cannot be changed in the IDE, since these PCD entries are fixed at build time. The PCD value is essentially a constant and acopy of the constant exists in each module that uses it. E.g.:

```
[PcdsFixedAtBuild.common]
gEmbeddedTokenSpaceGuid.PcdEmbeddedPrompt|"LS1043a"
```

## 48.4.4 How to Access BlockIo and I2C Devices

**BlockIo Devices:**

The UEFI specification defines BlockIo protocol for consumption by drivers of block storage devices like NAND, NOR, SPI Flash, and SD/MMC cards.

Any driver implementing BlockIo protocol defines the functions described in the protocol and installs a BlockIo interface as soon as it is loaded by the DXE dispatcher. The BlockIo interface provide services such as:

1. Block device read
2. Block device write
3. Block device reset
4. Block device flush

LS1043a UEFI implements BlockIo drivers for NAND, NOR, DSPI and SDXC.

At this stage there are no direct commands to test these block devices on UEFI shell (except for SDXC which is verified by accessing FAT file system installed on the SD card, which can be done through UEFI shell).

So in order to develop test applications for these drivers, developer can use the LS1043aFileSystem driver to test read/write access on NAND. NOR and DSPI flashes.

Developer can take reference from LS1043aTestBlockIoDevice (ArmPlatformPkg/Bds/Bds.c) to develop customized application.

The steps executed by the test function are:

1. Get handles for all drivers that install BlockIo protocol via gBS->LocateHandleBuffer.

   ```
   Status = gBS->LocateHandleBuffer (ByProtocol, &gEfiBlockIoProtocolGuid, NULL,
   &Size, &Handle);
   ```

2. Loop through each driver handle and perform the following actions

   a. Connect the driver handle with LS1043a File System driver via gBS->ConnectController. LS1043a File System Driver installs DiskIo and SimpleFIleSystem protocol on to the driver handle.

      ```
      Status = gBS->ConnectController (Handle[Index], NULL, NULL, FALSE);
      ```

   b. Get the SimpleFileSystem interface for the driver handle (the one that was installed in step 2).

      ```
      Status = gBS->HandleProtocol (Handle[Index],
      &gEfiSimpleFileSystemProtocolGuid, (VOID **)&FsProtocol);
      ```

   c. Get a File access interface using the FileSystemInterface through its OpenVolume function.

      ```
      Status = FsProtocol->OpenVolume (FsProtocol, &Fs);
      ```

   d. Open a file at a 4MB offset of test size 4KB on the block device, using the Open function of the file access interface.

      ```
      Status = Fs->Open (Fs, &File, L"0x400000|0x1000", EFI_FILE_MODE_READ, 0);
      ```

   e. Identify the device by comparing its signature with the known signatures of block devices implemented in the LS1043a UEFI code.

      ```
      MediaId = GetMediaId(File);
      ```

   f. Modify the file test size for the file depending on the device (this is due to varying block sizes for each block device).

g. Write a known pattern on a temporary buffer in RAM.

h. Pass the buffer to the Write function of the file access interface to be written on the block device.

```
Status = File->Write (File, &BufferSize, (VOID*)SourceBuffer);
```

.

i. Read from the device using the Read function of the file access interface to read data from the same location of the device, into a new buffer in RAM.

```
Status = File->Read (File, &BufferSize, (VOID*)DestinationBuffer);
```

j. Compare the pattern in both the source and destination buffer, byte by byte.

k. If the comparison yields complete match the device read/write works, otherwise it fails.

l. Disconnect the LS1043a File System driver from the device driver handle after printing the results in step k.

```
Status = gBS->DisconnectController (Handle[Index], NULL, NULL);
```

3. Free the buffer containing all the handle for BlockIo drivers, as retrieved in step 1.

#### I2cMaster Protocol:

The UEFI specification defines I2cMaster protocol for I2c bus needs to be operated in master mode.

Any I2c driver implementing I2cMaster protocol defines the functions described in the protocol and installs it as soon as it is loaded by the DXE dispatcher. The I2cMaster protocol provide APIs to

1. Set Bus Frequency (SetBusFrequency)
2. Reset I2c Host Controller (Reset)
3. Start I2c transaction(Read/Write) in master mode on I2c Host Controller (StartRequest)
4. Get structure pointing to I2c Host Controller Capabilities. (*I2cControllerCapabilities) (*NOT IMPLEMENTED*)

At this stage there are no direct commands to test these APIs on UEFI shell.

So in order to develop test applications for I2c drivers, developer can take reference from LS1043aTestI2c() (ArmPlatformPkg/Bds/Bds.c) to develop customized application.

The steps executed by the test function are:

1. Locate I2c Master protocol via gBS->LocateProtocol.

```
EFI_I2C_MASTER_PROTOCOL *I2c;
```

```
Status = gBS->LocateProtocol (&gEfiI2cMasterProtocolGuid,NULL, (VOID **)&I2c);
```

2. Now you can access I2cMaster APIs using *I2c* (pointer to I2cMaster Protocol, retrieved from LocateProtocol() function call), e.g.

   a. To set Bus Frequency of I2c Host controller,

```
Status = I2c->SetBusFrequency(I2c, &BusFreq)
```

b. To Reset I2c Controller,

```
Status = I2c->Reset(I2c);
```

c. To Read/Write using I2c,

```
EFI_I2C_REQUEST_PACKET *RequestPacket;
```

```
/* Fill Request packet structure*/
```

```
Status = I2c->StartRequest (I2c, EEPROM0_ADDRESS, RequestPacket, NULL, NULL);
```

The developer can take reference from this code to develop customized test applications for LS1043a UEFI. Please feel free to contact the UEFI team for more clarification.

*How To Reach Us*

Home Page:

freescale.com

Web Support:

freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale, the Freescale logo, and Kinetis are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. All other product or service names are the property of their respective owners. ARM and Cortex are registered trademarks of ARM Limited.

© 2015

QORIQSDKLS
Rev. A
23 December 2015