P4080 PCIe Adapter SDK User Guide

Production Release

Revision 1.1

1 Introduction

The P4080 PCIE is a PCIE Adapter board with a Freescale QorIQ P4080 SOC/Processor. Even though the form factor is a PCIE Adapter, the hardware design follows the Freescale's P4080 Development System, as much as possible. The P4080 PCIe Adapter is also referred to in some Freescale documentation as the N710 P4080PCIe adapter. The nomenclature for this document is P4080 PCIE Adapter.

The P4080 PCIE Adapter supports two modes of operation - a standalone host mode and a PCIe Endpoint mode. The adapter SDK comprises a P4080 based software framework to support communication over PCIE with the x86 host and an x86 host framework that includes Linux drivers and application libraries that enable access to the P4080 DPAA. The SDK of the Adapter provides support for the growing demand for intelligent network acceleration and application offload for converged datacenter applications such as Storage, Security, Deep Packet Inspection (DPI), Firewall, WAN Optimization, and Application Delivery (ADC) computing.

In most cases, operating and programming the P4080 PCIE is the same as the Freescale P4080DS. For additional reference, see the Freescale DevSys User guide, and the SDK 2.3 Documentation[2].

1.1 P4080 PCIE Adapter Overview

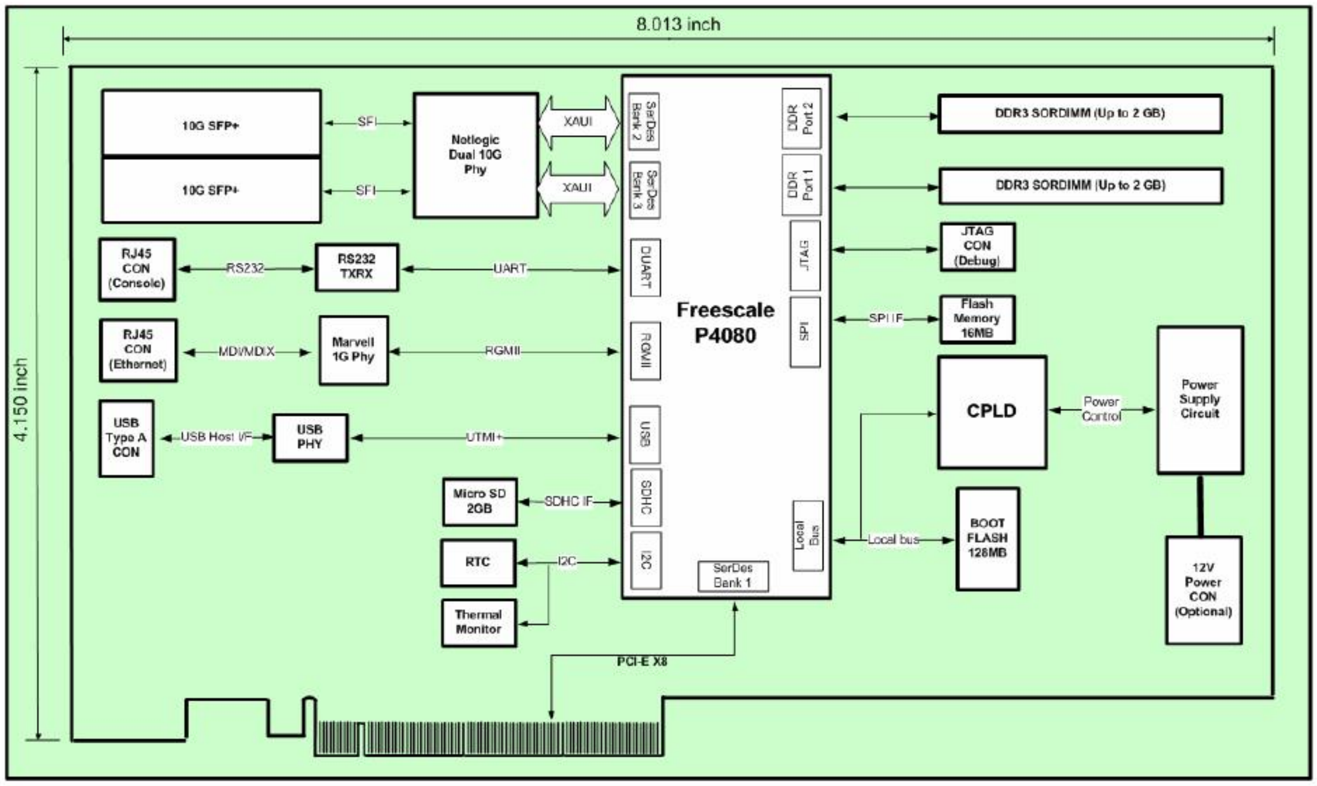


Figure 1 P4080 PCIE Adapter Block Diagram

The P4080 multi-core processor of the adapter combines eight Power Architecture e500mc cores with a Datapath Acceleration Architecture (DPAA) that includes the following acceleration blocks [1]:

- Frame Manger (FMAN) - Packet parsing, classification and distribution
- Queue Manager (QMAN) - Queue management for scheduling, packet sequencing and congestion management
- Buffer Manager (BMAN) - Hardware Buffer management for buffer allocation and de-allocation
- Cryptographic Security Acceleration (SEC 4.0)
- RegEx Pattern Matching (PME 2.0)

The card also provides the following integrated functions:

- One X8 PCIE Gen1 Interface
- 1 1G RGMII Management Port

- 2 10G XAUI Ports
- 1 RJ45 Serial Console port
- 1GB NOR Flash memory (for boot)
- 4GB memory
- 1 Micro SD card
- USB 2.0
- JTAG Debug interface, (Supported by Freescale's CodeWarrior IDE)
- CPLD that controls power sequence, reset, control signal and MDIO/I2C/SPI interface.

1.2 P4080 PCIE Adapter and P4080 DS key differences

For hardware details please refer to the Interface Masters Niagara 710A Hardware Specification [1]. Some key differences with the P4080 DS are listed here.

- Ethernet Management interface PHY chip is a Marvell MV88E1111
- Support was added to U-boot *drivers/net/fm/dtsec_phy.c*
- 10GE SFP+ PHY uses a Netlogic AEL2020 PHY.
- Support was added to U-boot *drivers/net/fm/tgec_phy.c*
- This support includes LED and related CPLD register changes
- Board Control on the DS, uses the Pixis FPGA, while the P4080 PCIE has its own unique CPLD

1.3 Intended Audience

This document is intended for software developers and system architects who work with the P4080 PCIe Adapter software.

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 User Space Datapath Acceleration Architecture (USDPAA) software for DPAA-based SoCs.
- At least broadly, the DPAA hardware blocks.

1.4 USDPAA Overview

USDPAA is a software framework that allows the development of Linux user space applications on the P4080 that have direct access to software portals for the DPAA buffer manager (BMan) and the DPAA queue manager (Qman). Refer to [2] for further details of USDPAA 2.3 SDK.

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.

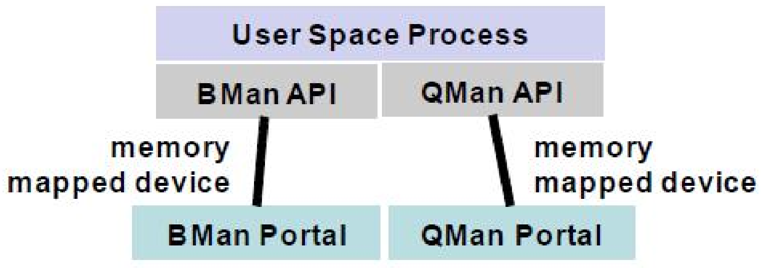


Figure 2 C-Language APIs Accesses Memory Mapped Device

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. USDPAA is intended to fulfill much the same function as the Freescale Light-Weight-Executive but in a different deployment context and in a more flexible manner.

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.

2 Production Release Features

P4080 PCIE Adapter SDK has two software components – a BSP component that runs on the P4080 adapter and x86 host software.

2.1 P4080 PCIE Adapter BSP

- RCW with Root Complex (to support stand-alone host mode) and End Point mode.
- U-boot with dual bank boot support.
- Support for standard u-boot commands.
- Linux USDPAA SDK 2.3 and DTS.
- 1G management port support.
- Linux SD card support.
- Linux USB support.
- Software Framework on P4080 to support exposure of multiple Hardware functions to Host.
- USDPAA Linux P4080 PCIE driver that enables control and data path between P4080 cores and x86 host with multi-ring support. The control path includes the initialization handshake between Host PCIe driver and P4080 PCIe driver, command requests and responses. The data path includes the exchange of data packets between the Hardware Function Drivers on Host to P4080 Hardware Functions and vice versa.
- USDPAA Linux P4080 PCIE driver also supports a command request/response mechanism for receiving host commands for multiple hardware functions and providing response to host.
- USDPAA Linux P4080 Ethernet driver that sends/receives network packets from/to two 10G ports on multi-core host. The Ethernet driver has support for multiple host descriptor Tx/Rx rings and PCD Frame Queues.

- USDPAA PCIE Packet Application that links in the PCIE Packet driver and Ethernet driver libraries and showcases the bi-directional control and data path between P4080 cores and multi-core host.
- Support of 8 Tx Frame Queues and 8 Tx confirmation Frame Queues for each 10G port on P4080.
- Support of 8 PCD Frame Queues for each 10G port on P4080.
- USDPAA APIs for support of Basic Direct and Basic Chaining DMA.
- Support for statistics of two 10G ports.

2.2 x86 Host software

- Software Framework on x86 Host to support exposure of multiple Hardware functions to x86 Host.
- PCIE Packet driver
  - Dynamically loaded and unloaded as a module. Also supports automatic load of modules on host boot-up.
  - Executes PCIE initialization.
  - Initialization handshake with P4080 driver.
  - Supports MSI for the device.
  - Command Request/Response mechanism to P4080.
  - API to upper-level (Ethernet) drivers – enqueue to Tx descriptor ring.
  - Callback for Rx ring of upper-level (Ethernet) drivers.
  - API to enqueue commands to the Command Ring.
  - Callback for handling command ring responses.

    - Read/Write to CCSR.

- Ethernet driver
    - Dynamically loaded and unloaded as a module. Also supports automatic load of modules on host boot-up.
    - Support of 8 Tx and Rx rings.
    - Support for both 10G ports.
    - Supports MSI for the device.
    - ifconfig statistics.
    - NAPI support.
    - Promiscuous mode.
    - Multicast filtering.
    - MTU change.
    - Support jumbo frames, 9600 KB max frame size.
    - MAC address change.
    - ifconfig up/down.
    - Tx checksum offload.
    - RX checksum verification and generation.
    - Limited Ethtool support – driver info, ring parameters, statistics, get/set Tx checksum.
- PCIe Tool - dump CCSR BAR and Read/Write into BAR memory based on offset

3 P4080 PCIe Adapter SDK Assumptions, Modes of Operation and Application Execution Model

3.1 Assumptions

- USDPAA Threads must be made affine to a core. This is due to support for stashing to per-core caches.
- Every USDPAA thread has (by definition) an allocated BMan software portal and an allocated QMan software portal. No other thread or entity accesses these portals.
- Threading is via standard Linux pthreads and the standard Linux GNU C library.
- There are no examples of using more than one USDPAA thread per core, and this use-case has not been well tested.
- Portals have a core affinity that is specified in the device tree.
- A fixed chunk of 64M Bytes of memory is allocated for USDPAA use. This memory is allocated for various purposes. Mainly, the DMA-able memory allocator provides access to it, and it is used for USDPAA buffers.
- A 4K Init Region is reserved at the top of the 64M Bytes USDPAA memory for use by the P4080 and x86 Host PCIe drivers to complete their initialization handshake.
- The P4080 USDPAA PCIe driver supports 8 TX and 8 RX host rings.
- The P4080 USDPAA Ethernet driver supports 8 PCD Frame Queues (FQs) and 8 TX FQs.
- The P4080 USDPAA PCIE Packet Process application runs on cores 1-7.
- x86 Host drivers have been compiled and tested only on hosts running Ubuntu 10.04.

3.2 Run-to-Completion USDPAA Application Execution Model

Both the stand-alone host mode and the EP mode support a run-to-completion model of execution of the USDPAA applications which is detailed in this section.

The full run-to-completion use case is defined by the following characteristics:

- No more than one USDPAA thread is used per core. 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" can be used. This parameter prevents the Linux scheduler from running a thread or process on an isolated core unless the scheduling mask of that thread or process is explicitly set to include the isolated core. USDPAA does not require isolation architecturally, but it may be used if the system goal is to, at least mostly, dedicate a core to running a single thread. Also, polling (see below) is wasteful if others threads are to run on the same core as the USDPAA thread.
- 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.
- 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.

Figure 3 illustrates the use of QMan in multi-step processing of a network frame:

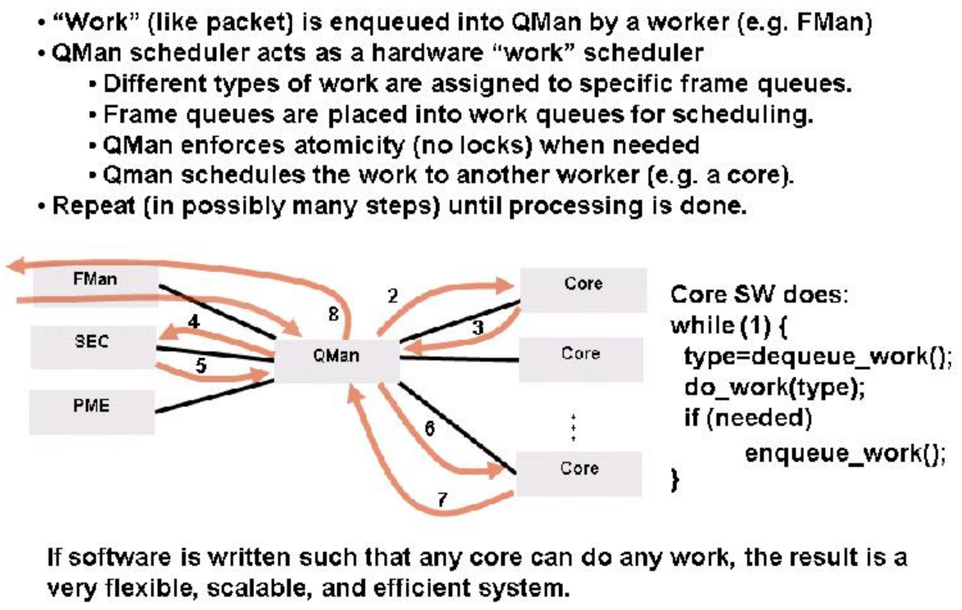
- Frame arrives at FMan.
- Frame is processed by a core and sent to SEC.
- SEC further processes the frame and returns it to a core, possibly a different core.
- Core sends frame to FMan for egress.

The numbers in the figure show the steps in the progression of a frame through the example.

The QMan scheduler determines the priority of processing at each step.

Specific types of "work" are carried by specific frame queues, the heads of which reside in specific QMan channel work queues. QMan inter-class scheduling operates at the work queue level and determines the order in which the messages that carry the "work" are delivered.

Work queues within QMan pool channels can be associated with classes of "work." Portals' QMan dequeue masks determine the set of channels from which the portal will dequeue. Since the portals are affine to cores, this determines which cores are configured to process which classes of work. Even the number of cores can be adjusted by changing the masks.



- “Work” (like packet) is enqueued into QMan by a worker (e.g. FMan)
- QMan scheduler acts as a hardware “work” scheduler
  - Different types of work are assigned to specific frame queues.
  - Frame queues are placed into work queues for scheduling.
  - QMan enforces atomicity (no locks) when needed
  - Qman schedules the work to another worker (e.g. a core).
- Repeat (in possibly many steps) until processing is done.

Core SW does:

```
while (1) {
  type=dequeue_work();
  do_work(type);
  if (needed)
        enqueue_work();
}
```

If software is written such that any core can do any work, the result is a very flexible, scalable, and efficient system.

Figure 3 Multistep Run-to-Completion Processing

3.3 Modes of Operation

The adapter can be deployed in one of the 2 configurations – either as a stand-alone host system (as a replacement for the P4080 DS) or a PCIE EP for an external host. Typically the stand-alone host mode is expected to be used as a replacement for the P4080 DS and the intended usage of the adapter will be predominantly in the EP mode.

Stand Alone Host Mode

The Stand-Alone Host Mode uses a Root Complex software configuration. Intended as a replacement for the P4080 DS, the stand-alone host mode supports all USDPAA applications that can be run on the P4080 DS. However the Root Complex mode is limited with no support to enumerate PCIe devices. The P4080 PCIE Packet Application, the USDPAA PCIE Driver and the USDPAA PCIE Ethernet driver aren’t available in this mode, since PCIE access to the host isn’t possible in this mode.

The 1G interface is assigned to Linux and the appropriate Linux DTS (p4080n710-linux.dts or p4080n710-usdpaa.dts) can be chosen, depending on whether the 10G interfaces are to be assigned to Linux or USDPAA.

PCIE End-Point Mode

The P4080 PCIE Adapter SDK supports a fully functional End Point mode. The Serdes configuration allows either:

- o X8 Gen1
  - Default chosen in our RCW
  - Has been fully validated
- o X4 Gen2
  - Would require an RCW change
  - This has NOT been validated, but the P4080 should support the Serdes Configuration.
  - If it is needed, a request can be made for factory validation

Unlike the P4080 DS which supports multiple Net interfaces, with daughter cards added, the P4080 PCIE has a fixed configuration, with the PCIE running X8, and Dual 10GE Network Interfaces. The RCW Serdes configuration 0x2 is set for this mode.

The End Point mode exposes 2 memory regions to the host. These are enumerated by the host as:

- BAR 0 - 32 bit memory mapped CCSR BAR that maps all P4080's memory mapped configuration, control, and status registers contained within a 16-Mbyte region.
- BAR1 – 32 bit memory mapped BAR that exposes 64 MB of P4080 memory used for DMA buffers and for an initialization handshake. A 4K INIT_REGION, is used by both the host and P4080 PCIe packet drivers to communicate init information such as number of descriptor rings, addresses of the rings, size of the rings, producer and consumer indices of the rings, and MSI vectors.

4 P4080 PCIE Adapter SDK Components

The P4080DS based Linux Target Image Builder (LTIB) BSP for the adapter includes rcw, u-boot, Linux kernel with device drivers and rootfs, USDPAA library and applications [2]. A framework is added to the BSP to create new USDPAA applications that requires communication with Host. This framework includes a static USDPAA packet driver library (usdpaa_eth.a) and a sample Ethernet driver (usdpaa_eth.so) shared library.

4.1 P4080 USDPAA based PCIE Framework

The P4080 USDPAA based PCIE software development framework comprises:

- P4080DS based Linux Target Image Builder (LTIB) BSP for the adapter which includes u-boot, Linux kernel with device drivers and rootfs, USDPAA library and applications [2]. A framework is added to the BSP to create new USDPAA applications that requires communication with Host. This framework includes a static USDPAA packet driver library (usdpaa_pkt.a) and a sample Ethernet driver (usdpaa_eth.so) shared library.
  The P4080 packet driver interfaces with the x86 Host Packet driver and the DPAA on P4080 enabling data communication between x86 Host and P4080. The layering of the PCIE framework requires upper level P4080 "hardware function" drivers to register themselves with the lower-level P4080 PCIe Packet Driver and use its APIs/callbacks for communication with the x86 host. The hardware function drivers supported in this SDK are the two 10G Ethernet port drivers. However, the Packet Driver can easily be extended to support other hardware functions of P4080 such as the security engine and PME. The P4080 Packet Driver provides APIs to upper level P4080 "hardware function" drivers to enqueue to the host descriptor rings. It also provides a callback mechanism to notify "hardware function" drivers of the packets dequeued from the host descriptor rings. In addition, the Packet Driver provides a command request/response mechanism that can be used by the hardware function drivers to receive command requests from their counterparts on the x86 host and send responses.
  The sample P4080 USDPAA Ethernet driver includes a Packet Flow Engine that sends/receives network packets and other hardware engine commands to/from the x86 host descriptor rings and DPAA. Sample XML policy and configuration files are provided to enable flow classification.

- x86 host framework that includes Linux Packet driver that enables access to the DPAA and sample Ethernet driver for the adapter's 10G ports.

Figure 4 provides the following details:

1) The SMP Linux USDPAA configuration will be run on the adapter.
2) A multi-threaded USDPAA application linked with usdpaa_pkt.a and usdpaa_eth.so is run on cores 1-7 of P4080. In the network packet transmit direction, this application reads P4080 FDs from Host descriptor rings, processes them and enqueues into FMan frame queues using QMan. In the receive path, it receives the QMan_FDs from FMan Ethernet ports, processes them, prepares P4080 FDs and enqueues them onto Host rings.
3) QorIQ Address region, Packet Driver Init Region, exposed via MMIO BAR, memory mapped over PCIe on host, and translated via inbound ATMUs. This memory region is used by Host Packet Driver and USDPAA Packet Driver on P4080 for the initialization handshake

4) Host Memory region for transmit buffers, memory mapped over PCIe on the adapter and translated via outbound ATMU.

5) Mapping memory regions - Host to QorIQ memory region using inbound ATMU, QorIQ to Host memory region using outbound ATMU.

6) Control flow between various components like FMAN, QMAN, BMAN, Linux Core on QorIQ and Host.

7) Usage of Command ring from Host to send commands to P4080 which are processed by Core 1 on P4080. These are forwarded to corresponding Hardware Function library on P4080.

8) API exposed by Host Packet Driver and P4080 Packet Driver library.

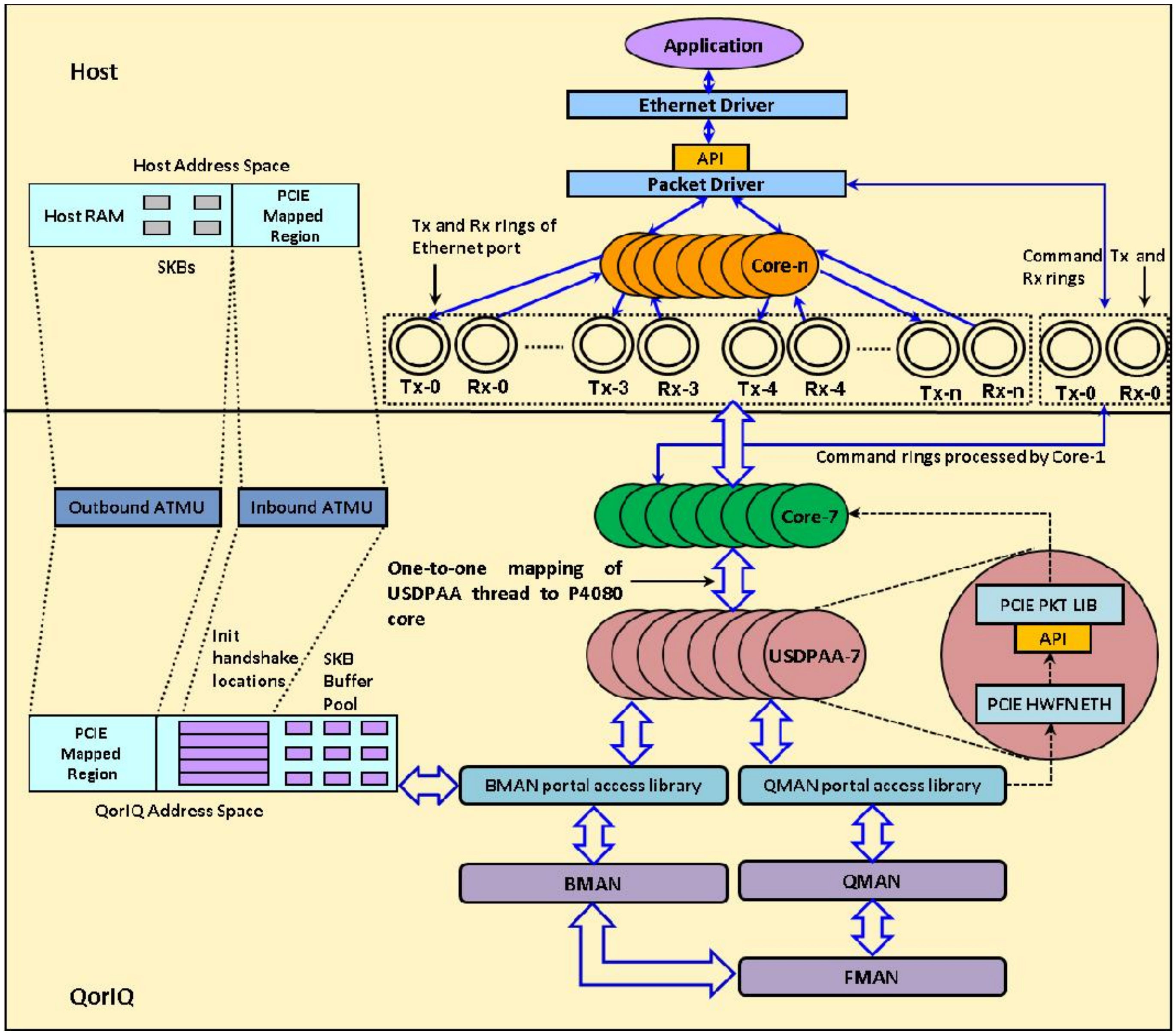


Figure 4 Software Development Framework

4.2 P4080 USDPAA PCIe Packet Application Overview

A USDPAA PCIE Packet Application, with "worker" threads that are responsible for packet processing runs on 7 P4080 cores. The packet application's main process and worker threads are responsible for different functions. The main process's tasks include:

- Executing global initialization common to all worker threads running on separate cores. This includes loading and extracting the DPA configuration from the fman driver and FMC configuration (policy and config xml files), mapping the DMA shared memory, mapping the PCIE address space, initializing the DMA controllers and their channels and creating the CLI sever.
- Creating the worker threads and triggering the "primary" worker to do data path dependent initialization.
- Start processing CLI commands in an indefinite loop.

The "primary" worker thread's (running on Core 1) functionality includes:

- Initializing the bman/qman portals
- Initializing buffer pools used by the Fman Rx ports
- Executing the P4080 portion of initialization handshake. The purpose of the initialization handshake is to initialize the data structures with the host Tx/Rx ring base addresses and producer/consumer indices and to setup the Tx/Rx flow tables with the Tx/Rx host ring indices and MSI vectors to be used on the datapath.
- Mapping the P4080 PCIe memory to userspace to provide access to the Host rings in user space.
- Allocate Hardware Function specific data structures that are necessary for data path separation between Hardware Function Libraries.
- Opens the shared object files of Hardware functions and gets the initialization and cleanup function symbols which are needed for Hardware Function initialization.
- Running the packet processing poll loop that does:

  - *pkt_drv_process_command_ring():* Poll for Command enqueues by the host driver to the Tx Command Ring. Commands are dequeued from the ring and appropriate handlers invoked for command processing. Examples of hardware function driver commands sent by the host include Hardware Function Driver load and unload. For the Ethernet Hardware Function, Frame Queues are also initialized by the handlers invoked as a result of command requests from Host Packet Driver.
  - *pkt_drv_host_poll():* Poll for Packet enqueues by host driver to the host Tx ring of the core. Packets are dequeued from the Tx ring and enqueued to the Tx FQ as described in the transmit flow in Section 4.5
  - *dma_channel_poll():* Poll for DMA completions in DMA chain.
  - *qman_poll():* Poll for received packets from the qman software portal. Packets are enqueued to the corresponding host Rx ring and MSI asserted to the corresponding core. This is described in the receive flow of Section 4.6
  - Poll for DMA completions in DMA chain.
  - *bman_poll():* Poll for BMan driver notifications to the Packet process application.
  - Poll for DMA completions in DMA chain.

The worker threads running on other cores execute the following tasks:

- Initializing the bman/qman portals for the core
- Running the packet processing poll loop that does:
  - *pkt_drv_host_poll():*Poll for packet enqueues by host driver to the host Tx ring of the core. Packets are dequeued from the Tx ring and enqueued to the Tx FQ as described in the transmit flow in Section 4.5
  - *dma_channel_poll():* Poll for DMA completions in DMA chain.
  - *qman_poll():* Poll for received packets from the qman software portal. Packets are enqueued to the corresponding host Rx ring and MSI asserted to the corresponding core. This is described in the receive flow of Section 4.6
  - Poll for DMA completions in DMA chain.
  - *bman_poll():* Poll for BMan driver notifications to the Packet process application.

- Poll for DMA completions in DMA chain.

4.3 P4080 USDPAA PCIe Packet Driver Overview

- The USDPAA PCIe Packet Driver which runs on P4080 is developed as a static library that links with the PCIe Packet Process.
- This Driver does initialization with the help of Host Packet Driver (initialization handshake) and starts USDPAA threads on 1-7 cores. Each thread is affine to a core and is assigned a dedicated Tx/Rx ring pair.
- It provides APIs for Packet Driver initialization (*pkt_drv_init()*), cleanup (*pkt_drv_finish()*), Hardware Function Driver registration (*pkt_drv_register_hw_function()*), transmit (*pkt_drv_host_poll()*) and receive (*pkt_drv_enqueue_host_ring()*) path processing and command ring processing (*pkt_drv_process_command_ring()*).
- As part of Packet Driver initialization, it does the initialization handshake, maps the PCIe memory to use space and allocates Hardware Function specific data structures.
- It does the transmit path processing by polling on transmit rings on Host for enqueued Frame Descriptors. It does the receive path processing by polling on Frame Queues (using the QMAN driver APIs). It uses DMA APIs to transfer data to and from Host memory.

4.4 P4080 USDPAA Ethernet Driver Overview

- The USDPAA Ethernet Driver on P4080 is developed as a shared library that links with the PCIe Packet Process.
- This Driver's initialization is started by USDPAA PCIe Packet Driver as illustrated in Figure 5. The PCIe Packet Driver supports dynamic discovery of the hardware function driver libraries by looking up the symbol of the initialization routine of the hardware function driver. It invokes the "hw_fn_eth_init()" call of the Ethernet driver triggering the Ethernet driver to register with USDPAA PCIe Driver. The Ethernet driver also does the initialization of the 10G interfaces of FMAN.
- It sets up transmit and receive Frame Queues and builds flow tables to direct the transmit and receive packet flows.

- It registers command queue callback function and Tx callback function to receive special purpose commands and Tx packets respectively from Host.
- This Driver invokes the QMAN Driver APIs to enqueue the Tx FDs to QMAN. It registers receive callback functions with QMAN Driver at the time of Frame Queue initialization to get receive packets from 10G interfaces.

4.5 Data path Processing

The call flow between the USDPAA process, PCIe Library and Ethernet Library is shown in Figure 5. The Tx and Rx data path is described below:

Tx Path

- The *pkt_drv_host_poll()* API of USDPAA PCIe Driver obtains the Tx ring's producer index (maintained in P4080 memory) and compares it with the consumer index (maintained as local copy of actual value on Host memory) to check if there are any pending Tx FDs to be serviced.
- If there are Tx FDs to be processed, it acquires a buffer from BMan buffer pool for SKB data and initiates a DMA of data from Host SKB to BMan buffer. The consumer index on Host is also updated using DMA.
- The *dma_channel_poll()* API is called by the application thread to check for DMA completions. It in turn invokes the *host_poll_cb()* callback registered by the hardware function driver when the Tx DMA is complete The FD is passed as a parameter to the callback function. For each callback invocation, a Tx flow table lookup is performed by *host_poll_cb()* to identify the Tx FQID corresponding to the Host Tx ring from which this Host FD is dequeued. The FD is updated to indicate to the FMAN to not send a Tx Confirmation to the caller but release the data buffer into the buffer pool.
- After identifying the Tx FQ, the FD is enqueued to the FQ using qman_enqueue() API to be sent out on one of 10G interfaces

Rx Path

- qman_poll() API invokes the FQ specific callbacks registered by Ethernet Hardware Function Library during FQ init. In each callback invocation, the Rx Flow Table is looked up to determine the corresponding Host Rx ring ID.
- For every Ethernet Rx packet, the *pkt_drv_enqueue_host_ring() API* is invoked to transfer the Rx packet to host memory**.** In this API, it is verified if the Rx ring has some free FDs. If there are free FDs, the DMA of Rx data is initiated from P4080 to Host memory by obtaining the Host address from Rx shadow ring's FD. The necessary fields inFD of the Host Rx Descriptor ring are modified and the Host producer index is updated.
- The dma_channel_poll() API, invoked by the application thread checks for DMA completions. If it finds Rx DMA completions, an MSI interrupt is sent to the Host to notify of an Rx packet.

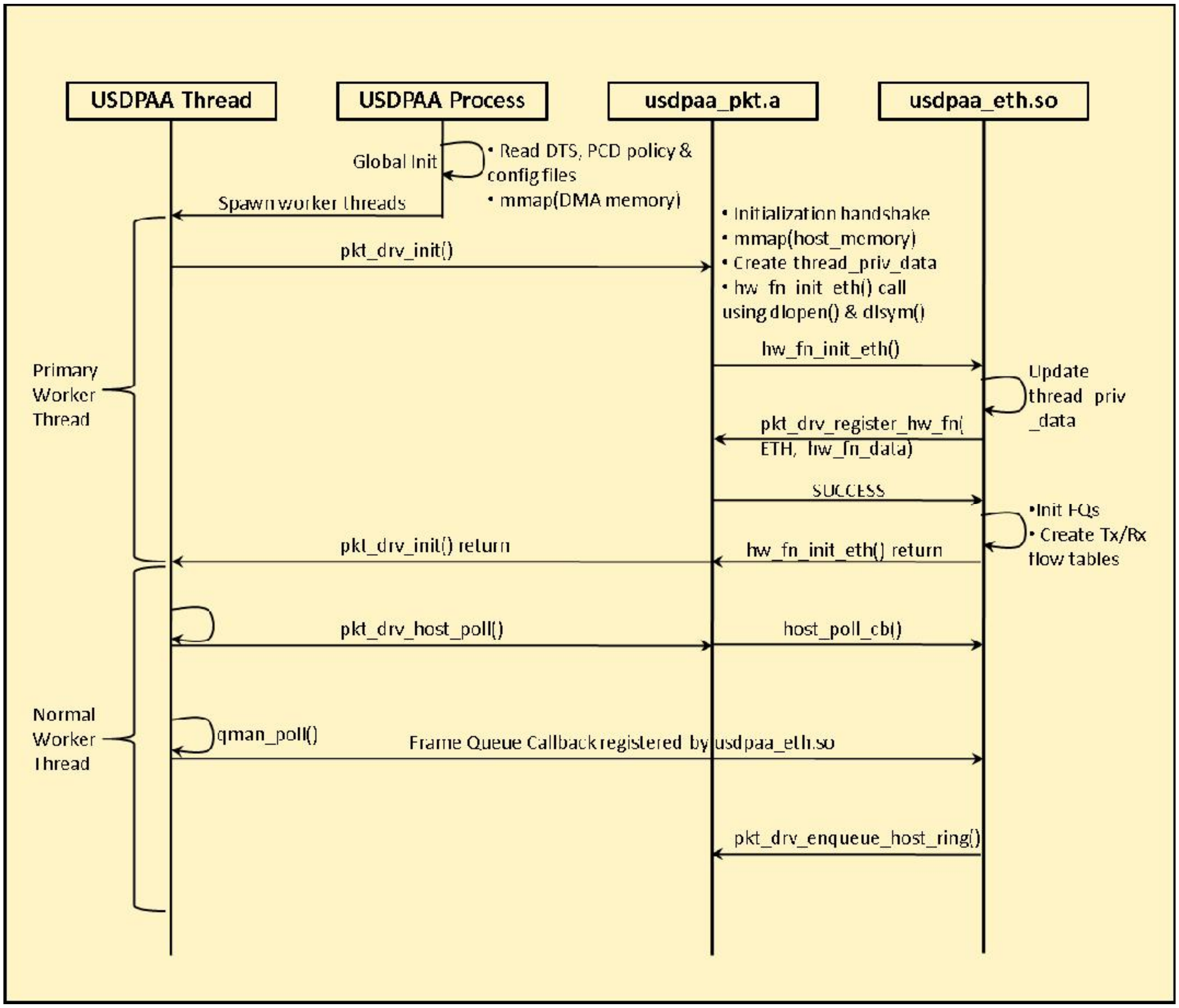


Figure 5: Call flow between USDPAA process and Libraries

4.6 Command Ring Processing

- Command ring is used to send special purpose requests from Host to P4080. These sent requests can be specific to a Hardware Function or specific to a PCIe Packet Driver.
- To send a new command, the command requester (Host Hardware Function Driver or Host Packet Driver) prepares a command data structure of type *"struct p4080_fd_t"* and fills *"type"* and

"sub_type" fields. The *"type"* field is filled based on *"enum P4080_FD_TYPES"* and *"sub_type"* is filled based on *"enum fd_sub_type"* or *"enum P4080_ETH_SUB_TYPES"*.

- The new command request is sent from host to P4080 using the API *"enqueue_cmd()"*.
- On P4080, *"pkt_drv_process_command_ring()"* API that runs on core 1, polls on command ring. If the command is intended to Packet Library (for some Hardware function independent) functionality, it is handled by the PCIe Packet Library. If it is intended to some Hardware Function Library, corresponding pre-registered command ring callback is invoked to process this request.
- After obtaining the command request, the *"pkt_drv_enqueue_cmd_response()"* API is invoked to enqueue the command response to Host. This is done immediately after getting the command request to maintain response ordering. The command request is processed and after completion of this, *"pkt_drv_confirm_cmd_response()"* API is invoked to mark the command response in the command Rx ring as completed. An MSI interrupt is generated by the *"pkt_drv_process_command_ring()"* API to the completed commands.
- Host Packet Driver, after getting MSI interrupt for completed commands, invokes the pre-registered command Rx callback function.

4.7 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.

4.8 Note on the current USDPAA 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. If it does not, stashing may be done to the wrong core's cache. This is not necessarily functionally incorrect, but it is not efficient.

4.9 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."

4.10 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.

5 Relationship to SDK Linux Ethernet Subsystem

The DPAA SDK FMan and Linux kernel ethernet driver software is defined as external to the USDPAA software. However, USDPAA, the ethernet driver in the kernel, and the FMan software are all interrelated. Figure 6 shows three use cases involving the three components.

1. QMan connects a FMan MAC only to the kernel ethernet driver, which exchanges frames with the Linux network stack.
2. QMan connects a FMan MAC only to a USDPAA application.
3. QMan connects a 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.

The current release of USDPAA demonstrates cases 1 and 2.

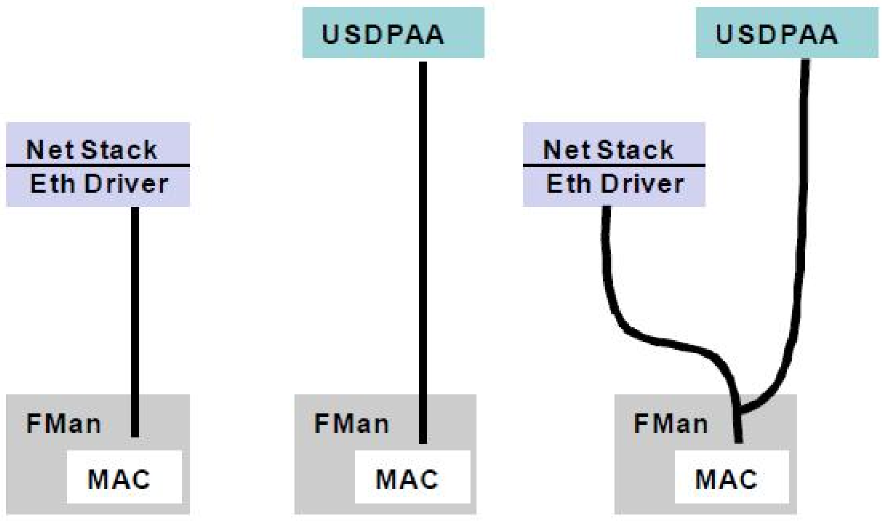


Figure 6 USDPAA, FMan, Ethernet Use-Cases

5.1 Selecting 10G Ethernet Interfaces for USDPAA

The Ethernet-related Linux device tree entries 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 1G management interface is always assigned to Linux. The following device tree snippet shows a Linux private interface and also 10G 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@4 {
   compatible = "fsl,p4080-dpa-ethernet", "fsl,dpa-ethernet";
   fsl,qman-channel = <&qpool4>;
   fsl,fman-mac = <&enet4>;
};
ethernet@9 {
   compatible = "fsl,p4080-dpa-ethernet", "fsl,dpa-ethernet";
   fsl,qman-channel = <&qpool4>;
   fsl,fman-mac = <&enet9>;
};
```

The first ethernet interface is used by the Linux ethernet driver. Ethernet interfaces 4 and 9 are used by USDPAA. The other interfaces in the dts are not available on the P4080 PCIE adapter. The table shows the ethernet interfaces are used in SerDes 0x2 configuration supported by the card.

Table 1 P4080 PCIE Adapter Ethernet Interfaces

Deice Tree Name	U-boot Name	U-Boot MAC Environment Variable	Linux Name (udev)	SerDes 0x2 Physical Position
ethernet@	FM1@DTSE	eth1addr	fm1-gb1	Motherboad RGMII
ethernet@	FM1@TGEC	eth4addr	fm1-10g	XAUI

ethernet@	FM2@TGEC	eth9addr	fm2-10g	XAUI

6 P4080 PCIE SDK Installation and Execution

The sections below summarize the process of installing and building the SDK provided within the USDPAA iso. The instructions below are a summary and may be augmented by consulting the SDK 2.3 documentation.

There are two steps to build from an LTIB ISO released image - installation and compilation.

6.1 P4080 PCIE Adapter Software Release Contents

The release software is given in the form of P4080-PCIE-PRODUCTION-10-19-2011.tgz. The following steps need to be executed in-order to extract the release source code.

- *$ tar -zxvf P4080-PCIE-PRODUCTION-10-19-2011.tgz*
  This extracts the sources in the form of P4080-PCIE-PRODUCTION-10-19-2011 directory.
- *$ cd /<path>/P4080-PCIE-PRODUCTION-10-19-2011*
  This directory contains the following contents.
  - x86 directory – x86 host device drivers and x86 host based CCSR Dump tool
  - docs – Contains the following documents
    - p4080_pcie_user_guide.pdf
    - n710_cpld_04.pdf
    - Niagara710_Hardware_spec_rev02.pdf
    - FSL-Production-TestSummary.xlsx
  - LTIB patch (ltib-e500mc-p4080-usdpaa-b2.3.patch) - LTIB Software changes

The software extensions to the LTIB SDK are mentioned in next section. The x86 directory has the following directory structure.

```
.
|-- drivers                            x86 device drivers
|   |-- include                        Common header files for all drivers
|   |   |-- p4080-common.h             Contains common data structures
|   |   |-- pkt_drv_apis.h           PCIe Packet driver APIs
|   |   `-- ring_buffer.h             Library for ring buffer management
|   |-- Makefile                       Makefile to compile all x86 drivers
|   |-- p4080-eth                      x86 Ethernet driver
|   |   |-- include              x86 Ethernet driver header files
|   |   |   `-- eth.h
|   |   |   `-- hashtable.h          Library for hash table implementation
|   |   |-- Makefile                   Makefile to compile Ethernet driver
|   |   `-- src                        x86 Ethernet driver source code
|  |      |-- eth.c                      Core Ethernet Driver implementation
|  |      |-- ethtool.c                  Ethtool implementation
|  |      |-- Kbuild
|  |      `-- Makefile
|  |-- pcie-core                x86 PCIe Packet driver
|  |  |-- include               x86 PCIe Packet driver header files
|  |  |  `-- pcie-core.h
|  |  |-- Makefile                     Makefile to compile x86 PCIe Packet driver
|  |  `-- src                          x86 PCIe Packet driver source code
|  |      |-- Kbuild
|  |      |-- Makefile
|  |      `-- pcie-core.c              Core implementation of PCIE Packet Driver
|-- Makefile                            Makefile to compile complete x86 source code
`-- utils                               Utility software
   |-- fsl_pcie_tool                    x86 based tool to dump CCSR
   |  |-- include                       Header files for CCSR tool
   |  |  `-- fsl_pcie_tool.h
   |  |-- Makefile                      Makefile to compile PCIe tool
   |  `-- src                           CCSR tool source code
   |      |-- fsl_pcie_tool.c          Core implementation of PCIe tool
   `-- Makefile                         Makefile to compile all utilities
```

6.2 Installation

P4080 PCIE Adapter SW patching

The LTIB ISO used as a base for the patch is **ltib-e500mc-p4080-usdpaa-b2.3.iso**. This is the base LTIB software on which the patch given in the release has to be applied. The following steps need to be executed in-order to apply the patch.

- On an x86_64/x86 Linux development system, mount the iso using the loop device

  ```
  $ mount -o loop ltib-e500mc-p4080-usdpaa-b2.3.iso /mnt
  ```

- `$ cd /mnt`
- `$ ./install`

  This would prompt for the installation path for LTIB. Give path to *"/<path>/P4080-PCIE-PRODUCTION-10-19-2011" (Note that P4080-PCIE-PRODUCTION-10-19-2011 is the* *release directory which is created when P4080-PCIE-PRODUCTION-10-19-2011.tgz is un-tarred)*

- *`$ cd /<path>/P4080-PCIE-PRODUCTION-10-19-2011/ltib-e500mc-20110331`*
- `$ patch -p1 < ../ltib-e500mc-p4080-usdpaa-b2.3.patch`

This will generate the LTIB with P4080 PCIE EP and RC mode custom platforms as well as it will patch Linux Kernel, U-boot and USDPAA with the necessary changes.

6.3 Compilation

LTIB

- Check if *"/opt/freescale"* directory exists, then delete it.
- Go to LTIB directory :

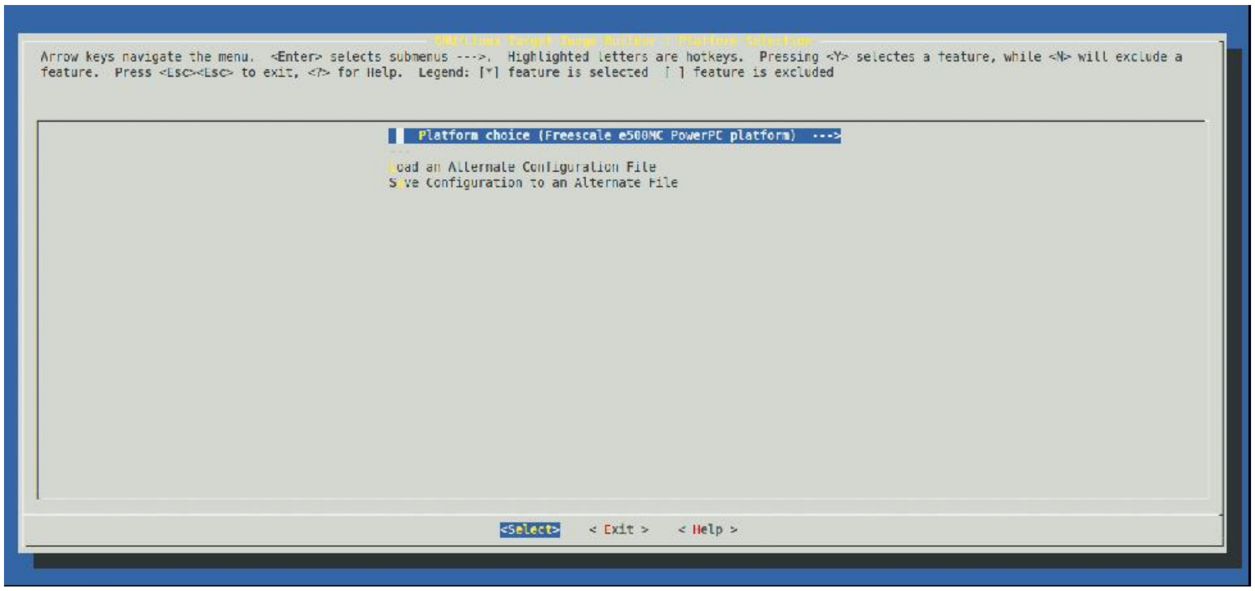
```
$ cd P4080-PCIE-PRODUCTION-10-19-2011/ltib-e500mc-20110331
```

- Start compilation:

```
$ ./ltib
```

- Once compilation starts, configuration screens are displayed.

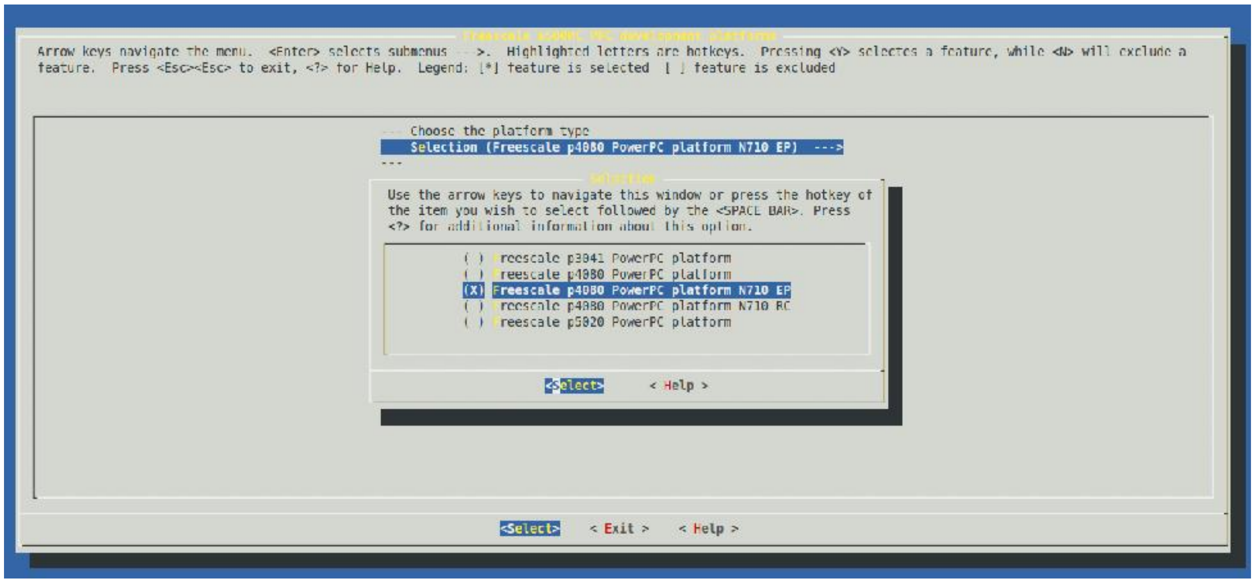
Screen 1



```
                  GNU/Linux Target Image Builder : Platform Selection
Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters are hotkeys.  Pressing <Y> selectes a feature, while <N> will exclude a
feature.  Press <Esc><Esc> to exit, <?> for Help.  Legend: [*] feature is selected  [ ] feature is excluded

                         Platform choice (Freescale e500MC PowerPC platform)  --->
                         ---
                         Load an Alternate Configuration File
                         Save Configuration to an Alternate File

                                <Select>    < Exit >    < Help >
```

- Hit enter for “Select” to prompt “Screen 2”. It prompts to save configuration. After saving the configuration, screen 2 will be displayed.

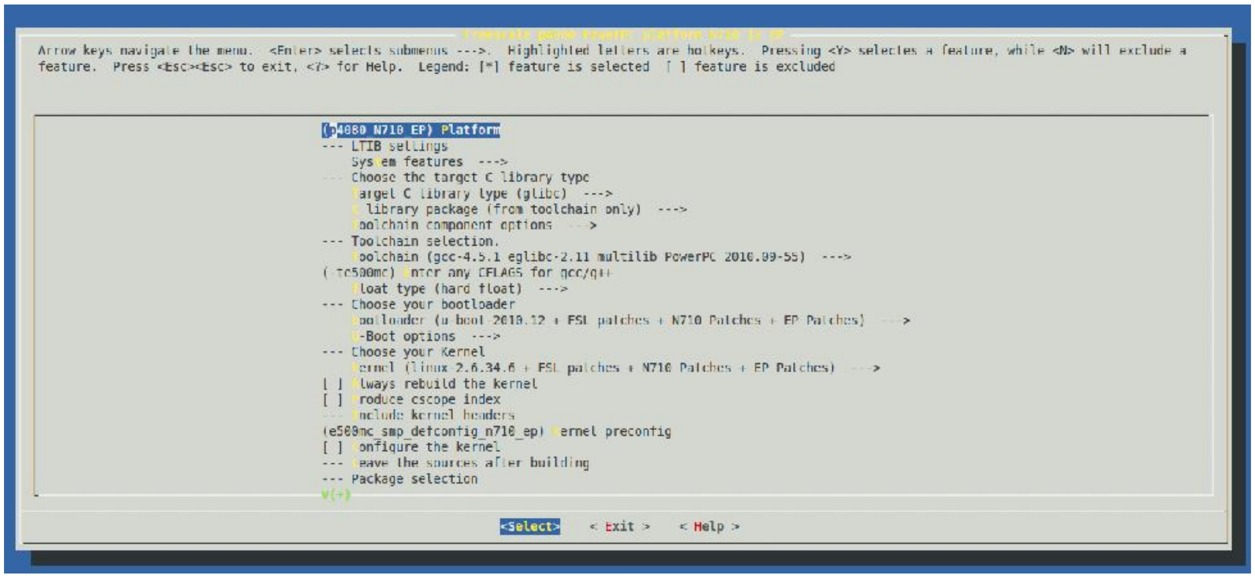
Screen 2



```
                  Freescale e500MC PPC development platform
Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters are hotkeys.  Pressing <Y> selectes a feature, while <N> will exclude a
feature.  Press <Esc><Esc> to exit, <?> for Help.  Legend: [*] feature is selected  [ ] feature is excluded

                        --- Choose the platform type
                            Selection (Freescale p4080 PowerPC platform N710 EP)  --->
                        ---
                                      Selection
                         Use the arrow keys to navigate this window or press the hotkey of
                         the item you wish to select followed by the <SPACE BAR>. Press
                         <?> for additional information about this option.

                                  ( ) Freescale p3041 PowerPC platform
                                  ( ) Freescale p4080 PowerPC platform
                                  (X) Freescale p4080 PowerPC platform N710 EP
                                  ( ) Freescale p4080 PowerPC platform N710 RC
                                  ( ) Freescale p5020 PowerPC platform

                                      <Select>      < Help >

                                <Select>    < Exit >    < Help >
```

- Select “Freescale P4080 PowerPC platform N710 EP” on second screen for Endpoint mode BSP or select “Freescale P4080 PowerPC platform N710 RC” for Root Complex mode BSP, save and exit. Third screen will be displayed corresponding RC/EP mode configuration.

Screen 3



```
                    Freescale P4080 PowerPC platform N710 EP
Arrow keys navigate the menu.  <Enter> selects submenus --->.  Highlighted letters are hotkeys.  Pressing <Y> selectes a feature, while <N> will exclude a
feature.  Press <Esc><Esc> to exit, <?> for Help.  Legend: [*] feature is selected  [ ] feature is excluded

                              (p4080 N710 EP) Platform
                              --- LTIB settings
                                  System features  --->
                              --- Choose the target C library type
                                  Target C library type (glibc)  --->
                                  C library package (from toolchain only)  --->
                                  Toolchain component options  --->
                              --- Toolchain selection.
                                  Toolchain (gcc-4.5.1 eglibc-2.11 multilib PowerPC 2010.09-55)  --->
                              (-Te500mc) Enter any CFLAGS for gcc/g++
                                  Floating type (hard float)  --->
                              --- Choose your bootloader
                                  Bootloader (u-boot-2010.12 + FSL patches + N710 Patches + EP Patches)  --->
                                  U-Boot options  --->
                              --- Choose your Kernel
                                  Kernel (linux-2.6.34.6 + FSL patches + N710 Patches + EP Patches)  --->
                              [ ] Always rebuild the kernel
                              [ ] Produce cscope index
                              --- Include kernel headers
                              (e500mc_smp_defconfig_n710_ep) Kernel preconfig
                              [ ] Configure the kernel
                              --- Leave the sources after building
                              --- Package selection
                              v(+)

                                   <Select>    < Exit >    < Help >
```

- Exit from the third screen and save the configuration when it prompts.
- In case if any of the configuration screens are not displayed as mentioned above, use the following commands.
  - `$ ./ltib -m clean`
  - `$ ./ltib -m distclean`
  - `$ ./ltib -m config`
- After getting the configuration screens, follow the above mentioned steps to change and save the configuration.

x86 Device Drivers

- Go to x86 device drivers directory:

```
$ cd P4080-PCIE-PRODUCTION-10-19-2011/x86
```

- Compile complete source code:

```
$make
```

- Install the PCIE Packet Driver in *"/lib/modules/<Kernel-Version>/kernel/drivers/pci/p4080/"* directory and x86 Host Ethernet Driver in *"/lib/modules/<Kernel-Version>/kernel/drivers/net/p4080/"* directory.

```
$ make install
```

- X86 Host Packet driver will be created in *"P4080-PCIE-PRODUCTION-10-19-2011/x86/drivers/pcie-core/src"* directory as *"pcie-core.ko"*
- X86 Host Ethernet driver will be created in *"P4080-PCIE-PRODUCTION-10-19-2011/x86/drivers/p4080-eth/src"* directory as *"p4080eth.ko"*

6.4 Physical Connection to the P4080 PCIE Adapter

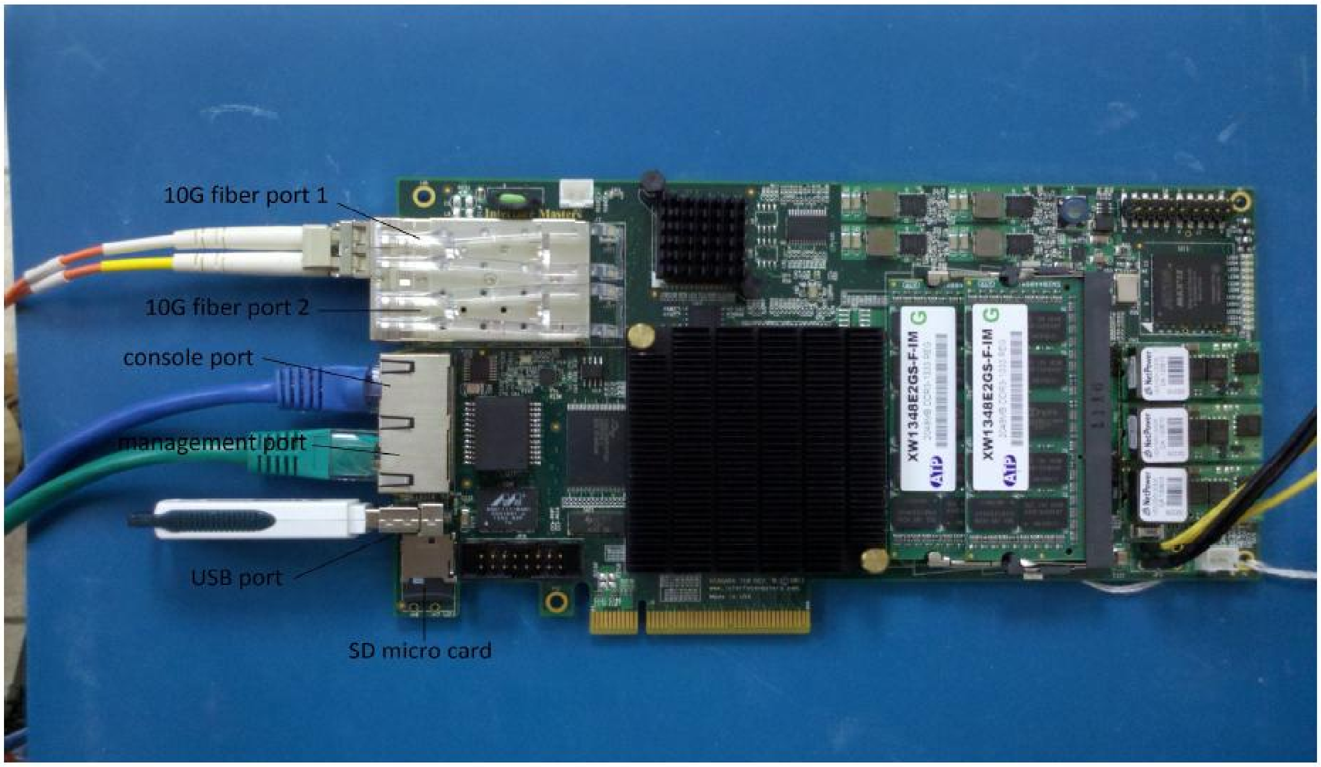


Figure 7: P4080 PCIE Adapter Setup

You will need to connect a terminal emulator to the console port on the P4080 PCIE Adapter as shown in the Figure 7. Settings should be

- speed = 115200, 8 data bits, 1 stop bit, no parity, no flow control.

You must also connect the adapter motherboard 1G connector to a network on which there is a tftp server. The tftp server setup is described below:

- Install tftpd related packages
  ```
  $ sudo apt-get install xinetd tftpd tftp
  ```
- Create /etc/xinetd.d/tftp file and put the following content.
  ```
  service tftp {
  protocol        = udp
  ```

```
port          = 69
socket_type   = dgram
wait          = yes
user          = nobody
server        = /usr/sbin/in.tftpd
server_args   = /tftpboot
disable       = no
}
```

- Create tftpboot directory
  ```
  $ sudo mkdir /tftpboot
  $ sudo chmod –R 777 /tftpboot
  $ sudo chown –R nobody /tftpboot
  ```
- Start tftpd using xinetd
  ```
  $ sudo /etc/init.d/xinetd start
  ```

Finally, you must connect the other P4080 PCIE 10G Ethernet interfaces to whatever hardware you will use to generate traffic to the USDPAA application.

6.5 Files Needed to Boot Linux on the P4080 PCIE Adapter

To run the USDPAA software, one must first boot Linux with the correct files. The ltib build leaves most of these files in directory rootfs/boot under P4080-PCIE-PRODUCTION-10-19-2011/ltib-e500mc-20110331. The paths below are relative to this directory.

- rootfs/boot/p4080n710/R_PPSXX_0x2/**rcw_0x2_rev2_high_ep_1.3GHz_clock.bin**
  Reset configuration word (rcw) that selects SerDes Protocol. Use the bin that reflects your configuration and is relevant for your use case. This file is suitable to program into the P4080 PCIE Adapter NOR flash. In the high speed bin, the P4080 cores run at 1.3 GHz.
- rootfs/boot/**u-boot.bin**
  U-Boot bootloader image suitable to program into the P4080 PCIE Adapter NOR flash.
- **fsl_fman_ucode_P3_P4_P5_101_8.bin**
  FMan microcode for P4080 rev. 2 silicon suitable to program into the P4080 PCIE Adapter NOR flash.
- rootfs/boot/**uImage**
  Linux kernel supporting USDPAA.
- rootfs/boot/**p4080n710-usdpaa.dtb**
  Device tree file configured for USDPAA usage.
- **initramfs.cpio.gz.uboot**

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 on the card. The first three must be programmed into the 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 P4080 PCIE Adapter 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.

6.6 Programming P4080 PCIE Adapter NOR Flash Bank 0

The P4080 PCIE Adapter 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 “n710_reset altbank” from the bank 0 u-boot prompt to boot into bank 4.

It is recommended 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. Programming Bank 0 should only be used, once the images have been validated in bank 4.

U-boot Boot Text

Below is a reference text of u-boot bootup

```
U-Boot MPS N710 2010.12 (Sep 20 2011 - 13:24:26)

CPU0:  P4080E, Version: 2.0, (0x82080020)
Core:  E500MC, Version: 2.0, (0x80230020)
Clock Configuration:
       CPU0:1333.333 MHz, CPU1:1333.333 MHz, CPU2:1333.333 MHz, CPU3:1333.333 M
       CPU4:1333.333 MHz, CPU5:1333.333 MHz, CPU6:1333.333 MHz, CPU7:1333.333 M
       CCB:666.667 MHz,
       DDR:666.667 MHz (1333.333 MT/s data rate) (Asynchronous), LBC:83.333 MHz
       FMAN1: 500 MHz
       FMAN2: 500 MHz
       PME:   333.333 MHz
L1:    D-cache 32 kB enabled
       I-cache 32 kB enabled
Sys ID: 0x0711, Board Version 0x 0, CPLD Ver: 0x0336-bit Addressing
Reset Configuration Word (RCW):
       00000000: 10600000 00000000 20201820 0000cccc
       00000010: 08402200 3c3c2000 de800000 61000000
       00000020: 00300000 00000000 00000000 008b0000
       00000030: 00000000 00000000 00000000 00000000
I2C:   ready
DRAM:  Initializing....using SPD
Detected RDIMM(s)
Detected RDIMM(s)
CS2 is disabled.
CS3 is disabled.
CS2 is disabled.
CS3 is disabled.
Implementing recommended DDR3 Init sequence
Enabling controller 0
Enabling controller 1
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

Relocating u-boot to 0x7ff30000
POST memory PASSED
FLASH: 128 MiB
L2:    128 KB enabled
Corenet Platform Cache: 2048 KB enabled
p4080_erratum_serdes8 enable Bank3
p4080_erratum_serdes8 enable Bank2
p4080_erratum_serdes9 reset Bank2
p4080_erratum_serdes9 reset Bank3
MMC:  FSL_ESDHC: 0
EEPROM: N710 v2
PCIe1: Endpoint,  with errors.  Clearing.  Now 0x00000000
Setting up PCIE Outbound Regions
        Region 0:
        bus start = 0x0
        bus end = 0x20000000
        phys_start = 0xc00000000
Outbound memory range: 0:20000000
PCICSRBAR @ 0xff000000
```

```
PCI reg:0 0000000c00000000:0000000000000000 0000000020000000 00000000
PCI reg:1 0000000ffe000000:00000000ff000000 0000000001000000 00000100
x8, regs @ 0xfe200000
fsl_pci_config_unlock CFG_READY
PCIe1: Bus 00 - 00
In:    serial
Out:   serial
Err:   serial
Net:   Fman: Uploading microcode version 101.8.0.
Dtsec: PHY is Marvell MV88E1111 (1410cc2)
AEL2020 Config Phy
AEL2020 Config Phy ERRATUM_SERDES9
        AEL2020 FM1 Phy
...waiting for ael2020 reset to clear
AEL2020 1.C001 = 0x2
10GE Port 0 SFP Module Detected, TX Enabled
Fman: Uploading microcode version 101.8.0.
AEL2020 Config Phy
AEL2020 Config Phy ERRATUM_SERDES9
        AEL2020 FM2 Phy
...waiting for ael2020 reset to clear
AEL2020 1.C001 = 0x1
10GE Port 1 SFP Module Detected, TX Enabled
FM1@DTSEC2, FM1@TGEC1, FM2@TGEC1
ADT7461 Detected
STTS2002 SPD 0 Detected
STTS2002 SPD 1 Detected
N710 Booted from Primary Flash Bank 0
Hit any key to stop autoboot:  0
```

Look at the u-boot output. The line

```
N710 Booted from Primary Flash Bank0
```

shows the bank from which u-boot was booted. In this case it was bank 0.

Images copied to TFTP Server

- Copy P4080 images into tftp directory on some Host machine. Assume *"/tftp"* is the directory.
    - Copy RCW

      ```
      $ cp P4080-PCIE-PRODUCTION-10-19-2011/ltib-e500mc-20110331/rootfs/boot/p4080n710/R_PPSXX_0x2/rcw_0x2_rev2_high_ep_1.3GHz_clock.bin /tftp
      ```

- Copy U-boot

  ```
  $cp    P4080-PCIE-PRODUCTION-10-19-2011/ltib-e500mc-20110331/rootfs/boot/u-boot.bin /tftp
  ```

- Copy device tree

  ```
  $cp    P4080-PCIE-PRODUCTION-10-19-2011/ltib-e500mc-20110331/rootfs/boot/p4080n710-usdpaa.dtb /tftp
  ```

- Copy Linux image

  ```
  $cp    P4080-PCIE-PRODUCTION-10-19-2011/ltib-e500mc-20110331/rootfs/boot/uImage /tftp
  ```

- Copy root file system

  ```
  $cp    P4080-PCIE-PRODUCTION-10-19-2011/ltib-e500mc-20110331/initramfs.cpio.gz.uboot /tftp
  ```

Programming Bank 0

- Power on the Host with the adapter. The adapter's boot up process will be started and it will
  display a count down message from 3 to 0 on Terminal. Press the enter key to abort normal auto boot, which should get you to a command:
  prompt: "=>".

- **Boot into Bank 4**

  ```
  =>  n710_reset altbank
  ```

  - At the u-boot prompt, Setup environment variables and verify the Ethernet port between adapter and Host by typing the following:

    ```
    => setenv ethact FM1@DTSEC2
    => setenv serverip 192.168.2.55
    => setenv gatewayip 192.168.2.1
    => setenv ipaddr 192.168.2.10
       => ping $serverip
       Using FM1@DTSEC2 device
       host 192.168.2.55 is alive
    ```

Interrupt the bootloader and execute the following *commands*

- Transfer RCW image through tftp

  ```
  tftp 0x02000000 rcw_0x2_rev2_high_ep_1.3GHz_clock.bin
  ```

- Program RCW image into flash using following commands

  ```
  protect off 0xec000000 0xec01ffff

  erase 0xec000000 +$filesize

  cp.b 0x02000000 0xec000000 $filesize
  ```

- Transfer uboot image through tftp

  ```
  tftp 0x01000000 u-boot.bin
  ```

- Program U-boot image into flash using following commands

  ```
  protect off 0xebf80000 0xebffffff

  erase 0xebf80000 +$filesize

  cp.b 0x01000000 0xebf80000 $filesize
  ```

- Transfer Linux image through tftp

  ```
  tftp 0x1000000 uImage
  ```

- Program Linux image into flash using following commands

  ```
  protect off 0xec020000 0xec6fffff

  erase 0xec020000 +$filesize

  cp.b 0x1000000  0xec020000 $filesize
  ```

- Transfer root filesystem image through tftp

  ```
  tftp 0x1000000  initramfs.cpio.gz.uboot
  ```

- Program root filesystem into flash using following commands

  ```
  protect off 0xed300000 0xeeffffff

  erase 0xed300000 +$filesize

  cp.b 0x1000000 0xed300000 $filesize
  ```

- **Boot into Bank 0**:

  ```
  n710_reset
  ```

  - At the u-boot prompt, Setup environment variables and verify the Ethernet port between adapter and Host by typing the following:

```
=> setenv ethact FM1@DTSEC2
=> setenv serverip 192.168.2.55
=> setenv gatewayip 192.168.2.1
=> setenv ipaddr 192.168.2.10
   => ping $serverip
   Using FM1@DTSEC2 device
   host 192.168.2.55 is alive
```

Interrupt the bootloader and execute the following commands

- Transfer Device tree image through tftp

  `tftp 0x01000000  p4080n710-usdpaa.dtb`

- Program device tree image into flash using following commands

  `protect off 0xe8800000 0xe88fffff`

  `erase 0xe8800000 +$filesize`

  `cp.b 0x01000000 0xe8800000 $filesize`

- Boot into Bank 0:

```
➪ n710_reset
```

6.7 Programming the P4080 PCIE Adapter NOR Flash Bank 4

Transfer the 6 images to the tftp server on the host as described in Section 6.5.
First, boot from bank 0 by doing a reset or power-on. Check that you see "Booted from Bank 0" since this is very important. Then, set network parameters using the u-boot environment variables described in Section 6.6.

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
```

- **Boot into Bank 0**, interrupt the bootloader and execute the following commands
  - Program RCW image into flash using following commands
    - `tftp 0x02000000  rcw_0x2_rev2_high_ep_1.3GHz_clock.bin`
    - `protect off 0xec000000 0xec01ffff`
    - `erase 0xec000000 +$filesize`

- `cp.b 0x02000000 0xec000000 $filesize`

- Program U-boot image into flash using following commands
  - `tftp 0x01000000 u-boot.bin`
  - `erase 0xebf80000 +$filesize`
  - `cp.b 0x01000000 0xebf80000 $filesize`
- Program Linux image into flash using following commands
  - `tftp 0x1000000 uImage`
  - `protect off 0xec020000 0xec6fffff`
  - `erase 0xec020000 +$filesize`
  - `cp.b 0x1000000  0xec020000 $filesize`
- Program root filesystem into flash using following commands
  - `tftp 0x1000000  initramfs.cpio.gz.uboot`
  - `protect off 0xed300000 0xedffffff`
  - `erase 0xed300000 +$filesize`
  - `cp.b 0x1000000 0xed300000 $filesize`

- **Boot into Bank 4**:

```
=>  n710_reset altbank
```

Interrupt the bootloader and execute the following commands

  - `protect off 0xe8800000 0xe88fffff`
  - `tftp 0x01000000  p4080n710-usdpaa.dtb`
  - `erase 0xe8800000 +$filesize`
  - `cp.b 0x01000000 0xe8800000 $filesize`

- Boot into bank 4:

```
=>  n710_reset altbank
```

6.8 Boot into Bank 4 and Set More Variables

Next, enter the command "n710_reset 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.

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; bootm 0xe8020000
0xe9300000 0xe8800000
saveenv
```

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
```

6.9 Booting Linux

At this point, you need to reset once again into bank 4 (u-boot command "n710_reset 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".

"ifconfig -a" should show only one FMan (fm) ethernet interface, fm1-gb1. This is the P4080 PCIE Adapter motherboard 1 G ethernet interface. You can set its IP address and use it as an ordinary Linux network interface if you wish.

7 P4080 PCIE Adapter Usage

7.1 P4080 PCIE Packet Process Application and x86 Host Drivers

Startup

1. After Linux on P4080 boots up, a login prompt will be displayed. Use username “root” and password “root” to login.
2. To start the necessary initialization and start the PCIE Packet Process on P4080 Linux, run the startup script:

   ```
   $ /etc/rc.d/init.d/usdpaa_pcie start
   ```

3. After this, load the Host device drivers on x86 host to complete the initialization handshake between P4080 application and Host Packet Driver.

   a. Load the Packet driver :

   ```
   $ insmod x86/drivers/pcie-core/src/pcie-core.ko
   ```

   b. Load the Ethernet driver:

   ```
   $ insmod x86/drivers/p4080-eth/src/p4080eth.ko
   ```

   c. Two 10G interfaces will be created as a result of above instructions. Check this using *“`ifconfig -a`”* command.

   d. Bring UP the 10G interface(s) and assign IP address:

   ```
   $ ifconfig <interface-name> 10.0.0.1
   ```

   e. Ping any remote machine connected to network.

4. After completion of initialization handshake, CLI client on the P4080 PCIE Adapter can be started by executing the following in Linux:

   ```
   $ /usr/bin/cli_client
   ```

   Note that only after completion of initialization handshake, can the CLI client be started.

7.1 Jumbo Frame Support

To get the Jumbo Frame support, Ethernet Driver has to be loaded with a command line argument "jumbo_frames". Load the Ethernet driver using:

```
$ insmod x86/drivers/p4080-eth/src/p4080eth.ko jumbo_frames=1
```

7.2 Flow classification

The default FMC configuration and policy files built into rootfs from *"P4080-PCIE-PRODUCTION-10-19-2011/ltib-e500mc-20110331/rpm/BUILD/usdpaa-0.3.0/apps/ppac"* directory. FMC configuration file us_config_serdes_0x2.xml sets the first 10G port to use the policy **hash_5tuple_policy1** and **hash_5tuple_policy2** for second 10G port. The policy file us_policy_hash_ipv4_src_dst.xml declares distributions **hash_5tuple_dist1** and **hash 5tuple_dist2** for the policy **hash_5tuple_policy1 and hash_5tuple_dist2** respectively to classify the TCP traffic.

The file has to be modified appropriately if UDP flow classification is required.

The FMC configuration file is configured as follows for setting up FMAN's 10G interface 1.

```
<cfgdata>
    <config>
      <engine name="fm0">
            <port type="10G" number="0" policy="hash_5tuple_policy1" />
      </engine>
      <engine name="fm1">
            <port type="10G" number="0" policy="hash_5tuple_policy2" />
      </engine>
    </config>
</cfgdata>
```

The FMC policy file is configured as follows for setting up the TCP distribution.

```
  <policy name="hash_5tuple_policy1">
  <dist_order>
        <distributionref name="hash_5tuple_dist1" />
        <distributionref name="default_5tuple_dist1" />
     </dist_order>
</policy>

    <distribution name="hash_5tuple_dist1">
        <queue count="8" base="0xe00"/>
        <key>
```

```
            <fieldref name="ipv4.src"/>
            <fieldref name="ipv4.dst"/>
            <fieldref name="tcp.sport"/>
            <fieldref name="tcp.dport"/>
        </key>
    </distribution>
```

7.3 Usage of 10G interfaces

When x86 Host boots up, the P4080 PCIE Adapter is shown as PCIe card in the output of *"lspci"* command. The verbose output of the P4080 PCIE Adapter as endpoint device is as follows.

```
01:00.0 Power PC: Freescale Semiconductor Inc P4080E (rev 20) (prog-if 01)
        Control: I/O+ Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr- Stepping- SERR+ FastB2B- DisINTx+
        Status: Cap+ 66MHz- UDF- FastB2B- ParErr- DEVSEL=fast >TAbort- <TAbort- <MAbort- >SERR- <PERR- INTx-
        Latency: 0, Cache Line Size: 64 bytes
        Interrupt: pin A routed to IRQ 29
        Region 0: Memory at f8000000 (32-bit, non-prefetchable) [size=16M]
        Region 1: Memory at f0000000 (32-bit, prefetchable) [size=64M]
        Capabilities: [44] Power Management version 3
                Flags: PMEClk- DSI- D1+ D2+ AuxCurrent=0mA PME(D0+,D1+,D2+,D3hot+,D3cold-)
                Status: D0 PME-Enable- DSel=0 DScale=0 PME-
        Capabilities: [4c] Express (v2) Endpoint, MSI 00
                DevCap: MaxPayload 256 bytes, PhantFunc 0, Latency L0s <64ns, L1 <1us
                        ExtTag- AttnBtn- AttnInd- PwrInd- RBE+ FLReset-
                DevCtl: Report errors: Correctable- Non-Fatal- Fatal+ Unsupported-
                        RlxdOrd+ ExtTag- PhantFunc- AuxPwr- NoSnoop+
                        MaxPayload 128 bytes, MaxReadReq 4096 bytes
                DevSta: CorrErr- UncorrErr- FatalErr- UnsuppReq- AuxPwr- TransPend-
                LnkCap: Port #0, Speed 5GT/s, Width x8, ASPM L0s, Latency L0 <2us, L1 unlimited
                        ClockPM- Suprise- LLActRep- BwNot-
                LnkCtl: ASPM Disabled; RCB 64 bytes Disabled- Retrain- CommClk-
                        ExtSynch- ClockPM- AutWidDis- BWInt- AutBWInt-
                LnkSta: Speed 2.5GT/s, Width x8, TrErr- Train- SlotClk- DLActive- BWMgmt- ABWMgmt-
        Capabilities: [88] Message Signalled Interrupts: Mask- 64bit+ Queue=0/4 Enable+
                Address: 00000000fee0300c  Data: 41c1
        Capabilities: [100] Advanced Error Reporting <?>
        Kernel driver in use: p4080-ep
```

In the above output, memory regions exposed by P4080 for use by the host are BAR0 and BAR1. This is highlighted in green color. Region 0 refers to BAR0 and Region 1 refers to BAR1. Region 0 opens window to 16MB CCSR space of P4080. Region 1 opens window to 64MB memory of P4080 which includes initialization handshake memory, static and dynamic buffer pool memory. P4080 Adapter's inbound ATMUs are configured to translate from host I/O address space assigned to BAR0 and BAR1 to corresponding P4080 memory address range.
MSI capability is initialized and can be observed from the lines highlighted in grey color.

The x86 Host Packet Driver directly interfaces with the P4080 PCIE Adapter using the Linux Kernel PCIe subsystem. The Ethernet device driver uses the x86 Host packet driver APIs to interact with the adapter. To load the x86 Host drivers follow the following steps.

- Load the Packet driver using:

  ```
  insmod x86/drivers/pcie-core/src/pcie-core.ko
  ```

- Load the Ethernet driver using:

  ```
  insmod x86/drivers/p4080-eth/src/p4080eth.ko
  ```

- Two 10G interfaces will be created as a result of above instructions. Check this using *"`ifconfig -a`"* command.
- Bring UP the 10G interface(s) and assign IP address using "`ifconfig <interface-name> 10.0.0.1`".
- Ping any remote machine connected to network.

7.4 Packet Statistics Using CLI on P4080

The P4080 PCIE Adapter has a management Ethernet port that provides serial console access into P4080 Linux. This can be used to setup mincom to some external host. After booting into Linux run *`$/usr/bin/cli_client`*
to start the CLI on P4080. This CLI offers the following commands.

?	Displays command help
help	Displays command help
tx-fm1-10g	Displays statistics of Tx Frame Queues associated to 10G interface of FMAN-1
rx-default-fm1-10g	Displays statistics of Rx Default Frame Queues associated to 10G interface of FMAN-1
rx-pcd-fm1-10g	Displays statistics of Rx PCD Frame Queues associated to 10G interface of FMAN-1

• **msi-interrupts-fm1-10g**	Displays statistics of interrupts associated to 10G interface of FMAN-1
• **tx-fm2-10g**	Displays statistics of Tx Frame Queues associated to 10G interface of FMAN-2
• **rx-default-fm2-10g**	Displays statistics of Rx Default Frame Queues associated to 10G interface of FMAN-2
• **rx-pcd-fm2-10g**	Displays statistics of Rx PCD Frame Queues associated to 10G interface of FMAN-2
• **msi-interrupt-fm2-10g**	Displays statistics of interrupts associated to 10G interface of FMAN-2
• **stop-packet-process**	Stops the packet driver application gracefully
• **exit**	Exits from CLI prompt
• **q**	Exits from CLI prompt

Sample output for Tx Packet statistics

```
cli> get-tx-stats-1
+--------------------------------------------------------------+
|CoreId       Tx FQId      Tx Packets        Tx Bytes          |
+--------------------------------------------------------------+
|Core-0       536            0                          0      |
|Core-1       537         5897                      579521     |
|Core-2       538            0                          0      |
|Core-3       539            0                          0      |
|Core-4       540            0                          0      |
|Core-5       541            6                          468    |
|Core-6       542            2                          84     |
|Core-7       543            0                          0      |
+--------------------------------------------------------------+
```

Sample output for Rx default Frame Queue statistics

```
cli> get-rx-default-stats-1
+----------------------------------------------------------------------------------------
------+
|CoreId       Rx FQId      Rx Packets        Rx Bytes       Rx Drop Packets        Rx
Drop Bytes |
+----------------------------------------------------------------------------------------
------+
|Core-0          101          0             0               0                  0
|
|Core-1          101          1            60               0                  0
|
|Core-2          101          981          96138            0                  0
|
|Core-3          101          980          96040            0                  0
|
|Core-4          101          980          96040            0                  0
|
|Core-5          101          980          96040            0                  0
|
|Core-6          101          979          95942            0                  0
|
|Core-7          101           980         96002            0                  0
|
+-----------------------------------------------------------------------------------
------+
```

Sample output for Rx PCD Frame Queue statistics

```
cli> get-rx-pcd-stats-1
+-----------------------------------------------------------------------------------
------+
|CoreId       Rx FQId      Rx Packets         Rx Bytes        Rx Drop Packets       Rx
Drop Bytes |
+-----------------------------------------------------------------------------------
------+
|Core-1         3584         0               0                 0                    0
|
|Core-2         3585         0               0                 0                    0
|
|Core-3         3586         0               0                 0                    0
|
|Core-4         3587         0               0                 0                    0
|
|Core-5         3588         0               0                 0                    0
|
|Core-6         3589         0               0                 0                    0
|
|Core-7         3590         59760           3944228           0                    0
|
|Core-1         3591         0               0                 0                    0
|
+-----------------------------------------------------------------------------------
------+
```

7.5 Stopping the interface usage

- **Host Interface Down:** "`ifconfig <interface-name> down`". This disables the corresponding FMan interface on P4080.

- **Host Interface Up :** "`ifconfig <interface-name> up`". This enables the corresponding FMan interface on P4080.

- **Graceful Shutdown of P4080 Packet Driver Application:** These operations bring the system back to its initialized state.

  1. On P4080 execute from CLI:

     ```
     $stop-packet-process
     ```

  2. Unload x86 Host drivers: On host execute "`rmmod p4080eth pciecore`"

     NOTE: It is mandatory to execute step 2 as part of the graceful shutdown as described above. Stopping the P4080 Packet Driver Application and restarting it without unloading x86 host drivers is not supported and will result in unexpected behavior.

- **Restart P4080 Packet Driver Application:**

  On P4080 execute "/etc/rc.d/init.d/usdpaa_pcie start". X86 Host drivers can be loaded after this and the Ethernet interfaces on can be brought UP to send the traffic.

- **Host Drivers Unload/Reload:** When host drivers are unloaded "rmmod p4080eth pciecore", there is no need to explicitly stop the P4080 Packet driver application. The Host drivers can be re-loaded and the traffic can be sent on any of the two 10G interfaces after this.

- Stopping the P4080 PCIe Packet Driver Application using "/etc/rc.d/init.d/usdpaa_pcie stop" is not supported.

7.6 Packet Statistics using sysfs on x86 Host

The x86 Host Ethernet driver exposes sysfs files to show the packet statistics for an Ethernet interface. The statistics can be obtained using *"cat /sys/smart_nic/stats"*. The output of the statistics is as follows.

Sample output for statistics on x86 Host Ethernet interface

```
#cat /sys/smart_nic/stats
Interface eth1 stats:  Ring0    Ring1    Ring2    Ring3    Ring4    Ring5    Ring6  Ring7
```

```
Tx pkt count:          0      5897    0      0      0      6      2      0
Tx pkt Err count:      0      0       0      0      0      0      0      0
Tx byte count:         0      579521  0      0      0      468    84     0
Rx pkt count:          0      1       981    980    980    980    979    980
Rx pkt drop count:     0      0       0      0      0      0      0      0
Rx byte count:         0      46      54936  54880  54880  54880  54824  54870

Interface eth2 stats:  Ring0  Ring1   Ring2  Ring3  Ring4  Ring5  Ring6  Ring7
Tx pkt count:          0      0       0      0      0      0      0      0
Tx pkt Err count:      0      0       0      0      0      0      0      0
Tx byte count:         0      0       0      0      0      0      0      0
Rx pkt count:          0      0       0      0      0      0      0      0
Rx pkt drop count:     0      0       0      0      0      0      0      0
Rx byte count:         0      0       0      0      0      0      0      0
```

7.7 Using PCIE tool on x86 Host

PCIE tool can be used on x86 Host machine to read and write P4080 CCSR values and into any BAR memory region opened by P4080 PCIE endpoint device. To use the tool, run *"P4080-PCIE-PRODUCTION-10-19-2011/x86/utils/src/fsl_pcie_tool"*. Various options to this command are listed below

- -r - Read the register
- -w - Write into the register
- -o - Offset of the register in CCSR space
- -b - Number of BAR register that is to be used for read and write operations
- -D - Display various CCSR register blocks
- -d - Display all the register values in a specific CCSR register block.

Sample output showing CCSR registers of FMan 10G interface

The following command will print all the CCSR blocks with various IDs. These IDs can be used to print a specific block register values.

```
$ x86/utils/src/fsl_pcie_tool  -D

$ x86/utils/src/fsl_pcie_tool  -d  134
```

```
+------------------------------------------------------------------------------------------------------------------------------------------+
|Frame Manager 1 - Ethernet 10GEC 1 Registers                                                                                              |
+------------------------------------------------------------------------------------------------------------------------------------------+
|E10GEC1_EC10G_ID(0x4f0000): 0x0001032c          |E10GEC1_COMMAND_CONFIG(0x4f0008): 0x00020043     |
|E10GEC1_MAC_ADDR_0(0x4f000c): 0x00bd0c00        |E10GEC1_MAC_ADDR_1(0x4f0010): 0x0000d6b1         |
|E10GEC1_MAXFRM(0x4f0014): 0x00002580            |E10GEC1_PAUSE_QUANT(0x4f0018): 0x0000f000        |
|E10GEC1_HASHTABLE_CTRL(0x4f002c): Non-Readable  |E10GEC1_STATUS(0x4f0040): 0x00000000             |
|E10GEC1_TX_IPG_LENGTH(0x4f0044): 0x0000000c     |E10GEC1_MAC_ADDR_2(0x4f0048): 0x00000000         |
|E10GEC1_MAC_ADDR_3(0x4f004c): 0x00000000        |E10GEC1_IMASK(0x4f0060): 0x00010ffb              |
|E10GEC1_IEVENT(0x4f0064): 0x00000000            |E10GEC1_TFRM_U(0x4f0080): 0x00000000             |
|E10GEC1_TFRM_L(0x4f0084): 0x015cece7            |E10GEC1_RFRM_U(0x4f0088): 0x00000000             |
|E10GEC1_RFRM_L(0x4f008c): 0x0001008d            |E10GEC1_RFCS_U(0x4f0090): 0x00000000             |
|E10GEC1_RFCS_L(0x4f0094): 0x00000000            |E10GEC1_RALN_U(0x4f0098): 0x00000000             |
|E10GEC1_RALN_L(0x4f009c): 0x00000000            |E10GEC1_TXPF_U(0x4f00a0): 0x00000000             |
|E10GEC1_TXPF_L(0x4f00a4): 0x00000000            |E10GEC1_RXPF_U(0x4f00a8): 0x00000000             |
|E10GEC1_RXPF_L(0x4f00ac): 0x00000000            |E10GEC1_RLONG_U(0x4f00b0): 0x00000000            |
|E10GEC1_RLONG_L(0x4f00b4): 0x00000000           |E10GEC1_RFLR_U(0x4f00b8): 0x00000000             |
|E10GEC1_RFLR_L(0x4f00bc): 0x00000000            |E10GEC1_TVLAN_U(0x4f00c0): 0x00000000            |
|E10GEC1_TVLAN_L(0x4f00c4): 0x00000000           |E10GEC1_RVLAN_U(0x4f00c8): 0x00000000            |
|E10GEC1_RVLAN_L(0x4f00cc): 0x00000000           |E10GEC1_TOCT_U(0x4f00d0): 0x00000008             |
|E10GEC1_TOCT_L(0x4f00d4): 0x11df76ba            |E10GEC1_ROCT_U(0x4f00d8): 0x00000000             |
|E10GEC1_ROCT_L(0x4f00dc): 0x0049205e            |E10GEC1_RUCA_U(0x4f00e0): 0x00000000             |
|E10GEC1_RUCA_L(0x4f00e4): 0x0001008d            |E10GEC1_RMCA_U(0x4f00e8): 0x00000000             |
|E10GEC1_RMCA_L(0x4f00ec): 0x00000000            |E10GEC1_RBCA_U(0x4f00f0): 0x00000000             |
|E10GEC1_RBCA_L(0x4f00f4): 0x00000000            |E10GEC1_TERR_U(0x4f00f8): 0x00000000             |
|E10GEC1_TERR_L(0x4f00fc): 0x00000000            |E10GEC1_TUCA_U(0x4f0108): 0x00000000             |
|E10GEC1_TUCA_L(0x4f010c): 0x015cecce            |E10GEC1_TMCA_U(0x4f0110): 0x00000000             |
|E10GEC1_TMCA_L(0x4f0114): 0x00000018            |E10GEC1_TBCA_U(0x4f0118): 0x00000000             |
|E10GEC1_TBCA_L(0x4f011c): 0x00000001            |E10GEC1_RDRP_U(0x4f0120): 0x00000000             |
|E10GEC1_RDRP_L(0x4f0124): 0x00000000            |E10GEC1_REOCT_U(0x4f0128): 0x00000000            |
|E10GEC1_REOCT_L(0x4f012c): 0x00492bee           |E10GEC1_RPKT_U(0x4f0130): 0x00000000             |
|E10GEC1_RPKT_L(0x4f0134): 0x0001009a            |E10GEC1_TRUND_U(0x4f0138): 0x00000000            |
|E10GEC1_TRUND_L(0x4f013c): 0x00000000           |E10GEC1_R64_U(0x4f0140): 0x00000000              |
|E10GEC1_R64_L(0x4f0144): 0x00000004             |E10GEC1_R127_U(0x4f0148): 0x00000000             |
|E10GEC1_R127_L(0x4f014c): 0x00010069            |E10GEC1_R255_U(0x4f0150): 0x00000000             |
|E10GEC1_R255_L(0x4f0154): 0x00000003            |E10GEC1_R511_U(0x4f0158): 0x00000000             |
|E10GEC1_R511_L(0x4f015c): 0x0000002a            |E10GEC1_R1023_U(0x4f0160): 0x00000000            |
|E10GEC1_R1023_L(0x4f0164): 0x00000000           |E10GEC1_R1518_U(0x4f0168): 0x00000000            |
|E10GEC1_R1518_L(0x4f016c): 0x00000000           |E10GEC1_R1519X_U(0x4f0170): 0x00000000           |
|E10GEC1_R1519X_L(0x4f0174): 0x00000000          |E10GEC1_TROVR_U(0x4f0178): 0x00000000            |
|E10GEC1_TROVR_L(0x4f017c): 0x00000000           |E10GEC1_TRJBR_U(0x4f0180): 0x00000000            |
|E10GEC1_TRJBR_L(0x4f0184): 0x00000000          |E10GEC1_TRFRG_U(0x4f0188): 0x00000000         |
|E10GEC1_TRFRG_L(0x4f018c): 0x00000000           |E10GEC1_RERR_U(0x4f0190): 0x00000000         |
|E10GEC1_RERR_L(0x4f0194): 0x00000000           |E10GEC1_MDIO_CFG_STAT(0x4f1030): 0x00007400   |
|E10GEC1_MDIO_CTRL(0x4f1034): Non-Readable      |E10GEC1_MDIO_DATA(0x4f1038): 0x00000000       |
|E10GEC1_MDIO_ADDR(0x4f103c): Non-Readable      |                                              |
+----------------------------------------------------------------------------------------------+
```

8 Working with ltib

The “ltib” system is a build environment and package manager for the Freescale SDK. A few tips on ltib usage follow.

- ltib --help : list ltib options
- ltib -m listpkgs : get a list of available packages. The USDPAA package is called “usdpaa”
- ltib --mode prep --pkg usdpaa : unpack usdpaa source code into directory rpm/BUILD. The proof of concept application is rpm/BUILD/usdpaa-0.3.0/apps/reflector.
- ltib : with no options, ltib will just build. For example, you could edit reflector/reflector.c and then just run ltib again to build it. It builds a new Linux file system for use on the p4080.

9 Known Limitations of this Release

- You may run only a single USDPAA application instance at a time. The application may be (usually is) multithreaded.
- When heavy traffic is received on 10G ports for which the policy file has a classification rule that does match the traffic type (for ex: policy file has TCP tuple classification but traffic received is UDP), bman_acquire failed logs are seen on P4080 console. However, this does not impact the traffic. Occasionally the RX rings on host are stalled
- The Ethtool support for the x86 host driver is limited to driver info, ring parameters, statistics, get/set Tx checksum. Link state reporting is not included as a P4080 USDPAA 10G Phy driver isn’t available.

10 C-API

10.1 P4080 USDPAA PCIe Packet Driver API

Initialization and Cleanup

```
/**
 * API: pkt_drv_register_hw_function : Each Hardware Function
 * (10G Ethernet/Security Engine/PME)  has to register
 * with PCIe Driver using this API. This API has to be invoked in the
 * context of Hardware Function initialization function
 * (such as hw_fn_init_eth()).
 * This API copies the Hardware function's Tx callback function and command
 * callback function into PCIe Driver context. The Tx callback function is
 * invoked by the PCIe Driver when it receives a Tx packet from Host. The
 * Command callback function is invoked when there is a command to the Hardware
 * Function from Host.
 *
 * @hw_fns - Hardware Function identifier
 * @hw_data - Pointer to Tx and command callback function pointers
 *
 * @return - SUCCESS (0) on successful registration otherwise ERROR (1)
 */
```

The prototype of the registration API is as follows.
int pkt_drv_register_hw_function(enum hw_fn hw_fns, struct hw_fn_data *hw_data);

- **hw_fns** – Identifier for a hardware function
  enum hw_fns {

  HW_FUNC_CMD,

  HW_FUNC_ETH0,

  HW_FUNC_ETH1,

  HW_FUNC_SEC,

  HW_FUNC_PME

  }

- **hw_data** – Hardware function specific data that is exchanged with the P4080 Packet Driver.

  struct hw_fn_data {

  hw_fn_tx_cb tx_cb;

  hw_fn_cmd_cb cmd_cb;

  };

The prototype of Tx and command ring callback functions are as follows

typedef int (*hw_fn_tx_cb)(struct qm_fd *fd, int cpu_id, int ring_id);

typedef int (*hw_fn_cmd_cb)(struct p4080_fd *fd);

```
/**
 * Callback: hw_fn_tx_cb : Callback function that is registered with PCIe
 * Driver and is invoked in dma_channel_poll() API when DMA transfer for
 * a Tx packet is successful
 *
 * @qm_fd - Qman FD that has to be enqueued into one of the
 * Tx Frame Queues.
 * @cpu_id - Core on which the Tx packet has come. This is used to select
 * the Tx Frame
 * Queue.
 * @ring_id - Used as an index into flow table to get the Frame Queue ID
 * and Ethernet port ID.
 *
 * @return - SUCCESS (0) on successful registration otherwise ERROR (1)
 */

/**
 * Callback: hw_fn_cmd_cb : Callback function that is registered with PCIe
 * Driver and is invoked in pkt_drv_process_command_ring() API when a
 * command is received from Host Packet Driver or Host Hardware Function
 * Driver.
 *
 * @p4080_fd- P4080 FD that  holds command data
 *
 * @return - SUCCESS (0) on successful registration otherwise ERROR (1)
 */

/**
 * API: pkt_drv_unregister_hw_function : Each Hardware Function has to
 * unregister with PCIe Driver using this API. This API has to be
 * invoked in the context of Hardware Function cleanup
 * function (such hw_fn_clean_eth()). This API clears the Tx callback
 * function and command callback function of the PCIe Driver context.
 *
 * @hw_fns - Hardware Function identifier
 *
 * @return - SUCCESS (0) on successful de-registration otherwise ERROR (1)
 */
```

The prototype of the de-registration API is as follows.

void pkt_drv_unregister_hw_function(enum hw_fn hw_fns);

- **hw_fns** – Identifier for a hardware function

  enum hw_fns {

  HW_FUNC_CMD,

  HW_FUNC_ETH0,

  HW_FUNC_ETH1,

  HW_FUNC_SEC,

  HW_FUNC_PME

  }

```
/**
 * API: pkt_drv_init : The PCIe Driver initialization is part of the global
 * initialization. This API has to be invoked by the primary USDPAA thread
 * at the end of global initialization. The initialization done in this API
 * includes handshake with Host PCIe Driver, PCIe memory mapping to user
 * space and Hardware Function specific initialization.
 *
 * @return - SUCCESS (0) on successful initialization otherwise ERROR (1)
 */
```

The prototype of the PCIe Driver initialization is as follows.

int pkt_drv_init(void);

```
/**
 * API: pkt_drv_finish : The PCIe Driver cleanup is part of the global
 * cleanup done by the USDPAA primary thread. This API is invoked by the
 * PCIe application when it is about to exit.
 *
 */
```

void pkt_drv_finish(void);

Data Path

```
/**
 * API: pkt_drv_host_poll : All USDPAA PCIe threads, including primary
 * thread, poll on the Host Tx rings for accessing Tx Frame Descriptors.
 * Each thread polls on a specific Tx ring that is assigned to it at
 * the time of initialization.
 *
 * The Tx
 * FDs obtained from the Tx rings are used to initiate a DMA transfer of the
 * host Tx data buffer. On completion of the DMA, the host_poll_cb is invoked
 * to complete the Tx datapath processing by the corresponding hardware
 * function driver. For example: typically for a network packet, the Ethernet
 * driver would enqueue the FD into
 * Tx FQ associated to the Tx ring.
 *
 * @cpu - Identifier of the CPU on which the PCIe Driver thread is
 * running. This is used as an indexing parameter or various data
 * structures
 */
```

The prototype of the Tx ring processing API is as follows.

void pkt_drv_host_poll(int cpu);

```
/**
 * API: pkt_drv_tx_confirm : A Hardware Engine like FMan can be configured to
 * send Tx confirmation via QMan to the originator of the Tx frame after
 * sending the packet out of the interface.  The QMAN Driver
 * calls a pre-registered callback of Hardware Function library to notify
 * the Tx confirmation.  The Hardware function library
 * should in turn invoke this API to notify the Tx confirmation to Host Packet
 * Driver.
 * @hw_fn_id - Identifier of a Hardware Function. The value is chosen
 * from "enum hw_fns"
 * @ring_id - Identifier of Host Tx ring. The range of values is
 * "[0, Number of Tx rings - 1]"
 */
```

The prototype of the Tx confirmation processing API is as follows.

void pkt_drv_tx_confirm(int hw_fn_id, int ring_id);

```
/**
 * API: pkt_drv_enqueue_host_ring :

 * This API should be invoked in by the hardware function drivers to
 *
 * transfer an  Rx FD to Host PCIe Driver. The receive path that would
 * typically involve this when the hardware function driver's callback is
 * invoked by the QMAN driver to notify of a received packet a Frame Queue. For
 * example: for the Ethernet driver, the QMan Driver invokes the Rx default or
 * PCD Frame Queue callback functions when there is a Rx packet from the
 * interface These QMAN invoked callback functions are registered with QMan
 * Driver by the Hardware Function library.
 * This API initiates the DMA of data from P4080 to Host memory.
 * The completion of the Rx DMA is detected via the  DMA poll  API
 * dma_channel_poll() which then  sends an MSI interrupt.

 * @qman_fd - FD received from QMan Driver.
 * @hw_fn_id - Identifier of Hardware Function. The value is chosen from
 * "enum hw_fns"
 * @ring_id - Identifier of Rx ring on Host. The range of values is
 * "[0, Number of Rx rings - 1]"
 */
```

The Prototype of this API is as follows.

int pkt_drv_enqueue_host_ring(const struct qm_fd *qman_fd, int hw_fn_id, int ring_id);

```
/**
 * API: pkt_drv_process_command_ring : Dequeues commands from command
 * Tx ring   * and calls the command ring callback of corresponding Hardware
Function.
 * The Hardware Function callback is registered with to the  USDPAA PCIe Driver
 * at the time of registration. This callback implements necessary action
 * that is specific to a Hardware Function.
 */
```

void pkt_drv_process_command_ring(void);

```
/**
 * API: pkt_drv_enqueue_cmd_response : This API  enqueues
 * command response into  * Command ring.
 * The "type" and "sub_type" fields of "struct p4080_fd"
 * are modified to match the response to a request.
 * Every hardware function driver is mandated to invoke this API immediately
 * in the context of the cmd_poll_cb to ensure in-order delivery of  command
 * responses to the host.  It should be noted that invocation of this API does
 * not signal completion of command processing. It is simply a mechanism to
 * enqueue responses in order. Upon completion of a command request, the
 * hardware function driver should invoke pkt_drv_confirm_cmd_response().
 *
 * @cmd_fd - Command request FD that is used to form the command response
 *
 * @return - Returns pointer to the response FD that is enqueued into
 * Rx command ring on Host. This needs to be passed as an argument
 * in the pkt_drv_confirm_cmd_response()
to mark the FD as
 * "completed request". This eliminates out-of-order processing of
 * command request.
 */
```

The prototype of this API is as follows.

struct p4080_fd *pkt_drv_enqueue_cmd_response(struct p4080_fd *cmd_fd);

- **cmd_fd:** FD used to form the response command FD. It returns the modified form of *"cmd_fd"* with *"type"* and *"sub_type"* fields set accordingly.

```
struct p4080_fd {
        u16 type;
        u16 sub_type;
        u32 __reserved;
        u64 phys_addr_of_buff;
        u64 cookie;
        struct qm_fd qman_fd;
}
```

  - **type** – Hardware function type
  - **sub_type** – For denoting sub commands in a Hardware Function
  - **__reserved** – Used for SKB headroom and tail rooms. Also used for marking command completion status.
  - **phys_addr_of_buff** – Physical address of SKB data
  - **cookie** – Virtual address of the SKB
  - **qman_fd** – Qman File Descriptor.

```
/**
 * API: pkt_drv_confirm_cmd_response : It confirms the completion of the
 * command processing in the "__reserved" field of the "struct p4080_fd"
 * structure by setting it to "1". Prior to invoking this API, the caller
 * needs to have invoked the pkt_drv_enqueue_cmd_response() API to ensure
 * in-order delivery of the command response.
 * Though the  pkt_drv_enqueue_cmd_response()
 * API enqueues the response commands FD into the Rx command ring, it
 * doesn't confirm command completion. This API enables the Host to see the
 * command response.
 *
 * @cmd_fd - Pointer to the FD whose completion status has to be set.
 */
```

The prototype of this API is as follows.

void pkt_drv_confirm_cmd_response(struct p4080_fd *fd);

- **cmd_fd:** FD for which the *"__reserved"* field is set to notify that this response command can be processed.

DMA

```
/**
```

```
 * API: dma_controller_init - Gets the CCSR map of DMA controller and
 * initializes the DMA channel attributes to enable the DMA transfer. This
 * API must be invoked before using any of the other DMA transfer APIs.
 * This is invoked by a USDPAA Application process process
 * in its main() function
 *
 * @return - SUCCESS (0) on successful initialization, otherwise ERROR (1)
 */
```

The prototype of this API is as follows.

int dma_controller_init(void);

```
/**
 * API: dma_copy - Initiates and checks for completion of a basic direct
 * DMA transfer. It blocks indefinitely on the DMA completion status
 * using dma_check() function call. After invoking dma_controller_init()
 * API, this API can be invoked for DMA transfer provided there are
 * physical addresses of source and destination memory locations.
 *
 * @dest - Physical address of the destination memory location
 * @src - Physical address of the source memory location
 * @xfer_size - Size of the DMA transfer
 * @cont_num - Number of DMA controller
 * @chan_num - Number of DMA channel
 *
 * @return - SUCCESS (0) on successful DMA transfer, otherwise ERROR (1)
 */
```

The prototype of this API is as follows.

int dma_copy(u64 dest, u64 src, u32 xfer_size, int cont_num, int chan_num);

```
/**
 * API: dma_ring_init - Allocates the memory for "count" number of DMA
 * link descriptors, chains the descriptors and initializes DMA channel
 * registers for chaining mode. This API has to be invoked in pkt_drv_init()
 * API which is invoked as part of global initialization. Each USDPAA thread
 * gets one DMA ring by invoking this API.
 *
 * @dring - Pointer to DMA ring context
 * @dma_chan - Pointer to CCSR DMA channel registers
 * @count - Number of expected entries in DMA ring
 *
 * @return - SUCCESS (0) on successful initialization, -ENOMEM on
 * failure to allocate ring memory
 */
```

The prototype of this API is as follows.

int dma_ring_init(struct dma_ring *dring, struct dma_channel *dma_chan, itn count);

```
struct dma_ring {
        u32 head;
        u32 tail;
        u32 free;
        u32 count;
        u8 *ring_mem;
        u32 ring_memsize;
        struct dma_channel *dma_chan;
        struct dma_sw_desc *ring_base;
};
```

- head – Head index to the DMA descriptor ring.
- tail – Tail index to the DMA descriptor ring.
- free – Index representing first free DMA descriptor
- count – Maximum number of DMA descriptors in the DMA ring.
- ring_mem – Pointer to the allocated physical memory for DMA ring
- ring_memsize – Total memory of the DMA ring
- dma_chan – CCSR reference to DMA channel
- ring_base – Aligned start address from *“ring_mem”*

```
struct dma_channel {
        u32 mr;
        u32 sr;
        u32 eclndar;
        u32 clndar;
        u32 satr;
        u32 sar;
        u32 datr;
        u32 dar;
        u32 bcr;
        u32 enlndar;
        u32 nlndar;
        u32 eclsdar;
        u32 clsdar;
        u32 enlsdar;
        u32 nlsdar;
        u32 ssr;
        u32 dsr;
        u8 reserved1[60];
}
```

- **mr** – DMA mode register
- **sr** – DMA status register
- **eclndar** – DMA current link descriptor extended address register
- **clndar** – DMA current link descriptor address register
- **satr** – Source attributes register
- **sar** – Source address register
- **datr** – Destination attributes register
- **dar** – Destination address register
- **bcr** – Byte count register
- **enlndar** – Next link descriptor extended address register
- **nlndar** – Next link descriptor address register
- **eclsdar** – Current list descriptor extended address register
- **clsdar** – Current list descriptor address register
- **enlsdar** – Next list descriptor extended address register

- **nlsdar** – Next list descriptor address register
- **ssr** – Source stride register
- **dsr** – Destination stride register
- **reserved1** – Reserved field

```
struct dma_sw_desc {
        struct dma_link_desc link_desc;
        u64 ring_index;
        u32 hw_fn_id;
        u32 operation;
        void *fd;
        void *ring_ptr;
        u8 align[8];
};
```

- **link_desc** – DMA link descriptor format
- **ring_index** – Current index of DMA ring
- **hw_fn_id** – Hardware Function identifier used for generating Rx interrupts
- **operation** – Type of operation. It is of type *"enum dma_operation"*
- **fd** – P4080 Frame Descriptor associated with the software descriptor.
- **ring_ptr** – Pointer to the Tx/Rx ring on Host.
- **align** – Unused field. For future alignment purposes.

```
struct dma_link_desc {
        u32 s_attr;
        u32 s_addr;
        u32 d_attr;
        u32 d_addr;
        u32 nlndea;
        u32 nlnda;
        u32 bcount;
        u32 rsvd;
}
```

- **s_attr** – Source transaction attributes
- **s_addr** – Source address of the DMA transfer
- **d_attr** – Destination transaction attributes
- **d_addr** – Destination attributes of the DMA transfer
- **nlndea** – Points to the next link descriptor in memory. After the DMA controller reads the link descriptor from memory, this field is loaded into the extended next link descriptor address register.
- **nlnda** – Points to the next link descriptor in memory. After the DMA controller reads the link descriptor from memory, this field is loaded into the next link descriptor address register.
- **bcount** – Contains the number of bytes to transfer
- **rsvd** – Reserved field

```
/**
 * API: dma_ring_free - Release memory allocated for DMA ring.
 *
 * @dring - Pointer to the DMA ring of type "struct dma_ring"
 *
 * @return - SUCCESS (0) on successful cleanup, otherwise ERROR (1)
 */
```

The prototype of this API is as follows.

int dma_ring_free(struct dma_ring *dring);

- dring – Pointer to DMA ring that needs to be freed. This memory is taken from the DMA memory (64MB) on P4080.

```
/**
 * API: dma_ring_enq - Enqueues a DMA job to the DMA ring. It picks
 * a software descriptor at the head of the DMA chain, updates the
 * fields of that descriptor. Removes EOLN flag on previous link
 * descriptor and triggers DMA channel start or continue
 *
 * @dring - Pointer to the DMA ring of type "struct dma_ring"
 * @src_addr - Physical address of the source memory location.
 * @dst_addr - Physical address of the destination memory location
 * @len - Size of DMA transfer
 * @op - Operation to be executed at the end of this DMA transfer
 * @p4080_fd - Reference to the FD that triggered this DMA transfer
 * @ring_ptr - Pointer to Host or P4080 FD ring
 * @pi - Ring index on the FD ring
 * @hw_fn_id - Hardware Function identifier
 *
 * @return - 0 on DMA enqueue success, -1 when DMA ring FULL
 */
```

The prototype of this API is as follows.

int dma_ring_enq(struct dma_ring *dring, u64 src_addr, u64 dst_addr, int len, int op, void *p4080_fd, void *ring_ptr, u64 pi, int hw_fn_id);

```
/**
 * API: dma_ring_used_desc - Computes number of DMA descriptors used
 * for DMA operation
 *
 * dring - Pointer to the DMA ring context
 *
 * @return - Number of used DMA descriptors
 */
```

The prototype of this API is as follows.

inline int dma_ring_used_desc(struct dma_ring *dring);

```
/**
 * API: dma_ring_free_desc - Computes number of DMA descriptors available
 * for DMA operation
 *
 * dring - Pointer to the DMA ring context
 *
 * @return - Number of free DMA descriptors
 */
```

The prototype of this API is as follows.

inline int dma_ring_free_desc(struct dma_ring *dring);

10.2 Host PCIE Packet Driver APIs

Initialization and Cleanup

```
/**
 * API: register_hardware_function : Each Upper-level Hardware Function Driver
 *  (10G Ethernet, PME, Security) on Host invokes
 * this API to register itself with the Host Packet Driver.
 *
 * @func_id - Identifier of a Hardware Function.
 * @init_data - Pointer to struct hw_eng_data data structure that is exchanged
 * between Hardware Function Driver and Host Packet Driver for initialization.
 *
 * @return - SUCCESS (0) on successful registration, ERROR (1) on failure.
 */
```

The prototype of this API is as follows.

int register_hardware_function(P4080_HW_FUNCTIONS func_id, struct hw_eng_data *init_data);

- **func_id** – This is the identifier for a Hardware Function that is to be registered with the Host Packet Driver. Its values are taken from *"enum P4080_HW_FUNCTIONS"*.

  typedef enum P4080_HW_FUNCTIONS {

  P4080_HW_MIN = 0,

  P4080_HW_ETH0,

  P4080_HW_ETH1,

  P4080_HW_SEC,

  P4080_HW_PME,

  P4080_HW_MAX

  } P4080_HW_FUNCTIONS;

- **init_data** – The initialization data structure that contains various input and output parameters used by Host Packet Driver during registration.

  typedef struct hw_eng_data {

```
        struct device *pci_dev;

        char pci_info[ETHTOOL_BUSINFO_LEN];

        u64 *msi_status_local;

        u64 *msi_status_remote;

        volatile u64 msi_status;

        p4080_fd_ring_t tx_ring[MAX_NUM_RINGS];

        p4080_fd_ring_t rx_ring[MAX_NUM_RINGS];

        void *priv_cb_arg;

        private_ring_callback *priv_cb;

        cmd_ring_callback *cmd_cb;

        u32 tx_ring_count;

        u32 rx_ring_count;

        u32 fd_size;

    } hw_eng_data_t;
```

- **pci_dev** – PCI Device context that is maintained by the Host Packet Driver and is given to Hardware Function Driver for assisting all kinds of memory mappings.
- **pci_info** – PCI Bus, Device and Function (BDF) information of the Smart NIC Adapter. This information is given to Hardware Function Driver for displaying in the *"ethtool -i <interface name>"*
- **msi_status_local** – Address of the MSI status that is locally maintained in Host memory. This is allocated by Host Packet Driver and the address is shared with Hardware Function Driver.
- **msi_status_remote** – Address of MSI status that exists in P4080 memory. The memory for this is allocated by P4080 Packet Driver and its address is shared with Host Packet Driver during initialization handshake. The Host Packet Driver

shares this address with Hardware Function Drivers during registration. The address is the Host domain's PCIe address that gets translated into P4080 domain's address.

- **msi_status** – Used to differentiate between Rx interrupt and Tx confirmation interrupt. This is set and cleared by the Hardware Function Driver and checked by the Host Packet Driver for knowing the type of interrupt.
- **tx_ring** – Transmit descriptor rings allocated by Host Packet Driver and are shared with Hardware Function Driver.
- **rx_ring** – Receive descriptor rings allocated by Host Packet Driver and are shared with Hardware Function Driver.
- **priv_cb_arg** – Argument pointer for the interrupt callback function.
- **priv_cb** – Rx ring interrupt callback function that is invoked when an interrupt is received for a Hardware Function Driver. The Hardware Function is chosen based on the MSI status value which is maintained for each Hardware Function. Its description is given after this API description.
- **cmd_cb –** Command ring interrupt callback function that is invoked when an interrupt is received for a Hardware Function. Its description is given after this API description.
- **tx_ring_count –** Transmit ring count that is shared with P4080 Packet Driver. This parameter is used to share the transmit ring count with Hardware Function Driver.
- **rx_ring_count –** Receive ring count that is shared with P4080 Packet Driver. This parameter is used to share the receive ring count with Hardware Function Driver.
- **fd_size –** Size of the Host Frame Descriptor.

```
struct p4080_fd_ring {

        void *desc;

        struct net_device_stats stats;

        u64 dma;
        u64 *rpi;
        u64 *rci;
        volatile u64 lpi;
        volatile u64 lci;
        volatile u64 lfi;
        u32 size;
        u32 count;
        u32 fd_size;
        atomic_t free_cnt;
        atomic_t state;
        u32 cpu;
        void *p4080_desc;
} p4080_fd_ring_t;
```

- **desc** – Pointer to the FDs in the Host Tx/Rx ring.
- **stats** – Device specific statistics per ring.
- **dma** – Physical address of the FD ring.
- **rpi** – Producer index located in P4080 memory.
- **rci** – Consumer index located in P4080 memory.
- **lci** – Consumer index located in Host memory.
- **lpi** – Producer index located in Host memory.
- **lfi** – Refill index maintained in Host memory for Rx ring.
- **size** – Length of descriptor ring in bytes
- **count** – Number of descriptors in the ring

- **fd_size** – Size of ring entry.
- **free_cnt -**
- **state** –
- **cpu** – CPU identifier on which the ring is created.
- **p4080_desc** – Pointer to the shadow ring located on P4080.

```
/**
 * Callback: private_ring_callback : Callback function, defined by
 * and registered by the Hardware Function Driver and invoked by
 * Packet  Driver when there is
 * a packet received for that Hardware Function.
 *
 * @arg - Pointer to "struct __ep_dpa_nic_device_s" data structure.
 */
```

The prototype of this callback function is as follows

typedef void (*private_ring_callback)(void *arg);

For the Host Ethernet driver:

- **arg -** Pointer to *"struct __ep_dpa_nic_device_s"* data structure.

struct __ep_dpa_nic_device_s {

struct net_device *netdev;

struct net_device_stats stats;

struct napi_struct napi;

struct delayed_work work;

hw_eng_data_t *net_dev_info;

u32 msi_enable;

u32 state;

struct task_struct *tx_clean_thread;

```
        struct bin_attribute *sysfs_attr;

        hash_table_t mc_htable;

        struct list_head mc_list_head;

} ep_dpa_nic_device_t;
```

- **netdev** – Standard net_device structure pointer.
- **stats** – Device specific statistics
- **napi** – New API context for Rx path bottom half processing
- **work** – Workqueue context used for bottom half processing of Tx path
- **net_dev_info** – Device specific information that is allocated by Hardware Function Driver.
- **msg_enable** – Net device message level
- **tx_clean_thread** – Tx path cleanup thread that runs forever and cleans the Tx ring.
- **sysfs_attr** – Sysfs attribute context
- **mc_hash_table** – Hash table structure for multicast filtering.
- **mc_list_head** – Linked list head of the multicast list.

```
/**
 * Callback: cmd_ring_callback : Callback function, defined  and registered

 * by the Hardware Function Driver. It is invoked by the
 * Packet Driver when there is a command response in the Command ring for
 * that hardware function.
 *
 * @cmd - Pointer to the "struct p4080_fd" context that comes as response
 * from P4080.
 */
```

The prototype of this callback function is as follows

typedef void (*cmd_ring_callback)(struct p4080_fd *cmd);

```
/**
 * API: unregister_hardware_function :
 * Each upper-level Hardware Function driver calls this function to
 * unregister itself from Host Packet Driver. This function is
 * invoked at the time of Driver unload. It sends an unload command to P4080
 * to notify that hardware function Driver is unloaded. The Host
 * Packet Driver cleans all the data structures of the Hardware Function that
 * requested the de-registration.
 *
 * @func_id - Identifier of a Hardware Function.
 */
```

The prototype of this API is as follows.

void unregister_hardware_function(P4080_HW_FUNCTIONS func_id);

- **func_id –** Hardware function identifier based on *"enum P4080_HW_FUNCTIONS"*

```
/**
 * API: get_p4080_address : It converts an address that belongs to Host domain
 * into P4080 PCIE domain address. The converted address will be the
 * pcie address  * in P4080 domain. When  accessed by P4080 software or
hardware,
 * in the P4080->Host direction, the address gets translated into
 * Host domain address
 * using outbound ATMU. The Host address 0x0 is mapped into 0xC_0000_0000 PCIe
 * address on P4080.
 *
 * @host_address - Address belongs to Host domain.
 *
 * @return - Address belongs to P4080 PCIE domain.
 */
```

unsigned long long get_p4080_address(unsigned long long host_address);

- **host_address** – Address in Host domain that needs to be converted into P4080 domain address.

```
/**
 * API: convert_p4080_add_to_host_virt : It converts an address belonging
 * to P4080 PCIE domain into Host domain address. The converted address will be
 * the PCIe address in Host domain that when accessed by host software in
 * the Host->P4080 direction, gets translated into P4080 PCIe domain using
 * inbound ATMU. 64MB DMA memory on P4080 is mapped into Host Domain.
 *
 * @p4080_address - Address belongs to P4080 domain.
 *
 * @return - Address belongs to Host domain.
 */
```

The Prototype of this API is as follows

unsigned long convert_p4080_add_to_host_virt(unsigned long p4080_address);

- **p4080_address** – Address in P4080 domain that needs to be converted into Host domain address.

```
/**
 * API: read_p4080_ccsr : Reads value of a register from P4080's CCSR.
 *
 * @register_offset - Offset of a register in CCSR space
 * @value - Value that is read from CCSR space
 *
 */
```

The prototype of this API is as follows

void read_p4080_ccsr(int register_offset, u32 *value);

- **register_offset** – Offset of a register in CCSR space that needs to be read from CCSR space.
- **value** – Value obtained from the register at offset *"register_offset"* in P4080 CCSR space.

```
/**
 * API: write_p4080_ccsr : Writes a value into P4080 CCSR.
 *
 * @register_offset - Offset of a register in CCSR space
 * @value - Value that is to be written into CCSR space
 *
 */
```

The prototype of this API is as follows

void write_p4080_ccsr(int register_offset, u32 value);

- **register_offset** – Offset of a register in CCSR space that needs to be written.
- **value** – Value that needs to be written into the register at offset *"register_offset"* in P4080 CCSR space.

```
/**
 * API: read64_init_region : Reads value of a memory location in P4080
 * INIT region. The INIT region of 4KB is located at the start of the 64MB
 * DMA memory.
 *
 * @init_offset - Offset of a location in INIT region that needs to be read
 *
 * @return - Returns 64 bit value from INIT region at "init_offset" location
 */
```

The prototype of this API is as follows

u64 read64_init_region(int init_offset);

- **init_offset** – Offset of the location in INIT region that needs to be read.

```
/**
 * API: write64_init_region : Writes a value into a location in P4080
 * INIT region.
 *
 * @init_offset - Offset of a location in INIT region that needs to be read
 * @value - Value to be written into INIT region location.
 *
 */
```

The prototype of the API is as follows

void write64_init_region(int init_offset, u64 value);

- **init_offset** – Offset of a location in INIT region that needs to be written.
- **value** – Value that needs to be written into a location specified by *"init_offset"* in P4080 INIT region.

```
/**
 * API: enqueue_cmd : Enqueues a command into Command Tx ring.
 * Upper-level Hardware Function Drivers on the Host use this API to send
 * a command request to their counterparts on P4080.
 *
 * @cmd - Pointer to a Frame Descriptor that is to be enqueued into
 * Command ring
 *
 * @return - SUCCESS (0) on successful enqueue into Command ring, otherwise
 * ERROR (1)
 */
```

The prototype of this API is as follows

int enqueue_cmd(struct p4080_fd *cmd);

- **cmd** – FD that needs to be enqueued into Command Tx ring.

```
struct p4080_fd {
        u16 type;
        u16 sub_type;
        u32 __reserved;
        u64 phys_addr_of_buff;
        u64 cookie;
        struct qm_fd qman_fd;
}
```

  - **type** – Hardware function type
  - **sub_type** – For denoting sub commands in a Hardware Function
  - **__reserved** – Used for SKB headroom and tail rooms. Also used for marking command completion status.
  - **phys_addr_of_buff** – Physical address of SKB data
  - **cookie** – Virtual address of the SKB
  - **qman_fd** – Qman File Descriptor.

```
/**
 * API: init_region_rx_shadow_offset : To avoid PCIE Reads from P4080 of Host
 * Rx Descriptor rings, P4080 PCIE Framework supports use of Rx Shadow Ring
 * that the Host driver writes to. This API gets the Rx Shadow ring
 *  address from
 * the INIT region. This address belongs to P4080 PCIE address domain.
 *
 * @func_id - Hardware Function identifier
 * @ring_id - Rx ring identifier
 *
 * @return - Offset of the address of Rx shadow ring in INIT region.
 */
```

The prototype of this API is as follows

int init_region_rx_shadow_offset(P4080_HW_FUNCTIONS func_id, int ring_id);

- **func_id** – Hardware function identifier whose ring's address needs to be obtained from INIT region.
- **ring_id** – Ring identifier whose address needs to be obtained from INIT region.

11 References

1. Niagara 710A Hardware Specification, v0.3, Interface Masters

2. P3041/P4080/P5020 BSP User's Guide for B2.3 SDK

1 Glossary

Glossary

Term	Definition
DMA	Direct Memory Access
LWE	Light-weight executive
BMan	Buffer Manager (hardware)
QMan	Queue Manager (hardware)
FMan	Frame Manager (hardware that contains ethernet MACs)
SEC	Security Coprocessor (hardware)
PME	Pattern Match Engine (hardware)
LWE-HV	LWE as implemented in P4080 SDK 2.1. This LWE runs as a guest on the Freescale Embedded Hypervisor.
fmc	FMan configuration application
sdk	software development kit

rcw	Reset Configuration Word - hardware boot-time parameters for the P4080
USDPAA	User Space Data Path Acceleration Architecture
USDPAA Thread	Linux user space thread that has been allocated a QMan and a BMan software portal for direct access.

13 Document History

Table 2 Document History

Rev	Author	Notes
1.0		Initial version.
1.1		Updated to reflect changes of production release on 10/19/2011.

Freescale Semiconductor

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/su
pport

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or
+1-480-768-2130
www.freescale.com/
support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku
Tokyo 153-0064
Japan
0120 191014 or
+81 3 5437 9125
support.japan@freescal
e.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescal
e.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor