Freescale Semiconductor
Application Note

Document Number:AN4760
Rev 0, 10/2013

Frame Manager Configuration Tool Example Configuration and Policy

Contents

1 Introduction

The Frame Manager Configuration Tool (FMC) is a command line tool used to configure Frame Manager's Parser, KeyGen, Controller, and Policer functions

FMC input file examples are provided in the QorIQ SDK. There are separate sets of example files for Linux and USDPAA . When invoking FMC, the following input files (written in NetPDL with Freescale extensions) are required:

- Standard protocol definition file
- Configuration file
- Policy file

A custom protocol file is optional.

The standard protocol definition file and example configuration and policy files are included in Freescale Linux SDK for QorIQ Processors. There is a set of example configuration and policy files for use with Linux and a set of example configuration and policy files for use with User Space DPAA (USDPAA).

For both Linux and USDPAA, this document provides a walk through of example configuration and policy files in order to demonstrate Frame Manager configuration. Additional examples that utilize Frame Manager's coarse classification and policer functions are listed and described. An example custom protocol file is presented last as an advanced topic.

Readers should be familiar with the *Frame Manager Configuration Tool User Guide* and prepared to write FMC configuration and policy files for their application or use case.

2 Terminology and Resources

Table 1. Terminology and Resources

Terminology	Refers to:	Associated resource
DPAA	Data Path Acceleration Architecture	• For a detailed description of DPAA, see *QorIQ Data Path Acceleration Architecture Reference Manual*. • For a review of the conceptual usage of the DPAA refer to the DPAA Primer in the white paper *QorIQ DPAA Primer for Software Architecture*
Ethernet interface	Frame Manager Ethernet MAC	• From an SoC perspective, the set of available interfaces is determined by the hardware reset configuration word (RCW). This is described in the Reset Clocking, and Initialization section of the SoC Reference Manual. Software plays a role in determining the interfaces available for use. • The *QorIQ SDK Ethernet User Manual* describes how to control the subset of Ethernet interfaces that are used by Freescale SDK software.
PCD function	Parse-classify-distribute (PCD) refers to the Frame Manager's (FMan) ability to parse a received frame's protocol headers, perform a classification based on protocol header fields, and distribute or enqueue the received frame to a specific frame queue identifier (FQID) based on the results of the classification.	• It is the FMan's Parser, KeyGen, Controller and Policer that are being used for PCD. FMC is used to configure the FMan to perform the desired PCD function for a given application. • In the absence of configuring FMan for PCD, FMan will enqueue received frames to a default frame queue. • The focus of this document is on "PCD Frame Queues", i.e., FQIDs that the Frame Manager uses when it enqueues a frame to Queue Manager after performing the parse-classify-distribute-police function.
Packet Headers	The examples in this document refer to various fields of standard protocol headers.	• For convenience, packet headers used in the examples are shown in *Appendix A* .
Frame Manager Hardware Port	• FMan supports several types of hardware ports: Ethernet receive (Rx) and transmit (Tx), Offline, and Host. For each Ethernet Interface, there is an associated Rx and Tx port. • This document focuses on Rx ports and Offline ports as these types of ports support FMan Parse, KeyGen, Controller and Policer functions.	*QorIQ Data Path Acceleration Architecture Reference Manual*

3 Using Configuration and Policy Files

A set of example FMC configuration and policy files is included in Freescale Linux SDK for QorIQ Processors. The objective of the example configuration and policy files described in this section is to maintain packet ordering per flow.

3.1 Objective: Maintaining Packet Order Per Flow

A flow can be defined using a packet's header fields. For example, for packets with UDP and IPv4 headers, a flow can be defined as the 5-tuple encompassing the packet's:

- IPv4 source field
- IPv4 destination field
- IPv4 protocol field
- UDP source port field
- UDP destination port field

One way to maintain the order of the packets received for a particular flow is to require that all packets from a flow be enqueued to a single frame queue and that a single core handles all of the packets received on that frame queue.

When the DPAA Ethernet driver initializes a PCD frame queue, it sets the frame queue descriptor's destination work queue field, DEST_WQ, to a dedicated channel. A dedicated channel is serviced exclusively by single software portal. By setting the DEST_WQ to a dedicated channel, a single software portal services the frame queue. This is in contrast to setting the DEST_WQ to a pool channel, where multiple software portals can service the frame queue. The DPAA Ethernet driver guarantees affinity between a software portal and a core. Hence, a single core processes all of the frames enqueued to a specific PCD frame queue. For further information regarding the frame queue initialization performed by the DPAA Ethernet driver, see **The Datapath Acceleration Architecture Linux Ethernet Driver Chapter 1.4**.

Frame Manager performs PCD on Rx frames and computes a FQID for each received frame. When Frame Manager enqueues a frame to Queue Manager it provides the frame descriptor (FD) and FQID. In this example, Frame Manager is configured so that frames belonging to the same flow are enqueued to the same FQID. Since frame queues are treated by Queue Manager as FIFO queues, and the DEST_WQ of each frame queue is a dedicated channel, packet ordering is maintained. For an overview of this technique for maintaining flow order, refer to the **QorIQ DPAA Primer for Software Architecture.**

3.2 Example Usage on Target Board

Policy file `/etc/fmc/config/policy_hash_128fq.xml` and the configuration file `/etc/fmc/config/config_10g.xml` are provided in Freescale Linux SDK. These example files were created with the objective of preserving packet ordering per flow. Example usage on target board:

```
$ cd /etc/fmc/config
$ fmc -c config_10g.xml -p policy_hash_128fq.xml -a
```

It is assumed that `/etc/fmc/config/config_10g.xml` has been appropriately modified for the interfaces that are enabled on the target board. We provide some examples of configuration file modifications below.

NOTE

If you want to examine `config_10g.xml` and `policy_hash_128fq.xml` on your host, these files are included in the SDK package "eth-config". After extracting the source code for the "eth-config" package, the xml files reside in the directory:

```
QorIQ-SDK-<version>-<date>-yocto/build_<target-board>_release/tmp/
work/ppce500mc-fsl-linx/eth-config-git-r3/git/
```

3.3 Walk-through Configuration File `config_10g.xml`

The contents of configuration file `config_10g.xml` are displayed in the code below. In this configuration file, the element **engine** has name “fm0”. Valid engine names are “fm0” and “fm1” corresponding to hardware blocks FMan1 and FMan2. For devices with only one frame manger, only “fm0” can be configured. For devices with two frame managers, “fm0” and/or “fm1” can be configured. In this example, there is one element **engine** with name “fm0” therefore FMan1 is configured.

```
<cfgdata>
    <config>
        <engine name="fm0">
            <port type="10G" number="0" policy="linux_fman_tester_policy_6"/>
        </engine>
    </config>
</cfgdata>
```

The child element **port** has attributes **type** , **number** and **policy**. The attribute values used in the examples are shown in the table below.

Table 2. `config_10g.xml` Child Element Port Example

'port' Attributes	Attribute Value in Example	Allowed Values
type	10G	10G, 1G or OFFLINE
number	0	0,1,2,...,(n-1) where n is the number of ports of that **type** .
policy	linux_fman_tester_policy_6	Individual policies must be defined in the policy file

The policy “linux_fman_tester_policy_6” describes to Frame Manager how it should compute the FQID for frames received on the 10GEC port. Frame Manager uses the computed FQID when it enqueues the Rx frame to Queue Manager.

Using `config_10g.xml` “as is” as input to FMC, network interfaces other than FMan 1’s 10GEC interface are configured for PCD and any traffic received on these other interfaces is not enqueued to the port’s default FQ.

The policy “linux_fman_tester_policy_6” is defined in the policy file `policy_hash_128fq.xml`. This policy file is discussed in the **next section** . For now let’s note that `policy_hash_128fq.xml` defines 16 policies (below) that are intended to be used to configure specific ports. Also listed in the table is a Queue Base, also described in this document. Note that it’s not required to use all of the policies that have been defined in a policy file.

Table 3. `policy_hash_128fq.xml` defines 16 policies

Policy	Port	Queue Base
`linux_fman_tester_policy_0`	FMan1 dTSEC1	`0x3800`
`linux_fman_tester_policy_1`	FMan1 dTSEC2	`0x3880`
`linux_fman_tester_policy_2`	FMan1 dTSEC3	`0x3900`
`linux_fman_tester_policy_3`	FMan1 dTSEC4	`0x3980`
`linux_fman_tester_policy_4`	FMan1 dTSEC5	`0x3a00`
`linux_fman_tester_policy_5`	FMan1 dTSEC6	`0x3a80`
`linux_fman_tester_policy_6`	FMan1 10GEC1	`0x3c00`
`linux_fman_tester_policy_7`	FMan1 10GEC2	`0x3c80`
`linux_fman_tester_policy_8`	FMan2 dTSEC1	`0x7800`
`linux_fman_tester_policy_9`	FMan2 dTSEC2	`0x7880`

Table continues on the next page...

Table 3. `policy_hash_128fq.xml` defines 16 policies (continued)

Policy	Port	Queue Base
linux_fman_tester_policy_10	FMan2 dTSEC3	0x7900
linux_fman_tester_policy_11	FMan2 dTSEC4	0x7980
linux_fman_tester_policy_12	FMan2 dTSEC5	0x7a00
linux_fman_tester_policy_13	FMan2 dTSEC6	0x7a80
linux_fman_tester_policy_14	FMan2 10GEC1	0x7c00
linux_fman_tester_policy_15	FMan2 10GEC2	0x7c80

To recap, after issuing the Linux command `"fmc -p policy_hash_128fq.xml -c config_10g.xml -a"` FMan1 10GEC port is configured for PCD according to policy “linux_fman_tester_policy_6” which is defined in the policy file `policy_hash_128fq.xml`.

3.4 Walk-through Policy File `8c-128fq-p.xml`

From policy file `policy_hash_128fq.xml`, the policy and distribution elements applicable to “linux_fman_tester_policy_6” are shown in the table below.

Table 4. Example policy and distribution elements

Element	Code Segment

policy:
linux_fman_tester_policy_6
	`<policy name="linux_fman_tester_policy_6">`
	`  <dist_order>`
	`  <distributionref name="udpeth6"/>`
	`  <distributionref name="tcpeth6"/>`
	`  <distributionref name="ipv4eth6"/>`
	`  <distributionref name="garbage_dist_6"/>`
	`  </dist_order>`
	`</policy>`

• distribution "udpeth6"
• applied FIRST
	`<distribution name="udpeth6">`
	`  <queue count="128" base="0x3c00"/>`
	`  <key>`
	`  <fieldref name="ipv4.src"/>`
	`  <fieldref name="ipv4.dst"/>`
	`  <fieldref name="ipv4.nextp"/>`
	`  <fieldref name="udp.sport"/>`
	`  <fieldref name="udp.dport"/>`
	`  </key>`
	`</distribution>`

• distribution "tcpeth6"
• applied SECOND
	`<distribution name="tcpeth6">`
	`  <queue count="128" base="0x3c00"/>`
	`  <key>`
	`  <fieldref name="ipv4.src"/>`
	`  <fieldref name="ipv4.dst"/>`
	`  <fieldref name="ipv4.nextp"/>`
	`  <fieldref name="tcp.sport"/>`
	`  <fieldref name="tcp.dport"/>`
	`  </key>`
	`</distribution>`

• distribution "ipv4eth6"
• applied THIRD
	`<distribution name="ipv4eth6">`
	`  <queue count="128" base="0x3c00"/>`
	`  <key>`
	`  <fieldref name="ipv4.src"/>`
	`  <fieldref name="ipv4.dst"/>`

Table continues on the next page...

Table 4. Example policy and distribution elements (continued)

Element	Code Segment
	<fieldref name="ipv4.tos"/> </key> </distribution>
• distribution "garbage_dist_6" • applied FOURTH	<distribution name="garbage_dist_6"> <queue count="1" base="0x3c00"/> </distribution>

The policy element is named “linux_fman_tester_policy_6”. A policy element lists the distributions that Frame Manager will use for ports configured to use the policy. In this example, 4 distributions are listed: “udpeth6”, “tcpeth6”, “ipv4eth6” and “garbage_dist_6”. Frame Manager will try the distributions in the order listed in the policy element. Here, FMan1 will first try the distribution “udpeth6” for traffic received on it’s 10GEC port.

3.4.1 Distribution "udpeth6"

From the code segments in the table above, we see that distribution “udpeth6” has 5 keys:

Table 5. Distribution "udpeth6" 5 Keys

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

The presence of a IPv4 header and a UPD header are required in order for frame manager to extract these 5 keys from a frame. So if a Rx frame has an IPv4 header and a UDP header then Frame Manager will use the “udpeth6” distribution to compute the frame’s FQID.

In addition to the child element **key** of the “udpeth6” distribution, there is also a child element **queue**.

Table 6. Distribution "udpeth6" Queue

Element	Attribute / Value pair
"queue"	count = "128" base = "0x3c00"

The element **queue** attribute **count** specifies the number of FQIDs and attribute **base** is the base FQID. For “udpeth6”, the number of FQIDs is 128 and the base FQID is 0x3c00 .

The diagram below shows the FQID calculation for distribution "udpeth6".

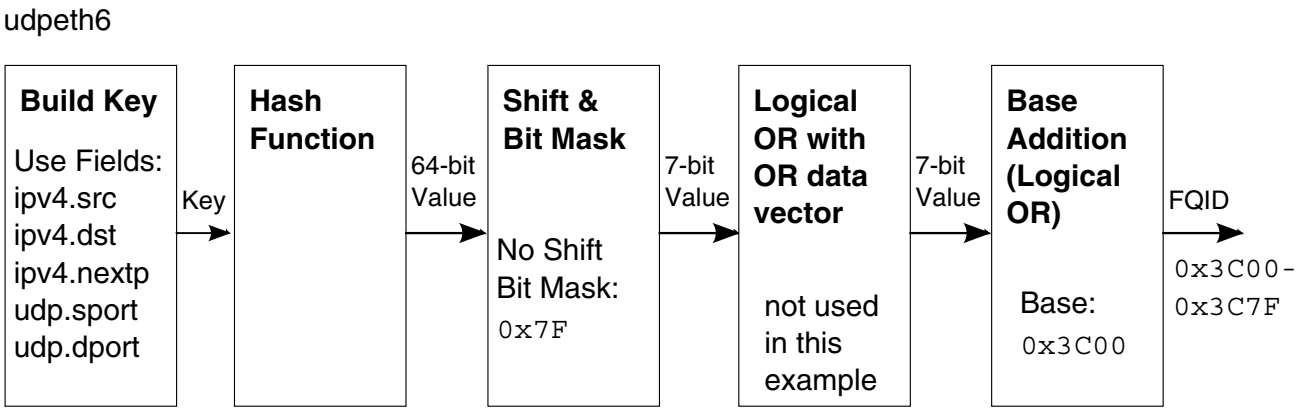

Figure 1. Frame Manager FQID Generation for Distribution "udpeth6"

Frame Manager KeyGen will concatenate the frame's "ipv4.src", "ipv4.dst", "ipv4.nextp", "udp.sport", and "udp.dport" fields and will use the concatenated fields as input to it's hash function. The result will be a 64-bit hash value. The 64-bit hash value will be ANDed with `0x7F` (yielding 128 possible FQIDs) and then ORed with `0x3c00`. All Rx frames that have IPv4 and UDP protocol headers will be enqueued to a FQID in the range `0x3c00 – 0x3c7f`. Rx frames belonging to the same flow (i.e., having the same "ipv4.src", "ipv4.dst", "ipv4.nextp", "udp.sport", and "udp.dport" fields) will be enqueued to the same FQID because their 64-bit hash value will be identical. Note that it is possible for multiple flows to be enqueued to the same frame queue.

3.4.2 Distribution "tcpeth6"

If an Rx frame does not have both an IPv4 header and a UDP header, then the next distribution listed in the policy "linux_frman_tester_policy_6" will be tried. In this example, the next distribution to try is the distribution "tcpeth6":

```
<distribution name="tcpeth6">
        <queue count="128" base="0x3c00"/>
        <key>
           <fieldref name="ipv4.src"/>
           <fieldref name="ipv4.dst"/>
           <fieldref name="ipv4.nextp"/>
           <fieldref name="tcp.sport"/>
           <fieldref name="tcp.dport"/>
        </key>
</distribution>
```

We see that distribution "tcpeth6" has 5 keys: "ipv4.src", "ipv4.dst", "ipv4.nextp", "tcp.sport", and "tcp.dport". The keys correspond to IPv4 and TCP protocol header fields as shown in the table below.

Table 7. Distribution "udpeth6" 5 Keys

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

The keys are IPv4 and TCP protocol header fields. If a Rx frame has an IPv4 header and a TCP header then Frame Manager will use distribution **tcpeth6** to compute the frame's FQID. For distribution "tcpeth6" the queue count is 128 and the base FQID is `0x3c00` (same base as used for distribution "udpeth6").

The diagram below shows the FQID calculation for distribution "tcpeth6".

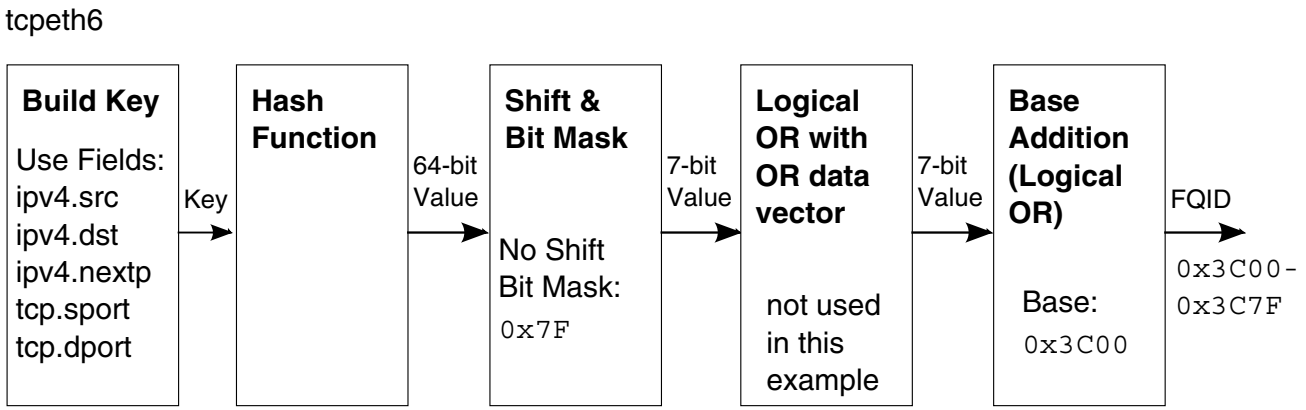

Figure 2. Frame Manager FQID Generation for Distribution "tcpeth6"

Frame Manager KeyGen will concatenate the frame's "ipv4.src", "ipv4.dst", "ipv4.nextp", "tcp.sport", and "tcp.dport" fields and generate a 64-bit hash value based on the concatenated fields. The 64-bit hash value will be ANDed with 0x7F. The resulting 7-bit value will be ORed with `0x3c00`. So all Rx Frames that have IPv4 and TCP protocol headers will be enqueued to a FQID in the range `0x3c00 – 0x3c7f`. Rx Frames belonging to the same flow (i.e., having the same "ipv4.src", "ipv4.dst", "ipv4.nextp", "tcp.sport", and "tcp.dport" fields) will be enqueued to the same frame queue.

3.4.3 Distribution "ipv4eth6"

If an Rx frame does not have both an IPv4 header and a TCP header, then the next distribution listed in the policy "linux_fman_tester_policy_6" will be tried. In this example, the next distribution to try is the distribution "ipv4eth6":

```
<distribution name="ipv4eth6">
        <queue count="128" base="0x3c00"/>
        <key>
           <fieldref name="ipv4.src"/>
           <fieldref name="ipv4.dst"/>
           <fieldref name="ipv4.tos"/>
        </key>
</distribution>
```

Distribution "ipv4eth6" has 3 keys: "ipv4.src", "ipv4.dst", and "ipv4.tos". The keys correspond to IPv4 protocol header fields as shown below.

Table 8. Distribution "udpeth6" 5 Keys

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

The keys are IPv4 protocol header fields. If a Rx frame has an IPv4 header then Frame Manager will use distribution "ipv4eth6" to compute the frame's FQID. For distribution "ipv4eth6" the queue count is 128 and the base FQID is `0x3c00` (same base as used for "udpeth6" and "tcpeth6").

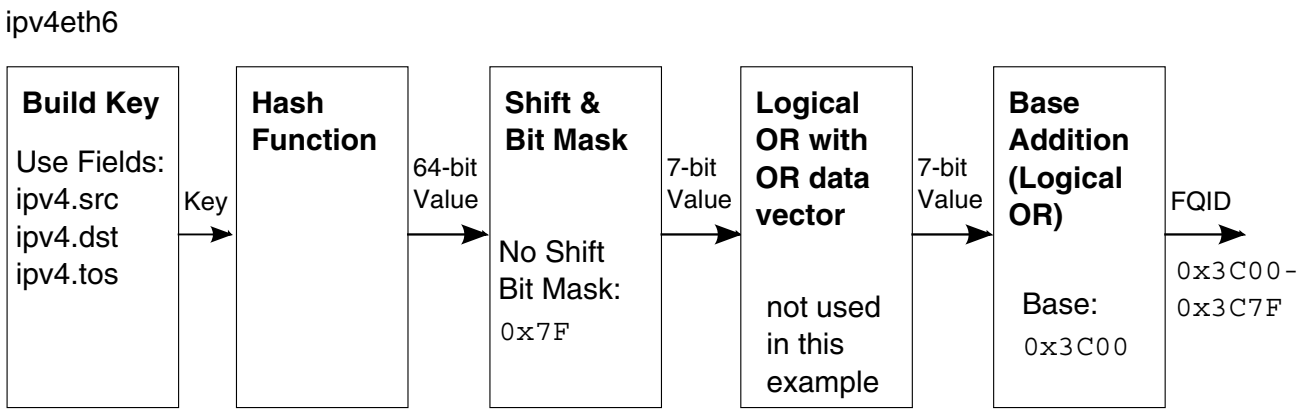

Figure 3. Frame Manager FQID Generation for Distribution "ipv4eth6"

Frame Manager KeyGen will concatenate the frame's "ipv4.src", "ipv4.dst", and "ipv4.tos" fields and generate a 64-bit hash value based on the concatenated fields. The 64-bit hash value will be ANDed with `0x7F`. The resulting 7-bit value will be ORed with `0x3c00`. So all Rx Frames that have IPv4 protocol headers will be enqueued to a FQID in the range `0x3c00 – 0x3c7f`. Rx Frames belonging to the same flow (i.e., with the same "ipv4.src", "ipv4.dst", and "ipv4.tos" fields) will be enqueued to the same frame queue.

3.4.4 Distribution "garbage_dist_6"

If an Rx frame does not have an IPv4 header, then the next distribution listed in the policy "linux_fman_tester_policy_6" will be tried. In this example, the next distribution to try is the distribution "garbage_dist_6":

```
<distribution name="garbage_dist_6">
        <queue count="1" base="0x3c00"/>
</distribution>
```

Distribution garbage_dist_6 does not have a child element **key** . In this case, there are no requirements for specific protocol headers to be present. For distribution "garbage_dist_6" the queue count is 1 (a single frame queue) and the base FQID is `0x3c00`. When frame manager uses distribution "garbage_dist_6" for a Rx frame, frame manager will enqueue the frame to FQID `0x3c00`.

NOTE

If distribution "garbage_dist_6" is listed first instead of last in the element **dist_order** , then all frames received on FMan1 10GEC interface will be enqueued to FQID `0x3c00`.

3.5 Modifying Configuration File `config_10g.xml`

Examples for two separate QorIQ SOCs are covered. The first example illustrates modifications for the P2041. The second example illustrates modifications for the P4080. Both examples show modifications to the same configuration file `config_10g.xml`.

3.5.1 QorIQ P2041

P2041 contains one Frame Manager instance with the following Ethernet MACs:

- 5 x dTSEC (1G)
- 1 x 10GEC (10G)

Assume that our RCW settings and software configuration are such that we can use the following interfaces:

- FMan1 dTSEC1 (SGMII)
- FMan1 dTSEC2 (SGMII)
- FMan1 dTSEC4 (RGMII)
- FMan1 dTSEC5 (RGMII)
- FMan1 10GEC (XAUI)

In this example, we configure FMan 1 dTSEC1, dTSEC2, and 10GEC Rx ports for PCD. The modified configuration file is shown below:

```
<cfgdata>
    <config>
        <engine name="fm0">
            <port type="1G" number="0" policy="linux_fman_tester_policy_0"/>
            <port type="1G" number="1" policy="linux_fman_tester_policy_1"/>
            <port type="10G" number="0" policy="linux_fman_tester_policy_6"/>
        </engine>
    </config>
</cfgdata>
```

Recall that for port type “1G”, numbering starts at 0, and so numbers 0 and 1 correspond in hardware to dTSEC1 and dTSEC2. For port type 10G, the number 0 corresponds to the single 10GEC. Here, dTSEC1, dTSEC2, and 10GEC will be configured to use policies “linux_fman_tester_policy_0”, “linux_fman_tester_policy_1”, and “linux_fman_tester_policy_6”, respectively. These policies are defined in the policy file `policy_hash_128fq.xml`. They are defined similarly to policy “linux_fman_tester_policy_6"

After examining the policy definitions in `policy_hash_128fq.xml` note that the difference between these policies amounts to the queue base value used by the distributions:

Table 9. `policy_hash_128fq.xml` policies

Policy	Port	Queue Base
`linux_fman_tester_policy_0`	FMan1 dTSEC1	`0x3800`
`linux_fman_tester_policy_1`	FMan1 dTSEC2	`0x3880`
`linux_fman_tester_policy_2`	FMan1 dTSEC3	`0x3900`
`linux_fman_tester_policy_3`	FMan1 dTSEC4	`0x3980`
`linux_fman_tester_policy_4`	FMan1 dTSEC5	`0x3a00`
`linux_fman_tester_policy_6`	FMan1 10GEC	`0x3c00`

For information regarding selection of the queue base value see **The Datapath Acceleration Architecture Linux Ethernet Driver Chapter 1.4.**

The reason for using distinct policies (i.e., a distinct queue base) for each port is to be able to determine the frame’s Rx port from the FQID. We could have used the same policy, e.g., policy “linux_fman_tester_policy_6” for all of the ports listed in the configuration file.

NOTE

The DPAA Ethernet driver only initializes 128 frame queues per enabled interface. If you are going to use any of the 16 policies defined in `policy_hash_128fq.xml` in your configuration file, make sure to select a policy corresponding to an enabled port. This guarantees that the FQIDs used by the policy have been initialized by the DPAA Ethernet driver.

3.5.2 QorIQ P4080

P4080 contains two Frame Manager instances each with the following Ethernet MACs:

- 4 x dTSEC (1G)
- 1 x 10GEC (10G)

Assume that the RCW settings and software configuration are such that we have enabled the following interfaces:

- FMan1 dTSEC2 (RGMII)
- FMan1 10GEC (XAUI)
- FMan2 dTSEC3 (SGMII)
- FMan2 dTSEC4 (SGMII)
- FMan2 10GEC (XAUI)

In this example we configure both XAUI and both SDMII interfaces for PCD. Modify the `config_10g.xml` file as shown in the code below:

```
<cfgdata>
    <config>
        <engine name="fm0">
            <port type="10G" number="0" policy="linux_fman_tester_policy_6"/>
        </engine>
        <engine name="fm1">
            <port type="1G" number="2" policy="linux_fman_tester_policy_10"/>
            <port type="1G" number="3" policy="linux_fman_tester_policy_11"/>
            <port type="10G" number="0" policy="linux_fman_tester_policy_14"/>
        </engine>
    </config>
</cfgdata>
```

We see that the element **config** has two child element **engines**: “fm0” corresponding to hardware block FMan1 and “fm1” corresponding to FMan2. So FMan1 10GEC Rx port is configured to use policy “linux_fman_tester_policy_6”. Now moving focus to engine “fm1”, recall that port numbering for port types starts at 0. Since we are configuring dTSEC3 and dTSEC4, the port numbers are “2” and “3”. Thus FMan2 dTSEC3, dTSEC4, and 10GEC Rx ports are configured to use policies “linux_fman_tester_policy_10”, “linux_fman_tester_policy_11”, and “linux_fman_tester_policy_14”, respectively. The policies were chosen according to the Policy/Port relationships listed below for `policy_hash_128fq.xml`.

Table 10. `policy_hash_128fq.xml` defines these four policies

Policy	Port	Queue Base
`linux_fman_tester_policy_6`	FMan1 10GEC	`0x3c00`
`linux_fman_tester_policy_10`	FMan2 dTSEC3	`0x7900`
`linux_fman_tester_policy_11`	FMan2 dTSEC4	`0x7980`
`linux_fman_tester_policy_14`	FMan2 10GEC	`0x7c00`

4 Configuring Frame Manager for a USDPAA Applications

This section provides a walk through of FMC configuration and policy files used for USDPAA applications. There's no requirement to be familiar with USDPAA in order to understand these examples.

4.1 Load Spreading

Similar to SMP Linux (section 3), distributions defined in USDPAA policy files are used to configure the Frame Manager to enqueue packets from the same flow to the same frame queue. However, the destination work queue for the PCD frame queues is a pool channel instead of a dedicated channel.

When a USDPAA application initializes a PCD frame queue, it sets the frame queue descriptor's destination work queue field to one of four pool channels. The cores running an application are all eligible to receive packets from the pool channels. In this way, Queue Manager is used to load-spread the received packets among the cores for efficient processing of the Rx packets. Packet ordering will not be maintained as packets from the same flow may be spread across multiple cores.

4.2 Example Usage on Target Board

Before running USDPAA applications ipfwd, ipsecfwd, reflector, and hello_reflector it is a requirement to configure frame manager for the interfaces that will receive traffic. The USDPAA applications process frames received on PCD frames queues and drop frames received on default frame queues. Recall that there is one default frame queue per receive port. If the receive ports are not configured for PCD, then Frame Manager will enqueue the Rx frames to default frame queues and subsequently the USDPAA application will drop the frames. Below are example FMC commands to run on the target board prior to starting a USDPAA application. The configuration and policy files used in these examples are provided in the USDPAA package.

For P2041 using Serdes Protocol 0x9 and P3041/P5020 using Serdes Protocol 0x36:

```
$ cd /usr/etc/
$ fmc -c usdpaa_config_p2_p3_p5_14g.xml -p usdpaa_policy_hash_ipv4.xml-a
```

P4080DS using Serdes Protocol 0xe:

```
$ cd /usr/etc/
$ fmc -c usdpaa_config_p4_serdes_0xe.xml -p usdpaa_policy_hash_ipv4.xml-a
```

NOTE

If you want to examine USDPAA FMC Input files on your host, these files are included in the SDK package "usdpaa". After extracting the source code for the package "usdpaa", the xml files reside in the directory:

```
QorIQ-SDK-V1.2-20120614-yocto/build_[target board]_release/tmp/
work/ppce500mc-fsl-linx/usdpaa-git-r7/git/apps/ppac/
```

4.3 Walk-through Configuration File `usdpaa_config_p4_serdes_0xe.xml`

The contents of configuration file `usdpaa_config_serdes_0xe.xml` are displayed below. The name of the configuration file contains "serdes_0xe" which is a reference to serdes protocol 0xe. Setting RCW[SRDS_PRTCL] equal to 0xe enables the following interfaces: FMan1 10GEC, FMan2 dTSEC3 SDMII, FMan 2 dTSEC4 SDMII, and FMAN 2 10GEC. These are the interfaces configured in `usdpaa_config_serdes_0xe.xml`.

```
<cfgdata>
    <config>
<engine name="fm0">
   <port type="10G" number="0" policy="hash_ipsec_src_dst_spi_policy5"/>
</engine>
<engine name="fm1">
   <port type="1G" number="2" policy="hash_ipsec_src_dst_spi_policy8"/>
   <port type="1G" number="3" policy="hash_ipsec_src_dst_spi_policy9"/>
   <port type="10G" number="0" policy="hash_ipsec_src_dst_spi_policy11"/>
```

```
</engine>
    </config>
</cfgdata>
```

In this example, FMan 1 10GEC, FMan2 dTSEC3, FMan 2 dTSEC4, and FMan2 10GEC are configured to use policies “hash_ipsec_src_dst_spi_policy5”, “hash_ipsec_src_dst_spi_policy8”, “hash_ipsec_src_dst_spi_policy9”, and “hash_ipsec_src_dst_spi_policy11”, respectively. (For a more detailed description of how configuration files are used to configure specific ports, see examples in the previous topic). The policies used in `usdpaa_config_serdes_0xe.xml` are defined in the policy file `usdpaa_policy_hash_ipv4.xml`.

We will examine the policy file `usdpaa_policy_hash_ipv4.xml` below. For now let’s note that this policy file defines 12 policies that are intended to be used to configure specific ports. The 12 policies along with the corresponding ports and queue base are listed in the table below.

Table 11. Policies defined in Policy file `8c-128fq-p.xml`

Policy	Port	Queue Base
hash_ipsec_src_dst_spi_policy0	FMan 1 dTSEC 1	0x400
hash_ipsec_src_dst_spi_policy1	FMan 1 dTSEC 2	0x500
hash_ipsec_src_dst_spi_policy2	FMan 1 dTSEC 3	0x600
hash_ipsec_src_dst_spi_policy3	FMan 1 dTSEC 4	0x700
hash_ipsec_src_dst_spi_policy4	FMan 1 dTSEC 5	0x800
hash_ipsec_src_dst_spi_policy5	FMan 1 10GEC	0x900
hash_ipsec_src_dst_spi_policy6	FMan 2 dTSEC 1	0xa00
hash_ipsec_src_dst_spi_policy7	FMan 2 dTSEC 2	0xb00
hash_ipsec_src_dst_spi_policy8	FMan 2 dTSEC 3	0xc00
hash_ipsec_src_dst_spi_policy9	FMan 2 dTSEC 4	0xd00
hash_ipsec_src_dst_spi_policy10	FMan 2 dTSEC 5	0xe00
hash_ipsec_src_dst_spi_policy11	FMan 2 10GEC	0xf00

4.4 Walk-through Policy File `usdpaa_policy_hash_ipv4.xml`

Instead of displaying the entire contents of policy file `usdpaa_policy_hash_ipv4.xml`, only the policy and distribution elements applicable to policy ”hash_ipsec_src_dst_spi_policy5” are shown in the table below. The other 11 policies are defined similarly. The difference is the queue base used by the distributions.

Table 12. Example Policy and Distribution Elements

Name	Code Example

policy: hash_ipsec_src_dst_spi_policy5	`<policy name="hash_ipsec_src_dst_spi_policy5">`
	`  <dist_order>`
	`  <distributionref name="hash_ipsec_src_dst_spi_dist5"/>`
	`  <distributionref name="hash_ipv4_src_dst_dist5"/>`
	`  <distributionref name="default_dist5"/>`
	`  </dist_order>`
	`</policy>`

• distribution "hash_ipsec_src_dst_spi_dist5"
• apply FIRST
	`<distribution name="hash_ipsec_src_dst_spi_dist5">`
	`  <queue count="32" base="0x900"/>`
	`  <key>`
	`  <fieldref name="ipv4.src"/>`

Table continues on the next page...

Table 12. Example Policy and Distribution Elements (continued)

Name	Code Example
	<fieldref name="ipv4.dst"/> <fieldref name="ipsec_esp.spi"/> </key> </distribution>
• distribution "hash_ipv4_src_dst_dist5" • apply SECOND	<distribution name="hash_ipv4_src_dst_dist5"> <queue count="32" base="0x900"/> <key> <fieldref name="ipv4.src"/> <fieldref name="ipv4.dst"/> </key> </distribution>
• distribution "default_dist5" • apply THIRD	<distribution name="default_dist5"> <queue count="1" base="0x5b"/> </distribution>

A policy element lists the distributions that Frame Manager will use for interfaces configured to use the policy. Frame Manger will try the distributions in the order listed in the policy element. According to policy “hash_ipsec_src_dst_spi_policy5", FMan 1 will first try to use distribution “hash_ipsec_src_dst_spi_dist5” for traffic received on it’s 10GEC port.

4.5 Distribution “hash_ipsec_src_dst_spi_dist5”

We see that distribution “hash_ipsec_src_dst_spi_dist5” has 3 keys. The keys and their corresponding protocol header and fields are shown in the table:

Table 13. IPv4 and ESP protocol header fields

Keys	Protocol Header	Field
ipv4.src	IPv4	Source IP Address
ipv4.dst	IPv4	Destination IP Address
ipsec_esp.spi	ESP	Security Parameters Index

The presence of a IPv4 header and a ESP header are required in order for Frame Manager to extract these 3 keys from a frame. So if a Rx frame has an IPv4 header and an ESP header then Frame Manager will use the distribution “hash_ipsec_src_dst_spi_dist5” to compute the frame’s FQID.

In addition to the child element **key** of the “hash_ipsec_src_dst_spi_dist5” distribution, there is also a child element **queue**.

Table 14. Distribution "hash_ipsec_src_dst_spi_dist5" Queue

Element	Attribute / Value pair
"queue"	count = "32" base = "0x900"

The queue attribute **base** is the base FQID and the queue attribute **count** specifies the number of FQIDs. For distribution “hash_ipsec_src_dst_spi_dist5”, the base FQID is 0x900 and number of FQIDs is 32.

The diagram below shows the FQID calculation for distribution "hash_ipsec_src_dst_spi_dist5"

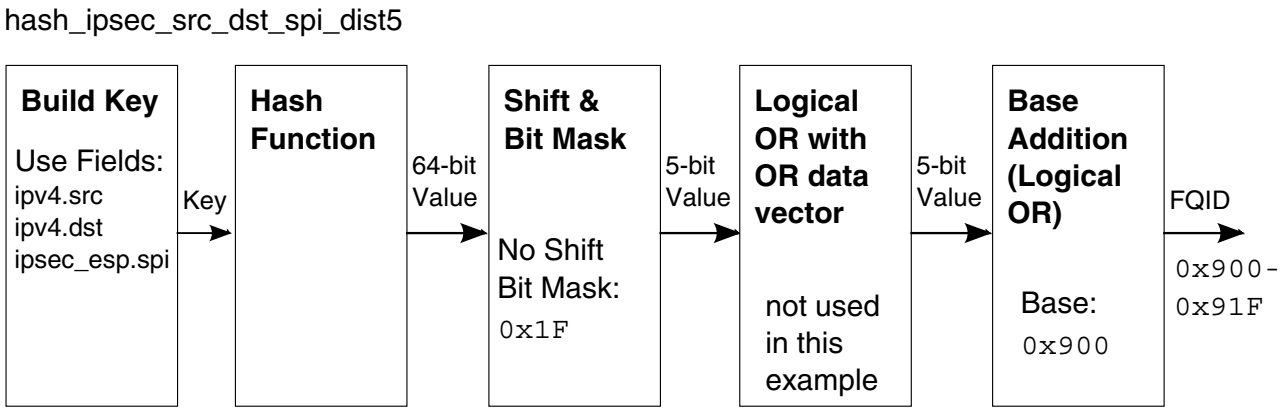

Figure 4. FQID calculation for hash_ipsec_src_dst_spi_dist5

Frame Manager KeyGen will concatenate the frame's "ipv4.src", "ipv4.dst", and "ipsec_esp.spi" fields and will use the concatenated fields as input to its hash function. The result will be a 64-bit hash value. In this case, the 64-bit hash value will be ANDed with `0x1F` (yielding 32 possible FQIDs) and then ORed with `0x900`. All Rx frames that have IPv4 and ESP protocol headers will be enqueued to a FQID in the range `0x900 – 0x91f` . Rx frames belonging to the same flow (i.e., having the same "ipv4.src", "ipv4.dst", "ipsec_esp.spi" fields) will be enqueued to the same FQID because their 64-bit hash value will be identical. Note that it is possible for multiple flows to be enqueued to the same frame queue.

4.6 Distribution `hash_ipsec_src_dst_spi_dist5`

If an Rx frame does not have both an IPv4 header and a ESP header, then the next distribution listed in the policy "hash_ipsec_src_dst_spi_policy5" will be tried. In this example, the next distribution to try is distribution "hash_ipv4_src_dst_dist5":

```
<distribution name="hash_ipv4_src_dst_dist5">
    <queue count="32" base="0x900"/>
    <key>
        <fieldref name="ipv4.src"/>
        <fieldref name="ipv4.dst"/>
    </key>
</distribution>
```

Distribution "hash_ipv4_src_dst_dist5" has 2 keys that correspond to IPv4 protocol header fields::

Table 15. IPv4 and ESP protocol header fields

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

The keys are IPv4 protocol header fields. If a Rx frame has an IPv4 header then Frame Manager will use the distribution "hash_ipv4_src_dst_dist5" to compute the frame's FQID.

For distribution "hash_ipv4_src_dst_dist5" the queue **count** is 32 and the **base** FQID is `0x900` (same base as used for distribution "hash_ipsec_src_dst_spi_dist5").

The diagram below shows the FQID calculation for distribution "hash_ipv4_src_dst_dist5"

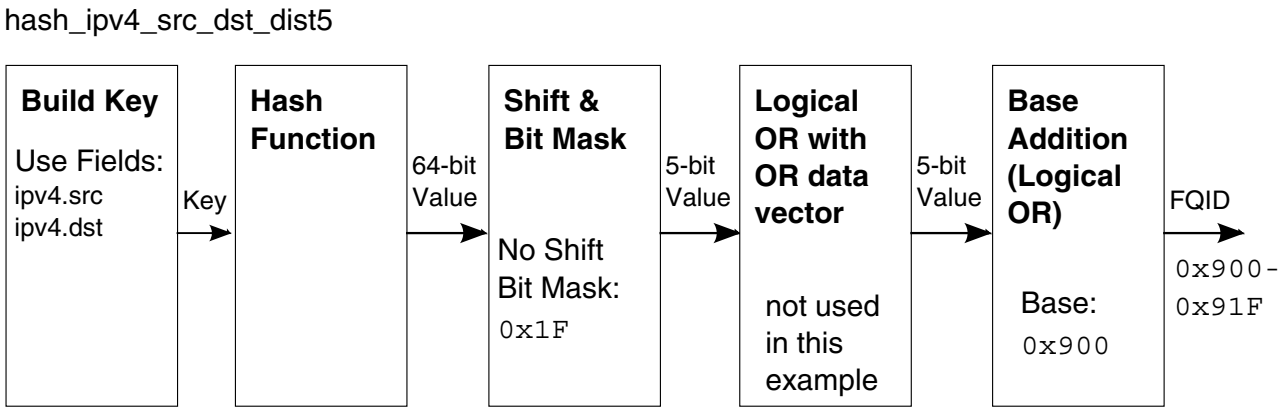

Figure 5. FQID calculation for hash_ipv4_src_dst_dist5

Frame Manager KeyGen will concatenate the frame's "ipv4.src" and "ipv4.dst" fields and generate a 64-bit hash value based on the concatenated fields. The 64-bit hash value will be ANDed with `0x1F`. The resulting 5-bit value will be ORed with `0x900` . So all Rx Frames that have IPv4 protocol headers will be enqueued to a FQID in the range `0x900 – 0x91f`. Rx Frames belonging to the same flow (i.e., with the same "ipv4.src" and "ipv4.dst" fields) will be enqueued to the same frame queue.

4.7 Distribution "default_dist5"

If an Rx frame does not have an IPv4 header, then the next distribution listed in policy "hash_ipsec_src_dst_spi_policy5" will be tried. In this example, the next distribution to try is distribution "default_dist5":

```
<distribution name="default_dist5">
    <queue count="1" base="0x5b"/>
</distribution>
```

Distribution "default_dist5" does not have a child element **key**. In this case, there are no requirements for specific protocol headers to be present. For distribution "default_dist5" the queue count is 1 (a single frame queue) and the base FQID is `0x5B` . When frame manager uses distribution "default_dist5" for a Rx frame, frame manager will enqueue the frame to FQID `0x5B` . As noted earlier, USDPAA applications ipfwd, ipsecfwd, reflector, and hello_reflector drop packets received on the default frame queue.

For frames received on PCD frame queues `0x900-0x91F`, USDPAA applications ipfwd, ipsecfwd, reflector, and hello_reflector will process the frames according to the application's purpose. For example, applications reflector and hello_reflector will "reflect" or transmit the frame on the port in which the frame was received.

4.8 Offline port example using P4080 to illustrate

So far we have considered PCD configuration for receive paths only, i.e., for packets received on Frame Manager 1G and 10G ports. Offline Ports can also perform PCD function. An SoCs CPUs or Security Engine may enqueue frames to Queue Manager that are destined to Offline Ports for PCD processing.

Now let's take a look at a configuration file that is used to configure a P4080 offline port for PCD. The name of the file is `usdpaa_config_p4_serdes_0xe_offline_host.xml`. The contents of the element **cfgdata** are shown here.

```
<cfgdata>
    <config>
        <engine name="fm0">
           <port type="10G" number="0" policy="hash_default_policy5"/>
           <port type="OFFLINE" number="1" policy="hash_ipsec_src_dst_spi_policy2"/>
```

```
        </engine>
        <engine name="fm1">
           <port type="1G" number="2" policy="hash_default_policy8"/>
           <port type="1G" number="3" policy="hash_default_policy9"/>
           <port type="10G" number="0" policy="hash_default_policy11"/>
        </engine>
    </config>
</cfgdata>
```

Engine “fm0” has two child element ports. The first element **port** has attribute **type** “10G”. The 10GEC port is configured to use policy “hash_default_policy5”. The second element **port** has attribute **type** “OFFLINE”, attribute **number** 1, and attribute **policy** “hash_ipsec_src_dst_spi_policy2”. Port numbering starts with number 0. SoC P4080 has hardware Offline/ Host Command Ports 1,2,…,7. So in this example, FMan1 Offline Port 2 is configured to use policy “hash_ipsec_src_dst_spi_policy2”. This policy is defined in the file `usdpaa_policy_hash_ipv4.xml`.

The learning from the material above covered the structure and purpose of the config and policy files with examples. From these examples we expect that you will be able to identify the pertinent policy and distribution elements and apply them to your scenario.

5 Coarse Classification Examples

This section provides a walk through of FMC configuration and policy files used for USDPAA applications. There's no requirement to be familiar with USDPAA in order to understand these examples.

So far we’ve seen examples of policy files that contain policy and distribution elements. These are required elements of a policy file. A classification element is optional. A classification element is used for cases where protocol fields are to be compared to fixed values, and based on the result of the comparison, Frame Manager will compute a FQID for the frame. This functionality is referred to as “coarse classification” or “exact match.” The Frame Manager’s Controller block provides this functionality.

5.1 Coarse Classification Examples

The table below shows policy, distribution, and classification elements of an example FMC policy file.

Table 16. Example Policy File Containing Classification Element

Element	Code Example

Policy	`<policy name="example_class_policy">`
	`  <dist_order>`
	`  <distributionref name="udp_dist"/>`
	`  <distributionref name="non_udp_dist"/>`
	`  </dist_order>`
	`</policy>`

Distribution	`<distribution name="udp_dist">`
	`  <queue count="1" base="0x400"/>`
	`  <protocols>`
	`  <protocolref name="udp"/>`
	`  <protocolref name=”ipv4”/>`
	`  </protocols>`
	`  <action type=”classification” name=”udp_classif”/>`
	`</distribution>`

Classification	`<classification name="udp_classif">`
	`  <key>`
	`  <fieldref name="udp.dport"/>`
	`  </key>`
	`  <entry>`

Table continues on the next page...

Table 16. Example Policy File Containing Classification Element (continued)

Element	Code Example
	<data>0x7918</data> <queue base=”0x600”/> </entry> <entry> <data>0x791A</data> <queue base=”0x602”/> </entry> <entry> <data>0x791C</data> <queue base=”0x604”/> </entry> </classification>
Distribution	<distribution name="non_udp_dist"> <queue count="1" base="0x800"/> </distribution>

From the table above we see that the policy “example_class_policy” lists two distributions that Frame Manager will use for ports configured to use the policy. Frame Manager will try the distributions in the order listed in the policy element. First we examine distribution “udp_dist.” Notice that there is no element **key**, however, there is an element **protocols**. Similarly to the element **key** that has child elements **fieldref** (e.g. “udp.sport”, “udp.dport”) the element **protocols** has child elements **protocolref**. In this case, the element **protocolref** attribute **names** are “udp” and “ipv4”. So if a Rx frame has a UDP and an IPv4 protocol header, then Frame Manager will use this distribution to determine the frame’s FQID. For distribution “udp_dist,” the base FQID is `0x400` and the number of FQIDs is 1. In absence of any further action, Frame Manager would enqueue frames with a UDP and IPv4 header to FQID `0x400`. Since distribution “udp_dist” has an element **action**, processing of frames that have both an UPD and IPv4 continues. Here, element **action** attribute **type** is “classification” and attribute **name** is “udp_classif”. So the next action for Frame Manager is to perform classificaiton (i.e., exact match) on these frames using classification “udp_classif”. If a frame does not have both a UDP and an IPv4 header, then Frame Manager will try the next distribution listed in the policy “example_class_policy”.

Assume that our Rx frame has a UDP and an IPv4 header, then the next action by Frame Manager will be to perform classification as specified by “udp_classif”. From the table above, classification “udp_classif” has child elements **key** and **entry**. Taking a look at the element **key**, we see that the element **fieldref** attribute **name** is “udp.dport”. This means that exact match will be performed using the frame’s UDP destination port field. Each element **entry** contains child elements **data** and **queue**. Let’s see how the elements **data** and **queue** are used. In this example, a frame’s UDP destination field is compared to data value `0x7918` and if equal, Frame Manager will enqueue to the frame to FQID `0x600`. If UDP destination port is equal to data value `0x71A` then frame manager will enqueue the frame to FQID `0x602`.If UDP destination port is equal to data value `0x791C` then frame manager will enqueue the frame to FQID `0x604`. If UDP destination port is not equal to any of these three data values, then frame manager will enqueue the frame to FQID `0x400` as determined from distribution “udp_dist” that proceeded classification “udp_classif.”

5.2 Adding ‘action’ child element to classification element

For this section, we modified the example used above. The modifications are as follows:

- Added child element action to classification “udp_classif”
- Added distribution “hash_dist”

Table 17. Example Classification Element with Action Child Element

Element	Code Example	
Policy	<policy name="classification_policy"> <dist_order> <distributionref name="udp_dist"/> <distributionref name="non_udp_dist"/> </dist_order> </policy>	
Distribution	<distribution name="udp_dist"> <queue count="1" base="0x400"/> <protocols> <protocolref name="udp"/> <protocolref name="ipv4"/> </protocols> <action type=”classification” name=”udp_classif”/> </distribution>	
Classification	<classification name="udp_classif"> <key> <fieldref name="udp.dport"/> </key> <entry> <data>0x7918</data> <queue base=”0x600”/> </entry> <entry> <data>0x791A</data> <queue base=”0x602”/> </entry> <entry> <data>0x791C</data> <queue base=”0x604”/> </entry> <action type=”distribution” condition=”on-miss” name=”hash_dist”/> </classification>	. . . . < change <
Distribution	<distribution name="hash_dist"> <queue count="32" base="0x900"/> <key> <fieldref name="ipv4.src"/> <fieldref name="ipv4.dst"/> </key> </distribution>	
Distribution	<distribution name="non_udp_dist"> <queue count="1" base="0x800"/> </distribution>	

The addition of the element **action** to “udp_classif” changes Frame Managers next action if the frame’s UDP destination port does match any of three data values `0x7918, 0x791A`, or `0x791C`. The element **action** attribute **condition** is “on-miss.” Inside a classification element the attribute **condition** “on-miss” is satisfied if the classification key does not match any of the data values. In this case, if the UDP destination port does not equal 0x7918, 0x791A, or 0x791C then the “on-miss” condition is satisfied. The **action** attribute **type** is “distribution” and the action attribute **name** is “hash_dist”. So the action for Frame Manager in the event the “on-miss” condition is satisfied is to use the distribution “udp_hash_dist” for the received frame.

Suppose that a frame’s UDP destination port is not equal to `0x7918, 0x791A`, or `0x791C`. Then Frame Manager will use distribution “hash_dist” to determine the frame’s FQID. Note that the frame has an IPv4 header, as this was one of the protocolrefs for distribution “udp_dist.” Frame Manager KeyGen will concatenate fields “ipv4.src” and “ipv4.dst” and then generate a 64-bit hash value. Since the **queue count** is “32” and the **base** FQID is `0x900`the FQID will be in the range `0x900-0x91F`.

NOTE

The use of an element **mask** that allows coarse classification for a range of values is supported.

5.3 Nested Classification Example

Frame Manager Controller supports tables stored in frame manager internal memory to perform the classification or exact match comparisons. Frame Manager Controller supports nested look-ups. In the code example below, an example of a nested look-up is given. Ethernet source and destination fields are compared to fixed values. Based on the result of the the first table-lookup, if there is an exact match, a second table lookup will be performed.

Table 18. Nested Look-up

Element	Code Example

Policy	`<distribution name="eth_distribution ">`
	`  <queue count="1" base="0x600"/>`
	`  <protocols>`
	`  <protocolref name="ethernet"/>`
	`  </protocols>`
	`  <action type=”classification” name=”classification_1”/>`
	`</distribution>`

Classification	`<classification name=" classification_1">`
	`  <key>`
	`  <fieldref name="ethernet.dst"/>`
	`  </key>`
	`  <entry>`
	`  <data>0x010101010101</data>`
	`  <queue base=”0x601”/>`
	`  <action type=”classification” name=”classification_2”/>`
	`  </entry>`
	`</classification>`

Classification	`<classification name=" classification_2">`
	`  <key>`
	`  <fieldref name="ethernet.src"/>`
	`  </key>`
	`  <entry>`
	`  <data>0x020202020202</data>`
	`  <queue base=”0x602”/>`
	`  </entry>`
	`</classification>`

The distribution “eth_distribuion” has an element **protocolref** with attribute **name** “ethernet.” In addition, it has an element **action** with attribute **type** “classification” and attribute **name** “classification_1.” So for frames with an Ethernet header, Frame Manager will perform classification as specified by classification element “classification_1.”

The classification “classification_1” specifies to Frame Manger to compare the frame’s Ethernet destination field to data value `0x010101010101`. Note that the size of the data value equals the size of the Ethernet destination field. In this example, the element entry has a child element **action.** The element **action** attribute **type** is “classification” and the attribute **name** is “classification_2.” This means that if there is an exact match, i.e., Ethernet destination field is equal to `0x010101010101`, then Frame Manager will perform a second classification on the frame using “classification_2”.

The flow chart below shows how the FQID is determined for frames with Ethernet headers:

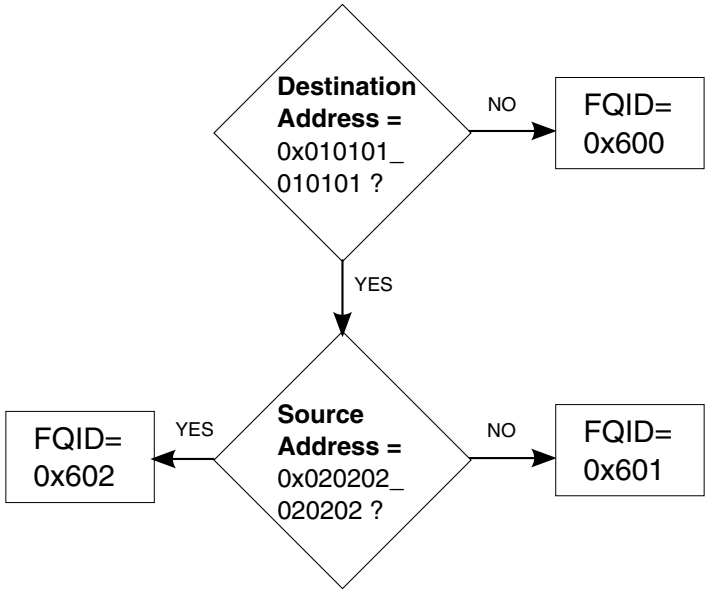

Figure 6. FQID Computation for Nested Classification Example

6 Policer Elements in the FMC Policy Files

Frame Manager policer supports implementation of differentiated services at line speed on Rx or offline parsing paths.

The Frame Manager policer holds 256 traffic profiles in internal memory, each profile implementing RFC-2698, RFC-4115, or pass-through mode. Each mode can work in either color-blind or color-aware mode and can pass or drop packets according to their resulting color.

6.1 Understanding Policy File with Policer Elements

The USDPAA package contains a FMC policy file with two policer elements. The name of this policy file is `usdpaa_policer.xml`. Similarly to the example policy files we have seen so far, `usdpaa_policer.xml` defines 12 policies that are intended to be used to configure specific Rx ports. Let's take a look at the definition of one of these policies: "hash_ipsec_src_dst_spi_policy0". The elements used in defining this policy are shown in the code shown below.

Table 19. Elements from `usdpaa_policer.xml`

Element	Code Example

Policy	`<policy name="hash_ipsec_src_dst_spi_policy0">`
	`  <dist_order>`
	`  <distributionref name="hash_ipv4_src_dst_dist0"/>`
	`  <distributionref name="default_dist0"/>`
	`  </dist_order>`
	`</policy>`

Distribution	`<distribution name="hash_ipv4_src_dst_dist0">`
	`  <queue count="32" base="0x400"/>`
	`  <key>`
	`  <fieldref name="ipv4.src"/>`
	`  <fieldref name="ipv4.dst"/>`
	`  </key>`
	`  <action name="policer_1g" type="policer"/>`
	`</distribution>`

Table continues on the next page...

Table 19. Elements from `usdpaa_policer.xml` (continued)

Element	Code Example

Distribution	`<distribution name="default_dist0">`
	`  <queue count="1" base="0x51"/>`
	`</distribution>`

Policer	`<policer name="policer_1g">`
	`  <algorithm>rfc2698</algorithm>`
	`  <color_mode>color_blind</color_mode>`
	`  <CIR>1000000</CIR>`
	`  <EIR>1400000</EIR>`
	`  <CBS>1000000</CBS>`
	`  <EBS>1400000</EBS>`
	`  <unit>packet</unit>`
	`</policer>`

From the code example above, we see that the policy “hash_ipsec_src_dst_spi_policy0” lists two distributions that Frame Manager will use for ports configured to use the policy. The first distribution listed is “hash_ipv4_src_dst_dist0” and the second distribution listed is “default_dist0”.

Examining distribution “hash_ipv4_src_dst_dist0” we see that the keys are “ipv4.src” and “ipv4.dst”. The element **queue** tells us that the number of frame queues is 32 and that the base FQID is `0x400`. So up to this point, frames with IPv4 headers will be assigned to FQIDs `0x400-0x41F`. Since this distribution has an element **action,** processing of the frame will continue after computation of the FQID. The element **action** attribute **type** is “policer” and the attribute **name** is “policer_1g”. After determination of the FQID, the frame will then be sent to the Frame Manager Policer.

The element **policer** defines a policer profile. The element **policer**, shown in the code above, defines a policer profile named “policer_1g”. The algorithm value is used to set the operation mode. Here we’re configuring the profile “policer_1g” to operate in “rfc2698” mode. So the policing will be based on two-rate, three-color marking algorithm RFC2698. Additionally, the color_mode value is “color_blind”. In color-blind mode the algorithm assumes that the packet stream is uncolored. The values CIR , EIR, CBS and EBS are used to set the algorithm’s committed information rate, peak information rate, committed burst size and peak burst size. The unit value can either be byte or packet, in this case the unit is “packet”.

There is no child element **action** in “policer_1g” and so after the Policer marks a frame either RED, YELLOW, or GREEN, Frame Manager will enqueue the frame to the FQID that was determined by distribution “hash_ipv4_src_dst_dist0”.

In this example, if the received frame does not have an IPv4 header, then Frame Manager will compute a FQID for the frame using the distribution “default_dist0” which will result in the frame being enqueued to FQID `0x51`.

6.2 Modifying the Policer Element

Let’s modify the policer element as shown below.

Table 20. Modified Policer Element

Element	Code Example

Policer	`<policer name="policer_1g">`
	`  <algorithm>rfc2698</algorithm>`
	`  <color_mode>color_blind</color_mode>`
	`  <CIR>1000000</CIR>`
	`  <EIR>1400000</EIR>`
	`  <CBS>1000000</CBS>`
	`  <EBS>1400000</EBS>`
	`  <unit>packet</unit>`
	`  <action condition=”on-red” type=”drop”/>`
	`</policer>`

In this example, “policer_1g” contains a child element **action**. The attribute **condition** is “on-red” and the attribute **type** is “drop”. For this policer profile, packets that are marked RED will be dropped by the frame manager (more precisely, they will be discarded by the Frame Manager Buffer Manager Interface). Packets that are marked GREEN and YELLOW will be enqueued to the FQID as determined by the distribution that preceded “policer_1g”.

6.3 Example Using Color-blind Pass-through Mode

The code segment below shows an example of a policer element that specifies the use of color-blind pass-through mode for the policer profile. The default_color value is “red” and so frames that are processed using this policer profile will be marked RED. Because there is an element **action** that specifies to drop frames that are marked red, all packets that are processed by this policer profile will be dropped.

Table 21. Color-blind pass-through mode profile

Element	Code Example

Policer	`<policer name=“drop_traffic">`
	`  <algorithm>pass_through</algorithm>`
	`  <color_mode>color_blind</color_mode>`
	`  <default_color>red</default_color>`
	`  <action condition="on-red" type=“drop"/>`
	`</policer>`

An example usage of policer profile “drop_traffic”:

Table 22. Usage of color-blind pass-through mode profile

Element	Code example

Distribution	`<distribution name="distribution_drop">`
	`  <queue count="1" base="0x51"/>`
	`  <action type=”policer” name = “drop_traffic”/>`
	`</distribution>`

All frames that are processed by “distribution_drop” will be dropped.

7 Frame Manager Soft Parser

In some cases, a user may want to configure Frame Manager to perform hashing or exact match classification using fields from custom protocols or shim headers.

This is supported by the soft parser. The soft parser enables the user to configure Frame Manager to perform hashing or exact match classification using fields from proprietary protocols or shim headers.

7.1 Examining a Custom Protocol File

Recall that the hard parser supports known and stable protocols (see Appendix B for a list of standard protocols). In order to configure the soft parser, a fourth file is used an input to FMC. This file is referred to as the custom protocol file or the soft parser file. In this file, a custom protocol or shim header is defined and the actions that should be taken by the soft parser are specified. There is an example soft parser file in the *Freescale Linux SDK for QorIQ Processors*. The contents of the example soft parser file `usdpaa_custom_coarse_classify_netpdl.xml` are displayed in the NetPDL code below. In this example, an ARP header is our custom protocol, as this header type is not recognized by the hard parser.

```
<netpdl name="IPv4 ARP" description="Recognize and mark IPv4 ARP frames">
  <protocol name="arp" longname="ARP Protocol" prevproto="ethernet">
   <format>
    <fields>
      <field type="fixed" name="htype" longname="Hardware type" size="2"/>
      <field type="fixed" name="ptype" longname="Protocol type" size="2"/>
      <field type="fixed" name="hlen" longname="Hardware address length" size="1"/>
      <field type="fixed" name="plen" longname="Protocol address length"  size="1"/>
      <field type="fixed" name="opcode" longname="Operation" size="2"/>
      <field type="fixed" name="sha" longname="Sender hardware address" size="6"/>
      <field type="fixed" name="spa" longname="Sender protocol address" size="4"/>
      <field type="fixed" name="tha" longname="Target hardware address" size="6"/>
      <field type="fixed" name="tpa" longname="Target protocol address" size="4"/>
     </fields>
    </format>
    <execute-code>
      <before>
        <if expr="ethernet.type != 0x806">
          <if-true>
            <!-- Confirms Ethernet layer and exits-->
            <action type="exit" confirm="yes" nextproto="return"/>
          </if-true>
        </if>
      </before>
      <after>
        <assign-variable name="$shimoffset_1" value="$nxtHdrOffset"/>
        <assign-variable name="$nxtHdrOffset" value="$nxtHdrOffset + $headerSize"/>
        <action type="exit" confirmcustom="shim1" nextproto="end_parse"/>
      </after>
    </execute-code>
  </protocol>
</netpdl>
```

Here we walk through the custom protocol shown above. The element **protocol** contains the elements required to define a custom protocol. The element **protocol** has attributes **name**, **longname**, and **prevproto**:

```
<protocol name="arp" longname="ARP Protocol" prevproto="ethernet">
```

The attribute **name** and **prevproto** are required and the attribute **longname** is optional. The attribute **name** defines the unique name of the custom protocol or shim header. This is the name that will be used in a policy file to refer to the custom protocol. The optional attribute **longname** is a user-friendly name for the protocol. The attribute **prevproto** lists the protocol that immediately precedes the custom protocol. In this example, the previous protocol is “ethernet”. The previous protocol must be a standard protocol that is supported by the Hard Parser (see **Appendix B**).

NOTE

It is allowed to define only one custom protocol or shim header after a standard protocol. In other words, it is not allowed to define two custom protocols with same **prevproto** value. Multiple custom protocols can be defined, but they must not follow the same standard protocol.

The element **format** is the first child element of the element **protocol**. The format fields are used to define the custom protocol. In this example, the format fields are used to define the ARP Header:

```
<format>
  <fields>
   <field  type="fixed"  name="htype"  longname="Hardware type"           size="2"/>
   <field  type="fixed"  name="ptype"  longname="Protocol type"           size="2"/>
   <field  type="fixed"  name="hlen"   longname="Hardware address length" size="1"/>
   <field  type="fixed"  name="plen"   longname="Protocol address length" size="1"/>
   <field  type="fixed"  name="opcode" longname="Operation"               size="2"/>
   <field  type="fixed"  name="sha"    longname="Sender hardware address" size="6"/>
   <field  type="fixed"  name="spa"    longname="Sender protocol address" size="4"/>
   <field  type="fixed"  name="tha"    longname="Target hardware address" size="6"/>
   <field  type="fixed"  name="tpa"    longname="Target protocol address" size="4"/>
  </fields>
</format>
```

The element **field** has attributes **type, name, longname** and **size**. The type shown here is “fixed” indicating that the field is byte-length. Each element **field** represents an ARP Header field:

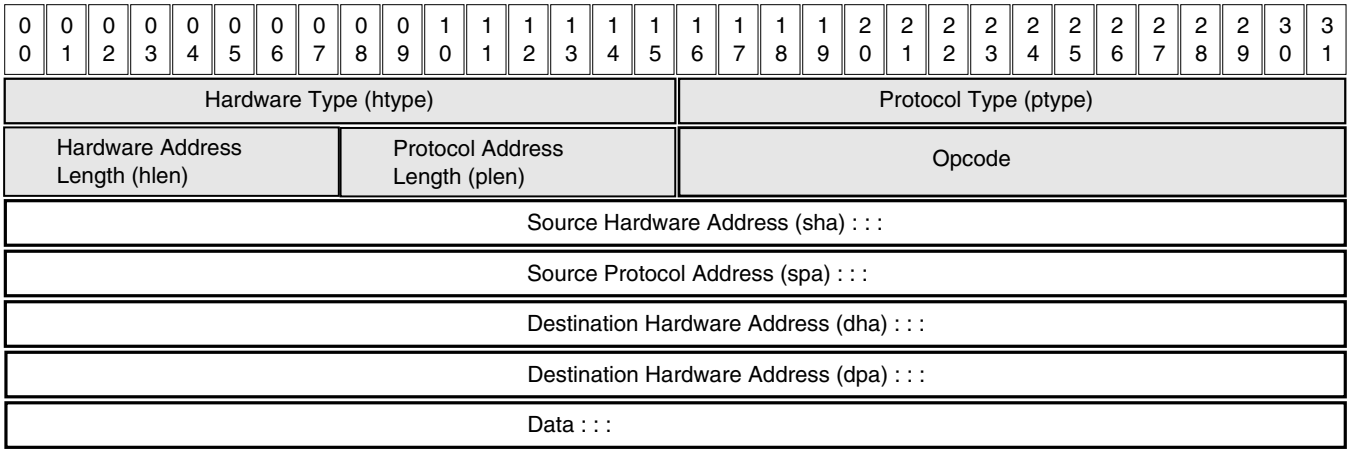

Figure 7. ARP header detail

We see that the first field in an ARP header is “Hardware type” and that this field is 2 bytes. This agrees with our definition provided in the first field element. Continuing in this manner, you can check that the fields accurately define the ARP header. Note that the last four fields of the ARP Header are variable length. In this example, the Source hardware address, Source protocol address, Destination hardware address, and Destination protocol address have been defined to be of size 6 bytes, 4 bytes, 6 bytes, 4 bytes, respectively. In summary, the format fields are used to define the fields of our custom protocol or shim header. This definition is used by the soft parser.

Following the element **format**, there is an element **execute-code**:

```
<execute-code>
  <before>
    <if expr="ethernet.type != 0x806">
      <if-true>
        <!-- Confirms Ethernet layer and exits-->
        <action type="exit" confirm="yes" nextproto="return"/>
      </if-true>
    </if>
  </before>
  <after>
    <assign-variable name="$shimoffset_1" value="$nxtHdrOffset"/>
    <assign-variable name="$nxtHdrOffset" value="$nxtHdrOffset + $headerSize"/>
    <action type="exit" confirmcustom="shim1" nextproto="end_parse"/>
  </after>
  </execute-code>
```

The element **execute-code** tells the soft parser what we want it to do. There is both a child element **before** and a child element **after**. It is not required to have both an element **before** and a element **after**, but there must be at least one of these in the element **execute-code**. The element **before** tells the soft parser what to while the parser is examining the previous protocol’s frame:

```
  <before>
    <if expr="ethernet.type != 0x806">
      <if-true>
        <!-- Confirms Ethernet layer and exits-->
        <action type="exit" confirm="yes" nextproto="return"/>
      </if-true>
    </if>
  </before>
```

In this example, the previous protocol is “Ethernet,” and so the element **before** tells the soft parser what to do while it is examining the Ethernet header. While the soft parser executes code in the **before** element, the “frame window” contains the ethernet header:

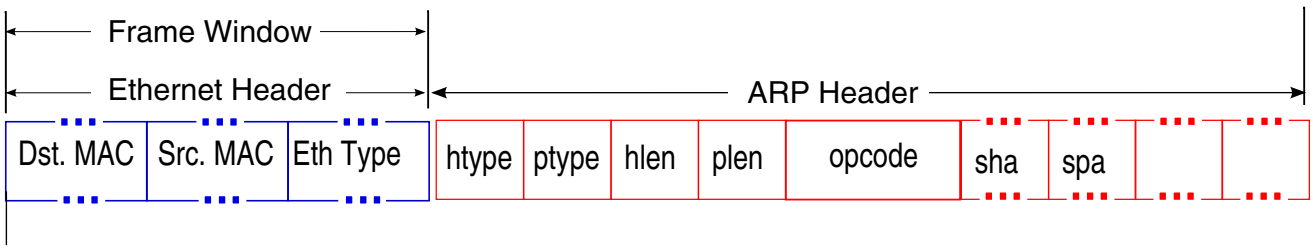

Figure 8. Ethernet header followed by ARP header

Here, the element **if** tells the soft parser to examine the field “ethernet.type” and if the field "ethernet.type" is not equal to `ox806` (ARP) the soft parser should perform the following actions:

1. “confirm” that the frame has an ethernet header (via the line-up confirmation vector),
2. “exit” the soft parser, and
3. “return” control to the hard parser (without advancing the frame window).

NOTE

If no further parsing of the packet headers is needed, instead of returning control to the hard parser, the action attribute **nextproto** should be set to “end_parse” instead of “return.” If the action attribute **nextproto** is set to “end_parse,” parsing of the frame’s headers is finished after the ethernet header is parsed.

If the field "ethernet.type" is equal to `0x806` (ARP) then the soft parser will move the frame window to the custom protocol and then execute the code in the element **after**:

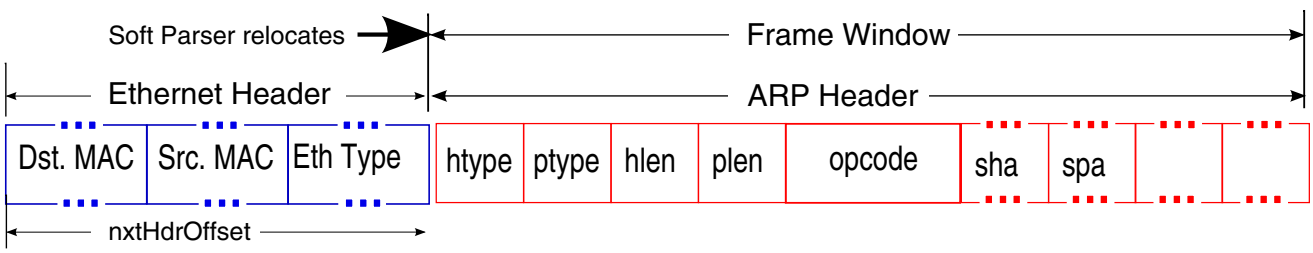

Figure 9. Ethernet header followed by ARP header after

```
<after>
  <assign-variable name="$shimoffset_1" value="$nxtHdrOffset"/>
  <assign-variable name="$nxtHdrOffset" value="$nxtHdrOffset + $headerSize"/>
  <action type="exit" confirmcustom="shim1" nextproto="end_parse"/>
</after>
```

Notice in element **after** there are child elements **assign-variable**. The custom protocol file supports the use of a set of defined variables (see **Frame Manager Configuration Tool User Guide Section A.1.6.3 Variables**). The element **assign-variable** assigns an expression to a variable. In custom protocol files, variables have prefix $. In this example, two variables have assignment statements: shimoffset_1 and nxtHdrOffset. These two variables point to elements in the Frame Manager “parse array”. The parse array is a data structure internal to the parser. See the table below for a description of the nxtHdrOffset and shimoffset_1.

Table 23. Parse array variables

Variable Name	Parse Array Byte	Description
nxtHdrOffset	47	Offset to header that has not yet been parsed.

Table continues on the next page...

Table 23. Parse array variables (continued)

Variable Name	Parse Array Byte	Description
shimoffset_1	32	Byte position within the frame's header where parsing reached the point to be considered shim1 (i.e., custom protocol)

Prior to executing the assignment statement code, the "nxtHdrOffset" represents the offset from the start of the ethernet header to the start of the ARP header.

The first assignment statement assigns to the variable "shimoffset_1" the value "nxtHdrOffset". The second assignment statement assigns to the variable nxtHdrOffset the value (nxtHdrOffset + headerSize):

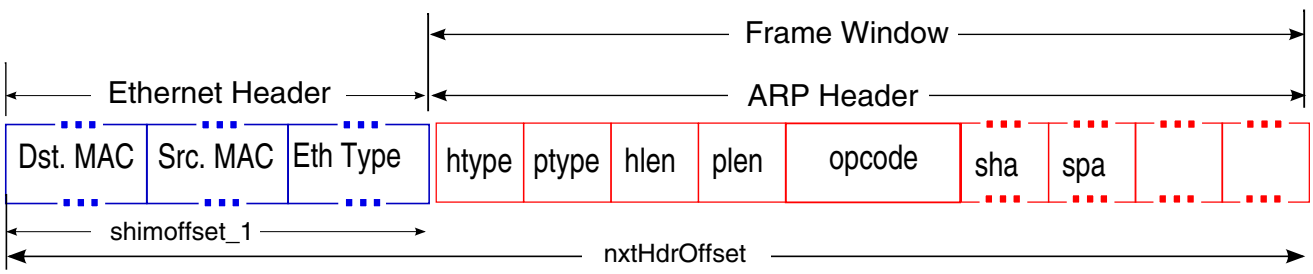

Figure 10. Ethernet header followed by ARP header finish

While the frame window contains the custom protocol, the variable headerSize is the custom protocol header size. So the second assignment statement advances nxtHderOffset to the end of the custom protocol.

The element **action** following the assignment statements tells the soft parser to perform the following actions:

1. "shim1" confirm that the frame has a custom header (via the line-up confirmation vector),
2. "exit" the soft parser, and
3. "end_parse" parsing of the frame headers is finished after parsing the custom header.

Note that shimoffset_1 and shim1 are used when referring to the custom protocol. If we were to define a second custom protocol, we would use shimoffset_2 and shim2.

7.2 Example Usage of a Custom Protocol in a Policy File

Once a custom protocol is defined, it can be used in a policy file. There is an example policy file `usdpaa_policy_hash_ipv4_arp_coarse_classify.xml` in *Freescale Linux SDK for QorIQ Processors* that configures Frame Manager to perform coarse classification (exact match) using fields from the ARP header. One distribution defined in `usdpaa_policy_hash_ipv4_arp_coarse_classify.xml` is distribution "arp_dist0." The distribution and classification elements used to define "arp_dist0" are displayed below.

Table 24. Distribution and classification elements

Element	Code Example

Distribution	`<distribution name="arp_dist0">`
	`  <queue base="1" count="1"/>`
	`  <protocols>`
	`  <protocolref name="arp"/>`
	`  </protocols>`

Table continues on the next page...

Table 24. Distribution and classification elements (continued)

Element	Code Example
	<action type="classification" name="arp_htype_clsf0"/> </distribution>
Classification	<classification name="arp_htype_clsf0"> <key> <fieldref name="arp.htype"/> </key> <entry> <data>0x1</data> <action type="classification" name="arp_ptype_clsf0"/> </entry> <action type="distribution" name="fman_drop_dist" condition="on-miss"/> </classification>
Classification	<classification name="arp_ptype_clsf0"> <key> <fieldref name="arp.ptype"/> </key> <entry> <data>0x800</data> <action type="classification" name="arp_opcode_clsf0"/> </entry> <action type="distribution" name="fman_drop_dist" condition="on-miss"/> </classification>
Classification	<classification name="arp_opcode_clsf0"> <key> <fieldref name="arp.opcode"/> </key> <entry> <data>0x1</data> <action type="classification" name="arp_tpa_clsf0"/> </entry> <action type="distribution" name="fman_drop_dist" condition="on-miss"/> </classification>
Classification	<classification name="arp_tpa_clsf0"> <key> <fieldref name="arp.tpa"/> </key> <entry> <data>0xc0a80a01</data> <action type="distribution" name="default_dist0"/> </entry> <action type="distribution" name="fman_drop_dist" condition="on-miss"/> </classification>

In the distribution “arp_dist0” there is an element **protocols**. Here **protocolref** is the custom protocol “arp.” If an Rx frame has a “arp" header then the element **action** specifies to Frame Manager to perform classification using “arp_htype_clsf0.” The soft parser will determine whether or not the packet contains an “arp” header.

The classification “arp_htype_clsf0” uses **fieldref** “arp.htype” and **data**`0x1`. If the ARP Hardware type field is equal to `0x1` then action will be for Frame Manager to perform a second classification “arp_ptype_clsf0”. If the ARP Hardware type is not equal to `0x1` then there is a “miss” and the action will be for Frame Manager to proceed to distribution “fman_drop_dist”.

The classification “arp_ptype_clsf0” uses **fieldref** “arp.ptype” and **data** `0x800`. If the ARP Protocol type field is equal to `0x800` then action will be for Frame Manager to perform a third classification “arp_opcode_clsf0.” If the ARP Hardware type is not equal to 0x800 then there is a “miss” and the action will be for Frame Manager to proceed to distribution “fman_drop_dist”.

The classification “arp_opcode_clsf0” uses **fieldref** “arp.opcode” and **data** “0x1.” If the ARP Operation field is equal to `0x1` then action will be for Frame Manager to perform a fourth classification “arp_tpa_clsf0.” If the ARP Hardware type is not equal to `0x800` then there is a “miss” and the action will be for Frame Manager to proceed to distribution “fman_drop_dist”.

The classification “arp_tpa_clsf0” uses **fieldref** “arp.tpa” and **data**`0xc0a80a01`. If the ARP Target protocol address field is equal to 0xc0a80a01 then action will be for Frame Manager to proceed to distribution “default_dist0.” If the ARP Hardware type is not equal to `0xc0a80a01` then there is a “miss” and the action will be for Frame Manager to proceed to distribution “fman_drop_dist”.

7.3 Example Usage on Target Board

When using a custom protocol file, the name of the custom protocol file is passed to FMC on the command line using the “-s” argument, e.g.,

```
$ cd /usr/etc/
$ fmc -c usdpaa_config_p4_serdes_0xe.xml
-p usdpaa_policy_hash_ipv4_arp_coarse_classify.xml
-s usdpaa_custom_coarse_classify_netpdl.xml -a
```

8 Appendix A

Standard Protocol headers used in examples.

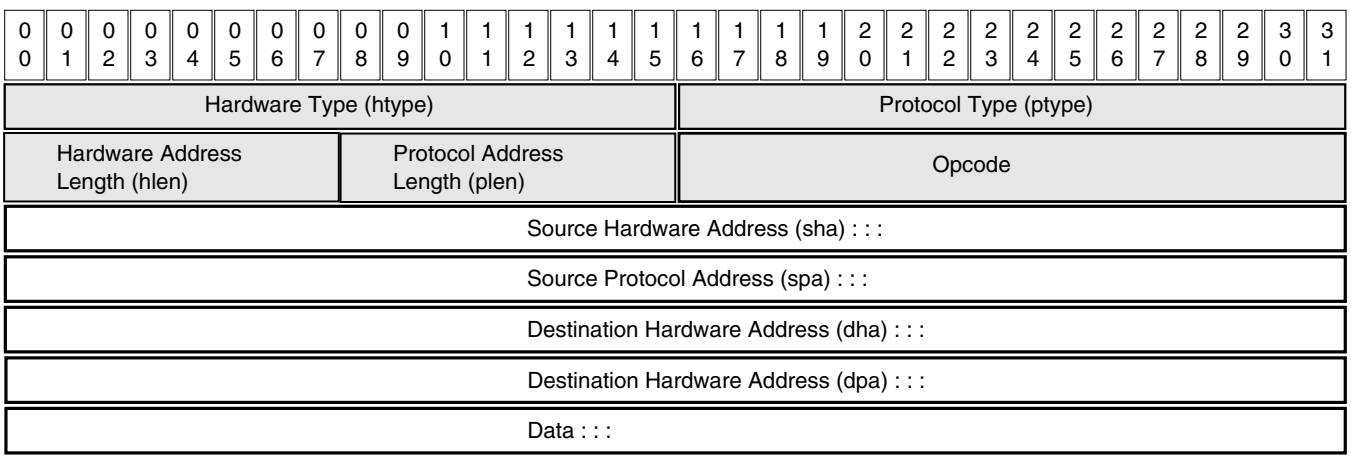

Figure 11. ARP Header

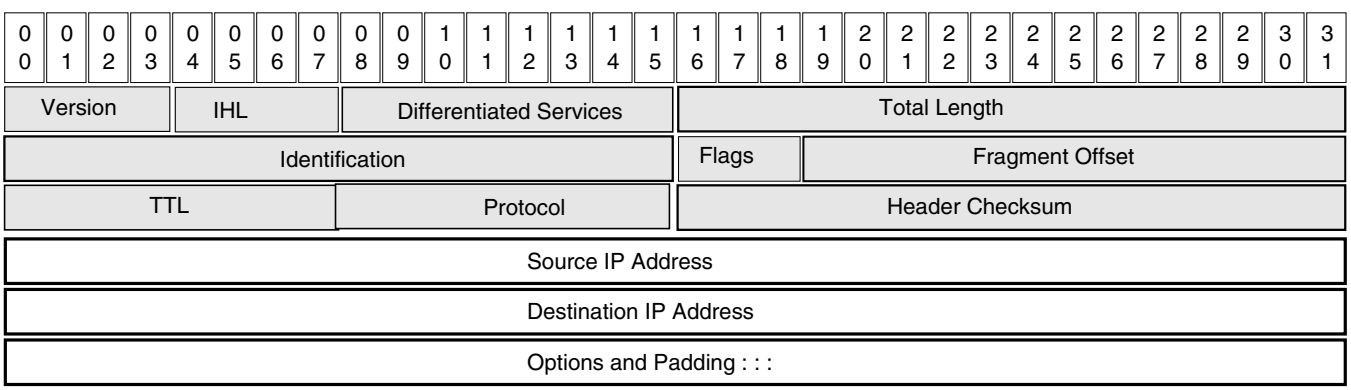

Figure 12. IP Header

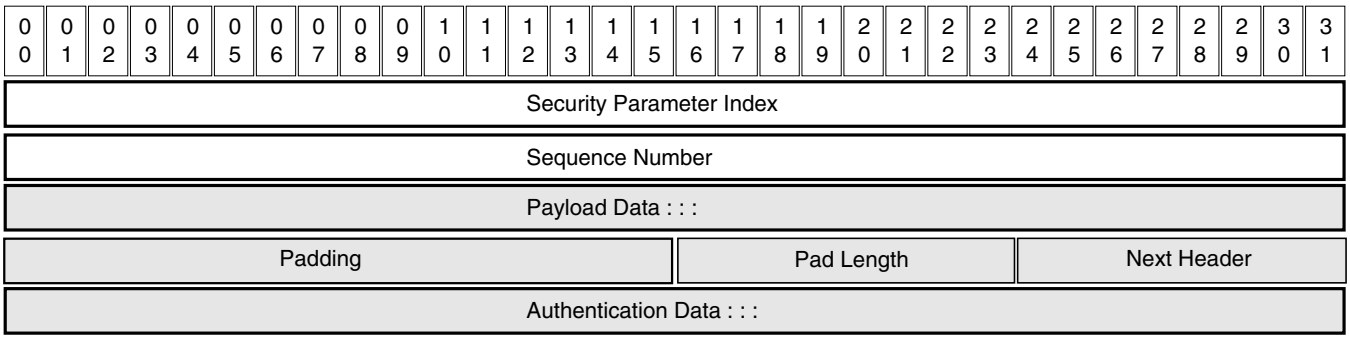

Figure 13. ESP Header

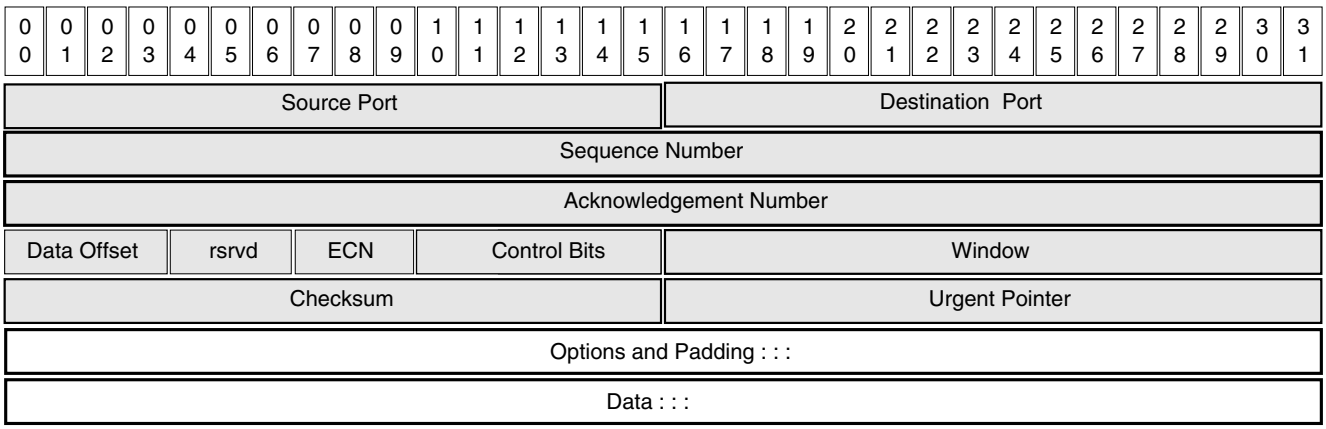

Figure 14. TCP Header

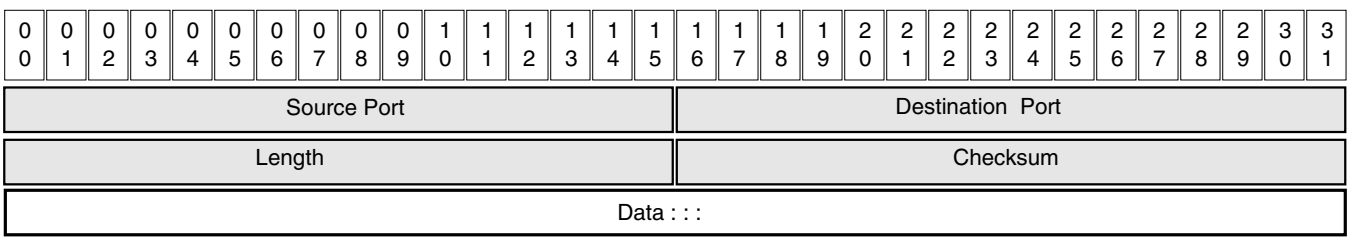

Figure 15. UDP Header

9 Appendix B

FMC policy files and custom protocol files should use the names as specified in the standard protocol definitions file.

9.1 Standard Protocols recognized by Hard Parser

The Hard Parser recognizes standard protocol headers. See **QorIQ Data Path Acceleration Architecture Reference Manual Chapter 8.8.4.7 Hard Header Examination Sequences (HXSx)** for detailed information regarding the standard protocols recognized by the hard parser. The standard protocol definitions file `hxs_pdl_v3.xml` defines for FMC these standard protocols and their fields. By default FMC uses the standard protocol definition file that resides in the directory `/etc/fmc/config`. It is not necessary to pass a command line argument to FMC providing the path to this file.

NOTE

On your host, the standard protocol definitions file resides in the directory: `QorIQ-SDK-<version>-<date>-yocto/build_<target_board>_release/tmp/work/<build_target>-fsl-linux>/fsl-image-<file_system_image>/rootfs/etc/fmc/config/hxs_pdl_v3.xml`

FMC policy files and custom protocol files should use the protocol names as specified in the standard protocol definitions file:

- ethernet
- vlan
- llc_snap
- mpls
- ipv4
- ipv6
- tcp
- udp
- gre
- pppoe
- minencap
- sctp
- dccp
- ipsec_ah
- ipsec_esp

10 Revision history

This table summarizes revisions to this document.

Table 25. Revision history

Revision	Date	Description
0	10/2013	Initial public release.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number AN4760
Revision 0, 10/2013