LNX PFE Driver User Manual

Rev. 1.3.0 — 13 April 2023 User manual

Revision History

Revision history

Revision	Change Description
preEAR 0.4.0	Initial version for preEAR (FPGA/x86 platform) [JPet]
preEAR 0.4.1	Added VDK info [JPet]
preEAR 0.4.3	Removed VDK, Added S32G [JPet]
EAR 0.8.0	Added device-tree config [JPet]
EAR 0.8.0 P1	Added libfci [JPet]
BETA 0.9.0	DTS update for kernel 5.4-rt [JPet]
BETA 0.9.1	Added Master/Slave [JPet]
BETA 0.9.2	Added performance info [JPet]
BETA 0.9.3	Added PTP [OSpac], Added cut1.1 info [JPet]
BETA 0.9.4	Added Reserved Memory Regions chapter [CMan], STR, DT updates [JPet]
BETA 0.9.5	Updated STR, DT, Added AUX [JPet] Removed cut 1.1 references. [CMan]
BETA 0.9.6	Added Diagnostic Options [CVN] Document template changed [LBob] New content: Introduction, Usage, Features, etc. Various corrections and rework. [CMan]
BETA 0.9.7	Added an example how to place PFE BMU2 pool in SRAM. [MHrd] Added information about Master-Slave FCI ownership. [LBob] Added Diagnostic Option Driver debug logs. [MHrd] Updates: Ingress QoS (includes new limitations), "Perf. Consideration", BMU2 in SRAM init (+perf. consid.), supported boards [CMan] Added note how to strip driver. [MHrd] Updated AUX netdevice info [JPet] Added Slave coherency settings prerequisites. [MHrd]
RTM 1.0.0	Updated cross-document references. [LBob] Added brief libfci_cli manual. [LBob] Added chapter about Egress QoS known limitations. [LBob] Device tree changes [MHrd] Updates: Reserved Memory Regions, Ingress QoS Known Limitations, Network Interfaces [CMan] Updates: "Supported development boards" table, "DT node config." and "Perf. Consideration" [CMan]
RTM 1.1.0	Added FCI ownership chapter. [MHrd] Added Ingress QoS L4 Port range based classification limitation. [CMan]

Revision history...*continued*

Revision	Change Description
1.2.0	Added description of libfci_cli daemon into brief libfci_cli manual. [LBob] Added Health Monitor. [MHrd] Added Extended mode support of PFE interfaces (RMII). [MHrd] Documented the 'G3 Routing Table entries in LMEM' optimization, see '4.5 Performance consideration'. [CMan]
1.3.0	Updated IEEE1588 Support, subsection External Timestamp Mode. [MHrd] Added Jumbo Frames support content, and Lossless Tx chapter. [CMan] Added HIF netdev. [JPet] Updated prerequisites for Linux Slave coherency settings. [MHrd]

1 Introduction

1.1 References

[1] *FCI API Reference*, available within the PFE Driver Release Package from FlexNet (nxp.com). The package can be obtained via the web portal under the section **Automotive SW - S32G - PFE Driver + Standard Firmware**.

[2] *S32G PFE Firmware User Manual*, available as part of the S32G2 PFE Firmware release package from FlexNet (nxp.com). The package can be obtained via the web portal under the section **Automotive SW - S32G - PFE Driver + Standard Firmware**.

[3] *Linux BSP 36.0 User Manual for S32G2 platforms*
Linux BSP 36.0 User Manual for S32G3 platforms
Available within the Linux Board Support Package (BSP) from FlexNet (nxp.com). The package can be obtained via the web portal under the section **Automotive SW - S32G - Linux BSP**

1.2 Acronyms and Definitions

Table 1. Acronyms and Definitions

Term	Definition
PFE	Packet Forwarding Engine (PFE) is a hardware based accelerator for classification and forwarding Ethernet traffic between selected interfaces. It requires a firmware for its operation.
PFE Firmware	Software that executes inside PFE, implementing the required functionality. The PFE Firmware is provided in the form of a binary executable by NXP. For details refer to the *PFE Firmware User Manual* [2].
S32G	S32G system on a chip device containing PFE.
Host	The Host or the Host Core is the general purpose CPU that runs the PFE Linux driver.
HIF, HIF Channel, or HIF port	The Host Interface Channel is the hardware entity that provides data packet flow between the Host (PFE Linux driver) and PFE. For simplicity, it's also referred to as a Host side PFE port.
EMAC, or EMAC port	Standard IEEE 802.3 MAC that is a network (LAN) side PFE port.
IHC	Communication channel in PFE between two HIF Channels.
FCI, or FCI API	Fast Control Interface - is the communication API that allows user space applications running on the Host to configure and retrieve the status of PFE. Refer to the *FCI API Reference* [1] for details.
LibFCI	Is the user space library that implements the FCI API on top of Netlink (RFC 3549). On the kernel side, the Netlink messages passed down by LibFCI are translated into PFE driver function calls.
Linux BSP	The NXP Auto Linux BSP (Board Support Package) is a collection of source code that can be used to build the U-Boot boot loader, the Linux kernel image, a root file system and, optionally, an ARM Trusted Firmware (TF-A) image for the supported boards. Refer to the *Linux BSP User Manual* [3] for details.
IP	Internet Protocol which can be both version 4 or 6.
VLAN	Virtual LAN as defined by IEEE 802.1Q.
L2, L3, L4	Layer of the ISO/OSI model.

1.3 Overview

The software stack that manages PFE has the following components, highlighted in the diagram below:

- Firmware - the PFE Firmware is detailed in the *PFE Firmware User Manual*[2].
- Ethernet Driver - the Linux PFE Ethernet driver is detailed in the current manual.
- LibFCI - the user space library implementing the FCI API, as detailed in the *FCI API Reference*[1].

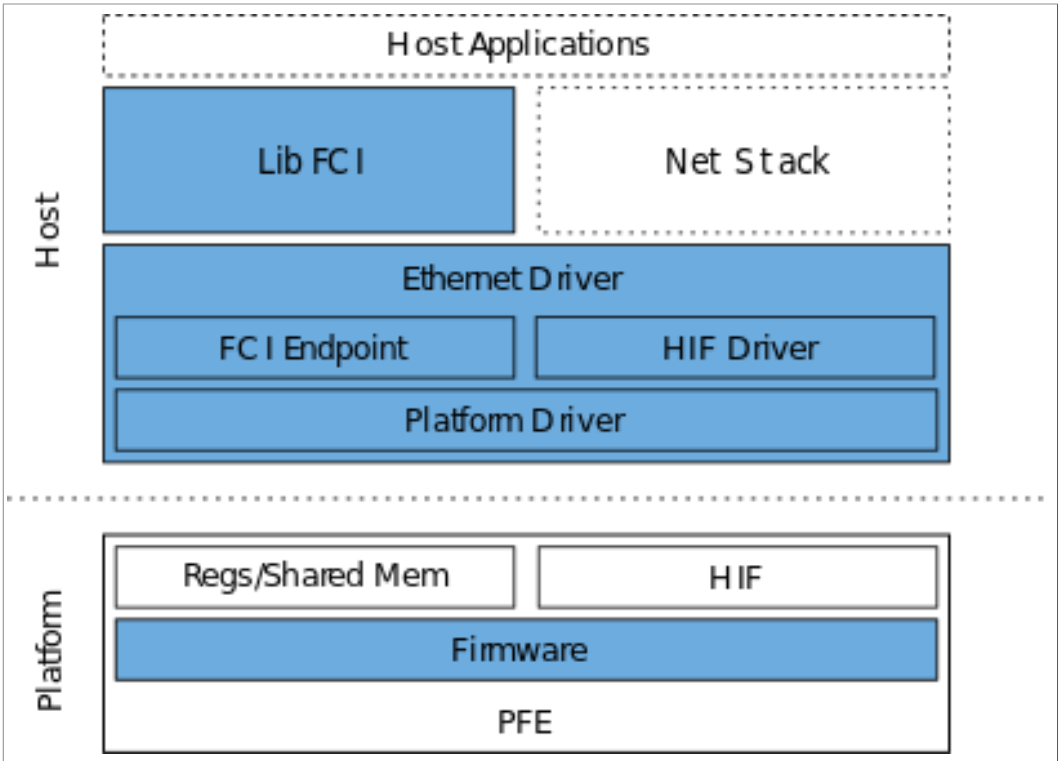

Figure 1. PFE software stack components

The Linux PFE Ethernet driver contains three main sub-blocks:

- The **Platform Driver** is an OS agnostic low-level driver that owns the hardware and firmware configuration. Hardware configuration is encapsulated by separate software modules, each module controls access to the underlying PFE block and provides a higher level API for its users.
- The **HIF driver** is the part of the Linux driver that manages the HIF channels and the packet traffic between PFE and the Linux networking stack.
- The **FCI endpoint** is the control path driver that implements the kernel space side of the FCI for runtime configuration and monitoring of PFE. Primarily it converts user space fast path configuration commands received via LibFCI into Platform Driver function calls.

The Linux PFE Ethernet driver runs inside the Linux Kernel space and connects PFE to the Linux networking stack. It provides access to physical Ethernet interfaces by exposing Linux network device interfaces that are configurable via standard Linux networking tools (i.e. iproute2, net-tools, iputils).

2 Features

2.1 Network Interfaces

PFE is a multi-port device. From the Linux driver perspective, PFE has up to 3 network facing ports that are standard IEEE 802.3 MAC ports, named EMAC0 to EMAC2 (Ethernet MACs), and up to 4 Host facing ports called HIF (Host Interface) Channels, identified as HIF0 to HIF3. Network packets are routed between these ports by the PFE Firmware, based on the default configuration provided by the Linux driver and based on additional configurations provided by the user via LibFCI. The PFE firmware uses 2 central constructs for packet flow configuration - software structures exported as part of the PFE Firmware API: **Physical Interfaces** (**PHY_IF**s) and **Logical Interfaces** (**LOG_IF**s).

2.1.1 Physical Interfaces (PHY_IF)

Every PFE port has a Physical Interface (PHY_IF) attached to it. PHY_IFs are created at driver load time and are permanent. They serve mainly for port level packet filtering and as an anchor for chaining packet classification rules of ingress traffic. The classification rules are specified by LOG_IFs.

Example - EMAC2 PHY_IF ("emac2") with Default LOG_IF ("pfe2") in promiscuous mode:

```
root@s32g274ardb2:~# libfci_cli phyif-print
[...]
2: emac2
      <ENABLED>
      <promisc:OFF, mode:DEFAULT, block-state:NORMAL>
      <vlan-conf:OFF, ptp-conf:OFF, ptp-promisc:OFF, q-in-q:ON>
      <discard-if-ttl-below-2:OFF>
      ingress: 2 egress: 6 discarded: 0 malformed: 0
      MAC:
          00:01:be:be:ef:33
          33:33:00:00:00:01
          01:00:5e:00:00:01
          33:33:ff:be:ef:33
      mirrors:
          rxmirr0: ---
          rxmirr1: ---
          txmirr0: ---
          txmirr1: ---
      logical interfaces:
       4: pfe2
            <ENABLED>
            <promisc:ON, match-mode:AND, discard-on-match:OFF, loopback:OFF>
            accepted: 2 rejected: 0 discarded: 0 processed: 2
            egress: hif2
            match-rules: ---
```

2.1.2 Logical Interfaces (LOG_IF)

Logical Interfaces (LOG_IFs) are attached to PHY_IFs and specify, based on classification rules, to what PFE port (or ports) should the ingress traffic be directed to. LOG_IFs can be added or removed, configured, enabled or disabled, at runtime via FCI. Multiple LOG_IFs can be attached to the same PHY_IF in a linked list, and the last added LOG_IF is the first one to classify the incoming traffic (on match). The first added LOG_IF is also called the *Default LOG_IF* and it usually has a catch all traffic rule (promiscuous mode).

Example - EMAC2 Default LOG_IF in promiscuous mode ("pfe2") attached by the Linux driver to the EMAC2 PHY_IF ("emac2") to direct the traffic to the HIF2 Port (identified by the "hif2" PHY_IF):

```
root@s32g274ardb2:~# libfci_cli logif-print
[...]
 4: pfe2
      <ENABLED>
      <promisc:ON, match-mode:AND, discard-on-match:OFF, loopback:OFF>
      accepted: 2 rejected: 0 discarded: 0 processed: 2
      parent: emac2
      egress: hif2
      match-rules: ---
[...]
```

2.1.3 Linux Network Interfaces

The Linux network interfaces, or Linux net devices, are instances of network traffic endpoints from the Host OS perspective.

Example - PFE Linux Network interface:

```
root@s32g274ardb2:~# ip link show dev pfe2
7: pfe2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP mode
 DEFAULT group default qlen 1000
    link/ether 00:01:be:be:ef:33 brd ff:ff:ff:ff:ff:ff
```

The Linux net devices are associated with at least one HIF port. A Linux net device can be associated with multiple HIF ports to allow for traffic load balancing. The association with multiple HIF ports is specified by the `'nxp,pfeng-hif-channels'` device tree property. The HIF port(s) can also be shared among multiple net devices, which is of particular importance for the cases where the driver instance can own only a limited number of HIF ports (e.g. one port only).

Known Limitation

Currently, the PFE Linux driver does not support load balancing, as it selects only the first HIF port for egress from the `'nxp,pfeng-hif-channels'` list.

The Linux driver creates *Default Logical Interfaces* (LOG_IFs) for some of the PFE ports it manages, depending on what interface modes are configured for the Linux net devices associated with the PFE ports. For the net devices bound to EMAC ports, the Linux Driver creates a Default LOG_IF for each bound EMAC port with the same name as the corresponding Linux Network Interfaces (i.e. "*pfeX*"), so that the ingress traffic can be directed to one of the HIF ports associated with the net device, see above example.

The driver also automatically creates Default Logical Interfaces for the HIF Ports associated with network interfaces (*hif0-logif* for hif0, *hif1-logif* for hif1 etc). The actual use of these LOG_IFs depends on network interface mode and individual user configuration of the HIF port (done via LibFCI).

Note: *See FCI API Reference [1] for HIF port configuration use cases.*

Example - Default HIF LOG_IF associated to the "emac2" PHY_IF which is bound to the "pfe2" network interface:

```
root@s32g274ardb2:~# libfci_cli logif-print
[...]
5: hif2-logif
      <ENABLED>
      <promisc:ON, match-mode:AND, discard-on-match:OFF, loopback:OFF>
      accepted: 13 rejected: 0 discarded: 0 processed: 13
```

```
      parent: hif2
      egress: emac2
      match-rules: ---
[...]
```

The PFE Linux Network Interfaces are specified in the PFE device tree node as `'ehternet'` sub-nodes with the `'"nxp,s32g-pfe-netif"'` compatibility strings.

There are three PFE Linux Network Interface types, two are bound to a PFE port (EMAC or HIF), the third is determined by the presence of `'nxp,pfeng-netif-mode-aux'` property:

1. EMAC bound network interfaces, the property `'nxp,pfeng-linked-phyif'` links to particular EMAC port.
2. HIF bound network interfaces, the property `'nxp,pfeng-linked-phyif'` links to particular "foreign" HIF port.
3. The Auxiliary (AUX) network interface, set when the `'nxp,pfeng-netif-mode-aux'` property is present.

2.1.4 EMAC bound Network Interfaces

For EMAC bound network interfaces the Linux net device instance owns an EMAC port. As part of its initialization, the net device is in charge of configuring the link layer, including the EMAC port, and it must manage the link status at runtime. For these interfaces, the PFE driver creates the Default LOG_IF of the EMAC port to ensure that, by default, the ingress traffic will be directed to the associated net device. The Default LOG_IF is configured to direct the traffic to one of the HIF ports associated with the net device.

When working together with an AUX interface, on reception, the EMAC bound network interface can be statically configured to receive only 'management' traffic, by adding the `'nxp,pfeng-netif-mode-mgmt-only'` property to its `'ethernet'` device tree node. In this configuration, 'non-management' traffic is relayed to the AUX net device. The following frames are always marked as 'management' by the firmware:

- PTP frames,
- Egress Timestamp Reports.

For a detailed list of protocols recognized and tagged as 'management', refer to the *PFE Firmware User Manual* [2].

On transmission, the EMAC bound network interfaces operate in the TX_INJECT mode. In this mode, the driver marks the custom HIF Tx header of every packet to instruct PFE to skip the classification stage and send the packets directly to the egress EMAC port configured in the custom HIF Tx packet header. The driver provides the same EMAC port for egress as the bound EMAC port used for ingress.

The following picture exemplifies a multi HIF port configuration, where each HIF port serves exactly one EMAC bound network interface in the TX_INJECT mode (A to C):

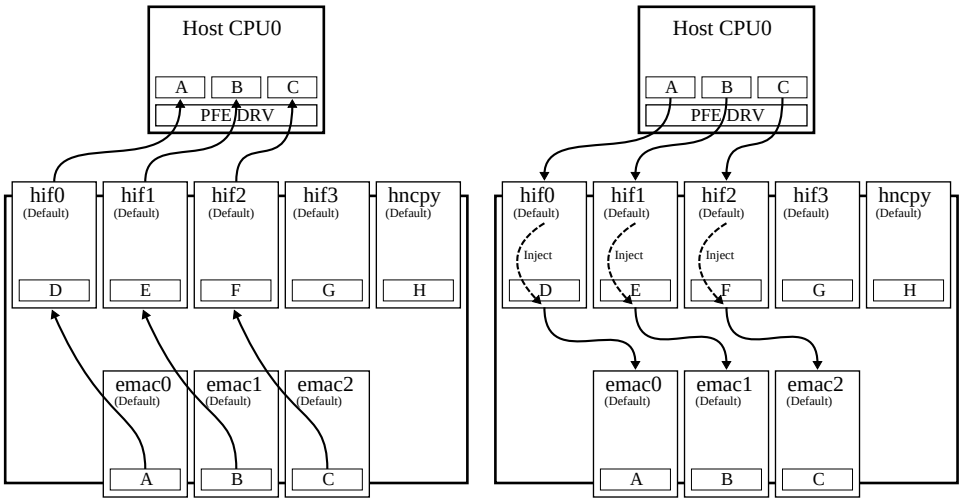

Figure 2. Multiple TX_INJECT interfaces with separate HIF ports

The following picture represents shared HIF port configuration used by multiple EMAC bound network interfaces in the TX_INJECT mode (A to C):

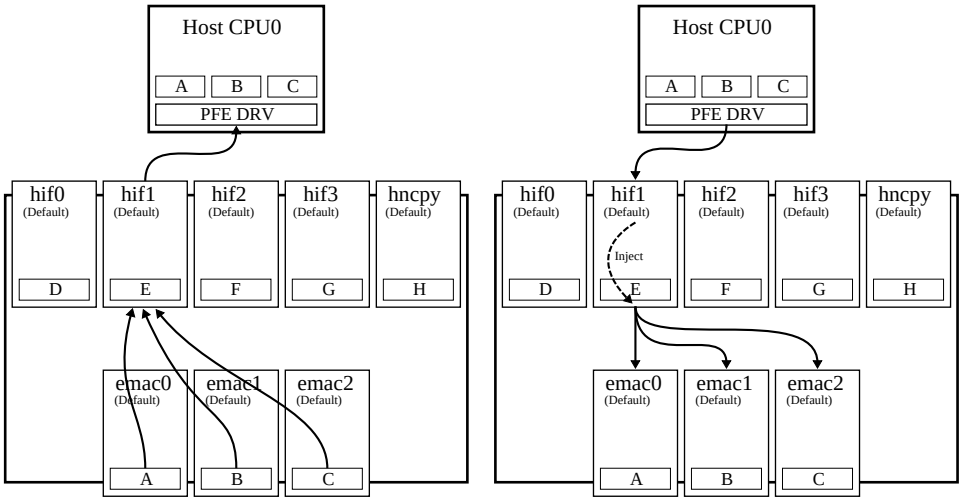

Figure 3. Shared HIF port by multiple TX_INJECT interfaces

When an EMAC bound network interface uses a shared HIF port, its associated Default HIF LOG_IF is disabled.

2.1.5 HIF bound Network Interfaces

Similar to EMAC bound network interfaces, the driver supports also HIF bound network interfaces in which case the Linux net device instance uses a "foreign" HIF port as destination for egress traffic. "Foreign" here means a HIF channel port that is not owned by the current driver instance as Host port (e.g. running on the A53 Host), but by a different driver instance (e.g. running on the M7 Host).

When working together with an AUX interface, on reception, the HIF bound network interface can be statically configured to receive only 'management' traffic, by adding the `'nxp,pfeng-netif-mode-mgmt-only'` property to its `'ethernet'` device tree node. In this configuration, 'non-management' traffic is relayed to the AUX net device. The following frames are always marked as 'management' by the firmware:

- PTP frames,
- Egress Timestamp Reports.

For a detailed list of protocols recognized and tagged as 'management', refer to the *PFE Firmware User Manual* [2].

On transmission, the HIF bound network interfaces operate in the TX_INJECT mode. In this mode, the driver marks the custom HIF Tx header of every packet to instruct PFE to skip the classification stage and send the packets directly to the egress HIF port configured in the custom HIF Tx packet header.

The following picture exemplifies the traffic flow between two HIF bound network interfaces (two separate driver instances):

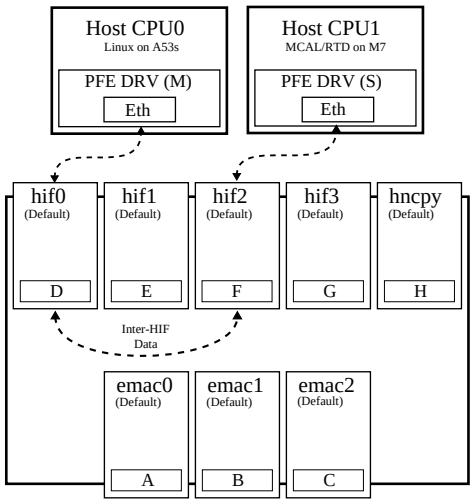

Figure 4. HIF to HIF traffic

2.1.6 Auxiliary (AUX) Network Interface

The Auxiliary (AUX) Network Interface does not own any EMAC or foreign HIF port. This interface captures ingress traffic that is not dispatched to an EMAC/HIF attached interface. On egress, packets are dispatched according to the classification rules programmed to the HIF port used by the AUX interface. For this to work, the driver creates and enables a Default HIF LOG_IF for the AUX interface associated HIF port, and its configuration can be changed from user space (via LibFCI) to forward traffic based on certain rules, depending on targeted use case.

The AUX interface can be used to build more complex configurations, including PFE accelerated L2 bridge use cases and/or L3 router use cases (see *FCI API Reference* [1]).

2.1.6.1 AUX Interface use cases

There are situations when the driver needs to dispatch a frame which has been received from some "unknown" source port, like for instance from the HIF Port owned by the PFE driver residing on another Host, as depicted in the following figures.

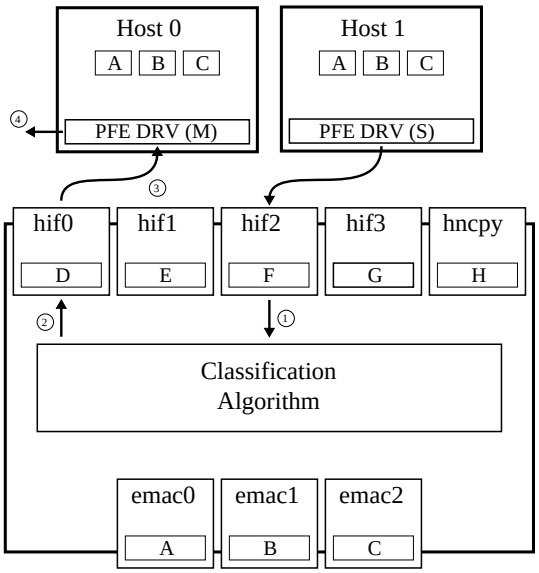

Figure 5. Unknown Rx Traffic without AUX

Here the sequence starts in driver instance running within Host 1 (Slave) which is sending a frame which will at the end reach the driver instance within the Host 0 (Master). Processing then includes the following steps:

1. The frame is received via HIF2.
2. The Classification rules within PFE decide to which Physical Interface the frame shall be forwarded.
3. The driver instance running within the Host 0 receives the frame and attempts to dispatch it to some of the available network interfaces (traffic endpoints).
4. Because there is no endpoint associated with the ingress interface HIF2 (F) the frame can't be dispatched and must be dropped.

To cover this scenario the driver must provide a default endpoint which would accept the "unknown" traffic, i.e. traffic coming from a source port for which the target driver has no associated network interface.

If the AUX interface (endpoint) is introduced to Host0 however, the reception of "unknown" traffic turns into following situation:

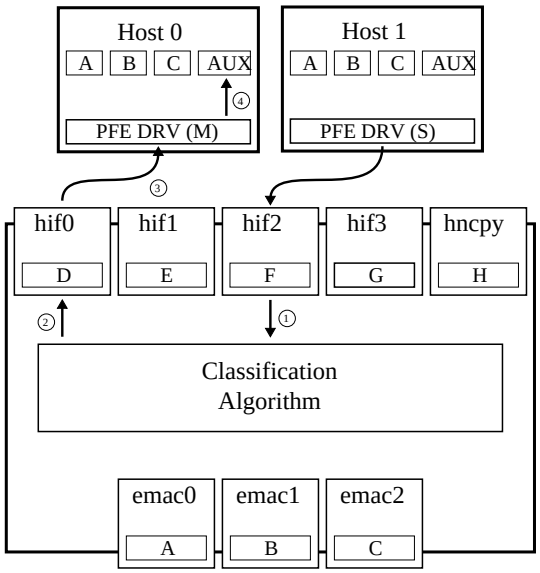

Figure 6. The AUX Endpoint RX

1. The Frame is received via HIF2.
2. The Classification Algorithm within PFE decides to which Physical Interface the frame shall be forwarded.
3. The driver instance running within Host 0 receives the frame and attempts to dispatch it to some of available endpoints.
4. Because none of standard endpoints A, B, C matches the ingress interface of the frame, the HIF2 (F), the frame is passed to the AUX endpoint.

Traffic being transmitted via the AUX interface can be delivered to PFE via an arbitrary HIF port associated to the network interface. However, each HIF port acts as a standalone Physical Interface (PHY_IF) and its configuration directly affects the way in which PFE will route the traffic.

2.2 Master-Slave instances

The driver can run in multiple instances within the same system to share the PFE provided connectivity using dedicated host interfaces. The intended use case is to run a Master instance within one Host, and one or more Slave instances from other Hosts that can have access to PFE, even from other operating systems (OSes) than the Host OS running the Master instance. To enable Master-Slave, the driver is extended into two binaries:

1. **`pfeng.ko`** which additionally contains Inter-HIF Communication (IHC) support and which acts as Master;
2. **`pfeng-slave.ko`**, which also contains IHC, but lacks the direct PFE controller manageability.

Note: *The Master-Slave feature requires additional configuration in the device tree (see also Section 4.4 device tree configuration).*

2.2.1 Prerequisites

Linux Slave coherency settings

The integrator must be aware that the *PFE Port Coherency Enable* (*PFE_COH_EN*) general-purpose register must be set only after the A53 partition is out of reset. Enabling PFE coherency before activating the A53 partition results in inconsistent PFE initialization state. When this condition occurs, the Master driver, running on the M7 cores, cannot receive or transmit any traffic. On the A53 side, the Linux Slave issues ETIMEDOUT errors for the IHC traffic. A typical faulty scenario would be booting M7, which tries to set *PFE_COH_EN* and starts the PFE Master driver, concurrently with the A53 partition coming out of reset.

The Slave driver module allows manage PFE HIF channel coherency, module option `manage_port_coherency` can enable it, the feature is disabled by default.

```
manage_port_coherency:   1 - enable HIF port coherency management,
                         default is 0 (int)
```

The Slave driver coherency management can be used only if there is no any other instance of Slave driver on A53.

2.2.2 FCI ownership

Only one driver instance in the Master-Slave setup can be an "FCI owner" and can issue FCI commands to configure PFE. For more details see Section 2.10.6.

2.2.3 Master

To build the PFE driver with Master functionality the following compile-time options are required (the full build procedure is described here):

```
make <options> PFE_CFG_MULTI_INSTANCE_SUPPORT=1 PFE_CFG_PFE_MASTER=1 ...
```

The Master functionality requires marking one of the configured HIF port instances for IHC transport in device tree by adding the `'nxp,pfeng-ihc-channel'` property to the PFE device node. For example:

```
pfe: pfe@46000000 {
        compatible = "nxp,s32g-pfe", "fsl,s32g274a-pfeng";
        reg = <0x0 0x46000000 0x0 0x1000000>,
              <x0 0x4007ca00 0x0 0x100>;
        reg-names = "pfe-cbus", "s32g-main-gpr";
        ...
        nxp,pfeng-hif-channels = <PFE_HIF_CHANNEL_0>,
                                 <PFE_HIF_CHANNEL_1>,
                                 <PFE_HIF_CHANNEL_2>;
        nxp,pfeng-ihc-channel = <PFE_HIF_CHANNEL_0>;
        ...
};
```

2.2.4 Slave

To build the PFE driver with Slave functionality, the following compile-time options are required (the full build procedure is described here):

```
make <options> PFE_CFG_MULTI_INSTANCE_SUPPORT=1 PFE_CFG_PFE_MASTER=0 ...
```

If the Slave driver is not intended to be an FCI owner, it can be compiled without FCI support:

```
make <options> PFE_CFG_MULTI_INSTANCE_SUPPORT=1 PFE_CFG_PFE_MASTER=0
 PFE_CFG_FCI_ENABLE=0 ...
```

Slave functionality requires two configuration options in the device tree:

- Setting the global configuration option `'nxp,pfeng-master-channel'` with Master's HIF channel number (see example in Section 2.2.3). This option can be overwritten by the command line parameter `'master_ihc_chnl'`.
- Marking one of the configured network interfaces for IHC transport in the device tree by setting `'nxp,pfeng-ihc-channel'` DT property, similarly to Master. See Section 2.2.3.

2.2.5 Runtime

2.2.5.1 Startup synchronization

A crucial aspect in running a Master-Slave scenario is to ensure that the driver instances synchronize correctly at start-up. The Slave should not start its initialization before the Master is functional, otherwise the PFE can be rendered non-functional. To prevent this, a Host/OS independent synchronization mechanism is build within the PFE driver. The mechanism, also called the Master Detection procedure, uses global system registers as mailboxes for signalling between the Master and Slave instances, and the synchronization is done in two steps: the 'PFE controller reset ready' step, and the 'Master initialization ready' step.

Master Detection procedure has dependencies on other Hosts and OSes. There may be scenarios where the Master Detection fails or cannot be used. For such scenarios, use the following fallback procedure:

1. Run the Master driver instance first and ensure that the network interface for IHC communication is up, before launching the Slave instances.
2. Load the Linux Slave driver instance with the Master Detection support **disabled** (command line parameter `disable_master_detection`):

```
insmod pfeng-slave.ko disable_master_detection=1
```

2.2.5.2 Slave-specific PFE configuration

At startup, Slave instance doesn't configure PFE in any way, so the traffic is delivered only to the Master.

Data traffic distribution for Slave must be configured via the FCI API, using one of accelerated features like PFE L2 bridge or Flexible router.

2.2.5.3 Master-Slave use case example: Shared EMACs with Flexible router

For simplicity in illustrating this use case, a single running Linux kernel instance is used. The Linux kernel network namespaces provide minimal isolation between the Master and Slave instances.

1. Load both drivers:

```
insmod pfeng.ko
insmod pfeng-slave.ko disable_netlink=1
```

2. Update PFE to support Slave netdevice pfe0sl (EMAC Flexible router op mode):

```
libfci_cli phyif-update -i emac0 --mode FLEXIBLE_ROUTER --E
libfci_cli logif-add -i emac0msbc --parent=emac0
libfci_cli logif-update -i emac0msbc -E --egres=hif0,hif3 --mr=DMAC --
dmac=ff:ff:ff:ff:ff:ff
```

3. Move Slave's netdevice to a separate namespace:

```
ip netns add slave
ip link set pfe0sl netns slave
```

4. Start Master's netdevice:

```
ifconfig pfe0 192.168.1.1
```

5. Start Slave's netdevice:

```
ip netns exec slave ifconfig pfe0sl 192.168.2.1
```

At this point, external traffic via EMAC is enabled for the Slave interface. All commands targeting the Slave's network interface (i.e. ifconfig, iperf3) must be prefixed by `ip netns exec slave`.

2.3 IEEE1588 Support

PFE is capable of providing precise, adjustable time reference with ability to timestamp ingress and egress PTP frames. Once enabled the driver ensures that ingress and egress PTP frames are timestamped at MAC level and provides timestamp to the stack.

Supported synchronization modes:

- End to End - SYNC, Follow_Up, Delay_Req, Delay_Resp
  - For UDP supported only on IP: 224.0.1.129, 224.0.1.130, 224.0.1.131, 224.0.1.132
- Peer to Peer - Pdelay_Req, Pdelay_Resp, Pdelay_Resp_Follow_Up
  - For UDP supported only on IP: 224.0.0.107

Transport layers:

- Ethernet - Supported
- UDP over IPv4 - Supported
- UDP over IPV6 - Supported

2.3.1 Clock configuration

The S32G platform allows multiple clock configurations, as depicted in Figure 7.

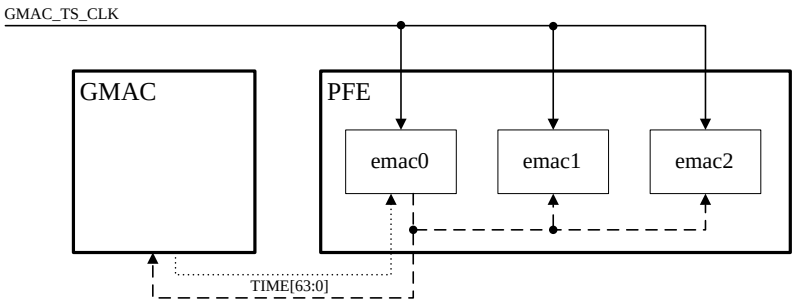

Figure 7. S32G2 IEEE1588 clock distribution

2.3.1.1 Internal Timestamp Mode

See Figure 7, full line.

In this mode the PFE_EMAC is maintaining system time using internal timer clocked by reference signal and uses the timer value to timestamp frames or report current system time. GMAC_TS_CLK is used as reference. This is the only supported configuration. For this to work you have to add "pfe_ts" clock to the device tree.

Example (from *s32g-pfe.dtsi*):

```
clocks = <&clks S32G_SCMI_CLK_PFE_AXI>,
     <&clks S32G_SCMI_CLK_PFE_PE>,
     <&clks S32G_SCMI_CLK_PFE_TS>;
clock-names = "pfe_sys", "pfe_pe", "pfe_ts";
```

Note: *In case the "pfe_ts" clock is not provided, the timestamping feature is disabled.*

In Internal Timestamp Mode the `GMAC_TS_CLK` (RM clock name) resp. `S32G_SCMI_CLK_PFE_TS` (SCMI clock name) should be configured with respect to recommended frequency:

Mode	Minimum Frequency
10 Mbps Full Duplex	5 MHz
10 Mbps Half Duplex	5 MHz
100 Mbps Full Duplex	5 MHz
100 Mbps Half Duplex	5 MHz
1 Gbps Full Duplex	5 MHz
1 Gbps Half Duplex	18.75 MHz
2.5 Gbps Full Duplex	11.72 MHz
2.5 Gbps Half Duplex	46.875 MHz

2.3.1.2 External Timestamp Mode

The S32G platform supports only two ways to share the time with all MACs.

1. EMAC0 is the time source and feeds GMAC, EMAC1 and EMAC2. See Figure 7, dashed line.
2. Dotted line on Figure 7 represents a limited sharing. GMAC is the time source and feeds EMAC0. EMAC1 and EMAC2 cannot use external clock in this case.

Device tree determines configuration of external time sources in `nxp,pfeng-emac-ts-ext-modes` property of PFE device node. The listed `PFE_PHYIF_EMAC_` are configured to use external time source. For example, Figure 7 dotted line, following setting represent the limited time sharing:

```
pfe: pfe@46000000 {
        ...
        nxp,pfeng-emac-ts-ext-modes = <PFE_PHYIF_EMAC_0>;
        ...
};
```

2.3.2 Driver interface

Driver implements standard Linux API for HW timestamping configuration.

2.3.2.1 IOCTL

The driver supports standard ioctl for timestamp configuration. This API is typically called by the PTP stack:

- SIOCSHWTSTAMP - Set configuration (struct `hwtstamp_config`)
  - TX configuration: Timestamping on/off switch available.
  - RX configuration: Message specific configuration is not supported, all packets are timestamped.
- SIOCGHWTSTAMP - Get configuration (struct `hwtstamp_config`)

2.3.2.2 PTP hardware clock device

After driver initialization there should be one device `/dev/ptpX` for each EMAC. Association of clock and interface is available in ethtool as "PTP Hardware Clock" number.

2.3.2.3 ethtool

Driver implements standard ethtool API for getting timestamp options via `ETHTOOL_GET_TS_INFO`.

Example output:

```
# ethtool -T pfe0
Time stamping parameters for pfe0:
Capabilities:
        hardware-transmit
        software-transmit
        hardware-receive
        software-receive
        software-system-clock
        hardware-raw-clock
PTP Hardware Clock: 0
Hardware Transmit Timestamp Modes:
        off
        on
Hardware Receive Filter Modes:
        none
        all
```

2.3.3 BSP yocto support

Hint: add package '**linuxptp**' to your Yocto configuration file. With this configuration you get access to Linux PTP stack **ptp4l** and clock synchronization demon "phc2sys" for advanced configurations.

```
IMAGE_INSTALL:append = " linuxptp"
```

2.4 Power Management

2.4.1 Prerequisites

- Linux kernel with PM support (it's default)
- Arm Trusted Firmware

2.4.1.1 Compile time

- Linux kernel source configured with `CONFIG_PM` and `CONFIG_SUSPEND`

2.4.1.2 Run time

- Linux kernel with support for Suspend to RAM (STR) together with kernel reset controller driver for S32G.
  Please read more about STR in *Linux BSP User Manual* [3], chapter *Power Management*.
- Hardware capable of doing suspend.

Note: *For hardware reasons, the power management features are supported only on S32GVNP-RDB2 boards.*

2.4.2 Suspend to RAM

The driver implements *.suspend()* and *.resume()* callbacks, which are used by system to manage power states.

2.4.2.1 Suspend

During suspend, the driver stops all managed components in the following order:

- Network interfaces (pfe0, pfe1 ...)
- PFE EMACs and its clocks (RX/TX)
- PFE HIF channels
- PFE Platform
- PFE clocks

2.4.2.2 Resume

On resume, first the S32G PFE partition reset is executed, then components get resumed in reverse order of suspension (see Section 2.4.2.1).

Note: *The driver does not preserve any configuration set by FCI API. In case of additional post-startup changes made by FCI API, it is user's reponsibility to restore them after return from suspend.*

2.5 Frame size and scatter-gather

The maximum supported L2 frame size for the standalone and the PFE Master drivers is 9022B (Jumbo Frames support enabled). For the PFE Slave driver the maximum supported L2 frame size is currently 1522B. Jumbo frame support in the PFE Linux drivers is conditional on the `jumbo_frames` FW feature flag being activated.

On ingress, when Jumbo Frames support is enabled, frames exceeding 2K bytes in size are received in multiple memory buffers (i.e. scattered). Frames exceeding the maximum supported L2 size are discarded at MAC level.

On egress, the maximum supported MTU is 9004B for Jumbo Frames and 1504B for standard frames. A 4 byte tag is needed, in addition to the normal Ethernet frame, by the PFE Linux network interfaces that act as DSA Masters for the SJA1105 switch.

The MTU is configurable in the range of 64B to 1504B/ 9004B (Jumbo Frames) with the following command:

`ip link set <pfeX> mtu <value>`

The driver supports transmission of packets that are fragmented into multiple consecutive buffers. This feature is called Tx scatter-gather (Tx SG) and optimizes the transmission of certain packet types and sizes (i.e. TCP) by reducing the number of memory buffer copies in the Linux network stack. Tx SG is enabled by default for every PFE driver interface, e.g.:

```
# ethtool -k pfe0 | grep scatter-gather
scatter-gather: on
        tx-scatter-gather: on
        tx-scatter-gather-fraglist: off [fixed]
```

Tx SG can be optionally turned off (for troubleshooting purposes) with the following ethtool command:

`ethtool -K <pfeX> sg [on|off]`

2.6 Checksum Offload

The driver supports offloading of packet checksum computation to PFE, on both reception (Rx) and transmission (Tx) paths to and from the Host.

On Rx, the driver notifies the Linux network stack that the device can compute and verify TCP and UDP checksums over IPv4 and IPv6, for the first level of protocol encapsulation of a packet.

On Tx, the driver offloads checksum insertion for plain (un-encapsulated) TCP and UDP over IPv4/IPv6 packets.

The Rx and Tx checksumming offload feature is enabled by default in the driver, and its status can be confirmed with the following command:

```
# ethtool -k pfe2 | grep checksumming
rx-checksumming: on
tx-checksumming: on
```

To disable / re-enable the Rx and/or Tx checksumming offload use the following ethtool command:

`ethtool -K <pfeX> rx [on|off] tx [on|off]`

Example - disabling both Rx and Tx checksumming offloads:

```
# ethtool -K pfe2 rx off tx off
Actual changes:
tx-checksum-ipv4: off
tx-checksum-ipv6: off
rx-checksum: off
```

2.6.1 Known limitations

PFE supports Tx checksum offloading for the following protocol layers.

Table 2. Supported Tx checksum offload protocols

Supported protocols	Supported frame header combinations
Plain TCP/UDP over IPv4 and IPv6	*ETH+IPV4+TCP* *ETH+IPV4+UDP* *ETH+IPV6+TCP* *ETH+IPV6+UDP*
TCP/UDP over IPv4/IPv6 with options/ extensions	*ETH+IPV4_OPTIONS+TCP* *ETH+IPV4_OPTIONS+UDP* *ETH+IPV6_EXT+TCP* *ETH+IPV6_EXT+UDP*
IEEE 802.1Q VLAN tagged (single tag) TCP/UDP over IPv4 and IPv6	*ETH+IPV4+TCP with single tag with TPID 8100* *ETH+IPV4+UDP with single tag with TPID 8100* *ETH+IPV6+TCP with single tag with TPID 8100* *ETH+IPV6+UDP with single tag with TPID 8100*

For any other TCP/UDP over IPv4/IPv6 protocol layering and options, Tx checksumming offload should be disabled (see ethool command above for runtime configuration).

2.7 Flow Control

The PFE driver supports MAC level flow control for transmission and reception based on IEEE 802.3x Pause packets.

When Tx Flow Control is enabled and packet reception is congested, the MAC starts transmitting Pause frames in full-duplex mode, or makes back pressure in half-duplex mode, to prevent packet loss by controlling the flow of packets from the remote sender.

When Rx Flow Control is enabled, the MAC receiver detects a Pause frame in full-duplex mode only and responds by stopping the MAC transmitter.

Currently, only Rx Flow Control is enabled by default in the PFE driver. Autonegotiation of flow control parameters between link partners is not supported.

To check the Flow Control status of a PFE network interface use `ethtool -a`, i.e.:

```
# ethtool -a pfe2
Pause parameters for pfe2:
Autonegotiate:  off
```

```
RX:             on
TX:             off
```

To disable / enable the Rx and/or Tx Flow Control use the following `ethtool` command:

`ethtool -A <pfeX> rx [on|off] tx [on|off]`

Example - enabling Tx flow control:

```
# ethtool -A pfe2 tx on
```

2.8 Lossless Tx from Host

Under certain conditions, the traffic originated by a PFE Linux Network Interface bound to an EMAC port may be lossy. By checking relevant statistics from the driver debug files, the cause of the packet loss may be traced to tail drop events at the EMAC facing TMU queue that forwards the transmitted packets from the Host. If that's the case, then the TMU queue drops can be prevented by enabling the `lltx_res_tmu_q_id` PFE driver module option:

```
lltx_res_tmu_q_id:    Reserved TMU queue ID for Host lossless Tx (LLTX),
                      range: 0-7; use 255 to disable LLTX (default: 255)
```

Assuming the traffic is dropped by TMU queue 0 - the default TMU queue for traffic originating from Host, then the Lossless Tx feature for that queue can be enabled as follows:

Example - enabling Lossless Tx for TMU queue 0:

```
# insmod pfeng.ko lltx_res_tmu_q_id=0
# dmesg | grep LLTX
pfeng 46000000.pfe pfe0 : Host LLTX enabled for TMU PHY_ID#0/ Q_ID#0
pfeng 46000000.pfe pfe1 : Host LLTX enabled for TMU PHY_ID#1/ Q_ID#0
pfeng 46000000.pfe pfe2 : Host LLTX enabled for TMU PHY_ID#2/ Q_ID#0
pfeng 46000000.pfe aux0 : Host LLTX disabled
```

When this feature is enabled, the PFE driver executes a (software) back pressure mechanism that controls packet transmission at the Linux Host level, at the cost of extra Tx packet processing cycles.

2.8.1 Known limitations

The Lossless Tx feature is supported only by the PFE Master and Standalone drivers. The PFE Slave driver does not support this feature.

Lossless Tx works for EMAC bound Linux Network Interfaces. The AUX interface or FCI Bridge mode interfaces are not supported.

If the TMU queue designated for Lossless Tx is forwarding traffic coming from other sources (e.g. fast path traffic) than the Network Interface owning the associated EMAC, then zero packet loss for that queue can no longer be guaranteed.

Currently, the Lossless Tx TMU queue ID is configurable at driver level (and not at Network Interface level).

2.9 Reserved Memory Regions

The PFE accesses various data structures stored in the Host memory. The location of some of these data structures is configurable to allow the integrator to place them in such a way as to either optimize performance (place some data structures in faster memory) and/or tune the application from security/safety perspective (place some data structures into dedicated containers with restricted access permissions).

The following reserved memory regions are currently defined for the Linux PFE driver:

- System Buffers exclusive region
- Buffer Descriptor Rings exclusive region
- DMA shared pool region

2.9.1 System Buffers exclusive regions

These are the memory regions where the Routing Table (RT) entries and the Ethernet frames (BMU2) are stored while they are being processed by PFE. These regions are defined in the device tree by dedicated `'reserved-memory'` sub-nodes with the following compatibility strings:

- `'nxp,s32g-pfe-bmu2-pool'`, for the BMU2 buffer pool;
- `'nxp,s32g-pfe-rt-pool'`, for the Routing Table entries.

The physical start address, alignment and size of these regions must meet some restrictions imposed by the corresponding PFE hardware blocks:

- physical address range: `0x20000 - 0xbfffffff`;
- the alignment must equal the size of the region for BMU2;
- the memory region size must be large enough to accommodate:
  - for BMU2: `PFE_CFG_BMU2_BUF_COUNT` buffers of the preconfigured size of `PFE_CFG_BMU2_BUF_SIZE`;
  - for RT: `PFE_CFG_RT_HASH_SIZE + PFE_CFG_RT_COLLISION_SIZE` Routing Table entries, refer to the *PFE FW User Manual*[2] (see pfe_ct.h) for the exact RT entry size.

The allocation of the PFE system buffers is also controlled by driver compile time configuration switches, according to the following table:

Table 3. Driver compile time switches for placing PFE system buffers into reserved memory regions

PFE buffer pool	Compile time option	Option value for the exclusive mem region	Option value for the "pfe-shared-pool" region
BMU2	`PFE_CFG_SYS_MEM`	`"pfe-bmu2-pool"` (default)	`"pfe_ddr"`
RT	`PFE_CFG_RT_MEM`	`"pfe-rt-pool"` (default)	`"pfe_ddr"`

For optimal memory usage and flexibility it is recommended to preserve the default values of these compile time options.

Placing the PFE System Buffers in SRAM or DDR

For improved fast-path performance, there is the option to place the Routing Table entries and/or the BMU2 pool in the internal SRAM memory, which starts at the physical address `0x34000000` on the S32G SoCs . For this to work, the corresponding device

tree node must point to a valid SRAM address range that is not reserved for other devices.

Configuration example for placing the BMU2 pool in SRAM:

```
pfe_reserved_bmu2: pfebufs@34000000 {
    compatible = "nxp,s32g-pfe-bmu2-pool";
    /* BMU2: 512 KB */
    reg = <0 0x34000000 0 0x80000>;
    no-map;
    status = "okay";
};
```

Similarly, to place the buffers in DDR, provide a valid physical address range in DDR (via the `reg` node property), and ensure that it does no overlap with other reserved memory regions.

Placing the PFE System Buffers in exclusive reserved memory regions is optional, although recommended. In case any of the dedicated reserved memory nodes is missing from the device tree, the PFE driver will attempt to allocate the corresponding buffers in the common `"pfe-shared-pool"` region.

2.9.2 Buffer Descriptor Rings exclusive region

This is a cache enabled, DMA coherent, memory region reserved for performance critical data structures. It is intended mainly for the Rx and Tx buffer descriptor rings. The memory region is defined in the device tree by a dedicated `'reserved-memory'` sub-node with the `'nxp,s32g-pfe-bdr-pool'` compatibility string. The memory region is cache coherent, and meets the DMA address range restrictions of the PFE hardware. Allocation in this region is also controlled by the `PFE_CFG_BDR_MEM` compile time configuration option, which is set by default to `'pfe-bdr-pool'`. In case this reserved memory node is missing from the device tree, or if `PFE_CFG_BDR_MEM` is set to `'pfe_ddr'`, the PFE driver will allocate the buffer descriptor rings inside the common, non-cacheable, `"pfe-shared-pool"` DMA region, incurring throughput degradation for the data traffic going through the Host.

2.9.3 DMA shared pool region

This is the common memory region reserved for the remaining DMA data structures used by PFE, that have not been accommodated by dedicated reserved memory regions. Identified in the device tree by the `'pfe-shared-pool'` name, it is a `'shared-dma-pool'` compatible memory region managed by the Linux kernel system code, and all DMA-API allocations are performed from here. Linux maps this zone as non-cacheable. Buffer pools that have the corresponding `PFE_CFG_*_MEM` compile time configuration options set to `'pfe_ddr'` are allocated here.

2.10 FCI use cases for Linux

Additional features provided by the PFE accelerator can be managed via the FCI API. The driver release package contains library sources that can be used to control PFE from a custom user application. Details about LibFCI library usage can be found in the *FCI API Reference* [1].

The driver release package also provides an example command line tool: `libfci_cli`. The tool is a demo application which demonstrates how to use the LibFCI library to manage fast-path accelerated features in PFE.

`libfci_cli` is also included in the root file system delivered with the Linux BSP.

For details on how to build LibFCI and `libfci_cli` refer to Section 3.2.

2.10.1 Fast path bridging use case example

To configure a simple L2 bridge with LibFCI in Linux follow these steps:

1. Bring up all the Linux interfaces required for bridge membership (set up the physical layer link):

```
# for eth in pfe0 pfe1 pfe2; do ip link set $eth up; done
```

   a. For S32G2 EVB, refer to Section 4.3.1 for how to set up the SJA1105 Linux DSA switch driver interfaces to disable DSA tagging.

2. Configure the hit/miss actions for the default bridge domain (identified by LibFCI as VLAN 1):

```
# libfci_cli bd-update --vlan 1 --ucast-hit FORWARD --ucast-miss FLOOD --
mcast-hit FORWARD --mcast-miss FLOOD
```

3. Add the EMAC interfaces of pfe<0-2> as members of the default bridge domain with VLAN tagging disabled:

```
# libfci_cli bd-insif --vlan 1 --i emac0 --tag OFF
# libfci_cli bd-insif --vlan 1 --i emac1 --tag OFF
# libfci_cli bd-insif --vlan 1 --i emac2 --tag OFF
```

4. Configure the EMAC interfaces according to the physical interface FCI API for bridged interfaces:
   - "VLAN_BRIDGE" mode
   - enable traffic promiscuity
   - blocking state mode "NORMAL"

```
# libfci_cli phyif-update --i emac0 -E --promisc ON --mode VLAN_BRIDGE --bs
 NORMAL
# libfci_cli phyif-update --i emac1 -E --promisc ON --mode VLAN_BRIDGE --bs
 NORMAL
# libfci_cli phyif-update --i emac2 -E --promisc ON --mode VLAN_BRIDGE --bs
 NORMAL
```

5. Check bridge configuration:

```
# libfci_cli bd-print
domain 01  [default]
    phyifs (tagged)   : ---
    phyifs (untagged) : emac0,emac1,emac2
    ucast-hit  action : 0 (FORWARD)
    ucast-miss action : 1 (FLOOD)
    mcast-hit  action : 0 (FORWARD)
    mcast-miss action : 1 (FLOOD)
domain 00  [fallback]
    phyifs (tagged)   : ---
    phyifs (untagged) : ---
    ucast-hit  action : 3 (DISCARD)
    ucast-miss action : 3 (DISCARD)
    mcast-hit  action : 3 (DISCARD)
    mcast-miss action : 3 (DISCARD)
Command successfully executed.
```

6. Check the settings of individual EMAC interfaces, i.e.:

```
# libfci_cli phyif-print -i emac0
 0: emac0
      <ENABLED>
      <promisc:ON, mode:VLAN_BRIDGE, block-state:NORMAL>
      <vlan-conf:OFF, ptp-conf:OFF, ptp-promisc:OFF, q-in-q:ON>
      <discard-if-ttl-below-2:OFF>
      ingress: 0 egress: 234 discarded: 0 malformed: 0
      MAC:
          00:01:be:be:ef:11
```

```
          33:33:00:00:00:01
          01:00:5e:00:00:01
          33:33:ff:be:ef:11
      mirrors:
          rxmirr0: ---
          rxmirr1: ---
          txmirr0: ---
          txmirr1: ---
      logical interfaces:
       0: pfe0
            <ENABLED>
            <promisc:ON, match-mode:AND, discard-on-match:OFF, loopback:OFF>
            accepted: 0 rejected: 0 discarded: 0 processed: 0
            egress: hif0
            match-rules: ---
Command successfully executed.
```

Refer to the *FCI API Reference* [1] for details on LibFCI configurations.

2.10.2 Ingress QoS use case examples

Ingress QoS (Quality of Service) is a configurable PFE hardware feature that allows congestion avoidance and congestion control at the physical ingress ports, before the surplus traffic can clog system resources. Ingress QoS implements the following traffic policing mechanisms, in this order:

- Classification of ingress traffic by the following priority classes (increasing from left to right): Unmanaged, Managed and Reserved;
- Weighted Random Early Drop (WRED);
- Ingress port level rate shaping.

After the traffic classification stage, the WRED policy defines four probability zones for dropping "Unmanaged" traffic at increasing rates based on the queue fill level (between the "min" and "Max" levels), and one zone for dropping all traffic except the flows classified as Reserved (between the "Max" and "FULL" levels), according to the example in the following figure:

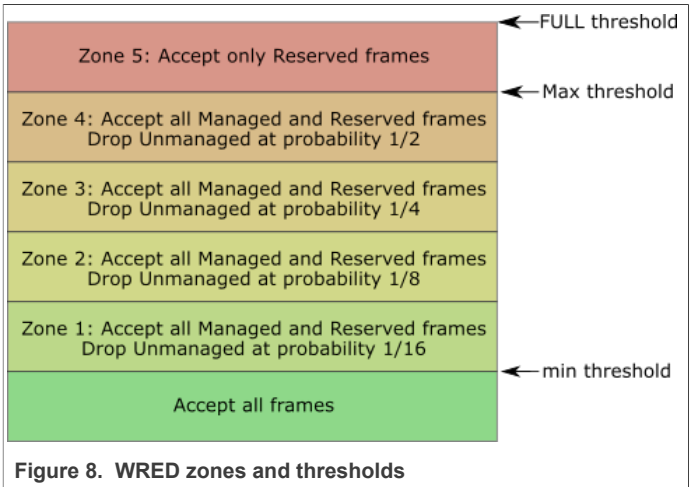

Figure 8. WRED zones and thresholds

The final policy stage is controlled by the ingress port shaper. The shaper parameters are defined as follows according to the IEEE 802.1Q standard specification for the data rate Credit Based Shaper (CBS):

```
hiCredit = maxInterferenceSize x (idleSlope/portTransmitRate)
```

where,

- `idleSlope` (in bits per second): the only input parameter, determines the maximum fraction of the `portTransmitRate` that is available to the queue (`bandwidthFraction`), as follows `bandwidthFraction = idleSlope/portTransmitRate`;
- `maxInterferenceSize` (in bits): maximum size in bits of any burst of traffic that can delay the transmission of a frame that is available for transmission for this traffic class. In our case, since there is one traffic class for the shaper (it's per port), `maxInterferenceSize` is exactly the maximum frame size supported by the underlying MAC (e.g. around 1500 x 8 bits);
- and `loCredit = maxFrameSize x (sendSlope/portTransmitRate)` where, `sendSlope = (idleSlope – portTransmitRate)`.

Rate shaper configuration example for *pfe0*

1. Enable the Ingress QoS module (resets hardware to default configuration):

```
# libfci_cli qos-pol-set -i emac0 -E
```

2. Configure the EMAC0 port level IEEE 802.1Q data rate Credit Based Shaper (CBS):
   - set *idleSlope* to 100 Mbps
   - set *hiCredit* to 1216 bits
   - set *loCredit* to -10944 bits

```
# libfci_cli  qos-pol-shp-update -i emac0 -E --shp=0 --shp-mode=DATA_RATE --
shp-type=PORT --isl=100000000 --crmin=-10944 --crmax=1216
```

3. Check the configuration:

```
# libfci_cli  qos-pol-shp-print -i emac0
shaper 0:
    <ENABLED>
    interface:  emac0
    shp-type:   0 (PORT)
    shp-mode:   0 (DATA_RATE)
    isl:        99975585
    credit-min: -10944
    credit-max: 1216
[...]
```

4. Configure pfe0 and bring it up, i.e. `ip link set pfe0 up`.

WRED configuration example for *pfe0*

1. Enable the Ingress QoS module (resets hardware to default configuration):

```
# libfci_cli qos-pol-set -i emac0 -E
```

2. Configure WRED policy for EMAC0's DMEM buffer pool:
   - Min, Max and Full thresholds are set to 256, 1008, resp. 1024, representing buffer depletion levels of the DMEM pool;
   - Configure drop probabilities for unmanaged traffic for the four drop zones in percentages, e.g.: 7%, 14%, 20%, resp. 50%;

```
# libfci_cli qos-pol-wred-update -i emac0 --wred-que=DMEM -E --thmin 256 --
thmax 1008 --thfull 1024 --zprob=7,14,20,50
```

3. Check the actual programmed values for the probability zones since they are subject to rounding errors - hardware values are in increments of 1/16:

```
# libfci_cli qos-pol-wred-print -i emac0
Wred for 'DMEM' ingress queue:
    <ENABLED>
    interface: emac0
    thld-min:  256
    thld-max:  1008
```

```
    thld-full: 1024
    zprob:     [0]<6>,[1]<13>,[2]<20>,[3]<46>
[...]
```

Configure an IPv4 flow as Reserved, based on source IP address, for pfe0:

1. Enable Ingress QoS (see above);
2. Add the flow specification - source IPv4 IP address 172.16.0.1/24:

```
# libfci_cli  qos-pol-flow-add -i emac0 --flowact RESERVED -s 172.16.0.1 --
s-pfx 24 --ft TYPE_ETH,TYPE_IP4,SIP
```

3. Check the hardware Classification table for pfe0:

```
# libfci_cli  qos-pol-flow-print -i emac0
flow 0:
    interface:   emac0
    flow-action: Mark traffic as RESERVED
    argumentless flow-types: 0x0009 (TYPE_ETH,TYPE_IP4)
    argumentful  flow-types:
      VLAN:      <vlan: 0> ; <vlan-mask: 0x0000>
      TOS:       <tos: 0x00> ; <tos-mask: 0x00>
      PROTOCOL:  <protocol: 0 (HOPOPT)> ; <protocol-mask: 0x00>
      SIP:       <sip: 172.16.0.1> ; <sip-pfx: 24>
      DIP:       <dip: 0.0.0.0> ; <sip-pfx: 0>
      SPORT:     <sport-min: 0> ; <sport-max: 0>
      DPORT:     <dport-min: 0> ; <dport-max: 0>
Command successfully executed.
```

2.10.3 QoS known limitations

1. **Egress QoS Scheduler configuration is not fully dynamic**

   affected FCI commands: `FPP_CMD_QOS_SCHEDULER + FPP_ACTION_UPDATE`
   affected CLI commands: `qos-sch-update`

   Description: It is not safe to configure properties of Egress QoS Scheduler once any sort of traffic is processed by PFE. An attempt to configure properties of Egress QoS Scheduler once any traffic has passed through PFE can lead to temporary malfunction of Egress QoS Scheduler.
   Workaround: Configure Egress QoS Schedulers immediately after driver module load and avoid doing traffic over EMACs before that.

2. **Ingress QoS configuration not dynamic**

   affected FCI command: `FPP_CMD_QOS_POLICER + FPP_ACTION_UPDATE`
   affected CLI command: `qos-pol-set`

   Description: Traffic is sometimes blocked on reception at EMAC level when the Ingress QoS (IQOS) module is enabled, and resumes after disabling the IQOS module. The issue is triggered in the following cases: when IQOS is re-enabled multiple times (same initial PFE driver load), or when IQOS is enabled after the PFE driver had significant traffic over the corresponding EMAC. The issue is caused by the fact that 'dynamic' configuration of the Ingress QoS module is not reliable.
   Workaround: Enable the Ingress QoS module once immediately after driver module load, and before having traffic over the corresponding EMAC.

3. **WRED "full" threshold not configurable**

   affected FCI command: `FPP_CMD_QOS_POLICER_WRED + FPP_ACTION_UPDATE`
   affected CLI command: `qos-pol-wred-update`, parameter: `thld-full`

   Description: The WRED "full" threshold (i.e. `thld-full`) is not configurable. The threshold at which all the ingress traffic is dropped is given by the actual queue size (i.e. DMEM or LMEM). Configuring lower values for `thld-full` than the ingress queues sizes has no effect.

Workaround: Ignore the `thld-full` parameter, and WRED it will assume by default the maximum possible size of the associated ingress queue.

4. **L4 Port range based classification not functional**

   affected FCI command: `FPP_CMD_QOS_POLICER_FLOW + FPP_ACTION_REGISTER`
   affected CLI command: `qos-pol-flow-add`, parameters: `sport-min`, `sport-max`, `dport-min`, and `dport-max`

   Description: Traffic that matches a L4 Source Port or Destination Port range based classification rule ends up classified as "Unmanaged" (lowest priority) regardless of the requested flow priority.
   Workaround: L4 Port based classification should not be used (there's no workaround for this feature to date).

2.10.4 libfci_cli demo application

Overview

Libfci_cli demo application is a command-line interface (CLI) tool which demonstrates how to use the LibFCI library (FCI commands) to manage fast-path accelerated features in PFE.

The application accepts its own specific CLI commands. It converts them to corresponding FCI commands (by using appropriate elements of the FCI API) and sends the FCI commands to PFE driver.

This demo application showcases configuration possibilities of PFE and can be used for quick assessments. However, it should not be used in production setups. For production, use directly the LibFCI library (which provides FCI commands) to develop a custom configuration code.

Usage of the tool

1. Standard use:

```
libfci_cli <CLI command> [options for the CLI command]

examples:
libfci_cli phyif-print
libfci_cli phyif-update  --interface emac0  --enable
```

   - CLI commands are not prefixed by anything.
   - Options for CLI commands are prefixed by leading dashes (`-` or `--`).

2. To print a list of all available CLI commands:
   (do not specify any CLI commands nor command options)

```
libfci_cli
```

3. To print detailed help for a particular CLI command:

```
libfci_cli <CLI command> --help
```

4. To print version information of the tool:

```
libfci_cli --version
```

Daemon

Libfci_cli demo application can start its own daemon (more precisely a dedicated background process). This feature demonstrates handling of FCI events[1]. Libfci_cli

1 Unsolicited FCI messages sent by PFE driver to userspace applications.

daemon should be used only for assessment of scenarios where some FCI events are of interest. It is not needed for assessment of standard configuation/printout scenarios.

CLI commands for daemon control (see their `-h` help for details):

- `daemon-print`[2]
- `daemon-update`
- `daemon-start`
- `daemon-stop`

Daemon features (togglable via `daemon-update`):

- Can print received FCI events into terminal. Printing is done automatically, immediately after the FCI event is received.
- Can print received FCI events into logfile. Printing is done automatically, immediately after the FCI event is received. Logfile is located in directory of the originating libfci_cli demo application.

2.10.5 libfci_cli command mapping

The following table shows which FCI commands (and actions) are used by CLI commands of the libfci_cli demo application.

Refer to the *FCI API Reference* [1] for details about FCI commands.

Table 4. libfci_cli command mapping

CLI command	FCI commands + FCI actions
phyif-print	FPP_CMD_PHY_IF + FPP_ACTION_QUERY FPP_CMD_PHY_IF + FPP_ACTION_QUERY_CONT (iterate through all PFE physical interfaces)
phyif-update	1. FPP_CMD_PHY_IF + FPP_ACTION_QUERY FPP_CMD_PHY_IF + FPP_ACTION_QUERY_CONT (get the target PFE physical interface) 2. FPP_CMD_PHY_IF + FPP_ACTION_UPDATE (update the target PFE physical interface)
phyif-mac-print	FPP_CMD_IF_MAC + FPP_ACTION_QUERY FPP_CMD_IF_MAC + FPP_ACTION_QUERY_CONT (iterate through all MAC addresses of the target PFE physical interface)
phyif-mac-add	FPP_CMD_IF_MAC + FPP_ACTION_REGISTER (add MAC address to the target PFE physical interface)
phyif-mac-del	FPP_CMD_IF_MAC + FPP_ACTION_DEREGISTER (remove MAC address from the target PFE physical interface)
logif-print	FPP_CMD_LOG_IF + FPP_ACTION_QUERY FPP_CMD_LOG_IF + FPP_ACTION_QUERY_CONT (iterate through all PFE logical interfaces)
logif-update	1. FPP_CMD_LOG_IF + FPP_ACTION_QUERY FPP_CMD_LOG_IF + FPP_ACTION_QUERY_CONT (get the target PFE logical interface) 2. FPP_CMD_LOG_IF + FPP_ACTION_UPDATE (update the target PFE logical interface)
logif-add	FPP_CMD_LOG_IF + FPP_ACTION_REGISTER (create a logical interface in PFE and associate it with a PFE physical interface)

2 Daemon-print displays configuration of the daemon. For printing of FCI events, see Daemon features.

Table 4. libfci_cli command mapping*...continued*

CLI command	FCI commands + FCI actions
logif-del	FPP_CMD_LOG_IF + FPP_ACTION_DEREGISTER (destroy the target logical interface in PFE)
mirror-print	FPP_CMD_MIRROR + FPP_ACTION_QUERY FPP_CMD_MIRROR + FPP_ACTION_QUERY_CONT (iterate through all mirroring rules in PFE)
mirror-update	1. FPP_CMD_MIRROR + FPP_ACTION_QUERY FPP_CMD_MIRROR + FPP_ACTION_QUERY_CONT (get the target mirroring rule from PFE) 2. FPP_CMD_MIRROR + FPP_ACTION_UPDATE (update the target mirroring rule in PFE)
mirror-add	FPP_CMD_MIRROR + FPP_ACTION_REGISTER (create a mirroring rule in PFE)
mirror-del	FPP_CMD_MIRROR + FPP_ACTION_DEREGISTER (destroy the target mirroring rule in PFE)
bd-print	FPP_CMD_L2_BD + FPP_ACTION_QUERY FPP_CMD_L2_BD + FPP_ACTION_QUERY_CONT (iterate through all bridge domains in PFE)
bd-update	1. FPP_CMD_L2_BD + FPP_ACTION_QUERY FPP_CMD_L2_BD + FPP_ACTION_QUERY_CONT (get the target bridge domain from PFE) 2. FPP_CMD_L2_BD + FPP_ACTION_UPDATE (update the target bridge domain in PFE)
bd-add	FPP_CMD_L2_BD + FPP_ACTION_REGISTER (create a bridge domain in PFE)
bd-del	FPP_CMD_L2_BD + FPP_ACTION_DEREGISTER (destroy the target bridge domain in PFE)
bd-insif	1. FPP_CMD_L2_BD + FPP_ACTION_QUERY FPP_CMD_L2_BD + FPP_ACTION_QUERY_CONT (get the target bridge domain from PFE) 2. FPP_CMD_L2_BD + FPP_ACTION_UPDATE (update interface list of the target bridge domain in PFE: set bitflags for PFE interfaces newly added to the domain)
bd-remif	1. FPP_CMD_L2_BD + FPP_ACTION_QUERY FPP_CMD_L2_BD + FPP_ACTION_QUERY_CONT (get the target bridge domain in PFE) 2. FPP_CMD_L2_BD + FPP_ACTION_UPDATE (update interface list of the target bridge domain in PFE: clear bitflags for PFE interfaces newly removed from the domain)
bd-flush	FPP_CMD_L2_FLUSH_STATIC FPP_CMD_L2_FLUSH_LEARNED FPP_CMD_L2_FLUSH_ALL (flush the MAC table in PFE)
bd-stent-print	FPP_CMD_L2_STATIC_ENT + FPP_ACTION_QUERY FPP_CMD_L2_STATIC_ENT + FPP_ACTION_QUERY_CONT (iterate through all static entries in the MAC table in PFE)
bd-stent-update	1. FPP_CMD_L2_STATIC_ENT + FPP_ACTION_QUERY FPP_CMD_L2_STATIC_ENT + FPP_ACTION_QUERY_CONT (get the target static entry from the MAC table in PFE) 2. FPP_CMD_L2_STATIC_ENT + FPP_ACTION_UPDATE (update the target static entry in the MAC table in PFE)

Table 4. libfci_cli command mapping...*continued*

CLI command	FCI commands + FCI actions
bd-stent-add	FPP_CMD_L2_STATIC_ENT + FPP_ACTION_REGISTER (create a static entry in the MAC table in PFE)
bd-stent-del	FPP_CMD_L2_STATIC_ENT + FPP_ACTION_DEREGISTER (delete the target static entry from the MAC table in PFE)
fptable-print	FPP_CMD_FP_TABLE + FPP_ACTION_QUERY FPP_CMD_FP_TABLE + FPP_ACTION_QUERY_CONT (iterate through all rules of the target FP table in PFE)
fptable-add	FPP_CMD_FP_TABLE + FPP_ACTION_REGISTER (create an FP table in PFE)
fptable-del	FPP_CMD_FP_TABLE + FPP_ACTION_DEREGISTER (destroy the target FP table in PFE)
fptable-insrule	FPP_CMD_FP_TABLE + FPP_ACTION_USE_RULE (insert a target FP rule into the target FP table in PFE)
fptable-remrule	FPP_CMD_FP_TABLE + FPP_ACTION_UNUSE_RULE (remove a target FP rule from the target FP table in PFE)
fprule-print	FPP_CMD_FP_RULE + FPP_ACTION_QUERY FPP_CMD_FP_RULE + FPP_ACTION_QUERY_CONT (iterate through all existing FP rules in PFE)
fprule-add	FPP_CMD_FP_RULE + FPP_ACTION_REGISTER (create an FP rule in PFE)
fprule-del	FPP_CMD_FP_RULE + FPP_ACTION_DEREGISTER (destroy the target FP rule in PFE)
route-print	FPP_CMD_IP_ROUTE + FPP_ACTION_QUERY FPP_CMD_IP_ROUTE + FPP_ACTION_QUERY_CONT (iterate through all existing routes in PFE) ***Note:*** *In context of PFE, the term 'route' has a specific meaning. Refer to the FCI API Reference [1] for details.*
route-add	FPP_CMD_IP_ROUTE + FPP_ACTION_REGISTER (create a route in PFE) ***Note:*** *In context of PFE, the term 'route' has a specific meaning. Refer to the FCI API Reference [1] for details.*
route-del	FPP_CMD_IP_ROUTE + FPP_ACTION_DEREGISTER (destroy the target route in PFE) ***Note:*** *In context of PFE, the term 'route' has a specific meaning. Refer to the FCI API Reference [1] for details.*
cntk-print	**For IPv4 conntracks:** FPP_CMD_IPV4_CONNTRACK + FPP_ACTION_QUERY FPP_CMD_IPV4_CONNTRACK + FPP_ACTION_QUERY_CONT (iterate through all existing IPv4 conntracks in PFE) **For IPv6 conntracks:** FPP_CMD_IPV6_CONNTRACK + FPP_ACTION_QUERY FPP_CMD_IPV6_CONNTRACK + FPP_ACTION_QUERY_CONT (iterate through all existing IPv6 conntracks in PFE) ***Note:*** *In context of PFE, the term 'conntrack' has a specific meaning. Refer to the FCI API Reference [1] for details.*

Table 4. libfci_cli command mapping...*continued*

CLI command	FCI commands + FCI actions
cntk-update	**For IPv4 conntracks:** 1. FPP_CMD_IPV4_CONNTRACK + FPP_ACTION_QUERY FPP_CMD_IPV4_CONNTRACK + FPP_ACTION_QUERY_CONT (get the target IPv4 conntrack from PFE) 2. FPP_CMD_IPV4_CONNTRACK + FPP_ACTION_UPDATE (update the target IPv4 conntrack in PFE) **For IPv6 conntracks:** 1. FPP_CMD_IPV6_CONNTRACK + FPP_ACTION_QUERY FPP_CMD_IPV6_CONNTRACK + FPP_ACTION_QUERY_CONT (get the target IPv6 conntrack from PFE) 2. FPP_CMD_IPV6_CONNTRACK + FPP_ACTION_UPDATE (update the target IPv6 conntrack in PFE) ***Note:*** *In context of PFE, the term 'conntrack' has a specific meaning. Refer to the FCI API Reference [1] for details.*
cntk-add	**For IPv4 conntracks:** FPP_CMD_IPV4_CONNTRACK + FPP_ACTION_REGISTER (create IPv4 conntrack in PFE) **For IPv6 conntracks:** FPP_CMD_IPV6_CONNTRACK + FPP_ACTION_REGISTER (create IPv6 conntrack in PFE) ***Note:*** *In context of PFE, the term 'conntrack' has a specific meaning. Refer to the FCI API Reference [1] for details.*
cntk-del	**For IPv4 conntracks:** FPP_CMD_IPV4_CONNTRACK + FPP_ACTION_DEREGISTER (destroy the target IPv4 conntrack in PFE) **For IPv6 conntracks:** FPP_CMD_IPV6_CONNTRACK + FPP_ACTION_DEREGISTER (destroy the target IPv6 conntrack in PFE) ***Note:*** *In context of PFE, the term 'conntrack' has a specific meaning. Refer to the FCI API Reference [1] for details.*
cntk-timeout	FPP_CMD_IPV4_SET_TIMEOUT (set timeouts for both IPv4/IPv6 conntracks in PFE) ***Note:*** *In context of PFE, the term 'conntrack' has a specific meaning. Refer to the FCI API Reference [1] for details.*
route-and-cntk-reset	**For IPv4 conntracks:** FPP_CMD_IPV4_RESET (flush all IPv4 routes and conntracks in PFE) **For IPv6 conntracks:** FPP_CMD_IPV6_RESET (flush all IPv6 routes and conntracks in PFE) ***Note:*** *In context of PFE, terms 'route' and 'conntrack' have specific meanings. Refer to the FCI API Reference [1] for details.*
fwfeat-print	FPP_CMD_FW_FEATURE + FPP_ACTION_QUERY FPP_CMD_FW_FEATURE + FPP_ACTION_QUERY_CONT (iterate through all available FW features in PFE)
fwfeat-set	FPP_CMD_FW_FEATURE + FPP_ACTION_UPDATE (enable or disable the target FW feature in PFE)
qos-que-print	FPP_CMD_QOS_QUEUE + FPP_ACTION_QUERY (iterate through all Egress QoS queues of the target PFE physical interface)

Table 4. libfci_cli command mapping*...continued*

CLI command	FCI commands + FCI actions
qos-que-update	1. FPP_CMD_QOS_QUEUE + FPP_ACTION_QUERY (get the target Egress QoS queue from PFE) 2. FPP_CMD_QOS_QUEUE + FPP_ACTION_UPDATE (update the target Egress QoS queue in PFE)
qos-sch-print	FPP_CMD_QOS_SCHEDULER + FPP_ACTION_QUERY (iterate through all Egress QoS schedulers of the target PFE physical interface)
qos-sch-update	1. FPP_CMD_QOS_SCHEDULER + FPP_ACTION_QUERY (get the target Egress QoS scheduler from PFE) 2. FPP_CMD_QOS_SCHEDULER + FPP_ACTION_UPDATE (update the target Egress QoS scheduler in PFE)
qos-shp-print	FPP_CMD_QOS_SHAPER + FPP_ACTION_QUERY (iterate through all Egress QoS shapers of the target PFE physical interface)
qos-shp-update	1. FPP_CMD_QOS_SHAPER + FPP_ACTION_QUERY (get the target Egress QoS shaper from PFE) 2. FPP_CMD_QOS_SHAPER + FPP_ACTION_UPDATE (update the target Egress QoS shaper in PFE)
qos-pol-print	FPP_CMD_QOS_POLICER + FPP_ACTION_QUERY (get status of the Ingress QoS policer of the target PFE physical interface) FPP_CMD_QOS_POLICER_WRED + FPP_ACTION_QUERY (iterate through all Ingress QoS WRED queues of the target PFE physical interface) FPP_CMD_QOS_POLICER_SHP + FPP_ACTION_QUERY (iterate through all Ingress QoS shapers of the target PFE physical interface) FPP_CMD_QOS_POLICER_FLOW + FPP_ACTION_QUERY FPP_CMD_QOS_POLICER_FLOW + FPP_ACTION_QUERY_CONT (iterate through all Ingress QoS flows of the target PFE physical interface)
qos-pol-set	1. FPP_CMD_QOS_POLICER + FPP_ACTION_QUERY (get the target Ingress QoS policer from PFE) 2. FPP_CMD_QOS_POLICER + FPP_ACTION_UPDATE (enable or disable the target Ingress QoS policer in PFE)
qos-pol-wred-print	FPP_CMD_QOS_POLICER_WRED + FPP_ACTION_QUERY (iterate through all Ingress QoS WRED queues of the target PFE physical interface)
qos-pol-wred-update	1. FPP_CMD_QOS_POLICER_WRED + FPP_ACTION_QUERY (get the target Ingress QoS WRED queue from PFE) 2. FPP_CMD_QOS_POLICER_WRED + FPP_ACTION_UPDATE (update the target Ingress QoS WRED queue in PFE)
qos-pol-shp-print	FPP_CMD_QOS_POLICER_SHP + FPP_ACTION_QUERY (iterate through all Ingress QoS shapers of the target PFE physical interface)
qos-pol-shp-update	1. FPP_CMD_QOS_POLICER_SHP + FPP_ACTION_QUERY (get the target Ingress QoS shaper from PFE) 2. FPP_CMD_QOS_POLICER_SHP + FPP_ACTION_UPDATE (update the target Ingress QoS shaper in PFE)
qos-pol-flow-print	FPP_CMD_QOS_POLICER_FLOW + FPP_ACTION_QUERY FPP_CMD_QOS_POLICER_FLOW + FPP_ACTION_QUERY_CONT (iterate through all Ingress QoS flows of the target PFE physical interface)
qos-pol-flow-add	FPP_CMD_QOS_POLICER_FLOW + FPP_ACTION_REGISTER (create Ingress QoS flow in PFE)
qos-pol-flow-del	FPP_CMD_QOS_POLICER_FLOW + FPP_ACTION_DEREGISTER (destroy the target Ingress QoS flow in PFE)

Table 4. libfci_cli command mapping*...continued*

CLI command	FCI commands + FCI actions
fci-ownership	1. FPP_CMD_FCI_OWNERSHIP_LOCK (get FCI ownership) 2. FPP_CMD_FCI_OWNERSHIP_UNLOCK (release FCI ownership)

2.10.6 FCI ownership

The FCI ownership feature ensures that at any given time there is only one driver instance in the multi-instance setup which can successfully issue FCI commands. Driver instances (clients) must be allowed to acquire FCI ownership by configuration on the Master instance. Then the Master serializes the execution of FCI commands on behalf of multiple FCI clients.

There can be only one FCI owner at a time. Only the FCI commands issued by the FCI owner can be executed successfully. FCI commands issued by a driver instance which does not have FCI ownership are never executed and return an error code instead.

Driver instance can acquire or release the FCI ownership via the following means:

- **Manually**, by requesting the FCI ownership via dedicated FCI commands (lock/unlock). If the ownership is acquired manually, it has to be manually released as well (responsibility of the requesting driver).
- **Automatically**, by acquiring floating FCI ownership if it is not held at the moment by other client. The floating FCI ownership is granted temporarily for each issued FCI command. Once the execution of the respective FCI command is finished, the FCI ownership is automatically released, so other sender can take FCI ownership

For purposes of FCI ownership, driver instances are uniquely identified by HIF channel, from where individual FCI command requests are originated (later referred as sender). FCI ownership configuration determines which driver instances / senders can be granted FCI ownership.

2.10.6.1 Linux Master instance FCI ownership configuration

It is possible to configure which driver instances in the Master-slave scenario are allowed to take FCI ownership.

FCI authorization configuration data can be passed to **pfeng.ko** module during load time by following parameter:

```
fci_ownership_mask:    Overrides bitmask of HIF channels that are allowed
                       to take FCI ownership (default 0: all senders allowed)
```

Example:

```
# insmod pfeng fci_ownership_mask=9
```

Value of `fci_ownership_mask` represents senders which are allowed to become FCI owners. The overall result sums individual HIF channels represented by following enum:

```
 FCI_OWNER_HIF_0 = (1 << 0),
 FCI_OWNER_HIF_1 = (1 << 1),
 FCI_OWNER_HIF_2 = (1 << 2),
 FCI_OWNER_HIF_3 = (1 << 3),
```

Important: *For backward compatibility reasons, FCI ownership provides a special "all senders allowed" mode. If* ***no bit*** *is set in the* `fci_ownership_mask`*, then all driver instances (senders) are by default allowed to take FCI ownership.*

3 Build Procedure

3.1 Building the driver

The PFE driver consists of number of smaller software modules. The main module producing the final driver is called *linux-pfeng* and depends on all the others. Successful build gives the final driver library *pfeng.ko*, which is an Ethernet driver for a particular Linux kernel version (against which it was built), located in `sw/linux-pfeng/` inside the project tree.

There are two ways to build the driver:

1. Integrated Yocto build as part of Linux BSP
2. Standalone build

3.1.1 Variant 1: Integrated Yocto build as part of Linux BSP

1. As prerequisite, save the PFE firmware files to any location on the local disk.
2. Build the standard NXP Auto Linux BSP image with Yocto to check that all necessary components are ready. This step is not only for verification, but it pre-compiles most of necessary BSP components which will be used later for creating the final SD card image that includes the PFE Linux driver.
3. Add the PFE driver(s)
   a. To add the PFE standalone driver include the following lines in `conf/local.conf`:
   ```
   DISTRO_FEATURES:append = " pfe"
   NXP_FIRMWARE_LOCAL_DIR = "/path/to/firmware/binaries/folder"
   ```
   b. To add the PFE multi instance mode drivers - the PFE Master and Slave drivers - add the following lines to `conf/local.conf`:
   ```
   DISTRO_FEATURES:append = " pfe pfe-slave"
   NXP_FIRMWARE_LOCAL_DIR = "/path/to/firmware/binaries/folder
   ```
   Note: The PFE Master driver or the PFE Slave driver can run independently, in case only one instance is needed in the current Linux host as the other instance would be running on another partition and/ or on another OS/ host.
4. Rebuild the Linux BSP image to include the PFF driver in the final SD card image.

Refer to the *Linux BSP User Manual* [3] for detailed Yocto build process instructions.

3.1.2 Variant 2: Standalone build

1. As prerequisite check all necessary development requirements:
   a. Host development GNU toolchain, including GNU-cc, GNU-make
   b. Linux kernel development files
2. Go to `sw/linux-pfeng/`.
3. Make sure that the following environment variables are set:
   a. KERNELDIR

Points to the directory where the Linux kernel development source files are located.

b. PLATFORM
The name of the GNU toolchain platform. In case of Linux BSP, it is “aarch64-linux-gnu”.

4. Clean working directories:

```
make KERNELDIR=<path to kernel> PLATFORM=aarch64-linux-gnu drv-clean
```

5. Building the driver in standalone mode:

```
make KERNELDIR=<path to kernel> PLATFORM=aarch64-linux-gnu all
```

6. Building the drivers in master-slave mode:
PFE Master driver -

```
# make PFE_CFG_MULTI_INSTANCE_SUPPORT=1 PFE_CFG_PFE_MASTER=1 KERNELDIR=<path
 to kernel> PLATFORM=aarch64-linux-gnu all
```

PFE Slave driver -

```
# make PFE_CFG_MULTI_INSTANCE_SUPPORT=1 PFE_CFG_PFE_MASTER=0 KERNELDIR=<path
 to kernel> PLATFORM=aarch64-linux-gnu all
```

Note: *The drivers produced by the standalone build contain debug information by default. If there is no need the debug symbols can be stripped and thus sizes of the driver files can be reduced significantly. To strip debug symbols, run:*

```
# aarch64-linux-gnu-strip --strip-debug pfeng.ko
# aarch64-linux-gnu-strip --strip-debug pfeng-slave.ko
```

3.2 Building the LibFCI library and libfci_cli

The Linux BSP contains recipes for building LibFCI and libfci-cli in the **`meta-alb/recipes-extended`** subdirectory.

Use the following commands to build the `libfci.a` library and the `libfci_cli` application:

```
user@dev:~/fsl-auto-yocto-bsp/build_s32g<XXX>$ bitbake libfci
...
user@dev:~/fsl-auto-yocto-bsp/build_s32g<XXX>$ bitbake libfci-cli
...
```

Yocto offers an extra task called devshell. The task will deposit all the libfci source code into a directory, apply all patches included in the recipe, and then will open a terminal in that directory:

```
user@dev:~/fsl-auto-yocto-bsp/build_s32g<XXX>$ bitbake -c devshell libfci
```

When invoked, all environmental variables are set up exactly like for compilation. The libfci.a library, which was built in the previous step, is located in the `sw/xfci/libfci/build/aarch64-linux-gnu-release` directory:

```
root@dev:~/fsl-auto-<...>/libfci/<...>/git# find . -name libfci.a
./sw/xfci/libfci/build/aarch64-fsl-linux-release/libfci.a
```

The associated header files are located here:

```
root@dev:~/fsl-auto-<...>/libfci/<...>/git# ls ./sw/xfci/libfci/public/
```

```
fpp_ext.h  fpp.h  libfci.h
```

The user space library (`libfci.a`) and the associated header files are enough to build FCI API based user space applications, like `libfci_cli`.

4 Usage

4.1 Supported development boards

Supported SoCs:

1. S32G2
2. S32G3

Supported development boards:

1. S32G2 RDB2 (S32G274A RDB2)
2. S32G2/S32G3 EVB
3. S32G3 RDB3 (S32G399A RDB3)
4. S32G3 EVB3

Table 5. Supported PFE interface modes in Linux BSP

PHY_IF (Linux i/f name)	Supported boards and interface modes									
	S32G2 RDB2		S32G3 RDB3		S32G2 EVB		S32G3 EVB		S32G3 EVB3	
EMAC0 (pfe0)	RGMII	✓	RGMII	✓	RGMII	✓	RGMII	✓	RGMII	✓
	SGMII 1G	✓	SGMII 1G	✓	SGMII 1G	✓	SGMII 1G	✓	SGMII 1G	✓
	SGMII 2.5G[1]	<✓>	SGMII 2.5G[1]	<✓>	SGMII 2.5G	<✓>	SGMII 2.5G	<✓>	SGMII 2.5G	<✓>
EMAC1 (pfe1)	RGMII[2]	✓	RGMII[2]	✓	RGMII	✓	RGMII	✓	RGMII	✓
	SGMII 1G	<✓>	SGMII 1G	✓	SGMII 1G	<✓>	SGMII 1G	✓	SGMII 1G	✓
	SGMII 2.5G	✗	SGMII 2.5G	<✓>	SGMII 2.5G	✗	SGMII 2.5G	<✓>	SGMII 2.5G	<✓>
EMAC2 (pfe2)	RGMII	<✓>	RGMII	<✓>	RGMII[3]	<✓>	RGMII[3]	<✓>	RGMII[3]	<✓>
	SGMII 1G	✗	SGMII 1G	✓	SGMII 1G	✓	SGMII 1G	✓	SGMII 1G	✓
	SGMII 2.5G	✗	SGMII 2.5G	✓	SGMII 2.5G	✗	SGMII 2.5G	✓	SGMII 2.5G	✓

[1] Connected to the SJA1110A switch
[2] Shares the same pins with GMAC
[3] Connected to the SJA1105Q switch

Table 6. Extended mode support of PFE interfaces in Linux BSP

PHY_IF (Linux i/f name)	Supported boards and interface modes	
	Modified S32G3 EVB3 + S32GRV-PLATEVB + modified ADTJA1101-RMII	
EMAC0 (pfe0)	RGMII	✓
	SGMII 1G	✓
	SGMII 2.5G	<✓>
EMAC1 (pfe1)	RGMII	✓
	SGMII 1G	✓
	SGMII 2.5G	<✓>
EMAC2 (pfe2)	RMII[1]	✓
	RGMII	<✓>
	SGMII 1G	✓
	SGMII 2.5G	✓

[1] Via ADTJA1101-RMII card attached in ETHERNET CARD (B PORT) of S32GRV-PLATEVB

Legend:

✓	Feature supported
<✓>	Feature supported and selected by default
✕	Feature not supported

Note:

To activate the interface modes that are disabled by default you need to reconfigure the corresponding PFE `'ethernet'` nodes in the board specific device tree files shipped with the Linux BSP.

Refer to the Linux BSP User Manual [3] and board specific user guides for connectivity details of the Ethernet ports.

Extended mode support of PFE interfaces listed in the Table 6 is not available on out of box HW. RMII on S32G platform is supported, requires custom HW setup and custom configurations that are described in separate documents.

4.2 Dependencies

The PFE driver has a direct dependency on the following Linux kernel drivers and configurations shipped with the Linux BSP:

1. PFE Controller reset driver
   Insures the reset of the PFE hardware when the PFE driver module is loaded. For that, the '*resets*' property needs to be configured in the SoC device tree (i.e. `s32g-pfe.dtsi`). Otherwise, PFE must be reset by the bootloader.
2. SerDes driver (SGMII support)
   SGMII PHY devices require the Linux kernel SerDes driver for internal link configuration. For that, the '*phys*' property needs to be configured in the SoC device tree. Otherwise, the SerDes must be preconfigured from the bootloader.
3. Clock framework
   The native Linux kernel clock framework needs to be functional and the '*clocks*' properties to be correctly configured for the PFE device tree nodes.

4.3 Interfaces setup

1. The *pfeng.ko* driver is embedded in the Auto Linux BSP SD card image and as such is automatically loaded on Linux start up.
2. To check available network interfaces:

   ```
   # ip a
   ```

   The PFE driver interfaces are named as follows:
   - <*pfeX*>, from 0 to 2, for main INJECT mode interfaces (connected to EMACs). Some of these may be missing depending on board type and device tree configuration;
   - <*pfeXsl*>, for slave mode interface (*pfeng-slave.ko* driver), depending on board type and device tree configuration;
   - <*hifX*>, from 0 to 3, for main INJECT mode interfaces (connected to HIFs). Some of these may be missing depending on board type and device tree configuration;
   - <*hifXsl*>, for slave mode interface (*pfeng-slave.ko* driver), depending on board type and device tree configuration;

- <*aux*>, for the AUX mode interface;

3. The main network interfaces (pfe0-2) come in unmanaged state. To configure them manually, use e.g.:

```
# ip addr add <interface_ip_address>/<mask> dev pfe<0-2>
# ip link set pfe<0-2> up
```

4. Test connection via *ping*:

```
# ping <remote_ip_address>
```

4.3.1 Interfaces setup - SJA1105 DSA switch case

Refer to the *Linux BSP User Manual* [3] for detailed information on the SJA1105 DSA Linux driver.

When the SJA1105 DSA Linux driver is successfully loaded on the S32G EVB board, *pfe2* becomes the DSA master interface for the switch and will show up in the interface listing as follows:

```
root@s32g274aevb:~# ip addr show
...
9: pfe2: <BROADCAST,MULTICAST> mtu 1504 qdisc noop state DOWN mode DEFAULT
 group default qlen 1000
    link/ether 00:01:be:be:ef:33 brd ff:ff:ff:ff:ff:ff
10: enet_p1@pfe2: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN
 mode DEFAULT group default qlen 1000
    link/ether 00:01:be:be:ef:33 brd ff:ff:ff:ff:ff:ff
11: enet_p2@pfe2: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN
 mode DEFAULT group default qlen 1000
    link/ether 00:01:be:be:ef:33 brd ff:ff:ff:ff:ff:ff
12: enet_p3@pfe2: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN
 mode DEFAULT group default qlen 1000
    link/ether 00:01:be:be:ef:33 brd ff:ff:ff:ff:ff:ff
13: enet_p4@pfe2: <BROADCAST,MULTICAST,M-DOWN> mtu 1500 qdisc noop state DOWN
 mode DEFAULT group default qlen 1000
    link/ether 00:01:be:be:ef:33 brd ff:ff:ff:ff:ff:ff
```

<*enet_pX*>, from 1 to 4, are the Linux network interfaces associated with the SJA1105 switch ports, and *pfe2* is the DSA master interface.

Two traffic configuration modes are possible, depending on desired use case:

1. **Host termination traffic mode configuration:**
   The simplest way to direct traffic to the Linux Host via pfe2 is to choose an external switch port that has the desired connectivity and use the DSA switch in single port mode configuration, according to the example below:

```
root@s32g274aevb:~# ip link set pfe2 up && ip addr add 172.16.1.1/24 dev
 enet_p3 && ip link set enet_p3 up
[  119.728990] pfeng 46000000.pfe: HIF2 started
[  119.729259] pfeng 46000000.pfe pfe2: configuring for fixed/rgmii link
 mode
[  119.729282] pfeng 46000000.pfe pfe2: Set TX clock to 125000000Hz
[  119.729556] pfeng 46000000.pfe pfe2: Set TX clock to 125000000Hz
[  119.729577] pfeng 46000000.pfe pfe2: Link is Up - 1Gbps/Full - flow
 control off
[  119.729998] IPv6: ADDRCONF(NETDEV_CHANGE): pfe2: link becomes ready
[  119.749868] sja1105 spi5.0 enet_p3: configuring for phy/rgmii-id link
 mode
root@s32g274aevb:~# [  122.813597] sja1105 spi5.0 enet_p3: Link is Up -
 1Gbps/Full - flow control off
[  122.813651] IPv6: ADDRCONF(NETDEV_CHANGE): enet_p3: link becomes ready

root@s32g274aevb:~# ping 172.16.1.5
PING 172.16.1.5 (172.16.1.5) 56(84) bytes of data.
```

```
64 bytes from 172.16.1.5: icmp_seq=1 ttl=64 time=0.648 ms
64 bytes from 172.16.1.5: icmp_seq=2 ttl=64 time=0.311 ms
64 bytes from 172.16.1.5: icmp_seq=3 ttl=64 time=0.287 ms
^C
```

Note that the SJA DSA switch can also be configured as a Linux bridge, with termination traffic directed to the Host via pfe2. In general, for successful on Host termination of the traffic that traverses the switch, DSA tagging of packets must be enabled. (check *Linux BSP User Manual* [3] for more details)

2. **Fast path traffic mode configuration:**
   When pfe2 is intended as a fast path bridge/ routing interface (configurable via FCI), the SJA1105 Linux DSA switch ports that forward traffic to/from pfe2 need to be assigned to a Linux VLAN aware bridge, as shown in the example below:

```
root@s32g274aevb:~# ip link set pfe2 up
root@s32g274aevb:~# ip link add br0 type bridge vlan_filtering 1 && ip link
 set br0 up && \
    for eth in enet_p1 enet_p2 enet_p3 enet_p4; do ip link set $eth master
 br0 && ip link set $eth up; done
[  707.939242] br0: port 1(enet_p1) entered blocking state
[  707.939331] br0: port 1(enet_p1) entered disabled state
...
[  707.947697] sja1105 spi5.0: Reset switch and programmed static config.
 Reason: VLAN filtering
...
```

At this point pfe2 can be used in fast-path bridging/ routing use cases, see the *FCI API Reference* [1].

4.4 Device tree node configuration

Every PFE Driver instance, be it standalone, master, or slave, has a device tree (DT) node that defines the allocated resources and their configuration.

The PFE DT node definition is organized into several levels:

1. The SoC family level is the first level of DT configurations and it includes the PFE properties that are common to all SoC revisions and boards. The definitions of PFE properties that are shared among all driver instance modes are included, via ***s32g-pfe.dtsi***, in the SoC family level DT file: *s32g.dtsi*.
2. The SoC generation level (i.e. G2/G3) , defined in files *s32g2.dtsi*, *s32g3.dtsi*, may include SoC revision/ generation specific DT configurations.
3. The board and board family level includes PHY and link layer configurations for the PFE standalone and Master driver nodes, like MDIO and link mode properties. These are included in DT files with the following naming patterns: *s32g2xxa-evb.dts*, *s32g3xxa-evb.dts*, *s32g274a-rdb2.dts*, etc. The board level files ending in `"-pfems"` contain common Slave driver mode configurations, included as ***s32g-pfe-slave.dtsi***, and board/ board family specific Slave drive mode configurations.

4.4.1 Device tree bindings

The PFE device node structure is defined by the device tree bindings file located in the Linux Kernel source code under `./Documentation/devicetree/bindings/net/nxp,s32g-pfe.yaml`, see the nxp,s32g-pfe.yaml appendix containing the definitions.

The following naming conventions are used to identify which property or sub-node is required by which driver instance:

- "*Master driver*" - refers to both the default standalone mode driver and the Master driver instance. A requirement that applies to the Master driver instance alone, and not to the standalone mode driver, is marked as "Master driver only".
- "*Slave driver*" - refers only to the Slave driver instance.
- "*Master-Slave mode*" - refers to a requirement that applies only to the Master-Slave driver mode, also known as multi-instance mode.
- In case the driver instance name or mode are not specified, the requirement is assumed to be common to all modes and instances.

4.5 Performance consideration

To optimize throughput of traffic termination on Host, the following configuration options are available:

- Restrict the Linux system memory size (`mem=2G`)
  Restrict the Linux system memory so that the whole available DDR range is covered by the 32 bit addressing space of PFE's DMA engine. For the S32G SoC the DDR is mapped starting with the address of `0x8000_0000`. This leaves only 2 GB of DDR that is reachable via 32 bit DMA addressing. Therefore, limit the memory to 2 GB in size to prevent copying of packet buffers between the DMA space and user space (aka bounce buffering).
  To enforce this configuration, pass the `mem=2G` argument to the kernel boot parameters. The argument can be appended in uboot to the `bootargs` environment variable.
- Enforce SMP interrupt affinity
  Ensure that the interrupt affinities of `pfe0` (IRQ name:`hif-0`), `pfe1` and `pfe2` are set to `CPU0`, `CPU1` respectively `CPU2`. This is done automatically by the PFE driver only if the `irqbalance` daemon is shut down before loading the driver.
  The recommended interrupt affinities can be configured manually with the following shell script:

```
# killall irqbalance
# irq () { cat /proc/interrupts | grep hif-$1 | sed -r "s/\s*([^:]+).*/\1/"; }
# echo 1 > /proc/irq/`irq 0`/smp_affinity
# echo 2 > /proc/irq/`irq 1`/smp_affinity
# echo 4 > /proc/irq/`irq 2`/smp_affinity
```

- Configure process affinity for the traffic termination application
  Assign the application process to a different CPU than the SMP interrupt affined one, but from the same cluster:

PFE driver interface	SMP interrupt affinity	Process affinity
`pfe0`	CPU0	CPU1
`pfe1`	CPU1	CPI0
`pfe2`	CPU2	CPU3

- Tuning ring sizes of HIF Channels
  Depending on use case, better throughput may be achieved by enlarging, or (sometimes) even by shrinking, the HIF Channel ring sizes. The default ring size is 256, and it is configurable through the driver compile time option `PFE_CFG_HIF_RING_LENGTH`.
  One downside of increasing the ring sizes is the increased memory demand, so the reserved memory region for the rings has to be enlarged (in the device tree), otherwise

not all HIF Channels can be enabled at the same time. Increasing ring sizes also accounts for increased traffic latency.

4.5.1 Fast path performance

For fast path use cases (like FCI VLAN bridging or routing) the configuration of DDR determines the maximum throughput that can be achieved, when the BMU2 buffer pool and the Routing Table (RT) are allocated in DDR. For increased fast path throughput (and improved system level performance), the user has the option to place the BMU2 buffer pool and the RT inside the S32G SoC internal SRAM memory region, see Placing the PFE System Buffers in SRAM or DDR. Allocating the BMU2 buffer pool inside SRAM also improves the traffic directed to the Linux Host. The downside is that the integrator needs to make sure that the targeted SRAM region is not used by someone else.

G3 Routing Table entries in LMEM

On S32G3 the PFE Routing Table may also be allocated inside a PFE local SRAM memory, called LMEM, making it accessible for fast path lookup via the local (faster) PFE bus (CBUS). This recently introduced optimization can be enabled on S32G3 through the `g3_rtable_in_lmem` PFE Linux driver module switch, i.e.:

```
# insmod pfeng g3_rtable_in_lmem=1
```

4.6 Diagnostic Options

4.6.1 Statistics and error messages

Statistics and error messages are gathered by different components of the PFE software stack and are available via different interfaces.

PFE components that export statistics are: `MAC interfaces`, `BMU`, `TMU`, `GPI`, `PFE driver` and `PFE Firmware`. The PFE Firmware exports statistics related to physical interface usage, classification algorithm, PE core statistics and VLAN statistics. For more information, see the *PFE Firmware User Manual* [2].

The stats can be collected from Debug files, LibFCI (refer to the *FCI API Reference* [1]) and the standard Linux network interface tools for host side statistics (e.g. iproute2, net-tools, iputils).

The error messages logged by the PFE driver and the PFE firmware are available in `dmesg`.

4.6.2 Debug files

The driver provides a set of file system nodes located in `/sys/kernel/debug/pfeng` representing various PFE functional blocks. The nodes provide detailed runtime information in case they are read and accept verbosity level value (1-10, bigger value means more verbose output) in case of write. Default verbosity value is 4 and can be changed during driver build time using the `PFE_CFG_VERBOSITY_LEVEL` configuration option. The driver currently supports the following nodes:

- `bmu1`, `bmu2`
  Information about BMU blocks and HW buffer pools.
- `class`
  Runtime statistics gathered by the Classifier and firmware related statistics.

For more information about class counters refer to the *PFE Firmware User Manual* [2], chapter '*Per-PE Statistics*' and '*Classification Algorithm Statistics*'.

- `emac0`, `emac1`, `emac2`
  Traffic statistics for every PFE EMAC.
- `gpi`
  GPI statistics and debug information.
- `hif0`, `hif1`, `hif2`
  HIF status and statistics.
- `tmu`
  TMU per-queue statistics for EMAC physical interfaces and debug information.
- `l2br`
  Mac learned for interfaces that are in l2bridge mode.
- `l2br_domain`
  VLAN traffic statistics.
- `rtable`
  Routing table entry statistics.

4.6.3 Dropped frame analysis

A frame can be dropped at any of the following PFE processing stages, before reaching the Host on ingress, or the network on egress:

- **Ingress emac physical interface**
  Drop stats are reported in the `emacX` file.
- **TMU**
  Per-queue drop stats are reported in the `tmu` file.
- **Firmware pkt processing**, which is done in three stages:
  - `ingress processing stage`
    Consists of interface status check, flexible filtering and STP port state checks.
  - `algorithm processing stage`
    Classifies the frame for transmission on one or more egress interfaces.
  - `egress processing stage`
    Contains checks of the egress interface(s) state.

  The statistics for ingress and egress processing stages are reported via LibFCI under physical interface counters (see *FCI API Reference*[1], chapter *fpp_phy_if_cmd_t Struct Reference*). The algorithm processing stage statistics are reported in `class` file.

4.6.4 Driver debug logs

Dynamic debug

Debug information contained in source code of the PFE Driver, in a form of *dev_dbg()* calls, can be dynamically enabled individually, per line. However, a precondition is that the NXP Auto Linux BSP kernel is compiled with *CONFIG_DYNAMIC_DEBUG* set.

Examples how to use dynamic debugging:

- Show all debug statements in PFE driver.

```
# cat /sys/kernel/debug/dynamic_debug/control | grep pfeng
```

- Enable a specific debug.

```
# echo 'file fci_spd.c line 78 +p' > /sys/kernel/debug/dynamic_debug/control
```

- Enable all debugs for PFE driver.

```
# echo 'module pfeng +p' > /sys/kernel/debug/dynamic_debug/control
```

- Load PFE driver and enable all debugs with function names, line numbers, module names, and thread IDs (IDs, but not from interrupt context).

```
# insmod pfeng dyndbg=+pflmt
```

See Dynamic debug in Linux kernel documentation for more details.

PFE driver msg_verbosity parameter

The PFE driver recognizes a parameter *msg_verbosity: 0 - 9, default 4 (int)*

The default parameter value *4* can be overriden, when it is explicitly specified at module load time. A provided parameter value greater equal 7 enables additional source file names and line information for all *pfe_platform* logs.

```
[22160.336968] pfeng 46000000.pfe: [pfe_platform_master.c:2924] PFE CBUS
 p0x46000000 mapped @ v0xffffffc016000000
```

Versus default

```
[22469.725503] pfeng 46000000.pfe: PFE CBUS p0x46000000 mapped @
 v0xffffffc016000000
```

Note: *The additional information is included for all of the pfe_platform logs and all their levels. It also applies to the dev_dbg(), but if kernel CONFIG_DYNAMIC_DEBUG feature is set.*

Examples how to load PFE driver:

- Enable all pfe_platform logs with file names and line numbers.

```
# insmod pfeng msg_verbosity=7
```

- Enable all the pfe_platform logs and dynamic debug logs.

```
# insmod pfeng dyndbg=+p msg_verbosity=7
```

- Enable all the pfe_platform logs and extended dynamic debug logs.

```
# insmod pfeng dyndbg=+pflmt msg_verbosity=7
```

4.6.5 Health Monitor

This diagnostics module monitors the PFE health status and reports runtime events related to:

- Interrupts
  - BMU events
- Polled events
  - Parity events
  - Watchdog events
  - BUS error events
  - Fail-stop
  - PFE ECC events
  - HIF events
  - EMAC events

  - Firmware events
- Runtime events
  - Platform events
  - Device events
  - Netdevice events

The Health Monitor module periodically checks and reports the following PFE Firmware error types:

- soft errors (error messages generated by the firmware),
- stalled firmware,
- exception state of the PE cores.

Events reported to Health Monitor are logged in `dmesg` and notification callbacks are executed.

Note: *Refer to the FCI API Reference [1] for details about `fci_register_cb` FCI event callback function. Alternatively see libfci_cli daemon feature demonstrating handling of FCI events.*

The Heath Monitor diagnostic event is defined by:

- type (error, warning, info)
- ID of the module that detected the event,
- text description.

Example of health monitor logs in `dmesg`:

```
[33665.833428] pfeng 46000000.pfe: ERR: (PARITY) event 120 - BMU2_SLV_INT-BMU2 slave parity error:
 [pfe_parity_csr.c:98]
[33665.833456] pfeng 46000000.pfe: ERR: (PARITY) event 121 - CLASS_SLV_INT-CLASS slave parity error:
 [pfe_parity_csr.c:98]
[33665.833464] pfeng 46000000.pfe: ERR: (PARITY) event 124 - LMEM_SLV_INT-LMEM slave parity error:
 [pfe_parity_csr.c:98]
```

Example of health monitor event caught by `libfci_cli` in daemon mode:

```
root@s32g399aevb3:~# ./libfci_cli daemon-start
DISCLAIMER: This is a DEMO application. It is not part of the production code deliverables.
Command successfully executed.
Fork the [libfci_cli daemon] : OK  (pid=917)
root@s32g399aevb3:~#
==== FCI_EVENT_beg ====================
timestamp   = 1608173516 (Thu Dec 17 02:51:56 2020)
fcode       = 0xF660     (FPP_CMD_HEALTH_MONITOR_EVENT)
len         = 70
payload_raw =
{
  |00|00|00|78|02|09|42|4D|55|32|5F|53|4C|56|5F|49|
  |4E|54|2D|42|4D|55|32|20|73|6C|61|76|65|20|70|61|
  |72|69|74|79|20|65|72|72|6F|72|00|00|00|00|00|00|
  |00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|00|
  |00|00|00|00|00|00|
}
payload_decoded =
{
  id   = 120
  type = 2     (ERROR)
  src  = 9     (HOST_FAIL_STOP)
  desc = BMU2_SLV_INT-BMU2 slave parity error
}
==== FCI_EVENT_end ====================
```

5 Appendix

5.1 nxp,s32g-pfe.yaml

```
# SPDX-License-Identifier: (GPL-2.0-only OR BSD-2-Clause)
# SPDX-License-Identifier: (GPL-2.0-only OR BSD-2-Clause)
# Copyright 2022-2023 NXP
```

```
%YAML 1.2
---
$id: "http://devicetree.org/schemas/net/nxp,s32g-pfe.yaml#"
$schema: "http://devicetree.org/meta-schemas/core.yaml#"

title: The PFE Ethernet accelerator with three MACs (Media Access Controller)

maintainers:
  - Jan Petrous <jan.petrous@nxp.com>
  - Claudiu Manoil <claudiu.manoil@nxp.com>

description: |
  The NXP S32G2/S32G3 automotive SoC contains PFE Ethernet controller.
  PFE offloads Ethernet packet processing from the host cores, yielding
  higher performance and lower power consumption than what software
  processing alone could achieve.

properties:
  "#address-cells": true
  "#size-cells": true

  compatible:
    items:
      - enum:
        - nxp,s32g-pfe
        - nxp,s32g-pfe-slave
    description:
      The single instance driver (aka standalone driver mode) and the Master
      driver require "nxp,s32g-pfe", the Slave driver requires "nxp,s32g-pfe-slave".

  reg:
    items:
      - description: The PFE IP physical base address and size
      - description: The SoC S32G MEM_GPR base address and size

  reg-names:
    items:
      - const: pfe-cbus
      - const: s32g-main-gpr

  clocks:
    items:
      - description: PFE SYS clock
      - description: PFE PE clock
      - description: TS clock

  clock-names:
    maxItems: 3
    oneOf:
      - items:
        - const: pfe_sys
        - const: pfe_pe
        - const: pfe_ts
      - items:
        - const: pfe_sys
        - const: pfe_pe

  interrupts:
    items:
      - description: HIF channel 0 interrupt
      - description: HIF channel 1 interrupt
      - description: HIF channel 2 interrupt
      - description: HIF channel 3 interrupt
      - description: BMU interrupt
      - description: UTIL interrupt
      - description: SAFETY interrupt

  interrupt-names:
    items:
      - const: hif0
      - const: hif1
      - const: hif2
      - const: hif3
      - const: bmu
      - const: upegpt
      - const: safety

  dma-coherent: true

  resets:
    maxItems: 1
    description: |
      Reference S32G reset controller used for PFE reset.

  resets-names:
    maxItems: 1

  memory-region:
    maxItems: 4
    items:
      - description: The BMU2 buffer pool, must be in the range 0x00020000 - 0xbfffffff
      - description: RT region
      - description: Non-cacheable DMA buffers
      - description: Buffer descriptor rings

  memory-region-names:
    maxItems: 4
    items:
      - const: pfe-bmu2-pool
      - const: pfe-rt-pool
      - const: pfe-shared-pool
```

```
      - const: pfe-bdr-pool

  nxp,fw-class-name:
    $ref: /schemas/types.yaml#/definitions/string
    description: |
      If present, name (or relative path) of the file within the
      firmware search path containing the firmware image used when
      initializing PFE CLASS hardware.

  nxp,fw-util-name:
    $ref: /schemas/types.yaml#/definitions/string
    description: |
      If present, name (or relative path) of the file within the
      firmware search path containing the firmware image used when
      initializing PFE UTIL hardware. Optional.

  nxp,pfeng-emac-ts-ext-modes:
    $ref: /schemas/types.yaml#/definitions/uint32-array
    description: |
      The set of PFE_EMACs required to work in external timestamping
      mode. The combination of external TS support is limited,
      check S32G Reference Manual for detailed info.
    maxItems: 3
    contains:
      enum:
        - PFE_PHYIF_EMAC_0
        - PFE_PHYIF_EMAC_1
        - PFE_PHYIF_EMAC_2

  nxp,pfeng-ihc-channel:
    $ref: /schemas/types.yaml#/definitions/uint32
    description: |
      The HIF channel number used for IHC transport.
    maxItems: 1
      enum:
        - PFE_PHYIF_HIF_0
        - PFE_PHYIF_HIF_1
        - PFE_PHYIF_HIF_2
        - PFE_PHYIF_HIF_3


  nxp,pfeng-master-channel:
    $ref: /schemas/types.yaml#/definitions/uint32
    description: |
      The HIF channel number used for IHC transport.
      The destination channel used by Master for
      receiving IHC messages. Here can also be used
      the HIF_NCPY channel.
    maxItems: 1
      enum:
        - PFE_PHYIF_HIF_0
        - PFE_PHYIF_HIF_1
        - PFE_PHYIF_HIF_2
        - PFE_PHYIF_HIF_3
        - PFE_PHYIF_HIF_NOCPY

  phys:
    maxItems: 3

  phy-names:
    description: |
      Required for SGMII mode. Should reference
      S32G SerDes XPCS instance. One per PFE_EMAC.
    maxItems: 3
    contains:
      enum:
        - emac0_xpcs
        - emac1_xpcs
        - emac2_xpcs

  # mdio
  mdio:
    $ref: mdio.yaml#
    description: |
      Optional node for embedded MDIO controller.
    properties:
      compatible:
        items:
          - const: nxp,s32g-pfe-mdio
      reg:
        maxItems: 1
        description: The index of embedded MDIO bus.
      unevaluatedProperties: false

  # ethernet
  ethernet:
    $ref: ethernet-controller.yaml#
    description: |
      Describes net device.
    properties:
      compatible:
        items:
          - const: nxp,s32g-pfe-netif
      local-mac-address: true
      phy-mode: true
      phy-handle: true
      nxp,pfeng-if-name:
        $ref: /schemas/types.yaml#/definitions/string
        description: |
          The netdev interface name.
      nxp,pfeng-hif-channels:
```

```
        $ref: /schemas/types.yaml#/definitions/uint32-array
        description: |
          The set of used HIF channels. Any combination of standard
          HIF channel can be used. HIF_NOCPY is not supported.
        minItems: 1
        maxItems: 4
          enum:
            - PFE_PHYIF_HIF_0
            - PFE_PHYIF_HIF_1
            - PFE_PHYIF_HIF_2
            - PFE_PHYIF_HIF_3
      nxp,pfeng-emac-id:
        $ref: /schemas/types.yaml#/definitions/uint32
        description: |
          The linked PFE_EMAC port. Use nxp,pfeng-linked-phyif instead.
        deprecated: true
      nxp,pfeng-linked-phyif:
        $ref: /schemas/types.yaml#/definitions/uint32
        description: |
          The linked PFE physical port.
        maxItems: 1
          enum:
            - PFE_PHYIF_EMAC_0
            - PFE_PHYIF_EMAC_1
            - PFE_PHYIF_EMAC_2
            - PFE_PHYIF_HIF_NOCPY
            - PFE_PHYIF_HIF_0
            - PFE_PHYIF_HIF_1
            - PFE_PHYIF_HIF_2
            - PFE_PHYIF_HIF_3
      nxp,pfeng-netif-mode-aux:
        $ref: /schemas/types.yaml#/definitions/flag
        description: |
          If present, the netdevice is used in AUX mode.
      nxp,pfeng-netif-mode-mgmt-only:
        $ref: /schemas/types.yaml#/definitions/flag
        description: |
          If present, receive only "management" frames
          (PTP, egress TS, mirrored), for EMAC bound netdevices.
      unevaluatedProperties: false

required:
  - compatible
  - reg
  - reg-names
  - interrupts
  - interrupt-names
  - '#address-cells'
  - '#size-cells'
  - ethernet

additionalProperties: false

unevaluatedProperties: false

allOf:
  - if:
      properties:
        compatible:
          contains:
            const: nxp,s32g-pfe
    then:
      required:
        - clocks
        - clock-names
    else:
      required:
        - nxp,pfeng-master-channel
```

6 Legal information

6.1 Definitions

Draft — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

6.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Terms and conditions of commercial sale — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at http://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

Suitability for use in automotive applications — This NXP product has been qualified for use in automotive applications. If this product is used by customer in the development of, or for incorporation into, products or services (a) used in safety critical applications or (b) in which failure could lead to death, personal injury, or severe physical or environmental damage (such products and services hereinafter referred to as "Critical Applications"), then customer makes the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, safety, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. As such, customer assumes all risk related to use of any products in Critical Applications and NXP and its suppliers shall not be liable for any such use by customer. Accordingly, customer will indemnify and hold NXP harmless from any claims, liabilities, damages and associated costs and expenses (including attorneys' fees) that NXP may incur related to customer's incorporation of any product in a Critical Application.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Translations — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

Security — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

6.3 Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

NXP — wordmark and logo are trademarks of NXP B.V.

Contents

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 13 April 2023
Document identifier: S32G_PFE_LNX_DRV_UM