LNX PFE Driver User Manual

Rev. BETA 0.9.6 —
18 November 2021

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]

1 Introduction

1.1 References

Table 1. References

#	Title
1	"Linux BSP 31.0 User Manual for S32G"
2	"FCI API Reference" document
3	"S32G PFE Firmware User Manual"
4	"Ethernet Controller Generic Binding", ethernet-controller.yaml (located in the Linux kernel source tree under ./Documentation)
5	"Ethernet PHY Generic Binding", ethernet-phy.yaml (located in the Linux kernel source tree under ./Documentation)
6	"MDIO Bus Generic Binding", mdio.yaml (located in the Linux kernel source tree under ./Documentation)

1.2 Acronyms and Definitions

Table 2. 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 Firmware UM.
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 Manual 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 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 Firmware UM.
- 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 UM.

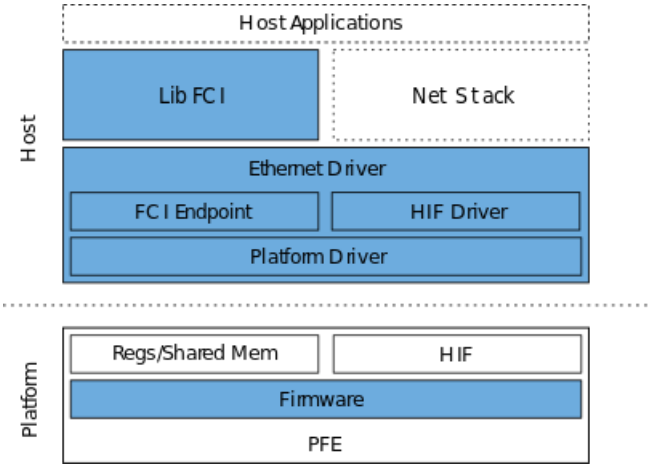

Figure 1. PFE software stack components

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 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 (unencapsulated) 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
```

2.1.1 Known limitations

PFE supports Tx checksum offloading for the following protocol layers.

Table 3. 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*

Table 3. Supported Tx checksum offload protocols*...continued*

Supported protocols	Supported frame header combinations
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).

2.2 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
```

2.3 Frame size and scatter-gather

The maximum supported L2 frame size is 1522B.

On ingress, each frame of up to 1522 bytes is received in a single memory buffer. Frames exceeding this size are discarded at MAC level.

On egress, the maximum supported MTU is 1504B, for the standard 1500B packet size plus a 4 byte tag needed 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 with the following command:

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

The driver also 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 (and re-enabled) with the following ethtool command:

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

2.4 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).

2.4.1 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: ---
```

2.4.2 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. Multiple LOG_IFs can be attached to the same PHY_IF, so to the same ingress port, and they can be configured, enabled or disabled, added or removed, at runtime via LibFCI applications. 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: ---
[...]
```

2.4.3 Linux Network Interfaces

The Linux network interfaces, or Linux netdevices, 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 netdevices are associated with at least one HIF port. A Linux netdevice can be associated with multiple HIF ports to allow for traffic load balancing. The association with multiple HIF ports is specified by the `'fsl,pfeng-hif-channels'` device tree property.

Known Limitation

Currently, the PFE Linux driver does not support load balancing, as it selects only the first HIF port for egress from the `'fsl,pfeng-hif-channels'` list. Concurrent reception from multiple HIF ports on the same network interface is also not fully supported.

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 netdevices associated with the PFE ports. For the interfaces bound to EMAC ports, the Linux Driver creates a Default LOG_IF for each bound EMAC port with the same name as the Linux Network Interface (i.e. "*pfeX*") to direct the ingress traffic to one of the HIF ports associated with Linux Interface, 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). 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 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 and their modes are specified in the PFE device tree node as `'ehternet'` sub-nodes with the `'fsl-pfeng-logif'` compatibility string.

There are two PFE Linux Network Interface types and their operation mode is set by the `'fsl,pfeng-logif-mode'` property:

1. EMAC bound network interfaces, with 2 operation modes -
   - `PFENG_LOGIF_MODE_TX_INJECT` (Default);
   - `PFENG_LOGIF_MODE_TX_CLASS`;
2. The Auxiliary (AUX) network interface - `PFENG_LOGIF_MODE_AUX`.

2.4.4 EMAC bound Network Interfaces

For EMAC bound network interfaces the Linux netdevice instance owns an EMAC port. As part of its initialization, the netdevice 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 netdevice. The Default LOG_IF is configured to direct the traffic to one of the HIF ports associated with the netdevice.

The EMAC attached netdevices are defined in the device tree as `'ethernet'` nodes with the `'fsl,pfeng-emac-link'` property set to the bound EMAC DT node.

On transmission, there are two modes by which these interfaces operate as detailed in the next sections.

2.4.4.1 TX_INJECT mode (default)

The TX_INJECT mode, when `'fsl,pfeng-logif-mode'` is set to `PFENG_LOGIF_MODE_TX_INJECT`, is also the default operation mode of an EMAC bound network interface. In this mode, the driver marks the custom HIF Tx header of every packet to instruct PFE to skip the classification stage and send these 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):

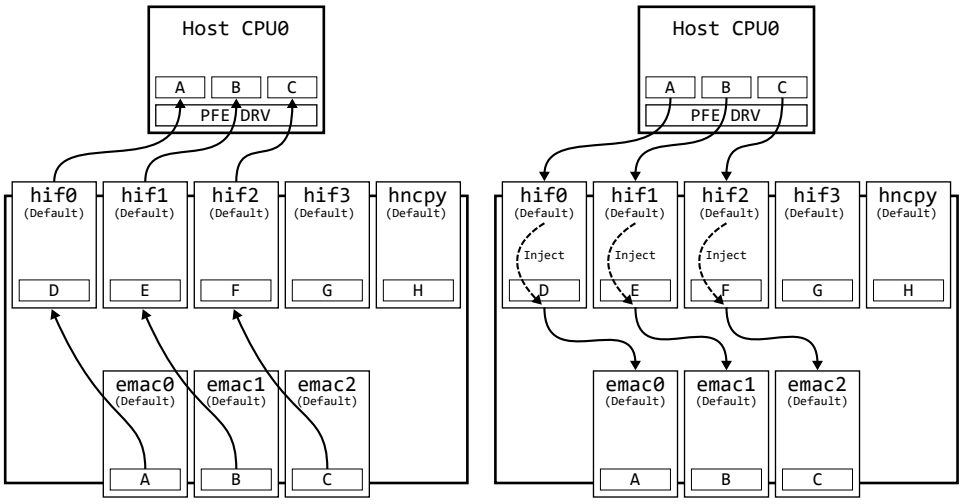

Figure 2. Multiple TX_INJECT interfaces with separate HIF ports

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

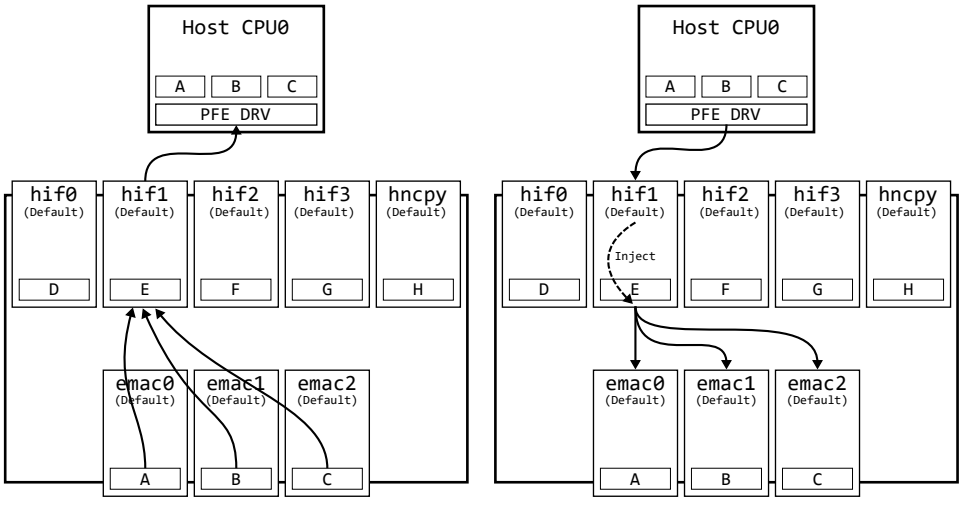

Figure 3. Shared HIF port by multiple TX_INJECT interfaces

When the network interface is in this mode, its associated Default HIF LOG_IF is disabled.

2.4.4.2 TX_CLASS mode

When `'fsl,pfeng-logif-mode'` is set to `PFENG_LOGIF_MODE_TX_CLASS`, the EMAC bound network interface operates in the TX_CLASS mode and the egress packets are dispatched according to the classification rules programmed to the HIF port on which the interface sends the packets. For this to work, the driver creates and enables a Default HIF LOG_IF for the HIF port used for transmission, and the egress port for the Default HIF LOG_IF is set to the attached EMAC port.

For these interfaces, additional LOG_IFs can be configured and attached to the HIF port via LibFCI, to allow for more advanced classification of egress traffic. Read more about advanced classification options in the FCI API Reference.

2.4.5 Auxiliary (AUX) Network Interface

The Auxiliary (AUX) Network Interface does not own an EMAC port. This interface captures ingress traffic that is not dispatched to an EMAC attached interface. On transmission, this interface behave exactly as the TX_CLASS mode interface as detailed in Section 2.4.4.2, with the exception the driver does not set any egress EMAC port for the Default HIF LOG_IF of the HIF port associated with the AUX interface. When in this mode, the Default HIF LOG_IF can be further configured from user space (via LibFCI) to forward traffic, depending on desired use case.

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

2.4.5.1 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 another Host PFE driver, as depicted in the following figures.

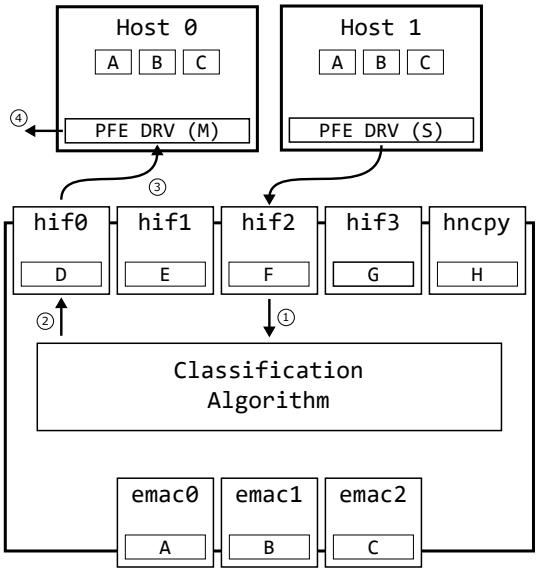

Figure 4. Unknown Rx Traffic without AUX

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

1. 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 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:

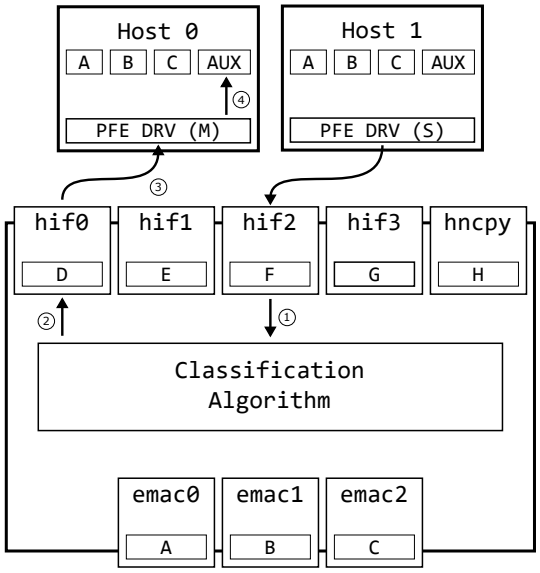

Figure 5. The AUX Endpoint RX

1. The Frame is received via HIF2.
2. 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.

2.5 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).*

2.5.1 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 `'fsl,pfeng-ihc'` property to the HIP port node. For example:

```
pfe_hif0: hif@0 {
        compatible = "fsl,pfeng-hif";
        #address-cells = <1>;
        #size-cells = <0>;
        status = "okay";
        interrupts = <GIC_SPI 190 IRQ_TYPE_EDGE_RISING>;
        reg = <0>;
        fsl,pfeng-hif-mode = <PFENG_HIF_MODE_EXCLUSIVE>;
        fsl,pfeng-ihc;
};
```

Note: *Only one network interface can have the `'fsl,pfeng-ihc'` property.*

2.5.2 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 ...
```

Slave functionality requires two configuration options in the device tree:

- Setting the global configuration option `'fsl,pfeng-master-hif-channel'` with Master's HIF channel number (see example in Section 2.5.1). 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 `'fsl,pfeng-ihc'` DT property, similarly to Master. See Section 2.5.1.

Note: *Only one network interface can have the `'fsl,pfeng-ihc'` property.*

2.5.3 Runtime

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.

Since the Master Detection procedure was implemented recently and since it has dependencies on other Hosts and OSes, there is also a fallback procedure in case synchronization fails:

- Disable the Master Detection support from the Linux Slave driver instance via the `'disable_master_detection'` module command line parameter;

- Run the Master driver instance first and ensure that the network interface for IHC communication is up, before launching the Slave instances.

The Slave instances also make specific PFE configurations to ensure that packets can reach them. PFE then distributes traffic to particular driver instances based on the destination MAC address of the ingress packets as follows:

- Packets received from EMAC ports with the destination MAC address matching a Slave instance's endpoint MAC address are delivered to that endpoint.
- All remaining traffic is delivered to the Master instance.
- Default MAC address-based traffic distribution can be changed using FCI.

2.6 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

2.6.1 Clock configuration

The S32G platform allows multiple clock configurations, as depicted in Figure 6.

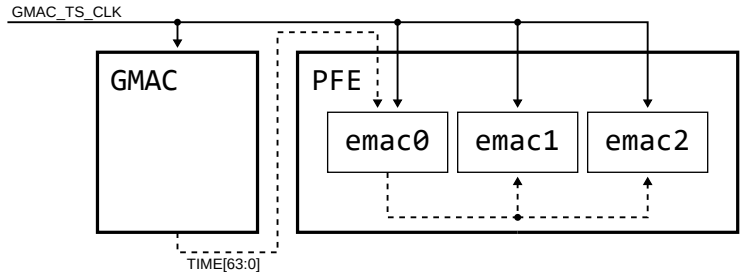

Figure 6. S32G2 IEEE1588 clock distribution

2.6.1.1 Internal Timestamp Mode

See Figure 6, 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 *fsl-s32g.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

2.6.1.2 External Timestamp Mode

See Figure 6, dashed line.

The internal PFE EMAC timer is disabled and the time information is delivered from an external source. This source can be the GMAC instance for EMAC0, and then EMAC0 becomes the external time source for EMAC1 and EMAC2.

Note: *This configuration is currently not supported.*

2.6.2 Driver interface

Driver implements standard Linux API for HW timestamping configuration.

2.6.2.1 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`)

2.6.2.2 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.

2.6.2.3 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
```

2.6.3 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"
```

2.7 Power Management

2.7.1 Prerequisites

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

2.7.1.1 Compile time

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

2.7.1.2 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 UserManual*', chapter '*Power Management*'.
- Hardware capable of doing suspend.

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

2.7.2 Suspend to RAM

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

2.7.2.1 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

2.7.2.2 Resume

On resume, first the S32G PFE partition reset is executed, then components get resumed in reverse order of suspension (see Section 2.7.2.1).

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.*

2.8 LibFCI

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 document.

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.

2.8.1 Fast path bridging use case example

To configure a simple VLAN unaware 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:
   - "BRIDGE" mode
   - enable traffic promiscuity
   - blocking state mode "NORMAL"

```
# libfci_cli phyif-update --i emac0 -E --promisc ON --mode BRIDGE --bs
 NORMAL
# libfci_cli phyif-update --i emac1 -E --promisc ON --mode BRIDGE --bs
 NORMAL
# libfci_cli phyif-update --i emac2 -E --promisc ON --mode 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: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 manual for details on LibFCI configurations.

2.8.2 Ingress QoS use case examples

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

- Classification of ingress traffic into 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:

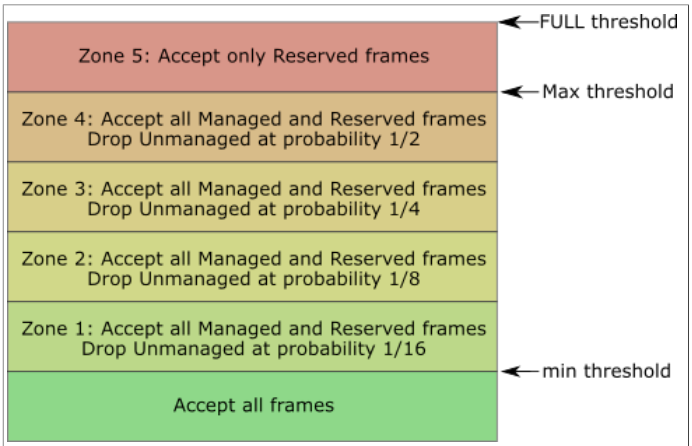

Figure 7. WRED zones and thresholds

Rate shaper configuration example for *pfe0*

1. Enable the Ingress QoS block (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 30
   - set *loCredit* to -1470

```
# libfci_cli  qos-pol-shp-update -i emac0 -E --shp=0 --shp-mode=DATA_RATE --
shp-type=PORT --isl=100000000 --crmin=-1470 --crmax=30
```

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: -1470
    credit-max: 30
[...]
```

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

WRED configuration example for *pfe0*

1. Enable the Ingress QoS block (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 emac0Wred 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 in 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.
```

2.8.2.1 Known Issues

1. Classification of more complex flows than simple ARP traffic is not functional at the moment - the issue is under investigation.
   Impact: All ingress traffic is classified as "Unmanaged" and subject to the configured WRED policy for "Unmanaged" flows.
2. The shaper output rate is roughly half of the requested idleSlope rate, for both bit rate and packet rate modes - the issue is under investigation.
   Impact: idleSlope has to be set to 2x the intended rate to compensate for the unintended scaling factor.

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 container with restricted access permissions).

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

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

2.9.1 PFE System Buffers region

This is the memory region where Ethernet frames should be stored while they are being processed by the PFE. The region is defined in the device tree by a dedicated 'reserved-memory' sub-node having the **`'fsl,pfe-bmu2-pool'`** compatibility string. The memory region is exclusively reserved for PFE's BMU2 hardware block managing the buffers for in-transit Ethernet frames. The physical start address, alignment and size of this region meet the restrictions imposed by the BMU2 hardware block. The allocation of the BMU2 buffer pool is also controlled by the `PFE_CFG_SYS_MEM` compile time configuration option. To allocate the BMU2 buffer pool inside this exclusive memory region, `PFE_CFG_SYS_MEM` should be set to `'pfe-bmu2-pool'` (current default). The BMU2 buffer pool can also be allocated inside the DMA shared pool region by setting `PFE_CFG_SYS_MEM` to 'pfe_ddr' but this is not recommended (sub-optimal).

2.9.2 Buffer Descriptor Rings region

This is a cache enabled memory region reserved for performance critical data structures. It is intended mainly for the Rx and Tx buffer descriptor rings, but it is also used for storing preallocated Tx packet metadata headers. The memory region is defined in the device tree by a dedicated `'reserved-memory'` sub-node having the **`'fsl,pfe-bdr-pool'`** compatibility string. This is a cache coherent memory region that 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 should be set to 'pfe-bdr-pool' (current default). By setting `PFE_CFG_BDR_MEM` to `'pfe_ddr'`, all allocations targeting the `PFE_CFG_BDR_MEM` memory pool (buffer descriptors and Tx headers) will be made in the non-cacheable DMA shared pool region, resulting in performance degradation.

2.9.3 DMA shared pool region

This is the common memory region reserved for the remaining DMA structures used by PFE, most notably by the routing table. 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. To allocate memory pools inside this shared memory region set the corresponding `PFE_CFG_*_MEM` compile time configuration options to `'pfe_ddr'`.

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 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
```

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 EVB (S32G274A EVB)
3. S32G3 EVB

Table 4. Supported interfaces for S32G2 RDB2 in Linux BSP

Physical interface	Linux name	RGMII	SGMII-1G	SGMII-2.5G
EMAC0	pfe0	Disabled	Disabled	Enabled and connected to SJA1110A
EMAC1	pfe1	Disabled - shares the same pins with GMAC	Disabled	Not Supported
EMAC2	pfe2	Enabled	Not Supported	Not Supported

Table 5. Supported interfaces for S32G2 EVB G2 and G3 in Linux BSP

Physical interface	Linux name	RGMII	SGMII-1G	SGMII-2.5G
EMAC0	pfe0	Disabled	Disabled	Enabled
EMAC1	pfe1	Enabled	Disabled	Not Supported
EMAC2	pfe2	Enabled and connected to SJA1105Q	Not Supported	Not Supported

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

Refer to the Linux BSP User Manual and board specific user guides for Ethernet connectivity details.

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. `fsl-s32g.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;
   - <*aux*>, for the AUX mode interface;
3. The main network interfaces (pfe0 [,pfe1], pfe2) are configured as DHCP clients. If DHCP is not desired, remove appropriate lines in '`/etc/network/interfaces`' and configure them manually, 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 UM for detailed information on the SJA1105 DSA Linux driver.

When the SJA1105 DSA Linux driver is successfully loaded on the S32G2 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:**
   To direct traffic to the Linux Host via pfe2, choose a switch interface that has link and use the simple single port mode configuration example below to check the connectivity:

```
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 in non-VLAN aware bridge mode, with termination traffic directed to the Host via pfe2. In general, in order to have the traffic going through the switch successfully terminated on the Host via pfe2, DSA tagging of packets must be enabled. Refer to the Linux BSP UM for SJA1105 Linux DSA switch driver configuration modes.

2. **Fast path traffic mode configuration:**
   When pfe2 is used as a fast path bridge/routing interface (configurable via FCI), the SJA1105 Linux DSA switch ports should be configured into a VLAN aware bridge to disable DSA tagging for packets traversing the switch, as shown in 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
[  707.960705] sja1105 spi5.0: Disabled switch tagging
...
```

   At this point, pfe2 can be used now for fast-path bridging/routing use cases configurable via LibFCI, see the FCI API manual.

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 device tree node definition is organized into two levels, with separate DT files for each level:

1. SoC family level DT configurations, for the PFE properties that are common on all SoC revisions and boards, located in: *fsl-s32g.dtsi*;
2. Board level DT configurations, for PHY layer board specific configurations mostly, e.g.: *fsl-s32g274a-evb.dts*, *fsl-s32g274a-rdb2.dts*, etc.;

The PFE device node structure will be detailed next. 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.

4.4.1 PFE device node

1. Compatible strings (required):
   - `"fsl,s32g274a-pfeng"` or `"fsl,s32g274a-pfe"` [*Master driver*]
   - `"fsl,s32g274a-pfeng-slave"` [*Slave driver*]
2. Required properties:
   - `reg`: list of register block specifiers (base address and size). Register blocks:
     – PFE status and configuration registers;
     – global syscon registers; [*Master driver*]
   - `clocks`: list of clock sources specifiers, should contain an entry for every entry in 'clock-names'; [*Master driver*]
   - `clock-names`: should contain at a minimum "pfe_sys" and "pfe_pe"; [*Master driver*]
   - `resets`: specifies the reset partition for PFE; [*Master driver*]
   - `reset-names`: must be "pfe_part", reset partition name entry for 'resets'; [*Master driver*]
   - `interrupts`: list of all PFE interrupt sources except HIF interrupts (per channel); [*Master driver*]
   - `interrupt-names`: must be "bmu", "upegpt", "safety"; [*Master driver*]
   - `interrupt-parent`: phandle to the interrupt controller node;
   - `memory-region`: list of phandles to 'reserved-memory' sub-nodes, specifying physical address ranges for the PFE driver buffer pools. Refer to Section 2.9 chapter for details. PFE reserved memory sub-nodes:
     a. reserved region for the BMU2 buffer pool: [*Master driver*]
        - compatible: `"fsl,pfe-bmu2-pool"`;
        - the valid address range must be within `0x00020000 - 0xbfffffff` and the start address must be aligned to the pool size.
     b. reserved region for non-cacheable DMA buffers:
        - compatible: `"shared-dma-pool"`
     c. reserved region for buffer descriptor rings:
        - compatible: `"fsl,pfe-bdr-pool"`

3. Optional properties:
   - `dma-coherent`: Enables hardware coherency for DMA buffers (recommended);
   - `nvmem-cells`: phandle to the SoC revision register located in the SIUL2 block;
   - `nvmem-cell-names`: must be "soc_revision";
   - `phys`: list of SerDes PCS device specifiers (internal PHYs), required for configuring SGMII interfaces;
   - `phy-names`: list of PCS device names, one for each EMAC, must be "emac0_xpcs", "emac1_xpcs" and "emac2_xpcs";
   - `fsl,fw-class-name`: PFE CLASS firmware file name;
   - `fsl,fw-util-name`: PFE UTIL firmware file name;
   - `fsl,pfeng-master-hif-channel`: Master's HIF channel id (0-3 or 4 for NOCPY) required by the Slave driver as the destination endpoint for Inter Host Communication (IHC). [*Slave driver*]
   - `fsl,l2br-default-vlan`: Default L2 Bridge VLAN ID. If missing, VLAN ID 1 is assumed as default. [*Master driver*]
4. Required child nodes by name - the driver node should contain at least one of each type:
   a. `'hif'` (compatible: `"fsl,pfeng-hif"`) - HIF channel instance node;
   b. `'emac'` (compatible: `"fsl,pfeng-emac"`) - EMAC instance node
   c. `'ethernet'` (compatible: `"fsl,pfeng-logif"`) - Linux netdevice instance node

4.4.2 HIF Channel node

1. Compatible string (required): `"fsl,pfeng-hif"`.
2. Required properties:
   - `reg`: HIF Channel ID, the valid range is 0-3 (regular channels);
   - `interrupts`: Interrupt source specification for this HIF Channel;
   - `fsl,pfeng-hif-mode`: HIF channel mode, can be `PFENG_HIF_MODE_EXCLUSIVE` or `PFENG_HIF_MODE_SHARED`. See Section 2.4.
   - `fsl,pfeng-ihc`: Marks the channel as IHC channel. Can be only one per driver instance. [*Master-Slave mode*]

4.4.3 EMAC node

1. Compatible string (required): `"fsl,pfeng-emac"`.
2. Required properties:
   - `reg`: EMAC ID, the valid range is 0-3;
   - `phy-mode`: See ethernet-phy.yaml DT bindings document;
3. Optional properties:
   - `clocks`: list of MII clock sources specifiers, should contain an entry for every entry in `'clock-names'`; [*Master driver*]
   - `clock-names`: "tx_sgmii", "tx_rgmii", "tx_rmii", "tx_mii", "rx_sgmii", "rx_rgmii", "rx_rmii", "rx_mii"; [*Master driver*]
4. Optional sub-node:
   - `'mdio'` (compatible: `"fsl,pfeng-mdio"`) - specifies the MDIO bus, as defined in the mdio.yaml DT binding;

4.4.4 Ethernet node

The `'ethernet'` node specifies a Linux netdevice instance, also referred to as traffic endpoint or logical interface in the PFE software stack.

1. Compatible string (required): `"fsl,pfeng-logif"`.
2. Required properties:
   - `fsl,pfeng-if-name`: Netdevice interface name visible in the Linux. The same name is used to identify the associated logical interface instance from the PFE Firmware.
   - `fsl,pfeng-logif-mode`: Logical interface mode, can be `PFENG_LOGIF_MODE_TX_INJECT`, `PFENG_LOGIF_MODE_TX_CLASS`, or `PFENG_LOGIF_MODE_AUX`. See Section 2.4.
   - `fsl,pfeng-hif-channels`: List of phandles of 'hif' nodes for every channel instance managed by this netdevice.
   - `fsl,pfeng-emac-link` : handle to corresponding 'emac' node for all pfeng-logif-modes except PFENG_LOGIF_MODE_AUX. See Section 2.4. [*Master driver*]
   - `fsl,pfeng-emac-id`: The ID of the EMAC owned by the Master driver. Valid range: 0-2. [*Slave driver*]
3. Optional properties:
   - `fsl,pfeng-emac-router`: Switch the linked EMAC to `IF_OP_FLEX_ROUTER` mode, see FCI UM. [*Slave only*]
   - `local-mac-address`: See ethernet-controller.yaml DT bindings.
   - `phy-handle`: See ethernet-controller.yaml DT bindings.
   - `fixed-link`: See 'fixed-link' property from ethernet-controller.yaml.

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

- Increase the ring sizes of HIF Channels
  Better throughput can be achieved by doubling 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 this configuration is that not all HIF Channels can be enabled at the same time or the reserved memory region for the rings has to be enlarged (in the device tree). It also accounts for increased traffic latency.

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 stats related to physical interface usage, classification algorithm, Per PE stats and VLAN statistics (For more information look in '*PFE_Firmware_S32G_UserManual*',).

The stats can be collected from Debug files, LibFCI (refer to the FCI API Reference manual) 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 UM, 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.

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 manual, chapter '*fpp_phy_if_cmd_t Struct Reference*'). The algorithm processing stage statistics are reported in `class` file.

5 Legal information

5.1 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.

5.2 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 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.

5.3 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.

Tables

Figures

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: 18 November 2021
Document identifier: S32G_PFE_LNX_DRV_UM