S32G Linux PFE Driver User's Manual

Contents

Chapter 1

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 Patch 1	Added libfci [JPet]

Table 1.1: Revision History

Chapter 2

Introduction

The Linux PFE driver is, in general, an OS-specific SW component responsible for the complex PFE management. It consists of three main functional blocks:

- The platform driver
  This part covers the initial, low-level PFE HW bring-up, configuration, firmware upload, and the SW representation of the PFE HW components. This part can be described as a low-level PFE driver providing the interface to the hardware (including the firmware).

- The data-path driver
  Data-path driver covers tasks related to the Ethernet data traffic in terms of passing the packets between the PFE and the networking stack. This includes the implementation of the Host Interface (HIF) driver and connection with the host OS-provided the networking stack.

- Control path driver
  The block in charge of the functionality related to runtime PFE engine configuration and monitoring, implementing the Fast Control Interface (FCI) endpoint which is accessible from the user-space through related FCI library.

The driver runs within the target host OS environment and connects the PFE with networking stack. It provides access to physical Ethernet interfaces via exposing logical interfaces to the OS, and the OS together with user's applications can use the Ethernet connectivity via standard OS-provided interfaces (i.e. network sockets).

Chapter 3

Building 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 particular Linux kernel. Its location within the project tree in `sw/linux-pfeng/`. There are two ways to build the driver:

1. Integrated build by NXP Automotive Linux BSP, powered by Yocto

2. Standalone

3.1.1 Variant 1: Building within NXP Auto Linux BSP source tree by Yocto

1. As prerequisite, save the PFE firmware file to any location on the local disk. Ask NXP representative for PFE firmware package.

2. Build standard NXP Auto Linux BSP to check that all necessary componens are ready. This step is not only for verification but it precompiles most of necessary part of BSP which will be used later for creating sdcard image with included pfeng Linux driver.

3. Configure additional component by adding the following two lines to the conf/local.conf

```
DISTRO_FEATURES_append = " pfe "
PFE_LOCAL_FIRMWARE_DIR = "<some directory path where the fw file was
    saved>"
```

4. Rebuild BSP to get PFE driver + firmware included in the sdcard image

3.1.2 Variant 2: Building standalone

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 and points to the right directories:

    (a) KERNELDIR

        The direcory where the Linux kernel development files are located.

    (b) Optionally also PLATFORM

        The name of the GNU toolchain platform. In case of NXP Auto Linux BSP, it is "aarch64-none-elf"

4. Clean working directories:

```
make KERNELDIR=<path to kernel> PLATFORM=aarch64-none-elf drv-clean
```

5. Build the driver:

```
make KERNELDIR=<path to kernel> PLATFORM=aarch64-none-elf all
```

Chapter 4

Usage

4.1   Prerequisites

4.1.1   NXP S32G274A silicon platform EVB or RDB

For usage the software on S32G EVB or RDB boards, the following shall be fullfiled:

- Compatible PFE firmware file.

  > ☞ *The compatibility requirements can be find in the driver public repository:*
  > *https://source.codeaurora.org/external/autobsp32/extra/pfeng*

- Auto Linux BSP version 24.0 or higher for creation of sdcard bootable image

  > ☞ *Note that PFE firmware is being delivered as a standalone product package.*

4.2   Supported development boards

NXP offers two development boards for S32G274A:

1. S32G-VNP-EVB

2. S32G-VNP-RDB

Please refer to board specific user guide to check possible ethernet connectivity. Also check '*NXP Automotive Linux BSP User Manual*' which provides comprehensive information for ethernet controllers, including PFE, with supported configurations for U-boot and Linux.

4.3   Running the driver

1. The *pfeng.ko* driver is embedded in the Auto Linux BSP sdcard image and as such is automatically loaded on Linux startup.

2. To check available network interfaces:

   ```
   root@s32g274aevb:~# ifconfig -a
   ```

   there should be some, based on config, *pfeX* interfaces.

3. Configure the IP addresses and bring the interfaces up by executing:

```
root@s32g274aevb:~# ifconfig pfe<0-2> <interface_ip_address>/<mask>
```

4. The *ping* utility can be used to test the connection:

```
root@s32g274aevb:~# ping <remote_ip_address>
```

4.4   The driver configuration - device tree

The driver is configurable by changes in the device-tree pfeng node. Usually, the device tree files are using multi-level organization, for example the *fsl-s32g274a-evb.dtb* is built by integration:

1. Family DT file *fsl-s32-gen1.dtsi*

2. SoC DT file *fsl-s32g274a.dtsi*

3. Board specific DT file *fsl-s32g274a-evb.dts*

Listing 4.1: PFE device tree node example

```
* NXP S32G274A PFE networking accelerator (pfeng)

Required properties:
- compatible : Should be "fsl,s32g274a-pfeng"
- reg : Address and length of the register set for the device
- interrupts : Should contain all pfeng interrupts: hif0..hif3,nocpy,bmu,
    upegpt,safety
- clocks : Should contain at least: pfe_sys, pfe_pe, xbar
- memory-region : Physical address space for PFE buffers, must be in the
    range 0x00020000 - 0xbfffffff
- phy-mode : See ethernet.txt file in the same directory

Optional properties:
- firmware-name : The name of PFE firmware

Required subnode:
- ethernet : specifies the logical network interface

Requires properties for 'ethernet' subnode:
- compatible : Should be "fsl,pfeng-logif"
- reg : Small number, indexing the network interfaces
- fsl,pfeng-hif-channel : The number of HIF channel (0-3)
- phy-mode : See ethernet.txt file in the same directory
- fsl,pfeng-if-name : Logical interface name visible in the Linux
- fsl,pfeng-emac-id : PFE EMAC id where the interface will be linked to

Optional properties for 'ethernet' subnode:
- local-mac-address : MAC address
- phy-handle : phandle to the PHY device connected to this device.
```

- fixed-link : Assume a fixed link. See fixed-link.txt in the same
    directory.
  Use instead of phy-handle.

Optional subnode for 'ethernet':
- mdio : specifies the mdio bus, used as a container for phy nodes
  according to phy.txt in the same directory

Requires properties for 'mdio' subnode:
- compatible = Should be "fsl,pfeng-mdio"

Example:

```
pfe@46080000 {
        compatible = "fsl,s32g274a-pfeng";
        reg = <0x0 0x46000000 0x0 0x1000000>,    /* PFE controller */
                <0x0 0x4007ca00 0x0 0x4>,        /* S32G274a syscon */
                <0x0 0x83400000 0x0 0xc00000>; /* PFE DDR 12M */
        #address-cells = <1>;
        #size-cells = <0>;
        memory-region = <&pfe_reserved>;
        interrupt-parent = <&gic>;
        interrupts =  <0 190 1>,                   /* hif0 */
                        <0 191 1>,                 /* hif1 */
                        <0 192 1>,                 /* hif2 */
                        <0 193 1>,                 /* hif3 */
                        <0 194 1>,                 /* bmu */
                        <0 195 1>,                 /* nocpy */
                        <0 196 1>,                 /* upe/gpt */
                        <0 197 1>;                 /* safety */
        interrupt-names = "hif0", "hif1", "hif2", "hif3",
                        "bmu", "nocpy", "upegpt", "safety";
        clocks = <&clks S32GEN1_CLK_PFE_SYS>,
                <&clks S32GEN1_CLK_PFE_PE>,
                <&clks S32GEN1_CLK_XBAR>;
        clock-names = "pfe_sys", "pfe_pe", "xbar";
        firmware-name = "s32g_pfe_class.fw";

        /* EMAC 0 */
        pfe0_if: ethernet@0 {
                compatible = "fsl,pfeng-logif";
                #address-cells = <1>;
                #size-cells = <0>;
                reg = <0>;                         /* If id */
                local-mac-address = [ 00 04 9F BE EF 00 ];
                fsl,pfeng-if-name = "pfe0";
                fsl,pfeng-hif-channel = <0>;     /* HIF channel 0 */
                fsl,pfeng-emac-id = <0>;         /* EMAC 0 */
                phy-mode = "sgmii";
                phy-handle = <&anyphy1>;

                /* MDIO on EMAC 0 */
                pfe0_mdio: mdio@0 {
                        /* on EVB occupied by USB ULPI */
                        status = "disabled";
                        compatible = "fsl,pfeng-mdio";
                        #address-cells = <1>;
                        #size-cells = <0>;
```

```
                reg = <0x0>;
        };
};

/* EMAC 1 */
pfe1_if: ethernet@1 {
        compatible = "fsl,pfeng-logif";
        #address-cells = <1>;
        #size-cells = <0>;
        reg = <1>;                          /* If id */
        local-mac-address = [ 00 04 9F BE EF 01 ];
        fsl,pfeng-if-name = "pfe1";
        fsl,pfeng-hif-channel = <1>;        /* HIF channel 1 */
        fsl,pfeng-emac-id = <1>;            /* EMAC 1 */
        phy-mode = "rgmii";
        phy-handle = <&anyphy2>;
        pinctrl-names = "default";
        pinctrl-0 = <&pinctrl0_pfe1_rgmii_c>,
                    <&pinctrl1_pfe1_rgmii_c>;

        /* MDIO on EMAC 1 */
        pfe1_mdio: mdio@0 {
                compatible = "fsl,pfeng-mdio";
                #address-cells = <1>;
                #size-cells = <0>;
                reg = <0x1>;
                pinctrl-names = "default";
                pinctrl-0 = <&pinctrl0_pfe1_mdio_c>,
                            <&pinctrl1_pfe1_mdio_c>;
                anyphy1: ethernet-phy@1 {
                        reg = <1>;
                };
                anyphy2: ethernet-phy@2 {
                        reg = <2>;
                };
        };
};
};
```

Chapter 5

LibFCI

5.1   The libfci

Additional features provided by the PFE can be managed using FCI API. The driver release package contains library sources which can be used to control PFE from custom user's application. Details about usage of the library can be found in *'FCI API Reference'* document.

5.2   Building libfci library

The simplest way, how to get in touch with libfci framework, is using the Auto Linux BSP facitilies. The BSP contains libfci recipe in `meta-alb/recipes-extended` subdirectory, which is intendent for quick jump-in to the FCI development. By default, the libfci is not enabled, so the first step is to create libfci.a library:

```
user@dev:~/fsl-auto-yocto-bsp/build_s32g274aevb$ bitbake -c compile libfci
```

5.3   Entering Yocto development environment

Yocto offers 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 open a terminal in that directory:

```
user@dev:~/fsl-auto-yocto-bsp/build_s32g274aevb$ bitbake -c devshell libfci
```

When invoked, all environmental variables are set up exactly like for compilation. Use predefined *$CC, $CXX, $PATH*. The libfci.a library, which was compiled in previous step, is located in `sw/xfci/libfci/build/release` directory, what can checked by *find* command, inside devshell terminal:

```
root@dev:~/fsl-auto-.../libfci/0.8.0-r0/git# find . -name libfci.a
./sw/xfci/libfci/build/release/libfci.a
```

The same way it can be checked for required header files:

```
root@dev:~/fsl-auto-.../libfci/0.8.0-r0/git# find . -name libfci.h
./sw/xfci/libfci/public/libfci.h
```

The above information is enough to have all necessary bits for libfci-enabled application development.