AN14355

S32G PFE with QNX Hypervisor

Rev. 1.1.0 — 18 July 2024 Application note

Document information

Information	Content
Keywords	QNX, Linux, PFE, QNX Hypervisor, VirtIO-Net
Abstract	How to use PFE within QNX Hypervisor on the S32G platform.

1 Introduction

This is S32G PFE QNX Hypervisor Application Note.

NXP and the NXP logo are trademarks of NXP.

All product or service names are the property of their respective owners.

2 General Description

This Application Note showcases how to use PFE and QNX Hypervisor on S32G platform. It shows how to prepare Host system and several virtualized Guest systems.

The demo presented in this Application Note can be used for running virtualized Guest QNX and Linux.

The demo was tested on *S32G3 Reference Design Board* (S32G-VNP-RDB3). Components of the setup that require building were built on *Ubuntu 22.04*.

3 Overview

The Application Note is designed to showcase two approaches to providing PFE-accelerated network connectivity for the guest systems:

- Indirect connection via VirtIO-Net virtual network bridge.
- Direct connection via PFE driver passthrough (requires PFE Master-Slave configuration, for more information visit PFE QNX Driver User Manual).

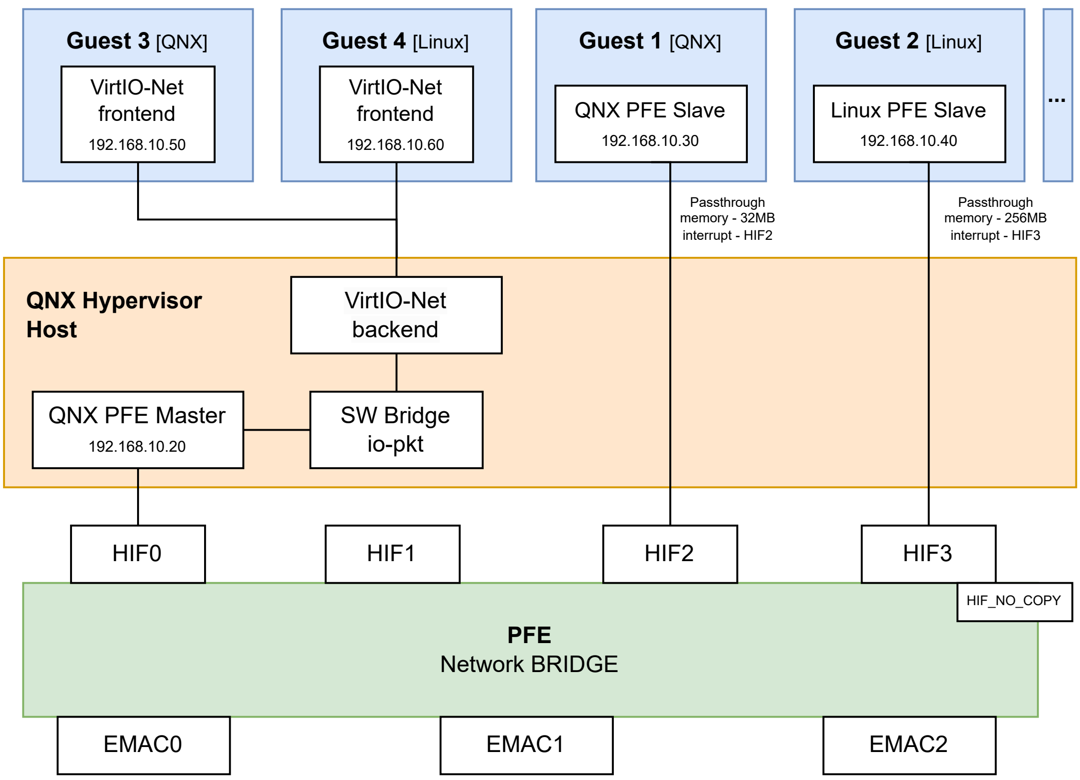


Figure 1. Demo Topology

Table 1. Systems used in the demo

Name	System	Note
Host	QNX SDP 7.1 Hypervisor 2.2	QNX Host OS with PFE Master driver and VirtIO-Net backend in a bridge
Guest 1	QNX SDP 7.1 Guest BSP	QNX Guest with PFE Slave driver connected to HIF2
Guest 2	NXP Automotive Linux BSP 41.0	Linux Guest with PFE Slave driver connected to HIF3
Guest 3	QNX SDP 7.1 Guest BSP	QNX Guest with VirtIO-Net frontend interface
Guest 4	NXP Automotive Linux BSP 41.0	Linux Guest with VirtIO-Net frontend interface

3.1 QNX Hypervisor

QNX Hypervisor is a virtualization technology by BlackBerry® QNX® that lets multiple operating systems run on one device. Key features include:

- Real-time performance - critical tasks run on time.
- Isolation - keeps virtual machines separate for safety.
- Resource management - efficiently shares CPU, memory, and I/O.
- Flexibility - supports different operating systems.
- Safety and security - meets high safety and security standards.

This demo uses QNX Hypervisor to run several operating systems on one S32G platform.

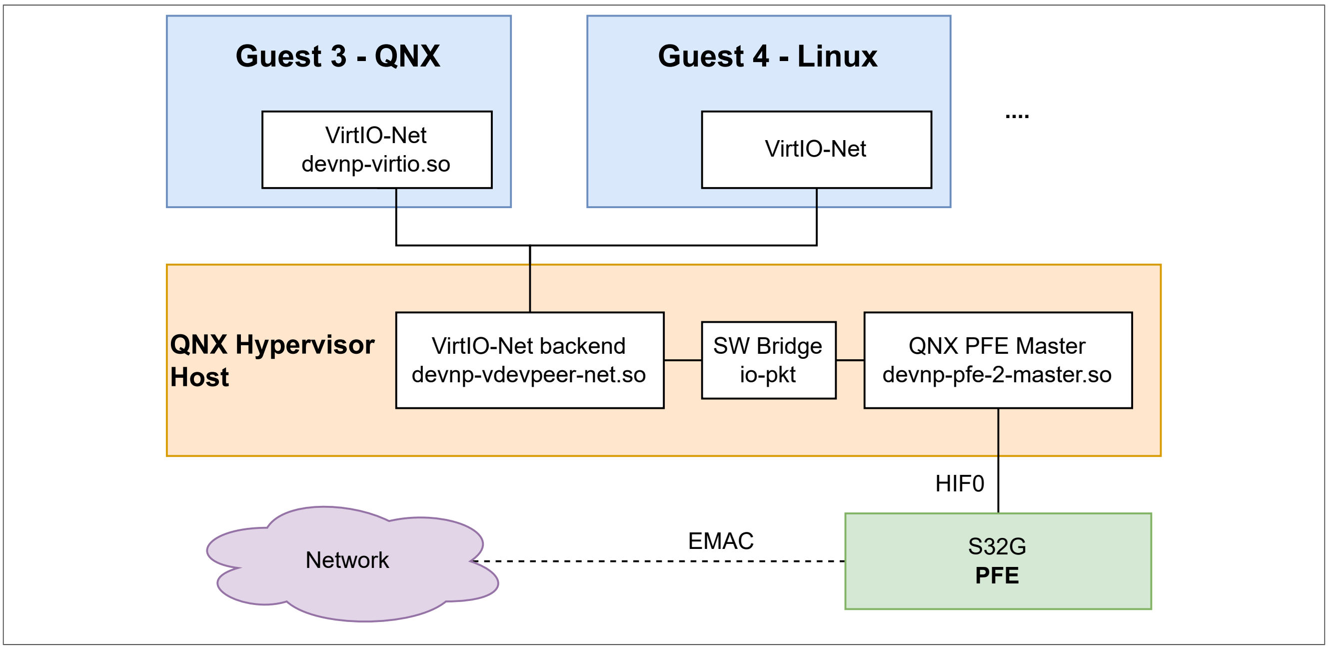
3.2 VirtIO-Net virtual network bridge

QNX Hypervisor has a feature (VirtIO-Net) for virtual network management. The VirtIO-Net can be used to create virtual endpoint interfaces in the io-pkt network stack. It also provides the possibility to create a software bridge between network interfaces, and to establish rules to control the traffic.

With VirtIO-Net, an interface of PFE Driver in Host OS and virtual interfaces of Guest systems can be pooled into a common bridge. This way, the Guest systems can send/receive network traffic.

This setup does not require PFE Multi-client scenario. It also allows to use more Guest systems than the PFE passthrough approach. However, it has slightly worse performance than PFE passthrough approach due to intermediate traffic processing in the software bridge.

For more information about the virtual network bridge, see QNX Hypervisor User's Guide Networking.



3.3 PFE driver passthrough

PFE drivers support Multi-client scenario. In this scenario, multiple PFE drivers simultaneously use the PFE. Typical usecase of this scenario is several operating systems, with each system running its own PFE driver.

To use PFE Multi-client scenario in conjunction with QNX Hypervisor and Guest systems, Hypervisor must be configured to forward memory and interrupts from the Host OS. This is needed because PFE Slave driver[1] needs direct access to PFE peripheral registers. The driver must also allocate memory buffers, which are then accessed by the PFE peripheral.

QNX Hypervisor has the ability to map memory 1:1, so access inside the Guest system can be directly attached to memory of the Host OS. Note that due to PFE HW limitations, memory addresses of PFE-related buffers must be within the **32bit** address range.

In QNX BSP, a custom memory area can be reserved during startup of the system (see QNX BSP Host Hypervisor). The area then stays reserved throughout the runtime and is not available for the OS. Such area can be directly mapped as a part of the Guest system memory, so the PFE driver of the Guest system can allocate (and the PFE peripheral can use) all the required buffers.

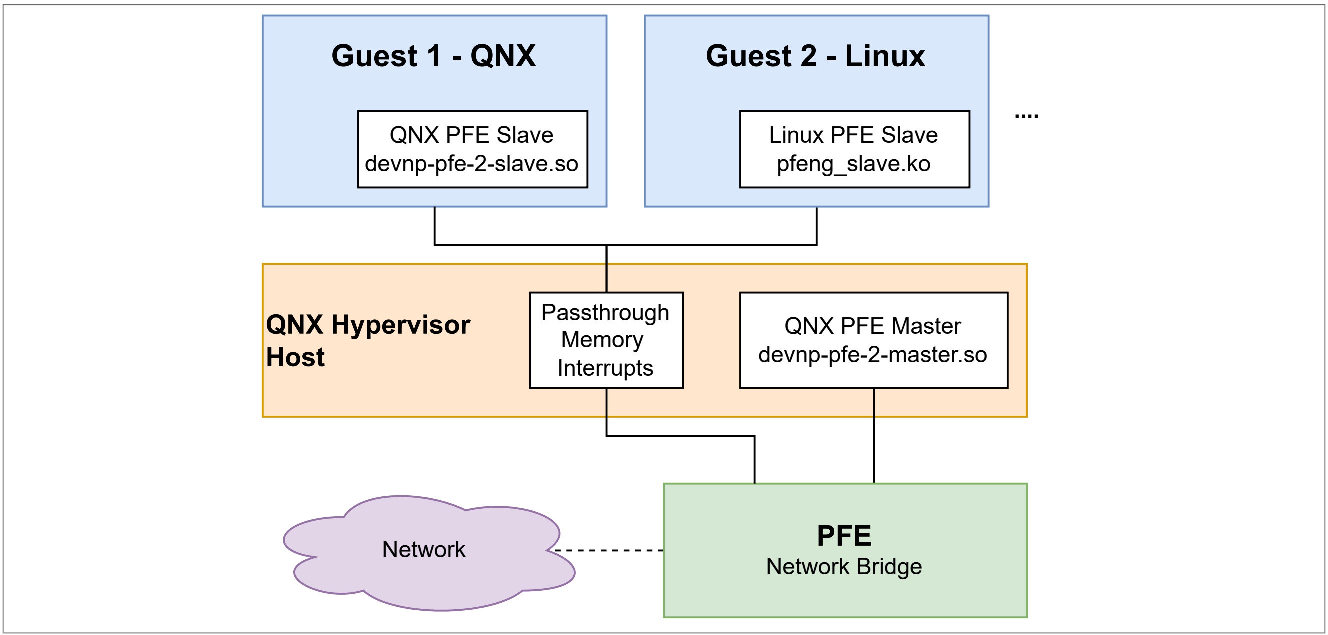
QNX PFE driver can be easily configured to use a particular reserved memory area.

Linux PFE driver uses the general system memory. Therefore, for Linux Guest systems, all the Guest memory must be mapped to the memory of the Host OS. This way, PFE peripheral will have access to every possible allocated memory buffer of Linux PFE driver.

PFE Multi-client scenario supports only up to 4 clients (this is limited by number of PFE HIF channels). However, PFE driver passthrough allows for maximum networking performance, because there is no intermediate software traffic processing.

Note: *PFE has also a HIF_NO_COPY channel. This channel is reserved for AUTOSAR use.*

1 Participant of PFE Multi-client scenario.



3.4 Memory and Interrupt passthrough configuration

For the demo, the memory and interrupts are forwarded to Guest systems as displayed in the following table. Mapped memory address for PFE registers is fixed and can't be modified. Users can change PFE driver memory location in system configuration.

Guest name	Passthrough from Host OS	Guest memory	Interrupt
Guest 1 (QNX)	HIF2 channel 32 MB Memory area PFE registers	256 MB guest virtual memory Passthrough from Host OS memory: * PFE driver 32 MB: 0x96000000 - 0x98000000 * PFE registers: 0x46000000 - 0x47000000 * PFE Master detect flag - 0x4007CAEC	pass intr gic:224 # HIF2 Vector Interrupt
Guest 2 (Linux)	HIF3 channel 256 MB Memory area PFE registers	Passthrough from Host OS memory: * System 256 MB: 0xB0000000 - 0xBFFFFFFF * PFE driver: 0xB0000000 - 0xB1000000 * PFE registers : 0x46000000 - 0x47000000 * PFE Master detect flag - 0x4007CAEC	pass intr gic:225 # HIF3 Vector Interrupt -> GIC_SPI 193 IRQ_TYPE_EDGE_RISING

4 Prerequisites

- S32G-VNP-**RDB3** SCH-53060 Reference Design Board.
- Micro SD card formated with partition FAT32 started from offset 8192 sectors (see SD card example partition).
- QNX Hypervisor Licenses:
  - *QNX Hypervisor - Subscription ver. 2.2 .*
  - *QNX Software Development Platform - Subscription ver. 7.1.0.*
- QNX Hypervisor packages installed from QNX Software center.
- Linux build dependencies for creating Linux image.
- GCC Compiler - aarch64-none-linux-gnu from ARM tools site.
- U-Boot with Hypervisor enabled - section U-Boot with Hypervisor.
- QNX PFE Driver.

- Linux PFE Driver.
- NXP Automotive Linux BSP.

4.1 Hypervisor packages

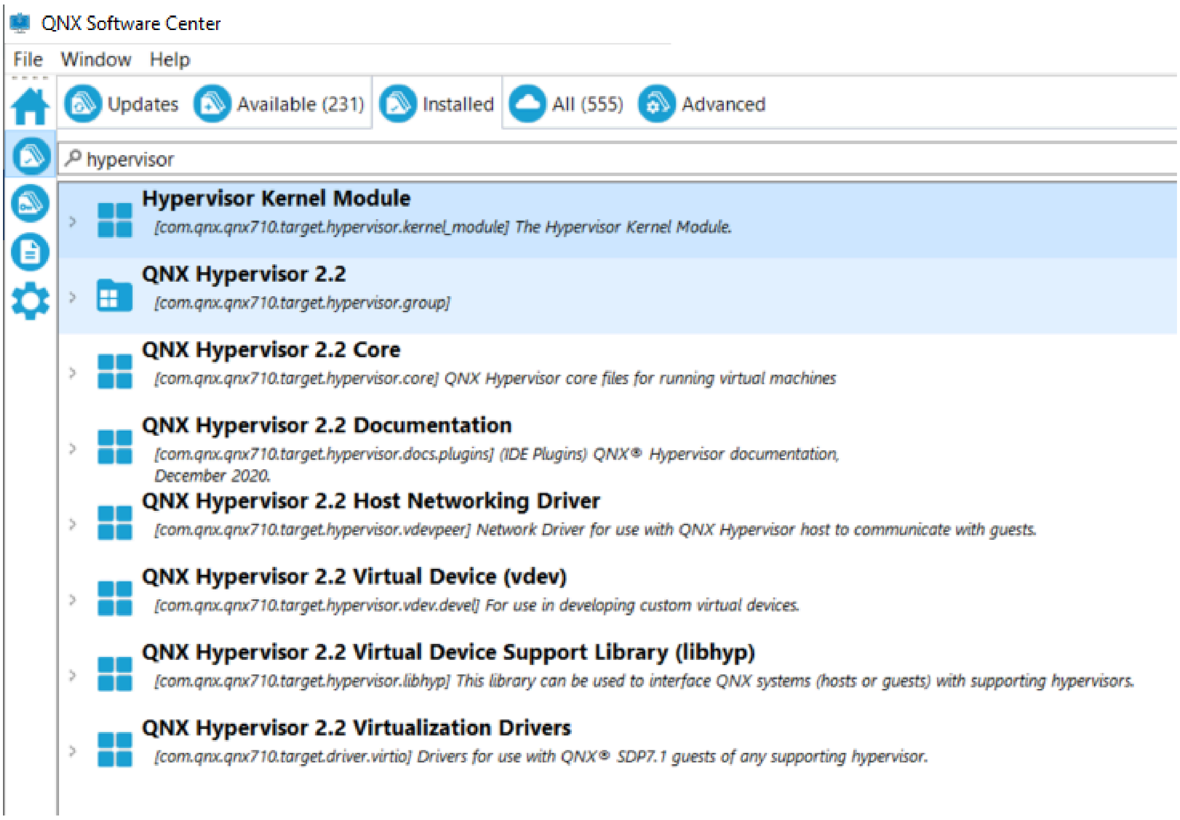


Figure 2. QNX Software Center Hypervisor modules

4.2 Necessary packages to install

In this Application Note, following packages were installed to Ubuntu 22.04 for building proces.

```
apt-get install build-essential device-tree-compiler libssl-dev openssl bc gawk repo flex
```

5 Components

For the demo, the following components must be prepared and built:

Component	Version
U-Boot bootloader	U-Boot BSP37 2020.04
Trusted Firmware-A	ATF BSP37 2.5
QNX® SDP 7.1 BSP for NXP S32G274A EVB	7.1 BuildID 51 - August 15, 2023
QNX® SDP 7.1 BSP for Hypervisor guest (SDP7.1) for generic ARM virtual machine	7.1 BuildID 13 - May 18, 2021

Component	Version
Linux Device Tree Blob for S32G	BSP41 (.dtb) for S32G
NXP Automotive Linux BSP 41.0	BSP41.0 for S32G3
Linux kernel	v5.15.153
PFE Firmware	1.9.0
QNX PFE Master driver	1.6.0
QNX PFE Slave driver	1.6.0
Linux PFE Slave driver (pfeng)	1.7.0

Recommended output file structure is in the table Content of SD card. Follow next sections for preparation of the necessary components.

5.1 U-Boot with Hypervisor support

U-Boot is a bootloader used to boot QNX system and configure PFE peripheral clocks. The system must boot into Exception Level 2 (EL2) to run a virtualized system.

5.1.1 Artifact name

u-boot-nodtb.bin

5.1.2 Where to get

Build it from source code. The source code can be obtained from the nxp-auto-linux/u-boot GitHub repository. Checkout version `bsp37.0-2020.04`.

```
git clone https://github.com/nxp-auto-linux/u-boot
cd u-boot && git checkout release/bsp37.0-2020.04
```

5.1.3 Modifications

It is necessary to enable EL2 Exception Level in U-Boot configuration for QNX Hypervisor to work.

```
make CROSS_COMPILE=aarch64-none-linux-gnu- s32g399ardb3_defconfig
make menuconfig
```

In the section **ARM architecture** select and enable **Enable Xen EL2 Booting...**, as shown on the image U-Boot enable Xen EL2 Booting. Apply changes by **Save** and **Exit**.

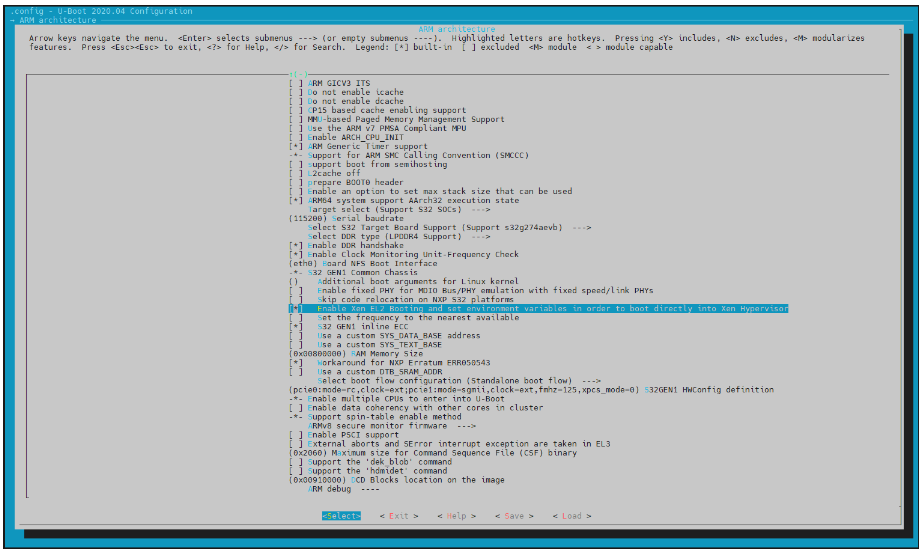


Figure 3. U-Boot enable Xen EL2 Booting

For GMAC network interface in QNX, it is necessary to enable clocks in the file `drivers/net/dwc_eth_qos_s32cc.c` by adding following function on line 202.

```
    setup_clocks_enet_gmac(mode,dev);
```

This patch file can make it easier to modify the file.

drivers/net/dwc_eth_qos_s32cc.c diff patch file

```
diff --git a/drivers/net/dwc_eth_qos_s32cc.c b/drivers/net/dwc_eth_qos_s32cc.c
index b9cfcdcc57..debf4a7f5e 100644
--- a/drivers/net/dwc_eth_qos_s32cc.c
+++ b/drivers/net/dwc_eth_qos_s32cc.c
@@ -202,6 +202,8 @@ static bool s32ccgmac_set_interface(struct udevice *dev, phy_interface_t mode)
        if (!s32cc)
                return false;

+       setup_clocks_enet_gmac(mode,dev);
+
        setup_iomux_enet_gmac(dev, mode);
        s32cc->mac_intf = mode;
```

5.1.4 How to build and deploy

Build U-Boot binary with the following command:

```
make CROSS_COMPILE=aarch64-none-linux-gnu- -j
```

The output file `u-boot-nodtb.bin` will be used in next step to build ATF.

5.1.5 U-Boot example configuration

During first start of U-Boot, it is necessary to setup environment variables. Following variables can be used to boot QNX on S32G RDB3 board:

```
setenv boot_qnx_atf 'mmc dev 0;  scmi_clk gate protocol@14 6 1; scmi_clk gate protocol@14 8 1;
 fatload mmc 0:1 0x83e00000 s32g399a-rdb3.dtb; run atf_fdt_0to3; run atf_fdt_4to7; run release_cpus;
 fatload mmc 0:1 0x80080000 ifs-s32g399a-rdb.ui; pfeng enable; s32ccgmac disable; s32ccgmac enable;
 bootm 0x80080000 - 0x83E00000'
```

```
setenv atf_fdt_0to3 'fdt addr 0x83e00000; fdt resize; fdt set /cpus/cpu@1 cpu-release-addr <0x0
 0xa0000010>; fdt set /cpus/cpu@100 cpu-release-addr <0x0 0xa0000010>; fdt set /cpus/cpu@101 cpu-
release-addr <0x0 0xa0000010>;'
setenv atf_fdt_4to7 'fdt set /cpus/cpu@2 cpu-release-addr <0x0 0xa0000010>; fdt set /cpus/cpu@3 cpu-
release-addr <0x0 0xa0000010>; fdt set /cpus/cpu@102 cpu-release-addr <0x0 0xa0000010>; fdt set /
cpus/cpu@103 cpu-release-addr <0x0 0xa0000010>;'
setenv release_cpus 'run cpu_trap; mp 1 release 0xa0000000; mp 2 release 0xa0000000; mp 3 release
 0xa0000000; mp 4 release 0xa0000000; mp 5 release 0xa0000000; mp 6 release 0xa0000000; mp 7 release
 0xa0000000;'
setenv cpu_trap 'dcache off; mw.l 0xa0000000 0xd503205f; mw.l 0xa0000004 0x58000060; mw.l 0xa0000008
 0xb4ffffc0; mw.l 0xa000000C 0xd61f0000; mw.q 0xa0000010 0x00000000; dcache on;'
setenv bootcmd 'run boot_qnx_atf'
setenv hwconfig 'pcie0:mode=rc,clock=ext;pcie1:mode=sgmii,clock=ext,fmhz=125,xpcs_mode=2G5'
setenv pfeng_mode 'enable,sgmii,sgmii,rgmii'
setenv skip_scmi_reset_agent '1'
saveenv
```

Note: *Make sure to copy only one line at once to serial terminal. Every line starts with setenv command. At the end, command saveenv is used to save the variables.*

5.2 Trusted Firmware-A

5.2.1 Artifact name

fip.s32

5.2.2 Where to get

Build it from source code. The source code can be obtained from the public nxp-auto-linux/arm-trusted-firmware GitHub space. Checkout version `bsp37.0-2.5`.

```
git clone https://github.com/nxp-auto-linux/arm-trusted-firmware.git
cd arm-trusted-firmware && git checkout release/bsp37.0-2.5
```

5.2.3 How to build and deploy

Build ATF with U-Boot binary from component U-Boot with Hypervisor for target platform `s32g3xxaevb3`.

```
make CROSS_COMPILE=aarch64-none-linux-gnu- ARCH=aarch64 PLAT=s32g3xxaevb3 \
    BL33=../u-boot/u-boot-nodtb.bin S32_HAS_HV=1 S32_USE_LINFLEX_IN_BL31=1 LOG_LEVEL=40
```

Write built output binary `fip.s32` to a prepared SD card with the dd commands.

Note: *The SD card device descriptor can be different.*

```
sudo dd if=build/s32g3xxaevb3/release/fip.s32 of=/dev/sdc conv=notrunc seek=0 bs=256 count=1
sudo dd if=build/s32g3xxaevb3/release/fip.s32 of=/dev/sdc conv=notrunc bs=512 seek=1 skip=1
sync
```

5.2.4 Prepared SD card example partition

This SD card partition was used for QNX Hypervisor testing:

```
Disk /dev/sdc: 7,28 GiB, 7817134080 bytes, 15267840 sectors
Disk model: STORAGE DEVICE
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0xb25fe294

Device     Boot Start      End  Sectors  Size Id Type
```

```
/dev/sdc1        8192 15267839 15259648  7,3G  c W95 FAT32 (LBA)
```

5.3 QNX BSP Host Hypervisor

QNX BSP is a set of packages, drivers, and applications that are possible to use in the QNX system. For this usecase, it's necessary to reserve PFE memory and include the Hypervisor binaries and libraries.

5.3.1 Artifact name

ifs-s32g399a-rdb.ui

5.3.2 Where to get

QNX BSP is distributed through QNX Software Center as *QNX® SDP 7.1 BSP for NXP S32G274A EVB*. User must have a valid QNX license for downloading suitable packages - see section Prerequisites. To use the BSP with Hypervisor, it is necessary to make some modifications. Downloaded BSP zip file `BSP_nxp-s32g-evb_br-710_be-710_SVN984052_JBN51.zip` import to QNX Momentics IDE as *QNX Source Package and BSP*. For more information, see the PFE QNX Integration Manual documentation.

5.3.3 Modifications

Reserve custom memory area during system startup for use with QNX Guests and PFE driver. In the file `nxp-s32g-evb\src\hardware\startup\boards\s32g\s32g399a-rdb\main.c` add the following functions:

nxp-s32g-evb\src\hardware\startup\boards\s32g\s32g399a-rdb\main.c:134

```
kprintf("Reserving RAM region for PFE driver on EVB/RDB\n");

/* 96MB - PFE QNX Master */
as_add_containing(0x90000000,0x96000000 - 1, AS_ATTR_RAM, "pfe_ddr","ram");
/* 32MB - PFE QNX Slave */
as_add_containing(0x96000000,0x98000000 - 1, AS_ATTR_RAM, "pfe_ddr_hv","ram");
/* 256MB - PFE Linux Slave */
as_add_containing(0xA0000000,0xB0000000 - 1, AS_ATTR_RAM, "hv_guest1","ram");
/* 256MB - PFE Linux Slave */
as_add_containing(0xB0000000,0xC0000000  -1, AS_ATTR_RAM, "hv_guest2","ram");
```

Next step is to edit the build file `nxp-s32g-evb\images\s32g399a-rdb.build` with bootscript to include QNX Hypervisor module with `module=qvm` and reserve memory as following:

nxp-s32g-evb\images\s32g399a-rdb.build:21

```
[+keeplinked] startup-s32g399a-rdb -P 8 -r 0x8e0000000,0x20000000,1 -r 0x90000000,0x8000000,1 -r
 0xA0000000,0x20000000,1
[+keeplinked module=qvm] PATH=/proc/boot:/bin:/usr/bin:/opt/bin:/sbin:/usr/sbin LD_LIBRARY_PATH=/proc/boot:/
lib:/usr/lib:/lib/dll:/lib/dll/pci:/opt/lib procnto-smp-instr -vvvvv
```

Add necessary libraries at the bottom of this file. They will be included into the output BSP filesystem:

nxp-s32g-evb\images\s32g399a-rdb.build:184

```
###########################################################################
## Hypervisor shared libs
###########################################################################

/sbin/qvm = qvm
/bin/qvm-check = qvm-check
/bin/smmuman = smmuman
libfdt.so
devnp-vdevpeer-net.so
```

nxp-s32g-evb\images\s32g399a-rdb.build:184

```
devnp-virtio.so
vdev-virtio-blk.so
vdev-virtio-console.so
vdev-virtio-net.so
vdev-pl011.so
brconfig
```

Include SSH server configuration. It makes Hypervisor guests more accessible.

nxp-s32g-evb\images\s32g399a-rdb.build:200

```
###########################################################################
## sshd support
###########################################################################
/usr/sbin/sshd=sshd
/usr/bin/scp=scp
/usr/bin/ssh=ssh
/usr/bin/ssh-keygen=ssh-keygen

[uid=0 gid=0 type=dir dperms=0755] /dev/shmem/ssh
[uid=0 gid=0 perms=0644 search=${QNX_TARGET}/etc/ssh] /dev/shmem/ssh/ssh_known_hosts=ssh_known_hosts

[perms=0744] sshd_config = {
Protocol         2
LoginGraceTime     600
PermitRootLogin yes
PermitEmptyPasswords yes
Subsystem       sftp    /usr/libexec/sftp-server
}

/root/.profile = {
export SYSNAME=nto
export TERM=xterm
export PATH=/proc/boot:/sbin:/bin:/usr/bin:/opt/bin/sbin:/usr/sbin
export LD_LIBRARY_PATH=/proc/boot:/lib:/usr/lib:/lib/dll:/lib/dll/pci
export PCI_HW_MODULE=/lib/dll/pci/pci_hw-fdt.so
export PCI_BKWD_COMPAT_MODULE=/lib/dll/pci/pci_bkwd_compat.so
export PCI_SLOG_MODULE=/lib/dll/pci/pci_slog2.so
export PCI_DEBUG_MODULE=/lib/dll/pci/pci_debug2.so
export PCI_BASE_VERBOSITY=2
}

[uid=0 gid=0 type=dir dperms=0755] /var/chroot/sshd
[type=link] /etc/ssh = /dev/shmem/ssh
[type=link] /var/etc/ssh = /dev/shmem/ssh
```

Modify install target in `nxp-s32g-evb\Makefile` to ensure all binaries and libraries from BSP build are copied to install folder:

nxp-s32g-evb\Makefile:32

```
install: $(if $(wildcard prebuilt/*),prebuilt)
    $(MAKE) -Csrc hinstall
    $(MAKE) -Csrc install
```

5.3.4 How to build and deploy

Build this BSP project with *QNX Momentics IDE* and copy the output file `nxp-s32g-evb\images\ifs-s32g399a-rdb.ui` to the root of the prepared SD card.

Note: *Make sure that during boot, there is a message 'Reserving RAM region for PFE driver on EVB/RDB' in the output log. If there is no such message, then make sure the* `nxp-s32g-evb\Makefile` *is properly modified (see section* *Modifications**) and then rebuild the project.*

5.4 QNX BSP Guest

QNX Guest BSP is prepared virtual BSP to use with QNX Hypervisor. It is necessary to reserve the memory for PFE driver during system startup.

5.4.1 Artifact name

qnx710-guest.ifs

5.4.2 Where to get

QNX Guest BSP is distributed through QNX Software Center as *QNX® SDP 7.1 BSP for Hypervisor guest (SDP7.1) for generic ARM virtual machines*. To use it with PFE driver, it needs some modifications.

Downloaded zip file import to QNX Momentics IDE as *QNX Source Package and BSP* the same way as in QNX BSP Host Hypervisor. New project with name `hypervisor-guest-armv7` should be created.

5.4.3 Modifications

Reserve memory during guest startup for use with PFE Slave driver. In the file `hypervisor-guest-armv7\src\hardware\startup\boards\armv8_fm\main.c` add following information:

hypervisor-guest-armv7\src\hardware\startup\boards\armv8_fm\main.c:176

```
kprintf("Reserving RAM region for PFE driver on Hypervisor Guest\n");
as_add_containing(0x96000000,0x98000000 - 1, AS_ATTR_RAM, "pfe_ddr","ram");
```

In the file `hypervisor-guest-armv7\images\build` do the following changes to disable PCI server:

hypervisor-guest-armv7\images\build

```
# Disable pci-server by commenting out these lines
52:         #pci-server --bus-scan-limit=0
53:         #waitfor /dev/pci
```

```
# Add libpci libraries
131: libpci.so
132: libpci.so.2.3
```

It is also possible to include other applications or libraries by modifying this build file, the same way as in QNX BSP Host Hypervisor.

5.4.4 How to build and deploy

Build this BSP project with QNX Momentics IDE and copy the output file `hypervisor-guest-armv7\images\qnx710-guest.ifs` into the folder with Guest 1 and Guest 3 QNX system on SD card.

Note: *Make sure that during Guest 1 start, there is a message 'Reserving RAM region for PFE driver on Hypervisor Guest' in the output log. If it doesn't show the message, modify the Makefile the same way as in QNX BSP Host Hypervisor.*

5.5 Linux Guest - File System

This file system is used by Linux Guests. It is customized for S32G3 RDB3 board and can be utilized in QNX Hypervisor.

5.5.1 Artifact name

fsl-image-base-s32g399ardb3.ext4

5.5.2 Where to get

Build it from source code. The source code is a part of [nxp-auto-linux/auto_yocto_bsp] repository.

5.5.3 How to build and deploy

Use `repo` utility to obtain auto_yoct_bsp project.

```
repo init -u https://github.com/nxp-auto-linux/auto_yocto_bsp -b release/bsp41.0
repo sync
```

Initialize environment for S32G3 RDB3 board build.

```
. nxp-setup-alb.sh -m s32g399ardb3
```

Add user-specific modifications (if any) into build configuration file `conf/local.conf`. By default, no additional modifications are needed.

Start the build.

```
bitbake fsl-image-base
```

Note: *Build operation is very CPU / resource intensive. It can run for several hours. Use a powerful computer to shorten the build time.*

The output File System image is located in `build_s32g399ardb3/tmp/deploy/images/s32g399ardb3/fsl-image-base-s32g399ardb3.ext4`. Copy the image to the SD card with other necessary files for Guest 2 and Guest 4.

Note: *It is also possible to use a generic Yocto Poky filesystem image for ARM64 QEMU machines.*

5.6 Linux Guest - Kernel

5.6.1 Artifact name

Image

5.6.2 Where to get

Use the kernel from step Linux Guest - File System, but apply additional modifications (see below).

The kernel sources are located in the folder `build_s32g399ardb3/tmp/work-shared/s32g399ardb3/kernel-source/`.

5.6.3 Modifications

To make the NXP Linux Kernel work correctly in QNX Hypervisor, it is necessary to disable the workaround for the NXP ERR050481 erratum.

Go to the kernel folder and execute the following commands:

```
make ARCH=arm64 CROSS_COMPILE=aarch64-none-linux-gnu- clean
make ARCH=arm64 CROSS_COMPILE=aarch64-none-linux-gnu- defconfig
```

```
# This command invokes a GUI configuration tool.
make ARCH=arm64 CROSS_COMPILE=aarch64-none-linux-gnu- menuconfig
```

In the configuration tool, enter the second section (`Platform selection`). In the section, uncheck the last option (`Enable workaround for the NXP ERR050481 erratum`). Save the configuration and exit the tool.

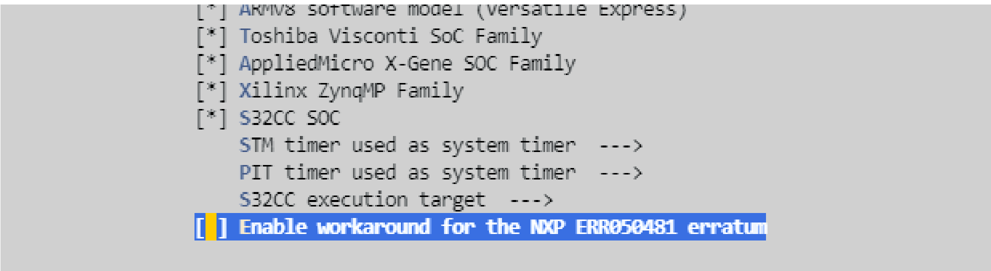


Figure 4. Disable workaround for the ERR050481

5.6.4 How to build and deploy

```
make ARCH=arm64 CROSS_COMPILE=aarch64-none-linux-gnu- -j
```

The output kernel file is located at `build_s32g399ardb3/tmp/work-shared/s32g399ardb3/kernel-source/arch/arm64/boot/Image`. Copy the file to the SD card Guest 2 and Guest 4 folders.

5.7 Device Tree Blob for S32G3 RDB3

Device Tree Blob (DTB) file is necessary for booting a QNX system on S32G platform. U-Boot is using this file for PFE clock configuration and board setup.

5.7.1 Artifact name

s32g399a-rdb3.dtb

5.7.2 Where to get

This component is a part of NXP Linux kernel build artifacts. It can be built from source, or used from previous step Linux BSP Guest.

Copy DTB file `build_s32g399ardb3/tmp/deploy/images/s32g399ardb3/s32g399a-rdb3.dtb` to the root folder of the SD card.

5.8 QNX PFE Driver

QNX PFE driver is used as a library for io-pkt network stack.

5.8.1 Artifact name

devnp-pfe-2-master.so

devnp-pfe-2-slave.so

5.8.2 Where to get

Build the QNX driver from source code. The archive is hosted on NXP FlexNet (nxp.flexnetoperations.com), under the *Automotive SW – S32G Standard Software*, under the product *Automotive SW - S32G - PFE Driver + Standard Firmware*. Download and extract zip file with PFE driver code `PFE-DRV_S32G_A53_QNX_1.6.0.zip`.

5.8.3 How to build and deploy

It is necessary to build:

- QNX PFE Master driver to be run on QNX Host system and connected to *PFE HIF0* channel.
- QNX PFE Slave driver to be run on QNX Guest system and connected to *PFE HIF2* channel.
- LibFCI CLI for configuration of PFE peripheral.

To setup QNX build environment, run the `qnxsdp-env` script in terminal before building the driver. The default path is following:

```
C:\Users\__USERNAME__\qnx710\target\qnxsdp-env.bat
```

QNX PFE Master

```
cd sw/devnp-pfe-2/
make PLATFORM=aarch64le clean -j
make PLATFORM=aarch64le PFE_CFG_HIF_DRV_MODE=1 PFE_CFG_PFE0_IF=6 PFE_CFG_PFE1_IF=6 \
 PFE_CFG_PFE2_IF=6 PFE_CFG_MULTI_INSTANCE_SUPPORT=1 PFE_CFG_PFE_MASTER=1 PFE_CFG_MASTER_IF=6 \
 PFE_CFG_AUX_INTERFACE=1 PFE_CFG_HIF_NOCPY_SUPPORT=0 PFE_CFG_PFE0_PROMISC=0 \
 PFE_CFG_PFE1_PROMISC=0 PFE_CFG_PFE2_PROMISC=0 ARTIFACT_JENKINS=devnp-pfe-2-master.so -j
```

Copy file `sw/devnp-pfe-2/build/aarch64le-release/devnp-pfe-2-master.so` to root of the SD card.

QNX PFE Slave

```
cd sw/devnp-pfe-2/
make PLATFORM=aarch64le clean -j
make PLATFORM=aarch64le PFE_CFG_HIF_DRV_MODE=1 PFE_CFG_PFE0_IF=8 PFE_CFG_PFE1_IF=8 \
 PFE_CFG_PFE2_IF=8 PFE_CFG_MULTI_INSTANCE_SUPPORT=1 PFE_CFG_PFE_MASTER=0 PFE_CFG_MASTER_IF=6 \
 PFE_CFG_AUX_INTERFACE=1 PFE_CFG_HIF_NOCPY_SUPPORT=0 PFE_CFG_PFE0_PROMISC=0 \
 PFE_CFG_PFE1_PROMISC=0 PFE_CFG_PFE2_PROMISC=0 ARTIFACT_JENKINS=devnp-pfe-2-slave.so -j
```

Copy file `sw/devnp-pfe-2/build/aarch64le-release/devnp-pfe-2-slave.so` to folder with the Guest 1 on the SD card.

LibFCI CLI

```
cd sw/libfci_cli/
make PLATFORM=aarch64le clean
make PLATFORM=aarch64le -j20
```

Copy file `sw/libfci_cli/build/aarch64le-release/libfci_cli` to root of the SD card.

5.9 Linux PFE Driver (pfeng)

Linux PFE driver is used as a kernel module. It must be used only with the kernel, which was linked during build.

5.9.1 Artifact name

pfeng-slave.ko

5.9.2 Where to get

Build the Linux driver from source code. The source code can be obtained from the public nxp-auto-linux/pfeng GitHub space. Checkout `release/linux_1.7.0`.

```
git clone https://github.com/nxp-auto-linux/pfeng.git
cd pfeng && git checkout release/linux_1.7.0
```

5.9.3 How to build and deploy

It is necessary to provide path for the Linux kernel, which is used with guest system from section Linux Guest Kernel. Without this kernel, the PFE driver will not be able to load.

```
cd sw/linux-pfeng/
make KERNELDIR=../../build_s32g399ardb3/tmp/work-shared/s32g399ardb3/kernel-source
 PFE_CFG_MULTI_INSTANCE_SUPPORT=1 \
 PFE_CFG_PFE_MASTER=0 PFE_CFG_FCI_ENABLE=0 PLATFORM=aarch64-none-linux-gnu all
```

Copy the file `sw/linux-pfeng/pfeng-slave.ko` to the folder with the Guest 2 on the SD card.

Note: *For more information, follow the instructions in Linux PFE Driver User Manual chapter Building the driver, Use the instructions to build PFE **Slave** driver.*

5.10 Linux Guest DTB for Hypervisor

PFE Linux driver needs a suitable *Device Tree Blob* for a valid configuration of PFE peripheral. QNX Hypervisor can provide this DTB as an extension to the default one for the Guest machine.

5.10.1 Artifact name

s32g-pfe-hv.dtb

5.10.2 Where to get

This is the minimal Device Tree configuration to use with PFE Linux Slave driver and to communicate with HIF3 interface. Save following code as a Device Tree Source file (s32g-pfe-hv.dts):

s32g-pfe-hv.dts

```
/dts-v1/;

/memreserve/ 0xB0000000 0x10000;

/ {
 #address-cells = <2>;
 #size-cells = <2>;

 pfesl_reserved_bdr: pfebufs@B0000000 {
  compatible = "nxp,s32g-pfe-bdr-pool";
  reg = <0 0xB0000000 0 0x10000>;
  status = "okay";
 };

    vgic: interrupt-controller@2c001000 {
        qvm,vdev = "gic";
        #interrupt-cells = <3>;
        #address-cells = <2>;
        interrupt-controller;
```

s32g-pfe-hv.dts

```
    };

    pfe_slave: pfe_slave@46000000 {
        compatible = "nxp,s32g-pfe-slave";
        status = "okay";
        reg = <0x0 0x46000000 0x0 0x1000000>,
                <0x0 0x4007ca00 0x0 0x100>;
        reg-names = "pfe-cbus", "s32g-main-gpr";
        #address-cells = <1>;
        #size-cells = <0>;
        interrupt-parent = <&vgic>;
        interrupts = <0 193 1>;  //  <GIC_SPI 193 IRQ_TYPE_EDGE_RISING>;
        interrupt-names = "hif3";
        clock-names = "pfe_sys", "pfe_pe", "pfe_ts";
        dma-coherent;
        memory-region = <&pfesl_reserved_bdr>;
        memory-region-names = "pfe-bdr-pool";
        nxp,pfeng-ihc-channel = <3>; // PFE_HIF_CHANNEL_3
        nxp,pfeng-master-channel = <0>; // PFE_HIF_CHANNEL_0

        /* Network interface 'pfe0sl' */
        pfesl_netif0: ethernet@100 {
            compatible = "nxp,s32g-pfe-netif";
            status = "okay";
            reg = <100>;
            nxp,pfeng-netif-mode-mgmt-only;
            local-mac-address = [ 00 04 9F BE FF 00 ];
            nxp,pfeng-if-name = "pfe0sl";
            nxp,pfeng-hif-channels = <3>;
            nxp,pfeng-linked-phyif = <0>;
        };

        /* Network interface 'pfe1sl' */
        pfesl_netif1: ethernet@101 {
            compatible = "nxp,s32g-pfe-netif";
            status = "okay";
            reg = <101>;
            nxp,pfeng-netif-mode-mgmt-only;
            local-mac-address = [ 00 04 9F BE FF 01 ];
            nxp,pfeng-if-name = "pfe1sl";
            nxp,pfeng-hif-channels = <3>;
            nxp,pfeng-linked-phyif = <1>;
        };

        /* Network interface 'pfe2sl' */
        pfesl_netif2: ethernet@102 {
            compatible = "nxp,s32g-pfe-netif";
            status = "okay";
            reg = <102>;
            nxp,pfeng-netif-mode-mgmt-only;
            local-mac-address = [ 00 04 9F BE FF 02 ];
            nxp,pfeng-if-name = "pfe2sl";
            nxp,pfeng-hif-channels = <3>;
            nxp,pfeng-linked-phyif = <2>;
        };

        /* Network interface 'aux0sl' */
        pfesl_aux0: ethernet@103 {
            compatible = "nxp,s32g-pfe-netif";
            status = "okay";
            reg = <103>;
            local-mac-address = [ 00 04 9F BE FF 80 ];
            nxp,pfeng-if-name = "aux0sl";
            nxp,pfeng-hif-channels = <3>;
            nxp,pfeng-netif-mode-aux;
        };

        /* Network interface 'hif0sl' */
        pfesl_hif0: ethernet@104 {
            compatible = "nxp,s32g-pfe-netif";
            status = "okay";
            reg = <104>;
            nxp,pfeng-netif-mode-mgmt-only;
            local-mac-address = [ 00 04 9F BE FF F0 ];
            nxp,pfeng-if-name = "hif0sl";
            nxp,pfeng-hif-channels = <3>;
            nxp,pfeng-linked-phyif = <6>;
        };
    };
};
```

5.10.3 How to build and deploy

To build Device Tree Blob, use the following command:

```
dtc -I dts -O dtb -o s32g-pfe-hv.dtb s32g-pfe-hv.dts
```

Copy output `s32g-pfe-hv.dtb` file to SD card folder with Linux PFE Guest 2.

5.11 PFE Firmware

PFE Firmware is a software component running within the PFE peripheral and is processing each packet reaching PFE.

Firmware binary file must be provided to the PFE peripheral by the PFE Master driver during every initialization.

5.11.1 Artifact name

s32g_pfe_class.fw

s32g_pfe_util.fw

5.11.2 Where to get

The archive with built binaries is hosted on NXP FlexNet (nxp.flexnetoperations.com), under the *Automotive SW – S32G Standard Software*, under the product *Automotive SW - S32G - PFE Driver + Standard Firmware*.

Download file `PFE-FW_S32G_1.9.0.zip`.

5.11.3 How to build and deploy

Extract zip file and copy following firmware binary files from to the root of the SD card:

- `s32g_pfe_class.fw`
- `s32g_pfe_util.fw`

6 Hypervisor Guests configurations

It is necessary to create configuration for each of the guests in the topology (as presented in Overview section) with QNX Hypervisor configuration qvmconf file.

Save content of the following sections with configuration to files on the SD card inside guest folders.

Guest name	Memory	Components	Filename
Guest 1 QNX PFE	256 MB system virtual memory 32 MB driver memory	QNX BSP Guest - qnx710-guest.ifs pl011, virtio-console - console virtio-blk - SD card files QNX Slave driver HIF2	*qnx-hv-pfe.qvmconf*
Guest 2 Linux PFE	256 MB system mapped memory	Image - kernel image file s32g-pfe-hv.dtb - Device Tree Blob pl011, virtio-console - console virtio-blk - fsl-image-base-s32g399ardb3.ext4 virtio-blk - SD card files Linux Slave driver HIF3	*linux-hv-pfe.qvmconf*

Guest name	Memory	Components	Filename
Guest 3 QNX VirtIO-Net	256 MB system virtual memory	QNX BSP Guest - qnx710-guest.ifs pl011, virtio-console - console virtio-net - network interface	*qnx-hv-virtio.qvmconf*
Guest 1 Linux VirtIO-Net	256 MB system virtual memory	Image - kernel image file pl011, virtio-console - console virtio-blk - fsl-image-base-s32g399ardb3.ext4	*linux-hv-virtio.qvmconf*

6.1 Guest 1 configuration

This guest uses the **QNX** system from file *qnx710-guest.ifs* (QNX BSP Guest) with direct **PFE** memory mapping and interrupt **passthrough** (for more info see table Memory and Interrupt passthrough configuration). It uses PFE QNX Slave driver connected to *HIF2* channel (QNX PFE Driver).

To control the guest with console interface, it is possible to use *virtio-console* interface from the QNX Host. User can also attach an SD card file system via *virtio-blk* interface to transfer files, so it is not necessary to include PFE driver inside BSP build. QNX Hypervisor can allocate 256 MB of virtual system memory and forward passthrough 32MB of Host OS memory for PFE driver use.

qnx-hv-pfe.qvmconf

```
ram 0x80000000,256M

load qnx710-guest.ifs

# UART console for startup phase
vdev pl011
    hostdev >-
    loc 0x1c090000
    intr gic:37

# Use virtio as the main console
vdev virtio-console
    loc 0x20000000
    intr gic:42

#SD card for testing
vdev virtio-blk
    loc 0x1c0d0000
    intr gic:41
    hostdev /dev/sd0
    name virtio-sd_card

###   PFE
# PFE registers memory
pass loc mem:0x46000000,0x1000000,rw=0x46000000
# Master-detect signalization 0x4007CAECU
pass loc mem:0x4007CAEC,0x4,r=0x4007CAEC

# PFE driver memory
pass loc mem:0x96000000,0x2000000,m=0x96000000

# PFE interrupts
#pass intr gic:222      # HIF0 Vector Interrupt
#pass intr gic:223      # HIF1 Vector Interrupt
pass intr gic:224       # HIF2 Vector Interrupt
#pass intr gic:225      # HIF3 Vector Interrupt
```

6.2 Guest 2 configuration

This guest is using the **Linux** system from file *fsl-image-base-s32g399ardb3.ext4* and kernel binary from file *Image*, with direct memory mapping and interrupt **passthrough** (for more info see table Memory and Interrupt passthrough configuration).

It uses **PFE** Linux Slave driver (Linux PFE driver) with direct PFE registers access and interrupt passthrough from *HIF3* and Device Tree configuration built in step Linux Guest DTB for Hypervisor. All of the system memory is mapped to the Host OS reserved RAM area with the name `hv_guest2`, started at physical memory address `0xB0000000` and size of 256 MB.

To control the guest with console interface, it is possible to use *virtio-console* interface from the QNX Host. User can also attach an SD card file system via *virtio-blk* interface to transfer files, so it is not necessary to include PFE driver inside BSP build.

linux-hv-pfe.qvmconf

```
pass loc mem:$asinfo_start{hv_guest2},$asinfo_length{hv_guest2},rwcm

# kernel
load Image

cmdline "console=ttyAMA0 earlycon=pl011,0x1c090000 rw rootfstype=ext4 root=/dev/vdb"

# Device-Tree
fdt load ./s32g-pfe-hv.dtb

# UART
vdev pl011
 loc 0x1c090000
 intr gic:37

# ROOT FS
vdev virtio-blk
 loc 0x1c0c0000
 intr gic:41
 hostdev fsl-image-base-s32g399ardb3.ext4

# SD card for testing
vdev virtio-blk
 loc 0x1c0d0000
 intr gic:42
 hostdev /dev/sd0
 name virtio-sd_card

###   PFE
# PFE registers memory
pass loc mem:0x46000000,0x1000000,rw=0x46000000
# Master-detect signalization 0x4007CAECU
pass loc mem:0x4007C400,0x100,rw=0x4007C400

# PFE interrupts
#pass intr gic:222       # HIF0 Vector Interrupt
#pass intr gic:223       # HIF1 Vector Interrupt
#pass intr gic:224       # HIF2 Vector Interrupt
pass intr gic:225        # HIF3 Vector Interrupt
```

6.3 Guest 3 configuration

This guest is also using the **QNX** system from file *qnx710-guest.ifs* (QNX BSP Guest) with **VirtIO-Net** interface.

To control of virtual machine via console, it is also used *virtio-console* interface from the QNX Host. It is not needed to attach any SD card, as virtio drivers are included in QNX Guest BSP. Guest does not need to passthrough any resources from the Host system.

The Guest name for better access from the Host OS is set to `qnx-guest`. Hypervisor allocates 256 MB of virtual system memory.

qnx-hv-virtio.qvmconf

```
system qnx-guest

ram 0x80000000,256M

load qnx710-guest.ifs

# UART console for startup phase
```

qnx-hv-virtio.qvmconf

```
vdev pl011
    hostdev >-
    loc 0x1c090000
    intr gic:37

# Use virtio as the main console
vdev virtio-console
    loc 0x20000000
    intr gic:42

# VirtIO-NET interface
vdev virtio-net
    loc 0x1c0e0000
    intr gic:40
    mac aa:aa:aa:aa:aa:aa
    name p2p_qnx
    peer /dev/vdevpeers/vp0
```

6.4 Guest 4 configuration

This guest is also using the same **Linux** file system and kernel as Guest 2 Linux but with **VirtIO-Net** interface.

To control of virtual machine via console, it is also used *virtio-console* interface from the QNX Host. It is not needed to attach any SD card, as virtio drivers are already included. Guest doesn't need to passthrough any resources from the Host system.

The Guest name for better access from the Host OS is set to `linux-guest`. Hypervisor allocates 256 MB of virtual system memory.

linux-hv-virtio.qvmconf

```
system linux-guest

ram 0x80000000,256M

# kernel
load Image

cmdline "console=ttyAMA0 earlycon=pl011,0x1c090000 rw rootfstype=ext4 root=/dev/vda"

# UART
vdev pl011
    loc 0x1c090000
    intr gic:37

# ROOT FS
vdev virtio-blk
    loc 0x1c0c0000
    intr gic:41
    hostdev fsl-image-base-s32g399ardb3.ext4

# PFE
vdev virtio-net
    loc 0x1c0e0000
    intr gic:40
    mac aa:aa:aa:aa:bb:cc
    name p2p_linux
    peer /dev/vdevpeers/vp1
```

7 Running the QNX Hypervisor

Summary of the steps to start the topology descibed in Overview:

1. Prepare and copy all of the files to the SD card
2. Run QNX Host system
3. Setup SSH connection using GMAC network interface
4. Start QNX PFE Master driver and create VirtIO-Net endpoints

5. Configure software bridge between PFE Master and VirtIO-Net network.
6. Start Guest 1 with PFE QNX
7. Start Guest 2 with PFE Linux
8. Start Guest 3 with VirtIO-Net QNX
9. Start Guest 4 with VirtIO-Net Linux
10. Test communication

7.1 Prepare the SD Card

The SD card must be prepared with one FAT32 which starts at offset of 8192 sectors. Write the ATF to the beginning of the SD card (to the offset) as described in section Trusted Firmware-A.

```
sudo dd if=build/s32g3xxaevb3/release/fip.s32 of=/dev/sdc conv=notrunc seek=0 bs=256 count=1
sudo dd if=build/s32g3xxaevb3/release/fip.s32 of=/dev/sdc conv=notrunc bs=512 seek=1 skip=1
```

Note: *The SD card can have different file descriptor then /dev/sdc.*

Copy all of the prepared components to the new SD card FAT32 partition. Final files structure can look like this:

Content of SD card FAT32 partition

```
├─ hv/guest1/
│   ├─ devnp-pfe-2-slave.so
│   ├─ qnx710-guest.ifs
│   └─ qnx-hv-pfe.qvmconf
│
├─ hv/guest2/
│   ├─ fsl-image-base-s32g399ardb3.ext4
│   ├─ Image
│   ├─ linux-hv-pfe.qvmconf
│   ├─ pfeng-slave.ko
│   └─ s32g-pfe-hv.dtb
│
├─ hv/guest3/
│   ├─ qnx710-guest.ifs
│   └─ qnx-hv-virtio.qvmconf
│
├─ hv/guest4/
│   ├─ fsl-image-base-s32g399ardb3.ext4
│   ├─ Image
│   └─ linux-hv-virtio.qvmconf
│
├─ devnp-pfe-2-master.so
├─ ifs-s32g399a-rdb.ui
├─ libfci_cli
├─ s32g_pfe_class.fw
├─ s32g_pfe_util.fw
└─ s32g399a-rdb3.dtb
```

7.2 Run QNX Host

After the SD card is inserted to the S32G3 RDB3 board and the board is turned ON the user can see U-Boot console output. At the first boot, it is necessary to pause the boot process and insert the environment configuration from section U-Boot with Hypervisor line by line.

After reboot, the QNX Host system should start (section QNX BSP Host Hypervisor) and the console should write *"Reserving RAM region for PFE driver on EVB/RDB"*. For more information, visit PFE QNX Integration Manual documentation.

To mount the SD card with all of the files use this command:

Note: *The partition might have a different descriptor than /dev/sd0t*.*

```
mount -t dos /dev/sd0t12 /sdcard
```

Verify that all necessary files are available.

```
ls -l /sdcard/
```

7.3 SSH Connection

QNX user console can't provide multiple sessions, so it is necessary to find a different way to connect and run more Guest virtual machines. The best way is to have multiple active connections through the SSH terminal. For that the GMAC interface can be used.

Check if the GMAC driver is running and set the IP address.

```
ifconfig dwc0 192.168.2.20
```

Note: *It is possible to set any IP addres from the same network subnet as user's PC.*

For SSH connection, it is necessary to generate an RSA key, use the right configuration and start SSH server.

```
ssh-keygen -t rsa -b 2048 -f /etc/ssh/ssh_host_rsa_key -N ''
cp /proc/boot/sshd_config /etc/ssh/
/usr/sbin/sshd
```

It is not possible to connect the computer directly to the GMAC interface (highlighted in the image [S32G-RDB3 board](#)) and connect with IP address using PuTTy or another SSH terminal to network port 22.

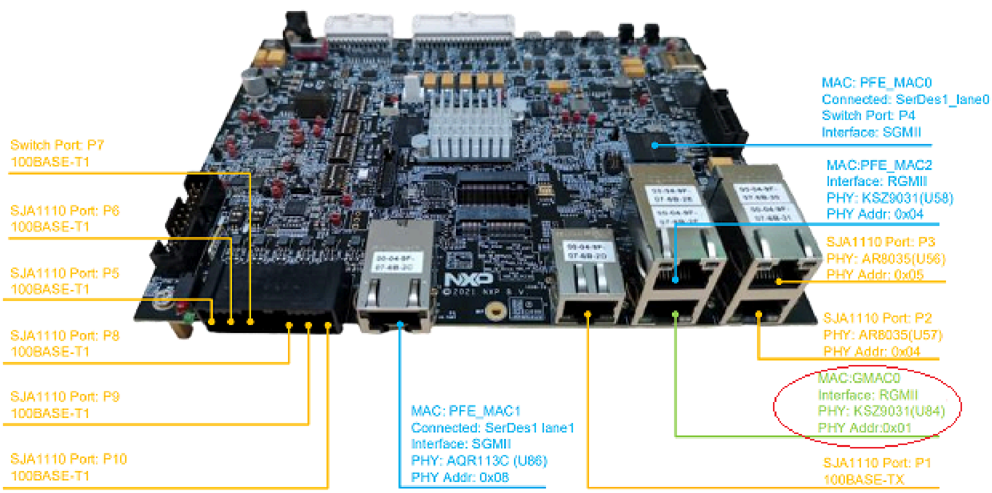


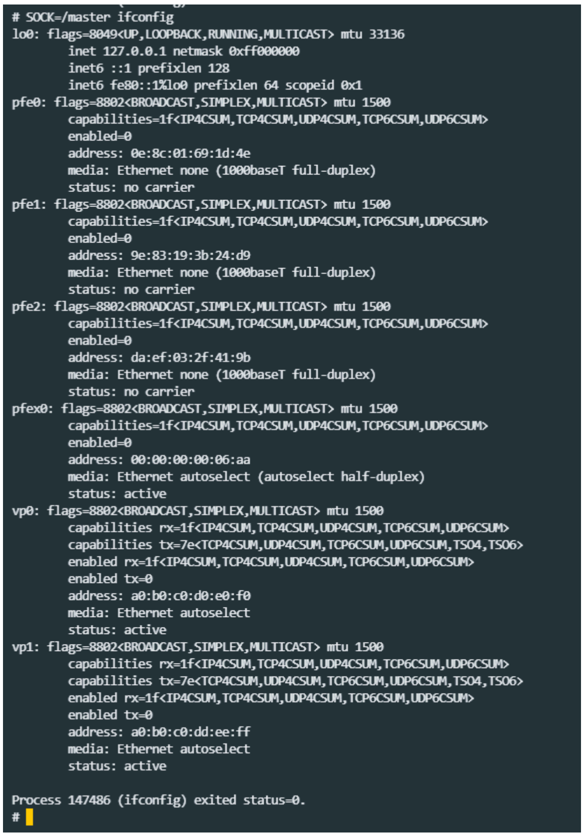
Figure 1. S32G-VNP-RDB3 Ethernet Ports Overview

7.4 Start the QNX Master driver with sw bridge

When the SSH connection is established and the QNX Host is prepared, it is possible to start *PFE QNX Master* driver with vdevpeer-net virtual interfaces for *VirtIO-Net backend*. This creates new io-pkt instance with memory area `pfe_ddr`. To avoid conflicts with already running GMAC io-pkt instance, it is necessary to use different system prefix `master`, so every command must be executed with prefix `SOCK=/master`.

```
io-pkt-v6-hc -p tcpip reply_ctxt=300,pkt_typed_mem=pfe_ddr,prefix="master" -t 8 -D \
 -d /sdcard/devnp-pfe-2-master.so
 pfe0_mac=0e8c01691d4e,pfe1_mac=9e83193b24d9,pfe2_mac=daef032f419b,class_fw=/sdcard/
s32g_pfe_class.fw,util_fw=/sdcard/s32g_pfe_util.fw \
 -d vdevpeer-net peer=/dev/qvm/qnx-guest/p2p_qnx,bind=/dev/vdevpeers/vp0,mac=a0b0c0d0e0f0 \
 -d vdevpeer-net peer=/dev/qvm/linux-guest/p2p_linux,bind=/dev/vdevpeers/vp1,mac=a0b0c0ddeeff
```

The output of `SOCK=/master ifconfig` should contain all of the network interfaces:



```
# SOCK=/master ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33136
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
pfe0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> mtu 1500
        capabilities=1f<IP4CSUM,TCP4CSUM,UDP4CSUM,TCP6CSUM,UDP6CSUM>
        enabled=0
        address: 0e:8c:01:69:1d:4e
        media: Ethernet none (1000baseT full-duplex)
        status: no carrier
pfe1: flags=8802<BROADCAST,SIMPLEX,MULTICAST> mtu 1500
        capabilities=1f<IP4CSUM,TCP4CSUM,UDP4CSUM,TCP6CSUM,UDP6CSUM>
        enabled=0
        address: 9e:83:19:3b:24:d9
        media: Ethernet none (1000baseT full-duplex)
        status: no carrier
pfe2: flags=8802<BROADCAST,SIMPLEX,MULTICAST> mtu 1500
        capabilities=1f<IP4CSUM,TCP4CSUM,UDP4CSUM,TCP6CSUM,UDP6CSUM>
        enabled=0
        address: da:ef:03:2f:41:9b
        media: Ethernet none (1000baseT full-duplex)
        status: no carrier
pfex0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> mtu 1500
        capabilities=1f<IP4CSUM,TCP4CSUM,UDP4CSUM,TCP6CSUM,UDP6CSUM>
        enabled=0
        address: 00:00:00:00:06:aa
        media: Ethernet autoselect (autoselect half-duplex)
        status: active
vp0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> mtu 1500
        capabilities rx=1f<IP4CSUM,TCP4CSUM,UDP4CSUM,TCP6CSUM,UDP6CSUM>
        capabilities tx=7e<TCP4CSUM,UDP4CSUM,TCP6CSUM,UDP6CSUM,TSO4,TSO6>
        enabled rx=1f<IP4CSUM,TCP4CSUM,UDP4CSUM,TCP6CSUM,UDP6CSUM>
        enabled tx=0
        address: a0:b0:c0:d0:e0:f0
        media: Ethernet autoselect
        status: active
vp1: flags=8802<BROADCAST,SIMPLEX,MULTICAST> mtu 1500
        capabilities rx=1f<IP4CSUM,TCP4CSUM,UDP4CSUM,TCP6CSUM,UDP6CSUM>
        capabilities tx=7e<TCP4CSUM,UDP4CSUM,TCP6CSUM,UDP6CSUM,TSO4,TSO6>
        enabled rx=1f<IP4CSUM,TCP4CSUM,UDP4CSUM,TCP6CSUM,UDP6CSUM>
        enabled tx=0
        address: a0:b0:c0:dd:ee:ff
        media: Ethernet autoselect
        status: active

Process 147486 (ifconfig) exited status=0.
#
```

More information about QNX PFE driver are descibed in PFE QNX Driver User Manual. Next configure software network bridge between pfex0 and virtual vp0 and vp1 interfaces:

```
SOCK=/master ifconfig bridge0 create
SOCK=/master brconfig bridge0 add pfex0 up
SOCK=/master brconfig bridge0 add vp0 up
SOCK=/master brconfig bridge0 add vp1 up
```

Now it is possible to set IP address and start the network interfaces:

```
SOCK=/master ifconfig pfex0 192.168.10.20
SOCK=/master ifconfig vp0 up
SOCK=/master ifconfig vp1 up
```

7.4.1 LibFCI CLI configuration

For communication with PFE and other network devices, user must set PFE configuration for VLAN_BRIDGE mode using the LibFCI interface. In this mode, every client can communicate with each other.

The libfci_cli application should be on the SD card and needs to be coppied to /tmp/ folder, as it is the only folder with write and execute permission in QNX system.

```
cp /sdcard/libfci_cli /tmp/ && chmod +x /tmp/libfci_cli
```

The following configuration includes all of the HIF channels and EMAC interfaces to the network bridge.

```
/tmp/libfci_cli bd-update --vlan 1 --uh FORWARD --um FLOOD --mh FORWARD --mm FLOOD
```

```
/tmp/libfci_cli bd-insif --vlan 1 --i hif0 --tag OFF
/tmp/libfci_cli bd-insif --vlan 1 --i hif1 --tag OFF
/tmp/libfci_cli bd-insif --vlan 1 --i hif2 --tag OFF
/tmp/libfci_cli bd-insif --vlan 1 --i hif3 --tag OFF
/tmp/libfci_cli bd-insif --vlan 1 --i emac0 --tag OFF
/tmp/libfci_cli bd-insif --vlan 1 --i emac1 --tag OFF
/tmp/libfci_cli bd-insif --vlan 1 --i emac2 --tag OFF
/tmp/libfci_cli phyif-update --i hif0 --E --promisc ON --mode VLAN_BRIDGE
/tmp/libfci_cli phyif-update --i hif1 --E --promisc ON --mode VLAN_BRIDGE
/tmp/libfci_cli phyif-update --i hif2 --E --promisc ON --mode VLAN_BRIDGE
/tmp/libfci_cli phyif-update --i hif3 --E --promisc ON --mode VLAN_BRIDGE
/tmp/libfci_cli phyif-update --i emac0 --E --promisc ON --mode VLAN_BRIDGE
/tmp/libfci_cli phyif-update --i emac1 --E --promisc ON --mode VLAN_BRIDGE
/tmp/libfci_cli phyif-update --i emac2 --E --promisc ON --mode VLAN_BRIDGE
```

Note: *It is better to copy it through SSH.*

7.5 Start Guest 1

To start this guest with **QNX** and **PFE**, new SSH session will be used (section SSH Connection). In the SD card folder with Guest 1 files, user can start *QNX Hypervisor Manager* using the configuration created in section Guest 1 QNX PFE configuration:

```
cd /sdcard/hv/guest1
qvm @qnx-hv-pfe.qvmconf
```

The QNX Guest should start and the console shows some basic information with the output message *"Reserving RAM region for PFE driver on Hypervisor Guest"*. Now the SD card can be mounted to access all the files.

```
devb-virtio virtio smem=0x1c0d0000,irq=41
mount -t dos /dev/hd0t12 /sdcard
```

Note: *The partition might have a different descriptor than /dev/sd0t*.*

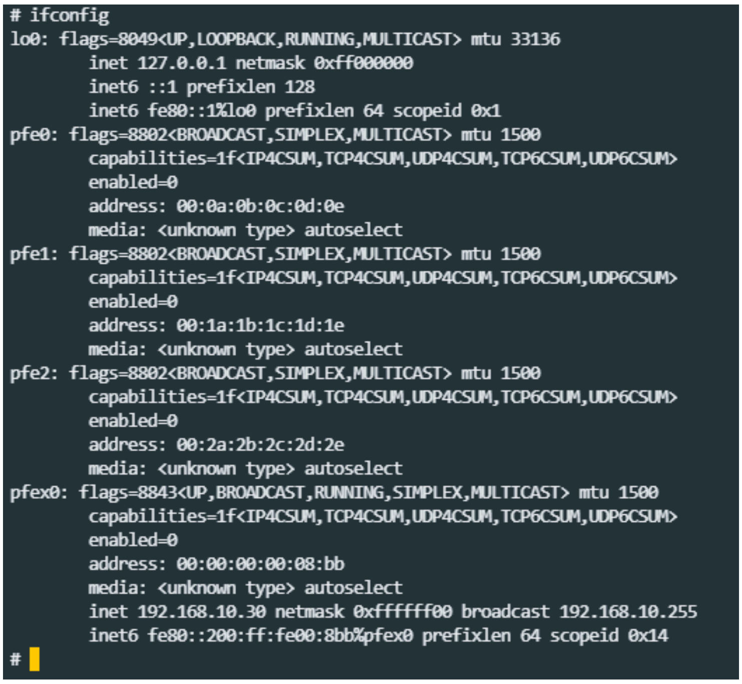
To initialize the PFE QNX Slave driver, io-pkt network stack needs to be started with the right arguments.

```
io-pkt-v6-hc -p tcpip pkt_typed_mem=pfe_ddr -d /sdcard/hv/guest1/devnp-pfe-2-slave.so \
 pfe0_mac=523148c01396,pfe1_mac=267d99456b01,pfe2_mac=a6e17c4ec0f9,pfex_mac=be93d6295e3b
```

Now it is possible to set IP address. It is necessary to use only **pfex0** interface for communication in VLAN_BRIDGE.

```
ifconfig pfex0 192.168.10.30
```

The output of `ifconfig` command should be like in the following image:



```
# ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33136
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
pfe0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> mtu 1500
        capabilities=1f<IP4CSUM,TCP4CSUM,UDP4CSUM,TCP6CSUM,UDP6CSUM>
        enabled=0
        address: 00:0a:0b:0c:0d:0e
        media: <unknown type> autoselect
pfe1: flags=8802<BROADCAST,SIMPLEX,MULTICAST> mtu 1500
        capabilities=1f<IP4CSUM,TCP4CSUM,UDP4CSUM,TCP6CSUM,UDP6CSUM>
        enabled=0
        address: 00:1a:1b:1c:1d:1e
        media: <unknown type> autoselect
pfe2: flags=8802<BROADCAST,SIMPLEX,MULTICAST> mtu 1500
        capabilities=1f<IP4CSUM,TCP4CSUM,UDP4CSUM,TCP6CSUM,UDP6CSUM>
        enabled=0
        address: 00:2a:2b:2c:2d:2e
        media: <unknown type> autoselect
pfex0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        capabilities=1f<IP4CSUM,TCP4CSUM,UDP4CSUM,TCP6CSUM,UDP6CSUM>
        enabled=0
        address: 00:00:00:00:08:bb
        media: <unknown type> autoselect
        inet 192.168.10.30 netmask 0xffffff00 broadcast 192.168.10.255
        inet6 fe80::200:ff:fe00:8bb%pfex0 prefixlen 64 scopeid 0x14
#
```

7.6 Start Guest 2

To start this guest with **Linux** and **PFE**, new SSH session will be used (section SSH Connection). In the folder with Guest 2 files, user can start *QNX Hypervisor Manager* using the configuration created in section Guest 2 Linux PFE configuration:

```
cd /sdcard/hv/guest2
qvm @linux-hv-pfe.qvmconf
```

The Linux Guest should start and prompt to user login. It is necessary to login with username `root`, and then mount the SD card to access the driver file.

```
mkdir /mnt/sdcard
mount /dev/vda1 /mnt/sdcard
```

Now user can load the PFE Linux pfeng slave driver.

Note: *It is possible to specify IDEX resend delay, or another options from Linux PFE Driver User Manual*

```
insmod /mnt/sdcard/hv/guest2/pfeng-slave.ko idex_resend_delays=300,300
```

User should be able to see all interfaces with the command `ifconfig -a`. It is necessary to use only **aux0sl** interface for communication in VLAN_BRIDGE.

```
ifconfig aux0sl 192.168.10.40
```

The output of `ifconfig` command should be like in the following image:

```
root@qemuarm64:~# ifconfig
aux0sl    Link encap:Ethernet  HWaddr 00:04:9F:BE:FF:80
          inet addr:192.168.10.40  Bcast:192.168.10.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Memory:46000000-46ffffff

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

7.7 Start Guest 3

To start this guest with **QNX** and **VirtIO-Net** driver, new connection to the SSH server needs to be created (section SSH Connection).

In the SD card folder with Guest 3 files, user can start *QNX Hypervisor Manager* using the configuration created in section Guest 3 QNX VirtIO-Net configuration:

```
cd /sdcard/hv/guest3
qvm @qnx-hv-virtio.qvmconf
```

The io-pkt instance with the **VirtIO-Net** driver can be started:

```
io-pkt-v6-hc -d /proc/boot/devnp-virtio.so smem=0x1c0e0000,irq=40
```

Now it is possible to set the IP address:

```
ifconfig vt0 192.168.10.50
```

The output of `ifconfig` command should be like in the following image:

```
# ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 33136
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
vt0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        address: aa:aa:aa:aa:aa:aa
        media: Ethernet autoselect
        status: active
        inet 192.168.10.50 netmask 0xffffff00 broadcast 192.168.10.255
        inet6 fe80::a8aa:aaff:feaa:aaaa%vt0 prefixlen 64 tentative scopeid 0x11
#
```

7.8 Start Guest 4

To start this guest with **Linux** and **VirtIO-Net**, user must connect again with a new SSH session.

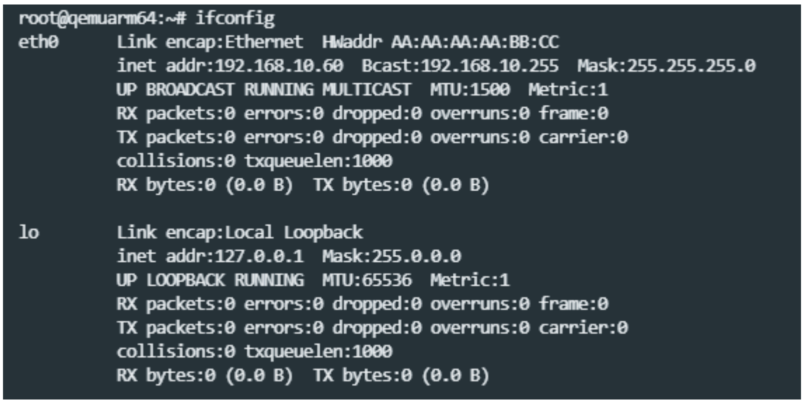
In the folder with Guest 4 files, it is possible to start a new *QNX Hypervisor Manager* the same way as with other guests using the configuration created in section Guest 4 Linux VirtIO-Net Linux.

```
cd /sdcard/hv/guest4
qvm @linux-hv-virtio.qvmconf
```

The network module is started automatically after the system boot by Linux VirtIO-Net generic driver and it is necessary just to set the IP address:

```
ifconfig eth0 192.168.10.60 up
```

The output of the `ifconfig` command should be like in the following image:



```
root@qemuarm64:~# ifconfig
eth0      Link encap:Ethernet  HWaddr AA:AA:AA:AA:BB:CC
          inet addr:192.168.10.60  Bcast:192.168.10.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

7.9 Test communication

Now, all of the systems should be able to communicate with each other and with external network. It is possible to ping or make some performance measurements among systems.

System	IP address
QNX Host	192.168.10.20
Guest 1 PFE QNX	192.168.10.30
Guest 2 PFE Linux	192.168.10.40
Guest 3 VirtIO-Net QNX	192.168.10.50
Guest 4 VirtIO-Net Linux	192.168.10.60

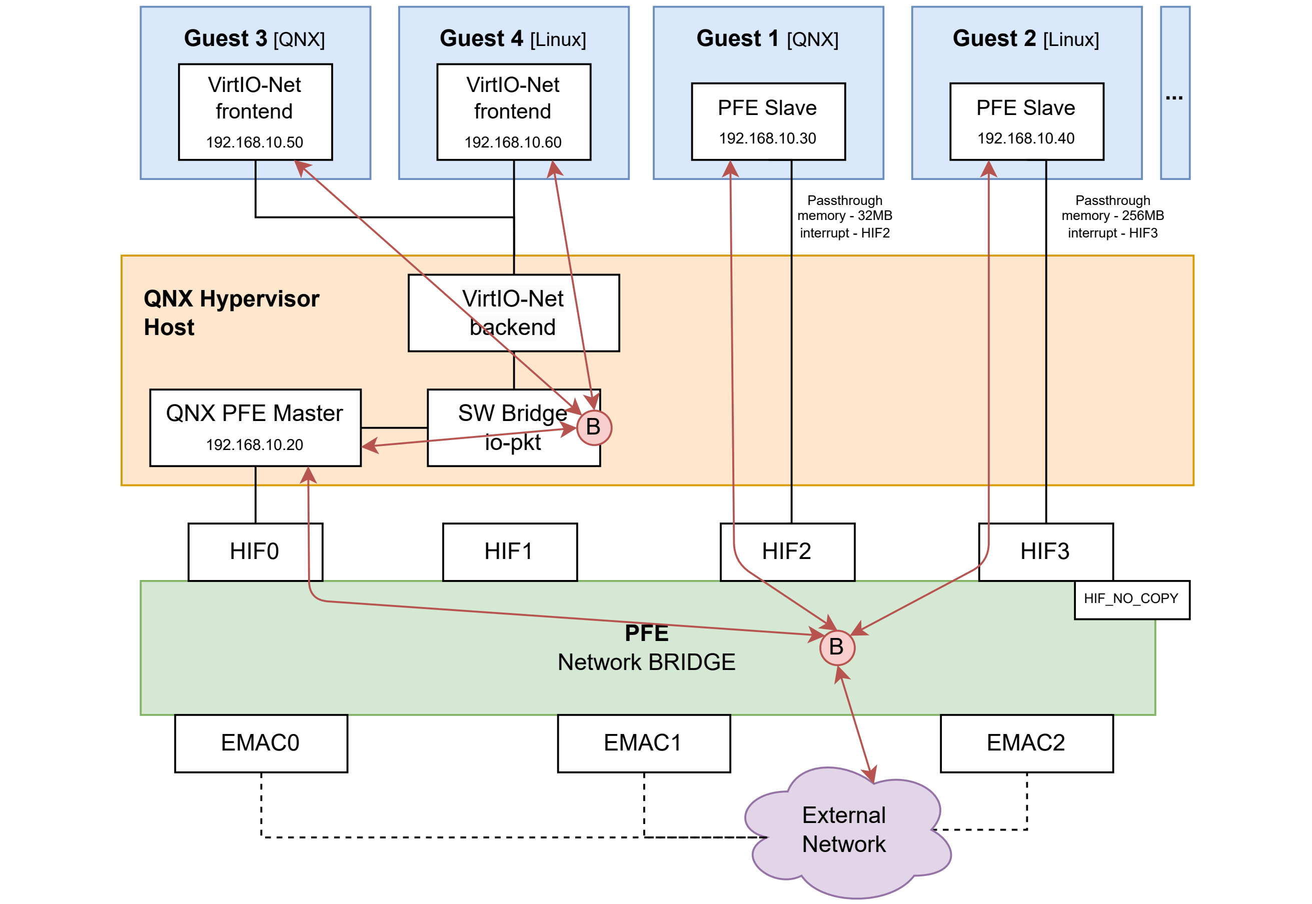


Figure 5. Demo Topology

Also it is possible to connect some external devices via PFE and communicate with each of the systems.

8 Troubleshooting

If there are problems with Hypervisor, it is possible to try some of these applications for debugging:

- `slog2info` - shows all logs output
- `pidin syspage=asinfo` - memory areas information to check
- `slay qvm` - force stop Hypervisor Manager
- `brconfig -a` - show software bridge info in Host system
- `/tmp/libfci_cli phyif-print` - print PFE configuration on Host

9 References

[1] *Linux PFE Driver User Manual*, available in Linux PFE driver source code repository; directory */doc/PFE_S32G_A53_LNX_UserManual.pdf*. The repository is at https://github.com/nxp-auto-linux/pfeng.

[2] *PFE QNX Driver User Manual*, available in QNX PFE driver source code repository; directory */doc/user_manual/PFE_QNX_DRV_S32G_UserManual.pdf*. QNX Driver can be obtained from FlexNet (nxp.flexnetoperations.com)

[3] *PFE QNX Driver Integration Manual*, available in QNX PFE driver source code repository; directory /*doc/qnx_drv_im/PFE_QNX_DRV_IntegrationManual.pdf*. QNX Driver can be obtained from FlexNet (nxp.flexnetoperations.com)

[4] *QNX Hypervisor 2.2 User's Guide - Networking*, available on page QNX online documentation

10 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:

11 Revision History

Document Revision History

Document ID	Release Date	Description
AN14355 v. 1.1.0	18 July 2024	• Update to use NXP Automotive Linux BSP instead of Poky Linux BSP. • Add QNX Makefile modification for linking problem. • Disable pci-server from QNX Guest startup file. • Fix QNX PFE driver build path.
AN14355 v. 1.0.1	20 June 2024	Update topology image, fix misleading information.
AN14355 v. 1.0.0	23 May 2024	Initial version.

Legal information

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.

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 https://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.

NXP B.V. — NXP B.V. is not an operating company and it does not distribute or sell products.

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

Date of release: 18 July 2024
Document identifier: AN14355