AN13329

Network subsystem troubleshooting on DPAA2 devices (illustrated with LX2160)

Rev. 0 — 30 August 2021 Application Note

Contents

1 Introduction

The goal of this document is to familiarize users with the configuration of networking subsystem on the LX2 SoC. With the basic commands for bringing up the network interfaces, it provides debugging techniques for various cases, such as:

- A component (port, PHY, or MAC) apparently does not work. Link is down on the PHY interface, packets are dropped at the DPNI level, there is no link at PCS/SerDes, or packets are not received by the MAC.
- There is a minimal software infrastructure. Take U-Boot for example, in which the Ethernet ports should successfully send/receive packets to/from a remote peer but, based on the information captured at the other side, no packets are sent out from the LX2 device or/and the network interface from the LX2 does not receive any frames.

The different LX2 setups that were used for showcasing are composed of RDBs and internal development platforms. The internal development platforms are more flexible in many ways. They were chosen, because it is possible to cover scenarios such as connecting two MACs from SerDes to SerDes without any PHY in between. On the RDB, this cannot be achieved (see Figure 1.a). The RDB is used in scenarios where the MAC is connected to a PHY (see Figure 1.b).

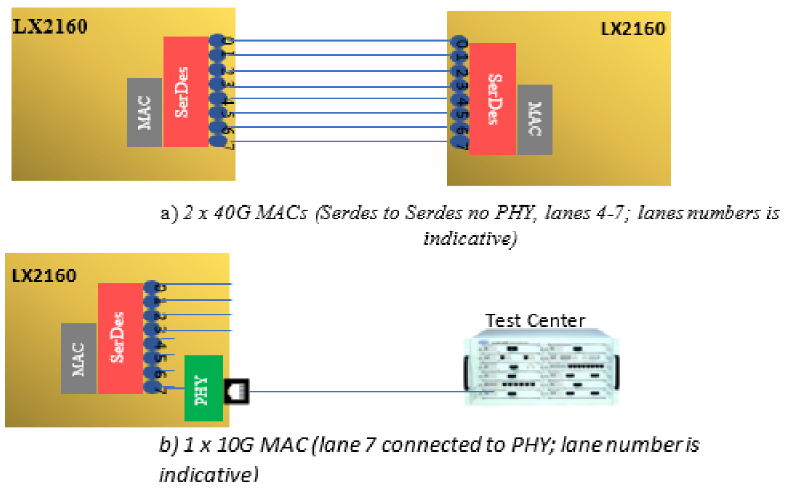

LX2160
MAC
SerDes
0 1 2 3 4 5 6 7
LX2160
SerDes
MAC
a) 2 x 40G MACs (Serdes to Serdes no PHY, lanes 4-7; lanes numbers is indicative)
LX2160
MAC
SerDes
PHY
Test Center
b) 1 x 10G MAC (lane 7 connected to PHY; lane number is indicative)

Figure 1. Different use case configurations

The reader should have a minimal background on the networking architecture for the Linux OS and basic knowledge on the DPAA2 infrastructure. For more details about the components of the DPAA2, see the *DPAA2 User Manual* (document DPAA2UM).

NOTE

The setup in Figure 1.b (RDB boards) may differ in a custom design. The 10G MAC can be connected directly (SerDes to SerDes) to another 10G MAC without a PHY.

2 LX2 network interface overview

The network interface configuration is shown in Figure 2.

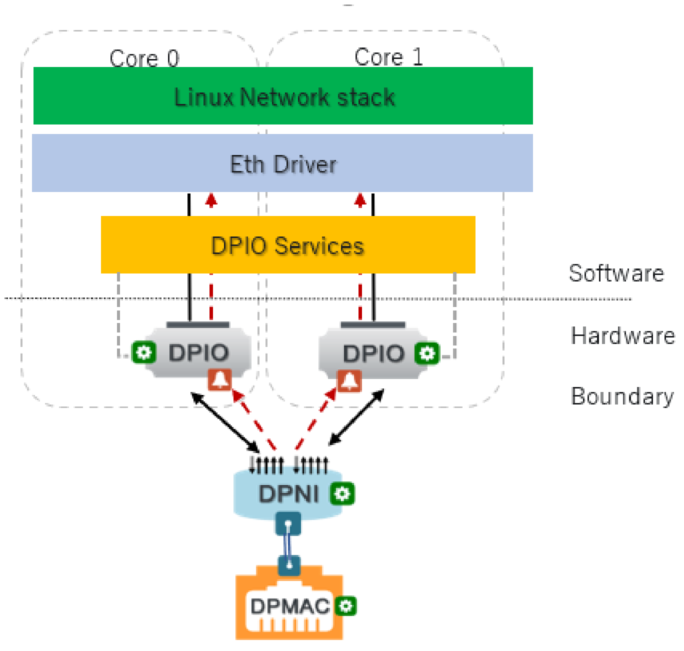


Figure 2. DPAA2 network interface

In Figure 2, the ingress and egress pipelines for a packet can be summarized as follows:

- On ingress:

  A packet is received from the line side and reaches into the FIFO of the DPMAC. The WRIOP port (DPNI), parses and classifies the frame, transfers it into a memory buffer (in the external system memory) from a buffer pool (DPBP) assigned to the DPNI, and enqueues it to one of the Rx queues configured for the interface. (in Figure 2, the frame can be consumed either by Core 0 or Core 1, depending on the classification result. A maximum of 16 cores can be configured for a network interface on the LX2).

- On egress:

  A packet is enqueued into one of the Tx queues available (the number of TX queues is equal to the number of RX queues) by one of the cores. The packet is dequeued by the WRIOP (DPNI), transferred from the system memory into the WRIOP memory, and sent to the DPMAC. The same memory buffer, containing the frame, is enqueued into a TX confirmation queue. From this queue, the core that sent the frame earlier (or another core) dequeues it and releases the buffer back into the DPNI's buffer pool (DPBP).

NOTE

A comprehensive description of the DPAA2 network infrastructure in the Linux OS is here.

The potential issues that you may encounter are:

- No link at PHY/MAC or PCS/SerDes
- Packets not received/sent
- Packets are dropped

The issues must be traced at the **PHY <--- >SerDes <--> DPMAC < --- > DPNI** levels and very rarely in the DPAA2 ETH driver itself (which is equivalent to the core side).

The simplest method to identify the blockage point is to check the counters on each of the above components. Observing the relevant counter incrementation (for example, the **ingress_discarded_frames** counter on the DPNI object) does not always reveal the source of the problem and it sometimes does not even give a clue about the approximate cause.

One approach of debugging is a step by step investigation by setting different loopback modes in various points (MAC, SerDes, PHY, PCS) and checking whether the frame(s) reached the expected level (MAC, DPNI, core). In the case of dropped frame(s), an error queue shall be enabled together with the debug prints. Both the loopback and error queue options may require additional changes in the software layer (U-Boot or Linux OS). These changes are described in this document.

These modes cannot be used if the goal is to check the performance on a port or to ensure that an interface (DPMAC) works according to the specifications of the selected SerDes protocol. For example, the SerDes protocol is SGMII and you try to validate the conversion from SGMII to KX by sending a couple of frames in the U-Boot.

2.1 Interface loopback modes

The loopback can be configured on different levels of a network interface:

- MAC
- PCS
- SerDes (serializer loopback, parallel interface loopback)
- PHY (if present)

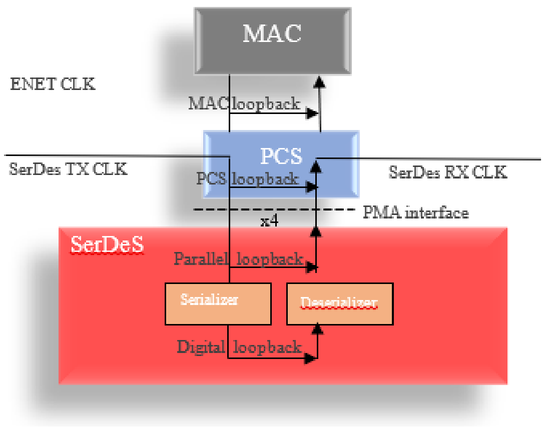


Figure 3. Loopback modes for 40G MAC

Observations:

- In the digital loopback mode, data is gated in the serializer (RX side only). The RX CLK is derived from the TX CLK.

The digital loopback can be used as a **sanity check**. If no valid link is observed at the PCS level when enabling this mode, it means that something is wrong at the SerDes level (a hardware issue or an incorrect SerDes configuration).

- In the parallel loopback, two cases can be distinguished:
  - (tx_clk to tx_clk) "tx_clk" is used to clock both sides of the FIFO (used to transfer data from TX to RX) and it is bypassed through to the “rx_clk”. On the parallel transmitter side, the data is transmitted and it can reach the remote peer. This mode can be used as a **sanity check** as well.
  - "rx_clk" is recovered from the input data on the "rx_p/rx_n" differential pair inputs. It requires a valid clock from the peer. This mode is not used in this document.
- In the PCS loopback mode, data is gated in the PCS as follows: On the TX path, data is transmitted normally, whereas on the RX, the data received from the PMA is discarded. The receive clocks must be stable and running at proper

frequencies. If there is no peer, the clock is recovered from the internal PLL. This enables you to test the PCS loopback even if the peer has an invalid configuration.

- In the MAC loopback mode, data is gated in the MAC. No data exits the MAC.
- If the investigation of a possible issue (for example, no traffic is received on the DPNI) is carried out in the U-Boot, enabling MAC loopback and PCS loopback may be problematic. The MAC loopback cannot be set without significant changes, because of the network design configuration. The interface is created (the DPMAC is enabled and configured) when a command that sends traffic (for example, ping) is executed from the CLI. It means that the content of the DPMAC COMMAND_CONFIG register that holds the loopback bit is overwritten every time the ping command is executed. The PCS loopback configured for DPMACs in the SGMII/QSGMII mode is overwritten as well, with two exceptions: USXGMII and XFI. In these two modes, the PCS content is not overwritten during the DPMAC enablement. In this document, the PCS loopback is successfully applied from the U-Boot, because the exemplified interfaces are in the XFI/USXGMII mode.

3 Troubleshooting in U-Boot

3.1 Introduction

This section describes the cases in which the traffic is not received on an LX2 port or when it does not reach the destination. The setup is similar to the representation in Figure 1.

Before proceeding with further details on the use cases, here is a short description of the Management Complex (MC).

MC is an NXP provided firmware that runs on the e200 control cores. It implements the DPAA2 object model abstraction logic (hiding all the complexity of the underlying hardware – for example, QBMAN and WRIOP) for the upper software layers that run on the General Purpose Cores (GPPs). MC is a trusted entity and it executes only private trusted code. It can be viewed as a hypervisor slave device. GPP processes do not have direct access to most of the DPAA2 resources. For example, if you want to update the configuration of a frame queue using the QBMAN APIs directly (bypassing the MC API implementation), you cannot access the real frame queue ID and you cannot change the queue configuration. Instead, perform all necessary DPAA2 management using commands to the MC. For more details about MC and DPAA2 object model, see the *DPAA2 User Manual* (document DPAA2UM).

If the MC firmware is not loaded by the boot loader (U-Boot), the DPAA2 network subsystem is not initialized, and it cannot be used in U-Boot or Linux OS (no interfaces are probed).

To use an interface in the U-Boot and later in the Linux OS, load the MC:

```
fsl_mc start mc <firmware_addr> [DPC_addr]
```

The **Data Path Configuration** (DPC) file contains board-specific and system-specific information that may override the default DPAA hardware configuration. See the *DPAA2 User Manual* (document DPAA2UM).

The following are some observations about the DPC:

- It is found at flash offset = 0xe00000/SD card offset = 0x7000.
- It has a structure similar to the device tree.
- It can be compiled using the "dtc" tool. A blob similar to the DTB is generated.
- It is optional. The MC can boot without it. The U-Boot displays the following warning if no DPC is found: `fsl-mc: WARNING: No DPC image found.`
- Some examples are here.

3.1.1 Displaying the MC log buffer if the firmware cannot start

If an error is returned when the the MC firmware is started (see the snippet below), you can investigate its cause by displaying the log buffer in the U-Boot (assuming there is no console to print the MC output log).

```
fsl-mc: Booting Management Complex ... WARNING: Firmware returned an error (GSR: 0x19)
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x19)
```

The steps applied to print the log buffer contents in the U-Boot CLI are:

- Check that the current DPC has the following configuration:

```
[…]

mc_general
{
  log
  {
    mode = "LOG_MODE_ON";
    level = "LOG_LEVEL_WARNING";
  };
[…]
```

To display the current DPC, execute the following sequence of commands:

```
=> fdt addr 0x20e00000
=> fdt print
```

- Determine the MC firmware base. At the U-Boot prompt, execute “md” for address **0x8340020** (this is MCFBA{L/H}):

```
=> md 0x8340020 10
08340020: e0000006 00000027 00060000 00000000    ....'...........
08340030: 00000000 00000000 00000000 00000000    ................
08340040: 00000000 00000000 00000000 00000000    ................
08340050: 00000000 00000000 00000000 00000000    ................
```

The value at **0x08340024** is the MCFBAH address. In this case, it is **0x00000027**. The value at **0x08340020** is the MCFBAL address. In this case, it is **0xe0000000**. The MCFBA base address is built as **0x27e0000000**.

- Dump the log buffer structure at offset **0x01000000** from the MCFBA base address **0x27e1000000**:

```
=> md 0x27e1000000
27e1000000: 4d430100 00000000 01400000 00300000    ..CM......@...0.
27e1000010: 000000b1 00000000 00000000 00000000    ................
27e1000020: 00000000 00000000 00000000 00000000    ................
27e1000030: 00000000 00000000 00000000 00000000    ................
```

0x4d430100 is the magic number, **0x400000** is the log buffer offset, and **0x00300000** is the log buffer length.

- Dump the content of the log buffer. The output size can be increased by specifying the number of objects. In the following caption, the number of objects is 100:

```
=> md 0x27e1400000 100
```

3.1.2 MACs physical offset

Table 1 lists the physical offsets for the MACs on the LX2. These offsets are useful when you read the PCS (or any register of the MAC ports) that are interrogated in the debug process.

Table 1. MAC base addresses

MAC	Base address
1	0x8c07000
2	0x8c0b000
3	0x8c0f000
4	0x8c13000
5	0x8c17000
6	0x8c1b000
7	0x8c1f000
8	0x8c23000
9	0x8c27000
10	0x8c2b000
11	0x8c2f000
12	0x8c33000
13	0x8c37000
14	0x8c3b000
15	0x8c3f000
16	0x8c43000
17	0x8c47000
18	0x8c4b000

3.2 U-Boot use case with 40G MAC

In this setup (Figure 4), a 40G MAC (MAC2) is connected directly (SerDes to SerDes) to another 40G MAC (MAC 2) or to another device that has a 40G port.

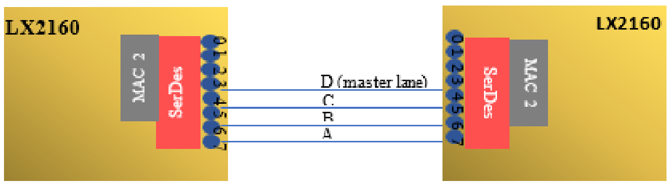


Figure 4. 40G MAC2 connected to 40G MAC2 (lanes A-D)

Hypothesis:

MAC2 uses lanes A-D on the SerDes1 module. The MAC is defined in the DPC file as:

```
[…]
board_info
{
  ports
  {
     mac@2
    {
      link_type = "MAC_LINK_TYPE_FIXED";
     };
```

The link type shows that there is no transceiver between the MAC and the peer. The connection is “TYPE_FIXED”.

If you want to send some packets from the left side to test the connectivity, launch the ping command from the CLI.

Result:

The outcome is that no reply is received from the right side either, because there is no link or something happens with the packets on the way between the two peers. The next caption contains the user commands and the result.

```
fsl-mc: Booting Management Complex ... SUCCESS
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x1)
Hit any key to stop autoboot:  0
[…]
=> setenv ethact DPMAC2@xlaui4
=> ping 1.1.1.2
Using DPMAC2@xlaui4 device
Abort
ping failed; host 1.1.1.2 is not alive
```

The challenge here is to determine if a valid link is reported in the 40G PCS device status register.

The first step to exclude any SerDes problems (either configuration or hardware design integration issues - if the LX2 SoC is part of a custom design) is to execute a sanity check.

The next section describes the process.

3.2.1 SerDes sanity check

This test checks the PCS link status when the SerDes is configured in the digital loopback.

In this mode, the link must always be up. Users can be skeptical, because they do not have access to the firmware source code and they consider the MC the root cause for a non-working setup. To rule out any potential problems that may be introduced by the MC firmware, the firmware must not be started in the validation procedure. This is equivalent to omitting the execution of the command that loads the MC binary file in the U-Boot environment:

```
#search through env variables and eliminate this command. Usually the mcinitcmd contains
it. Remove it from there and restart the board

#remove fsl_mc start mc 0x20a00000  0x20e00000;
```

To set a SerDes in the digital loopback, execute the following commands for lanes A-D (MAC2):

```
mw 0x1EA08A0 0x10000000
mw 0x1EA09A0 0x10000000
mw 0x1EA0AA0 0x10000000
mw 0x1EA0BA0 0x10000000
```

To read the link status bit, the MAC2 internal MDIO bus must be accessed. The pseudo code of this operation is described as `“read <MAC> <PCS> <reg>”`.

Here is a summary of the sanity check steps:

```
#in the U-Boot CLI modify the U-Boot env such that mc firmware is not started then reset the board
(after the reset, MC is not loaded)

#set lanes A-D in digital loopback
mw 0x1EA08A0 0x10000000
mw 0x1EA09A0 0x10000000
mw 0x1EA0AA0 0x10000000
mw 0x1EA0BA0 0x10000000

#read the PCS status register from MAC2. see (*)
```

The next section describes an implementation variant to execute read/write operations on the internal MDIO bus.

3.2.1.1 Internal MDIO command implementation

This command uses the skeleton of the standalone applications from the U-Boot tree. The U-Boot version used in this document is here.

The next snippet expands the **examples** folder inside the U-Boot tree and shows the **mdio** folder and its contents that were forked from the standalone application. Inside the **mdio** folder is the MDIO source that contains the implementation to access the internal MDIO bus for a given MAC.

```
examples/
├── api
│   ├── crt0.S
│   ├── demo.c
│   ├── glue.c
│   ├── glue.h
│   ├── libgenwrap.c
│   └── Makefile
├── Makefile
├── mdio
│   ├── libstubs.o
│   ├── Makefile
│   ├── mdio
│   ├── mdio.bin
│   ├── mdio.c
│   ├── mdio.o
│   ├── mdio.srec
│   ├── mdio.su
│   ├── ppc_longjmp.S
│   ├── ppc_setjmp.S
│   ├── stubs.c
│   ├── stubs.o
│   └── stubs.su
└── standalone
    ├── atmel_df_pow2.c
    ├── hello_world
    ├── hello_world.bin
    ├── hello_world.c
    ├── hello_world.o
    ├── hello_world.srec
    ├── hello_world.su
    ├── libstubs.o
    ├── Makefile
    ├── nds32.lds
    ├── ppc_longjmp.S
    ├── ppc_setjmp.S
```

```
     ├── README.smc91111_eeprom
     ├── sched.c
     ├── smc91111_eeprom.c
     ├── smc911x_eeprom.c
     ├── sparc.lds
```

The content of the MDIO source can be used as a reference implementation. The code below works for both Clause 45 and Clause 22 MDIO commands. Clause 45 is activated by default. This document presents use cases only with Clause 45.

```
diff --git a/examples/Makefile b/examples/Makefile
index d440bc5655..f0bcb03f42 100644
--- a/examples/Makefile
+++ b/examples/Makefile
@@ -6,6 +6,6 @@ ifdef FTRACE
 subdir-ccflags-y += -finstrument-functions -DFTRACE
 endif

-subdir-y += standalone
+subdir-y += standalone mdio
 subdir-$(CONFIG_API) += api
 endif

diff --git a/examples/mdio/mdio.c b/examples/mdio/mdio.c
new file mode 100644
index 0000000000..10d1f18f40
--- /dev/null
+++ b/examples/mdio/mdio.c
@@ -0,0 +1,227 @@
+// SPDX-License-Identifier: GPL-2.0+
+/*
+ * (C) Copyright 2000
+ * Wolfgang Denk, DENX Software Engineering, wd@denx.de.
+ */
+
+#include <common.h>
+#include <exports.h>
+#include <stdlib.h>
+#include <stdio.h>
+#include <string.h>
+#include <asm/io.h>
+#include <linux/errno.h>
+#include <asm/arch/soc.h>
+
+
+#define MEMAC_MDIO_CFG  0x30
+#define MEMAC_MDIO_CTRL 0x34
+#define MEMAC_MDIO_DATA 0x38
+#define MEMAC_MDIO_ADDR 0x3c
+#define USAGE "Usage: go 0x80300000 <read|write> <mac offset> <mmd> <reg> [data]\n"
+
+struct mdio_info {
+    u32 *mdio_cfg;
+    u32 *mdio_ctrl;
+    u32 *mdio_addr;
+    u32 *mdio_data;
+    u32 phy;
+    u32 ext_phy;
+    u32 reg;
+    u32 mac_addr;
+    u32 data;
+};
+
+static void mdio_write(struct mdio_info *bus, int cl45)
+{
+    u32 temp;
+
+    bus->phy = (bus->ext_phy << 5) | bus->phy;
+    temp = in_le32(bus->mdio_cfg);
+    if (cl45) {
+        temp = (temp & 0xFF80) | 0x1C | 0x40;
+    } else {
+        temp = (temp & 0xFF80) | 0x1C;
+    }
+
+    out_le32(bus->mdio_cfg, temp);
+
+    temp = in_le32(bus->mdio_cfg);
+    temp = temp & 1;
+    while (temp) {
+        printf("mdio_cfg bsy set\n");
+        temp = in_le32(bus->mdio_cfg);
+        temp = temp & 1;
+        udelay(500);
+    }
+
+    if (cl45) {
+        out_le32(bus->mdio_ctrl, bus->phy);
+        out_le32(bus->mdio_addr, bus->reg);
+    } else {
+        out_le32(bus->mdio_ctrl, bus->reg);
+    }
+
+    temp = in_le32(bus->mdio_cfg);
+    temp = temp & 1;
+    while (temp) {
+        printf("mdio_cfg bsy set\n");
+        temp = in_le32(bus->mdio_cfg);
+        temp = temp & 1;
+        udelay(500);
+    }
+
+    out_le32(bus->mdio_data, bus->data);
+
+    temp = in_le32(bus->mdio_data);
+    temp = temp | 0x80000000;
+
+    while (temp) {
+        printf("mdio_data bsy set\n");
+        temp = in_le32(bus->mdio_data);
+        temp = temp & 0x80000000;
+        udelay(500);
+    }
+
+    return;
+}
+
+static void mdio_read(struct mdio_info *bus, int cl45)
+{
+    u32 temp;
```

```
+
+    bus->phy = (bus->ext_phy << 5) | bus->phy;
+    temp = in_le32(bus->mdio_cfg);
+    if (cl45) {
+        temp = (temp & 0xFF80) | 0x1C | 0x40;
+    } else {
+        temp = (temp & 0xFF80) | 0x1C;
+    }
+    out_le32(bus->mdio_cfg, temp);
+
+    temp = in_le32(bus->mdio_cfg);
+    temp = temp & 1;
+    while (temp) {
+        printf("mdio_cfg bsy set\n");
+        temp = in_le32(bus->mdio_cfg);
+        temp = temp & 1;
+        udelay(500);
+        }
+
+    if (cl45) {
+        out_le32(bus->mdio_ctrl, bus->phy);
+        out_le32(bus->mdio_addr, bus->reg);
+
+        temp = in_le32(bus->mdio_cfg);
+        temp = temp & 1;
+        while (temp) {
+            printf("mdio_cfg bsy set\n");
+            temp = in_le32(bus->mdio_cfg);
+                    temp = temp & 1;
+                    udelay(500);
+            }
+    }
+    if (cl45)
+        temp = bus->phy | 0x8000;
+    else
+        temp = bus->reg | 0x8000;
+
+    if (cl45)
+        out_le32(bus->mdio_ctrl, temp);
+    else {
+        out_le32(bus->mdio_ctrl, temp);
+        temp = in_le32(bus->mdio_cfg);
+        temp = temp & 1;
+        while (temp) {
+            printf("mdio_cfg bsy set\n");
+            temp = in_le32(bus->mdio_cfg);
+            temp = temp & 1;
+            udelay(500);
+        }
+    }
+
+    temp = in_le32(bus->mdio_data);
+    temp = temp | 0x80000000;
+
+    while (temp) {
+        printf("mdio_data bsy set\n");
+        temp = in_le32(bus->mdio_data);
+        temp = temp & 0x80000000;
+        udelay(500);
+        }
```

```
+
+    temp = in_le32(bus->mdio_data);
+    printf ("\n\noutput data: 0x%08x\n", temp);
+
+    return;
+}
+
+int mdio (int argc, char * const argv[])
+{
+    int i;
+    struct mdio_info internal_mdio = {0};
+    u32 temp;
+    static const char *mapping[] = {
+        [0] = "",
+        [1] = "operation",
+        [2] = "mac offset",
+        [3] = "dev addr",
+        [4] = "reg addr",
+        [5] = "data",
+        [6] = ""
+    };
+
+    /* Print the ABI version */
+    app_startup(argv);
+    printf ("Example expects ABI version %d\n", XF_VERSION);
+    printf ("Actual U-Boot ABI version %d\n", (int)get_version());
+
+    printf ("Internal MDIO read / write \n");
+
+    printf ("argc = %d\n", argc);
+    for (i = 1; i < argc; ++i) {
+        printf ("%s = \"%s\"\n",
+            mapping[i],
+            argv[i] ? argv[i] : "<NULL>");
+    }
+    internal_mdio.mac_addr = (u32)strtoul(argv[2], NULL, 0);
+    internal_mdio.phy = (u32)strtoul(argv[3], NULL, 0);
+    internal_mdio.reg = (u32)strtoul(argv[4], NULL, 0);
+    internal_mdio.mdio_cfg  = (u32 *)(internal_mdio.mac_addr +
+                           MEMAC_MDIO_CFG);
+    internal_mdio.mdio_ctrl = (u32 *)(internal_mdio.mac_addr +
+                           MEMAC_MDIO_CTRL);
+    internal_mdio.mdio_addr = (u32 *)(internal_mdio.mac_addr +
+                           MEMAC_MDIO_ADDR);
+        internal_mdio.mdio_data = (u32 *)(internal_mdio.mac_addr +
+                           MEMAC_MDIO_DATA);
+    if (!strcmp(argv[1], "read")) {
+        int cl = 0;
+
+        if (argc == 5)
+            cl = 1;
+
+        if ((argc != 5) && (argc != 6))
+            printf(USAGE);
+        else
+            mdio_read(&internal_mdio, cl);
+    }
+
+    if (!strcmp(argv[1], "write")) {
+        int cl = 0;
+
+        internal_mdio.data = (u32)strtoul(argv[5], NULL, 0);
+
+        if (argc == 6)
+            cl = 1;
+
+
+        if ((argc != 6) && (argc != 7))
+            printf(USAGE);
+        else
+            mdio_write(&internal_mdio, cl);
+    }
+
+    printf ("\n\n");
+    return (0);
+}
```

The MDIO standalone application is built together with the U-Boot and the resulted image (**mdio.bin**) is loaded into the U-Boot and executed. The detailed steps for building the U-Boot using a standalone toolchain are here.

NOTE

There is no need to change the U-Boot on the target device when building this application. Compile the U-Boot as usual and transfer the resulted MDIO binary to the target boards.

After the MDIO standalone application is compiled, determine the entry point address in the DDR memory where the binary is loaded and launched from the U-Boot CLI:

```
aarch64-linux-gnu-readelf   ./examples/mdio/mdio -a | grep -i entry

Entry point address:               0x80300000
```

In this case, the address is **0x80300000**.

Coming back to the sanity check process, the last step (see SerDes Sanity check) is to read the PCS status register:

```
# Display the env variable that loads the mdio binary
printenv elf_load
elf_load=dcache off; tftp 0x80300000 b37577/mdio.bin; dcache on;

run elf_load

#(*) Read the PCS status register for MAC2 (see Table 1)
go 0x80300000 read 0x8c0b000 3 1

## Starting application at 0x80300000 ...
Example expects ABI version 9
Actual U-Boot ABI version 9
Internal MDIO read / write
argc = 5
operation = "read"
mac offset = "0x8c0b000"
dev addr = "3"
reg addr = "1"
mdio_cfg bsy set
mdio_data bsy set

output data: 0x00000004
## Application terminated, rc = 0x0
```

The **"go 0x80300000 read 0x8c0b000 3 1" -** command launches the MDIO binary that is transferred at address 0x80300000. Its input arguments are:

- Operation type: read/write, read in the above case
- MAC base address, **0x8c0b000** in the current case
- MMD device address, PCS in this case - offset **3**
- Register address in the MMD space, in this example - Status register - address **1**

NOTE

Run the above command multiple times to avoid latched data.

3.2.1.2 SerDes sanity check result interpretation

The **output data** row from the read operation shows that the **Receive Link Status (bit 2 in Status Register)** is set (value of 0x00000004 - see the previous snippet). It means that there is a valid link in the PCS in the digital loopback mode, without running the MC firmware. This excludes a configuration issue on the SerDes.

If the **Receive Link Status bit** is not set, this points either to a configuration issue (RCW for example) or a problem in the design (decoupling caps, AVDD out of spec, and so on).

The next phases involve sending traffic from the LX2 to the peer from the right side, while different loopback modes are set (Figure 3 Loopback modes for 40G MAC). The idea is to incrementally verify that the traffic can be sent and received by the DPAA2 hardware without a problem.

3.2.2 Digital loopback

This case may seem redundant, because digital loopback was described in the previous section. The approach described here uses packets to verify both transmit and receive paths. Packets are gated in the serializer on the Rx side and looped from Tx to Rx.

The purpose of this test is to validate that the traffic (for example, ARP requests) can be successfully sent and received by the DPNI interface.

The simplified view of the traffic flow is shown in Figure 5.

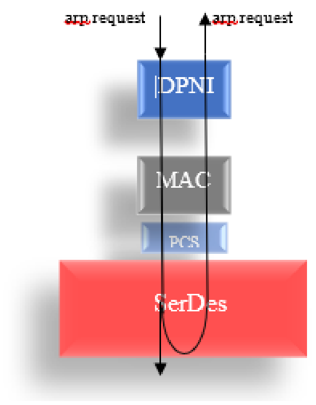


Figure 5. Digital loopback

The previous section shows that there is always a valid link present in the digital loopback.

The goal in this situation is to validate that the transmitted packets are looped and received back.

To eliminate any possible configuration problems related to link propagation between DPMAC and DPNI, some changes are applied in the U-Boot code and in the DPC file.

NOTE

The modifications from the U-Boot are valid also for the PSC loopback (see PCS loopback).

3.2.2.1 U-Boot changes

The objective of these changes is to report that the link is always up when a PHY is configured in the dpaa2 Ethernet driver probing sequence. For the PHY-less setup, these updates have no effect.

```
diff --git a/drivers/net/ldpaa_eth/ldpaa_eth.c b/drivers/net/ldpaa_eth/ldpaa_eth.c
index a3b9c152b2..1612827db2 100644
--- a/drivers/net/ldpaa_eth/ldpaa_eth.c
+++ b/drivers/net/ldpaa_eth/ldpaa_eth.c
@@ -17,7 +17,7 @@
 #include <fsl-mc/ldpaa_wriop.h>
 #include "ldpaa_eth.h"
-
+#define DEBUG
 #ifdef CONFIG_PHYLIB
 static int init_phy(struct eth_device *dev)
 {
diff --git a/drivers/net/phy/aquantia.c b/drivers/net/phy/aquantia.c
index 8ece926dd3..a149501119 100644
--- a/drivers/net/phy/aquantia.c
+++ b/drivers/net/phy/aquantia.c
@@ -554,6 +554,10 @@ int aquantia_startup(struct phy_device *phydev)
        int i = 0;
        phydev->duplex = DUPLEX_FULL;
+
+       phydev->link = 1;
+       return 0;
+
        /* if the AN is still in progress, wait till timeout. */
        if (!aquantia_link_is_up(phydev)) {
diff --git a/drivers/net/phy/phy.c b/drivers/net/phy/phy.c
index 33bce26e7f..3b32d7f234 100644
--- a/drivers/net/phy/phy.c
+++ b/drivers/net/phy/phy.c
@@ -222,6 +222,8 @@ int genphy_update_link(struct phy_device *phydev)
 {
        unsigned int mii_reg;
+       phydev->link = 1;
+       return 0;
        /*
         * Wait if the link is up, and autonegotiation is in progress
         * (ie - we're capable and it's not done)
```

The first update enables debug prints in the DPAA2 driver and the other two change the generic PHY driver and the AQR driver (AQR PHY is present on the LX2 RDB) behavior such that a linkup is always reported. If the link down event is reported by those functions, then the U-Boot init sequence for the DPAA2 Ethernet interfaces always fails when a ping command is launched.

NOTE

The U-Boot version used in the current document is here.

3.2.2.2 DPC changes

```
[…]
board_info {
                ports {
```

```
                            mac@2 {
                                   link_type = "MAC_LINK_TYPE_FIXED";
                                   debug_link_check="off";
                           };
[…]
```

The debug_link_check is always on, if it is not specified in the DPC. It has no effect if it is used on a MAC other than TYPE_FIXED. This option must be used in a debug scenario where you want to put the MAC or PCS into the loopback for testing purposes. If there is a problem at the peer or SERDES level, a "link down" event propagates to the DPNI. In this situation, the Rx/Tx queues are disabled by the software layer (Linux DPAA2 driver or U-Boot) and no traffic can be sent/received. When this variable is set to "off", the link is always reported as up, even if it is not. Traffic can be sent/received by the WRIOP ports, making the PCS/MAC loopback a valid setup. This way you can validate that packets can be received/sent at the PCS/MAC level.

3.2.2.3 Compiling U-Boot and DPC

After the U-Boot and DPC changes are applied, generate the firmware image and the DPC binary blob and write them to the target board. This document does not detail any build steps. They can be taken from here or from the *Layerscape Software Development Kit User Guide*.

There is an alternative to modify the DPC without deploying it into the flash on the running target.

This operation must be executed before starting the MC firmware. It saves the effort for compiling the DPC, deploying it, and flashing it to the board. This method can be used for any possible change that may be applied on the DPC.

```
#Do not start the MC firmware.(remove “fsl_mc start mc 0x20a00000  0x20e00000” from env and reboot
the target, if the MC was already started)
#Assuming that the DPC is at address 0x20e00000 in flash (mapped to DDR), move the DPC to #a new DDR
address (to be able to dynamically change it)
fdt addr 0x20e00000
fdt move 0x20e00000 0x85000000 0x5000
#Add the debug_link_check option
fdt addr 0x85000000
fdt rm /board_info/ports/mac@2
fdt mknode /board_info/ports mac@2
fdt set    /board_info/ports/mac@2 link_type  "MAC_LINK_TYPE_FIXED";
fdt set    /board_info/ports/mac@2 debug_link_check  "off";
#start the MC using the DPC from the new DDR address
fsl_mc start mc 0x20a00000 0x85000000;
```

3.2.2.4 Digital loopback verification steps

The stages of running traffic in the digital loopback mode are as follows:

```
#after the firmware and DPC have been deployed on the target, reboot. For dynamically changing the
DPC without rebooting and flashing it, see previous section.
#Note MC firmware must be started in order to be able to use the interfaces
#A log as below should be seen:

fsl-mc: Booting Management Complex ... SUCCESS
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x1)
Hit any key to stop autoboot:  0

#set lanes A-D in digital loopback
mw 0x1EA08A0 0x10000000
mw 0x1EA09A0 0x10000000
mw 0x1EA0AA0 0x10000000
mw 0x1EA0BA0 0x10000000

#set ethact to be MAC2
```

```
setenv ethact DPMAC2@xlaui4

#run the ping command
=> ping 1.1.1.2

ldpaa_dpmac_bind, DPMAC Type= dpmac
ldpaa_dpmac_bind, DPMAC ID= 2
ldpaa_dpmac_bind, DPMAC State= 0
ldpaa_dpmac_bind, DPNI Type= dpni
ldpaa_dpmac_bind, DPNI ID= 0
ldpaa_dpmac_bind, DPNI State= 0
DPMAC link status: 1 - up
DPNI link status: 1 - up
Using DPMAC2@xlaui4 device
```

When the ping command is launched, do not expect to see an echo response. The purpose of this test is to receive the packets back, because the SerDes is in a digital loopback. The command is let to run for a couple of seconds. Then it is interrupted by pressing CTRL + C. The following output or a similar one (DPNI counters can differ but must not be 0) should be displayed in the console:

```
DPNI counters ..
DPNI_CNT_ING_ALL_FRAMES= 3
DPNI_CNT_ING_ALL_BYTES= 180
DPNI_CNT_ING_MCAST_FRAMES= 0
DPNI_CNT_ING_MCAST_BYTES= 0
DPNI_CNT_ING_BCAST_FRAMES= 3
DPNI_CNT_ING_BCAST_BYTES= 180
DPNI_CNT_EGR_ALL_FRAMES= 3
DPNI_CNT_EGR_ALL_BYTES= 126
DPNI_CNT_EGR_MCAST_FRAMES= 0
DPNI_CNT_EGR_MCAST_BYTES= 0
DPNI_CNT_EGR_BCAST_FRAMES= 3
DPNI_CNT_EGR_BCAST_BYTES= 126
DPNI_CNT_ING_FILTERED_FRAMES= 0
DPNI_CNT_ING_DISCARDED_FRAMES= 0
DPNI_CNT_ING_NOBUFFER_DISCARDS= 0
DPNI_CNT_EGR_DISCARDED_FRAMES= 0
DPNI_CNT_EGR_CNF_FRAMES= 0

DPMAC counters ..
DPMAC_CNT_ING_BYTE=15
DPMAC_CNT_ING_FRAME_DISCARD=11
DPMAC_CNT_ING_ALIGN_ERR =11
DPMAC_CNT_ING_BYTE=15
DPMAC_CNT_ING_ERR_FRAME=23
DPMAC_CNT_EGR_BYTE =23
DPMAC_CNT_EGR_ERR_FRAME =27
Abort
ping failed; host 1.1.1.2 is not alive
```

3.2.2.5 Digital loopback result interpretation

The DPNI_CNT_ING_ALL_FRAMES and DPNI_CNT_EGR_ALL_FRAMES are equal to 3.

It means that 3 ARP requests were sent and received by the WRIOP port (DPNI).

The sent packets are arp and not echo requests, because before sending the echo request, the destination must be identified.

In parallel, it can be noticed that the PCS has a link due to the digital loopback configuration:

```
# Load the MDIO binary

go 0x80300000 read 0x8c0b000 3 1

## Starting application at 0x80300000 ...
Example expects ABI version 9
Actual U-Boot ABI version 9
Internal MDIO read / write
argc = 5
operation = "read"
mac offset = "0x8c0b000"
dev addr = "3"
reg addr = "1"
mdio_cfg bsy set
mdio_data bsy set

output data: 0x00000004

## Application terminated, rc = 0x0
```

NOTE

Run the above command multiple times to avoid latched data.

This result excludes any issues in both DPMAC and DPNI.

In case of a negative result (if no packets were received at the DPNI level), a possible issue is at the PCS (though very unlikely). The next stage checks this block by putting it into loopback.

3.2.3 PCS loopback

In a PCS loopback, data is transmitted normally. On the Rx side, it is discarded at the PMA level (from a SerDes perspective). The ARP requests sent from the U-Boot are looped inside a PCS and received in the WRIOP port (DPNI).

NOTE

It is not mandatory to have a link in the PCS when testing this configuration. If there is a link problem for whatever reason (at the peer side for example) or packets cannot be observed going in or out from LX2, a PCS loopback is a way to ensure that packets can be received without any issues at the DPNI/DPMAC level. From a clock perspective, there is no problem to use the PCS loop, because even if the peer is in a wrong state (it does not have a valid clock), the clock at the PCS (when in loopback) is recovered from the internal PLL.

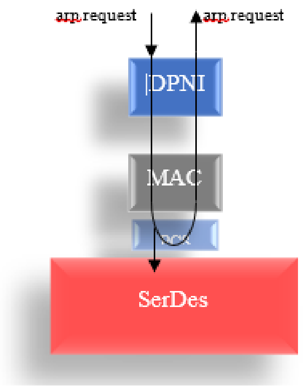


Figure 6. PCS loopback

Reading the PCS status register can yield 0x4 (link) or 0x0 (no link) and in both cases, the loopback should work correctly:

```
#load the MDIO binary

go 0x80300000 read 0x8c0b000 3 1

## Starting application at 0x80300000 ...
Example expects ABI version 9
Actual U-Boot ABI version 9
Internal MDIO read / write
argc = 5
operation = "read"
mac offset = "0x8c0b000"
dev addr = "3"
reg addr = "1"
mdio_cfg bsy set
mdio_data bsy set

#In this case there is no link
output data: 0x00000000

## Application terminated, rc = 0x0
```

NOTE

Run the above command multiple times to avoid latched data.

3.2.3.1 PCS loopback verification steps

The stages for running with traffic in PCS loopback are listed below. One observation is that before setting the PCS control register loopback bit (**bit 14**) , its contents must be read then set the loopback bit.

The changes applied in U-Boot and DPC from the previous section (Digital loopback) must be used here as well.

```
#The same images should be used for this case
#Note MC firmware must be started in order to be able to use the interfaces
#A log as below should be seen:

fsl-mc: Booting Management Complex ... SUCCESS
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x1)
Hit any key to stop autoboot:  0

#Display the env variable that loads the mdio binary

printenv elf_load
elf_load=dcache off; tftp 0x80300000 b37577/mdio.bin; dcache on;

run elf_load

#set the PCS in loopback mode. First read the content of the control register

#read the PCS control register
go 0x80300000 read 0x8c0b000 3 0
## Starting application at 0x80300000 ...
Example expects ABI version 9
Actual U-Boot ABI version 9
Internal MDIO read / write
argc = 5
operation = "read"
mac offset = "0x8c0b000"
```

```
dev addr = "3"
reg addr = "0"
mdio_cfg bsy set
mdio_data bsy set

output data: 0x0000204c

#set the loopback bit

=> go 0x80300000 write 0x8c0b000 3 0 0x604c
## Starting application at 0x80300000 ...
Example expects ABI version 9
Actual U-Boot ABI version 9
Internal MDIO read / write
argc = 6
operation = "write"
mac offset = "0x8c0b000"
dev addr = "3"
reg addr = "0"
data = "0x604c"
mdio_cfg bsy set
mdio_data bsy set


## Application terminated, rc = 0x0


#set ethact to be MAC2
setenv ethact DPMAC2@xlaui4

#run the ping command
=> ping 1.1.1.2

ldpaa_dpmac_bind, DPMAC Type= dpmac
ldpaa_dpmac_bind, DPMAC ID= 2
ldpaa_dpmac_bind, DPMAC State= 0
ldpaa_dpmac_bind, DPNI Type= dpni
ldpaa_dpmac_bind, DPNI ID= 0
ldpaa_dpmac_bind, DPNI State= 0
DPMAC link status: 1 - up
DPNI link status: 1 - up
Using DPMAC2@xlaui4 device
```

When the ping command is launched, do not expect to see an echo response. The purpose of this test is to receive the packets back, because the PCS is in a loopback. The command is let to run for a couple of seconds. Then it is interrupted by pressing CTRL + C. The next output or a similar one (DPNI counters can differ but must not be 0) should be displayed in the console:

```
DPNI counters ..
DPNI_CNT_ING_ALL_FRAMES= 2
DPNI_CNT_ING_ALL_BYTES= 120
DPNI_CNT_ING_MCAST_FRAMES= 0
DPNI_CNT_ING_MCAST_BYTES= 0
DPNI_CNT_ING_BCAST_FRAMES= 2
DPNI_CNT_ING_BCAST_BYTES= 120
DPNI_CNT_EGR_ALL_FRAMES= 2
DPNI_CNT_EGR_ALL_BYTES= 84
DPNI_CNT_EGR_MCAST_FRAMES= 0
DPNI_CNT_EGR_MCAST_BYTES= 0
DPNI_CNT_EGR_BCAST_FRAMES= 2
```

```
DPNI_CNT_EGR_BCAST_BYTES= 84
DPNI_CNT_ING_FILTERED_FRAMES= 0
DPNI_CNT_ING_DISCARDED_FRAMES= 0
DPNI_CNT_ING_NOBUFFER_DISCARDS= 0
DPNI_CNT_EGR_DISCARDED_FRAMES= 0
DPNI_CNT_EGR_CNF_FRAMES= 0
```

3.2.3.2 PCS loopback result interpretation

DPNI_CNT_ING_ALL_FRAMES and DPNI_CNT_EGR_ALL_FRAMES equal to 2.

It means that 2 ARP requests are sent and received back by the WRIOP port (DPNI).

The result excludes any issue in both DPMAC and DPNI.

In case of a negative result (if no packets were received at the DPNI level), a possible problem is at the MAC level. The MAC loopback cannot be investigated in the U-Boot (see Interface loopback modes). This does not exclude a PCS issue, although this would be a very unlikely situation.

3.3 U-Boot use case with 10G MAC

In this setup (Figure 7), a 10G MAC (MAC3) is connected to an AQR PHY that is further connected to a traffic generator device (or a workstation).

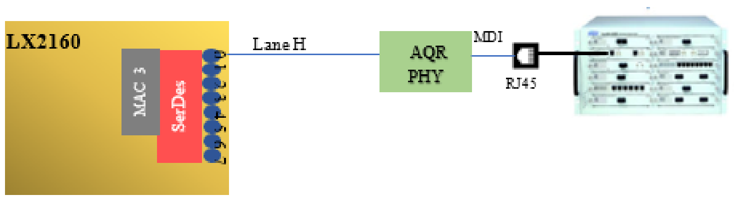


Figure 7. 10G MAC3 connected to AQR PHY (lane H)

Hypothesis:

MAC3 uses Lane H on the SerDes1 module to connect to the AQR PHY (LX2 RDB). The MAC is defined in the DPC file as follows:

```
[…]
board_info {
                ports {
                         mac@3 {
                                link_type = "MAC_LINK_TYPE_PHY";
                                enet_if="USXGMII";
                        };
[…]
```

The link type shows that the MAC is connected to a PHY. The connection is called “TYPE_PHY”.

If you want to send packets from the left side to test the connectivity, launch the ping command from the CLI.

Result:

The outcome is that no reply is received from the right side (or no packets are captured by the Traffic Generator). It is either because there is no link or because something happens with the packets on the way between the two peers. The following code snippet contains the user commands and the result.

```
fsl-mc: Booting Management Complex ... SUCCESS
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x1)
Hit any key to stop autoboot:  0
```

```
[…]
=> setenv ethact DPMAC3@xgmii
=> ping 1.1.1.2
Using DPMAC3@xgmii device
Abort
ping failed; host 1.1.1.2 is not alive
```

The only differences are that there is only one lane for MAC3 (lane H, base address 0x1EA0FA0) and its base address (0x8c0f000) is different from MAC2. For this use case, the debug steps listed in U-Boot use case with 40G MAC can be applied:

- Digital loopback sanity check
- Digital loopback with traffic
- PCS loopback with traffic

This section covers only the PHY loopbacks.

3.3.1 PHY loopbacks

There are three types of loopbacks covered:

- PCS loopback - the data is looped in the PCS and received up at the DPNI.
- PMA loopback - the data is looped in the PMA and received at the DPNI.
- PHY XS loopback - network loopback. The data sent from the peer is looped in the PHY XS and received by the peer. It is useful to validate the peer.

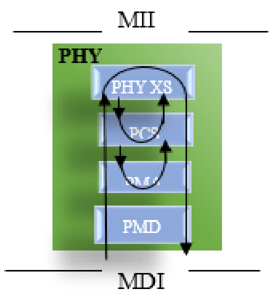


Figure 8. PHY loopback modes

NOTE

When testing the first two modes, there is no need to have a valid link with the right peer (the Traffic Generator).

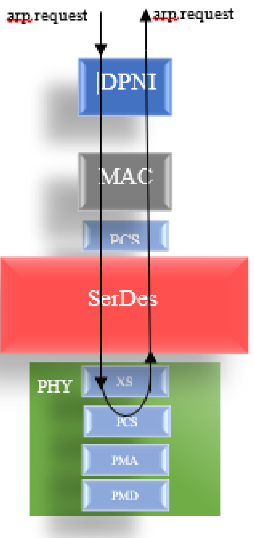
This document does not cover the technical details about these loopbacks. See the PHY data sheet for more information.

The loopback is configured using the U-Boot standard MDIO command. The PHY uses Clause 45 commands. It means that besides the PHY address and the register offset, the MMD device type must be provided as well (PCS, PMA, or PHY XS).

3.3.1.1 PHY PCS loopback

In this mode, when a ping command is executed, the ARP packets are generated from the U-Boot, sent down to the PHY (Figure 9), looped back in the PCS, and they reach back to the WRIOP port. All the packets are gated at the PCS level if no other option is set. It means that no data shows up at the peer side.

Figure 9. PHY PCS loopback



3.3.1.1.1 PHY PCS loopback verification steps

Keep the same changed U-Boot image from U-Boot use case with 40G MAC. The DPC file can be restored to its initial content, because the debug_link_check option has no effect when the MAC is defined as TYPE_PHY in the DPC (see DPC changes).

The steps for running this stage are:

```
#The same images should be used for this case
#Note MC firmware must be started in order to be able to use the interfaces
#A log as below should be seen:


fsl-mc: Booting Management Complex ... SUCCESS
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x1)
Hit any key to stop autoboot:  0

[…]

#set ethact to be MAC3

=> setenv ethact DPMAC3@xgmii

#list the PHYs and select the PHY that corresponds to MAC3

mdio list
FSL_MDIO0:
0 - Cortina CS4223 <--> DPMAC2@xlaui4
1 - AR8035 <--> DPMAC17@rgmii-id
2 - AR8035 <--> DPMAC18@rgmii-id
4 - Aquantia AQR107 <--> DPMAC3@xgmii
5 - Aquantia AQR107 <--> DPMAC4@xgmii
```

```
FSL_MDIO1:
0 - Inphi in112525_S03P <--> DPMAC6@25g-aui

#display mdio options to see the arguments of the command
=> mdio
mdio - MDIO utility commands

Usage:
mdio list                       - List MDIO buses
mdio read <phydev> [<devad>.]<reg> - read PHY's register at <devad>.<reg>
mdio write <phydev> [<devad>.]<reg> <data> - write PHY's register at <devad>.<reg>
mdio rx <phydev> [<devad>.]<reg> - read PHY's extended register at <devad>.<reg>
mdio wx <phydev> [<devad>.]<reg> <data> - write PHY's extended register at <devad>.<reg>
<phydev> may be:
   <busname>  <addr>
   <addr>
   <eth name>
<addr> <devad>, and <reg> may be ranges, e.g. 1-5.4-0x1f.

#set PHY 4 in PCS loopback – set bit 14 in PCS control register (offset 0)
#first read then set bit 14 to 1

mdio read 4 3.0
4 is not a known ethernet
Reading from bus FSL_MDIO0
PHY at address 4:
3.0 - 0x2040

mdio write 4 3.0 0x6040


#run the ping command
=> ping 1.1.1.2

ldpaa_dpmac_bind, DPMAC Type= dpmac
ldpaa_dpmac_bind, DPMAC ID= 3
ldpaa_dpmac_bind, DPMAC State= 0
ldpaa_dpmac_bind, DPNI Type= dpni
ldpaa_dpmac_bind, DPNI ID= 0
ldpaa_dpmac_bind, DPNI State= 0
DPMAC link status: 1 - up
DPNI link status: 1 - up
Using DPMAC3@xgmii device
```

When the ping command is launched, do not expect to see an echo response. The purpose of this test is to receive the packets back, because the PHY PCS is in a loopback. The command runs for a couple of seconds and it is then interrupted by pressing CTRL + C. The following (or similar) output (the DPNI counters can differ but they must not be 0) should be displayed in the console:

```
DPNI counters ..
DPNI_CNT_ING_ALL_FRAMES= 2
DPNI_CNT_ING_ALL_BYTES= 120
DPNI_CNT_ING_MCAST_FRAMES= 0
DPNI_CNT_ING_MCAST_BYTES= 0
DPNI_CNT_ING_BCAST_FRAMES= 2
DPNI_CNT_ING_BCAST_BYTES= 120
DPNI_CNT_EGR_ALL_FRAMES= 2
DPNI_CNT_EGR_ALL_BYTES= 84
DPNI_CNT_EGR_MCAST_FRAMES= 0
```

```
DPNI_CNT_EGR_MCAST_BYTES= 0
DPNI_CNT_EGR_BCAST_FRAMES= 2
DPNI_CNT_EGR_BCAST_BYTES= 84
DPNI_CNT_ING_FILTERED_FRAMES= 0
DPNI_CNT_ING_DISCARDED_FRAMES= 0
DPNI_CNT_ING_NOBUFFER_DISCARDS= 0
DPNI_CNT_EGR_DISCARDED_FRAMES= 0
DPNI_CNT_EGR_CNF_FRAMES= 0
```

3.3.1.1.2 PHY PCS loopback result interpretation

DPNI_CNT_ING_ALL_FRAMES and DPNI_CNT_EGR_ALL_FRAMES equal 2.

It means that two ARP requests were sent and received back by the WRIOP port (DPNI).

The result excludes any issue with the PHY or with the other layers from the LX2.

In case of a negative result (if no packets were received at the DPNI level), a possible issue is at the PHY level (if the tests for the other layers on the LX2 have passed). Consult the PHY data sheet. The test included here is for the AQR PHY. The U-Boot does have a driver for it (see the changes applied in U-Boot changes), which means that the PHY is configured when the DPAA2 MACs are probed (grepping for ldpaa_eth_netdev_init in the U-Boot source reveals the init_phy function) For a different PHY, check whether a driver is needed or the generic driver can apply the basic settings.

3.3.1.2 PHY PMA loopback

In this mode, when a ping command is executed, the ARP packets are generated from the U-Boot, sent to the PHY (Figure 10), looped back in the PMA, and they reach back to the WRIO port. All the packets are gated at the PMA level when no other option is set. It means that no data show up at the peer side.

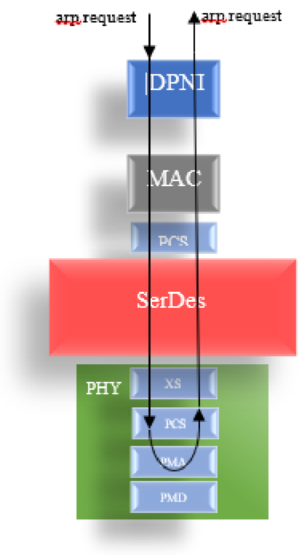


Figure 10. PHY PMA loopback

3.3.1.2.1 PHY PMA loopback verification steps

Keep the same changed U-Boot image from U-Boot use case with 40G MAC. The DPC file can be restored to its initial content, because the debug_link_check option has no effect when the MAC is defined as TYPE_PHY in the DPC (see DPC changes).

The steps to run this stage are:

```
#The same images should be used for this case
#Note MC firmware must be started in order to be able to use the interfaces
#A log as below should be seen:


fsl-mc: Booting Management Complex ... SUCCESS
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x1)
Hit any key to stop autoboot:  0

[…]

#set ethact to be MAC3

=> setenv ethact DPMAC3@xgmii

#list the PHYs and select the PHY that corresponds to MAC3

mdio list
FSL_MDIO0:
0 - Cortina CS4223 <--> DPMAC2@xlaui4
1 - AR8035 <--> DPMAC17@rgmii-id
2 - AR8035 <--> DPMAC18@rgmii-id
4 - Aquantia AQR107 <--> DPMAC3@xgmii
5 - Aquantia AQR107 <--> DPMAC4@xgmii
FSL_MDIO1:
0 - Inphi in112525_S03P <--> DPMAC6@25g-aui


#set PHY 4 in PMA loopback – set bit 0 in PMA(MMD offset 1) control register (offset 0)
#first read then set bit 0 to 1

mdio read 4 1.0

4 is not a known ethernet
Reading from bus FSL_MDIO0
PHY at address 4:
1.0    - 0x2040
2.0
mdio write 4 1.0 0x2041


#run the ping command
=> ping 1.1.1.2

ldpaa_dpmac_bind, DPMAC Type= dpmac
ldpaa_dpmac_bind, DPMAC ID= 3
ldpaa_dpmac_bind, DPMAC State= 0
ldpaa_dpmac_bind, DPNI Type= dpni
ldpaa_dpmac_bind, DPNI ID= 0
ldpaa_dpmac_bind, DPNI State= 0
DPMAC link status: 1 - up
DPNI link status: 1 - up
Using DPMAC3@xgmii device
```

When the ping command is launched, do not expect to see an echo response. The purpose of this test is to receive the packets, because the PHY PMA is in a loopback. The command runs for a couple of seconds and it is then interrupted by presing CTRL + C. The following (or similar) output (the DPNI counters can differ, but they must not be 0) should show up in the console:

```
DPNI counters ..
DPNI_CNT_ING_ALL_FRAMES= 2
DPNI_CNT_ING_ALL_BYTES= 120
DPNI_CNT_ING_MCAST_FRAMES= 0
DPNI_CNT_ING_MCAST_BYTES= 0
DPNI_CNT_ING_BCAST_FRAMES= 2
DPNI_CNT_ING_BCAST_BYTES= 120
DPNI_CNT_EGR_ALL_FRAMES= 2
DPNI_CNT_EGR_ALL_BYTES= 84
DPNI_CNT_EGR_MCAST_FRAMES= 0
DPNI_CNT_EGR_MCAST_BYTES= 0
DPNI_CNT_EGR_BCAST_FRAMES= 2
DPNI_CNT_EGR_BCAST_BYTES= 84
DPNI_CNT_ING_FILTERED_FRAMES= 0
DPNI_CNT_ING_DISCARDED_FRAMES= 0
DPNI_CNT_ING_NOBUFFER_DISCARDS= 0
DPNI_CNT_EGR_DISCARDED_FRAMES= 0
DPNI_CNT_EGR_CNF_FRAMES= 0
```

3.3.1.2.2 PHY PMA loopback result interpretation

DPNI_CNT_ING_ALL_FRAMES and DPNI_CNT_EGR_ALL_FRAMES equal 2.

It means that two ARP requests are sent and received by the WRIOP port (DPNI).

The result excludes any issues with the PHY or with the other layers from the LX2.

See also the comments in PHY PCS loopback result interpretation in case of a negative result.

3.3.1.3 PHY XS loopback

In this mode, the data is not sent from the LX2 side but from the Traffic Generator. The packets must be looped in the PHY XS layer and received in the Traffic Generator.

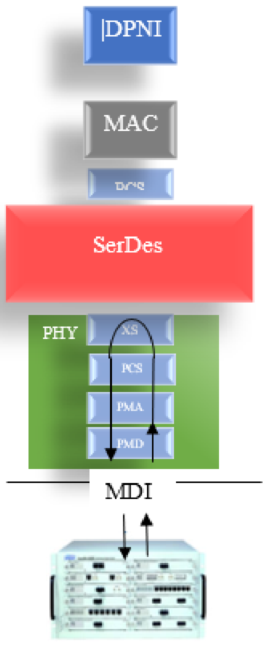


Figure 11. PHY XS loopback

3.3.1.3.1 PHY XS loopback verification steps

Keep the same changed U-Boot image from U-Boot use case with 40G MAC. The DPC file can be restored to its initial content, because the debug_link_check option has no effect when the MAC is defined as TYPE_PHY in the DPC (see DPC changes).

The steps to run this stage are:

```
#The same images should be used for this case
#Note MC firmware must be started in order to be able to use the interfaces
#A log as below should be seen:


fsl-mc: Booting Management Complex ... SUCCESS
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x1)
Hit any key to stop autoboot:  0

[…]

#list the PHYs and select the PHY that corresponds to MAC3

mdio list
FSL_MDIO0:
0 - Cortina CS4223 <--> DPMAC2@xlaui4
1 - AR8035 <--> DPMAC17@rgmii-id
2 - AR8035 <--> DPMAC18@rgmii-id
4 - Aquantia AQR107 <--> DPMAC3@xgmii
5 - Aquantia AQR107 <--> DPMAC4@xgmii
FSL_MDIO1:
0 - Inphi in112525_S03P <--> DPMAC6@25g-aui


#set PHY 4 in XS loopback – set bit 14 in XS(address 4) control register (offset 0)
#first read then set bit 14 to 1
```

```
mdio read 4 4.0
4 is not a known ethernet
Reading from bus FSL_MDIO0
PHY at address 4:
4.0    - 0x2040

mdio write 4 4.0 0x6040
```

Inject a couple of packets from the Traffic Generator. All of them must be received. If no packets are received, check the PHY data sheet and the PHY configuration. In this case, it is unlikely that there is an issue at the Traffic Generator level.

4 Troubleshooting in Linux OS

4.1 Introduction

Similarly to the U-Boot chapter, this section covers the cases in which the traffic is not received on an LX2 port as expected or the traffic does not reach the destination. The setup is similar to that shown in Figure 1.

Before delving into the use cases and considering the questions that are asked over time by multiple users regarding the DPAA2 architecture (for example; “What does a DPCON do?”, “Do I need a DPCON to receive interrupts?”, “What is the relation between DPCON, DPIO, and the GPPs?” “Is there any example of a poll mode driver without interrupts?”, and so on), the following pages show an overview of the DPAA2 Linux OS architecture. It is not mandatory to read this part for the debugging scenarios. It is recommended when the investigations move to the driver code.

4.2 DPAA2 Linux OS architecture overview

When choosing a programming mode for a DPAA2 network interface, there are two options:

- The poll mode is implemented in the U-Boot. In this case, the interrupts are not enabled. The U-Boot DPAA2 Ethernet driver polls continuously, as shown in the following snippet:

```
[...]
qbman_swp_pull(dflt_dpio->sw_portal , &pulldesc);
do {
  dq = qbman_swp_dqrr_next(swp);
} while (get_timer(time_start) < timeo && !dq);
[...]
```

A command such as ping is executed until the timeout expires or frames are received on the Rx queue.

NOTE

In this mode, manage the polling event to avoid the consumption of CPU cycles for polling only.

The polling pertains to the U-Boot, where only one core is active and the interface becomes available when a command that uses one of the ports (for example, ping) is executed in the CLI. When the command finishes (for example, when some packets are received/sent), the interface is disabled and the core goes back to poll for a CLI input. The poll mode is implemented in the DPAA2 ETH driver from DPDK as well.

- The interrupt mode is used in the Linux OS (together with the NAPI). This functionality is detailed in the following lines.

The ingress path with all its architectural components is shown in Figure 12.

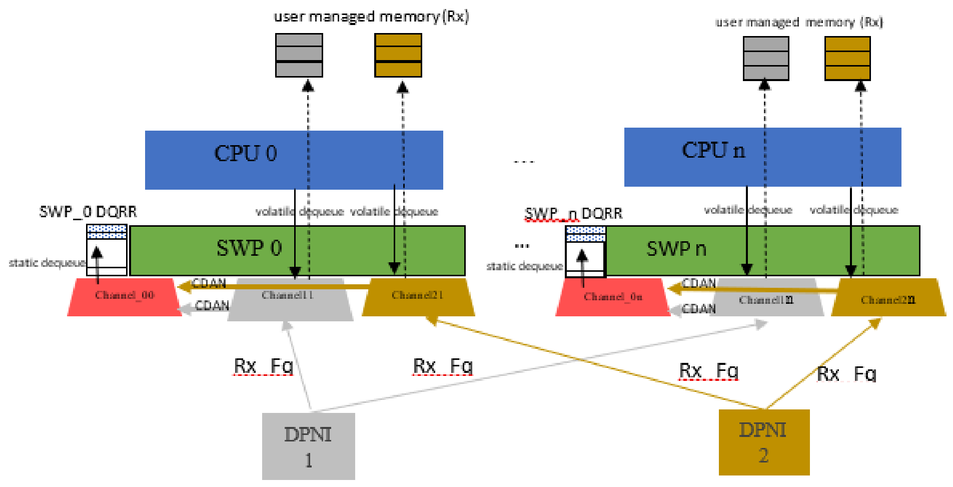


Figure 12. Ingress path

Each DPNI has its own resources (the colors indicate which objects belong to DPNI). It has a number of channels and each channel affines to a software portal. It has a number of frame queues affined for each core. If multiple traffic classes are defined on the ingress side, then there are multiple frame queues per core. If there is one traffic class per DPNI, then there is one queue per core. The traffic classes are not represented in Figure 12 for simplicity reasons.

Each Rx queue belongs to WQ1 (work queue - a DPAA2 hardware resource) in each channel, affined to each core: for DPNI1, the Rx queue for CPU0 belongs to WQ1 in Channel11 and the Rx queue for CPUn belongs to WQ1 in Channel 1n.

The frames received on DPNI1 are enqueued to queues belonging to Channel11 through to Channel1n and they are dequeued by CPU0 […] CPUn from the corresponding affined channel.

The following snippet from the DPAA2 ETH driver shows the connections between the channel, the WQ, and the frame queue:

```
/* setting the channel and the WQ for each Rx queue  */
        queue.destination.id = fq->channel->dpcon_id;
        queue.destination.type = DPNI_DEST_DPCON;
/*WQ1 – the second WQ from a group of 8. Each channel can have 2 or 8 WQs. WQ0 has the highest
priority */
        queue.destination.priority = 1;
        queue.user_context = (u64)(uintptr_t)fq;
        err = dpni_set_queue(priv->mc_io, 0, priv->mc_token,
                             DPNI_QUEUE_RX, fq->tc, fq->flowid,
                             DPNI_QUEUE_OPT_USER_CTX | DPNI_QUEUE_OPT_DEST,
                             &queue);
```

NOTE

The Linux OS version used in this document is here.

A channel is equivalent to a DPCON object. In Figure 12, the DPNI1 has DCPCON1 to DPCONn ("n" DPCONs in total) and DPNI2 has the same number of DPCONs. The "n" represents the number of software portals (cores) assigned to a DPNI. It can be a maximum of 16 for the LX2.

There is a special DPCON per each software portal - DPCON0. DPCON0 is equivalent with Channel00 on CPU0 with Channel0n on CPUn and is shared by DPNI1 and DPNI2.

Each DPNI channel is configured to deliver CDANs (Channel Data Availability Notifications) to DPCON0 (see Figure 12 for how the CDAN from each DPNI's private channel is delivered in the same DPCON0 on the same CPU. The DPCON0 is called the CDAN channel. The CDAN notification is delivered to the first WQ from the DPCON0 – WQ0 (the highest priority WQ).

These CDAN channels relay the CDANs into a software portal memory location called DQRR, where they are consumed and processed by the CPU.

The following snippet shows how the CDAN channels are configured for each online CPU:

```
for_each_online_cpu(i)
[…]
  /* Register DPCON notification with MC */
                dpcon_notif_cfg.dpio_id = nctx->dpio_id;
                dpcon_notif_cfg.priority = 0;
                dpcon_notif_cfg.user_ctx = nctx->qman64;
                err = dpcon_set_notification(priv->mc_io, 0,
                                             channel->dpcon->mc_handle,
                                             &dpcon_notif_cfg);
[…]
```

Each private channel is configured to deliver CDANs:

```
[…]
    channel = alloc_channel(priv);
                if (IS_ERR_OR_NULL(channel)) {
                        err = PTR_ERR_OR_ZERO(channel);
                        if (err != -EPROBE_DEFER)
                                dev_info(dev,
                                         "No affine channel for cpu %d and above\n", i);
                        goto err_alloc_ch;
                }
                priv->channel[priv->num_channels] = channel;
                nctx = &channel->nctx;
                nctx->is_cdan = 1;
[…]
```

For a clearer picture of the ingress flow, Figure 13 describes how a packet is delivered to the CPU and placed in the user managed memory.

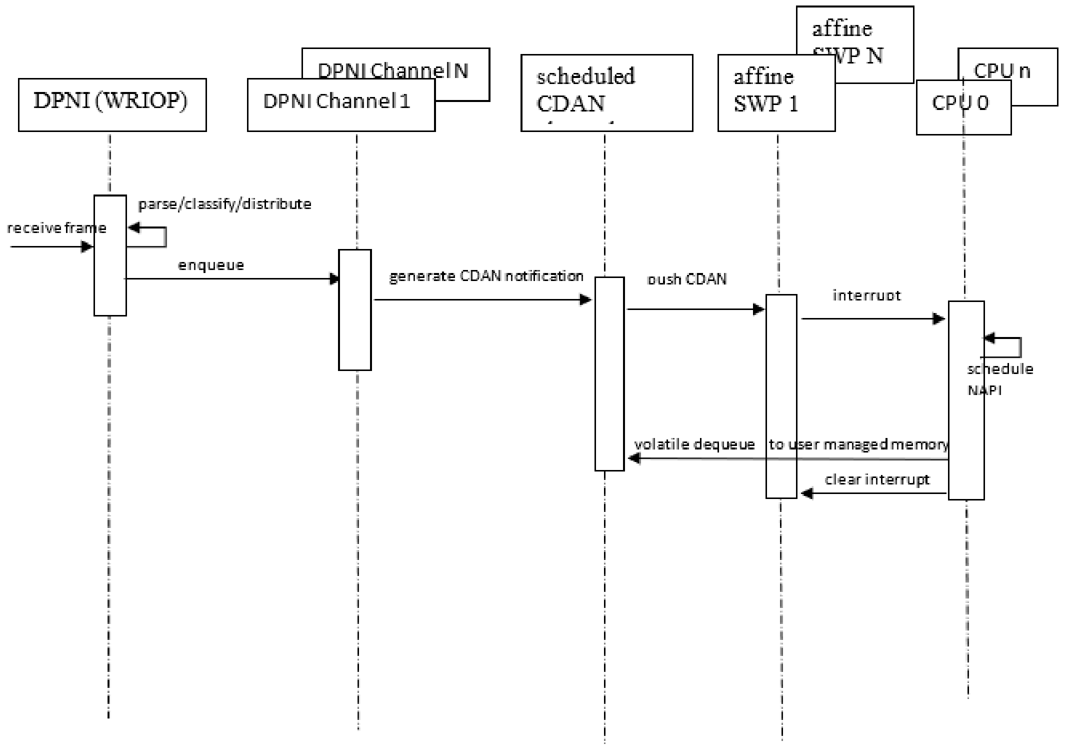


Figure 13. Packet diagram

A frame that arrives in a DPNI and passes the MAC filters (in CTLU) is subject to an advanced exact-match filter (also in the CTLU). If a frame matches one of the exact match table entries, it is steered to one of the Frame Queue Destination IDs. Otherwise, the frame enters a hash distribution stage, which delivers the frame to one of the CPUs (still via DPNI's queues) according to its determined traffic class. The frame (if not malformed or in other way erroneous) is eventually enqueued to a FQ placed in one of the DPNI's private channels (DPCON1 to DPCONn) configured in the volatile dequeue mode.

Each DPNI's private channel is configured to deliver CDAN notifications into a CDAN channel.

Each CDAN channel is affined to a CPU, but the CDAN channels are shared across multiple DPNIs. There are as many CDAN channels as there are CPUs and it is the same with the DPNI's private channels.

Unlike the DPNI's private channels, the shared CDAN channels are configured in the scheduled dequeue mode. They push the newly received CDANs as soon as the scheduling logic dictates. Each CDAN channel has only one SWP to service, which makes it CPU-affined.

When receiving the CDAN, the SWP raises an interrupt to the GPP. The portal interrupt handler consumes the CDAN and identifies the DPNI that produced the notification:

```
irqreturn_t dpaa2_io_irq(struct dpaa2_io *obj)
{
        const struct dpaa2_dq *dq;
        int max = 0;
        struct qbman_swp *swp;
        u32 status;
        swp = obj->swp;
        status = qbman_swp_interrupt_read_status(swp);
        if (!status)
                return IRQ_NONE;
[…]
       dq = qbman_swp_dqrr_next(swp);
        while (dq) {
                if (qbman_result_is_SCN(dq)) {
                        struct dpaa2_io_notification_ctx *ctx;
                        u64 q64;
                        q64 = qbman_result_SCN_ctx(dq);
                        ctx = (void *)(uintptr_t)q64;
                        ctx->cb(ctx);
                } else {
                        pr_crit("fsl-mc-dpio: Unrecognised/ignored DQRR entry\n");
                }
                qbman_swp_dqrr_consume(swp, dq);
                ++max;
                if (max > DPAA_POLL_MAX)
                        goto done;
                dq = qbman_swp_dqrr_next(swp);
        }
```

NOTE

When a CDAN is placed on its WQ, the CDAN generation is disabled automatically in the channel which generated the CDAN. No more CDANs are issued from this channel until it is reenabled by software.

The interrupt handler calls the DPNI CDAN callback (ctx->cb(ctx)) that schedules a NAPI instance:

```
static void cdan_cb(struct dpaa2_io_notification_ctx *ctx)
{
        struct dpaa2_eth_channel *ch;
        ch = container_of(ctx, struct dpaa2_eth_channel, nctx);
        /* Update NAPI statistics */
        ch->stats.cdan++;
        napi_schedule_irqoff(&ch->napi);
}
```

A volatile dequeue command from the NAPI instance is issued from the source channel of the CDAN into the DPNI's user-defined memory on a particular CPU:

```
do {
                err = pull_channel(ch);
                if (unlikely(err))
                        break;
                /* Refill pool if appropriate */
                refill_pool(priv, ch, priv->bpid);
                store_cleaned = consume_frames(ch, &fq);
[…]
} while (store_cleaned);
```

The CPU further consumes the frame:

```
static int consume_frames(struct dpaa2_eth_channel *ch,
                          struct dpaa2_eth_fq **src)
{
[…]
fq->consume(priv, ch, fd, fq);
```

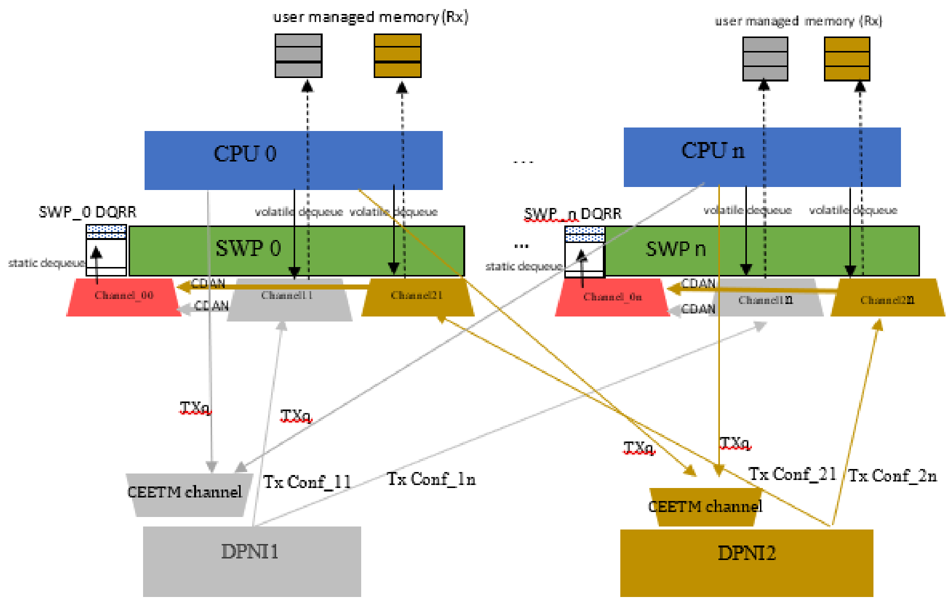
The CPU then sends it to the stack:

```
/* Main Rx frame processing routine */
static void dpaa2_eth_rx(struct dpaa2_eth_priv *priv,
                         struct dpaa2_eth_channel *ch,
                         const struct dpaa2_fd *fd,
                              struct dpaa2_eth_fq *fq)
{
[…]
if (frame_is_tcp(fd, fas))
                  napi_gro_receive(&ch->napi, skb);
         else
                  list_add_tail(&skb->list, ch->rx_list);
```

This is a very short description of the ingress path.

The egress path is shown in Figure 14.

Figure 14. Egress Path



Each DPNI has a private CEETM (Customer Edge Egress Traffic Management) channel with 16 class queues. The DPAA2 driver supports only 8 class queues and 1 CEETM channel. The number of Tx queues for each interface equals the number of Rx queues. Each inbound flow has a correspondent in the outbound direction.

The CPUs use the software portals to enqueue the frames in the Tx queues.

Each queue belongs to a traffic class (0..7), which gives the queue priority value, which is used when the final enqueue action is executed on the CPU.

The priority is determined as follows:

```
       queue_mapping = skb_get_queue_mapping(skb);
         if (net_dev->num_tc) {
                  prio = netdev_txq_to_tc(net_dev, queue_mapping);
```

The priority is used to select the corresponding Tx queue that belongs to the traffic class mapped to that priority. In the resulted queue, the CPU enqueues the data as follows:

```
for (i = 0; i < DPAA2_ETH_ENQUEUE_RETRIES; i++) {
                  err = priv->enqueue(priv, fq, &fd, prio, 1, NULL);
                  if (err != -EBUSY)
                          break;
```

```
        }
[…]
```

The above enqueue callback expands to:

```
err = dpaa2_io_service_enqueue_multiple_fq(fq->channel->dpio,
                                                    fq->tx_fqid[prio],
                                                    fd, num_frames);
```

The corresponding class queue is identified according to the priority value (in hardware 0 is the highest, 7 is the lowest) that selected the destination queue inside the CEETM channel. It is based on an internal CEETM mapping table.

Depending on how the frames are scheduled and sent to the physical port, it is up to the class queue scheduler (in the CEETM) to decide which class queue to pick from the 8 available. It decides according to their priorities. By default, the 8 class queues have a strict priority from 0 to 7.

For more details about the CEETM, see the LX2 reference manual.

After a frame is transmitted, the buffer occupied by it must be freed either into the buffer pool (each interface has a private buffer pool to which the frame is copied by the WRIOP from the internal FIFO) or to the allocator (in Linux OS).

Freeing the packet to the buffer pool can be done using an external buffer deallocation mechanism. If this option is enabled, a DPNI automatically frees a buffer to its buffer pool when the frame is transferred from its external buffer to the internal FIFO. It can be used on the forwarding path.

Nearly all the traffic locally originated must be freed to the allocator. This is done using the Tx confirmation mechanism. This mechanism implies a dedicated queue (one queue per core) and the following requirements apply:

- The traffic is spread across all CPUs. Each flow (CPU) has its own Tx confirmation queue, so no CPU processes the Tx confirmation traffic from the other CPUs.
- Due to the affinity of flows to the CPUs on each egress flow, the Tx confirmed buffers should be freed by the same CPU that allocated them.
- The Tx confirmation queues have a higher priority than any ingress queues. The Tx confirmation queues are special Rx queues. They work in a way similar to the Rx queues. The code that set the Rx queue is similar to the code that sets the Tx confirmation queues. The only significant difference is the WQ priority. Each Tx confirmation queue belongs to WQ0. Each WQ0 is in Channel11 to Channel1n when it belongs to DPNI1 or Channel21 to Channel2n when it belongs to DPNI2, together with WQ1 used for the Rx queues.

```
        queue.destination.id = fq->channel->dpcon_id;
        queue.destination.type = DPNI_DEST_DPCON;
        queue.destination.priority = 0;
        queue.user_context = (u64)(uintptr_t)fq;
        err = dpni_set_queue(priv->mc_io, 0, priv->mc_token,
                             DPNI_QUEUE_TX_CONFIRM, 0, fq->flowid,
                             DPNI_QUEUE_OPT_USER_CTX | DPNI_QUEUE_OPT_DEST,
                             &queue);
```

There is one error queue per DPNI configured in a similar way with the Tx confirmation and Rx queues. They belong to the same WQ1 as the Rx queues. By default, the error queue is disabled in the kernel configuration. In case of an error, the packets are discarded.

```
        q.destination.id = fq->channel->dpcon_id;
        q.destination.type = DPNI_DEST_DPCON;
        q.destination.priority = 1;
        q.user_context = (u64)fq;
        err = dpni_set_queue(priv->mc_io, 0, priv->mc_token,
                             DPNI_QUEUE_RX_ERR, 0, 0, q_opt, &q);
```

NOTE

The error queue is not shown in the figure. It uses the same CDAN mechanism as the Rx and Tx confirmation queues.

Now that the Rx and Tx path were described, some considerations on the interrupts and DPAA2 objects follow.

The MC logical objects can interrupt the CPU(s) by executing Message Interrupts (MSI). The MSIs are generated by writing specific values to the dedicated addresses. The addresses are mapped to the SoC interrupt controller's MSI mechanism, although the MC should not be aware of the implementation details of the MSI mechanism. The allocation of an MSI address and the data to write to that address are the responsibility of the CPU device driver, because they are system resources. The MC simply writes the specified data to the given address (assuming that it falls into the SoC CCSR access window) whenever there is a need to generate the associated interrupt.

The interrupts handling API is similar among all MC logical objects. Each logical object may define one or more interrupts. Each interrupt (IRQ) is associated with a pair of MSI address and data. Every interrupt can have up to 32 causes and the interrupt model supports masking/unmasking each cause independently. Grouping the causes into an interrupt is defined by the object's API. The error causes are grouped separately from normal runtime events.

In the case of DPNI, whenever a link change event is transmitted by the lower layers, the associated IRQ handler is called. In the DPAA2 ETH driver, the events that may trigger the interrupt are highlighted in the following snippet:

```
static irqreturn_t dpni_irq0_handler_thread(int irq_num, void *arg)
{
        u32 status = ~0;
        struct device *dev = (struct device *)arg;
        struct fsl_mc_device *dpni_dev = to_fsl_mc_device(dev);
        struct net_device *net_dev = dev_get_drvdata(dev);
        int err;
        err = dpni_get_irq_status(dpni_dev->mc_io, 0, dpni_dev->mc_handle,
                                  DPNI_IRQ_INDEX, &status);
        if (unlikely(err)) {
                netdev_err(net_dev, "Can't get irq status (err %d)\n", err);
                return IRQ_HANDLED;
        }
        if (status & DPNI_IRQ_EVENT_LINK_CHANGED)
                link_state_update(netdev_priv(net_dev));
        if (status & DPNI_IRQ_EVENT_ENDPOINT_CHANGED)
                set_mac_addr(netdev_priv(net_dev));
        return IRQ_HANDLED;}
```

For the DPMAC, the events that can trigger an interrupt are in the following snippet:

```
static irqreturn_t dpaa2_mac_irq_handler(int irq_num, void *arg)
{
[…]
        /* DPNI-initiated link configuration; 'ifconfig up' also calls this */
        if (status & DPMAC_IRQ_EVENT_LINK_CFG_REQ) {
                if (cmp_dpmac_ver(priv, DPMAC_LINK_AUTONEG_VER_MAJOR,
                                  DPMAC_LINK_AUTONEG_VER_MINOR) < 0)
                        err = dpmac_get_link_cfg(mc_dev->mc_io, 0,
                                                 mc_dev->mc_handle, &link_cfg);
                else
                        err = dpmac_get_link_cfg_v2(mc_dev->mc_io, 0,
                                                    mc_dev->mc_handle,
                                                    &link_cfg);
                if (unlikely(err))
                        goto out;
                configure_link(priv, &link_cfg);
        }
        if (status & DPMAC_IRQ_EVENT_LINK_DOWN_REQ)
```

```
                phy_stop(ndev->phydev);
        if (status & DPMAC_IRQ_EVENT_LINK_UP_REQ)
                phy_start(ndev->phydev);
```

The MC internal link manager periodically checks the connection state between objects (for example, in case of the DPNI, the connection state with a DPMAC, if the network interface is composed of a DPNI connected to a physical MAC) and when it receives an event (for example, the PHY lost the link on the line side or there is no link in the PCS for some reason), it propagates the state change to the objects and triggers an interrupt.

The sequence below describes the general flow of events from the point when the data value is written until the CPU receives the interrupt.

1. Data is written to the GIC ITS GITS_TRANSLATER register.

   There is only one ITS in the GIC-500. It means that all devices write to the same physical address to trigger interrupts. A non-spoofable "device id" is expressed as part of the transaction as well. In the case of DPAA2 devices/objects, all objects in the same container/DPRC share the same "device-id". For the Arm-based SoC, this is the same as the stream ID. The "data" value is an interrupt number.
2. The data and stream ID must pass through the SMMU unchanged.
3. The GIC ITS does a lookup of the "device id" in a device table to identify an Interrupt Translation Table (ITT)
4. The data value indexes into the ITT, identifying an interrupt table entry.
5. The table entry has a collection ID that identifies a collection table entry that points to the destination CPU. The table entry also identifies the physical LPI number.
6. With the destination CPU identified, the ITS writes to the level 0 redistributor for the destination CPU to trigger the physical LPI.
7. The interrupt occurs at the CPU.
8. The CPU reads GICC_IAR, getting the physical LPI number.

4.3 Data Path Layout (DPL)

To enable the DPAA2 network interfaces on a running Linux OS, load the DPL file during boot time (in the boot loader, before booting the kernel image).

The DPL describes a set of objects that are created when the system (Linux OS) is initialized.

The following are some aspects related to the DPL:

- The DPL is found at flash offset = 0xd00000/SD Card offset = 0x6800.
- The DPL is similar to the DTS. However, it does not describe the hardware attributes, but the initial topology of logical objects that the MC firmware should create.
- The DPL can be compiled using the "dtc" tool. A blob similar to the DTB is generated.
- The DPL is applied after the MC boots (it is not executed automatically): `fsl_mc apply dpl <dpl_addr>`.
- When the DPL is applied, all the objects from the U-Boot (DPNI, DPMAC, and so on) are deleted and the network interfaces are no longer available from the U-Boot. To use the interfaces in the boot loader again, reboot the board without applying the DPL.
- Some examples are at this link.

4.3.1 Booting Linux OS with a minimal DPL file

In troubleshooting contexts (those described in this document), it is recommended to use a DPL file that does not have any networking or infrastructure objects inside it (DPNI, DPMAC, DPCON, DPIO, and so on), but only management objects (DPMCPs and root container).

Such approach eliminates any types of configuration errors that may appear with multiple network interfaces that are configured to allocate as many resources as possible. For example, in a DPL, each DPNI has 8 traffic classes (each traffic class consumes 64 CTLU entries – 8 * 64 CTLU entries per interface) and 16 Rx queues (each queue belongs to a DPCON with 2 or 8 work queue channels). Other networking objects (DPDMUX, L2SW) may be also present.

After the Linux OS boots, errors may occur in a context like this due to insufficient resources (insufficient CTLU memory and insufficient work queue channels), if the resources defined for each networking component are not computed carefully. This leads to extra debugging effort together with the initial troubleshooting context.

That is why a minimal DPL should be deployed on the LX2 target.

The following snippet shows the content of a minimal DPL or an "empty DPL". This DPL is used for all the debugging use cases described in the following sections:

```
/dts-v1/;
/ {
 dpl-version = <10>;
 /****************************************************************
  * Containers
  ****************************************************************/
 containers {
  dprc@1 {
   compatible = "fsl,dprc";
   parent = "none";
   options = "DPRC_CFG_OPT_SPAWN_ALLOWED" , "DPRC_CFG_OPT_ALLOC_ALLOWED",
"DPRC_CFG_OPT_IRQ_CFG_ALLOWED";
   objects {
    /* ------------ DPMCPs --------------*/
    obj_set@dpmcp {
     type = "dpmcp";
     ids = <1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33
34 35 36 37 38 39 40>;
    };
   };
  };
 };
 /****************************************************************
  * Objects
  ****************************************************************/
 objects {
  /* ------------ DPMCP --------------*/
  dpmcp@1 {
   compatible="fsl,dpmcp";
  };
  dpmcp@2 {
   compatible="fsl,dpmcp";
  };
  dpmcp@3 {
   compatible="fsl,dpmcp";
  };
  dpmcp@4 {
   compatible="fsl,dpmcp";
  };
  dpmcp@5 {
   compatible="fsl,dpmcp";
  };
  dpmcp@6 {
   compatible="fsl,dpmcp";
  };
  dpmcp@7 {
   compatible="fsl,dpmcp";
  };
  dpmcp@8 {
   compatible="fsl,dpmcp";
  };
  dpmcp@9 {
   compatible="fsl,dpmcp";
  };
  dpmcp@10 {
   compatible="fsl,dpmcp";
  };
  dpmcp@11 {
   compatible="fsl,dpmcp";
  };
  dpmcp@12 {
   compatible="fsl,dpmcp";
  };
  dpmcp@13 {
   compatible="fsl,dpmcp";
  };
  dpmcp@14 {
   compatible="fsl,dpmcp";
  };
  dpmcp@15 {
   compatible="fsl,dpmcp";
  };
  dpmcp@16 {
   compatible="fsl,dpmcp";
  };
  dpmcp@17 {
   compatible="fsl,dpmcp";
  };
  dpmcp@18 {
   compatible="fsl,dpmcp";
  };
  dpmcp@19 {
   compatible="fsl,dpmcp";
  };
  dpmcp@20 {
  compatible="fsl,dpmcp";};
  dpmcp@21 {
  compatible="fsl,dpmcp";};
  dpmcp@22 {
  compatible="fsl,dpmcp";};
  dpmcp@23 {
  compatible="fsl,dpmcp";};
  dpmcp@24 {
  compatible="fsl,dpmcp";};
  dpmcp@25 {
  compatible="fsl,dpmcp";};
  dpmcp@26 {
  compatible="fsl,dpmcp";};
  dpmcp@27 {
  compatible="fsl,dpmcp";};
  dpmcp@28 {
  compatible="fsl,dpmcp";};
  dpmcp@29 {
  compatible="fsl,dpmcp";};
  dpmcp@30 {
  compatible="fsl,dpmcp";};
  dpmcp@31 {
  compatible="fsl,dpmcp";};
  dpmcp@32 {
  compatible="fsl,dpmcp";};
  dpmcp@33 {
  compatible="fsl,dpmcp";};
  dpmcp@34 {
  compatible="fsl,dpmcp";};
  dpmcp@35 {
  compatible="fsl,dpmcp";};
  dpmcp@36 {
  compatible="fsl,dpmcp";};
  dpmcp@37 {
  compatible="fsl,dpmcp";};
  dpmcp@38{
  compatible="fsl,dpmcp";};
  dpmcp@39 {
  compatible="fsl,dpmcp";};
  dpmcp@40 {
  compatible="fsl,dpmcp";};
 };
};
```

Generate the DPL binary blob from the above source and write it to the target board. This document does not describe any build steps. The steps are described here or in the *Layerscape Software Development Kit User Guide*.

The following snippets describe the steps for booting the Linux OS with an empty DPL. They show how to create a network interface and dump the running configuration into an output file that can be used as a final DPL:

```
#after the empty DPL has been deployed on the target, reboot. #boot the MC firmware
fsl-mc: Booting Management Complex ... SUCCESS
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x1)
Hit any key to stop autoboot:  0
#apply the dpl file and boot the kernel
fsl_mc apply dpl 0x20d00000;tftp $fdtaddr kernel_dtb;tftp $loadaddr kernel_img;  booti $loadaddr -
$fdtaddr
```

When the Linux OS console is available, create the DPMAC and DPNI objects and connect them:

```
restool dpmac create --mac-id=1
restool dprc assign dprc.1 --object=dpmac.1 --plugged=1

ls-addni -nq=16 -t=8 dpmac.1
Created interface: eth1 (object:dpni.0, endpoint: dpmac.1
ifconfig -a
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet6 fe80::204:9fff:fe05:c3c8  prefixlen 64  scopeid 0x20<link>
        ether 00:04:9f:05:c3:c8  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 14  bytes 1076 (1.0 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

NOTE

If the above command fails, set the debug level as below before creating any network interfaces. When the commands fail, check the MC console.

```
#display the ls-debug options
ls-debug -h
```

```
Usage: /usr/local/bin/ls-debug [OPTIONS]
The options are:
        -h, --help              -       This message.
        -ts, --timestamp=X      -       Enable/Disable timestamp printing, X is ON or OFF
        -c, --console=X         -       Enable/Disable printing in UART console, X is ON or OFF
        -l, --log=X             -       Enable/Disable printing in DDR log, X is ON or OF
        -u, --uart=X            -       Set UART id of the console, X = [0 - 4], 0 = OFF
        -ll, --level=X          -       Set logging level, X = [0 - 5]
                                                - 0: GLOBAL
                                                - 1: DEBUG
                                                - 2: INFO
                                                - 3: WARRNING
                                                - 4: ERROR
                                                - 5: CRITICAL
        -m, mem, --memory       -       Dump information about memory modules available
        dpxy.z                  -       Dump information about MC respective object
#run ls-debug with log level debug
ls-debug --log=on --console=on --level=2
dpdbg.0 created
DDR log printing ON
UART console printing ON
Log level set to 2
#display the mc console and see what errors are printed
cat `find /dev/ -name "*mc_console"`
```

If the network interface is created successfully, save the current running configuration:

```
#dump the running config into an output file that can be used as a DPL later on
restool dprc generate-dpl dprc.1 > running_dpl.dts
```

4.4 Linux OS use case with 40G MAC

In this setup (see Figure 15) a 40G MAC (MAC1) is connected directly (SerDes to SerDes) to another 40G MAC (MAC 1) or to another device that has a 40G port.

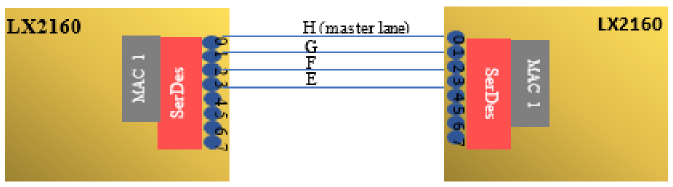


Figure 15. 40G MAC1 connected to 40G MAC1 (lanes E-H)

Hypothesis:

MAC1 uses lanes E-H on the SerDes1 module. The MAC is defined in the DPC file as follows:

```
[…]
board_info {
                ports {
                         mac@1 {
                                link_type = "MAC_LINK_TYPE_FIXED";
                        };
```

The link type shows that there is no transceiver between the MAC and the peer. The connection is considered “TYPE_FIXED”.

Send some packets from the left side to test the connectivity. For this purpose, launch a ping command from the Linux OS.

Result:

No reply is received from the right side either because there is no link or something happens with the packets on the way between the two peers. The following snippet contains the user commands and the result:

```
restool dpmac create --mac-id=1
restool dprc assign dprc.1 --object=dpmac.1 --plugged=1

ls-addni -nq=16 -t=8 dpmac.1
Created interface: eth1 (object:dpni.0, endpoint: dpmac.1

ifconfig eth1 1.1.1.1 up

#the other side is configured in a similar way execpt that its ip will be 1.1.1.2

#from the left side send a ping command
ping 1.1.1.2
PING 1.1.1.2 (1.1.1.2) 56(84) bytes of data.

PING 1.1.1.2 (1.1.1.2) 56(84) bytes of data.
From 1.1.1.1 icmp_seq=1 Destination Host Unreachable
From 1.1.1.1 icmp_seq=2 Destination Host Unreachable
From 1.1.1.1 icmp_seq=3 Destination Host Unreachable
```

Check various counters and attributes to verify the state of the interface:

```
#check the operational flags on the network interface

ifconfig eth1



#it can be observed that there is no RUNNING flag

eth1: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 1.1.1.1  netmask 255.0.0.0  broadcast 1.255.255.255
        inet6 fe80::7c9e:86ff:fed6:d0e6  prefixlen 64  scopeid 0x20<link>
        ether 7e:9e:86:d6:d0:e6  txqueuelen 1000  (Ethernet)
        RX packets 386  bytes 37388 (37.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 390  bytes 37708 (37.7 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

#if there is no RUNNING flag it means there is a link issue somewhere: check the DPMAC
#link status next

restool dpmac info dpmac.1

dpmac version: 4.7
dpmac object id/portal id: 1
plugged state: plugged
endpoint state: 0
endpoint: dpni.0, link is down
DPMAC link type: DPMAC_LINK_TYPE_FIXED
DPMAC ethernet interface: DPMAC_ETH_IF_XFI
maximum supported rate 40000 Mbps

#check also the DPNI link status
restool dpni info dpni.0
dpni version: 7.17
dpni id: 0
plugged state: plugged
endpoint state: 0
endpoint: dpmac.1, link is down
link status: 0 - down
mac address: 7e:9e:86:d6:d0:e6
dpni_attr.options value is: 0
```

In the above snippet, there is no link at the DPMAC level, DPNI level, and network device level (the RUNNING flag is missing on ETH1). There is either something incorrectly configured on the LX2 or the right-side peer has an issue.

If there is a valid link on the three components mentioned earlier, the next step is to check the counters at each level and see if there is a change:

```
#network interface counters

        RX packets 386  bytes 37388 (37.3 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 390  bytes 37708 (37.7 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

#MAC 1 counters
DPMAC link type: DPMAC_LINK_TYPE_FIXED
DPMAC ethernet interface: DPMAC_ETH_IF_XFI
maximum supported rate 40000 Mbps
Counters:
rx all frames: 390
rx frames ok: 390
rx frame errors: 0
rx frame discards: 0
rx u-cast: 354
rx b-cast: 0
rx m-cast: 36
rx 64 bytes: 0
rx 65-127 bytes: 390
rx 128-255 bytes: 0
rx 256-511 bytes: 0
rx 512-1023 bytes: 0
rx 1024-1518 bytes: 0
rx 1519-max bytes: 0
rx frags: 0
rx jabber: 0
rx align errors: 0
rx oversized: 0
rx pause: 0
rx bytes: 39268
tx frames ok: 390
tx u-cast: 354
tx m-cast: 36
tx b-cast: 0
tx frame errors: 0
tx undersized: 0
tx b-pause: 0
tx bytes: 39268

#DPNI0 counters
mac address: 7e:9e:86:d6:d0:e6
dpni_attr.options value is: 0
num_queues: 16
num_cgs: 1
```

```
num_rx_tcs: 1
num_tx_tcs: 1
mac_entries: 16
vlan_entries: 0
qos_entries: 0
fs_entries: 64
qos_key_size: 0
fs_key_size: 56
ingress_all_frames: 386
ingress_all_bytes: 37388
ingress_multicast_frames: 32
ingress_multicast_bytes: 2696
ingress_broadcast_frames: 0
ingress_broadcast_bytes: 0
egress_all_frames: 390
egress_all_bytes: 37708
egress_multicast_frames: 36
egress_multicast_bytes: 3016
egress_broadcast_frames: 0
egress_broadcast_bytes: 0
ingress_filtered_frames: 4
ingress_discarded_frames: 0
ingress_nobuffer_discards: 0
egress_discarded_frames: 0
egress_confirmed_frames: 390
```

Regardless of the result (there is no link or there is a link but the counters show no activity when running a ping), the first step to exclude any SerDes issues (either configuration or hardware design integration issues - if the LX2 SoC is part of a custom design) is to execute a sanity check.

4.4.1 SerDes Sanity check

The goal of this test is to check the PCS link status when the SerDes is configured in a digital loopback.

In this mode, the link must always be up. To rule out any potential problems that may be introduced by the MC firmware, the firmware must not be started in the validation procedure. This is equivalent to omitting the execution of the command that loads the MC binary file in the U-Boot environment and boots the Linux OS without any networking support:

```
#search through env variables and eliminate this command. Usually the mcinitcmd contains it. Remove
it from there and restart the board

#do not call fsl_mc start mc 0x20a00000  0x20e00000;

#do not apply the DPL as well

#Boot Linux
tftp $fdtaddr kernel_dtb;tftp $loadaddr kernel_img;  booti $loadaddr - $fdtaddr
```

After the Linux OS boots, check that the MC has not started by running:

```
restool -m
error: Did not find a device file
```

The fact that the MC is not loaded means that the SerDes configuration applied by the RCW did not undergo any further updates.

To set the SerDes in a digital loopback, execute the following script for lanes E-H (MAC1).

```
#Usage: loopback_serdes.sh <[A-D][E-H]> <[0x10000000] [0x00000000]>
./loopback_serdes.sh E-H 0x10000000
Set serdes loopback on lanes E-H
```

```
[0000] 0x01ea0ca0: 0x10000000 (written)
[0000] 0x01ea0da0: 0x10000000 (written)
[0000] 0x01ea0ea0: 0x10000000 (written)
[0000] 0x01ea0fa0: 0x10000000 (written)
```

The content of the script that sets the loopback for the SerDes is as follows:

```
cat ./loopback_serdes.sh
#Digital loopback script
if [ $# -eq 0 ]
then
           echo "No arguments supplied. Usage: loopback_serdes.sh <[A-D][E-H]> <[0x10000000]
[0x00000000]>"
elif [ $1 == 'A-D' ]
then
        echo "Set serdes loopback on lanes A-D"
        ./iomem w32 0x1EA08A0 $2
        ./iomem w32 0x1EA09A0 $2
        ./iomem w32 0x1EA0AA0 $2
        ./iomem w32 0x1EA0BA0 $2
elif [ $1 == 'E-H' ]
then
        echo "Set serdes loopback on lanes E-H"
        ./iomem w32 0x1EA0CA0 $2
        ./iomem w32 0x1EA0DA0 $2
        ./iomem w32 0x1EA0EA0 $2
        ./iomem w32 0x1EA0FA0 $2
else
        echo "Usage: loopback_serdes.sh <[A-D][E-H]> <[0x10000000] [0x00000000]>"
fi
```

NOTE

The IOMEM utility is not part of the LSDK package. DEVMEM or DEVMEM2 can be used instead.

The last operation is to check that a valid link is present in the PCS. It does not matter what the status of the link was before this sanity check.

Verifying the link implies to read the PCS status register through the internal MDIO interface of MAC1. This was done in the U-Boot using the tool described in Internal MDIO command implementation.

In the Linux OS case, a script that implements the internal MDIO access is executed:

```
#Reading the PCS Status register 1; must be read several times because bit 2 is latched low

#./mdio_read_c45.sh <MAC base address> <MMD address> <register address> [external phy address]


./mdio_read_c45.sh 0x8c07000 3 1


DIV=0x0000145c
[0000] 0x08c07030: 0x0000145c (written)
[0000] 0x08c07034: 0x00000003 (written)
[0000] 0x08c0703c: 0x00000001 (written)
[0000] 0x08c07034: 0x00008003 (written)

DATA=0x00000004
 #./mdio_read_c45.sh <MAC base address> <MMD address> <register address> [external phy address]
./mdio_read_c45.sh 0x8c07000 3 1
DIV=0x0000145c
```

```
[0000] 0x08c07030: 0x0000145c (written)
[0000] 0x08c07034: 0x00000003 (written)
[0000] 0x08c0703c: 0x00000001 (written)
[0000] 0x08c07034: 0x00008003 (written)
DATA=0x00000004
```

The contents of the script that implements the internal MDIO access can be used as a reference for other accesses on the internal MDIO. In this case, the PCS device is accessed, which is equivalent to MMD address 3. To configure other devices or registers provide their corresponding values as arguments to the script.

The last argument of the script (which is optional) can be used to access registers for external PHY devices using Clause 45.

```
#Script contents
cat ./mdio_read_c45.sh
#Internal MDIO Clause 45 read
MEMAC1_MDIO_CFG=0x30
MEMAC1_MDIO_CTRL=0x34
MEMAC1_MDIO_DATA=0x38
MEMAC1_MDIO_ADDR=0x3c
MEMAC1_MDIO_CFG=$(($1 + $MEMAC1_MDIO_CFG))
MEMAC1_MDIO_CTRL=$(($1 + $MEMAC1_MDIO_CTRL))
MEMAC1_MDIO_DATA=$(($1 + $MEMAC1_MDIO_DATA))
MEMAC1_MDIO_ADDR=$(($1 + $MEMAC1_MDIO_ADDR))
PHY=$2
if [ -z "$4" ]
then
        EXT_PHY=0
else
        EXT_PHY=$4
fi
PHY=$((($EXT_PHY << 5) | $PHY))
REG=$3
val=$((`./iomem r32 $MEMAC1_MDIO_CFG | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val & 0xFF80))
val=$(($val | 0x1C))
val=$(($val | 0x40))
printf "\nDIV=0x%08x\n"  $val
./iomem w32 $MEMAC1_MDIO_CFG $val
val=$((`./iomem r32 $MEMAC1_MDIO_CFG | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val & 0x1))
while [ "$val" -ne 0 ]
do
               echo "MDIO_CFG_BSY "
               val=$((`./iomem r32 $MEMAC1_MDIO_CFG | awk -F':' '{print substr($2,0,12)}'`))
               val=$(($val & 0x1))
               sleep 1
done
./iomem w32 $MEMAC1_MDIO_CTRL $PHY
./iomem w32 $MEMAC1_MDIO_ADDR $REG
val=$((`./iomem r32 $MEMAC1_MDIO_CFG | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val & 0x1))
while [ "$val" -ne 0 ]
do
               echo "MDIO_CFG_BSY "
               val=$((`./iomem r32 $MEMAC1_MDIO_CFG | awk -F':' '{print substr($2,0,12)}'`))
               val=$(($val & 0x1))
               sleep 1
done
val=$PHY
val=$(($val | 0x8000))
```

```
./iomem w32 $MEMAC1_MDIO_CTRL $val
val=$((`./iomem r32 $MEMAC1_MDIO_DATA | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val & 0x80000000))
while [ "$val" -ne 0 ]
do
               echo "MDIO_CFG_BSY "
               val=$((`./iomem r32 $MEMAC1_MDIO_DATA | awk -F':' '{print substr($2,0,12)}'`))
               val=$(($val & 0x80000000))
               sleep 1
done
val=$((`./iomem r32 $MEMAC1_MDIO_DATA | awk -F':' '{print substr($2,0,12)}'`))
printf "\nDATA=0x%08x\n"  $val
```

4.4.1.1 SerDes sanity check digital loopback result interpretation

The data from the output of the *mdio_read_c45.sh* file is interpreted as follows: bit 2 from the PCS status register means "Receive link status".

In this case, the link is up, which is the expected result (0x00000004).

NOTE

This bit is latched low, which means that the previous command script must be run several times to get the current value.

The result shows that in digital loopback mode, without running the MC firmware, there is a valid link in the PCS. This excludes a configuration issue on the SerDes. If the **"Receive Link Status"** bit is not set, this points either to a configuration issue (RCW for example) or something in the design (decoupling caps, AVDD out of spec, and so on).

4.4.1.2 SerDes sanity with parallel loopback

Another option to validate that there is no issue at the SerDes level is to apply a parallel loopback – tx_clk to tx_clk case (see Interface loopback modes). The tx_clk is used to clock both sides of the FIFO (used to transfer data from Tx to Rx) and it is bypassed through to the rx_clk.

The steps for executing in this mode are identical to those from the digital loopback. The main point is that the MC is not booted:

```
#search through env variables and eliminate this command. Usually the mcinitcmd contains it. Remove
it from there and restart the board
#do not call fsl_mc start mc 0x20a00000  0x20e00000;
#do not apply the DPL as well
#Boot Linux
tftp $fdtaddr kernel_dtb;tftp $loadaddr kernel_img;  booti $loadaddr - $fdtaddr
```

To set the SerDes in a parallel loopback, the following script should be executed for lanes E-H (MAC1):

```
#Usage: parallel_loopback.sh [A-D]|[E-H]
#content was redirected to /dev/null because of long output
./parallel_loopback_tx_clk_tx_clk.sh E-H 2>&1 > /dev/null
```

The script details are as follows:

```
#This script sets parallel loopback using tx_clk to tx_clk
cat ./parallel_loopback_tx_clk_tx_clk.sh
if [ $# -eq 0 ]
then
        echo "Usage: parallel_loopback.sh [A-D]|[E-H]"
  exit
elif [ $1 == 'A-D' ]
then
```

```
        echo "Set serdes parallel loopback on lanes A-D"
  LNNRRSTCTL=0x1ea0840
  LNNTRSTCTL=0x1ea0820
  LNNTCSR0=0x1ea08a0
  LNNTTLCR1=0x1EA0884
  LNATTLCR2=0x1EA0888
elif [ $1 == 'E-H' ]
then
        echo "Set serdes parallel loopback on lanes A-D"
  LNNRRSTCTL=0x1ea0c40
  LNNTRSTCTL=0x1ea0c20
  LNNTCSR0=0x1ea0ca0
  LNNTTLCR1=0x1EA0c84
  LNATTLCR2=0x1EA0c88
else
        echo "Usage: parallel_loopback.sh [A-D]|[E-H]"
  exit
fi
LNNRRSTCTL=0x1ea0c40
LNNTRSTCTL=0x1ea0c20
LNNTCSR0=0x1ea0ca0
LNNTTLCR1=0x1EA0c84
LNATTLCR2=0x1EA0c88
echo "Steps to activate this mode(Read the prev values from specified regs and set/unset  the
corresponding bits. \
1. Assert low lnx_(m)_rx_reset_b, lnx_(m)_tx_reset_b and lnx_(m)_rptr_fifo_reset_b."
echo "SerDes Lane m TX Reset Control Register (LNATRSTCTL) -bit 5"
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val & ~0x00000020))
./iomem w32 $LNNTRSTCTL $val
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTRSTCTL $val
LNNTRSTCTL=$(($LNNTRSTCTL + 0x100))
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val & ~0x00000020))
./iomem w32 $LNNTRSTCTL $val
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTRSTCTL $val
LNNTRSTCTL=$(($LNNTRSTCTL + 0x100))
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val & ~0x00000020))
./iomem w32 $LNNTRSTCTL $val
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTRSTCTL $val
LNNTRSTCTL=$(($LNNTRSTCTL + 0x100))
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val & ~0x00000020))
./iomem w32 $LNNTRSTCTL $val
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTRSTCTL $val
LNNTRSTCTL=$(($LNNTRSTCTL - 0x300))
echo "SerDes Lane m Test Control/Status Register 0 (LNATCSR0) -bit 10"
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val & ~0x00000400))
./iomem w32 $LNNTCSR0 $val
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTCSR0 $val
LNNTCSR0=$(($LNNTCSR0 + 0x100))
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val & ~0x00000400))
```

```
./iomem w32 $LNNTCSR0 $val
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTCSR0 $val
LNNTCSR0=$(($LNNTCSR0 + 0x100))
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val & ~0x00000400))
./iomem w32 $LNNTCSR0 $val
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTCSR0 $val
LNNTCSR0=$(($LNNTCSR0 + 0x100))
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val & ~0x00000400))
./iomem w32 $LNNTCSR0 $val
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTCSR0 $val
LNNTCSR0=$(($LNNTCSR0 - 0x300))
echo "2. Assert lnx_(m)_srds_lpbk_sel[1:0] = 2’b11 -bits 29-28"
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val | 0x30000000))
./iomem w32 $LNNTCSR0 $val
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTCSR0 $val
LNNTCSR0=$(($LNNTCSR0 + 0x100))
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
/*this means tx clk tx clk*/
val=$(($val | 0x30000000))
# val=$(($val & 0xEFFFFFFF))
./iomem w32 $LNNTCSR0 $val
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTCSR0 $val
LNNTCSR0=$(($LNNTCSR0 + 0x100))
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val | 0x30000000))
# val=$(($val & 0xEFFFFFFF))
./iomem w32 $LNNTCSR0 $val
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTCSR0 $val
LNNTCSR0=$(($LNNTCSR0 + 0x100))
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val | 0x30000000))
# val=$(($val & 0xEFFFFFFF))
./iomem w32 $LNNTCSR0 $val
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTCSR0 $val
LNNTCSR0=$(($LNNTCSR0 - 0x300))
echo "6. Negate high lnx_(m)_tx_reset_b. -bit 5"
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val | 0x00000020))
./iomem w32 $LNNTRSTCTL $val
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTRSTCTL $val
LNNTRSTCTL=$(($LNNTRSTCTL + 0x100))
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val | 0x00000020))
./iomem w32 $LNNTRSTCTL $val
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTRSTCTL $val
LNNTRSTCTL=$(($LNNTRSTCTL + 0x100))
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val | 0x00000020))
```

```
./iomem w32 $LNNTRSTCTL $val
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTRSTCTL $val
LNNTRSTCTL=$(($LNNTRSTCTL + 0x100))
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val | 0x00000020))
./iomem w32 $LNNTRSTCTL $val
val=$((`./iomem r32 $LNNTRSTCTL | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTRSTCTL $val
LNNTRSTCTL=$(($LNNTRSTCTL - 0x300))
echo "7. Negate high lnx_(m)_rptr_fifo_reset_b. -bit 10"
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val | 0x00000400))
./iomem w32 $LNNTCSR0 $val
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTCSR0 $val
LNNTCSR0=$(($LNNTCSR0 + 0x100))
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val | 0x00000400))
./iomem w32 $LNNTCSR0 $val
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTCSR0 $val
LNNTCSR0=$(($LNNTCSR0 + 0x100))
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val | 0x00000400))
./iomem w32 $LNNTCSR0 $val
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTCSR0 $val
LNNTCSR0=$(($LNNTCSR0 + 0x100))
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val | 0x00000400))
./iomem w32 $LNNTCSR0 $val
val=$((`./iomem r32 $LNNTCSR0 | awk -F':' '{print substr($2,0,12)}'`))
printf "\nLNNRRSTCTL=0x%08x val=0x%08x\n"  $LNNTCSR0 $val
LNNTCSR0=$(($LNNTCSR0 - 0x300))
```

Verifying the link implies reading the PCS status register through the internal MDIO interface of MAC1 in the same way as in the digital loopback case:

```
#Reading the PCS Status register 1
./mdio_read_c45.sh 0x8c07000 3 1
DIV=0x0000145c
[0000] 0x08c07030: 0x0000145c (written)
[0000] 0x08c07034: 0x00000003 (written)
[0000] 0x08c0703c: 0x00000001 (written)
[0000] 0x08c07034: 0x00008003 (written)
DATA=0x00000004
```

The output of the *mdio_read_c45.sh* shows that the link is up (it is similar to a digital loopback).

NOTE

This bit is latched low, which means that the previous command script must be run several times to get the current value.

The result shows that in the parallel loopback mode, without running the MC firmware, there is a valid link in the PCS. This excludes a configuration issue on the SerDes. If the "Receive Link Status" bit is not set, it points either to a configuration issue (RCW) or something in the design (decoupling caps, AVDD out of specification, and so on).

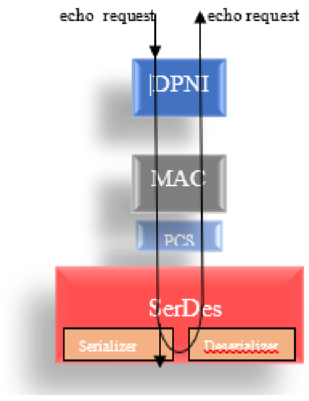
4.4.2 Digital loopback

The goal of this test is to show that the traffic (for example, echo requests) can be successfully sent and received by the DPNI interface (frames are reflected in the serializer/deserializer modules of the SerDes).

In a digital loopback, the serial transmitted data is looped back into the serial receiver path. The data is gated at Rx side only. On the Tx side, the data can be received on the remote peer.

The traffic flow is shown in Figure 16.

Figure 16. Digital loopback



The previous section demonstrates that a valid link is always present in the digital loopback/parallel loopback.

The goal in this situation is to validate that the transmitted packets are looped and received back.

To eliminate any possible configuration problems related to link propagation between DPMAC and DPNI, some changes are applied in the DPC file.

NOTE

There is no need for any change at the U-Boot level when validating the network interfaces functionality in the Linux OS. All the changes applied in the U-Boot section must be ignored in the Linux OS sections.

The DPC changes are as follows:

```
[…]
board_info {
                ports {
                         mac@1 {
                                link_type = "MAC_LINK_TYPE_FIXED";
                                debug_link_check="off";
                        };
[…]
```

Use this option in a debug scenario (for TYPE_FIXED MACs only), when you want to put the MAC or PCS into a loopback for testing purposes. If there is a problem at the peer or SERDES levels, a "link down" event propagates to the DPNI. In this situation, the Rx/Tx queues are disabled by the software layer (Linux OS DPAA2 driver) and no traffic can be sent/received. When this variable is set to "off", the link is always reported as up, even if it is not. Traffic can be sent/received by the WRIOP ports, making the PCS/MAC loopback a valid setup. This way you can validate that packets can be received/sent at the PCS/MAC level.

4.4.2.1 Applying the DPC

After the DPC changes are executed, generate the firmware image and the DPC binary blob and write them to the target board. This document does not describe any build steps. The steps are described here or in the *Layerscape Software Development Kit User Guide*.

There is an alternative to modify the DPC without deploying it into the flash on the running target.

This operation must be applied before starting the MC firmware. It saves the effort for compiling the DPC, deploying it, and flashing it to the board.

```
#Do not start the MC firmware.(remove “fsl_mc start mc 0x20a00000  0x20e00000” from env and reboot
the target, if the MC was already started)
#Assuming that the DPC is at address 0x20e00000 in flash (mapped to DDR), move the DPC to
#a new DDR address (to be able to dynamically change it)
fdt addr 0x20e00000
fdt move 0x20e00000 0x85000000 0x5000
#Add the debug_link_check option
fdt addr 0x85000000
fdt rm /board_info/ports/mac@1
fdt mknode /board_info/ports mac@1
fdt set    /board_info/ports/mac@1 link_type  "MAC_LINK_TYPE_FIXED";
fdt set    /board_info/ports/mac@1 debug_link_check  "off";
#start the MC using the DPC from the new DDR address
fsl_mc start mc 0x20a00000 0x85000000;
```

4.4.2.2 Verification steps

The stages for running with traffic in the digital loopback mode are as follows:

```
#after the firmware and DPC have been deployed on the target, reboot.(for dynamically changing the
DPC without rebooting and flashing it, see previous section) #Note MC firmware must be started in
order to be able to use the interfaces  #A log as below should be seen:
fsl-mc: Booting Management Complex ... SUCCESS
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x1)
Hit any key to stop autoboot:  0
#apply the dpl file and boot the kernel;
fsl_mc apply dpl 0x20d00000;tftp $fdtaddr kernel_dtb;tftp $loadaddr kernel_img;  booti $loadaddr -
$fdtaddr
```

The DPL applied is the "empty DPL" shown in Booting Linux with a minimal DPL file.

When the Linux OS console is available, create the DPMAC and DPNI objects and connect them:

```
restool dpmac create --mac-id=1
restool dprc assign dprc.1 --object=dpmac.1 --plugged=1
ls-addni -nq=16 -t=8 dpmac.1
Created interface: eth1 (object:dpni.0, endpoint: dpmac.1)
```

Assign an IP to the interface and add a static ARP entry that represents the destination that is pinged in the loopback mode:

```
ifconfig eth1 1.1.1.1 up

arp -s 1.1.1.2 0:0:0:0:0:1 dev eth1


ifconfig eth1
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 1.1.1.1  netmask 255.0.0.0  broadcast 1.255.255.255
        inet6 fe80::cc24:15ff:fec8:b1b3  prefixlen 64  scopeid 0x20<link>
        ether ce:24:15:c8:b1:b3  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 9  bytes 726 (726.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

NOTE

The RUNNING flag does not necessarily mean that a physical link is valid when the debug_link_check option is set to off in the DPC file. This option permits the interface to send or receive traffic on its queues without stopping them at the Linux OS stack level if a physical link is not present (for debugging purposes). A valid link is determined only by reading the PCS status register.

The next step is to set the SerDes in a digital loopback for lanes E-H (MAC1).

```
#Usage: loopback_serdes.sh <[A-D][E-H]> <[0x10000000] [0x00000000]>
./loopback_serdes.sh E-H 0x10000000
Set serdes loopback on lanes E-H
[0000] 0x01ea0ca0: 0x10000000 (written)
[0000] 0x01ea0da0: 0x10000000 (written)
[0000] 0x01ea0ea0: 0x10000000 (written)
[0000] 0x01ea0fa0: 0x10000000 (written)
```

Verifying the link implies reading the PCS status register through the internal MDIO interface of MAC1:

```
#Reading the PCS Status register 1; must be read several times because bit 2 is latched low
#./mdio_read_c45.sh <MAC base address> <MMD address> <register address> [external phy address]
./mdio_read_c45.sh 0x8c07000 3 1
DIV=0x0000145c
[0000] 0x08c07030: 0x0000145c (written)
[0000] 0x08c07034: 0x00000003 (written)
[0000] 0x08c0703c: 0x00000001 (written)
[0000] 0x08c07034: 0x00008003 (written)
DATA=0x00000004
```

In a digital loopback, the link status should always be up (0x00000004).

The stages for running with traffic in the digital loopback mode are as follows:

```
#Launch in background tcpdump tcpdump -i eth1 icmp&
#Send two echo requests to destination 1.1.1.2. Try the command several times.
ping -c 2 1.1.1.2 -i 0.1
PING 1.1.1.2 (1.1.1.2) 56(84) bytes of data.
10:35:23.732991 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 1, length 64
10:35:23.733037 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 1, length 64
10:35:23.833876 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 2, length 64
10:35:23.833909 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 2, length 64
^C
--- 1.1.1.2 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 100ms
```

TCPDUMP captures four echo requests, although the command sent only two.

4.4.2.3 Digital loopback result interpretation

The output of the TCPDUMP shows four frames, which is the expected result. Two frames are the echo requests sent on the Tx and captured by the TCPDUMP. The other two frames are the same packets reflected back (looped) by the SerDes digital loopback.

The result excludes any issue in the DPMAC/DPNI/SerDes. The use case in question is Linux use case with 40G MAC). It can be concluded that on the LX2 **DPNI - DPMAC - SerDes** path, traffic can be sent/received.

In case of a negative result, if no packets were received in the Linux OS stack (this means that only two packets were seen in the TCPDUMP), a possible problem would be at the SerDes serializer (though very unlikely). The next stage checks the SerDes parallel loopback. The parallel interface comes right after the serializer.

You can optionally check the different counters on DPMAC and DPNI:

```
#check DPMAC counters
restool dpmac info dpmac.1
#check DPNI counters
restool dpni info dpni.0
```

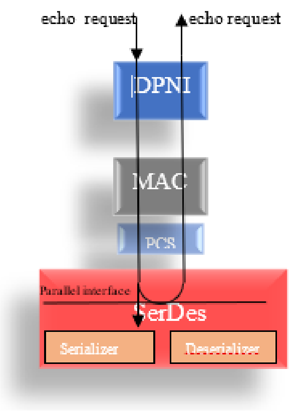
4.4.3 Parallel loopback

The goal of this test is to validate that traffic (for example, echo requests) can be successfully sent and received by the DPNI interface (reflected in the parallel interface of the SerDes).

On the parallel transmitter side, the data is transmitted and it reaches the remote peer. While on the receiver, the data is gated.

The traffic flow is shown in Figure 17.

Figure 17. Parallel loopback



The goal is to validate that the transmitted packets are looped and received back. It is similar to the digital loopback.

The same DPC changes (as those applied in a digital loopback) are kept also for this situation.

```
[…]
board_info {
                ports {
                         mac@1 {
                                link_type = "MAC_LINK_TYPE_FIXED";
                                debug_link_check="off";
                        };
[…]
```

```
Do not start the MC firmware. Remove "fsl_mc start mc 0x20a00000 0x20e00000" from the ENV and reboot
the target if the MC is already started.
#Assuming that the DPC is at address 0x20e00000 in flash (mapped to DDR), move the DPC to #a new DDR
address (to be able to dynamically change it)
fdt addr 0x20e00000
fdt move 0x20e00000 0x85000000 0x5000
#Add the debug_link_check option
fdt addr 0x85000000
fdt rm /board_info/ports/mac@1
fdt mknode /board_info/ports mac@1
fdt set   /board_info/ports/mac@1 link_type  "MAC_LINK_TYPE_FIXED";
fdt set   /board_info/ports/mac@1 debug_link_check  "off";
```

```
#start the MC using the DPC from the new DDR address
fsl_mc start mc 0x20a00000 0x85000000;
```

4.4.3.1 Verification steps

The stages for running with traffic in the parallel loopback mode are as follows:

```
#after the firmware and DPC have been deployed on the target, reboot.(for dynamically changing the
DPC without rebooting and flashing it, see previous section)
#Note MC firmware must be started in order to be able to use the interfaces
#A log as below should be seen:
fsl-mc: Booting Management Complex ... SUCCESS
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x1)
Hit any key to stop autoboot:  0
#apply the dpl file and boot the kernel;
fsl_mc apply dpl 0x20d00000;tftp $fdtaddr kernel_dtb;tftp $loadaddr kernel_img;  booti $loadaddr -
$fdtaddr
```

The DPL applied is the "empty DPL" defined in Booting Linux with a minimal DPL file.

When the Linux OS console is available, create the DPMAC and DPNI objects and connect them:

```
restool dpmac create --mac-id=1
restool dprc assign dprc.1 --object=dpmac.1 --plugged=1
ls-addni -nq=16 -t=8 dpmac.1
Created interface: eth1 (object:dpni.0, endpoint: dpmac.1)
```

Assign an IP to the interface and add a static ARP entry that represents the destination that is pinged in the loopback mode:

```
ifconfig eth1 1.1.1.1 up

arp -s 1.1.1.2 0:0:0:0:0:1 dev eth1


ifconfig eth1
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 1.1.1.1  netmask 255.0.0.0  broadcast 1.255.255.255
        inet6 fe80::cc24:15ff:fec8:b1b3  prefixlen 64  scopeid 0x20<link>
        ether ce:24:15:c8:b1:b3  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 9  bytes 726 (726.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

NOTE

The RUNNING flag does not mean that a physical link is valid when the debug_link_check option is set to "off" in the DPC file. This option permits the interface to send or receive traffic on its queues without stopping them at the Linux OS stack level if a physical link is not present for debugging purposes. A valid link is determined only by reading the PCS status register.

The next step is to set the SerDes in a parallel loopback for lanes E-H (MAC1).

```
#Usage: parallel_loopback.sh [A-D]|[E-H]

#content was redirected to /dev/null because of long output

./parallel_loopback_tx_clk_tx_clk.sh E-H 2>&1 > /dev/null
```

The previous sections show that a valid link is always present in a digital loopback/parallel loopback.

It means that after applying the parallel loopback, a valid link must be observed in the PCS status register:

```
#Reading the PCS Status register 1; must be read several times because bit 2 is latched low
#./mdio_read_c45.sh <MAC base address> <MMD address> <register address> [external phy address]
./mdio_read_c45.sh 0x8c07000 3 1
DIV=0x0000145c
[0000] 0x08c07030: 0x0000145c (written)
[0000] 0x08c07034: 0x00000003 (written)
[0000] 0x08c0703c: 0x00000001 (written)
[0000] 0x08c07034: 0x00008003 (written)
DATA=0x00000004
```

The stages for running with traffic in the parallel loopback mode are as follows:

```
#Launch in background tcpdump

tcpdump -i eth1 icmp&

#Send two echo requests to destination 1.1.1.2. Try the command several times.

ping -c 2 1.1.1.2 -i 0.1
PING 1.1.1.2 (1.1.1.2) 56(84) bytes of data.
10:35:23.732991 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 1, length 64
10:35:23.733037 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 1, length 64
10:35:23.833876 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 2, length 64
10:35:23.833909 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 2, length 64
^C
--- 1.1.1.2 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 100ms
```

You can see that the TCPDUMP captures four echo requests, although the command sent only two requests.

4.4.3.2 Parallel loopback result interpretation

The output of the TCPDUMP shows four frames, which is the expected result. Two frames are the echo requests sent on the Tx and captured by the TCPDUMP. The other two are the same packets reflected back (looped) by the parallel interface loopback.

The result excludes any issue in the DPMAC/DPNI/SerDes. The use case in question is Linux use case with 40G MAC. On the LX2 DPNI - DPMAC - SerDes path, traffic can be sent/received.

In case of a negative result (this means that only two packets are seen in the TCPDUMP), a possible problem would be at the PCS (though very unlikely). The next stage checks the PCS loopback mode.

You can optionally check the different counters on DPMAC and DPNI:

```
#check DPMAC counters
restool dpmac info dpmac.1
#check DPNI counters
restool dpni info dpni.0
```

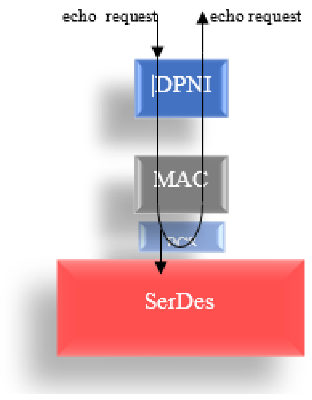
4.4.4 PCS loopback

In the PCS loopback, the data is transmitted normally. On the Rx side, the data is discarded at the PMA level (from a SerDes perspective). This time the echo requests are looped inside the PCS and received in the Linux OS stack.

NOTE

It is not mandatory to have a link in the PCS when testing this configuration. If there is a link problem for whatever reason (for example, at the peer side) or packets cannot be observed going in or out from the LX2, the PCS loopback is a way to ensure that packets can be received without any issues at the DPNI/DPMAC level. From a clock perspective, there is no problem to use the PCS loop, because even if the peer is in a wrong state (it does not have a valid clock), the clock at the PCS (when in a loopback) is recovered from the internal PLL.

Figure 18. PCS loopback



The goal is to check that the transmitted packets are looped and received back. It is similar to the parallel and digital loopbacks.

The same DPC changes as those applied in the digital loopback/parallel loopback should be kept also for this situation:

```
[…]
board_info {
                ports {
                         mac@1 {
                                link_type = "MAC_LINK_TYPE_FIXED";
                                debug_link_check="off";
                        };
[…]
```

```
Do not start the MC firmware. Remove “fsl_mc start mc 0x20a00000 0x20e00000” from the ENV and reboot
the target when the MC is already started.
#Assuming that the DPC is at address 0x20e00000 in flash (mapped to DDR), move the DPC to
#a new DDR address (to be able to dynamically change it)
fdt addr 0x20e00000
fdt move 0x20e00000 0x85000000 0x5000
#Add the debug_link_check option
fdt addr 0x85000000
fdt rm /board_info/ports/mac@1
fdt mknode /board_info/ports mac@1
fdt set    /board_info/ports/mac@1 link_type  "MAC_LINK_TYPE_FIXED";
fdt set    /board_info/ports/mac@1 debug_link_check  "off";
#start the MC using the DPC from the new DDR address
fsl_mc start mc 0x20a00000 0x85000000;
```

4.4.4.1 Verification steps

The stages for running with traffic in the PCS loopback mode are as follows:

```
#after the firmware and DPC have been deployed on the target, reboot.(for dynamically changing the
DPC without rebooting and flashing it, see previous section)
```

```
#Note MC firmware must be started in order to be able to use the interfaces
#A log as below should be seen:

fsl-mc: Booting Management Complex ... SUCCESS
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x1)
Hit any key to stop autoboot:  0

#apply the dpl file and boot the kernel;
fsl_mc apply dpl 0x20d00000;tftp $fdtaddr kernel_dtb;tftp $loadaddr kernel_img;  booti $loadaddr -
$fdtaddr
```

The DPL applied is the "empty DPL" described in Booting Linux with a minimal DPL file.

When the Linux OS console is available, create the DPMAC and DPNI objects and connect them:

```
restool dpmac create --mac-id=1
restool dprc assign dprc.1 --object=dpmac.1 --plugged=1
ls-addni -nq=16 -t=8 dpmac.1
Created interface: eth1 (object:dpni.0, endpoint: dpmac.1)
```

Assign an IP to the interface and add a static ARP entry that represents the destination that is pinged in the loopback mode:

```
ifconfig eth1 1.1.1.1 up

arp -s 1.1.1.2 0:0:0:0:0:1 dev eth1


ifconfig eth1
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 1.1.1.1  netmask 255.0.0.0  broadcast 1.255.255.255
        inet6 fe80::cc24:15ff:fec8:b1b3  prefixlen 64  scopeid 0x20<link>
        ether ce:24:15:c8:b1:b3  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 9  bytes 726 (726.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

NOTE

The RUNNING flag does not mean that a physical link is valid when the debug_link_check option is set to "off" in the DPC file. This option permits the interface to send or receive traffic in its queues without stopping them at the Linux OS stack level when a physical link is not present for debug purposes. A valid link is determined only by reading the PCS status register.

In the PCS loopback mode, it is not necessary to have a link up set in the PCS status register. That is why the PCS status register is not interrogated.

The next step is to set the PCS of MAC1 in a loopback:

```
#Read the PCS control register (offet 1)
./mdio_read_c45.sh 0x8c07000 3 0
DIV=0x0000145c
[0000] 0x08c07030: 0x0000145c (written)
[0000] 0x08c07034: 0x00000003 (written)
[0000] 0x08c0703c: 0x00000000 (written)
[0000] 0x08c07034: 0x00008003 (written)
DATA=0x0000204c
# Set PCS control register loopback bit  (bit 14) #./mdio_write_c45.sh <MAC base address> <MMD
address> <register address> <data> [external phy address]
./mdio_write_c45.sh 0x8c07000 3 0 0x604c
```

```
DIV=0x0000145c
[0000] 0x08c07030: 0x0000145c (written)
[0000] 0x08c07034: 0x00000003 (written)
[0000] 0x08c0703c: 0x00000000 (written)
[0000] 0x08c07038: 0x0000604c (written)
#Check that the data was written
./mdio_read_c45.sh 0x8c07000 3 0
DIV=0x0000145c
[0000] 0x08c07030: 0x0000145c (written)
[0000] 0x08c07034: 0x00000003 (written)
[0000] 0x08c0703c: 0x00000000 (written)
[0000] 0x08c07034: 0x00008003 (written)
DATA=0x0000604c
```

The contents of the script that implements the internal MDIO writes can be used as a reference for other accesses on the internal MDIO. In this case, the PCS device was accessed, which is equivalent with MMD address 3. To configure other devices or registers, provide their corresponding values as arguments to the script.

The last argument of the script (which is optional) can be used to write to registers that belong to external PHY devices using Clause 45.

```
#Script contents
cat ./mdio_write_c45.sh
#Internal MDIO Clause 45 write
MEMAC1_MDIO_CFG=0x30
MEMAC1_MDIO_CTRL=0x34
MEMAC1_MDIO_DATA=0x38
MEMAC1_MDIO_ADDR=0x3c
MEMAC1_MDIO_CFG=$(($1 + $MEMAC1_MDIO_CFG))
MEMAC1_MDIO_CTRL=$(($1 + $MEMAC1_MDIO_CTRL))
MEMAC1_MDIO_DATA=$(($1 + $MEMAC1_MDIO_DATA))
MEMAC1_MDIO_ADDR=$(($1 + $MEMAC1_MDIO_ADDR))
DATA=$4
PHY=$2
if [ -z "$5" ]
then
 EXT_PHY=0
else
 EXT_PHY=$5
fi
PHY=$((($EXT_PHY << 5) | $PHY))
REG=$3
val=$((`./iomem r32 $MEMAC1_MDIO_CFG | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val & 0xFF80))
val=$(($val | 0x1C))
val=$(($val | 0x40))
printf "\nDIV=0x%08x\n"  $val
./iomem w32 $MEMAC1_MDIO_CFG $val
val=$((`./iomem r32 $MEMAC1_MDIO_CFG | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val & 0x1))
while [ "$val" -ne 0 ]
do
               echo "MDIO_CFG_BSY "
               val=$((`./iomem r32 $MEMAC1_MDIO_CFG | awk -F':' '{print substr($2,0,12)}'`))
               val=$(($val & 0x1))
               sleep 1
done
./iomem w32 $MEMAC1_MDIO_CTRL $PHY
./iomem w32 $MEMAC1_MDIO_ADDR $REG
val=$((`./iomem r32 $MEMAC1_MDIO_CFG | awk -F':' '{print substr($2,0,12)}'`))
```

```
val=$(($val & 0x1))
while [ "$val" -ne 0 ]
do
               echo "MDIO_CFG_BSY "
               val=$((`./iomem r32 $MEMAC1_MDIO_CFG | awk -F':' '{print substr($2,0,12)}'`))
               val=$(($val & 0x1))
               sleep 1
done
./iomem w32 $MEMAC1_MDIO_DATA $DATA
val=$((`./iomem r32 $MEMAC1_MDIO_DATA | awk -F':' '{print substr($2,0,12)}'`))
val=$(($val & 0x80000000))
while [ "$val" -ne 0 ]
do
               echo "MDIO_CFG_BSY "
               val=$((`./iomem r32 $MEMAC1_MDIO_DATA | awk -F':' '{print substr($2,0,12)}'`))
               val=$(($val & 0x80000000))
               sleep 1
done
```

The stages for running with traffic in the PCS loopback mode are as follows:

```
#Launch in background tcpdump tcpdump -i eth1 icmp&
#Send two echo requests to destination 1.1.1.2. Try the command several times.
ping -c 2 1.1.1.2 -i 0.1
PING 1.1.1.2 (1.1.1.2) 56(84) bytes of data.
10:35:23.732991 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 1, length 64
10:35:23.733037 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 1, length 64
10:35:23.833876 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 2, length 64
10:35:23.833909 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 2, length 64
^C
--- 1.1.1.2 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 100ms
```

The TCPDUMP captures four echo requests, although the command sent only two.

4.4.4.2 PCS loopback result interpretation

The output of the TCPDUMP shows four frames, which is the expected result. Two frames are the echo requests sent on the Tx and captured by the TCPDUMP. The other two frames are the same packets reflected back (looped) in the PCS.

The result excludes any issues in DPMAC/DPNI/PCS. The use case in question is Linux use case with 40G MAC. On the LX2 **DPNI - DPMAC - SerDes** path, traffic can be sent/received.

In case of a negative result (this means that only two packets are seen in the TCPDUMP), a possible problem would be at the MAC level.

The next stage sets the MAC1 in the loopback mode.

You can optionally check the different counters on DPMAC and DPNI:

```
#check DPMAC counters
restool dpmac info dpmac.1
#check DPNI counters
restool dpni info dpni.0
```

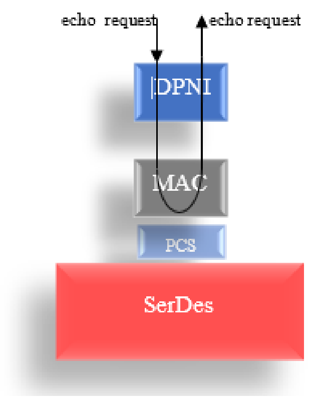
4.4.5 MAC loopback

In the MAC loopback, the MAC is isolated from the PCS. The MAC transmit is returned to MAC receive. No data is transmitted to external interfaces.

NOTE

It is not mandatory to have a link in the PCS when testing this configuration. If there is a link problem for whatever reason (at the peer side for example) or packets cannot be observed going in or out from the LX2, the MAC loopback ensures that at the DPNI/DPMAC level, packets can be received without any issues.

Figure 19. MAC loopback



The goal is to validate that the transmitted packets are looped and received back. It is similar to the PCS, parallel, and digital loopbacks.

The same DPC changes as those applied in the digital loopback/parallel loopback should be kept also for this situation:

```
[…]
board_info {
                ports {
                         mac@1 {
                                link_type = "MAC_LINK_TYPE_FIXED";
                                debug_link_check="off";
                        };
[…]
```

```
Do not start the MC firmware. Remove "fsl_mc start mc 0x20a00000 0x20e00000" from the ENV and reboot
the target when the MC is already started.
#Assuming that the DPC is at address 0x20e00000 in flash (mapped to DDR), move the DPC to
#a new DDR address (to be able to dynamically change it)
fdt addr 0x20e00000
fdt move 0x20e00000 0x85000000 0x5000
#Add the debug_link_check option
fdt addr 0x85000000
fdt rm /board_info/ports/mac@1
fdt mknode /board_info/ports mac@1
fdt set    /board_info/ports/mac@1 link_type  "MAC_LINK_TYPE_FIXED";
fdt set    /board_info/ports/mac@1 debug_link_check  "off";
#start the MC using the DPC from the new DDR address
fsl_mc start mc 0x20a00000 0x85000000;
```

4.4.5.1 Verification steps

The stages for running with traffic in the MAC loopback mode are as follows:

```
#after the firmware and DPC have been deployed on the target, reboot.(for dynamically changing the
DPC without rebooting and flashing it, see previous section)
#Note MC firmware must be started in order to be able to use the interfaces
```

```
#A log as below should be seen:

fsl-mc: Booting Management Complex ... SUCCESS
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x1)
Hit any key to stop autoboot:  0

#apply the dpl file and boot the kernel;
fsl_mc apply dpl 0x20d00000;tftp $fdtaddr kernel_dtb;tftp $loadaddr kernel_img;  booti $loadaddr -
$fdtaddr
```

The DPL applied is the "empty DPL" described in Booting Linux with a minimal DPL file.

When the Linux OS console is available, create the DPMAC and DPNI objects and connect them:

```
restool dpmac create --mac-id=1
restool dprc assign dprc.1 --object=dpmac.1 --plugged=1
ls-addni -nq=16 -t=8 dpmac.1
Created interface: eth1 (object:dpni.0, endpoint: dpmac.1)
```

Assign an IP to the interface and add a static ARP entry that represents the destination that is pinged in the loopback mode:

```
ifconfig eth1 1.1.1.1 up

arp -s 1.1.1.2 0:0:0:0:0:1 dev eth1


ifconfig eth1
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 1.1.1.1  netmask 255.0.0.0  broadcast 1.255.255.255
        inet6 fe80::cc24:15ff:fec8:b1b3  prefixlen 64  scopeid 0x20<link>
        ether ce:24:15:c8:b1:b3  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 9  bytes 726 (726.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

NOTE

The RUNNING flag does not mean that a physical link is valid when the debug_link_check option is set to "off" in the DPC file. This option permits the interface to send or receive traffic on its queues without stopping them at the Linux OS stack level if a physical link is not present for debugging purposes. A valid link is determined only by reading the PCS status register.

In the MAC loopback mode, it is not necessary to have a link up set in the PCS status register. MAC1 is isolated from the PCS. That is why the PCS status register is not interrogated.

The next step is to set the MAC1 in the loopback is as follows:

```
#Read the CEMAC_COMMAND_CONFIG register of MAC1 – offset 0x8 from MAC1 base address
./iomem r32 0x8c07008
[0000] 0x08c07008: 0x08020013 (read)
#Set bit 21 to 1 – bit 21 (CGMII loopback enable)
./iomem w32 0x8c07008 0x8020413
#Verify the write operation
./iomem r32 0x8c07008
[0000] 0x08c07008: 0x08020413 (read)
```

The stages for running with traffic in the MAC loopback mode are as follows:

```
#Launch in background tcpdump

tcpdump -i eth1 icmp&

#Send two echo requests to destination 1.1.1.2. Try the command several times.

ping -c 2 1.1.1.2 -i 0.1
PING 1.1.1.2 (1.1.1.2) 56(84) bytes of data.
10:35:23.732991 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 1, length 64
10:35:23.733037 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 1, length 64
10:35:23.833876 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 2, length 64
10:35:23.833909 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 2, length 64
^C
--- 1.1.1.2 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 100ms
```

The TCPDUMP captures four echo requests, although the command sent only two.

4.4.5.2 MAC loopback result interpretation

The output of the TCPDUMP shows four frames, which is the expected result. Two frames are the echo requests sent on the Tx and captured by the TCPDUMP. The other two are the same packets reflected back (looped) in the MAC.

The result excludes any issues in DPMAC/DPNI. The use case in question is Linux use case with 40G MAC. On the LX2 **DPNI - DPMAC - SerDes** path, traffic can be sent/received.

In case of a negative result (this means that only two packets are seen in the TCPDUMP), a possible problem would be that the packets are dropped at the MAC/DPNI level.

You can optionally check the different counters on DPMAC and DPNI as follows:

```
#check DPMAC counters
restool dpmac info dpmac.1
#check DPNI counters
restool dpni info dpni.0
```

A separate section describes a troubleshooting scenario in which the error queue is enabled. If the error queue is active and packets are discarded at the DPNI level due to an error, it is very likely that they are enqueued in this queue. By default, this queue is disabled in the DPAA2 Ethernet driver.

4.5 Linux use case with 10G MAC

In this setup (Figure 19), a 10G MAC (MAC4) is connected to an AQR PHY that further connects to the traffic generator device (or station).

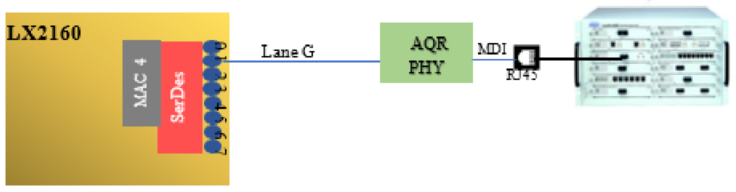


Figure 20. 10G MAC4 connected to AQR PHY (lane G)

Hypothesis:

MAC4 uses lanes G on the SerDes1 module to connect to the AQR PHY (LX2 RDB). The MAC is defined in the DPC file as follows:

```
[…]
board_info {
                ports {
                         mac@4 {
                                link_type = "MAC_LINK_TYPE_PHY";
          enet_if="USXGMII";
                        };
[…]
```

The link type shows that the MAC is connected to a PHY. The connection is “TYPE_PHY”.

Send some packets from the left side to test the connectivity. For this purpose, launch a ping command from the Linux OS.

Result:

The outcome is that no reply is received from the right side or that no packets are captured by the traffic generator, either because there is no link or something happens with the packets on the way between the two peers. The following snippet contains the user commands and the result:

```
restool dpmac create --mac-id=4
restool dprc assign dprc.1 --object=dpmac.4 --plugged=1

ls-addni -nq=16 -t=8 dpmac.4
Created interface: eth1 (object:dpni.0, endpoint: dpmac.4)

ifconfig eth1 1.1.1.1 up

#the other side is configured in a similar way execpt that its ip will be 1.1.1.2

#from the left side send a ping command
ping 1.1.1.2
PING 1.1.1.2 (1.1.1.2) 56(84) bytes of data.

PING 1.1.1.2 (1.1.1.2) 56(84) bytes of data.
From 1.1.1.1 icmp_seq=1 Destination Host Unreachable
From 1.1.1.1 icmp_seq=2 Destination Host Unreachable
From 1.1.1.1 icmp_seq=3 Destination Host Unreachable
```

The same debug steps as those in Linux use case with 40G MAC can be applied to this use case. The only differences are that there is only one lane for MAC4 (lane G, base address 0x1EA0EA0) and its base address (0x8c13000) is different from MAC1 (see Table 1 *MAC base addresses*):

- Digital loopback and parallel loopback sanity checks.
- Digital loopback with traffic.
- Parallel loopback with traffic.
- PCS loopback with traffic. For 10G PCS, the data is gated also on the Tx side, which means that no traffic is seen on the remote peer.
- MAC loopback with traffic.

This section covers only the PHY loopbacks.

4.5.1 PHY loopbacks

The following loopbacks are identical to those used in the U-Boot debugging scenario:

- PCS loopback (data is looped in the PCS and received at the DPNI).
- PMA loopback (data is looped in the PMA and received at the DPNI).

- PHY XS loopback (network loopback). The data sent from the peer is looped in the PHY XS and received by the peer. It is useful to validate the peer.

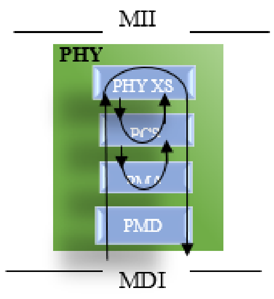


Figure 21. PHY loopbacks

NOTE

When testing the first two modes, the link status changes at the PHY level may update the operational status of the network device (the RUNNING flag) and generate a link down event on the DPNI and DPMAC.

To avoid it, the following modifications are applied on the kernel image before booting:

```
diff --git a/drivers/staging/fsl-dpaa2/mac/mac.c b/drivers/staging/fsl-dpaa2/mac/mac.c
index 5d1498189b23..71b5091ff8fd 100644
--- a/drivers/staging/fsl-dpaa2/mac/mac.c
+++ b/drivers/staging/fsl-dpaa2/mac/mac.c
@@ -142,7 +142,11 @@ static void dpaa2_mac_link_changed(struct net_device *netdev)
        /* the PHY just notified us of link state change */
        phydev = netdev->phydev;
-       state.up = !!phydev->link;
+       state.up = 1;
+       phydev->link = 1;
+       phydev->duplex = 1;
+       phydev->speed = 10000;
+
        if (phydev->link) {
                state.rate = phydev->speed;
@@ -164,11 +168,12 @@ static void dpaa2_mac_link_changed(struct net_device *netdev)
        /* Call the dpmac_set_link_state() only if there is a change in the
         * link configuration
         */
+#if 0
        if (priv->old_state.up == state.up &&
            priv->old_state.rate == state.rate &&
            priv->old_state.options == state.options)
                return;
-
+#endif
        priv->old_state = state;
        phy_print_status(phydev);
```

With the above changes, the link status is reported as up. It means that it is not necessary to have a valid link with the peer to test the PCS and PMA loop modes. If you decide to test the interface in different loopback modes (SerDes, PCS, MAC), it is recommended to apply the above changes as well.

NOTE

The Linux OS version used in the current document is here.

This document does not provide technical details about these three loopbacks. See the PHY data sheet for a detailed description.

The loopback is configured using the MDIO scripts (mdio_read_c45.sh, mdio_write_c45.sh) provided in Linux use case with 40G MAC. The PHY uses Clause 45 commands, which means that besides the PHY address and register offset, the MMD device type must be provided as well (PCS, PMA, or PHY XS).

4.5.1.1 PHY PCS loopback

When ping command is executed in this mode, echo requests are generated from the Linux OS stack, sent to the PHY (Figure 22), looped back in the PCS, they reach back into the WRIOP port, and then they are caught by the TCPDUMP utility. All the packets are gated at the PCS level if no other option is set. It means that no data is seen at the peer side.

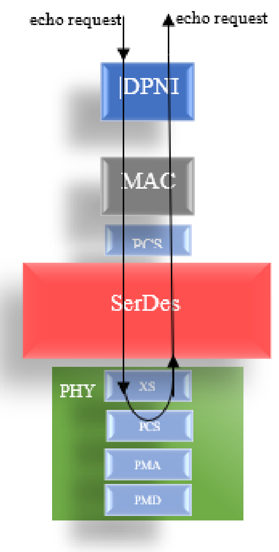


Figure 22. PHY PCS loopback

4.5.1.1.1 PHY PCS loopback verification steps

No debug_link_check option is added to the DPC, which means that no additional changes should be applied to it.

The steps for running this stage are as follows:

```
#Compile the kernel with the changes mentioned earlier

#Start the MC firmware


fsl-mc: Booting Management Complex ... SUCCESS
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x1)
Hit any key to stop autoboot:  0

#apply the dpl file and boot the updated  kernel;
fsl_mc apply dpl 0x20d00000;tftp $fdtaddr kernel_dtb;tftp $loadaddr kernel_img;  booti $loadaddr -
$fdtaddr
```

The DPL applied is the "empty DPL" described in Booting Linux with a minimal DPL file.

When the Linux OS console is available, create the DPMAC and DPNI objects and connect them:

```
restool dpmac create --mac-id=4
restool dprc assign dprc.1 --object=dpmac.4 --plugged=1
```

```
ls-addni -nq=16 -t=8 dpmac.4
Created interface: eth1 (object:dpni.0, endpoint: dpmac.4)
```

Assign an IP to the interface and add a static ARP entry that represents the destination that is pinged in the loopback mode:

```
ifconfig eth1 1.1.1.1 up

arp -s 1.1.1.2 0:0:0:0:0:1 dev eth1


ifconfig eth1
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 1.1.1.1  netmask 255.0.0.0  broadcast 1.255.255.255
        inet6 fe80::cc24:15ff:fec8:b1b3  prefixlen 64  scopeid 0x20<link>
        ether ce:24:15:c8:b1:b3  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 9  bytes 726 (726.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

NOTE

The PCS loopback can run without a valid link with the peer. It means that the RUNNING flag above may not reflect the real link status.

The following steps describe how to set the loopback on the PHY PCS:

```
#Read the PHY PCS control register 0 and set bit 14 (loopback)

#./mdio_read_c45.sh <external mdio bus address> <MMD> <register> <external PHY addr>

# Set PCS control register loopback bit  (bit 14)
#./mdio_write_c45.sh < external mdio bus address > <MMD address> <register address> <data> [external
phy address]
```

To determine the MDIO external bus address and the external PHY address connected to MAC4, the device tree on the running board is consulted:

```
#dump the device tree from sysfs
dtc -I fs -O dts -o dts /proc/device-tree
#the output file is "dts" #Inside it identify MAC4:
[…]
                               dpmac@4 {
                                        phy-connection-type = "usxgmii";
                                        compatible = "fsl,qoriq-mc-dpmac";
                                        reg = <0x4>;
                                        phy-handle = <0x25>;
                                };
[…]
#search for phy-handle 0x25
[…]
#Observe the external MDIO address and the phy adddress mdio@8b96000
{ […] ethernet-phy@5
    { interruEthernetx0 0x3 0x4>;
    compatible = "Ethernet-phy-ieee802.3-c45";
    reg = <0x5>; phandle = <0x25>;
}; […]
```

The above snippet shows the external MDIO address (0x8b96000) and the PHY address (0x5). These two addresses are input to the mdio_read_c45 script:

```
./mdio_read_c45.sh 0x8b96000 3 0 5
DIV=0x0000815c
[0000] 0x08b96030: 0x0000815c (written)
[0000] 0x08b96034: 0x000000a3 (written)
[0000] 0x08b9603c: 0x00000000 (written)
[0000] 0x08b96034: 0x000080a3 (written)
DATA=0x00002040
```

Write the data with loopback bit set into the PCS status register for PHY 5 and verify the write operation:

```
./mdio_write_c45.sh 0x8b96000 3 0 0x6040 5
#Check that the data was written
./mdio_read_c45.sh 0x8b96000 3 0 5
DIV=0x0000815c
[0000] 0x08b96030: 0x0000815c (written)
[0000] 0x08b96034: 0x000000a3 (written)
[0000] 0x08b9603c: 0x00000000 (written)
[0000] 0x08b96034: 0x000080a3 (written)
DATA=0x00006040
```

The stages for running with traffic in the PHY PCS loopback mode are as follows:

```
#Launch in background tcpdump

tcpdump -i eth1 icmp&

#Send two echo requests to destination 1.1.1.2; Try the command several times.

ping -c 2 1.1.1.2 -i 0.1
PING 1.1.1.2 (1.1.1.2) 56(84) bytes of data.
10:35:23.732991 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 1, length 64
10:35:23.733037 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 1, length 64
10:35:23.833876 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 2, length 64
10:35:23.833909 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 2, length 64
^C
--- 1.1.1.2 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 100ms
```

The TCPDUMP captures four echo requests, although the command sent only two.

4.5.1.1.2 PHY PCS loopback result interpretation

The output of the TCPDUMP show four frames, which is the expected result. Two frames are the echo requests sent on the Tx and captured by the TCPDUMP. The other two are the same packets reflected back (looped) in the PHY PCS.

The result excludes any issues with the PHY or with the other layers from the LX2.

In case of a negative result (this means that only two packets are seen in the TCPDUMP), a possible problem would be at the PHY level, if the tests for the other layers on the LX2 have passed successfully. See the PHY data sheet. Remember that the test included here is for the AQR PHY. If a different PHY is used, check if a driver is available for it or if the generic driver is used.

You can optionally check the different counters on DPMAC and DPNI:

```
#check DPMAC counters
restool dpmac info dpmac.4
```

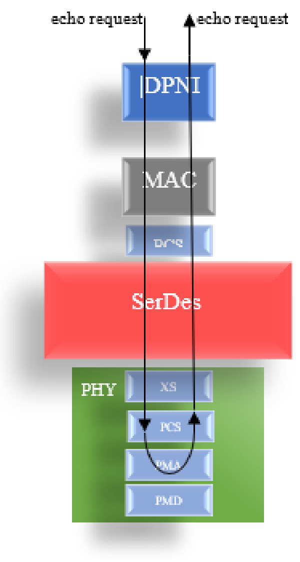
```
#check DPNI counters
restool dpni info dpni.0
```

See Investigating ingress discarded traffic.

4.5.1.2 PHY PMA loopback

When ping command is executed in this mode, echo requests are generated from the Linux OS stack, sent to the PHY (Figure 23), looped back in the PMA, they reach back into the WRIOP port, and then they are caught by the TCPDUMP utility. All the packets are gated at PMA level if no other option is set. It means that no data is observed at the peer side.

Figure 23. PHY PMA loopback



4.5.1.2.1 PHY PMA loopback verification steps

No debug_link_check option is added in the DPC, which means that no additional changes should be applied on it.

The steps for running this stage are as follows:

```
#Compile the kernel with the changes mentioned earlier
#Start the MC firmware
fsl-mc: Booting Management Complex ... SUCCESS
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x1)
Hit any key to stop autoboot:  0
#apply the dpl file and boot the updated  kernel;
fsl_mc apply dpl 0x20d00000;tftp $fdtaddr kernel_dtb;tftp $loadaddr kernel_img;  booti $loadaddr -
$fdtaddr
```

The DPL applied is the "empty DPL" described in Booting Linux with a minimal DPL file.

When the Linux OS console is available, create the DPMAC and DPNI objects and connect them:

```
restool dpmac create --mac-id=4
restool dprc assign dprc.1 --object=dpmac.4 --plugged=1
ls-addni -nq=16 -t=8 dpmac.4
Created interface: eth1 (object:dpni.0, endpoint: dpmac.4)
```

Assign an IP to the interface and add a static ARP entry that represents the destination that is pinged in the loopback mode:

```
ifconfig eth1 1.1.1.1 up

arp -s 1.1.1.2 0:0:0:0:0:1 dev eth1


ifconfig eth1
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 1.1.1.1  netmask 255.0.0.0  broadcast 1.255.255.255
        inet6 fe80::cc24:15ff:fec8:b1b3  prefixlen 64  scopeid 0x20<link>
        ether ce:24:15:c8:b1:b3  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 9  bytes 726 (726.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

NOTE

The PMA loopback can run without having a valid link with the peer. It means that the RUNNING flag from above may not reflect the real link status.

The following steps describe how to set the loopback on the PHY PMA:

```
#Read PMA Control register 0. PMA is at address 1.
./mdio_read_c45.sh 0x8b96000 1 0 5
DIV=0x0000815c
[0000] 0x08b96030: 0x0000815c (written)
[0000] 0x08b96034: 0x000000a1 (written)
[0000] 0x08b9603c: 0x00000000 (written)
[0000] 0x08b96034: 0x000080a1 (written)
DATA=0x00002040
#Set bit 0 in PMA Control register, to 1
./mdio_write_c45.sh 0x8b96000 1 0 0x2041 5
DIV=0x0000815c
[0000] 0x08b96030: 0x0000815c (written)
[0000] 0x08b96034: 0x000000a1 (written)
[0000] 0x08b9603c: 0x00000000 (written)
[0000] 0x08b96038: 0x00002041 (written)
#Verify the write
./mdio_read_c45.sh 0x8b96000 1 0 5
DIV=0x0000815c
[0000] 0x08b96030: 0x0000815c (written)
[0000] 0x08b96034: 0x000000a1 (written)
[0000] 0x08b9603c: 0x00000000 (written)
[0000] 0x08b96034: 0x000080a1 (written)
DATA=0x00002041
```

The stages for running with traffic in the PHY PMA loopback mode are as follows:

```
#Launch in background tcpdump tcpdump -i eth1 icmp&
#Send two echo requests to destination 1.1.1.2. Try the command several times.
ping -c 2 1.1.1.2 -i 0.1
PING 1.1.1.2 (1.1.1.2) 56(84) bytes of data.
10:35:23.732991 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 1, length 64
10:35:23.733037 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 1, length 64
10:35:23.833876 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 2, length 64
10:35:23.833909 IP one.one.one.one > 1.1.1.2: ICMP echo request, id 1419, seq 2, length 64
^C
```

```
--- 1.1.1.2 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 100ms
```

The TCPDUMP captures four echo requests, although the command sent only two.

4.5.1.2.2 PHY PMA loopback result interpretation

The output of the TCPDUMP shows four frames, which is the expected result. Two frames are the echo requests sent on the Tx and captured by the TCPDUMP, while the other two are the same packets reflected back (looped) in the PHY PMA.

The result excludes any issue with the PHY or with the other layers from the LX2.

In case of a negative result, if no packets were received in the Linux OS stack (this means that only two packets are seen in TCPDUMP), a possible problem would be at PHY level (if the tests for the other layers on the LX2 have passed successfully). See the PHY data sheet. Remember that the test included here is for the AQR PHY. If a different PHY is used, check if a driver is available for it or if the generic driver is used.

You can optionally check the different counters on DPMAC and DPNI:

```
#check DPMAC counters
restool dpmac info dpmac.4
#check DPNI counters
restool dpni info dpni.0
```

See Investigating ingress discarded traffic.

4.5.1.3 PHY XS loopback

In this mode, data is not sent from the LX2 side but from the traffic generator. The packets must be looped in the PHY XS layer and received in the traffic generator.

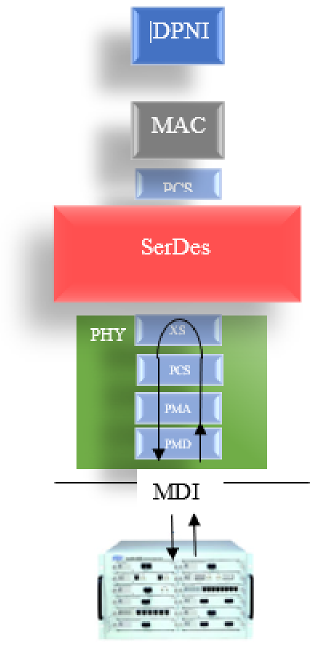


Figure 24. PHY XS loopback

4.5.1.3.1 PHY XS loopback verification steps

No changes should be applied on the kernel or in the DPC file.

The steps for running this stage are as follows:

```
#Start the MC firmware
fsl-mc: Booting Management Complex ... SUCCESS
fsl-mc: Management Complex booted (version: 10.29.0, boot status: 0x1)
Hit any key to stop autoboot:  0
#apply the dpl file and boot kernel;
fsl_mc apply dpl 0x20d00000;tftp $fdtaddr kernel_dtb;tftp $loadaddr kernel_img;  booti $loadaddr -
$fdtaddr
```

The DPL applied is the "empty DPL" described in Booting Linux with a minimal DPL file.

When the Linux OS console is available, create the DPMAC and DPNI objects and connect them:

```
restool dpmac create --mac-id=4
restool dprc assign dprc.1 --object=dpmac.4 --plugged=1
ls-addni -nq=16 -t=8 dpmac.4
Created interface: eth1 (object:dpni.0, endpoint: dpmac.4)
```

Assign an IP to the interface and bring it up:

```
ifconfig eth1 1.1.1.1 up
```

The following steps describe how to set the loopback on the PHY XS:

```
#Read from PHY XS(MMD address 4), Standar Control register - offset 0
./mdio_read_c45.sh 0x8b96000 4 0 5
DIV=0x0000815c
[0000] 0x08b96030: 0x0000815c (written)
[0000] 0x08b96034: 0x000000a4 (written)
[0000] 0x08b9603c: 0x00000000 (written)
[0000] 0x08b96034: 0x000080a4 (written)
DATA=0x00002040
#Set bit 14 in  Standard Control register, to 1
./mdio_write_c45.sh 0x8b96000 4 0 0x6040 5
DIV=0x0000815c
[0000] 0x08b96030: 0x0000815c (written)
[0000] 0x08b96034: 0x000000a4 (written)
[0000] 0x08b9603c: 0x00000000 (written)
[0000] 0x08b96038: 0x00006040 (written)
#Verify the write
./mdio_read_c45.sh 0x8b96000 4 0 5
DIV=0x0000815c
[0000] 0x08b96030: 0x0000815c (written)
[0000] 0x08b96034: 0x000000a4 (written)
[0000] 0x08b9603c: 0x00000000 (written)
[0000] 0x08b96034: 0x000080a4 (written)
DATA=0x00006040
```

From the traffic generator, inject a couple of packets. All of them must be received back. If no packets are received, check the PHY data sheet and the PHY configuration. In this case, it is unlikely to be an issue at the traffic generator level.

4.6 Investigating ingress discarded traffic

On the ingress side, a received frame can be in one of the following cases:

- Dropped at the DPMAC level. This happens when the internal FIFO overflows. The counter can be retrieved using restool (for example, "restool dpmac info dpmac.3") and it is represented by the "rx frame discards" output string.
- Received in one of the Rx queues.
- Filtered at the DPNI level. This counts the frames discarded due to the classifier (or policer) decision to drop the frame. The counter can be retrieved using restool (for example, "restool dpni info dpni.0") and it is represented by the “**ingress_filtered_frames**” output string. This counter is also incremented if a filtered frame went to the error queue (because of a parser error) and got rejected in the classifier.
- Discarded at the DPNI level. This happens if the error queue is not activated for the Ethernet driver. The counter can be retrieved using restool (for example, "restool dpni info dpni.0") and it is represented by the “**ingress_discarded_frames**” output string. There is one exception that is discussed later (see *).
- Received on the DPNI’s error queue, if the frame has a physical error or a parsing error.

  If the frame has a physical error, the “rx frame errors” counter is incremented at the MAC level. This is seen when running "restool dpmac info dpmac.3".

  The following snippet shows how the discard and error behavior is configured inside the driver.

```
        err_cfg.errors = DPAA2_FAS_RX_ERR_MASK;
        err_cfg.set_frame_annotation = 1;
#ifdef CONFIG_FSL_DPAA2_ETH_USE_ERR_QUEUE
        err_cfg.error_action = DPNI_ERROR_ACTION_SEND_TO_ERROR_QUEUE;
#else
        err_cfg.error_action = DPNI_ERROR_ACTION_DISCARD;
#endif
        err = dpni_set_errors_behavior(priv->mc_io, 0, priv->mc_token,
                                       &err_cfg);
```

The Ethernet Linux driver for DPAA2 interfaces does not have the error queue enabled by default (the CONFIG_FSL_DPAA2_ETH_USE_ERR_QUEUE kernel option is not set).

The above bullets are described in Table 2, which represents the destination (or the enqueue or discard decision) of an input frame that is received by a DPMAC.

(*) There is a case that is not in the table. The frames on the ingress that are not enqueued due to a QMan (hardware queue manager) condition, such as WRED, tail drop, out of service FQ, and so on. In this situation, the frames are rejected on the pre-enqueue and the discard counter on the DPNI is incremented.

Table 2. Ingress frames enqueue or discard decision

Input frame	Dropped (DPMAC)	Received RxQ (DPNI)	Received ErrQ (DPNI)	Discarded (DPNI)	Filtered (DPNI)
Normal frame destined for the host	-	✓	-	-	-
Frame destined to the host with L3 error	-	-	✓(if ErrQ activated)	✓ (only if ErrQ inactive)	-
Frame destined to the host with FCS error	-	-	✓(if ErrQ activated)	✓ (only if ErrQ inactive)	-
Normal frame not destined for the host	-	-	-	-	✓

Table continues on the next page...

Table 2. Ingress frames enqueue or discard decision (continued)

Input frame	Dropped (DPMAC)	Received RxQ (DPNI)	Received ErrQ (DPNI)	Discarded (DPNI)	Filtered (DPNI)
Frame not destined for the host with L3 error	-	-	✓	-	✓
Frame not destined for the host with FCS error	-	-	✓	-	-
Jumbo frame/high congestion state	✓	-	-	-	-

To enable the error queue, perform the following steps:

- In the Linux OS kernel configuration, set the following options to Y:

```
CONFIG_FSL_DPAA2_ETH_USE_ERR_QUEUE=y
CONFIG_DYNAMIC_DEBUG=y
```

- Apply the following changes in the driver source (optional) to print detailed information (Frame Annotation Status, Frame Annotation Parse Result, and Frame Attribute Flags) for a frame that is received in the error queue:

```
diff --git a/drivers/net/ethernet/freescale/dpaa2/dpaa2-eth.c b/drivers/net/ethernet/freescale/dpaa2/
dpaa2-eth.c
index 34e34e690a61..b6ed028786e8 100644
--- a/drivers/net/ethernet/freescale/dpaa2/dpaa2-eth.c
+++ b/drivers/net/ethernet/freescale/dpaa2/dpaa2-eth.c
@@ -515,6 +515,191 @@ static void dpaa2_eth_rx(struct dpaa2_eth_priv *priv,
 }
 #ifdef CONFIG_FSL_DPAA2_ETH_USE_ERR_QUEUE
+
+
+/* Interpretation of the 96-bit Frame Attribute Flags
+ * array of the frame's Parse Result.
+ * This function should be called only while
+ * guarded by net_ratelimit().
+ */
+static void dpaa2_print_faf(struct dpaa2_eth_priv *priv,
+                            struct dpaa2_fapr *fapr)
+{
+ struct dpaa2_faf_bit {
+  char *name;
+  int position;
+ } faf_bits[] = {
+  { .position = 0,  .name = "IPv6 Route hdr2 present" },
+  { .position = 1,  .name = "GTP Primed Detected" },
+  { .position = 2,  .name = "VLAN Prio frame detected" },
+  { .position = 3,  .name = "PTP(1588) frame detected" },
+  { .position = 4,  .name = "VXLAN Present" },
+  { .position = 5,  .name = "VXLAN Parsing error" },
+  { .position = 6,  .name = "Ethernet slow protocol" },
+  { .position = 7,  .name = "IKE Present" },
+  { .position = 8,  .name = "Shim Shell Soft Parsing Error" },
+  { .position = 9,  .name = "Parsing Error" },
+  { .position = 10, .name = "Ethernet MAC Present" },
+  { .position = 11, .name = "Ethernet Unicast" },
+  { .position = 12, .name = "Ethernet Multicast" },
+  { .position = 13, .name = "Ethernet Broadcast" },
+  { .position = 14, .name = "BPDU frame" },
+  { .position = 15, .name = "FCoE detected" },
+  { .position = 16, .name = "FIP detected" },
+  { .position = 17, .name = "Ethernet Parsing Error" },
+  { .position = 18, .name = "LLC+SNAP Present" },
+  { .position = 19, .name = "Unknown LLC/OUI" },
+  { .position = 20, .name = "LLC+SNAP Error" },
+  { .position = 21, .name = "VLAN 1 Present" },
+  { .position = 22, .name = "VLAN n Present" },
+  /* Bit position 23 is reserved */
+  { .position = 24, .name = "VLAN Parsing Error" },
+  { .position = 25, .name = "PPPoE+PPP Present" },
+  { .position = 26, .name = "PPPoE+PPP Parsing Error" },
+  { .position = 27, .name = "MPLS 1 Present" },
+  { .position = 28, .name = "MPLS n Present" },
+  { .position = 29, .name = "MPLS Parsing Error" },
+  { .position = 30, .name = "ARP frame Present" },
+  { .position = 31, .name = "ARP Parsing Error" },
+  { .position = 32, .name = "L2 Unknown Protocol" },
+  { .position = 33, .name = "L2 Soft Parsing Error" },
+  { .position = 34, .name = "IPv4 1 Present" },
+  { .position = 35, .name = "IPv4 1 Unicast" },
+  { .position = 36, .name = "IPv4 1 Multicast" },
+  { .position = 37, .name = "IPv4 1 Broadcast" },
+  { .position = 38, .name = "IPv4 n Present" },
+  { .position = 39, .name = "IPv4 n Unicast" },
+  { .position = 40, .name = "IPv4 n Multicast" },
+  { .position = 41, .name = "IPv4 n Broadcast" },
+  { .position = 42, .name = "IPv6 1 Present" },
+  { .position = 43, .name = "IPv6 1 Unicast" },
+  { .position = 44, .name = "IPv6 1 Multicast" },
+  { .position = 45, .name = "IPv6 n Present" },
+  { .position = 46, .name = "IPv6 n Unicast" },
+  { .position = 47, .name = "IPv6 n Multicast" },
+  { .position = 48, .name = "IP 1 option present" },
+  { .position = 49, .name = "IP 1 Unknown Protocol" },
+  { .position = 50, .name = "IP 1 Packet is a fragment" },
+  { .position = 51, .name = "IP 1 Packet is an initial fragment" },
+  { .position = 52, .name = "IP 1 Parsing Error" },
+  { .position = 53, .name = "IP n option present" },
+  { .position = 54, .name = "IP n Unknown Protocol" },
+  { .position = 55, .name = "IP n Packet is a fragment" },
+  { .position = 56, .name = "IP n Packet is an initial fragment" },
+  { .position = 57, .name = "ICMP detected" },
+  { .position = 58, .name = "IGMP detected" },
+  { .position = 59, .name = "ICMPv6 detected" },
+  { .position = 60, .name = "UDP Light detected" },
+  { .position = 61, .name = "IP n Parsing Error" },
+  { .position = 62, .name = "Min. Encap Present" },
+  { .position = 63, .name = "Min. Encap S flag set" },
+  { .position = 64, .name = "Min. Encap Parsing Error" },
+  { .position = 65, .name = "GRE Present" },
+  { .position = 66, .name = "GRE R bit set" },
+  { .position = 67, .name = "GRE Parsing Error" },
+  { .position = 68, .name = "L3 Unknown Protocol" },
+  { .position = 69, .name = "L3 Soft Parsing Error" },
+  { .position = 70, .name = "UDP Present" },
+  { .position = 71, .name = "UDP Parsing Error" },
+  { .position = 72, .name = "TCP Present" },
+  { .position = 73, .name = "TCP options present" },
+  { .position = 74, .name = "TCP Control bits 6-11 set" },
+  { .position = 75, .name = "TCP Control bits 3-5 set" },
+  { .position = 76, .name = "TCP Parsing Error" },
+  { .position = 77, .name = "IPSec Present" },
+  { .position = 78, .name = "IPSec ESP found" },
+  { .position = 79, .name = "IPSec AH found" },
+  { .position = 80, .name = "IPSec Parsing Error" },
+  { .position = 81, .name = "SCTP Present" },
+  { .position = 82, .name = "SCTP Parsing Error" },
+  { .position = 83, .name = "DCCP Present" },
+  { .position = 84, .name = "DCCP Parsing Error" },
+  { .position = 85, .name = "L4 Unknown Protocol" },
+  { .position = 86, .name = "L4 Soft Parsing Error" },
+  { .position = 87, .name = "GTP Present" },
+  { .position = 88, .name = "GTP Parsing Error" },
+  { .position = 89, .name = "ESP Present" },
+  { .position = 90, .name = "ESP Parsing Error" },
+  { .position = 91, .name = "iSCSI detected" },
+  { .position = 92, .name = "Capwap-control detected" },
+  { .position = 93, .name = "Capwap-data detected" },
+  { .position = 94, .name = "L5 Soft Parsing Error" },
+  { .position = 95, .name = "IPv6 Route hdr1 present" },
+ };
+ u64 faf_word;
+ u64 mask;
+ int i;
+
+ netdev_dbg(priv->net_dev, "Frame Attribute Flags:");
+ for (i = 0; i < ARRAY_SIZE(faf_bits); i++) {
+  if (faf_bits[i].position < 32) {
+   /* Low part of FAF.
+    * position ranges from 31 to 0, mask from 0 to 31.
+    */
+   mask = 1ull << (31 - faf_bits[i].position);
+   faf_word = __le32_to_cpu(fapr->faf_lo);
+  } else {
+   /* High part of FAF.
+    * position ranges from 95 to 32, mask from 0 to 63.
+    */
+   mask = 1ull << (63 - (faf_bits[i].position - 32));
+   faf_word = __le64_to_cpu(fapr->faf_hi);
+  }
+  if (faf_word & mask) {
+   netdev_dbg(priv->net_dev, "FAF bit %d : %s",
+              faf_bits[i].position, faf_bits[i].name);
+  }
+ }
+}
+
+/* This function should be called only while
+ * guarded by net_ratelimit().
+ */
+static void dpaa2_print_parse_result(struct dpaa2_eth_priv *priv,
+                                     struct dpaa2_fapr *fapr)
+{
+ struct dpaa2_fapr_field {
+  char *name;
+  u16   value;
+ } fapr_fields[] = {
+  { "NxtHdr", __le16_to_cpu(fapr->nxt_hdr) },
+  { "Shim Offset 1", fapr->shim_offset_1 },
+  { "Shim Offset 2", fapr->shim_offset_2 },
+  { "IP1 PID Offset", fapr->ip1_pid_offset },
+  { "Eth Offset", fapr->eth_offset },
+  { "LLC+SNAP Offset", fapr->llc_snap_offset },
+  { "VLAN TCI Offset 1", fapr->vlan_tci_offset_1 },
+  { "VLAN TCI Offset N", fapr->vlan_tci_offset_n },
+  { "Last EtherType Offset", fapr->last_ethertype_offset },
+  { "PPPoE Offset", fapr->pppoe_offset },
+  { "MPLS Offset 1", fapr->mpls_offset_1 },
+  { "MPLS Offset N", fapr->mpls_offset_n },
+  { "L3 Offset 1", fapr->l3_offset_1 },
+  { "L3 Offset N", fapr->l3_offset_n },
+  { "GRE Offset", fapr->gre_offset },
+  { "L4 Offset", fapr->l4_offset },
+  { "L5 Offset", fapr->l5_offset },
+  { "RoutingHdr Offset 1", fapr->routing_hdr_offset_1 },
+  { "RoutingHdr Offset 2", fapr->routing_hdr_offset_2 },
+  { "NxtHdr Offset", fapr->nxt_hdr_offset },
+  { "IPv6Frag Offset", fapr->ipv6_frag_offset },
+  { "Gross Running Sum", __le16_to_cpu(fapr->gross_running_sum) },
+  { "Running Sum", __le16_to_cpu(fapr->running_sum) },
+  { "Parse Error Code", fapr->parse_error_code },
+  { "NxtHdrFrag Offset", fapr->nxt_hdr_frag_offset },
+  { "IPN PID Offset", fapr->ip_proto_offset_n },
+ };
+ int i;
+
+ netdev_dbg(priv->net_dev, "Parse Result:");
+ for (i = 0; i < ARRAY_SIZE(fapr_fields); i++) {
+  netdev_dbg(priv->net_dev, "%21s : 0x%02x",
+             fapr_fields[i].name, fapr_fields[i].value);
+ }
+ dpaa2_print_faf(priv, fapr);
+}
+
+
 /* Processing of Rx frames received on the error FQ
  * We check and print the error bits and then free the frame
  */
@@ -528,6 +713,8 @@ static void dpaa2_eth_rx_err(struct dpaa2_eth_priv *priv,
  void *vaddr;
  struct rtnl_link_stats64 *percpu_stats;
  struct dpaa2_fas *fas;
+ struct dpaa2_fapr *fapr;
+
  u32 status = 0;
  u32 fd_errors;
  bool has_fas_errors = false;
@@ -544,14 +731,19 @@ static void dpaa2_eth_rx_err(struct dpaa2_eth_priv *priv,
    netdev_dbg(priv->net_dev, "RX frame FD err: %08x\n",
        fd_errors);
  }
-
+
+ netdev_dbg(priv->net_dev, "Rx frame on error queue\n");
  /* check frame errors in the FAS field */
  if (has_fas_errors) {
```

```
+  fapr = dpaa2_get_fapr(vaddr, false);
   fas = dpaa2_get_fas(vaddr, false);
   status = le32_to_cpu(fas->status);
-  if (net_ratelimit())
+  if (net_ratelimit()) {
    netdev_dbg(priv->net_dev, "Rx frame FAS err: 0x%08x\n",
        status & DPAA2_FAS_RX_ERR_MASK);
+   dpaa2_print_parse_result(priv, fapr);
+  }
+
  }
  free_rx_fd(priv, fd, vaddr);
@@ -3798,7 +3990,12 @@ static int bind_dpni(struct dpaa2_eth_priv *priv)
   dev_err(dev, "Failed to configure Rx classification key\n");
  /* Configure handling of error frames */
- err_cfg.errors = DPAA2_FAS_RX_ERR_MASK;
+  err_cfg.errors = (DPNI_ERROR_EOFHE | \
+    DPNI_ERROR_FLE |   \
+    DPNI_ERROR_FPE |  \
+    DPNI_ERROR_PHE |   \
+    DPNI_ERROR_L3CE |   \
+    DPNI_ERROR_L4CE);
  err_cfg.set_frame_annotation = 1;
 #ifdef CONFIG_FSL_DPAA2_ETH_USE_ERR_QUEUE
  err_cfg.error_action = DPNI_ERROR_ACTION_SEND_TO_ERROR_QUEUE;
--
```

- Recompile the kernel.
- Boot the target with the compiled image.
- On the booted target, enable the debug prints in the console.

```
echo 8 > /proc/sys/kernel/printk
echo -n 'file dpaa2-eth.c +p' > /sys/kernel/debug/dynamic_debug/control
```

If a frame with the L3 error is injected on the ingress side, a similar snippet is displayed in the console:

```
fsl_dpaa2_eth dpni.0 eth1: RX frame FD err: 00000040
[  183.995273] fsl_dpaa2_eth dpni.0 eth1: Rx frame on error queue
[  184.001095] fsl_dpaa2_eth dpni.0 eth1: Rx frame FAS err: 0x00000024
[  184.007350] fsl_dpaa2_eth dpni.0 eth1: Parse Result:
[  184.012303] fsl_dpaa2_eth dpni.0 eth1:                NxtHdr : 0xfd
[  184.018558] fsl_dpaa2_eth dpni.0 eth1:         Shim Offset 1 : 0xff
[  184.024812] fsl_dpaa2_eth dpni.0 eth1:         Shim Offset 2 : 0xff
[  184.031066] fsl_dpaa2_eth dpni.0 eth1:        IP1 PID Offset : 0x17
[  184.037320] fsl_dpaa2_eth dpni.0 eth1:            Eth Offset : 0x00
[  184.043575] fsl_dpaa2_eth dpni.0 eth1:       LLC+SNAP Offset : 0xff
[  184.049830] fsl_dpaa2_eth dpni.0 eth1:     VLAN TCI Offset 1 : 0xff
[  184.056084] fsl_dpaa2_eth dpni.0 eth1:     VLAN TCI Offset N : 0xff
[  184.062338] fsl_dpaa2_eth dpni.0 eth1: Last EtherType Offset : 0x0c
[  184.068592] fsl_dpaa2_eth dpni.0 eth1:          PPPoE Offset : 0xff
[  184.074846] fsl_dpaa2_eth dpni.0 eth1:         MPLS Offset 1 : 0xff
[  184.081101] fsl_dpaa2_eth dpni.0 eth1:         MPLS Offset N : 0xff
[  184.087354] fsl_dpaa2_eth dpni.0 eth1:           L3 Offset 1 : 0x0e
[  184.093608] fsl_dpaa2_eth dpni.0 eth1:           L3 Offset N : 0x0e
[  184.099862] fsl_dpaa2_eth dpni.0 eth1:            GRE Offset : 0xff
[  184.106116] fsl_dpaa2_eth dpni.0 eth1:             L4 Offset : 0xff
[  184.112370] fsl_dpaa2_eth dpni.0 eth1:             L5 Offset : 0xff
[  184.118624] fsl_dpaa2_eth dpni.0 eth1:   RoutingHdr Offset 1 : 0xff
[  184.124878] fsl_dpaa2_eth dpni.0 eth1:   RoutingHdr Offset 2 : 0xff
[  184.131132] fsl_dpaa2_eth dpni.0 eth1:         NxtHdr Offset : 0x22
[  184.137386] fsl_dpaa2_eth dpni.0 eth1:       IPv6Frag Offset : 0xff
[  184.143640] fsl_dpaa2_eth dpni.0 eth1:     Gross Running Sum : 0xf7b8
[  184.150072] fsl_dpaa2_eth dpni.0 eth1:           Running Sum : 0xffff
[  184.156500] fsl_dpaa2_eth dpni.0 eth1:      Parse Error Code : 0x41
[  184.162754] fsl_dpaa2_eth dpni.0 eth1:     NxtHdrFrag Offset : 0xff
[  184.169008] fsl_dpaa2_eth dpni.0 eth1:        IPN PID Offset : 0xff
[  184.175264] fsl_dpaa2_eth dpni.0 eth1: Frame Attribute Flags:
[  184.180997] fsl_dpaa2_eth dpni.0 eth1: FAF bit 9 : Parsing Error
[  184.186992] fsl_dpaa2_eth dpni.0 eth1: FAF bit 10 : Ethernet MAC Present
[  184.193680] fsl_dpaa2_eth dpni.0 eth1: FAF bit 11 : Ethernet Unicast
[  184.200021] fsl_dpaa2_eth dpni.0 eth1: FAF bit 34 : IPv4 1 Present
[  184.206189] fsl_dpaa2_eth dpni.0 eth1: FAF bit 35 : IPv4 1 Unicast
[  184.212357] fsl_dpaa2_eth dpni.0 eth1: FAF bit 49 : IP 1 Unknown Protocol
[  184.219132] fsl_dpaa2_eth dpni.0 eth1: FAF bit 52 : IP 1 Parsing Error
[  184.225647] fsl_dpaa2_eth dpni.0 eth1: FAF bit 68 : L3 Unknown Protocol
```

Interpretation:

- FD err: 00000040 – FAERR bit set means "Error in Frame Annotation".
- Rx frame FAS err: 0x00000024 means "PHE (parse error status – error during parsing)" and "L3CE (Layer 3 checksum error)". The bits are set in the "Frame Annotation Status".

If a frame with an FCS error is injected on the ingress side, a similar snippet is displayed in the console:

```
fsl_dpaa2_eth dpni.0 eth1: RX frame FD err: 00000040
[  857.422116] fsl_dpaa2_eth dpni.0 eth1: Rx frame on error queue
[  857.427938] fsl_dpaa2_eth dpni.0 eth1: Rx frame FAS err: 0x00001000
[  857.434193] fsl_dpaa2_eth dpni.0 eth1: Parse Result:
[  857.439146] fsl_dpaa2_eth dpni.0 eth1:                NxtHdr : 0x00
[  857.445401] fsl_dpaa2_eth dpni.0 eth1:         Shim Offset 1 : 0xff
[  857.451656] fsl_dpaa2_eth dpni.0 eth1:         Shim Offset 2 : 0xff
[  857.457910] fsl_dpaa2_eth dpni.0 eth1:        IP1 PID Offset : 0xff
[  857.464164] fsl_dpaa2_eth dpni.0 eth1:            Eth Offset : 0xff
[  857.470419] fsl_dpaa2_eth dpni.0 eth1:       LLC+SNAP Offset : 0xff
[  857.476672] fsl_dpaa2_eth dpni.0 eth1:     VLAN TCI Offset 1 : 0xff
[  857.482926] fsl_dpaa2_eth dpni.0 eth1:     VLAN TCI Offset N : 0xff
[  857.489181] fsl_dpaa2_eth dpni.0 eth1: Last EtherType Offset : 0xff
[  857.495435] fsl_dpaa2_eth dpni.0 eth1:          PPPoE Offset : 0xff
[  857.501689] fsl_dpaa2_eth dpni.0 eth1:         MPLS Offset 1 : 0xff
[  857.507943] fsl_dpaa2_eth dpni.0 eth1:         MPLS Offset N : 0xff
[  857.514196] fsl_dpaa2_eth dpni.0 eth1:           L3 Offset 1 : 0xff
[  857.520450] fsl_dpaa2_eth dpni.0 eth1:           L3 Offset N : 0xff
[  857.526704] fsl_dpaa2_eth dpni.0 eth1:            GRE Offset : 0xff
[  857.532958] fsl_dpaa2_eth dpni.0 eth1:             L4 Offset : 0xff
[  857.539212] fsl_dpaa2_eth dpni.0 eth1:             L5 Offset : 0xff
[  857.545466] fsl_dpaa2_eth dpni.0 eth1:   RoutingHdr Offset 1 : 0xff
[  857.551721] fsl_dpaa2_eth dpni.0 eth1:   RoutingHdr Offset 2 : 0xff
[  857.557975] fsl_dpaa2_eth dpni.0 eth1:         NxtHdr Offset : 0xff
[  857.564229] fsl_dpaa2_eth dpni.0 eth1:       IPv6Frag Offset : 0xff
[  857.570483] fsl_dpaa2_eth dpni.0 eth1:     Gross Running Sum : 0x00
[  857.576737] fsl_dpaa2_eth dpni.0 eth1:           Running Sum : 0x00
[  857.582991] fsl_dpaa2_eth dpni.0 eth1:      Parse Error Code : 0x00
[  857.589245] fsl_dpaa2_eth dpni.0 eth1:     NxtHdrFrag Offset : 0xff
[  857.595499] fsl_dpaa2_eth dpni.0 eth1:        IPN PID Offset : 0xff
[  857.601754] fsl_dpaa2_eth dpni.0 eth1: Frame Attribute Flags:
```

Interpretation:

- FD err: 00000040 – FAERR bit set means "Error in Frame Annotation".
- Rx frame FAS err: 0x00001000 means "**FPE (Frame Physical Error)" bit set in "Frame Annotation Status**".

In case of a physical frame error, the parse result is invalid (it has only FFs). This suggests that the parser cannot collect any info for the received frame. This always happens for a frame that has a physical error.

If the frames are discarded (the discard counter is incremented on the DPNI) and nothing is enqueued to the error queue, it means that the rejection happens on the pre-enqueue due to various reasons (WRED, tail drop, out of service FQ, congestion). This means that these frames are not enqueued by QMAN. The counter that can be checked for this event is called FQDC and it can be computed as follows:

```
#Example for MAC3 the counter can be checked at:
0x8c0000 + 0x4000  * 3 + 0x110
```

In the context of the FQDC counter increasing continuously, it is recommended to have the following change in the running kernel (the explanation is in the patch itself):

```
diff --git a/drivers/net/ethernet/freescale/dpaa2/dpaa2-eth.c b/drivers/net/ethernet/freescale/dpaa2/
dpaa2-eth.c
index 1ee793e2a664..172f55c9dade 100644
--- a/drivers/net/ethernet/freescale/dpaa2/dpaa2-eth.c
+++ b/drivers/net/ethernet/freescale/dpaa2/dpaa2-eth.c
@@ -4358,6 +4358,30 @@ static int dpaa2_eth_probe(struct fsl_mc_device *dpni_dev)
        if (err)
                goto err_bind;
+       /* Setting (creating a CGID and assigning it on a frame queue that does
+        * not have a congestion group associated  or modifying
+        * an existing CGID on a frame queue to a different CGID) a CGID on a
+        * frame queue that is in service(has frames in it),
+        * will cause the byte/frame counter of that CGR to become corrupted.
+        *
+        * Below is the detailed solution and the impact if it is not applied
+        * precisely.
+        *
+        * Enable tail drop at probe time to prevent instantaneous counter
+        * of a congestion group to be 0 while traffic might be in flight on
+        * the ingress queues. The instantaneous counter will be
+        * decremented by the number of frames that are dequeued from the ingress
+        * queue, once the interface is up. Because its value is 0,
+        * this subtraction yields  an invalid overflow  value.(7FFFFFFFFF)
+        * The outcome will be greater than
+        * any configured value for the tail drop threshold, for any queue in the
+        * group;
+        * as a consequence, the group will always discard frames and this will
+        * trigger a situation in which all the frames received on a queue that
+        * belongs to the congestion group, are discarded continuously.
+        */
+       dpaa2_eth_set_rx_taildrop(priv, false, priv->pfc_enabled);
+
```

This patch is relative to the source reference used in this document, but it can be applied easily on other kernel versions (if it is not already there).

If frames are received normally by the WRIOP port but dropped in the Linux OS stack, use some of the available tools (drop watch support, function tracer, kernel probes, and so on).

5 Additional tips

If frames are no longer received/sent (by checking the DPMAC/DPNI counters) in a debugging context, after changing various settings at the PHY level or the SerDes level, and although there is link and the CDR lock bit is set, try the following procedure:

```
#Reset the SerDes Rx and TX lanes
 echo "Reset LX2 lanes - Tx and RX"
  #LNaTRSTCTL E-H
        /root/iomem w32 0x1ea0c20 0x48000070
        /root/iomem w32 0x1ea0c20 0x80000010
        /root/iomem w32 0x1ea0d20 0x48000070
        /root/iomem w32 0x1ea0d20 0x80000010
        /root/iomem w32 0x1ea0e20 0x48000070
        /root/iomem w32 0x1ea0e20 0x80000010
        /root/iomem w32 0x1ea0f20 0x48000070
        /root/iomem w32 0x1ea0f20 0x80000010
  #LNaTRSTCTL A-D
        /root/iomem w32 0x1ea0820 0x48000070
        /root/iomem w32 0x1ea0820 0x80000010
        /root/iomem w32 0x1ea0920 0x48000070
        /root/iomem w32 0x1ea0920 0x80000010
        /root/iomem w32 0x1ea0a20 0x48000070
        /root/iomem w32 0x1ea0a20 0x80000010
        /root/iomem w32 0x1ea0b20 0x48000070
        /root/iomem w32 0x1ea0b20 0x80000010
   #LNaRRSTCTL E-H
        /root/iomem w32 0x1ea0c40 0x480010B0
        /root/iomem w32 0x1ea0c40 0x80000010
        /root/iomem w32 0x1ea0d40 0x480010B0
        /root/iomem w32 0x1ea0d40 0x80000010
        /root/iomem w32 0x1ea0e40 0x480010B0
        /root/iomem w32 0x1ea0e40 0x80000010
        /root/iomem w32 0x1ea0f40 0x480010B0
        /root/iomem w32 0x1ea0f40 0x80000010
    #LNaRRSTCTL A-D
        /root/iomem w32 0x1ea0840 0x480010B0
        /root/iomem w32 0x1ea0840 0x80000010
        /root/iomem w32 0x1ea0940 0x480010B0
        /root/iomem w32 0x1ea0940 0x80000010
        /root/iomem w32 0x1ea0a40 0x480010B0
        /root/iomem w32 0x1ea0a40 0x80000010
        /root/iomem w32 0x1ea0b40 0x480010B0
        /root/iomem w32 0x1ea0b40 0x80000010
        echo "LX2 reset lane done"
```

6 Revision history

Table 3. Revision history

Revision number	Date	Substantive changes
0	30 August 2021	Initial release.

Date of release: 30 August 2021

Document identifier: AN13329