DPDK

DATA PLANE DEVELOPMENT KIT

Rawdev Drivers

Release 19.08.2

Nov 15, 2019

CONTENTS

The following are a list of raw device PMDs, which can be used from an application through rawdev API.

The following are a list of raw device PMDs, which can be used from an application through rawdev API.

CHAPTER

ONE

NXP DPAA2 CMDIF DRIVER

The DPAA2 CMDIF is an implementation of the rawdev API, that provides communication between the GPP and AIOP (Firmware). This is achieved via using the DPCI devices exposed by MC for GPP <–> AIOP interaction.

More information can be found at NXP Official Website.

1.1 Features

The DPAA2 CMDIF implements following features in the rawdev API;

- Getting the object ID of the device (DPCI) using attributes
- I/O to and from the AIOP device using DPCI

1.2 Supported DPAA2 SoCs

- LS2084A/LS2044A
- LS2088A/LS2048A
- LS1088A/LS1048A

1.3 Prerequisites

See ../platform/dpaa2 for setup information

Currently supported by DPDK:

- NXP SDK **19.03+**.
- MC Firmware version **10.14.0** and higher.
- Supported architectures: **arm64 LE**.
- Follow the DPDK Getting Started Guide for Linux to setup the basic DPDK environment.

Note: Some part of fslmc bus code (mc flib - object library) routines are dual licensed (BSD & GPLv2).

1.4 Pre-Installation Configuration

1.4.1 Config File Options

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_PMD_DPAA2_CMDIF_RAWDEV` (default `y`)

  Toggle compilation of the `lrte_pmd_dpaa2_cmdif` driver.

1.5 Enabling logs

For enabling logs, use the following EAL parameter:

```
./your_cmdif_application <EAL args> --log-level=pmd.raw.dpaa2.cmdif,<level>
```

Using `pmd.raw.dpaa2.cmdif` as log matching criteria, all Event PMD logs can be enabled which are lower than logging `level`.

1.5.1 Driver Compilation

To compile the DPAA2 CMDIF PMD for Linux arm64 gcc target, run the following `make` command:

```
cd <DPDK-source-directory>
make config T=arm64-dpaa2-linux-gcc install
```

1.6 Initialization

The DPAA2 CMDIF is exposed as a vdev device which consists of dpci devices. On EAL initialization, dpci devices will be probed and then vdev device can be created from the application code by

- Invoking `rte_vdev_init("dpaa2_dpci")` from the application
- Using `--vdev="dpaa2_dpci"` in the EAL options, which will call rte_vdev_init() internally

Example:

```
./your_cmdif_application <EAL args> --vdev="dpaa2_dpci"
```

1.6.1 Platform Requirement

DPAA2 drivers for DPDK can only work on NXP SoCs as listed in the `Supported DPAA2 SoCs`.

CHAPTER

TWO

NXP DPAA2 QDMA DRIVER

The DPAA2 QDMA is an implementation of the rawdev API, that provide means to initiate a DMA transaction from CPU. The initiated DMA is performed without CPU being involved in the actual DMA transaction. This is achieved via using the DPDMAI device exposed by MC.

More information can be found at NXP Official Website.

2.1 Features

The DPAA2 QDMA implements following features in the rawdev API;

- Supports issuing DMA of data within memory without hogging CPU while performing DMA operation.
- Supports configuring to optionally get status of the DMA translation on per DMA operation basis.

2.2 Supported DPAA2 SoCs

- LX2160A
- LS2084A/LS2044A
- LS2088A/LS2048A
- LS1088A/LS1048A

2.3 Prerequisites

See ../platform/dpaa2 for setup information

Currently supported by DPDK:

- NXP SDK **19.03+**.
- MC Firmware version **10.14.0** and higher.
- Supported architectures: **arm64 LE**.
- Follow the DPDK Getting Started Guide for Linux to setup the basic DPDK environment.

Note: Some part of fslmc bus code (mc flib - object library) routines are dual licensed (BSD & GPLv2).

2.4 Pre-Installation Configuration

2.4.1 Config File Options

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_PMD_DPAA2_QDMA_RAWDEV` (default `y`)

  Toggle compilation of the `lrte_pmd_dpaa2_qdma` driver.

2.5 Enabling logs

For enabling logs, use the following EAL parameter:

```
./your_qdma_application <EAL args> --log-level=pmd.raw.dpaa2.qdma,<level>
```

Using `pmd.raw.dpaa2.qdma` as log matching criteria, all Event PMD logs can be enabled which are lower than logging `level`.

2.5.1 Driver Compilation

To compile the DPAA2 QDMA PMD for Linux arm64 gcc target, run the following `make` command:

```
cd <DPDK-source-directory>
make config T=arm64-dpaa2-linux-gcc install
```

2.6 Initialization

The DPAA2 QDMA is exposed as a vdev device which consists of dpdmai devices. On EAL initialization, dpdmai devices will be probed and populated into the rawdevices. The rawdev ID of the device can be obtained using

- Invoking `rte_rawdev_get_dev_id("dpdmai.x")` from the application where x is the object ID of the DPDMAI object created by MC. Use can use this index for further rawdev function calls.

2.6.1 Platform Requirement

DPAA2 drivers for DPDK can only work on NXP SoCs as listed in the `Supported DPAA2 SoCs`.

CHAPTER

THREE

IFPGA RAWDEV DRIVER

FPGA is used more and more widely in Cloud and NFV, one primary reason is that FPGA not only provides ASIC performance but also it's more flexible than ASIC.

FPGA uses Partial Reconfigure (PR) Parts of Bit Stream to achieve its flexibility. That means one FPGA Device Bit Stream is divided into many Parts of Bit Stream(each Part of Bit Stream is defined as AFU-Accelerated Function Unit), and each AFU is a hardware acceleration unit which can be dynamically reloaded respectively.

By PR (Partial Reconfiguration) AFUs, one FPGA resources can be time-shared by different users. FPGA hot upgrade and fault tolerance can be provided easily.

The SW IFPGA Rawdev Driver (**ifpga_rawdev**) provides a Rawdev driver that utilizes Intel FPGA Software Stack OPAE(Open Programmable Acceleration Engine) for FPGA management.

3.1 Implementation details

Each instance of IFPGA Rawdev Driver is probed by Intel FpgaDev. In coordination with OPAE share code IFPGA Rawdev Driver provides common FPGA management ops for FPGA operation, OPAE provides all following operations: - FPGA PR (Partial Reconfiguration) management - FPGA AFUs Identifying - FPGA Thermal Management - FPGA Power Management - FPGA Performance reporting - FPGA Remote Debug

All configuration parameters are taken by vdev_ifpga_cfg driver. Besides configuration, vdev_ifpga_cfg driver also hot plugs in IFPGA Bus.

All of the AFUs of one FPGA may share same PCI BDF and AFUs scan depend on IFPGA Rawdev Driver so IFPGA Bus takes AFU device scan and AFU drivers probe. All AFU device driver bind to AFU device by its UUID (Universally Unique Identifier).

To avoid unnecessary code duplication and ensure maximum performance, handling of AFU devices is left to different PMDs; all the design as summarized by the following block diagram:

```
+---------------------------------------------------------------+
|                       Application(s)                          |
+----------------------------.----------------------------------+
                             |
                             |
+----------------------------'----------------------------------+
|                    DPDK Framework (APIs)                      |
+----------|------------|--------.---------------------|--------+
          /              \                             |
         /                \                            |
+-------'-------+  +-------'-------+          +--------'--------+
|    Eth PMD    |  |   Crypto PMD  |          |                 |
```

```
+--------.--------+ +--------.--------+          |                   |
         |                   |                   |                   |
         |                   |                   |                   |
+--------'--------+ +--------'--------+          |       IFPGA       |
|  Eth AFU Dev    | |Crypto AFU Dev   |          |   Rawdev Driver   |
+--------.--------+ +--------.--------+          |(OPAE Share Code)  |
         |                   |                   |                   |
         |                   |           Rawdev  |                   |
+--------'-------------------'--------+    Ops   |                   |
|               IFPGA Bus             | -------->|                   |
+------------------.------------------+          +---------.---------+
                   |                                       |
     Hot-plugin -->|                                       |
                   |                                       |
+------------------'--------------------+        +---------'---------+
|        vdev_ifpga_cfg driver          |        |  Intel FpgaDev    |
+---------------------------------------+        +-------------------+
```

3.2 Build options

- `CONFIG_RTE_LIBRTE_IFPGA_BUS` (default `y`)

  Toggle compilation of IFPGA Bus library.

- `CONFIG_RTE_LIBRTE_IFPGA_RAWDEV` (default `y`)

  Toggle compilation of the `ifpga_rawdev` driver.

3.3 Run-time parameters

This driver is invoked automatically in systems added with Intel FPGA, but PR and IFPGA Bus scan is triggered by command line using `--vdev 'ifpga_rawdev_cfg` EAL option.

The following device parameters are supported:

- `ifpga` [string]

  Provide a specific Intel FPGA device PCI BDF. Can be provided multiple times for additional instances.

- `port` [int]

  Each FPGA can provide many channels to PR AFU by software, each channels is identified by this parameter.

- `afu_bts` [string]

  If null, the AFU Bit Stream has been PR in FPGA, if not forces PR and identifies AFU Bit Stream file.

CHAPTER

FOUR

IOAT RAWDEV DRIVER FOR INTEL® QUICKDATA TECHNOLOGY

The `ioat` rawdev driver provides a poll-mode driver (PMD) for Intel® QuickData Technology, part of Intel® I/O Acceleration Technology (Intel I/OAT). This PMD, when used on supported hardware, allows data copies, for example, cloning packet data, to be accelerated by that hardware rather than having to be done by software, freeing up CPU cycles for other tasks.

4.1 Hardware Requirements

On Linux, the presence of an Intel® QuickData Technology hardware can be detected by checking the output of the `lspci` command, where the hardware will be often listed as "Crystal Beach DMA" or "CBDMA". For example, on a system with Intel® Xeon® CPU E5-2699 v4 @2.20GHz, lspci shows:

```
# lspci | grep DMA
00:04.0 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal Be
00:04.1 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal Be
00:04.2 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal Be
00:04.3 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal Be
00:04.4 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal Be
00:04.5 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal Be
00:04.6 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal Be
00:04.7 System peripheral: Intel Corporation Xeon E7 v4/Xeon E5 v4/Xeon E3 v4/Xeon D Crystal Be
```

On a system with Intel® Xeon® Gold 6154 CPU @ 3.00GHz, lspci shows:

```
# lspci | grep DMA
00:04.0 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
00:04.1 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
00:04.2 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
00:04.3 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
00:04.4 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
00:04.5 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
00:04.6 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
00:04.7 System peripheral: Intel Corporation Sky Lake-E CBDMA Registers (rev 04)
```

4.2 Compilation

For builds done with `make`, the driver compilation is enabled by the `CONFIG_RTE_LIBRTE_PMD_IOAT_RAWDEV` build configuration option. This is enabled by default in builds for x86 platforms, and disabled in other configurations.

For builds using `meson` and `ninja`, the driver will be built when the target platform is x86-based.

4.3 Device Setup

The Intel® QuickData Technology HW devices will need to be bound to a user-space IO driver for use. The script `dpdk-devbind.py` script included with DPDK can be used to view the state of the devices and to bind them to a suitable DPDK-supported kernel driver. When querying the status of the devices, they will appear under the category of "Misc (rawdev) devices", i.e. the command `dpdk-devbind.py --status-dev misc` can be used to see the state of those devices alone.

4.3.1 Device Probing and Initialization

Once bound to a suitable kernel device driver, the HW devices will be found as part of the PCI scan done at application initialization time. No vdev parameters need to be passed to create or initialize the device.

Once probed successfully, the device will appear as a `rawdev`, that is a "raw device type" inside DPDK, and can be accessed using APIs from the `rte_rawdev` library.

4.4 Using IOAT Rawdev Devices

To use the devices from an application, the rawdev API can be used, along with definitions taken from the device-specific header file `rte_ioat_rawdev.h`. This header is needed to get the definition of structure parameters used by some of the rawdev APIs for IOAT rawdev devices, as well as providing key functions for using the device for memory copies.

4.4.1 Getting Device Information

Basic information about each rawdev device can be queried using the `rte_rawdev_info_get()` API. For most applications, this API will be needed to verify that the rawdev in question is of the expected type. For example, the following code snippet can be used to identify an IOAT rawdev device for use by an application:

```
for (i = 0; i < count && !found; i++) {
        struct rte_rawdev_info info = { .dev_private = NULL };
        found = (rte_rawdev_info_get(i, &info) == 0 &&
                        strcmp(info.driver_name,
                                        IOAT_PMD_RAWDEV_NAME_STR) == 0);
}
```

When calling the `rte_rawdev_info_get()` API for an IOAT rawdev device, the `dev_private` field in the `rte_rawdev_info` struct should either be NULL, or else be set to point to a structure of type `rte_ioat_rawdev_config`, in which case the size of the configured device input ring will be returned in that structure.

4.4.2 Device Configuration

Configuring an IOAT rawdev device is done using the `rte_rawdev_configure()` API, which takes the same structure parameters as the, previously referenced, `rte_rawdev_info_get()` API. The main difference is that, because the parameter is used as input rather than output, the `dev_private` structure element cannot be NULL, and must point to a valid `rte_ioat_rawdev_config` structure, containing the ring size to be used by the device. The ring size must be a power of two, between 64 and 4096.

The following code shows how the device is configured in `test_ioat_rawdev.c`:

```
#define IOAT_TEST_RINGSIZE 512
        struct rte_ioat_rawdev_config p = { .ring_size = -1 };
        struct rte_rawdev_info info = { .dev_private = &p };

        /* ... */

        p.ring_size = IOAT_TEST_RINGSIZE;
        if (rte_rawdev_configure(dev_id, &info) != 0) {
                printf("Error with rte_rawdev_configure()\n");
                return -1;
        }
```

Once configured, the device can then be made ready for use by calling the `rte_rawdev_start()` API.

4.4.3 Performing Data Copies

To perform data copies using IOAT rawdev devices, the functions `rte_ioat_enqueue_copy()` and `rte_ioat_do_copies()` should be used. Once copies have been completed, the completion will be reported back when the application calls `rte_ioat_completed_copies()`.

The `rte_ioat_enqueue_copy()` function enqueues a single copy to the device ring for copying at a later point. The parameters to that function include the physical addresses of both the source and destination buffers, as well as two "handles" to be returned to the user when the copy is completed. These handles can be arbitrary values, but two are provided so that the library can track handles for both source and destination on behalf of the user, e.g. virtual addresses for the buffers, or mbuf pointers if packet data is being copied.

While the `rte_ioat_enqueue_copy()` function enqueues a copy operation on the device ring, the copy will not actually be performed until after the application calls the `rte_ioat_do_copies()` function. This function informs the device hardware of the elements enqueued on the ring, and the device will begin to process them. It is expected that, for efficiency reasons, a burst of operations will be enqueued to the device via multiple enqueue calls between calls to the `rte_ioat_do_copies()` function.

The following code from `test_ioat_rawdev.c` demonstrates how to enqueue a burst of copies to the device and start the hardware processing of them:

```
struct rte_mbuf *srcs[32], *dsts[32];
unsigned int j;

for (i = 0; i < RTE_DIM(srcs); i++) {
        char *src_data;

        srcs[i] = rte_pktmbuf_alloc(pool);
        dsts[i] = rte_pktmbuf_alloc(pool);
        srcs[i]->data_len = srcs[i]->pkt_len = length;
        dsts[i]->data_len = dsts[i]->pkt_len = length;
        src_data = rte_pktmbuf_mtod(srcs[i], char *);

        for (j = 0; j < length; j++)
                src_data[j] = rand() & 0xFF;

        if (rte_ioat_enqueue_copy(dev_id,
                        srcs[i]->buf_iova + srcs[i]->data_off,
                        dsts[i]->buf_iova + dsts[i]->data_off,
                        length,
```

```
                                        (uintptr_t)srcs[i],
                                        (uintptr_t)dsts[i],
                                        0 /* nofence */) != 1) {
                        printf("Error with rte_ioat_enqueue_copy for buffer %u\n",
                                        i);
                        return -1;
                }
        }
        rte_ioat_do_copies(dev_id);
```

To retrieve information about completed copies, the API `rte_ioat_completed_copies()` should be used. This API will return to the application a set of completion handles passed in when the relevant copies were enqueued.

The following code from `test_ioat_rawdev.c` shows the test code retrieving information about the completed copies and validating the data is correct before freeing the data buffers using the returned handles:

```
if (rte_ioat_completed_copies(dev_id, 64, (void *)completed_src,
                (void *)completed_dst) != RTE_DIM(srcs)) {
        printf("Error with rte_ioat_completed_copies\n");
        return -1;
}
for (i = 0; i < RTE_DIM(srcs); i++) {
        char *src_data, *dst_data;

        if (completed_src[i] != srcs[i]) {
                printf("Error with source pointer %u\n", i);
                return -1;
        }
        if (completed_dst[i] != dsts[i]) {
                printf("Error with dest pointer %u\n", i);
                return -1;
        }

        src_data = rte_pktmbuf_mtod(srcs[i], char *);
        dst_data = rte_pktmbuf_mtod(dsts[i], char *);
        for (j = 0; j < length; j++)
                if (src_data[j] != dst_data[j]) {
                        printf("Error with copy of packet %u, byte %u\n",
                                        i, j);
                        return -1;
                }
        rte_pktmbuf_free(srcs[i]);
        rte_pktmbuf_free(dsts[i]);
}
```

4.4.4 Querying Device Statistics

The statistics from the IOAT rawdev device can be got via the xstats functions in the `rte_rawdev` library, i.e. `rte_rawdev_xstats_names_get()`, `rte_rawdev_xstats_get()` and `rte_rawdev_xstats_by_name_get`. The statistics returned for each device instance are:

- `failed_enqueues`
- `successful_enqueues`
- `copies_started`
- `copies_completed`

CHAPTER

FIVE

NTB RAWDEV DRIVER

The `ntb` rawdev driver provides a non-transparent bridge between two separate hosts so that they can communicate with each other. Thus, many user cases can benefit from this, such as fault tolerance and visual acceleration.

This PMD allows two hosts to handshake for device start and stop, memory allocation for the peer to access and read/write allocated memory from peer. Also, the PMD allows to use doorbell registers to notify the peer and share some information by using scratchpad registers.

5.1 BIOS setting on Intel Skylake

Intel Non-transparent Bridge needs special BIOS setting. Since the PMD only supports Intel Skylake platform, introduce BIOS setting here. The referencce is https://www.intel.com/content/dam/support/us/en/documents/server-products/Intel_Xeon_Processor_Scalable_Family_BIOS_User_Guide.pdf

- Set the needed PCIe port as NTB to NTB mode on both hosts.
- Enable NTB bars and set bar size of bar 23 and bar 45 as 12-29 (2K-512M) on both hosts. Note that bar size on both hosts should be the same.
- Disable split bars for both hosts.
- Set crosslink control override as DSD/USP on one host, USD/DSP on another host.
- Disable PCIe PII SSC (Spread Spectrum Clocking) for both hosts. This is a hardware requirement.

5.2 Build Options

- `CONFIG_RTE_LIBRTE_PMD_NTB_RAWDEV` (default `y`)

  Toggle compilation of the `ntb` driver.

5.3 Device Setup

The Intel NTB devices need to be bound to a DPDK-supported kernel driver to use, i.e. igb_uio, vfio. The `dpdk-devbind.py` script can be used to show devices status and to bind them to a suitable kernel driver. They will appear under the category of “Misc (rawdev) devices”.

5.4 Limitation

- The FIFO hasn't been introduced and will come in 19.11 release.
- This PMD only supports Intel Skylake platform.

CHAPTER

SIX

OCTEON TX2 DMA DRIVER

OCTEON TX2 has an internal DMA unit which can be used by applications to initiate DMA transaction internally, from/to host when OCTEON TX2 operates in PCIe End Point mode. The DMA PF function supports 8 VFs corresponding to 8 DMA queues. Each DMA queue was exposed as a VF function when SRIOV enabled.

6.1 Features

This DMA PMD supports below 3 modes of memory transfers

1. Internal - OCTEON TX2 DRAM to DRAM without core intervention
2. Inbound - Host DRAM to OCTEON TX2 DRAM without host/OCTEON TX2 cores involvement
3. Outbound - OCTEON TX2 DRAM to Host DRAM without host/OCTEON TX2 cores involvement

6.2 Prerequisites and Compilation procedure

See ../platform/octeontx2 for setup information.

6.3 Pre-Installation Configuration

6.3.1 Config File Options

The following options can be modified in the `config` file.

- `CONFIG_RTE_LIBRTE_PMD_OCTEONTX2_DMA_RAWDEV` (default `y`)

  Toggle compilation of the `lrte_pmd_octeontx2_dma` driver.

6.4 Enabling logs

For enabling logs, use the following EAL parameter:

```
./your_dma_application <EAL args> --log-level=pmd.raw.octeontx2.dpi,<level>
```

Using `pmd.raw.octeontx2.dpi` as log matching criteria, all Event PMD logs can be enabled which are lower than logging `level`.

6.5 Initialization

The number of DMA VFs (queues) enabled can be controlled by setting sysfs entry, *sriov_numvfs* for the corresponding PF driver.

```
echo <num_vfs> > /sys/bus/pci/drivers/octeontx2-dpi/0000\:05\:00.0/sriov_numvfs
```

Once the required VFs are enabled, to be accessible from DPDK, VFs need to be bound to vfio-pci driver.

6.6 Device Setup

The OCTEON TX2 DPI DMA HW devices will need to be bound to a user-space IO driver for use. The script `dpdk-devbind.py` script included with DPDK can be used to view the state of the devices and to bind them to a suitable DPDK-supported kernel driver. When querying the status of the devices, they will appear under the category of "Misc (rawdev) devices", i.e. the command `dpdk-devbind.py --status-dev misc` can be used to see the state of those devices alone.

6.7 Device Configuration

Configuring DMA rawdev device is done using the `rte_rawdev_configure()` API, which takes the mempool as parameter. PMD uses this pool to submit DMA commands to HW.

The following code shows how the device is configured

```
struct dpi_rawdev_conf_s conf = {0};
struct rte_rawdev_info rdev_info = {.dev_private = &conf};

conf.chunk_pool = (void *)rte_mempool_create_empty(...);
rte_mempool_set_ops_byname(conf.chunk_pool, rte_mbuf_platform_mempool_ops(), NULL);
rte_mempool_populate_default(conf.chunk_pool);

rte_rawdev_configure(dev_id, (rte_rawdev_obj_t)&rdev_info);
```

6.8 Performing Data Transfer

To perform data transfer using OCTEON TX2 DMA rawdev devices use standard `rte_rawdev_enqueue_buffers()` and `rte_rawdev_dequeue_buffers()` APIs.

6.9 Self test

On EAL initialization, dma devices will be probed and populated into the raw devices. The rawdev ID of the device can be obtained using

- Invoke `rte_rawdev_get_dev_id("DPI:x")` from the application where x is the VF device's bus id specified in "bus:device.func" format. Use this index for further rawdev function calls.
- This PMD supports driver self test, to test DMA internal mode from test application one can directly calls `rte_rawdev_selftest(rte_rawdev_get_dev_id("DPI:x"))`