rte_rawdevice: Implementing Programmable Accelerators using Generic Offload

Hemant Agrawal, Shreyansh Jain - NXP
DPDK Summit - San Jose – 2017
Problem Statement: Why a `rawdevice`?

- Device ‘flavour’ currently available in DPDK are limited by their characteristics

<table>
<thead>
<tr>
<th>Ethernet Device</th>
<th>Crypto Device</th>
<th>Event Device</th>
<th>Wireless Device</th>
<th>XYZ accelerator Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>librte_ether</td>
<td>librte_cryptodev</td>
<td>librte_eventdev</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

What happens for cases like these? How to integrate them with DPDK Framework?

- A generic ‘flavor’ of device is required which can represent non-generic cases
  - Custom or Specific function IP Block – Compression Engine, Pattern Matching Engine etc.
  - Leveraging Device Bus model for their scan->probe->consume cycle
  - Accelerating adoption of such blocks without creating new lib/* for each new type of device
Problem Statement: Why a `rawdevice`?

► Why `rawdevice` is better than device specific APIs
  ▪ Applications prefers uniform device view: start/stop, queue/ring config, enqueue/dequeue
  ▪ Uniform programming model across devices – all accelerators under rawdevice
  ▪ Quick turnaround time – changes to lib/* for a new devices is a longer cycle

► A generic set of APIs for applications – covering a broad category of accelerators/IPs
  ▪ Command/Control APIs: start/stop, configure a device, query configuration
  ▪ Data I/O APIs: enqueue/dequeue single or multiple buffers
  ▪ Query APIs: Statistics, register dumps
  ▪ Firmware Management APIs: load, unload, version information
A `rte_rawdevice` is a raw/generic device without any standard configuration or input/output method assumption.

The configure, info operation will be opaque structures.

The queue/ring operations will not assume any data or buffer format.

Specific PMDs should expose any specific config APIs – not expecting portability.
Definition of a `rawdevice` (2/2)

**rte_rawdevice** – A generic device for non-generic IP Blocks

```c
rte_rawdev {  
rte_rawdev_data *data;  
rte_rawdev_ops *dev_ops;  
rte_device *dev;  
rte_driver *driver;  
attached : 1;  
};
```

```c
rte_rawdev_data {  
socket_id;  
devid;  
nb_queues;  
private; /* opaque info */  
name;  
}
```

```c
rte_rawdev_ops {  
start/stop/reset;  
queue setup/teardown;  
enqueue/dequeue bufs;  
xstats get/reset;  
firmware load/unload/version;  
};
```

Opaque private data can store any device-driver handshake data for the device. Only interpreted by application and driver.

More common operations can be added to this to make it more 'generic'.
Accelerator Offload Use-case on NXP SoC

- NXP Platform has a programmable engine, called ‘AIOP’
- The engine can exposes a NIC interface and a command-control interfaces for GPP-side, detectable on fsl-mc bus.
- The application need to configure the engine in order to use it.
- NXP provides a library exposing the application level APIs and convert them to command messages.
- Some of the example use-cases are ovs offload or wireless offload.
Accelerator Offload Use-case on NXP SoC

1. AIOP device is scanned over ‘fslmc’ bus and probed through a DPAA2 driver
2. DPAA2 driver creates a rawdevice and initializes it. Hereafter, this device is available as a port for the application to use
3. Application opens the rawdevice port. It can then access rawdevice APIs for device configuration/firmware management/state
4. Some other custom APIs are exposed directly from PMD for application to use
Example: Layering bbdev over rawdevice

`bbdev` or Wireless Base Band device – recently proposed by Amr Mokhtar

rte_bbdev_ops {
  configure; start; stop; close;
  info_get, stats_get, stats_reset;
  queue_setup/release/start/stop;
};

rte_bbdev {
  enqueue_enc_ops;
  enqueue_dec_ops;
  dequeue_enc_ops;
  dequeue_dec_ops;
  ...
}

rte_rawdev_ops {
  configure/start/stop/close/reset;
  xstats get/reset;
  queue_setup/release/configure;
};

rte_rawdev {
  rte_rawdev_data *data;
  rte_rawdev_ops *dev_ops;
  rte_device *dev;
  rte_driver *driver;
  attached : 1;
};

An example linkage
‘drivers/raw/bb_pmd’ calls RTE_PMD_REGISTER_PCI(…)

`bbdev` is scanned by standard Bus implementation (assuming PCI)
  - During probe, device is identified by ‘drivers/raw/bb_pmd’ and initialized
  - rte_rawdevice instance is created and populated;
  - Either have custom APIs exposed for extra functions, or overload the rte_rawdevice (private data)

Application can use ‘bbdev’ through rawdevice port number
What next?

- Generalizing across well known devices like FPGA, Compression IP
- Generic adapters for ethernet/crypto/eventdev devices
- How to add more operations without affecting core structures?
  - ~IOCTls?
  - Opaque structures containing device specific operations
Questions?

Hemant Agrawal hemant.agrawal@nxp.com
Shreyansh Jain shreyansh.jain@nxp.com
Properties for raw device

rte_raw_dev_data
- uint8_t dev_id
- unit8_t nb_queues;
- uint8_t dev_started:1;
- void *dev_private
- void *dev_info
- Struct rte_driver *driver
- Char name[RTE_RAW_MAX_NAME]

rte_raw_dev_ops
- dev_info_get
- dev_configure
- dev_start
- dev_stop
- dev_close
- ....
- queue_def_conf
- queue_setup
- queue_release
- Dump
- Xstats_get
- Xstats_reset

rte_raw_fw_ops
- fw_load
- fw_status
- fw_clock_sync
- fw_config
- fw_unload
- fw_stats
What is different from rte_prgdev?

- The last proposal of rte_prgdev, mainly focused on firmware image management.
- rte_raw_dev focus is attempting to provide a uniform device view and accelerator access to the applications.
- rte_raw_dev is not discounting firmware management, but makes it an optional component.
- rte_raw_dev can serve as a staging device for un-common newly added device flavors.
  - Any commonly used rte_raw based device can be converted into it’s own specific flavor.
SoCs – Flexible Programming
Architecture

- Packet Processing

1. **Autonomous:**
Packets are received, processed and sent within the HW Engine. HW engine controller can be programmed with different autonomous applications.

2. **Semi Autonomous:**
Packets are received by HW Engine. HW Engine controller does part of processing. GPP cores do rest of processing and send the result packets out.

3. **Non-Autonomous:**
Entire packet processing happens within GPP cores with no help from HW controller.

- **Other acceleration – any kind of HW offload:**
  - Pattern Matching
  - Data Compression

Other related terms:
- FMAN
- DPAA
- Data Path Cores
- Control Path Cores
- GPP Core
- GPP Core (2)
- HW Engine
- SEC
- PCD
- Pattern
- Data Comp
- Eth
- Controller (1)
- DPAA