DPDK

Ideas for adding generic HW accelerators to DPDK

Hemant Agrawal, NXP

DPDK Summit Userspace - Dublin- 2017

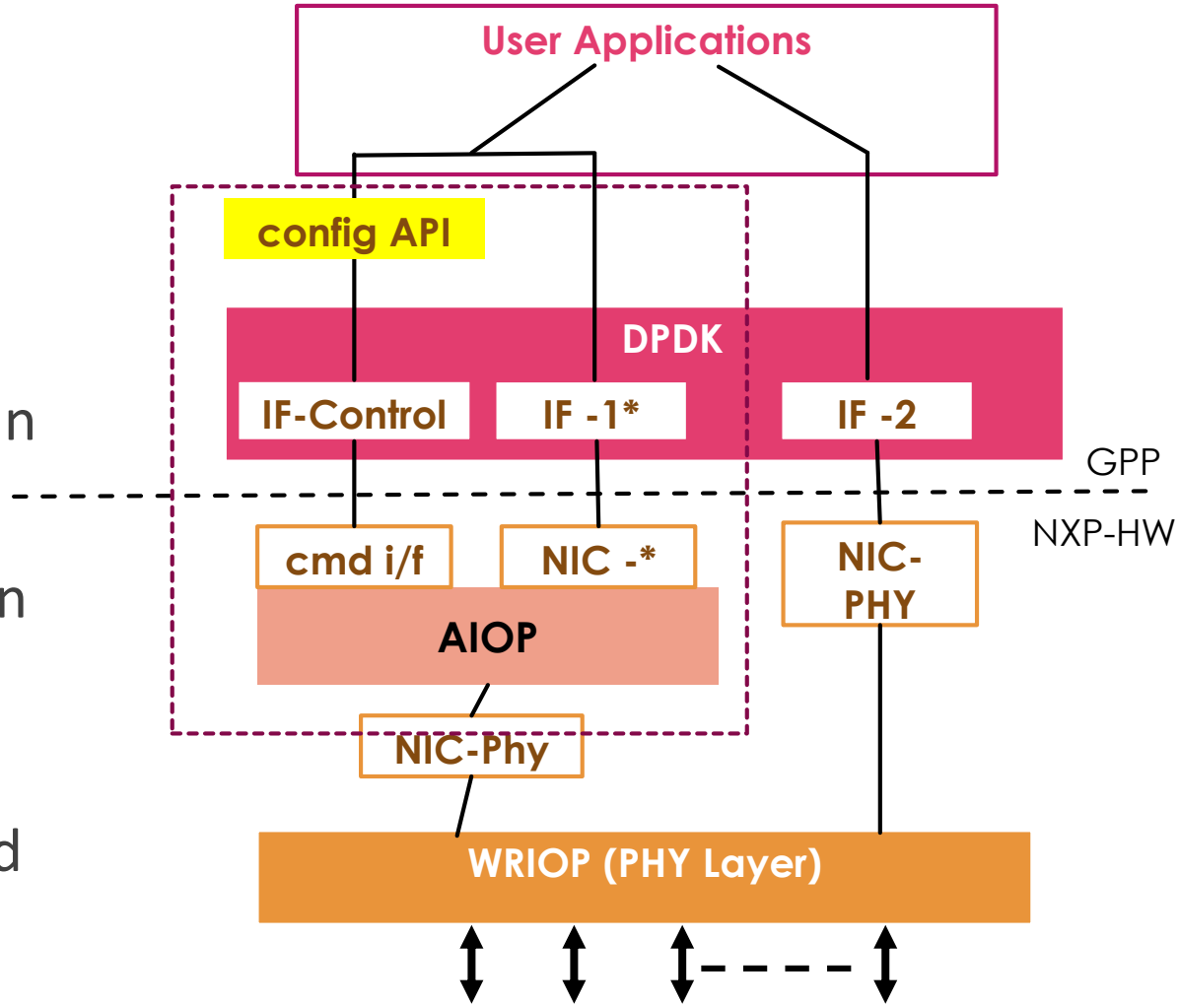
Problem Statement

- SoCs may have many types of different accelerators, which may not be common or use completely different set of capabilities.

- How to expose them via DPDK?
  - Should we create new flavor of device type for each unique accelerator?
  - The applications using these accelerator may not be portable across architectures.

An offload use-case of NXP

- 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.



Why in DPDK?

- Why to add it into DPDK and not use vendor specific SDK APIs.
  - Application prefers uniform device view: Start/Stop, queue/ring config
  - Uniform programming model across devices – ease of application development for users
  - Some of these accelerators may need closer integration e.g. eventdev – single place to get all events.
- Can we find a common ground for such – differently configured – accelerators in DPDK?
  - Management – difficult to find a common/generic ground
  - Input/output – Can be abstracted out.

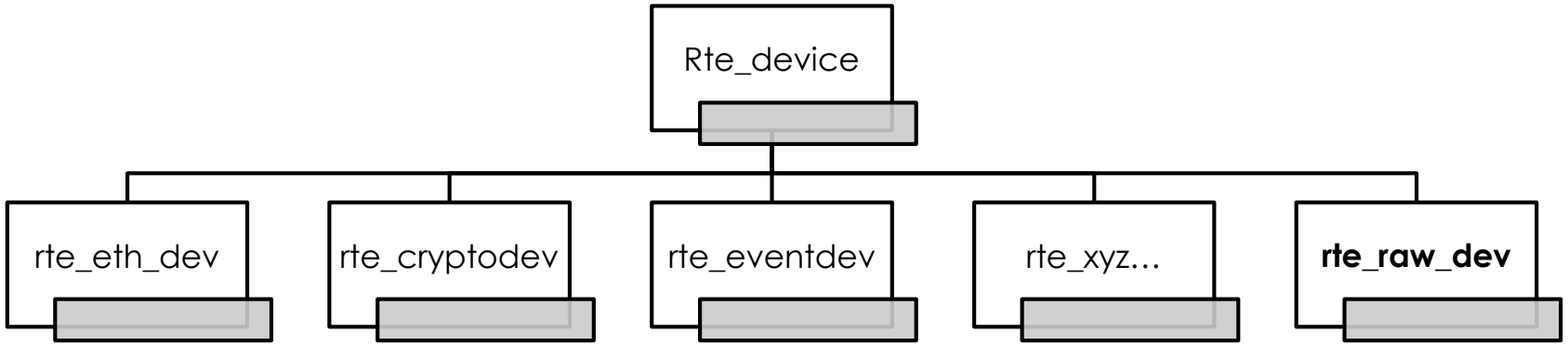
Requirements for Accelerators Interfacing

An abstract, generic APIs for applications to program hardware without knowing the details of programmable devices.

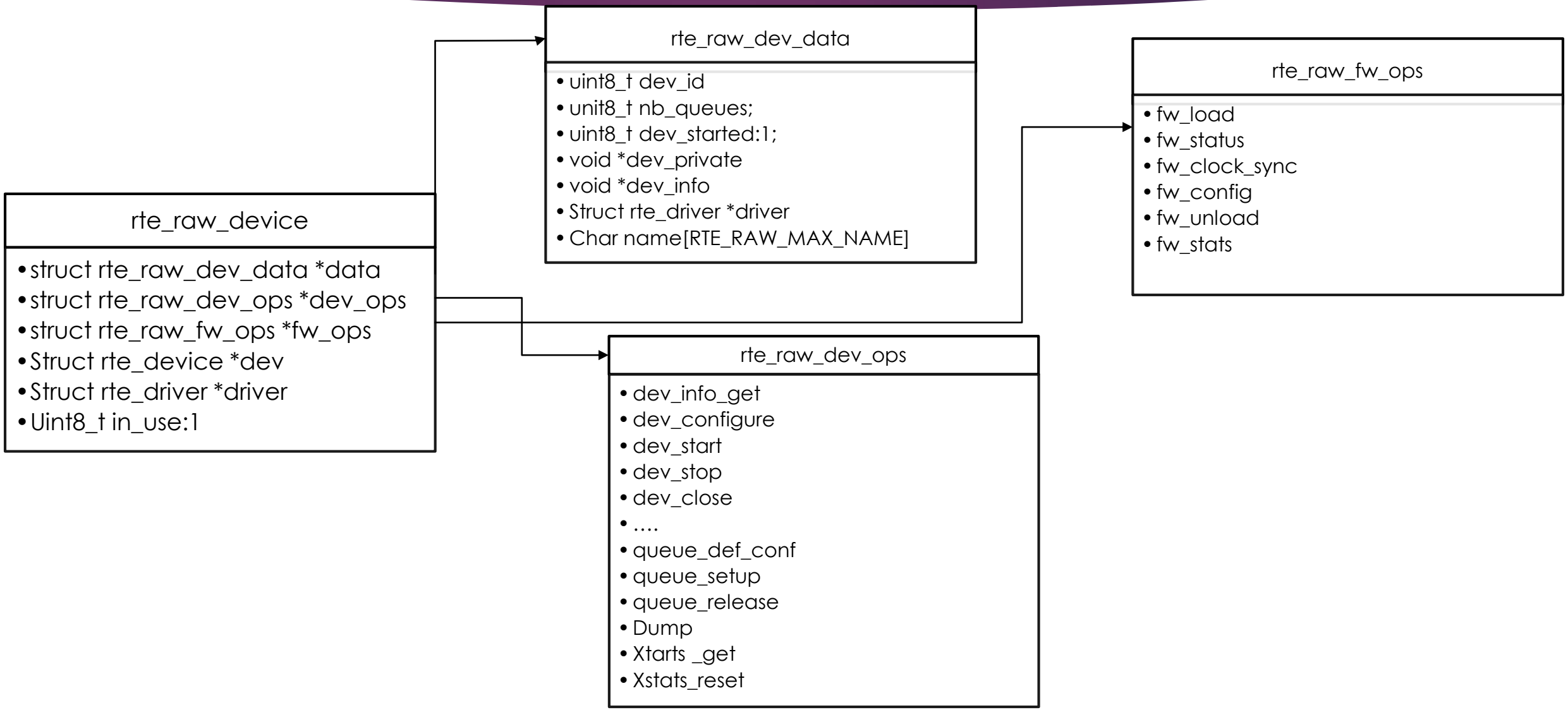
- ✓ ***Command/Control APIs*** – Add, delete, enable, disable, modify, config - *services* etc.
  - Synchronous or Asynchronous request/response model
- ✓ **Data I/O APIs** – enqueue/dequeue.
- ✓ ***Query APIs*** – Query details: Status, statistics etc.
- ✓ ***Notification APIs*** – unsolicited notifications generated by the offload engine. Example : logs, events, exception packets etc.
- ✓ **Firmware Management** - load/unload/status of the firmware image.

Introducing rte_raw_device

- A *rte_raw_device* is a raw/generic device without any standard configuration or input/output method assumption.
- An virtual device – on demand creation by the applications.
- 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.



Properties for raw device



What is different from rte_prgdev ?

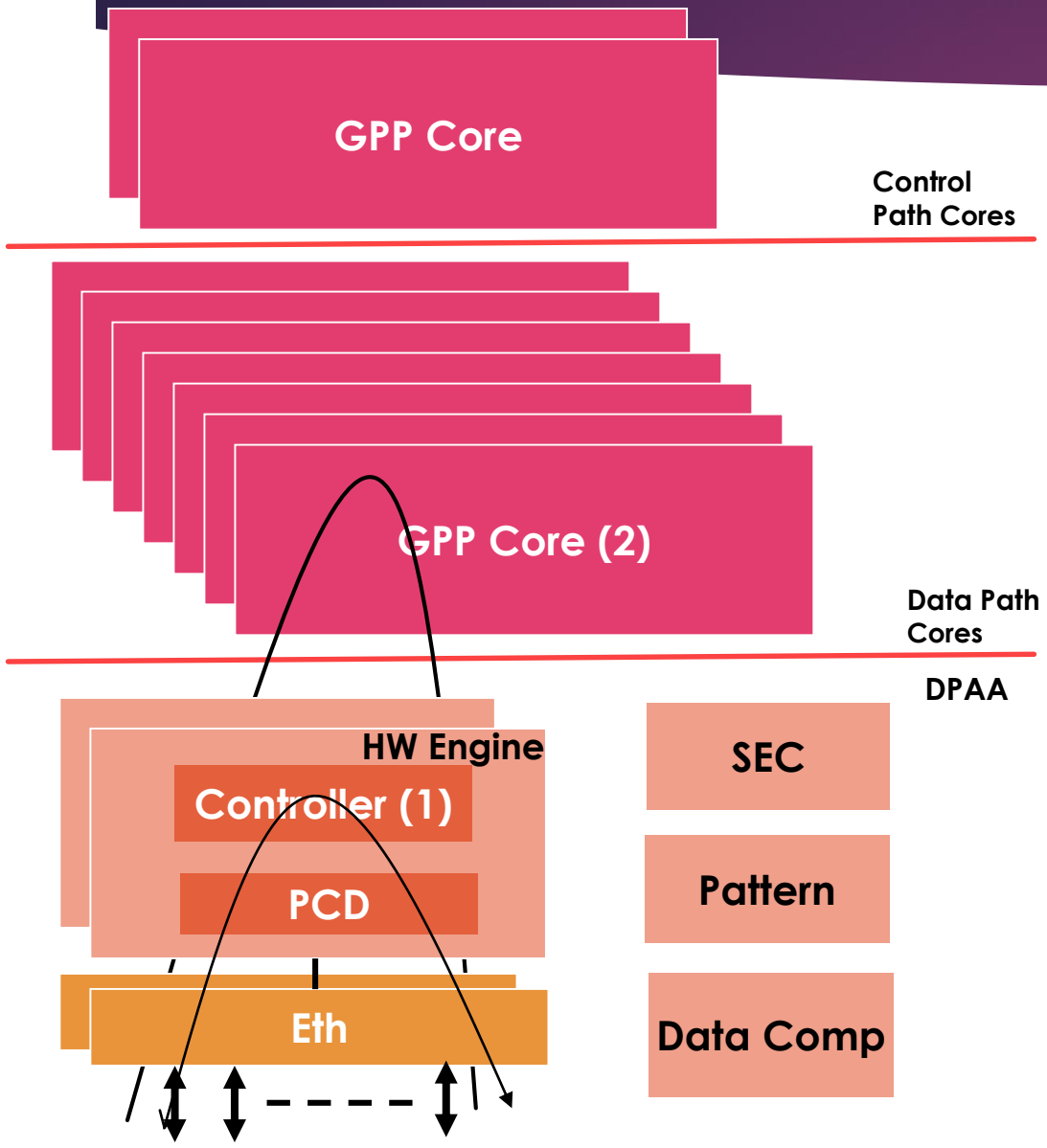
- The last proposal of rte_prgdev, mainly focused on firmware image management.
- rte_raw_dev focus is attempting to provide a uniform device view and accelerator access to the applications.
- rte_raw_dev is not discounting firmware management, but makes it an optional component.
- rte_raw_dev can serve 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.

Questions?

Hemant Agrawal

hemant.agrawal@nxp.com

SoCs – Flexible Programming Architecture



- **Packet Processing**
  - **(1) Autonomous:** Packets are received, processed and sent within the HW Engine. HW engine controller can programmed with different autonomous applications.
  - **(1) & (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.
  - **(2) 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**