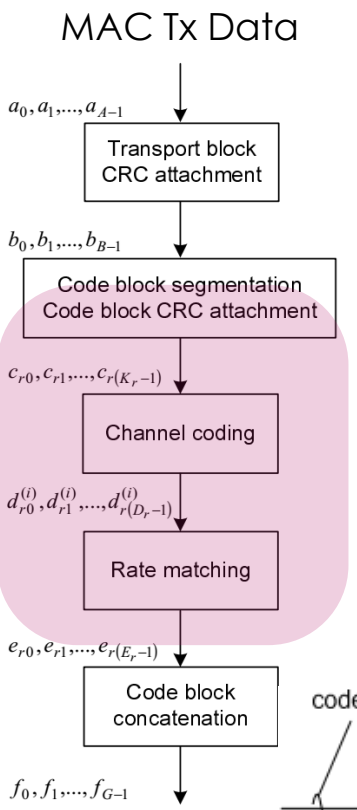
DPDK

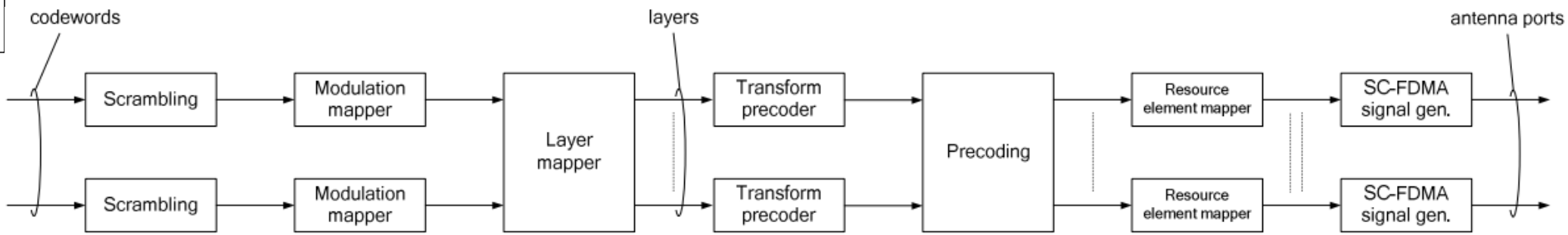
Wireless Base Band Device (bbdev)

Amr Mokhtar

DPDK Summit Userspace - Dublin- 2017

why baseband..?

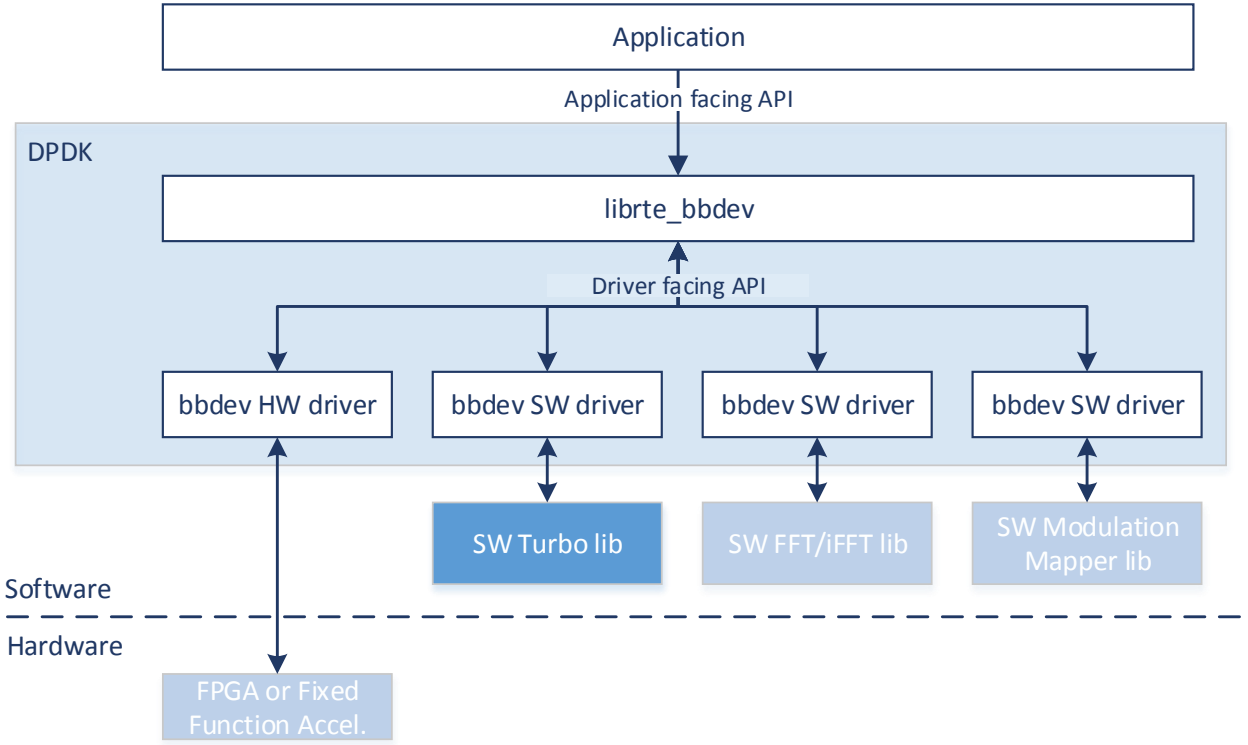




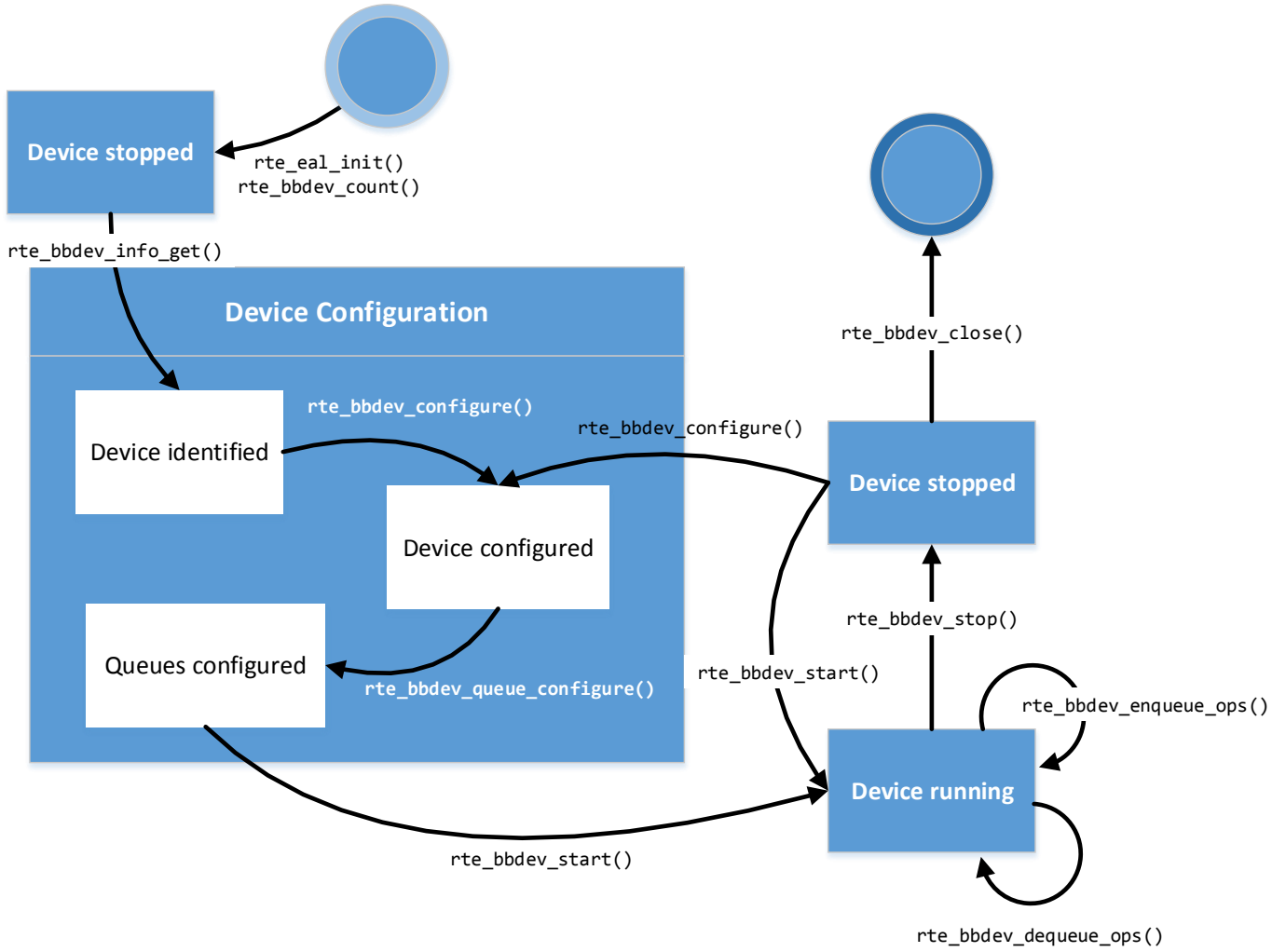
* Reference: 3GPP TS 36.211 & 36.212

architecture

- Common programing framework for wireless workloads
- Seamless HW/SW abstraction interface for underlying operations
- Pluggable driver support for various stages of wireless packet processing (new driver registers itself and reports its capabilities)

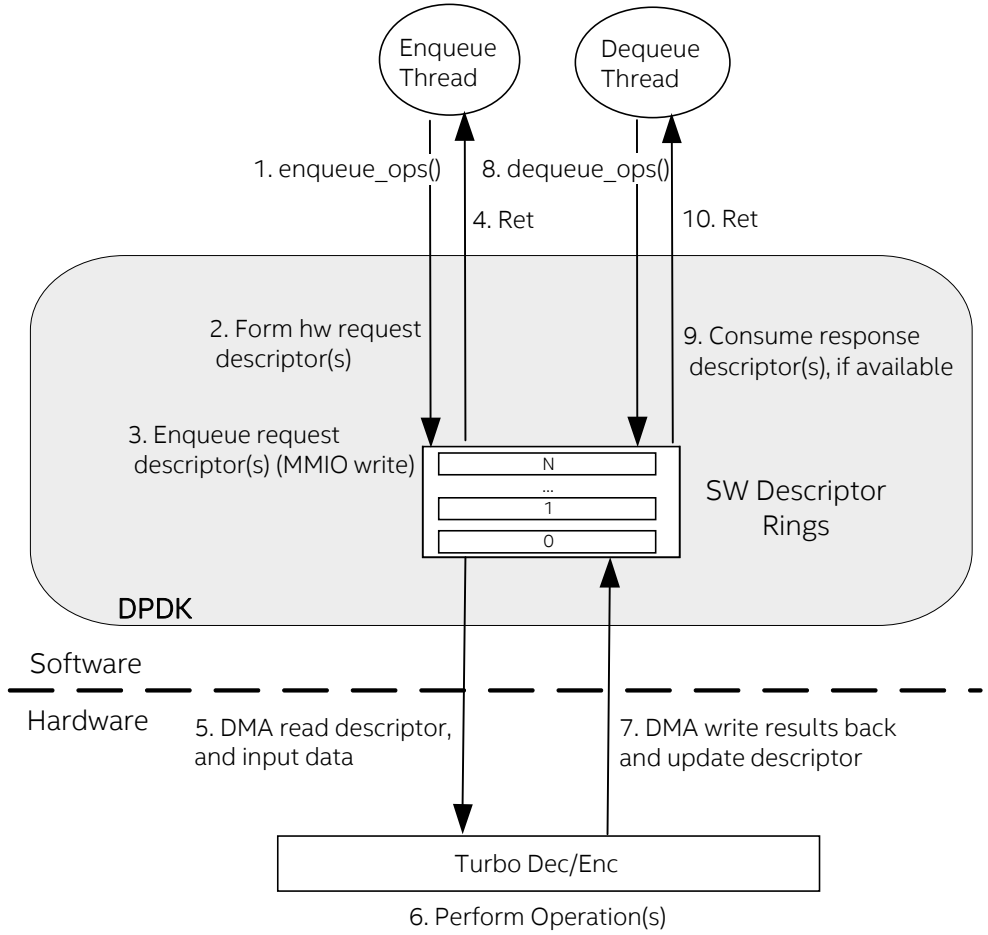


workflow


Device stopped
rte_eal_init()
rte_bbdev_count()
rte_bbdev_info_get()
Device Configuration
Device identified
rte_bbdev_configure()
Device configured
Queues configured
rte_bbdev_queue_configure()
rte_bbdev_start()
rte_bbdev_close()
rte_bbdev_configure()
Device stopped
rte_bbdev_stop()
rte_bbdev_start()
rte_bbdev_enqueue_ops()
Device running
rte_bbdev_dequeue_ops()

lookaside model - hardware

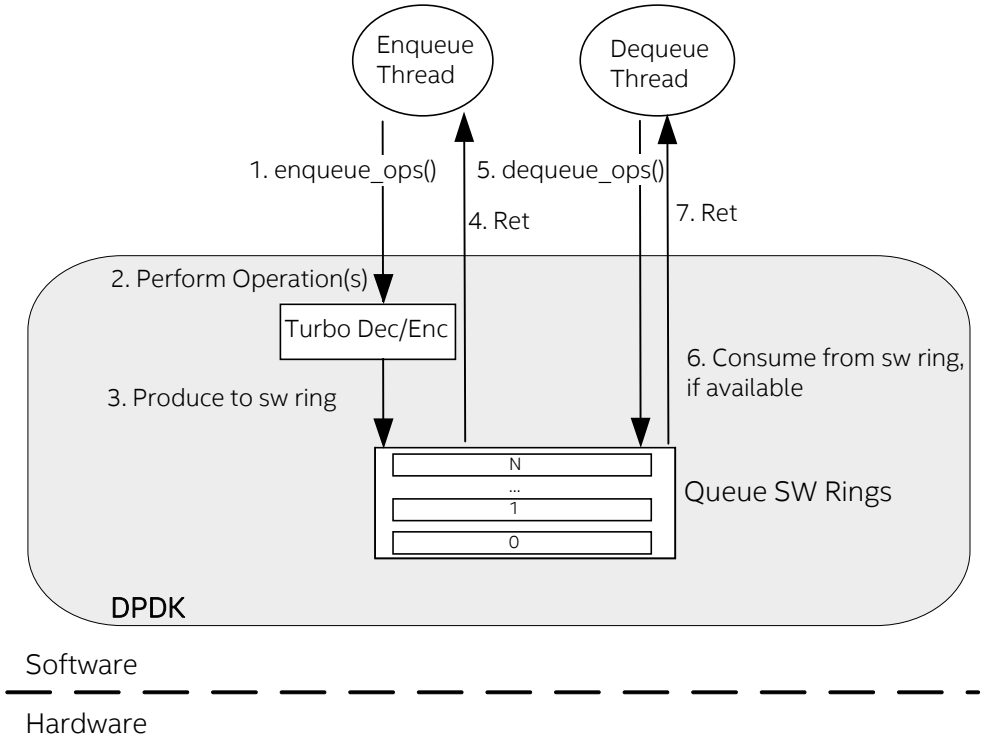
1. Application calls the API to submit an offload request to the user-space device driver
2. Driver forms the descriptor in ring in memory, including pointers to data buffers
3. The driver enqueues the descriptor by writing to the relevant MMIO Register
4. The driver returns from the API call back to the application thread
5. HW DMA reads the descriptor(s) created in step 2, and input data buffers
6. HW performs the operation(s)
7. Once complete the HW will DMA write the output buffers and overwrite the descriptor(s) indicating to SW that this request is complete.
8. Application calls to API to check for completed requests (dequeue)
9. Driver checks if response descriptors have been written back
10. Driver returns results to application if descriptors have been written back, or empty response if not.



** Enqueue thread and dequeue thread may be the same*

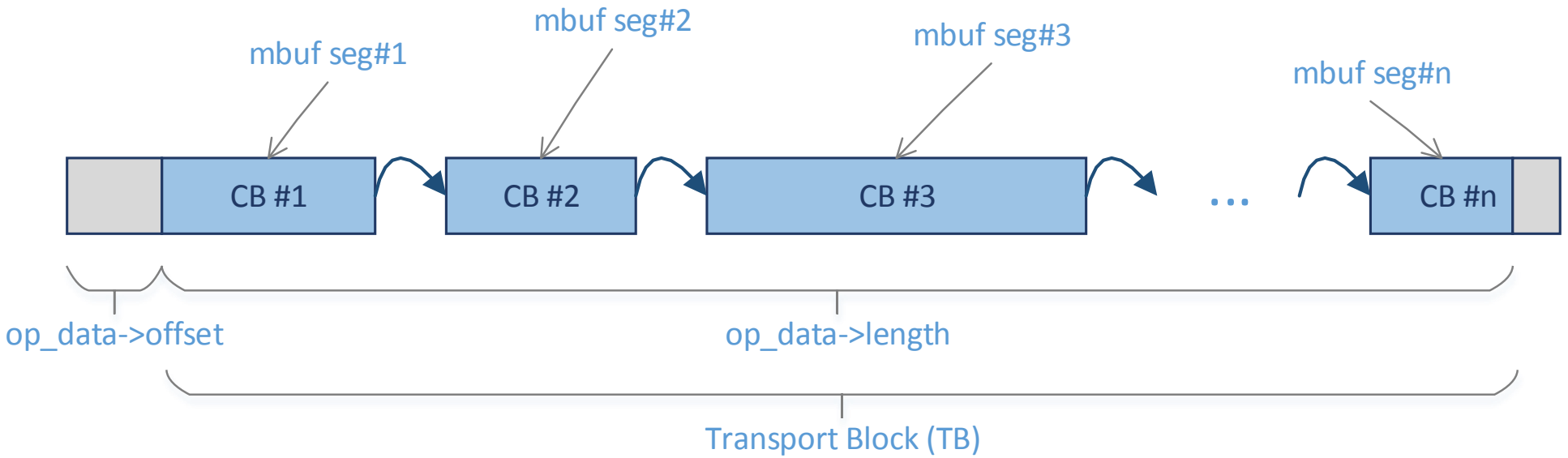
lookaside model - software

1. Application calls the API to submit an offload request to the user-space device driver
2. Driver forms its internal structures and perform operation(s) sequentially.
3. The driver enqueues the outcomes to internal software rings.
4. The driver returns from the API call back to the application thread.
5. Application calls to API to check for completed requests (dequeue).
6. Driver checks if some results were produced on the tip of the ring, then pull it out.
7. Driver returns the pulled out results to application if there were any available, or empty response if not.



** Enqueue thread and dequeue thread may be the same*

Note on mbuf* usage in bbdev




```
/** Data input and output buffer for Turbo operations */
struct rte_bbdev_op_data {
	struct rte_mbuf *data;
	/**< First mbuf segment with input/output data. Each segment represents
	 * one Code Block.
	 */
	uint32_t offset;
	/**< The starting point for the Turbo input/output, in bytes, from the
	 * start of the first segment's data buffer. It must be smaller than the
	 * first segment's data_len!
	 */
	uint32_t length;
	/**< Length of Transport Block - number of bytes for Turbo Encode/Decode
	 * operation for input; length of the output for output operation.
	 */
};
```

** This mbuf formality is experimental and subject to change*

bbdev APIs

- Device Management APIs
- Queue Management APIs
- Operation Management APIs
- Interrupts Support APIs
- Statistics APIs

bbdev APIs >>

- Device creation is based on the same principles as DPDK cryptodev and ethdev.
  - Register driver configuration structure with DPDK EAL using the existing **`RTE_PMD_REGISTER_PCI`** macro.
  - Physical devices are identified by PCI ID during the EAL PCI scan and allocated a unique device identifier.
- Device initiation is also along the same principles as DPDK cryptodev and ethdev.
  - Devices are first configured
    - `int` **`rte_bbdev_configure`**`(uint8_t dev_id, uint16_t num_queues,`
      `const struct rte_bbdev_conf *conf);`
  - Devices queues are then configured before the device is started and used.
    - `int` **`rte_bbdev_queue_configure`**`(uint8_t dev_id, uint16_t queue_id,`
      `const struct rte_bbdev_queue_conf *conf)`

bbdev APIs – Device Management


```
uint8_t rte_bbdev_count(void);

bool rte_bbdev_is_valid(uint8_t dev_id);

uint8_t rte_bbdev_next(uint8_t dev_id);

int rte_bbdev_configure(uint8_t dev_id, uint16_t num_queues,
        const struct rte_bbdev_conf *conf);

int rte_bbdev_info_get(uint8_t dev_id, struct rte_bbdev_info *dev_info);

int rte_bbdev_start(uint8_t dev_id);

int rte_bbdev_stop(uint8_t dev_id);

int rte_bbdev_close(uint8_t dev_id);
```

bbdev APIs – Queue Management


```
int rte_bbdev_queue_configure(uint8_t dev_id, uint16_t queue_id,
        const struct rte_bbdev_queue_conf *conf);

int rte_bbdev_queue_start(uint8_t dev_id, uint16_t queue_id);

int rte_bbdev_queue_stop(uint8_t dev_id, uint16_t queue_id);

int rte_bbdev_queue_info_get(uint8_t dev_id, uint16_t queue_id,
        struct rte_bbdev_queue_info *dev_info);
```

```
/** Different operation types supported by the device */
enum rte_bbdev_op_type {
        RTE_BBDEV_OP_NONE,  /**< Dummy operation that does nothing */
        RTE_BBDEV_OP_TURBO_DEC,  /**< Turbo decode */
        RTE_BBDEV_OP_TURBO_ENC,  /**< Turbo encode */
        RTE_BBDEV_OP_TYPE_COUNT,  /**< Count of different op types */
};
```

DPDK BBDEV APIs – Operation Management


```
static inline uint16_t rte_bbdev_enqueue_ops(uint8_t dev_id, uint16_t queue_id,
        struct rte_bbdev_op **ops, uint16_t num_ops)

static inline uint16_t rte_bbdev_dequeue_ops(uint8_t dev_id, uint16_t queue_id,
        struct rte_bbdev_op **ops, uint16_t num_ops)
```

```
/** Structure specifying a single operation */
struct rte_bbdev_op {
        enum rte_bbdev_op_type type;  /**< Type of this operation */
        int status;  /**< Status of operation that was performed */
        struct rte_mempool *mempool;  /**< Mempool which op instance is in */
        void *opaque_data;  /**< Opaque pointer for user data */

        union {
                struct rte_bbdev_op_turbo_dec *turbo_dec;
                struct rte_bbdev_op_turbo_enc *turbo_enc;
        };
};
```

bbdev APIs – Interrupt Support


```
int rte_bbdev_callback_register(uint8_t dev_id, enum rte_bbdev_event_type event,
        rte_bbdev_cb_fn cb_fn, void *cb_arg);

int rte_bbdev_callback_unregister(uint8_t dev_id, enum rte_bbdev_event_type event,
        rte_bbdev_cb_fn cb_fn, void *cb_arg);

int rte_bbdev_queue_intr_enable(uint8_t dev_id, uint16_t queue_id);

int rte_bbdev_queue_intr_disable(uint8_t dev_id, uint16_t queue_id);

int rte_bbdev_queue_intr_ctl(uint8_t dev_id, uint16_t queue_id, int epfd, int op,
        void *data);
```

bbdev APIs – Statistics


```
int rte_bbdev_stats_get(uint8_t dev_id, struct rte_bbdev_stats *stats);

int rte_bbdev_stats_reset(uint8_t dev_id);

int rte_bbdev_info_get(uint8_t dev_id, struct rte_bbdev_info *dev_info);
```

Questions?

Amr Mokhtar

amr.mokhtar@intel.com