16. Cryptography Device Library

The cryptodev library provides a Crypto device framework for management and provisioning of hardware and software Crypto poll mode drivers, defining generic APIs which support a number of different Crypto operations. The framework currently only supports cipher, authentication, chained cipher/authentication and AEAD symmetric and asymmetric Crypto operations.

16.1. Design Principles

The cryptodev library follows the same basic principles as those used in DPDK’s Ethernet Device framework. The Crypto framework provides a generic Crypto device framework which supports both physical (hardware) and virtual (software) Crypto devices as well as a generic Crypto API which allows Crypto devices to be managed and configured and supports Crypto operations to be provisioned on Crypto poll mode driver.

16.2. Device Management

16.2.1. Device Creation

Physical Crypto devices are discovered during the PCI probe/enumeration of the EAL function which is executed at DPDK initialization, based on their PCI device identifier, each unique PCI BDF (bus/bridge, device, function). Specific physical Crypto devices, like other physical devices in DPDK can be white-listed or black-listed using the EAL command line options.

Virtual devices can be created by two mechanisms, either using the EAL command line options or from within the application using an EAL API directly.

From the command line using the –vdev EAL option

```
--vdev  'crypto_aesni_mb0,max_nb_queue_pairs=2,socket_id=0'
```

Note

- If DPDK application requires multiple software crypto PMD devices then required number of `--vdev` with appropriate libraries are to be added.
- An Application with crypto PMD instances sharing the same library requires unique ID.

Example: `--vdev 'crypto_aesni_mb0' --vdev 'crypto_aesni_mb1'`

Or using the rte_vdev_init API within the application code.

```
rte_vdev_init("crypto_aesni_mb",
              "max_nb_queue_pairs=2,socket_id=0")
```

All virtual Crypto devices support the following initialization parameters:

- `max_nb_queue_pairs` - maximum number of queue pairs supported by the device.
- `socket_id` - socket on which to allocate the device resources on.

16.2.2. Device Identification

Each device, whether virtual or physical is uniquely designated by two identifiers:

- A unique device index used to designate the Crypto device in all functions exported by the cryptodev API.
- A device name used to designate the Crypto device in console messages, for administration or debugging purposes. For ease of use, the port name includes the port index.

16.2.3. Device Configuration

The configuration of each Crypto device includes the following operations:

- Allocation of resources, including hardware resources if a physical device.
- Resetting the device into a well-known default state.
- Initialization of statistics counters.

The rte_cryptodev_configure API is used to configure a Crypto device.

```
int rte_cryptodev_configure(uint8_t dev_id,
                            struct rte_cryptodev_config *config)
```

The `rte_cryptodev_config` structure is used to pass the configuration parameters for socket selection and number of queue pairs.

```
struct rte_cryptodev_config {
    int socket_id;
    /**< Socket to allocate resources on */
    uint16_t nb_queue_pairs;
    /**< Number of queue pairs to configure on device */
};
```

16.2.4. Configuration of Queue Pairs

Each Crypto devices queue pair is individually configured through the `rte_cryptodev_queue_pair_setup` API. Each queue pairs resources may be allocated on a specified socket.

```
int rte_cryptodev_queue_pair_setup(uint8_t dev_id, uint16_t queue_pair_id,
            const struct rte_cryptodev_qp_conf *qp_conf,
            int socket_id)

struct rte_cryptodev_qp_conf {
    uint32_t nb_descriptors; /**< Number of descriptors per queue pair */
    struct rte_mempool *mp_session;
    /**< The mempool for creating session in sessionless mode */
    struct rte_mempool *mp_session_private;
    /**< The mempool for creating sess private data in sessionless mode */
};
```

The fields `mp_session` and `mp_session_private` are used for creating temporary session to process the crypto operations in the session-less mode. They can be the same other different mempools. Please note not all Cryptodev PMDs supports session-less mode.

16.2.5. Logical Cores, Memory and Queues Pair Relationships

The Crypto device Library as the Poll Mode Driver library support NUMA for when a processor's logical cores and interfaces utilize its local memory. Therefore Crypto operations, and in the case of symmetric Crypto operations, the session and the mbuf being operated on, should be allocated from memory pools created in the local memory. The buffers should, if possible, remain on the local processor to obtain the best performance results and buffer descriptors should be populated with mbufs allocated from a mempool allocated from local memory.

The run-to-completion model also performs better, especially in the case of virtual Crypto devices, if the Crypto operation and session and data buffer is in local memory instead of a remote processor's memory. This is also true for the pipe-line model provided all logical cores used are located on the same processor.

Multiple logical cores should never share the same queue pair for enqueuing operations or dequeuing operations on the same Crypto device since this would require global locks and hinder performance. It is however possible to use a different logical core to dequeue an operation on a queue pair from the logical core which it was enqueued on. This means that a crypto burst enqueue/dequeue APIs are a logical place to transition from one logical core to another in a packet processing pipeline.

16.3. Device Features and Capabilities

Crypto devices define their functionality through two mechanisms, global device features and algorithm capabilities. Global devices features identify device wide level features which are applicable to the whole device such as the device having hardware acceleration or supporting symmetric and/or asymmetric Crypto operations.

The capabilities mechanism defines the individual algorithms/functions which the device supports, such as a specific symmetric Crypto cipher, authentication operation or Authenticated Encryption with Associated Data (AEAD) operation.

16.3.1. Device Features

Currently the following Crypto device features are defined:

- Symmetric Crypto operations
- Asymmetric Crypto operations
- Chaining of symmetric Crypto operations
- SSE accelerated SIMD vector operations
- AVX accelerated SIMD vector operations
- AVX2 accelerated SIMD vector operations
- AESNI accelerated instructions
- Hardware off-load processing

16.3.2. Device Operation Capabilities

Crypto capabilities which identify particular algorithm which the Crypto PMD supports are defined by the operation type, the operation transform, the transform identifier and then the particulars of the transform. For the full scope of the Crypto capability see the definition of the structure in the *DPDK API Reference*.

```
struct rte_cryptodev_capabilities;
```

Each Crypto poll mode driver defines its own private array of capabilities for the operations it supports. Below is an example of the capabilities for a PMD which supports the authentication algorithm SHA1_HMAC and the cipher algorithm AES_CBC.

```
static const struct rte_cryptodev_capabilities pmd_capabilities[] = {
    {   /* SHA1 HMAC */
        .op = RTE_CRYPTO_OP_TYPE_SYMMETRIC,
        .sym = {
            .xform_type = RTE_CRYPTO_SYM_XFORM_AUTH,
            .auth = {
                .algo = RTE_CRYPTO_AUTH_SHA1_HMAC,
                .block_size = 64,
                .key_size = {
                    .min = 64,
                    .max = 64,
                    .increment = 0
                },
                .digest_size = {
                    .min = 12,
                    .max = 12,
                    .increment = 0
                },
                .aad_size = { 0 },
                .iv_size = { 0 }
            }
        }
    },
    {   /* AES CBC */
        .op = RTE_CRYPTO_OP_TYPE_SYMMETRIC,
        .sym = {
            .xform_type = RTE_CRYPTO_SYM_XFORM_CIPHER,
            .cipher = {
                .algo = RTE_CRYPTO_CIPHER_AES_CBC,
                .block_size = 16,
                .key_size = {
                    .min = 16,
                    .max = 32,
                    .increment = 8
                },
                .iv_size = {
                    .min = 16,
                    .max = 16,
                    .increment = 0
                }
            }
        }
    }
}
```

16.3.3. Capabilities Discovery

Discovering the features and capabilities of a Crypto device poll mode driver is achieved through the `rte_cryptodev_info_get` function.

```
void rte_cryptodev_info_get(uint8_t dev_id,
                            struct rte_cryptodev_info *dev_info);
```

This allows the user to query a specific Crypto PMD and get all the device features and capabilities. The `rte_cryptodev_info` structure contains all the relevant information for the device.

```
struct rte_cryptodev_info {
    const char *driver_name;
    uint8_t driver_id;
    struct rte_device *device;

    uint64_t feature_flags;

    const struct rte_cryptodev_capabilities *capabilities;

    unsigned max_nb_queue_pairs;

    struct {
        unsigned max_nb_sessions;
    } sym;
};
```

16.4. Operation Processing

Scheduling of Crypto operations on DPDK's application data path is performed using a burst oriented asynchronous API set. A queue pair on a Crypto device accepts a burst of Crypto operations using enqueue burst API. On physical Crypto devices the enqueue burst API will place the operations to be processed on the devices hardware input queue, for virtual devices the processing of the Crypto operations is usually completed during the enqueue call to the Crypto device. The dequeue burst API will retrieve any processed operations available from the queue pair on the Crypto device, from physical devices this is usually directly from the devices processed queue, and for virtual device's from a `rte_ring` where processed operations are placed after being processed on the enqueue call.

16.4.1. Private data

For session-based operations, the set and get API provides a mechanism for an application to store and retrieve the private user data information stored along with the crypto session.

For example, suppose an application is submitting a crypto operation with a session associated and wants to indicate private user data information which is required to be used after completion of the crypto operation. In this case, the application can use the set API to set the user data and retrieve it using get API.

```
int rte_cryptodev_sym_session_set_user_data(
        struct rte_cryptodev_sym_session *sess, void *data, uint16_t size);

void * rte_cryptodev_sym_session_get_user_data(
        struct rte_cryptodev_sym_session *sess);
```

Please note the `size` passed to set API cannot be bigger than the predefined `user_data_sz` when creating the session header mempool, otherwise the function will return error. Also when `user_data_sz` was defined as `0` when creating the session header mempool, the get API will always return `NULL`.

For session-less mode, the private user data information can be placed along with the `struct rte_crypto_op`. The `rte_crypto_op::private_data_offset` indicates the start of private data information. The offset is counted from the start of the rte_crypto_op including other crypto information such as the IVs (since there can be an IV also for authentication).

16.4.2. Enqueue / Dequeue Burst APIs

The burst enqueue API uses a Crypto device identifier and a queue pair identifier to specify the Crypto device queue pair to schedule the processing on. The `nb_ops` parameter is the number of operations to process which are supplied in the `ops` array of `rte_crypto_op` structures. The enqueue function returns the number of operations it actually enqueued for processing, a return value equal to `nb_ops` means that all packets have been enqueued.

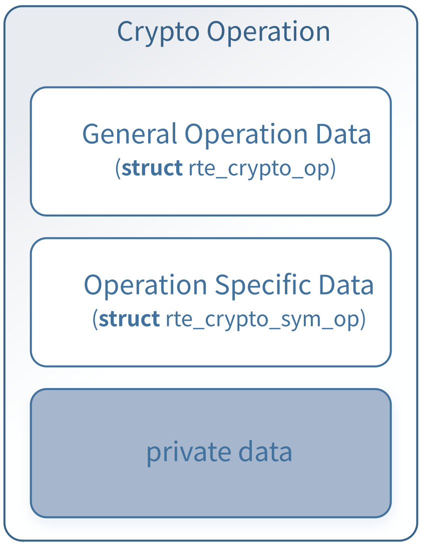
```
uint16_t rte_cryptodev_enqueue_burst(uint8_t dev_id, uint16_t qp_id,
                        struct rte_crypto_op **ops, uint16_t nb_ops)
```

The dequeue API uses the same format as the enqueue API of processed but the `nb_ops` and `ops` parameters are now used to specify the max processed operations the user wishes to retrieve and the location in which to store them. The API call returns the actual number of processed operations returned, this can never be larger than `nb_ops`.

```
uint16_t rte_cryptodev_dequeue_burst(uint8_t dev_id, uint16_t qp_id,
                                     struct rte_crypto_op **ops, uint16_t nb_ops)
```

16.4.3. Operation Representation

An Crypto operation is represented by an rte_crypto_op structure, which is a generic metadata container for all necessary information required for the Crypto operation to be processed on a particular Crypto device poll mode driver.



The operation structure includes the operation type, the operation status and the session type (session-based/less), a reference to the operation specific data, which can vary in size and content depending on the operation being provisioned. It also contains the source mempool for the operation, if it allocated from a mempool.

If Crypto operations are allocated from a Crypto operation mempool, see next section, there is also the ability to allocate private memory with the operation for applications purposes.

Application software is responsible for specifying all the operation specific fields in the `rte_crypto_op` structure which are then used by the Crypto PMD to process the requested operation.

16.4.4. Operation Management and Allocation

The cryptodev library provides an API set for managing Crypto operations which utilize the Mempool Library to allocate operation buffers. Therefore, it ensures that the crypto operation is interleaved optimally across the channels and ranks for optimal processing. A `rte_crypto_op` contains a field indicating the pool that it originated from. When calling `rte_crypto_op_free(op)`, the operation returns to its original pool.

```
extern struct rte_mempool *
rte_crypto_op_pool_create(const char *name, enum rte_crypto_op_type type,
                          unsigned nb_elts, unsigned cache_size, uint16_t priv_size,
                          int socket_id);
```

During pool creation `rte_crypto_op_init()` is called as a constructor to initialize each Crypto operation which subsequently calls `__rte_crypto_op_reset()` to configure any operation type specific fields based on the type parameter.

`rte_crypto_op_alloc()` and `rte_crypto_op_bulk_alloc()` are used to allocate Crypto operations of a specific type from a given Crypto operation mempool. `__rte_crypto_op_reset()` is called on each operation before being returned to allocate to a user so the operation is always in a good known state before use by the application.

```
struct rte_crypto_op *rte_crypto_op_alloc(struct rte_mempool *mempool,
                                          enum rte_crypto_op_type type)

unsigned rte_crypto_op_bulk_alloc(struct rte_mempool *mempool,
                                  enum rte_crypto_op_type type,
                                  struct rte_crypto_op **ops, uint16_t nb_ops)
```

`rte_crypto_op_free()` is called by the application to return an operation to its allocating pool.

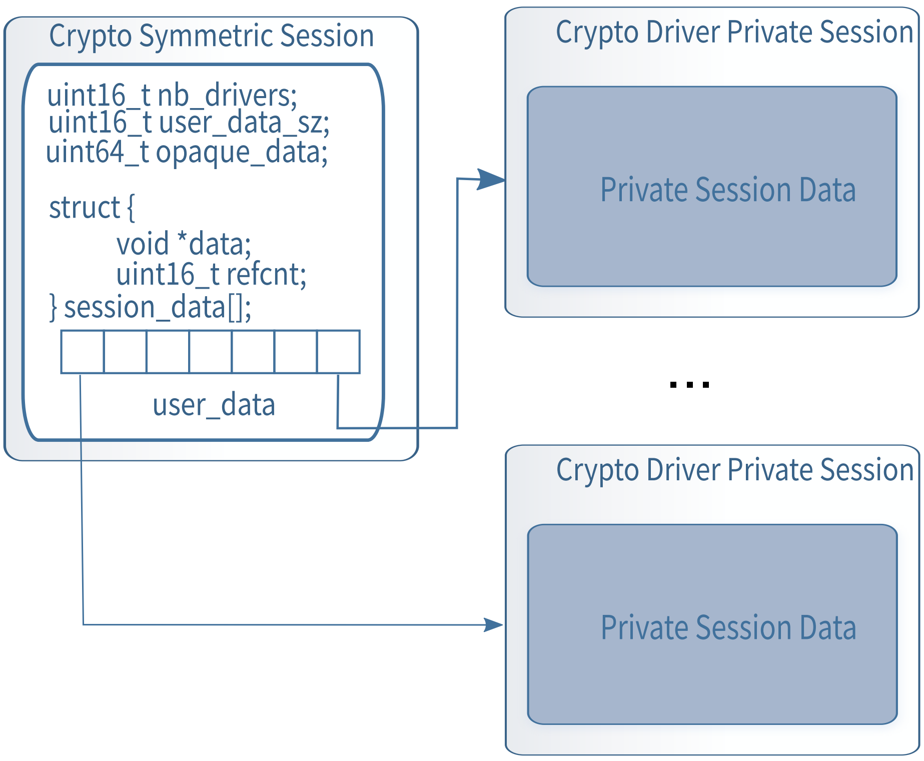
```
void rte_crypto_op_free(struct rte_crypto_op *op)
```

16.5. Symmetric Cryptography Support

The cryptodev library currently provides support for the following symmetric Crypto operations; cipher, authentication, including chaining of these operations, as well as also supporting AEAD operations.

16.5.1. Session and Session Management

Sessions are used in symmetric cryptographic processing to store the immutable data defined in a cryptographic transform which is used in the operation processing of a packet flow. Sessions are used to manage information such as expand cipher keys and HMAC IPADs and OPADs, which need to be calculated for a particular Crypto operation, but are immutable on a packet to packet basis for a flow. Crypto sessions cache this immutable data in a optimal way for the underlying PMD and this allows further acceleration of the offload of Crypto workloads.



The Crypto device framework provides APIs to create session mempool and allocate and initialize sessions for crypto devices, where sessions are mempool objects. The application has to use `rte_cryptodev_sym_session_pool_create()` to create the session header mempool that creates a mempool with proper element size automatically and stores necessary information for safely accessing the session in the mempool's private data field.

To create a mempool for storing session private data, the application has two options. The first is to create another mempool with elt size equal to or bigger than the maximum session private data size of all crypto devices that will share the same session header. The creation of the mempool shall use the traditional `rte_mempool_create()` with the correct `elt_size`. The other option is to change the `elt_size` parameter in `rte_cryptodev_sym_session_pool_create()` to the correct value. The first option is more complex to implement but may result in better memory usage as a session header normally takes smaller memory footprint as the session private data.

Once the session mempools have been created, `rte_cryptodev_sym_session_create()` is used to allocate an uninitialized session from the given mempool. The session then must be initialized using `rte_cryptodev_sym_session_init()` for each of the required crypto devices. A symmetric transform chain is used to specify the operation and its parameters. See the section below for details on transforms.

When a session is no longer used, user must call `rte_cryptodev_sym_session_clear()` for each of the crypto devices that are using the session, to free all driver private session data. Once this is done, session should be freed using `rte_cryptodev_sym_session_free` which returns them to their mempool.

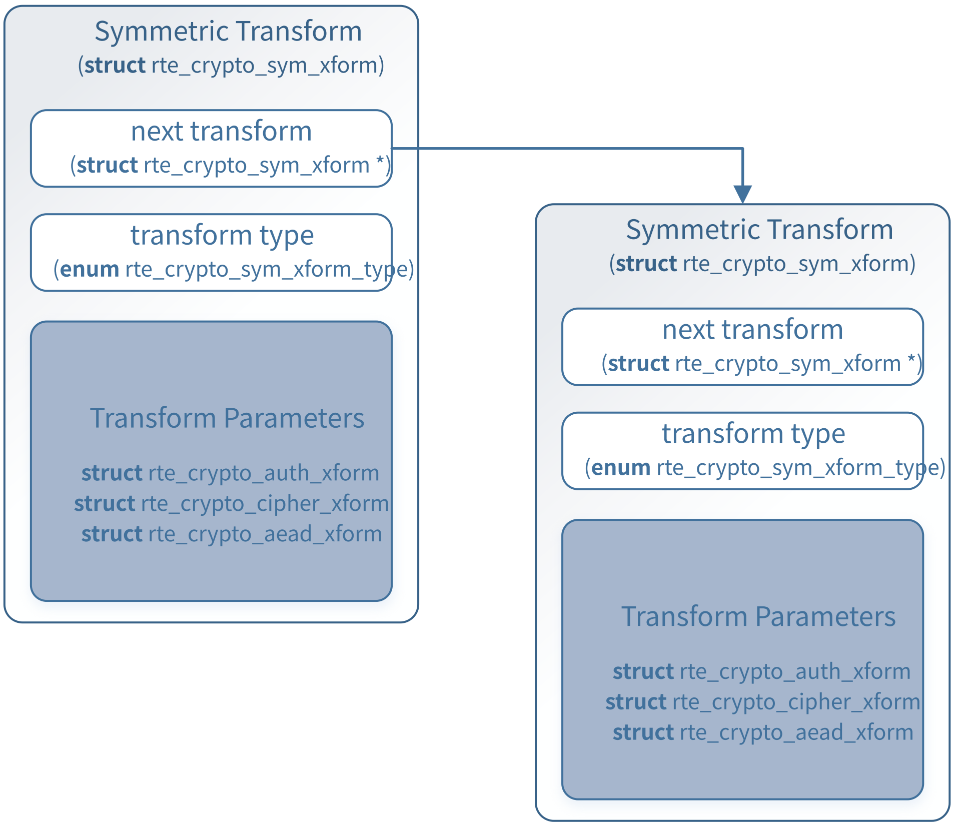
16.5.2. Transforms and Transform Chaining

Symmetric Crypto transforms (`rte_crypto_sym_xform`) are the mechanism used to specify the details of the Crypto operation. For chaining of symmetric operations such as cipher encrypt and authentication generate, the next pointer allows transform to be chained together. Crypto devices which support chaining must publish the chaining of symmetric Crypto operations feature flag. Allocation of the xform structure is in the application domain. To allow future API extensions in a backwardly compatible manner, e.g. addition of a new parameter, the application should zero the full xform struct before populating it.

Currently there are three transforms types cipher, authentication and AEAD. Also it is important to note that the order in which the transforms are passed indicates the order of the chaining.

```
struct rte_crypto_sym_xform {
    struct rte_crypto_sym_xform *next;
    /**< next xform in chain */
    enum rte_crypto_sym_xform_type type;
    /**< xform type */
    union {
        struct rte_crypto_auth_xform auth;
        /**< Authentication / hash xform */
        struct rte_crypto_cipher_xform cipher;
        /**< Cipher xform */
        struct rte_crypto_aead_xform aead;
        /**< AEAD xform */
    };
};
```

The API does not place a limit on the number of transforms that can be chained together but this will be limited by the underlying Crypto device poll mode driver which is processing the operation.


Symmetric Transform
(struct rte_crypto_sym_xform)
next transform
(struct rte_crypto_sym_xform *)
transform type
(enum rte_crypto_sym_xform_type)
Transform Parameters
struct rte_crypto_auth_xform
struct rte_crypto_cipher_xform
struct rte_crypto_aead_xform
Symmetric Transform
(struct rte_crypto_sym_xform)
next transform
(struct rte_crypto_sym_xform *)
transform type
(enum rte_crypto_sym_xform_type)
Transform Parameters
struct rte_crypto_auth_xform
struct rte_crypto_cipher_xform
struct rte_crypto_aead_xform

16.5.3. Symmetric Operations

The symmetric Crypto operation structure contains all the mutable data relating to performing symmetric cryptographic processing on a referenced mbuf data buffer. It is used for either cipher, authentication, AEAD and chained operations.

As a minimum the symmetric operation must have a source data buffer (`m_src`), a valid session (or transform chain if in session-less mode) and the minimum authentication/ cipher/ AEAD parameters required depending on the type of operation specified in the session or the transform chain.

```
struct rte_crypto_sym_op {
   struct rte_mbuf *m_src;
   struct rte_mbuf *m_dst;

   union {
      struct rte_cryptodev_sym_session *session;
      /**< Handle for the initialised session context */
      struct rte_crypto_sym_xform *xform;
      /**< Session-less API Crypto operation parameters */
   };

   union {
      struct {
         struct {
            uint32_t offset;
            uint32_t length;
         } data; /**< Data offsets and length for AEAD */

         struct {
            uint8_t *data;
            rte_iova_t phys_addr;
         } digest; /**< Digest parameters */

         struct {
            uint8_t *data;
            rte_iova_t phys_addr;
         } aad;
         /**< Additional authentication parameters */
      } aead;

      struct {
         struct {
            struct {
               uint32_t offset;
               uint32_t length;
            } data; /**< Data offsets and length for ciphering */
         } cipher;

         struct {
            struct {
               uint32_t offset;
               uint32_t length;
            } data;
            /**< Data offsets and length for authentication */

            struct {
               uint8_t *data;
               rte_iova_t phys_addr;
            } digest; /**< Digest parameters */
         } auth;
      };
   };
};
```

16.6. Synchronous mode

Some cryptodevs support synchronous mode alongside with a standard asynchronous mode. In that case operations are performed directly when calling `rte_cryptodev_sym_cpu_crypto_process` method instead of enqueuing and dequeuing an operation before. This mode of operation allows cryptodevs which utilize CPU cryptographic acceleration to have significant performance boost comparing to standard asynchronous approach. Cryptodevs supporting synchronous mode have `RTE_CRYPTODEV_FF_SYM_CPU_CRYPTO` feature flag set.

To perform a synchronous operation a call to `rte_cryptodev_sym_cpu_crypto_process` has to be made with vectorized operation descriptor (`struct rte_crypto_sym_vec`) containing:

- `num` - number of operations to perform,
- pointer to an array of size `num` containing a scatter-gather list descriptors of performed operations (`struct rte_crypto_sgl`). Each instance of `struct rte_crypto_sgl` consists of a number of segments and a pointer to an array of segment descriptors `struct rte_crypto_vec`;
- pointers to arrays of size `num` containing IV, AAD and digest information,
- pointer to an array of size `num` where status information will be stored for each operation.

Function returns a number of successfully completed operations and sets appropriate status number for each operation in the status array provided as a call argument. Status different than zero must be treated as error.

For more details, e.g. how to convert an mbuf to an SGL, please refer to an example usage in the IPsec library implementation.

16.7. Sample code

There are various sample applications that show how to use the cryptodev library, such as the L2fwd with Crypto sample application (L2fwd-crypto) and the IPsec Security Gateway application (ipsec-secgw).

While these applications demonstrate how an application can be created to perform generic crypto operation, the required complexity hides the basic steps of how to use the cryptodev APIs.

The following sample code shows the basic steps to encrypt several buffers with AES-CBC (although performing other crypto operations is similar), using one of the crypto PMDs available in DPDK.

```
/*
 * Simple example to encrypt several buffers with AES-CBC using
 * the Cryptodev APIs.
 */

#define MAX_SESSIONS       1024
#define NUM_MBUFS          1024
#define POOL_CACHE_SIZE    128
#define BURST_SIZE         32
#define BUFFER_SIZE        1024
#define AES_CBC_IV_LENGTH  16
#define AES_CBC_KEY_LENGTH 16
#define IV_OFFSET          (sizeof(struct rte_crypto_op) + \
                           sizeof(struct rte_crypto_sym_op))

struct rte_mempool *mbuf_pool, *crypto_op_pool;
struct rte_mempool *session_pool, *session_priv_pool;
unsigned int session_size;
int ret;

/* Initialize EAL. */
ret = rte_eal_init(argc, argv);
if (ret < 0)
   rte_exit(EXIT_FAILURE, "Invalid EAL arguments\n");

uint8_t socket_id = rte_socket_id();

/* Create the mbuf pool. */
mbuf_pool = rte_pktmbuf_pool_create("mbuf_pool",
                                NUM_MBUFS,
                                POOL_CACHE_SIZE,
                                0,
                                RTE_MBUF_DEFAULT_BUF_SIZE,
                                socket_id);
if (mbuf_pool == NULL)
   rte_exit(EXIT_FAILURE, "Cannot create mbuf pool\n");

/*
 * The IV is always placed after the crypto operation,
 * so some private data is required to be reserved.
 */
unsigned int crypto_op_private_data = AES_CBC_IV_LENGTH;

/* Create crypto operation pool. */
crypto_op_pool = rte_crypto_op_pool_create("crypto_op_pool",
                                        RTE_CRYPTO_OP_TYPE_SYMMETRIC,
                                        NUM_MBUFS,
                                        POOL_CACHE_SIZE,
                                        crypto_op_private_data,
                                        socket_id);
if (crypto_op_pool == NULL)
   rte_exit(EXIT_FAILURE, "Cannot create crypto op pool\n");

/* Create the virtual crypto device. */
char args[128];
const char *crypto_name = "crypto_aesni_mb0";
snprintf(args, sizeof(args), "socket_id=%d", socket_id);
ret = rte_vdev_init(crypto_name, args);
if (ret != 0)
   rte_exit(EXIT_FAILURE, "Cannot create virtual device");

uint8_t cdev_id = rte_cryptodev_get_dev_id(crypto_name);

/* Get private session data size. */
```

```
session_size = rte_cryptodev_sym_get_private_session_size(cdev_id);

#ifdef USE_TWO_MEMPOOLS
/* Create session mempool for the session header. */
session_pool = rte_cryptodev_sym_session_pool_create("session_pool",
                                MAX_SESSIONS,
                                0,
                                POOL_CACHE_SIZE,
                                0,
                                socket_id);

/*
 * Create session private data mempool for the
 * private session data for the crypto device.
 */
session_priv_pool = rte_mempool_create("session_pool",
                                MAX_SESSIONS,
                                session_size,
                                POOL_CACHE_SIZE,
                                0, NULL, NULL, NULL,
                                NULL, socket_id,
                                0);

#else
/* Use of the same mempool for session header and private data */
    session_pool = rte_cryptodev_sym_session_pool_create("session_pool",
                                MAX_SESSIONS * 2,
                                session_size,
                                POOL_CACHE_SIZE,
                                0,
                                socket_id);

    session_priv_pool = session_pool;

#endif

/* Configure the crypto device. */
struct rte_cryptodev_config conf = {
    .nb_queue_pairs = 1,
    .socket_id = socket_id
};

struct rte_cryptodev_qp_conf qp_conf = {
    .nb_descriptors = 2048,
    .mp_session = session_pool,
    .mp_session_private = session_priv_pool
};

if (rte_cryptodev_configure(cdev_id, &conf) < 0)
    rte_exit(EXIT_FAILURE, "Failed to configure cryptodev %u", cdev_id);

if (rte_cryptodev_queue_pair_setup(cdev_id, 0, &qp_conf, socket_id) < 0)
    rte_exit(EXIT_FAILURE, "Failed to setup queue pair\n");

if (rte_cryptodev_start(cdev_id) < 0)
    rte_exit(EXIT_FAILURE, "Failed to start device\n");

/* Create the crypto transform. */
uint8_t cipher_key[16] = {0};
struct rte_crypto_sym_xform cipher_xform = {
    .next = NULL,
    .type = RTE_CRYPTO_SYM_XFORM_CIPHER,
    .cipher = {
        .op = RTE_CRYPTO_CIPHER_OP_ENCRYPT,
        .algo = RTE_CRYPTO_CIPHER_AES_CBC,
        .key = {
            .data = cipher_key,
            .length = AES_CBC_KEY_LENGTH
        },
        .iv = {
            .offset = IV_OFFSET,
            .length = AES_CBC_IV_LENGTH
        }
    }
};

/* Create crypto session and initialize it for the crypto device. */
struct rte_cryptodev_sym_session *session;
session = rte_cryptodev_sym_session_create(session_pool);
if (session == NULL)
    rte_exit(EXIT_FAILURE, "Session could not be created\n");

if (rte_cryptodev_sym_session_init(cdev_id, session,
            &cipher_xform, session_priv_pool) < 0)
    rte_exit(EXIT_FAILURE, "Session could not be initialized "
            "for the crypto device\n");

/* Get a burst of crypto operations. */
struct rte_crypto_op *crypto_ops[BURST_SIZE];
if (rte_crypto_op_bulk_alloc(crypto_op_pool,
                    RTE_CRYPTO_OP_TYPE_SYMMETRIC,
                    crypto_ops, BURST_SIZE) == 0)
    rte_exit(EXIT_FAILURE, "Not enough crypto operations available\n");

/* Get a burst of mbufs. */
struct rte_mbuf *mbufs[BURST_SIZE];
if (rte_pktmbuf_alloc_bulk(mbuf_pool, mbufs, BURST_SIZE) < 0)
    rte_exit(EXIT_FAILURE, "Not enough mbufs available");

/* Initialize the mbufs and append them to the crypto operations. */
unsigned int i;
for (i = 0; i < BURST_SIZE; i++) {
    if (rte_pktmbuf_append(mbufs[i], BUFFER_SIZE) == NULL)
        rte_exit(EXIT_FAILURE, "Not enough room in the mbuf\n");
    crypto_ops[i]->sym->m_src = mbufs[i];
}

/* Set up the crypto operations. */
for (i = 0; i < BURST_SIZE; i++) {
    struct rte_crypto_op *op = crypto_ops[i];
    /* Modify bytes of the IV at the end of the crypto operation */
    uint8_t *iv_ptr = rte_crypto_op_ctod_offset(op, uint8_t *,
                                    IV_OFFSET);

    generate_random_bytes(iv_ptr, AES_CBC_IV_LENGTH);

    op->sym->cipher.data.offset = 0;
    op->sym->cipher.data.length = BUFFER_SIZE;

    /* Attach the crypto session to the operation */
    rte_crypto_op_attach_sym_session(op, session);
}

/* Enqueue the crypto operations in the crypto device. */
uint16_t num_enqueued_ops = rte_cryptodev_enqueue_burst(cdev_id, 0,
                                crypto_ops, BURST_SIZE);

/*
 * Dequeue the crypto operations until all the operations
 * are processed in the crypto device.
 */
uint16_t num_dequeued_ops, total_num_dequeued_ops = 0;
do {
    struct rte_crypto_op *dequeued_ops[BURST_SIZE];
    num_dequeued_ops = rte_cryptodev_dequeue_burst(cdev_id, 0,
                                    dequeued_ops, BURST_SIZE);
    total_num_dequeued_ops += num_dequeued_ops;

    /* Check if operation was processed successfully */
    for (i = 0; i < num_dequeued_ops; i++) {
        if (dequeued_ops[i]->status != RTE_CRYPTO_OP_STATUS_SUCCESS)
            rte_exit(EXIT_FAILURE,
                    "Some operations were not processed correctly");
    }

    rte_mempool_put_bulk(crypto_op_pool, (void **)dequeued_ops,
                                    num_dequeued_ops);
} while (total_num_dequeued_ops < num_enqueued_ops);
```

16.8. Asymmetric Cryptography

The cryptodev library currently provides support for the following asymmetric Crypto operations; RSA, Modular exponentiation and inversion, Diffie-Hellman public and/or private key generation and shared secret compute, DSA Signature generation and verification.

16.8.1. Session and Session Management

Sessions are used in asymmetric cryptographic processing to store the immutable data defined in asymmetric cryptographic transform which is further used in the operation processing. Sessions typically stores information, such as, public and private key information or domain params or prime modulus data i.e. immutable across data sets. Crypto sessions cache this immutable data in a optimal way for the underlying PMD and this allows further acceleration of the offload of Crypto workloads.

Like symmetric, the Crypto device framework provides APIs to allocate and initialize asymmetric sessions for crypto devices, where sessions are mempool objects. It is the application’s responsibility to create and manage the session mempools. Application using both symmetric and asymmetric sessions should allocate and maintain different sessions pools for each type.

An application can use `rte_cryptodev_get_asym_session_private_size()` to get the private size of asymmetric session on a given crypto device. This function would allow an application to calculate the max device asymmetric session size of all crypto devices to create a single session mempool. If instead an application creates multiple asymmetric session mempools, the Crypto device framework also provides `rte_cryptodev_asym_get_header_session_size()` to get the size of an uninitialized session.

Once the session mempools have been created, `rte_cryptodev_asym_session_create()` is used to allocate an uninitialized asymmetric session from the given mempool. The session then must be initialized using `rte_cryptodev_asym_session_init()` for each of the required crypto devices. An asymmetric transform chain is used to specify the operation and its parameters. See the section below for details on transforms.

When a session is no longer used, user must call `rte_cryptodev_asym_session_clear()` for each of the crypto devices that are using the session, to free all driver private asymmetric session data. Once this is done, session should be freed using `rte_cryptodev_asym_session_free()` which returns them to their mempool.

16.8.2. Asymmetric Sessionless Support

Asymmetric crypto framework supports session-less operations as well.

Fields that should be set by user are:

Member xform of struct rte_crypto_asym_op should point to the user created rte_crypto_asym_xform. Note that rte_crypto_asym_xform should be immutable for the lifetime of associated crypto_op.

Member sess_type of rte_crypto_op should also be set to RTE_CRYPTO_OP_SESSIONLESS.

16.8.3. Transforms and Transform Chaining

Asymmetric Crypto transforms (`rte_crypto_asym_xform`) are the mechanism used to specify the details of the asymmetric Crypto operation. Next pointer within xform allows transform to be chained together. Also it is important to note that the order in which the transforms are passed indicates the order of the chaining. Allocation of the xform structure is in the application domain. To allow future API extensions in a backwardly compatible manner, e.g. addition of a new parameter, the application should zero the full xform struct before populating it.

Not all asymmetric crypto xforms are supported for chaining. Currently supported asymmetric crypto chaining is Diffie-Hellman private key generation followed by public generation. Also, currently API does not support chaining of symmetric and asymmetric crypto xforms.

Each xform defines specific asymmetric crypto algo. Currently supported are: * RSA * Modular operations (Exponentiation and Inverse) * Diffie-Hellman * DSA * None - special case where PMD may support a passthrough mode. More for diagnostic purpose

See *DPDK API Reference* for details on each rte_crypto_xxx_xform struct

16.8.4. Asymmetric Operations

The asymmetric Crypto operation structure contains all the mutable data relating to asymmetric cryptographic processing on an input data buffer. It uses either RSA, Modular, Diffie-Hellman or DSA operations depending upon session it is attached to.

Every operation must carry a valid session handle which further carries information on xform or xform-chain to be performed on op. Every xform type defines its own set of operational params in their respective rte_crypto_xxx_op_param struct. Depending on xform information within session, PMD picks up and process respective op_param struct. Unlike symmetric, asymmetric operations do not use mbufs for input/output. They operate on data buffer of type `rte_crypto_param`.

See *DPDK API Reference* for details on each rte_crypto_xxx_op_param struct

16.9. Asymmetric crypto Sample code

There’s a unit test application test_cryptodev_asym.c inside unit test framework that show how to setup and process asymmetric operations using cryptodev library.

The following sample code shows the basic steps to compute modular exponentiation using 1024-bit modulus length using openssl PMD available in DPDK (performing other crypto operations is similar except change to respective op and xform setup).

```c
/*
 * Simple example to compute modular exponentiation with 1024-bit key
 *
 */
#define MAX_ASYM_SESSIONS   10
#define NUM_ASYM_BUFS       10

struct rte_mempool *crypto_op_pool, *asym_session_pool;
unsigned int asym_session_size;
int ret;

/* Initialize EAL. */
ret = rte_eal_init(argc, argv);
if (ret < 0)
    rte_exit(EXIT_FAILURE, "Invalid EAL arguments\n");

uint8_t socket_id = rte_socket_id();

/* Create crypto operation pool. */
crypto_op_pool = rte_crypto_op_pool_create(
                                "crypto_op_pool",
                                RTE_CRYPTO_OP_TYPE_ASYMMETRIC,
                                NUM_ASYM_BUFS, 0, 0,
                                socket_id);
if (crypto_op_pool == NULL)
    rte_exit(EXIT_FAILURE, "Cannot create crypto op pool\n");

/* Create the virtual crypto device. */
char args[128];
const char *crypto_name = "crypto_openssl";
snprintf(args, sizeof(args), "socket_id=%d", socket_id);
ret = rte_vdev_init(crypto_name, args);
if (ret != 0)
    rte_exit(EXIT_FAILURE, "Cannot create virtual device");

uint8_t cdev_id = rte_cryptodev_get_dev_id(crypto_name);

/* Get private asym session data size. */
asym_session_size = rte_cryptodev_get_asym_private_session_size(cdev_id);

/*
 * Create session mempool, with two objects per session,
 * one for the session header and another one for the
 * private asym session data for the crypto device.
 */
asym_session_pool = rte_mempool_create("asym_session_pool",
                                MAX_ASYM_SESSIONS * 2,
                                asym_session_size,
                                0,
                                0, NULL, NULL, NULL,
                                NULL, socket_id,
                                0);

/* Configure the crypto device. */
struct rte_cryptodev_config conf = {
    .nb_queue_pairs = 1,
    .socket_id = socket_id
};
struct rte_cryptodev_qp_conf qp_conf = {
    .nb_descriptors = 2048
};

if (rte_cryptodev_configure(cdev_id, &conf) < 0)
    rte_exit(EXIT_FAILURE, "Failed to configure cryptodev %u", cdev_id);
```

```
if (rte_cryptodev_queue_pair_setup(cdev_id, 0, &qp_conf,
                    socket_id, asym_session_pool) < 0)
   rte_exit(EXIT_FAILURE, "Failed to setup queue pair\n");

if (rte_cryptodev_start(cdev_id) < 0)
   rte_exit(EXIT_FAILURE, "Failed to start device\n");

/* Setup crypto xform to do modular exponentiation with 1024 bit
    * length modulus
    */
struct rte_crypto_asym_xform modex_xform = {
         .next = NULL,
         .xform_type = RTE_CRYPTO_ASYM_XFORM_MODEX,
         .modex = {
               .modulus = {
                     .data =
                     (uint8_t *)
                     ("\xb3\xa1\xaf\xb7\x13\x08\x00\x0a\x35\xdc\x2b\x20\x8d"
                     "\xa1\xb5\xce\x47\x8a\xc3\x80\xf4\x7d\x4a\xa2\x62\xfd\x61\x7f"
                     "\xb5\xa8\xde\x0a\x17\x97\xa0\xbf\xdf\x56\x5a\x3d\x51\x56\x4f"
                     "\x70\x70\x3f\x63\x6a\x44\x5b\xad\x84\x0d\x3f\x27\x6e\x3b\x34"
                     "\x91\x60\x14\xb9\xaa\x72\xfd\xa3\x64\xd2\x03\xa7\x53\x87\x9e"
                     "\x88\x0b\xc1\x14\x93\x1a\x62\xff\xb1\x5d\x74\xcd\x59\x63\x18"
                     "\x11\x3d\x4f\xba\x75\xd4\x33\x4e\x23\x6b\x7b\x57\x44\xe1\xd3"
                     "\x03\x13\xa6\xf0\x8b\x60\xb0\x9e\xee\x75\x08\x9d\x71\x63\x13"
                     "\xcb\xa6\x81\x92\x14\x03\x22\x2d\xde\x55"),
                     .length = 128
               },
               .exponent = {
                     .data = (uint8_t *)("\x01\x00\x01"),
                     .length = 3
               }
         }
};
/* Create asym crypto session and initialize it for the crypto device. */
struct rte_cryptodev_asym_session *asym_session;
asym_session = rte_cryptodev_asym_session_create(asym_session_pool);
if (asym_session == NULL)
   rte_exit(EXIT_FAILURE, "Session could not be created\n");

if (rte_cryptodev_asym_session_init(cdev_id, asym_session,
            &modex_xform, asym_session_pool) < 0)
   rte_exit(EXIT_FAILURE, "Session could not be initialized "
            "for the crypto device\n");

/* Get a burst of crypto operations. */
struct rte_crypto_op *crypto_ops[1];
if (rte_crypto_op_bulk_alloc(crypto_op_pool,
                    RTE_CRYPTO_OP_TYPE_ASYMMETRIC,
                    crypto_ops, 1) == 0)
   rte_exit(EXIT_FAILURE, "Not enough crypto operations available\n");

/* Set up the crypto operations. */
struct rte_crypto_asym_op *asym_op = crypto_ops[0]->asym;

   /* calculate mod exp of value 0xf8 */
static unsigned char base[] = {0xF8};
asym_op->modex.base.data = base;
asym_op->modex.base.length = sizeof(base);
   asym_op->modex.base.iova = base;

/* Attach the asym crypto session to the operation */
rte_crypto_op_attach_asym_session(op, asym_session);
```

```
/* Enqueue the crypto operations in the crypto device. */
uint16_t num_enqueued_ops = rte_cryptodev_enqueue_burst(cdev_id, 0,
                                        crypto_ops, 1);

/*
 * Dequeue the crypto operations until all the operations
 * are processed in the crypto device.
 */
uint16_t num_dequeued_ops, total_num_dequeued_ops = 0;
do {
    struct rte_crypto_op *dequeued_ops[1];
    num_dequeued_ops = rte_cryptodev_dequeue_burst(cdev_id, 0,
                                    dequeued_ops, 1);
    total_num_dequeued_ops += num_dequeued_ops;

    /* Check if operation was processed successfully */
    if (dequeued_ops[0]->status != RTE_CRYPTO_OP_STATUS_SUCCESS)
            rte_exit(EXIT_FAILURE,
                    "Some operations were not processed correctly");

} while (total_num_dequeued_ops < num_enqueued_ops);
```

16.9.1. Asymmetric Crypto Device API

The cryptodev Library API is described in the DPDK API Reference