Technical report

Comparison between Intel® DPDK's OpenSSL crypto and ZLIB compression Poll Mode Drivers

Written by Gereltsetseg Altangerel/Ph.D. student/
Supervised by Tejfel Máté/Ph.D./

ELTE Eötvös Loránd University, Budapest, Hungary
Faculty of Informatics, 3in Research Group, Martonvásár, Hungary

1

[1] The research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications).

Table of Contents

1. Introduction

Nowadays packet processing is implemented in software systems such as the Unix operating system instead of dedicated hardware. Although this solution gives large flexibility (deploying features rapidly in software) and cost-effectiveness, it limits packet processing speed. Main factors for limiting throughput are CPU bottleneck and usage of main memory. The latter one is divided into three problems: 1. per packet memory allocation and deallocation 2. complex sk_buff data structure 3. multiple memory copies. [1]. In order to solve these problems, high-speed packet processing frameworks such as Data Plane Development Kit (DPDK), netmap, and PF_RING are developed by using commodity hardware. These frameworks replace or extend conventional concepts by implementing their own driver and kernel level improvements. Moreover, the Intel® DPDK framework provides a set of software libraries and drivers for fast packet processing in dataplane applications of the Intel architectures [2, 3, 5]. In this technical report, based on application using compress ZLIB and crypto OpenSSL Poll mode drivers(PMDs) of DPDK, we compared these two PMDs in terms of libraries and functions.

2. Intel® DPDK overview

Figure 1 shows the DPDK components: EAL(Environment Abstraction Layer) and several libraries that are optimized for high performance as mentioned above. First of all, DPDK fulfills basic tasks, similar to what the Linux network stack does: allocating memory for network packets, buffering packet descriptors in ring-like structures and passing the packets from the NIC to the application (and vice versa). We are presenting EAL, and libraries that are necessary for these assignments include memory, queue and buffer management in the following sections briefly..

2.1 Environment Abstraction Layer (EAL)

The EAL is the main concept behind the DPDK. The EAL is a set of programming tools that let the DPDK work in a specific hardware environment and under a specific operating system. In the official DPDK repository, libraries and drivers that are part of the EAL are saved in the rte_eal directory. [6] Moreover, EAL is responsible for gaining access to low-level resources such as hardware and memory space. Also, it provides a generic interface that hides the environment specifics from the applications and libraries and is the responsibility of the initialization routine to decide how to allocate these resources (memory space, devices, timers, consoles, and so on).[5] In other words, the EAL performs all works of the kernel such as loading, launching applications, and reserving memory.[2,5] Please look at figure 2 to see the difference between packet processing with DPDK and without DPDK.

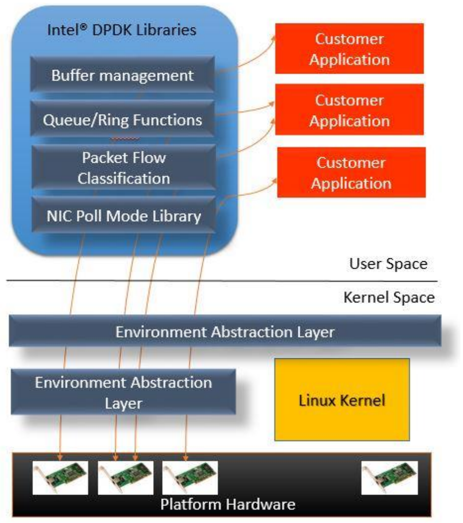

Figure 1: Intel® DPDK EAL and libraries

2.2 Managing Queues: rte_ring

In the traditional packet processing, packets received by the network card are sent to a ring buffer, which acts as a receiving queue. Packets received in the DPDK are also sent to a queue implemented on the rte_ring library. More information about the library's description could be found from the developer's guide and comments in the source code.[5]

2.3 Memory Management: rte_mempool

The EAL provides a mapping of physical memory [2]. It creates a table of descriptors which are called rte_memseg which each point to a contiguous portion of memory to circumvent the problem that available physical memory can have gaps. Those segments can then be divided into memory zones. These zones are the basic memory unit used by any further object created from an application or other libraries. One of these objects is the rte_mempool provided by the lib_rte_mempool library.

2.4 Buffer Management: rte_mbuf

In the Linux network stack, all network packets are represented by the the sk_buff data structure. In DPDK, this is done using the rte_mbuf struct, which is provided the rte_mbuf library. The buffer management approach in DPDK is reminiscent of the approach used in FreeBSD: instead of one big sk_buff struct, there are many smaller rte_mbuf buffers. The buffers are created before the DPDK application is launched and are stored in a mempool (memory is allocated by rte_mempool).

3. DPDK Poll mode drivers (PMDs)

In Figure 2, the left side shows the traditional way of packet processing on Linux, and the right side shows the packet processing way with DPDK. The traditional packet processing is very complicated and leads to some problems such as consuming a lot of bus cycles, context switching, interrupts and so on.

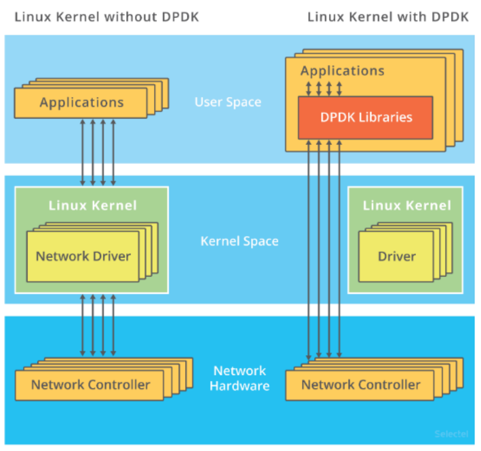

Figure 2: Linux kernel with and without DPDK

As we can see, the kernel in the right side is being skipped: interactions with the network controller are performed via special drivers and libraries. These special drivers are the DPDK poll mode drivers (PMD) which organize all further communication between the user application using DPDK libraries and network card. More specifically, a PMD accesses the RX and TX descriptors directly without any interrupts (with the exception of Link Status Change interrupts) to quickly receive, process and deliver packets in the user's application.[2] Furthermore, DPDK has developed poll mode drivers for all supported network cards and virtual devices.

4. Compression Zlib and crypto OpenSSL PMDs

A Poll Mode Driver (PMD) consists of APIs to configure the devices and their respective queues.

OpenSSL Crypto PMD: 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.[2]

Zlib Compress PMD: The compression framework provides a generic set of APIs to perform compression services as well as to query and configure compression devices both physical(hardware) and virtual(software) to perform those services. The framework currently only supports lossless compression schemes: Deflate and LZS.[2]

4.1 Compression operation representation

Compression operation is described via struct rte_comp_op, which contains both input and output data. The operation structure includes the operation type (stateless or stateful), the operation status and the priv_xform/stream handle, source, destination and checksum buffer pointers. It also contains the source mempool from which the operation is allocated. PMD updates consumed field with amount of data read from source buffer and produced field with amount of data of written into destination buffer along with status of operation. See section Produced, Consumed And Operation Status for more details.

Compression operations mempool also has an ability to allocate private memory with the operation for application's purposes. Application software is responsible for specifying all the operation specific fields in the rte_comp_op structure which are then used by the compression PMD to process the requested operation. [2]

4.2 Crypto 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.[2]

4.3 How compression and crypto operation to be processed?

There are two kinds of crypto or compression devices: physical and virtual device. A queue pair on a compress device accepts a burst of compress operations using enqueue burst API. On physical compress devices, the enqueue burst API will place the operations to be processed on the input queue of a device's hardware. In terms of virtual devices, the compress operations' processing is usually completed when the enqueue is invoked to the compressing device. The dequeue burst API will retrieve any processed operations available from the queue pair on the compress 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 place after being processed on the enqueue call. This procedure is same in crypto operation. If you want to get more information about crypto and compression operations descriptions, please access to official references.[2]

4.4 Comparison of Crypto OpenSLL and Zlib Compression PMDs

Appendix 2 is the source code for compression application. This code was tested in Ubuntu 18.04.2, DPDK 19.02 and 19.05. Refer to Appendix 1 for DPDK and compression PMD installation instructions.

All functions and structures associated with the source code are explained in the following reference table. For example, at the end of the table, the "Line 29" mark means that function or structure in Line 29 of the source code is explained in this cell of the table.

	Crypto PMD	Compression PMD
1	**int rte_eal_init (int argc, char ** argv) -** Initialize the Environment Abstraction Layer (EAL). This function is to be executed on the MASTER lcore only, as soon as possible in the application's main() function. The function finishes the initialization process before main() is called. It puts the SLAVE lcores in the WAIT state. **Parameters** **argc** A non-negative value. If it is greater than 0, the array members for argv[0] through argv[argc] (non-inclusive) shall contain pointers to strings. **argv** An array of strings. The contents of the array, as well as the strings which are pointed to by the array, may be modified by this function. **Returns** -*On success*, the number of parsed arguments, which is greater or equal to zero. After the call to rte_eal_init(), all arguments argv[x] with x < ret may have been modified by this function call and should not be further interpreted by the application. The EAL does not take any ownership of the memory used for either the argv array, or its members. -*On failure,* -1 and rte_errno is set to a value indicating the cause for failure. In some instances, the application will need to be restarted as part of clearing the issue. **”line 29”**	
	The RTE mempool structure. (#include <rte_mempool.h>, this header is included in rte_compressdev.h) **rte_mempool –** data structure for creating packet memory buffers and operation pools. **Struct Data Fields** char name [RTE_MEMZONE_NAMESIZE] – Name of mempool void * pool_config - Ring or pool to store objects. struct rte_memzone * mz unsigned int flags int socket_id uint32_t size uint32_t cache_size uint32_t elt_size uint32_t header_size uint32_t trailer_size unsigned private_data_size int32_t ops_index struct rte_mempool_cache * local_cache uint32_t populated_size struct rte_mempool_objhdr_list elt_list uint32_t nb_mem_chunks struct rte_mempool_memhdr_list mem_list void * pool_data	

	uint64_t pool_id **"line 33"**	
2	**rte_pktmbuf_pool_create(const char * name, unsigned n, unsigned cache_size, uint16_t priv_size, uint16_t data_room_size, int socket_id)** - Create a mbuf pool. This function creates and initializes a packet mbuf pool. It is a wrapper to rte_mempool functions. **Parameters** **name** The name of the mbuf pool. **n** The number of elements in the mbuf pool. The optimum size (in terms of memory usage) for a mempool is when n is a power of two minus one: n = (2^q - 1). **cache_size** Size of the per-core object cache. See rte_mempool_create() for details. **priv_size** Size of application private are between the rte_mbuf structure and the data buffer. This value must be aligned to RTE_MBUF_PRIV_ALIGN. **data_room_size** Size of data buffer in each mbuf, including RTE_PKTMBUF_HEADROOM. socket_id The socket identifier where the memory should be allocated. The value can be SOCKET_ID_ANY if there is no NUMA constraint for the reserved zone. **socket_id** The socket identifier where the memory should be allocated. The value can be SOCKET_ID_ANY if there is no NUMA constraint for the reserved zone **Returns** The pointer to the new allocated mempool, on success. NULL on error with rte_errno set appropriately. Possible rte_errno values include: -E_RTE_NO_CONFIG - function could not get pointer to rte_config structure -E_RTE_SECONDARY - function was called from a secondary process instance -EINVAL - cache size provided is too large, or priv_size is not aligned. -ENOSPC - the maximum number of memzones has already been allocated -EEXIST - a memzone with the same name already exists -ENOMEM - no appropriate memory area found in which to create memzone **"line 36"**	
	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) -Creates a crypto operation pool **Parameters** **name** pool name **type** crypto operation type, use RTE_CRYPTO_OP_TYPE_UNDEFINED for a pool which supports all operation types **nb_elts** number of elements in pool **cache_size** Number of elements to cache on lcore, see rte_mempool_create for further details about cache size **priv_size** Size of private data to allocate with each operation **socket_id** Socket to allocate memory on **Returns** On success pointer to mempool On failure NULL	struct rte_mempool* __rte_experimental **rte_comp_op_pool_create** (const char * name, unsigned int nb_elts, unsigned int cache_size, uint16_t user_size, int socket_id)-Creates an operation pool **Parameters** **name** Compress pool name **nb_elts** Number of elements in pool **cache_size** Number of elements to cache on lcore, see rte_mempool_create for further details about cache size **user_size** Size of private data to allocate for user with each operation **socket_id** Socket to identifier allocate memory on **Returns** On success pointer to mempool On failure NULL **"line 47"**
	struct rte_mempool* __rte_experimental **rte_cryptodev_sym_session_pool_create**(const char * name, uint32_t nb_elts, uint32_t elt_size, uint32_t cache_size, uint16_t priv_size, int socket_id)-Create a symmetric session mempool.	

	Parameters **name** The unique mempool name. **nb_elts** The number of elements in the mempool. **elt_size** The size of the element. This value will be ignored if it is smaller than the minimum session header size required for the system. For the user who want to use the same mempool for sym session and session private data it can be the maximum value of all existing devices' private data and session header sizes. **cache_size** The number of per-lcore cache elements **priv_size** The private data size of each session. **socket_id** The socket_id argument is the socket identifier in the case of NUMA. The value can be SOCKET_ID_ANY if there is no NUMA constraint for the reserved zone. **Returns** On success return size of the session On failure returns 0	
3	**int rte_vdev_init (const char * name, const char * args)** - Initialize a driver specified by name. (Create compress, decompress, encrypt and decrypt device) **Parameters** **name** The pointer to a driver name to be initialized. **args** The pointer to arguments used by driver initialization. **Returns** 0 on success, negative on error **"line 59, 67"**	
	struct rte_cryptodev_config - Crypto device configuration structure is used to pass the configuration parameters for socket selection and number of queue pairs. **Data Fields** **int socket_id** Socket to allocate resources on **uint16_t nb_queue_pairs** Number of queue pairs to configure on device	**rte_compressdev_config** - structure is used to pass configuration parameters. **Data Fields** **int socket_id** Socket on which to allocate resources **uint16_t nb_queue_pairs** Total number of queue pairs to configure on a device **uint16_t max_nb_priv_xforms** Max number of private_xforms which will be created on the device **uint16_t max_nb_streams** Max number of streams which will be created on the device **"line 73"**
	rte_cryptodev_qp_conf - Crypto device queue pair configuration structure. **Data Fields** **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	

	Device configuration	
	int **rte_cryptodev_configure** (uint8_t dev_id, struct rte_cryptodev_config *config) - Configure a device. This function must be invoked first before any other function in the API. This function can also be re-invoked when a device is in the stopped state. **Parameters** **dev_id** The identifier of the device to configure. **config** The crypto device configuration structure. **Returns** 0: Success, device configured. <0: Error code returned by the driver configuration function.	int __rte_experimental **rte_compressdev_configure** (uint8_t dev_id, struct rte_compressdev_config *config) Configure a device. This function must be invoked first before any other function in the API. This function can also be re-invoked when a device is in the stopped state. **Parameters** **dev_id** Compress device identifier **config** The compress device configuration **Returns** 0: Success, device configured. <0: Error code returned by the driver configuration function. **”line 80,83,100,103”**
	Configuration of Queue Pairs	
	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) - Allocate and set up a receive queue pair for a device. **Parameters** **dev_id** The identifier of the device. **queue_pair_id** The index of the queue pairs to set up. The value must be in the range [0, nb_queue_pair 1] previously supplied to rte_cryptodev_configure(). qp_conf The pointer to the configuration data to be used for the queue pair. **socket_id** The socket_id argument is the socket identifier in case of NUMA. The value can be SOCKET_ID_ANY if there is no NUMA constraint for the DMA memory allocated for the receive queue pair. **Returns** 0: Success, queue pair correctly set up. <0: Queue pair configuration failed	int __rte_experimental **rte_compressdev_queue_pair_setup** (uint8_t dev_id, uint16_t queue_pair_id, uint32_t max_inflight_ops, int socket_id) - Allocate and set up a receive queue pair for a compress and decompress device. This should only be called when the device is stopped. **Parameters** **dev_id** Compress device identifier **queue_pair_id** The index of the queue pairs to set up. The value must be in the range [0, nb_queue_pair - 1] previously supplied to rte_compressdev_configure() **max_inflight_ops** Max number of ops which the qp will have to accommodate simultaneously **socket_id** The socket_id argument is the socket identifier in case of NUMA. The value can be SOCKET_ID_ANY if there is no NUMA constraint for the DMA memory allocated for the receive queue pair **Returns** 0: Success, queue pair correctly set up. <0: Queue pair configuration failed **”line 86, 90, 106, 110”**
	Start device	
	int **rte_cryptodev_start** (uint8_t dev_id) - Start an device. The device start step is the last one and consists of setting the configured offload features and in starting the transmit and the receive units of the device. On success, all basic functions exported by the API (link status, receive/transmit, and so on) can be invoked. **Parameters**	int __rte_experimental **rte_compressdev_start** (uint8_t dev_id) - Start a device. The device start step is called after configuring the device and setting up its queue pairs. On success, data-path functions exported by the API (enqueue/dequeue, etc) can be invoked. **Parameters** **dev_id** Compress device identifier **Returns**

	dev_id The identifier of the device. **Returns** 0: Success, device started. <0: Error code of the driver device start function.	0: Success, device started. <0: Error code of the driver device start function. **"line 94, 114"**

	rte_crypto_sym_xform-Symmetric crypto transform structure.
	This is used to specify the crypto transforms required, multiple transforms can be chained together to specify a chain transforms such as authentication then cipher, or cipher then authentication. Each transform structure can hold a single transform, the type field is used to specify which transform is contained within the union
	
	Data Fields
	struct rte_crypto_sym_xform * next next xform in chain
	enum rte_crypto_sym_xform_type type xform type
	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
		rte_comp_xform Struct Reference
		
		Compression transform structure: This is used to specify the compression transforms required. Each transform structure can hold a single transform, the type field is used to specify which transform is contained within the union.
		
		enum rte_comp_xform_type type
		struct rte_comp_compress_xform compress
		struct rte_comp_decompress_xform decompress
		
		Data Fields
		enum rte_comp_xform_type type
		Compression transform types
		Enumerator:
		RTE_COMP_COMPRESS
		Compression service - compress
		RTE_COMP_DECOMPRESS
		Compression service – decompress
		
		struct rte_comp_compress_xform compress xform for compress operation
		
		rte_comp_compress_xform struct reference
		
		enum rte_comp_algorithm algo
		union {
		struct rte_comp_deflate_params deflate
		};
		int level
		uint8_t window_size
		enum rte_comp_checksum_type chksum
		enum rte_comp_hash_algorithm hash_algo
		
		Compression Algorithms (algo)
		Enumerator:
		-RTE_COMP_ALGO_NULL
		No Compression algorithm No compression. Pass-through, data is copied unchanged from source buffer to destination buffer.
		-RTE_COMP_ALGO_DEFLATE
		DEFLATE compression algorithm
		https://tools.ietf.org/html/rfc1951
		-RTE_COMP_ALGO_LZS
		LZS compression algorithm
		https://tools.ietf.org/html/rfc2395
		
		Compression Huffman Type - used by DEFLATE algorithm

		Enumerator: **-RTE_COMP_HUFFMAN_DEFAULT** PMD may choose which Huffman codes to use **-RTE_COMP_HUFFMAN_FIXED** Use Fixed Huffman codes **-RTE_COMP_HUFFMAN_DYNAMIC** Use Dynamic Huffman codes **uint8_t window_size** Base two log value of sliding window to be used. If window size can't be supported by the PMD then it may fall back to a smaller size. This is likely to result in a worse compression ratio. **Compression Hash Algorithms** Enumerator: **-RTE_COMP_HASH_ALGO_NONE** No hash **-RTE_COMP_HASH_ALGO_SHA1** SHA1 hash algorithm **-RTE_COMP_HASH_ALGO_SHA2_256** SHA256 hash algorithm of SHA2 family **struct rte_comp_decompress_xform** decompress **Data Fields** enum rte_comp_algorithm algo enum rte_comp_checksum_type chksum uint8_t window_size enum rte_comp_hash_algorithm hash_algo **Note:** All of the description for decompression are same with compression **”line 120, 135”**
	rte_cryptodev_sym_session struct **rte_cryptodev_sym_session** * **rte_cryptodev_sym_session_create** (struct rte_mempool *mempool) - Create symmetric crypto session header (generic with no private data) **Parameters** **mempool** Symmetric session mempool to allocate session objects from **Returns** On success return pointer to sym-session On failure returns NULL	
	rte_cryptodev_info - Crypto device information **const char** * driver_name **uint8_t** driver_id **struct rte_device** * device **uint64_t** feature_flags **struct rte_cryptodev_capabilities** * capabilities **unsigned** max_nb_queue_pairs	**rte_compressdev_info** - comp device information **const char** * driver_name **uint64_t** feature_flags **struct rte_compressdev_capabilities** * capabilities **uint16_t** max_nb_queue_pairs

	uint16_t min_mbuf_headroom_req **uint16_t** min_mbuf_tailroom_req **unsigned** max_nb_sessions	**”line 144, 158”**
	void rte_cryptodev_info_get(uint8_t dev_id, struct rte_cryptodev_info * dev_info) Retrieve the contextual information of a device. **Parameters** **dev_id** The identifier of the device. **dev_info** A pointer to a structure of type rte_cryptodev_info to be filled with the contextual information of the device. Note: The capabilities field of dev_info is set to point to the first element of an array of struct rte_cryptodev_capabilities. The element after the last valid element has it's op field set to RTE_CRYPTO_OP_TYPE_UNDEFINED.	**void __rte_experimental rte_compressdev_info_get** (uint8_t dev_id, struct rte_compressdev_info * dev_info) Retrieve the contextual information of a device. **Parameters** **dev_id** Compress device identifier **dev_info** A pointer to a structure of type rte_compressdev_info to be filled with the contextual information of the device. Note: The capabilities field of dev_info is set to point to the first element of an array of struct rte_compressdev_capabilities. The element after the last valid element has it's op field set to RTE_COMP_ALGO_LIST_END. **”line 147, 160”**
		int __rte_experimental **rte_compressdev_private_xform_create**(uint8_t dev_id, const struct rte_comp_xform * xform, void ** private_xform) -- This should alloc a private_xform from the device's mempool and initialise it. The application should call this API when setting up for stateless processing on a device. If it returns non-shareable, then the appl cannot share this handle with multiple in-flight ops and should call this API again to get a separate handle for every in-flight op. The handle returned is only valid for use with ops of op_type STATELESS. **Parameters** **dev_id** Compress device identifier **xform** xform data **private_xform** Pointer to where PMD's private_xform handle should be stored **Returns** if successful returns 0 and valid private_xform handle <0 in error cases Returns -EINVAL if input parameters are invalid. Returns -ENOTSUP if comp device does not support the comp transform. Returns -ENOMEM if the private_xform could not be allocated. **”line 150, 163, 220, 297”**
	static unsigned static 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) - Bulk	int __rte_experimental **rte_comp_op_bulk_alloc** (struct rte_mempool *mempool, struct rte_comp_op **ops, uint16_t nb_ops) - Bulk allocate operations from a mempool with default parameters set

	allocate crypto operations from a mempool with default parameters set **Parameters** **Mempool** crypto operation mempool **type** operation type to allocate **ops** Array to place allocated crypto operations **nb_ops** Number of crypto operations to allocate **Returns** nb_ops if the number of operations requested were allocated. 0 if the requested number of ops are not available. None are allocated in this case. (struct rte_mempool *mempool, enum rte_crypto_op_type type, struct rte_crypto_op **ops, uint16_t nb_ops) - Bulk allocate crypto operations from a mempool with default parameters set	**Parameters** **mempool** Compress operation mempool **ops** Array to place allocated operations **nb_ops** Number of operations to allocate **Returns** nb_ops: Success, the nb_ops requested was allocated 0: Not enough entries in the mempool; no ops are retrieved. **"line 176, 179"**
	static int rte_pktmbuf_alloc_bulk (struct rte_mempool * pool, struct rte_mbuf ** mbufs, unsigned count) Allocate a bulk of mbufs, initialize refcnt and reset the fields to default values. **Parameters** **pool** The mempool from which mbufs are allocated. **mbufs** Array of pointers to mbufs **count** Array size **Returns** 0: Success -ENOENT: Not enough entries in the mempool; no mbufs are retrieved. **"line 185, 188, 191, 194"**	
	static char * **rte_pktmbuf_append** (struct rte_mbuf *m, uint16_t len) - Append len bytes to an mbuf. Append len bytes to an mbuf and return a pointer to the start address of the added data. If there is not enough tailroom in the last segment, the function will return NULL, without modifying the mbuf. **Parameters** **m** The packet mbuf. **len** The amount of data to append (in bytes). **Returns** A pointer to the start of the newly appended data, or NULL if there is not enough tailroom space in the last segment **"line 202, 209"**	
	rte_crypto_op_ctod_offset(c, t, o) ((t)((char *)(c) + (o))) A macro that points to an offset from the start of the crypto operation structure (rte_crypto_op) The returned pointer is cast to type t. **Parameters** c The crypto operation. o The offset from the start of the crypto operation. t The type to cast the result into.	**rte_comp_op_ctod_offset(c, t, o)** \\ ((t)((char *)(c) + (o))) A macro that returns the physical address that points to an offset from the start of the comp operation (rte_comp_op) **Parameters** c The comp operation o The offset from the start of the comp operation to calculate address from **(it is not used in the code)**

	The C library function void *memcpy(void *str1, const void *str2, size_t n) copies n characters from memory area str2 to memory area str1 void *memcpy(void *str1, const void *str2, size_t n) Parameters str1 − This is pointer to the destination array where the content is to be copied, type-casted to a pointer of type void*. str2 − This is pointer to the source of data to be copied, type-casted to a pointer of type void*. n − This is the number of bytes to be copied. Return Value This function returns a pointer to destination, which is str1.	
	rte_pktmbuf_mtod(m, t) A macro that points to the start of the data in the mbuf. The returned pointer is cast to type t. Before using this function, the user must ensure that the first segment is large enough to accommodate its data. **Parameters** m The packet mbuf. t The type to cast the result into. **"line 239, 282, 308, 340"**	

	rte_crypto_op Struct Reference
	Cryptographic Operation.
	
	This structure contains data relating to performing cryptographic operations. This operation structure is used to contain any operation which is supported by the cryptodev API, PMDs should check the type parameter to verify that the operation is a support function of the device. Crypto operations are enqueued and dequeued in crypto PMDs using the rte_cryptodev_enqueue_burst() / rte_cryptodev_dequeue_burst() .
	
	Data Fields
	struct rte_mempool * mempool
	rte_iova_t phys_addr
	union {
	struct rte_crypto_sym_op sym [0]
	struct rte_crypto_asym_op asym [0]
	};
	uint8_t type
	uint8_t status
	uint8_t sess_type
	uint8_t reserved [3]
	uint16_t private_data_offset
	
	Explanation of Data Fields
		rte_comp_op Struct Reference
		Compression Operation.
		
		This structure contains data relating to performing a compression operation on the referenced mbuf data buffers.
		
		Comp operations are enqueued and dequeued in comp PMDs using the rte_compressdev_enqueue_burst() / rte_compressdev_dequeue_burst() APIs.
		
		Data Fields
		struct rte_mempool * mempool
		enum type
		rte_iova_t iova_addr
		struct rte_mbuf * m_src
		struct rte_mbuf * m_dst
		enum rte_comp_flush_flag flush_flag
		uint64_t input_chksum
		uint64_t output_chksum
		uint32_t consumed
		uint32_t produced
		uint64_t debug_status
		uint8_t status
		void * private_xform
		void * stream
		uint32_t offset Note:

struct rte_mempool * mempool crypto operation mempool which operation is allocated from

typedef uint64_t rte_iova_t physical address of crypto operation
struct rte_crypto_sym_op sym [0]
Symmetric operation parameters
uint8_t type operation type

operation uint8_t status -- status - this is reset to RTE_CRYPTO_OP_STATUS_NOT_PROCESSED on allocation from mempool and will be set to RTE_CRYPTO_OP_STATUS_SUCCESS after crypto operation is successfully processed by a crypto PMD

uint16_t private_data_offset
Offset to indicate start of private data (if any). The offset is counted from the start of the rte_crypto_op including IV. The private data may be used by the application to store information which should remain untouched in the library/driver

rte_crypto_sym_op Struct Reference

struct rte_mbuf * m_srcSymmetric Cryptographic Operation.

This structure contains data relating to performing symmetric cryptographic processing on a referenced mbuf data buffer.

When a symmetric crypto operation is enqueued with the device for processing it must have a valid rte_mbuf structure attached, via m_src parameter, which contains the source data which the crypto operation is to be performed on. While the mbuf is in use by a crypto operation no part of the mbuf should be changed by the application as the device may read or write to any part of the mbuf. In the case of hardware crypto devices some or all of the mbuf may be DMAed in and out of the device, so writing over the original data, though only the part specified by the rte_crypto_sym_op for transformation will be changed. Out-of-place (OOP) operation, where the source mbuf is different to the destination mbuf, is a special case. Data will be copied from m_src to m_dst. The part copied includes all the parts of the source mbuf that will be operated on, based on the cipher.data.offset+cipher.data.length and auth.data.offset+auth.data.length values in the rte_crypto_sym_op. The part indicated by the cipher parameters will be transformed, any extra data around this indicated by the auth parameters will be

uint32_t length Note:
uint8_t * digest

NOTE:

-op_type- Compression operation type: This field is not included in the official referral file. But it is used for real implementation.

- The 'offset' field specified in the reference file is invalid, but src.offset / dst.offset is valid for actual implementation.

-Also, 'length' specified in the reference doesn't work in actual implementation, but Also, src.length/ dst.length does work.

Data Fields

type- Compression operation type (Use op_type instead of this field.)

Enumerator:

-RTE_COMP_OP_STATELESS
All data to be processed is submitted in the op, no state or history from previous ops is used and none will be stored for future ops. Flush flag must be set to either FLUSH_FULL or FLUSH_FINAL.

-RTE_COMP_OP_STATEFUL
There may be more data to be processed after this op, it's part of a stream of data. State and history from previous ops can be used and resulting state and history can be stored for future ops, depending on flush flag.

struct rte_mempool * mempool --Pool from which operation is allocated
rte_iova_t iova_add -- IOVA address of this operation, IO address of the buffer

struct rte_mbuf * **m_src** -- source mbuf The total size of the input buffer(s) can be retrieved using rte_pktmbuf_pkt_len(m_src). The max data size which can fit in a single mbuf is limited by the uint16_t rte_mbuf.data_len to 64k-1. If the input data is bigger than this it can be passed to the PMD in a chain of mbufs if the PMD's capabilities indicate it supports this.

struct rte_mbuf * m_dst -- destination mbuf The total size of the output buffer(s) can be retrieved using rte_pktmbuf_pkt_len(m_dst). The max data size which can fit in a single mbuf is limited by the uint16_t copied unchanged from source to destination mbuf. Also in OOP operation the cipher.data.offset and auth.data.offset apply to both source and destination mbufs. As these offsets are relative to the data_off parameter in each mbuf this can result in the data written to the destination buffer being at a different alignment, relative to buffer start, to the data in the source buffer.

```
struct rte_mbuf * m_dst
struct rte_cryptodev_sym_session *          session
struct rte_crypto_sym_xform *      xform
struct rte_security_session *          sec_session
uint32_t offset
uint32_t length
struct {
  uint32_t   offset
  uint32_t   length
}          data
uint8_t *            data
rte_iova_t           phys_addr
struct {
  uint8_t *   data
  rte_iova_t   phys_addr
}          digest
struct {
  uint8_t *   data
  rte_iova_t   phys_addr
}          aad
struct {
  uint32_t   offset
  uint32_t   length
}          data
struct {
  uint32_t   offset
  uint32_t   length
}          data
struct {
  uint8_t *   data
  rte_iova_t   phys_addr
}          digest
```

uint32_t offset
Starting point for AEAD processing, specified as number of bytes from start of packet in source buffer.

Starting point for cipher processing, specified as number of bytes from start of data in the source buffer. The result of the cipher operation will be written back into the output buffer starting at this location.

uint32_t length
The message length, in bytes, of the source buffer on which the cryptographic operation will be computed. This must be a multiple of the block size

rte_mbuf.data_len to 64k-1. If the output data is expected to be bigger than this a chain of mbufs can be passed to the PMD if the PMD's capabilities indicate it supports this.

enum rte_comp_flush_flag   flush_flag -- Defines flush characteristics for the output data. Only applicable in compress direction

uint64_t input_chksum -- An input checksum can be provided to generate a cumulative checksum across sequential blocks in a STATELESS stream. Checksum type is as specified in xform chksum_type.

uint64_t   **output_chksum** -- If a checksum is generated it will be written in here. Checksum type is as specified in xform chksum_type.

uint32_t   **consumed** -- The number of bytes from the source buffer which were compressed/decompressed.

uint32_t   **produced** -- The number of bytes written to the destination buffer which were compressed/decompressed.

uint64_t   **debug_status** – Status of the operation is returned in the status param. This field allows the PMD to pass back extra pmd-specific debug information. Value is not defined on the API.

uint8_t status -- Operation status - use values from enum rte_comp_status. This is reset to RTE_COMP_OP_STATUS_NOT_PROCESSED on allocation from mempool and will be set to RTE_COMP_OP_STATUS_SUCCESS after operation is successfully processed by a PMD.

void *   private_xform -- Stateless private PMD data derived from an rte_comp_xform. A handle returned by rte_compressdev_private_xform_create() must be attached to operations of op_type RTE_COMP_STATELESS.

void *   stream -- Private PMD data derived initially from an rte_comp_xform, which holds state and history data and evolves as operations are processed. rte_compressdev_stream_create() must be called on a device for all STATEFUL data streams and the resulting stream attached to the one or more operations associated with the data stream. All operations in a stream must be sent to the same device.

uint32_t offset
–**src.offset** in real code Starting point for compression or decompression, specified as number of bytes from

	The message length, in bytes, of the source buffer on which the cryptographic operation will be computed. This must be a multiple of the block size if a block cipher is being used. This is also the same as the result length.	start of packet in source buffer. This offset starts from the first segment of the buffer, in case the m_src is a chain of mbufs. Starting point for checksum generation in compress direction. -**dst.offset** in real code-- Starting point for writing output data, specified as number of bytes from start of packet in dest buffer. This offset starts from the first segment of the buffer, in case the m_dst is a chain of mbufs. Starting point for checksum generation in decompress direction. **uint32_t length** -src.length in real code-- The length, in bytes, of the data in source buffer to be compressed or decompressed. Also the length of the data over which the checksum should be generated in compress direction **uint8_t * digest** -- Output buffer to store hash output, if enabled in xform. Buffer would contain valid value only after an op with flush flag = RTE_COMP_FLUSH_FULL/FLUSH_FINAL is processed successfully. Length of buffer should be contiguous and large enough to accommodate digest produced by specific hash algo. **”line 173”**
	Enqueue/Dequeue Burst APIs	
	static uint16_t **rte_cryptodev_enqueue_burst** (uint8_t dev_id, uint16_t qp_id, struct rte_crypto_op **ops, uint16_t nb_ops) Enqueue a burst of operations for processing on a crypto device. The rte_cryptodev_enqueue_burst() function is invoked to place crypto operations on the queue qp_id of the device designated by its dev_id. The nb_ops parameter is the number of operations to process which are supplied in the ops array of rte_crypto_op structures. The rte_cryptodev_enqueue_burst() 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. Parameters dev_id The identifier of the device. qp_id The index of the queue pair which packets are to be enqueued for processing. The value must be in the range [0, nb_queue_pairs - 1] previously supplied to rte_cryptodev_configure.	uint16_t __rte_experimental **rte_compressdev_enqueue_burst** (uint8_t dev_id, uint16_t qp_id, struct rte_comp_op **ops, uint16_t nb_ops) Enqueue a burst of operations for processing on a compression device. The rte_compressdev_enqueue_burst() function is invoked to place comp operations on the queue qp_id of the device designated by its dev_id. The nb_ops parameter is the number of operations to process which are supplied in the ops array of rte_comp_op structures. The rte_compressdev_enqueue_burst() 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. Note All compression operations are Out-of-place (OOP) operations, as the size of the output data is different to the size of the input data. The rte_comp_op contains both input and output parameters and is the vehicle for the application to pass data into and out of the PMD. While an op is inflight, i.e. once it has been enqueued, the private_xform or

	ops The address of an array of nb_ops pointers to rte_crypto_op structures which contain the crypto operations to be processed. nb_ops The number of operations to process. Returns The number of operations actually enqueued on the crypto device. The return value can be less than the value of the nb_ops parameter when the crypto devices queue is full or if invalid parameters are specified in a rte_crypto_op.	stream attached to it and any mbufs or memory referenced by it should not be altered or freed by the application. The PMD may use or change some of this data at any time until it has been returned in a dequeue operation. The flush flag only applies to operations which return SUCCESS. In OUT_OF_SPACE cases whether STATEFUL or STATELESS, data in dest buffer is as if flush flag was FLUSH_NONE. flush flag only applies in compression direction. It has no meaning for decompression. : operation ordering is not maintained within the queue pair. Parameters dev_id Compress device identifier qp_id The index of the queue pair on which operations are to be enqueued for processing. The value must be in the range [0, nb_queue_pairs - 1] previously supplied to rte_compressdev_configure ops The address of an array of nb_ops pointers to rte_comp_op structures which contain the operations to be processed nb_ops The number of operations to process Returns The number of operations actually enqueued on the device. The return value can be less than the value of the nb_ops parameter when the comp devices queue is full or if invalid parameters are specified in a rte_comp_op. **”line 254, 311”**
	static uint16_t **rte_cryptodev_dequeue_burst** (uint8_t dev_id, uint16_t qp_id, struct rte_crypto_op **ops, uint16_t nb_ops) Dequeue a burst of processed crypto operations from a queue on the crypto device. The dequeued operation are stored in rte_crypto_op structures whose pointers are supplied in the ops array. The rte_cryptodev_dequeue_burst() function returns the number of ops actually dequeued, which is the number of rte_crypto_op data structures effectively supplied into the ops array. A return value equal to nb_ops indicates that the queue contained at least nb_ops operations, and this is likely to signify that other processed operations remain in the devices output queue. Applications implementing a "retrieve as many processed operations as possible" policy can check this specific case and keep invoking the rte_cryptodev_dequeue_burst() function until a value less than nb_ops is returned. The rte_cryptodev_dequeue_burst() function does not provide any error notification to avoid the corresponding overhead.	uint16_t __rte_experimental **rte_compressdev_dequeue_burst** (uint8_t dev_id, uint16_t qp_id, struct rte_comp_op **ops, uint16_t nb_ops) Dequeue a burst of processed compression operations from a queue on the comp device. The dequeued operation are stored in rte_comp_op structures whose pointers are supplied in the ops array. The rte_compressdev_dequeue_burst() function returns the number of ops actually dequeued, which is the number of rte_comp_op data structures effectively supplied into the ops array. A return value equal to nb_ops indicates that the queue contained at least nb_ops operations, and this is likely to signify that other processed operations remain in the devices output queue. Applications implementing a "retrieve as many processed operations as possible" policy can check this specific case and keep invoking the rte_compressdev_dequeue_burst() function until a value less than nb_ops is returned. The rte_compressdev_dequeue_burst() function does not provide any error notification to avoid the corresponding overhead.

	Parameters **dev_id** The symmetric crypto device identifier **qp_id** The index of the queue pair from which to retrieve processed packets. The value must be in the range [0, nb_queue_pair - 1] previously supplied to rte_cryptodev_configure(). **ops** The address of an array of pointers to rte_crypto_op structures that must be large enough to store nb_ops pointers in it. **nb_ops** The maximum number of operations to dequeue. **Returns** The number of operations actually dequeued, which is the number of pointers to rte_crypto_op structures effectively supplied to the ops array.	Note : operation ordering is not maintained within the queue pair. : In case op status = OUT_OF_SPACE_TERMINATED, op.consumed=0 and the op must be resubmitted with the same input data and a larger output buffer. op.produced is usually 0, but in decompression cases a PMD may return > 0 and the application may find it useful to inspect that data. This status is only returned on STATELESS ops. : In case op status = OUT_OF_SPACE_RECOVERABLE, op.produced can be used and next op in stream should continue on from op.consumed+1 with a fresh output buffer. Consumed=0, produced=0 is an unusual but allowed case. There may be useful state/history stored in the PMD, even though no output was produced yet. **Parameters** **dev_id** Compress device identifier **qp_id** The index of the queue pair from which to retrieve processed operations. The value must be in the range [0, nb_queue_pair - 1] previously supplied to rte_compressdev_configure() **ops** The address of an array of pointers to rte_comp_op structures that must be large enough to store nb_ops pointers in it **nb_ops** The maximum number of operations to dequeue **Returns** The number of operations actually dequeued, which is the number of pointers to rte_comp_op structures effectively supplied to the ops array. **”line 273, 329”**

Table1: Functions and structs of OpenSSL and Zlib PMDs

5. Compression application

There are two compression PMDs: ZlIB and ISA-L. We developed a compression program using ZLIB PMD and described the general order of the application as follows:

1. **Preparing environment for deployment:** Installing and initialazation of DPDK and ZLIB compress PMD.
2. **Creating compress devices and memories**
   A. Creating memory pools for packet and compress operations
   B. Creating compress and decompress devices
   C. Configuring devices
   D. Configuring queue pair on the devices

    E. Starting compress devices.
    F. Defining compress and decompress xforms.
    G. Allocating memory to compress operation and processing
    H. Allocating memory to packets.

3. **Setup and process compression operations**
    A. Defining compress/decompress parameters
    B. Attaching compress xform to operation
    C. Enqueueing compress operation on the compress device for processing
    D. Dequeueing processed operation (compressed data) from the compress device.
    E. Attaching decompress xform to operation
    F. Enqueueing compressed data on the decompress device.
    G. Dequeueing uncompressed data/processed data from the decompress device.

6. Conclusion

In this technical report, we first described the general overview of DPDK, DPDK libraries and PMDs. Then we compared structs and functions of the applications using OpenSSL crypto and ZLIB compression PMD in Table 1. When programming this compress application, we found some differences in the rte_comp_op struct fields between DPDK's official reference file and actual implementation. For example, an op_type field is not specified in DPDK's official reference file, but this field is valid for actual implementation. In addition, the reference file contains only "offset" and "long" fields in rte_comp_op struct, but 'src.offset / dst.offset' and 'src.length / dst.lenght' were valid in the real code. We noted this kind of differences in Table 1. Finally, we described the general order of our compressed application. Appendix 1 and 2 include the DPDK, PMDs installation instructions, and compression source code for the PMD program.

Acknowledgment

I would like to thank the European Union, and the European Social Fund (EFOP-3.6.2-16-2017-00013, Thematic Fundamental Research Collaborations Grounding Innovation in Informatics and Infocommunications) for funding this work. I would also like to thank my supervisor for a great deal of valuable advice.

Reference

1. Dominik Scholz(2014), A Look at Intel’s Dataplane Development Kit
2. Intel DPDK: Programmers guide, DPDK 19.02 and 19.05
3. Intel DPDK: Packet Processing on Intel Architecture, Presentation slides, 2012
4. Test application of compress PMD: http://git.dpdk.org/dpdk/plain/app/test/test_compressdev.c
5. Andrej Yemelianov(2016), Introduction to DPDK: Architecture and Principles (https://blog.selectel.com/introduction-dpdk-architecture-principles/)

Appendix 1: DPDK, crypto OpenSLL PMD, and compression ZLIB PMD installation guide

1. Download DPDK

```
wget https://fast.dpdk.org/rel/dpdk-19.02.tar.xz
tar xJf dpdk-<version>.tar.xz
```

2. Go to DPDK directory

```
cd dpdk-<version>
```

3. Install the following prerequisite packages

```
apt-get install libpcap-dev gcc make hugepages nim
apt-get install linux-headers-generic
apt-get install libnuma-dev
apt-get install libelf-dev
```

3. Install ZLIB and OpenSSL library

```
apt-get install zlib1g-dev
apt-get install libssl-dev
```

4. In order to enable this virtual compression PMD and crypto OpenSSL, user must:

Set CONFIG_RTE_LIBRTE_PMD_ZLIB=y in config/common_base.
Set CONFIG_RTE_LIBRTE_PMD_OPENSSL=y in config/common_base.

5. Build and install dpdk

```
make config T=x86_64-native-linuxapp-gcc
make
make install
```

Appendix 2: Source code of application using ZLIB compression PMD.

```
#include <rte_compressdev.h>
#include <unistd.h>
#include <rte_bus_vdev.h>

#define DEFAULT_WINDOW_SIZE   15
#define DEQUEUE_WAIT_TIME     10000
#define NUM_MAX_XFORMS        16
#define NUM_MAX_INFLIGHT_OPS  512
#define NUM_MBUFS             8192
#define POOL_CACHE_SIZE       256
#define NUM_OPS               1
#define BUFFER_SIZE           32
#define PRIV_SIZE             16

void print_bytes(unsigned char *, size_t);
void print_bytes(unsigned char *c, size_t n)
```

```
    {
    size_t i;
    for(i = 0; i < n; i++)
        printf("%02X ", c[i]);
    printf("\n");
    }
int main(int argc, char **argv)
    {
srand ((unsigned int) time (NULL));
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();
struct rte_mempool *mbuf_pool, *op_pool;

/* 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");
unsigned int comp_op_private_data = PRIV_SIZE;

/* Create compression operation pool. */
op_pool = rte_comp_op_pool_create(      "op_pool",
                                        NUM_MBUFS,
                                        POOL_CACHE_SIZE,
                                        comp_op_private_data,
                                        socket_id);
if (op_pool == NULL)
    rte_exit(EXIT_FAILURE, "Cannot create crypto op pool\n");

/** Compress device ***********************************************************/
char args[128];
const char *compress_name = "compress_zlib";
snprintf(args, sizeof(args), "socket_id=%d", socket_id);
ret = rte_vdev_init(compress_name, args);
if (ret != 0)
    rte_exit(EXIT_FAILURE, "Cannot create virtual device");
uint8_t cdev_id_compress = rte_compressdev_get_dev_id(compress_name);

/** Decompress device ***********************************************************/
const char *compress_name2 = "compress_zlib1";
snprintf(args, sizeof(args), "socket_id=%d", socket_id);
ret = rte_vdev_init(compress_name2, args);
if (ret != 0)
    rte_exit(EXIT_FAILURE, "Cannot create virtual device");
uint8_t cdev_id_decompress = rte_compressdev_get_dev_id(compress_name2);

/* Configure the compress device. Allocation of resourses */
```

```
struct rte_compressdev_config conf = { //structure is used to pass configuration parameters.
                .socket_id = rte_socket_id(),
                .nb_queue_pairs = 1,
                .max_nb_priv_xforms = NUM_MAX_XFORMS,
                .max_nb_streams = 1
        };

if (rte_compressdev_configure(cdev_id_compress, &conf) < 0) // rte_compressdev_configure API is used to configure a compression device
    rte_exit(EXIT_FAILURE, "Failed to configure compressdev %u", cdev_id_compress);

if (rte_compressdev_configure(cdev_id_compress, &conf) == 0)
    printf(  "Compress device is succesfully configured. \n");

if (rte_compressdev_queue_pair_setup(cdev_id_compress, 0, NUM_MAX_INFLIGHT_OPS,
                        rte_socket_id()) < 0)
    rte_exit(EXIT_FAILURE, "Failed to setup queue pair\n");

if (rte_compressdev_queue_pair_setup(cdev_id_compress, 0, NUM_MAX_INFLIGHT_OPS,
                        rte_socket_id()) == 0)
    printf("The queue pair is successfully configured in the compress device. \n");

if (rte_compressdev_start(cdev_id_compress) < 0)
    rte_exit(EXIT_FAILURE, "Failed to start device\n");
else
    printf("Compress device successfully created \n");

/* Configure the decompress device. Allocation of resources */
if (rte_compressdev_configure(cdev_id_decompress, &conf) < 0)
    rte_exit(EXIT_FAILURE, "Failed to configure compressdev %u", cdev_id_decompress);

if (rte_compressdev_configure(cdev_id_decompress, &conf) == 0)
    printf(  "Decompress device is succesfully configured. \n");

if (rte_compressdev_queue_pair_setup(cdev_id_decompress, 0, NUM_MAX_INFLIGHT_OPS,
                        rte_socket_id()) < 0)
    rte_exit(EXIT_FAILURE, "Failed to setup queue pair\n");

if (rte_compressdev_queue_pair_setup(cdev_id_decompress, 0, NUM_MAX_INFLIGHT_OPS,
                        rte_socket_id()) == 0)
    printf("The queue pair is successfully configured in the decompress device. \n");

if (rte_compressdev_start(cdev_id_decompress) < 0)
    rte_exit(EXIT_FAILURE, "Failed to start device\n");
else
    printf("Decompress device successfully created \n");

/* Setup compress transform */
struct rte_comp_xform compress_xform = {
    .type = RTE_COMP_COMPRESS,
    .compress = {
        .algo = RTE_COMP_ALGO_DEFLATE,
        .deflate = {
            .huffman = RTE_COMP_HUFFMAN_DEFAULT
        },
        .level = RTE_COMP_LEVEL_PMD_DEFAULT,
        .chksum = RTE_COMP_CHECKSUM_NONE,
        .window_size = DEFAULT_WINDOW_SIZE,
        .hash_algo = RTE_COMP_HASH_ALGO_NONE
    }
};

/* Setup decompress transform */
struct rte_comp_xform decompress_xform = {
    .type = RTE_COMP_DECOMPRESS,
    .decompress = {
        .algo = RTE_COMP_ALGO_DEFLATE,
        .chksum = RTE_COMP_CHECKSUM_NONE,
        .window_size = DEFAULT_WINDOW_SIZE,
    }
};

struct rte_compressdev_info dev_info;
void *priv_xform = NULL;
int shareable;
rte_compressdev_info_get(cdev_id_compress, &dev_info);
if(dev_info.capabilities->comp_feature_flags & RTE_COMP_FF_SHAREABLE_PRIV_XFORM)
        {
    rte_compressdev_private_xform_create(cdev_id_compress, &compress_xform, &priv_xform);
        printf("success");
        }
else
        {
    shareable = 0;
    printf("Non-shareable is configured in the compress device. \n");
        }
struct rte_compressdev_info dev_info2;
void *priv_xform2 = NULL;
rte_compressdev_info_get(cdev_id_decompress, &dev_info2);
if(dev_info.capabilities->comp_feature_flags & RTE_COMP_FF_SHAREABLE_PRIV_XFORM)
        {
    rte_compressdev_private_xform_create(cdev_id_decompress, &decompress_xform, &priv_xform2);
    printf("success");
        }
else
        {
    shareable = 0;
    printf("Non-shareable is configured in the decompress device.  \n");
        }

/* Create an op pool and allocate ops */
struct rte_comp_op *comp_ops[NUM_OPS], *processed_ops[NUM_OPS] ; /*rte_comp_op structure contains
data relating to performing
                                                              a compression operation on
the re    ferenced mbuf data buffers.*/

if (rte_comp_op_bulk_alloc(op_pool, comp_ops, NUM_OPS) == 0)
    rte_exit(EXIT_FAILURE, "Not enough crypto operations available\n");

 if (rte_comp_op_bulk_alloc(op_pool, processed_ops, NUM_OPS) == 0)
    rte_exit(EXIT_FAILURE, "Not enough crypto operations available\n");

```

```
/* Prepare source and destination mbufs for compression operations*/
struct rte_mbuf *comp_mbufs[BUFFER_SIZE], *uncomp_mbufs[BUFFER_SIZE];

if (rte_pktmbuf_alloc_bulk(mbuf_pool, comp_mbufs, NUM_OPS) < 0)
    rte_exit(EXIT_FAILURE, "Not enough room in the source mbuf");

if (rte_pktmbuf_alloc_bulk(mbuf_pool, comp_mbufs, NUM_OPS) == 0)
    printf("Allocate a source bulk of comp_mbufs, initialize refcnt and reset the fields to default values\n");

if (rte_pktmbuf_alloc_bulk(mbuf_pool, uncomp_mbufs, NUM_OPS) < 0)
    rte_exit(EXIT_FAILURE, "Not enough room in the dst mbuf");

if (rte_pktmbuf_alloc_bulk(mbuf_pool, uncomp_mbufs, NUM_OPS) == 0)
    printf("Allocate a destination bulk of mbufs, initialize refcnt and reset the fields to default values\n");

/* Prepare source and destination mbufs for compression operations*/
unsigned int i;

for (i = 0; i < NUM_OPS; i++)
        {
        if (rte_pktmbuf_append(uncomp_mbufs[i], BUFFER_SIZE) == NULL)
        rte_exit(EXIT_FAILURE, "Not enough room in the mbuf\n");
        comp_ops[i]->m_src=uncomp_mbufs[i];
        }

for (i = 0; i < NUM_OPS; i++)
        {
    if (rte_pktmbuf_append(comp_mbufs[i], BUFFER_SIZE) == NULL)
                rte_exit(EXIT_FAILURE, "Not enough room in the mbuf\n");
            comp_ops[i]->m_dst=comp_mbufs[i];
        }
uint16_t num_enqd, num_deqd;

// Set up the compress operations.
for (i = 0; i < NUM_OPS; i++)
        {
if (!shareable)
        {
        if(rte_compressdev_private_xform_create(cdev_id_compress, &compress_xform, &comp_ops[i]->private_xform)==0)
                {
                printf("Compresss xform created in compress device \n");
                }

        }
else
        {
        comp_ops[i]->private_xform = priv_xform;
        }

        comp_ops[i]->op_type = RTE_COMP_OP_STATELESS;
        comp_ops[i]->flush_flag = RTE_COMP_FLUSH_FINAL;
        comp_ops[i]->src.offset = 0;
        comp_ops[i]->dst.offset = 0;
        comp_ops[i]->src.length = BUFFER_SIZE;
        printf("Original data:  ");
        int j;
        struct rte_mbuf *m= comp_ops[i]->m_src;
        unsigned char *data = rte_pktmbuf_mtod(m, unsigned char*);  //A macro that points to the start of the data in the mbuf.
        for (j = 1; j < 16; j++)
        data[j] = 9;
        comp_ops[i]->input_chksum = 0;
        print_bytes(data, 16);

    /*rte_compressdev_enqueue_burst---Enqueue a burst of operations for processing on a compression device.
     *The rte_comp_op contains both input and output parameters and is the
     * vehicle for the application to pass data into and out of the PMD. While an
     * op is inflight, i.e. once it has been enqueued, the private_xform
     * attached to it and any mbufs or memory referenced by it should not be altered
     * or freed by the application. The PMD may use or change some of this data at
     * any time until it has been returned in a dequeue operation.
     *
     */
num_enqd = rte_compressdev_enqueue_burst(cdev_id_compress, 0, comp_ops , NUM_OPS);
if(comp_ops[i]->status == RTE_COMP_OP_STATUS_SUCCESS)
        {
        printf("Enqueue operation completed succesfully in compress device  \n");
        }
                printf("comp_ops[i]->consumed \n");
                printf("%d\n", comp_ops[i]->consumed);
                printf("comp_ops[i]->produced \n");
                printf("%d\n", comp_ops[i]->produced);
                printf("processed_ops[i]->consumed \n");
                printf("%d\n", processed_ops[i]->consumed);
                printf(" processed_ops[i]->produced \n");
                printf("%d\n", processed_ops[i]->produced);
        }

uint16_t total_processed_ops=0;
usleep(DEQUEUE_WAIT_TIME);
    do {
/*Dequeue a burst of processed compression operations from a queue on the compress device.*/
num_deqd = rte_compressdev_dequeue_burst(cdev_id_compress, 0 , processed_ops, NUM_OPS);
total_processed_ops += num_deqd;
/* Check if operation was processed successfully */
for (i = 0; i < num_deqd; i++) {
        if (processed_ops[i]->status != RTE_COMP_OP_STATUS_SUCCESS)
            rte_exit(EXIT_FAILURE,"Some operations were not processed correctly");
        else {
            printf("Dequeu operation completed successfully in compress device \n");
            struct rte_mbuf *m = processed_ops[i]->m_dst;
            unsigned char *data = rte_pktmbuf_mtod(m, unsigned char*);
            printf("Compressed data: ");
            print_bytes(data, 16);
           }
                    }

rte_mempool_put_bulk(op_pool, (void **)processed_ops,num_deqd);
     }
while (total_processed_ops < num_enqd); //push next op

```

```
//------------------------------------------------------------------
for (i = 0; i < NUM_OPS; i++) {
if (!shareable)
        {
/*rte_compressdev_private_xform_create should alloc a private_xform from the device's mempool and initialise it.*/
        rte_compressdev_private_xform_create(cdev_id_decompress, &decompress_xform, &comp_ops[i]->private_xform);
                printf("Decompress xform created in decompress device \n");
    }
else
    {
        comp_ops[i]->private_xform = priv_xform2;
        }
    printf("Compressed data in decompress device:  ");
    comp_ops[i]->m_src=processed_ops[i]->m_dst;
    comp_ops[i]->m_dst=uncomp_mbufs[i];
    struct rte_mbuf *m= comp_ops[i]->m_src;
    unsigned char *data = rte_pktmbuf_mtod(m, unsigned char*);
    print_bytes(data, 16);

num_enqd = rte_compressdev_enqueue_burst(cdev_id_decompress, 0, comp_ops , NUM_OPS);
if(comp_ops[i]->status == RTE_COMP_OP_STATUS_SUCCESS)
        {
        printf("Enqueue operation completed succesfully in decompress device  \n");
        }
printf("comp_ops[i]->consumed \n");
printf("%d\n", comp_ops[i]->consumed);
printf("comp_ops[i]->produced \n");
printf("%d\n", comp_ops[i]->produced);
printf("processed_ops[i]->consumed \n");
printf("%d\n", processed_ops[i]->consumed);
printf(" processed_ops[i]->produced \n");
printf("%d\n", processed_ops[i]->produced);
                          }
total_processed_ops=0;
usleep(DEQUEUE_WAIT_TIME);
 do {
/*Dequeue a burst of processed compression operations from a queue on the compress device.*/
num_deqd = rte_compressdev_dequeue_burst(cdev_id_decompress, 0 , processed_ops, NUM_OPS);
total_processed_ops += num_deqd;
/* Check if operation was processed successfully */
for (i = 0; i < num_deqd; i++) {
if (processed_ops[i]->status != RTE_COMP_OP_STATUS_SUCCESS)
            rte_exit(EXIT_FAILURE,
                    "Some operations were not processed correctly");
else
        {   printf("Dequeue operation completed succesfully in decompress device  \n");

            struct rte_mbuf *m = processed_ops[i]->m_dst;
            unsigned char *data = rte_pktmbuf_mtod(m, unsigned char*);
            printf("Original data in decompress device: ");
            print_bytes(data, 16);

        }
    }
 }
while (total_processed_ops < num_enqd);
}
```