Intel® QuickAssist Technology

API Programmer's Guide

September 2017

Document Number: 330684-002US

Legal Disclaimer

Contents

Figures

Tables

Revision History

Date	Revision	Description
September 2017	002	Updates to code samples.
June 2014	001	First public version of the document. Based on Intel Confidential document number 442844-1.3 with the revision history of that document retained for reference purposes.
May 2014	1.3	Added new API function, cpaCySymSessionCtxGetDynamicSize(), to Section 3.2.2.2, plus other minor updates.
January 2014	1.2	Updated Listing 26, 27, and 30 in Section 3.2
March 2013	1.1	Updates for stateless data compression samples: • Updated Section 4.3 • Added Section 4.4
September 2012	1.0	Initial release of document.

§

1 About This Document

1.1 Purpose

This API programmer's guide describes the sample code that demonstrates how to use the Intel® QuickAssist Technology APIs.

1.2 Intended Audience

This document is intended to be used by software engineers who wish to develop application software that uses the Intel® QuickAssist Technology APIs to accelerate the supported workloads and/or services.

1.3 Using This Document

This document is structured as follows:

- Chapter 2 describes aspects common to all Intel QuickAssist Technology APIs.
- Chapter 3 describes the Intel QuickAssist Technology Cryptographic API.
- Chapter 4 describes the Intel QuickAssist Technology Data Compression API.

Code for all the examples in this document is contained in the software package and, after installation, can be found in a sub-directory of the following directory:

```
quickassist\lookaside\access_layer\src\sample_code\functional
```

1.4 Related Documents

Table 1. Related Documents

Title	Reference Number
Intel® QuickAssist Technology Cryptographic API Reference Manual	330685
Intel® QuickAssist Technology Data Compression API Reference Manual	330686
Intel® QuickAssist Technology Performance Optimization Guide	330687
Intel® QuickAssist Technology Programmer's Guide for Linux - Hardware Version 1.7*	336210
Intel® Communications Chipset 8925 to 8955 Series Software Programmer's Guide	330751

Title	Reference Number
Intel® Communications Chipset 8900 to 8920 Series Software Programmer's Guide	330753
Intel® Atom™ Processor C2000 Product Family for Communications Infrastructure Software Programmer's Guide	330755
NIST publication SP800-38C *Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality*	SP800-38C
NIST publication SP800-38D *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*	SP800-38D
NIST SP 800-90, March 2007 *Recommendation for Random Number Generation Using Deterministic Random Bit Generators (Revised)*	SP 800-90

1.5 Conventions and Terminology

The following conventions are used in this manual:

- `Courier font` is used for code examples, command line entries, API names, filenames, directory paths, and executables.
- **Bold** text - graphical user interface entries and buttons

The following terms and acronyms are used in this manual:

Table 2. Terminology

Term	Definition
AAD	Additional Authenticated Data
API	Application Programming Interface
CCM	Counter mode with Cipher-block chaining Message authentication code
CPM	Content Processing Module
CY	Cryptographic
DC	Data Compression
DRBG	Deterministic Random Bit Generator
DSA	Digital Signature Algorithm
EC	Elliptic Curve
GCM	Galois Counter Mode

Term	Definition
HMAC	Hashed Message Authenticate Code
ICV	Integrity Check Value
MAC	Message Authentication Code
NRBG	Non-Deterministic Random Bit Generator
PKE	Public Key Encryption
QAT	Intel® QuickAssist Technology
RBG	Random Bit Generation
RSA	A public key encryption algorithm created by Rivest, Shamir, and Adleman
SSL	Secure Sockets Layer
TLS	Transport Layer Security (SSL successor)

2 Base API and API Conventions

This chapter describes aspects common to all Intel® QuickAssist Technology APIs, starting with the base API and followed by conventions.

2.1 Intel® QuickAssist Technology Base API

The Base API is a top level API definition for Intel® QuickAssist Technology. It contains structures, data types and definitions that are common across the interface.

2.1.1 Data Buffer Models

Data buffers are passed across the API interface in one of the following formats:

- Flat Buffers represent a single region of physically contiguous memory, and are described in detail in Section 2.1.1.1.
- Scatter Gather Lists are essentially an array of flat buffers, for cases where the memory is not all physically contiguous. These are described in detail in Section 2.1.1.2.

2.1.1.1 Flat Buffers

Flat buffers are represented by the type `CpaFlatBuffer`, defined in the file `cpa.h`. It consists of two fields:

- Data pointer `pData`: points to the start address of the data or payload. The data pointer is a virtual address; however, the actual data pointed to is required to be in contiguous and DMAable physical memory. This buffer type is typically used when simple, unchained buffers are needed.
- Length of this buffer `dataLenInBytes`: specified in bytes.

For data plane APIs (`cpa_sym_dp.h` and `cpa_dc_dp.h`), a flat buffer is represented by the type `CpaPhysFlatBuffer`, also defined in `cpa.h`. This is similar to the `CpaFlatBuffer` structure; the difference is that, in this case, the data pointer, `bufferPhysAddr`, is a physical address rather than a virtual address.

Figure 1 shows the layout of a flat buffer.

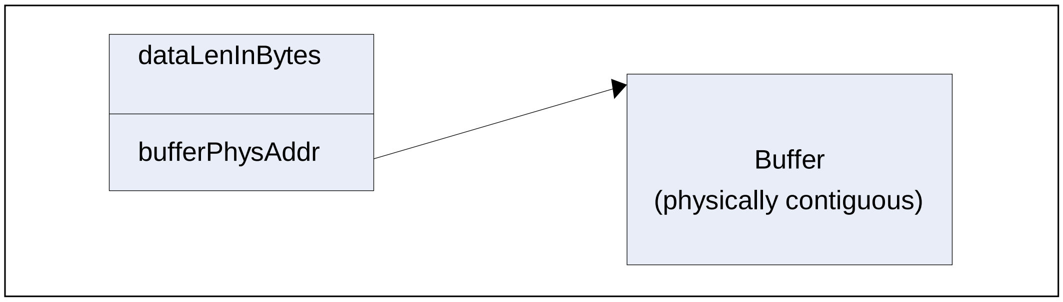


Figure 1. Flat Buffer Diagram

2.1.1.2 Scatter Gather Lists

A scatter gather list is defined by the type `CpaBufferList`, also defined in the file `cpa.h`. This buffer structure is typically used where more than one flat buffer can be provided to a particular API. The buffer list contains four fields, as follows:

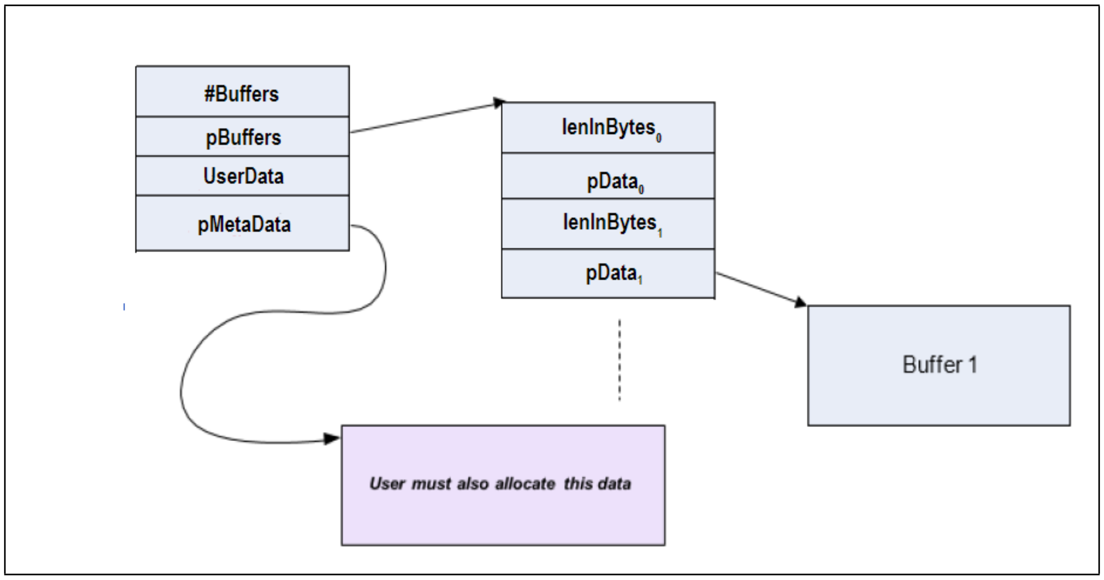
- Number of buffers in the list
- Pointer to an unbounded array of flat buffers
- `UserData`: an opaque field; is not read or modified internally by the API. This field could be used to provide a pointer back into an application data structure providing the context of the call.
- Pointer to meta data required by the API: The meta data is required for internal use by the API. The memory for this buffer needs to be allocated by the client as contiguous data. The size of this meta data buffer is obtained by calling `cpaCyBufferListGetMetaSize` for crypto, `cpaBufferLists` and `cpaDcBufferListGetMetaSize` for data compression.

The memory required to hold the `CpaBufferList` structure and the array of flat buffers is not required to be physically contiguous. However, the flat buffer data pointers and the meta data pointer are required to reference physically contiguous DMAable memory.

Note: There is a performance impact when using scatter gather lists instead of flat buffers. Refer to the Intel® QuickAssist Technology Performance Optimization Guide for additional information.

Figure 2 shows a graphical representation of a scatter gather buffer list.

Figure 2. Scatter Gather List Diagram



For data plane APIs (cpa_sym_dp.h and cpa_dc_dp.h) a region of memory which is not physically contiguous is described using the `CpaPhysBufferList` structure. This is similar to the `CpaBufferList` structure; the difference, in this case, the individual flat buffers are represented using physical rather than virtual addresses.

2.2 Intel® QuickAssist Technology API Conventions

2.2.1 Instance Discovery

Intel® QuickAssist Technology API supports multiple instances. An instance represents a “channel” to a specific hardware accelerator. Multiple instances can access the same hardware accelerator (i.e. the relationship between instances and a hardware accelerator is N:1). The instance is identified using the `CpaInstanceHandle` handle type. This handle represents a specific instance within the system and is passed as a parameter to all API functions that operate on instances.

Instance discovery is achieved through service specific API invocations. This section describes instance discovery for data compression (DC), however, the flow of the calls is similar for the cryptographic service.

Listing 1. Getting an Instance

```
void sampleDcGetInstance(CpaInstanceHandle *pDcInstHandle)
{
    CpaInstanceHandle dcInstHandles[MAX_INSTANCES];
    Cpa16U numInstances = 0;
    CpaStatus status = CPA_STATUS_SUCCESS;

    *pDcInstHandle = NULL;

    status = cpaDcGetNumInstances(&numInstances);
    if ((status == CPA_STATUS_SUCCESS) && (numInstances > 0)) {
        status = cpaDcGetInstances(MAX_INSTANCES, dcInstHandles);
        if (status == CPA_STATUS_SUCCESS) {
            *pDcInstHandle = dcInstHandles[0];
        }
    }

    if (0 == numInstances) {
        PRINT_ERR("No instances found for 'SSL'\n");
        PRINT_ERR("Please check your section names in the config file.\n");
        PRINT_ERR("Also make sure to use config file version 2.\n");
    }
}
```

In this example, the number of DC instances available to the application is queried via the `cpaDcGetNumInstances` call. The application obtains the instance handle of the first instance.

The next example shows the application querying the capabilities of the data compression implementation and verifying the required functionality is present. Each service implementation exposes the capabilities that have been implemented and are available. Capabilities include algorithms, common features, and limits to variables. Each service has a unique capability matrix, and each implementation identifies and describes its particular implementation through its capabilities API.

Listing 2. Querying and starting an Instance

```
status = cpaDcQueryCapabilities(dcInstHandle, &cap);
if (status != CPA_STATUS_SUCCESS) {
    return status;
}

if (!cap.statefulDeflateCompression || !cap.statefulDeflateDecompression ||
    !cap.checksumCRC32) {
    PRINT_ERR("Error: Unsupported functionality\n");
    return CPA_STATUS_FAIL;
}

/*
 * Set the address translation function for the instance
 */
status = cpaDcSetAddressTranslation(dcInstHandle, sampleVirtToPhys);
if (CPA_STATUS_SUCCESS == status) {

    /* Start DataCompression component
     * In this example we are performing static compression so
     * an intermediate buffer is not required */
    PRINT_DBG("cpaDcStartInstance\n");
    status = cpaDcStartInstance(dcInstHandle, 0, NULL);
}
```

In the example, the application requires stateful deflate compression and decompression and support for CRC32 checksums. The example also sets the address translation function for the instance. The specified function is used by the API to perform any required translation of a virtual address to a physical address. Finally, the instance is started.

2.2.2 Modes of Operation

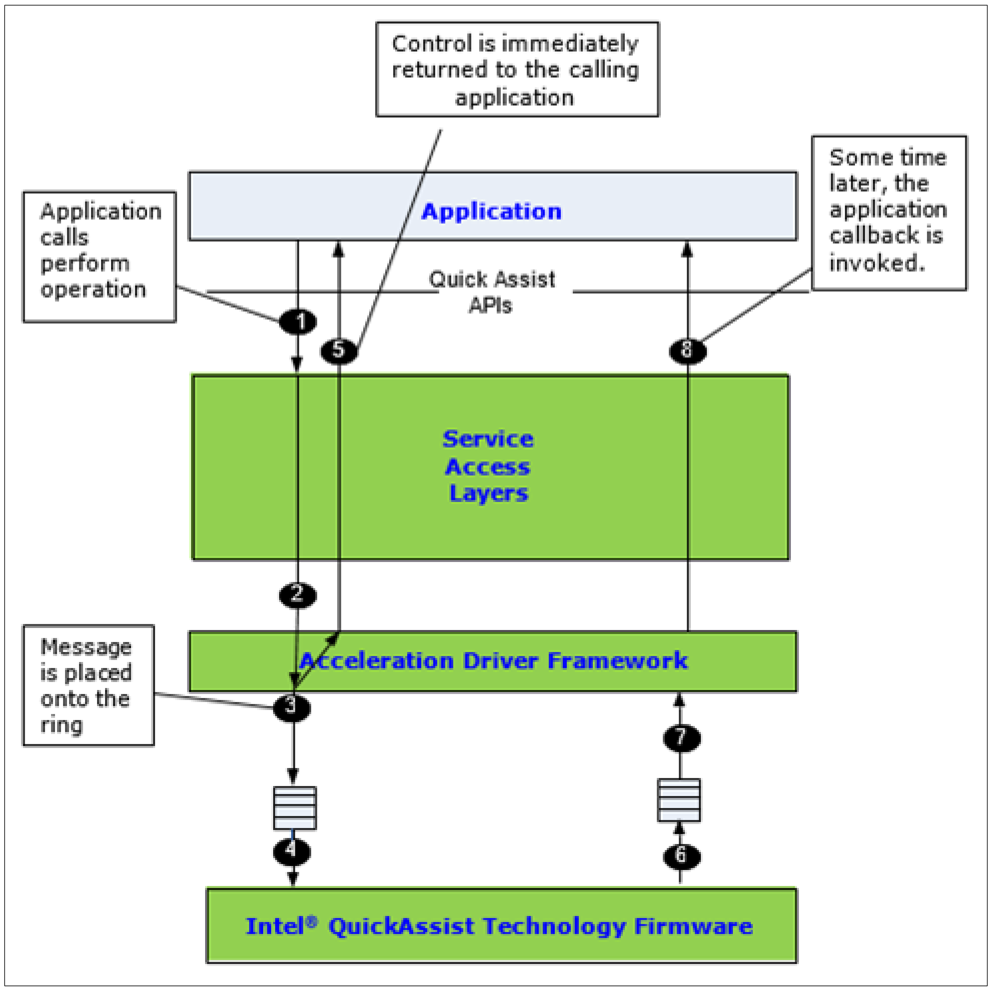
The Intel® QuickAssist Technology API supports both synchronous and asynchronous modes of operation. For optimal performance, the application should be capable of submitting multiple outstanding requests to the acceleration engines. Submitting multiple outstanding requests minimizes the processing latency on the acceleration engines. This can be done by submitting requests asynchronously or by submitting requests in synchronous mode using multi-threading in the application.

Developers can select the mode of operation that best aligns with their application and system architecture.

2.2.2.1 Asynchronous Operation

To invoke the API asynchronously, the user supplies a callback function to the API as shown in Figure 3. Control returns to the client once the request has been sent to the hardware accelerator, and the callback is invoked when the engine completes the operation. The mechanism used to invoke the callback is implementation dependent. For some implementations, the callback will be invoked as part of an interrupt handler bottom half. For other implementations, the callback will be invoked in the context of a polling thread. In this case, the user application is responsible for creating and scheduling this polling thread. Please refer to implementation specific documentation (such as the Intel® Communications Chipset 8900 to 8920 Series Software Programmer's Guide) for more details.

Figure 3. Asynchronous Operation

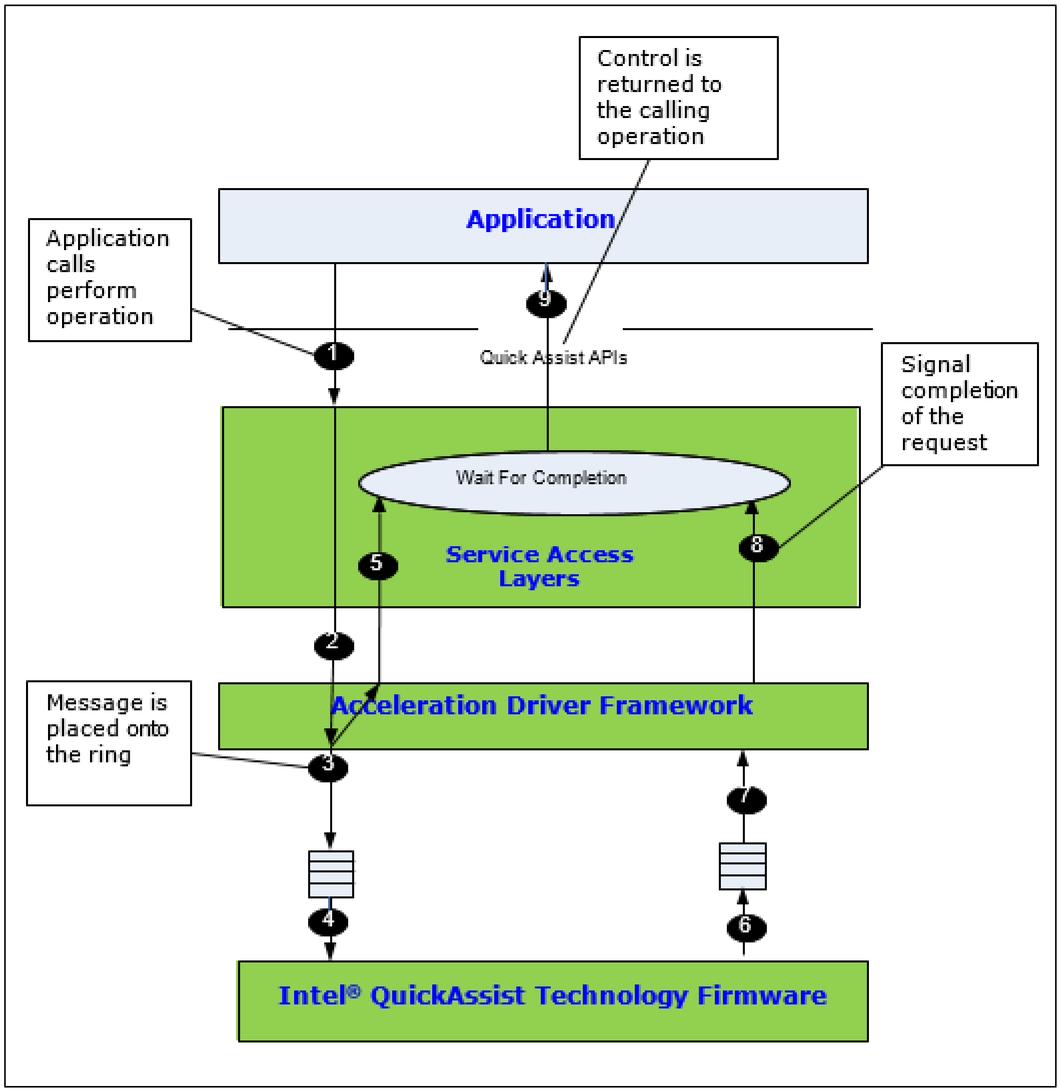

Control is immediately returned to the calling application
Application calls perform operation
Application
Some time later, the application callback is invoked.
Quick Assist APIs
1
5
8
Service Access Layers
2
Message is placed onto the ring
Acceleration Driver Framework
3
7
4
6
Intel® QuickAssist Technology Firmware

2.2.2.2 Synchronous Operation

Synchronous operation is specified by supplying a NULL function pointer in the callback parameter of the perform API as shown in Figure 4. In this case, the function does not return until the operation is complete. The calling thread may pend on a semaphore or other synchronization primitive after sending the request to the execution engine.

Upon the completion of the operation, the synchronization primitive will unblock, and execution will resume. Synchronous mode is therefore blocking and should not be used when invoking the function from a context in which sleeping is not allowed (for example, in interrupt context on Linux*).

Figure 4. Synchronous Operation


Control is returned to the calling operation
Application
Application calls perform operation
Quick Assist APIs
Signal completion of the request
Wait For Completion
Service Access Layers
Message is placed onto the ring
Acceleration Driver Framework
Intel® QuickAssist Technology Firmware
1
2
3
4
5
6
7
8
9

2.2.3 Memory Allocation and Ownership

The convention is that all memory needed by an API implementation is allocated outside of that implementation. In other words, the APIs are defined such that the memory needed to execute operations is supplied by a client entity or platform control entity rather than having memory allocated internally.

Memory used for parameters is owned by the side (caller or callee) that allocated the memory. An owner is responsible for de-allocating the memory when it is no longer needed.

Generally, memory ownership does not change. For example, if a program allocates memory and then passes a pointer to the memory as a parameter to a function call, the caller retains ownership and is still responsible for de-allocation of the memory. This is the default behavior and any function which deviates from this behavior will clearly state so in the function definition.

For optimal performance, data pointers should be 8-byte aligned. In some cases this is a requirement, while in most other cases, it is a recommendation for performance. Please refer to the service-specific API manual for optimal usage of the particular API.

2.2.4 Data Plane APIs

The Intel® QuickAssist Technology APIs for symmetric cryptography and for data compression supports both "traditional" APIs (i.e. `cpa_cy_sym.h` and `cpa_dc.h`) and "data plane" APIs (i.e. `cpa_cy_sym_dp.h` and `cpa_dc_dp.h`)[1]. The data plane APIs are recommended for applications running in a data plane environment where the cost of offload (i.e. the cycles consumed by the driver sending requests to the accelerator) needs to be minimized. To minimize the cost of offload, several constraints have been placed on these APIs. If these constraints are too restrictive for a given application the more general-purpose "traditional" APIs can be used (at an increased cost of offload).

The data plane APIs can be used if the following constraints are acceptable:

- There is no support for partial packets or stateful requests.
- Thread safety is not supported. Each software thread should have access to its own unique instance (CpaInstanceHandle).
- Only asynchronous invocation is supported.
- Polling is used, rather than interrupts, to dispatch callback functions. Callbacks will be invoked in the context of a polling thread. The user application is responsible for creating and scheduling this polling thread. Polling functions are not defined by the QuickAssist API. Implementations will provide their own polling functions. Please refer to implementation specific documentation (for example, the Intel® Communications Chipset 8900 to 8920 Series Software Programmer's Guide) for further information on polling functions.
- Buffers and buffer lists are passed using physical addresses, to avoid virtual to physical address translation costs.
- Alignment restrictions may be placed on the operation data (i.e. `CpaCySymDpOpData` and `CpaDcDpOpData`) and buffer list (i.e. `CpaPhysBufferList`) structures passed to the data plane APIs. For example, the operation data may need to be at least 8-byte aligned, contiguous, resident, DMA-accessible memory. Please refer to implementation specific documentation for more details.

____

1 Note there is no "data plane" support for asymmetric cryptography services.

- For CCM and GCM modes of AES, when performing decryption and verification, if verification fails, then the message buffer will not be zeroed.

The data plane APIs distinguish between enqueuing a request and submitting that request to the accelerator to be performed. This allows the cost of submitting a request (which can be expensive, in terms of cycles, for some hardware-based implementations) to be amortized over all enqueued requests on that instance (`CpaInstanceHandle`).

- To enqueue one request and to optionally submit all previously enqueued requests the function `cpaCySymDpEnqueueOp` (or `cpaDcDpEnqueueOp` for data compression service) can be used.
- To enqueue multiple requests and to optionally submit all previously enqueued requests the function `cpaCySymDpEnqueueOpBatch` (or `cpaDcDpEnqueueOpBatch` for data compression service) can be used.
- To submit all previously enqueued requests the function `cpaCySymDpPerformOpNow` (or `cpaDcDpPerformOpNow` for data compression service) can be used.

Different implementations of this API may have different performance trade-offs. Please refer to the documentation for your implementation for details.

3 Intel® QuickAssist Technology Cryptographic API

This chapter describes the sample code for the Intel® QuickAssist Technology Cryptographic API, beginning with an API overview, and followed by descriptions of various scenarios to illustrate usage of the API.

3.1 Overview

The Intel® QuickAssist Technology Cryptographic API can be categorized into the following broad areas:

- Common: This is defined by the file cpa_cy_common.h. This includes functionality for initialization and shutdown of the service.
- Instance Management: The file cpa_cy_im.h defines the functions for managing instances. A given implementation of the API can present multiple instances of the cryptographic service, each representing a logical or virtual "device". Request ordering is guaranteed within a given instance of the service.
- Symmetric: The following files constitute the symmetric API:
  - The cpa_cy_sym.h file contains the symmetric API, used for ciphers, hashing/message digests, "algorithm chaining" (combining cipher and hash into a single call) and authenticated ciphers.
  - The cpa_sy_sym_dp.h file also contains the symmetric API, used for ciphers, hashing/message digests, "algorithm chaining" (combining cipher and hash into a single call) and authenticated ciphers. This API is recommended for data plane applications, in which the cost of offload (i.e. the cycles consumed by the API in sending requests to the hardware and processing the responses) needs to be minimized. To use this API a number of constraints need to be acceptable to the application; these are listed in Section 2.2.4.
  - The cpa_cy_key.h file contains the API for key generation for SSL and TLS.
- Asymmetric: The following files constitute the asymmetric API:
  - The cpa_cy_rsa.h file defines the API for RSA.
  - The cpa_cy_dsa.h file defines the API for DSA.
  - The cpa_cy_dh.h file defines the API for Diffie-Hellman.

- The `cpa_cy_ec.h` file defines the API for "base" elliptic curve cryptography.
- The `cpa_cy_ecdsa.h` file defines the API for EC DSA.
- The `cpa_cy_ecdh.h` file defines the API for EC Diffie-Hellman.
- The `cpa_cy_prime.h` file defines the API for prime number testing.
- The `cpa_cy_ln.h` file defines the API for large number math (modular exponentiation, and so on).

- Random Bit Generation (RBG): The following files constitute the RBG API:
  - The `cpa_cy_drbg.h` file defines the API for deterministic random bit generation.
  - The `cpa_cy_nrbg.h` file defines the API for non-deterministic random bit generation.
  - The `cpa_cy_rand.h` file defines an API for random number generation. Note that this API has been deprecated, and is superseded by the DRBG and NRBG APIs defined in `cpa_cy_drbg.h` and `cpa_cy_nrbg.h`.

The Cryptographic API uses the base API (cpa), which defines base data types used across all services of the Intel® QuickAssist Technology API.

3.1.1 Sessions

The symmetric and DRBG APIs have the concept of sessions. The remaining APIs do not. The meaning of a session within these APIs is defined below.

3.1.2 Priority

The Cryptographic symmetric API has support for priorities. Priority can be specified on a per-session basis. Two levels of priority are supported: high priority and normal priority. Implementations may use a strict priority order, or a weighted round robin- based priority scheme.

3.2 Using the Symmetric Cryptography API

This section contains examples of how to use the symmetric API. It describes general concepts and how to use the symmetric API to perform various types of cipher and hash. Note these examples are simplified and demonstrate how to use the APIs and build the structures required for various use cases. These examples may not demonstrate the optimal way to use the API to get maximum performance for a particular implementation. Please refer to implementation specific documentation (for example, the Intel® Communications Chipset 8900 to 8920 Series Software Programmer's Guide) and performance sample code for a guide on how to use the API for best performance.

Note that all of the symmetric examples follow the same basic steps:

- Define a callback function (if the API is to be invoked asynchronously)
- Discover and start up the cryptographic service instance
- Create and initialize a session
- Invoke multiple symmetric operations (cipher and/or hash) on the session
- Tear down the session
- Stop the Cryptographic service instance

3.2.1 General Concepts

This section describes the following concepts:

- Session
- In-Place and Out-of-Place Support
- Partial Support

3.2.1.1 Session

In the case of the symmetric API, a session is a handle which describes the cryptographic parameters to be applied to a number of buffers. This might be the buffers within a single file, or all the packets associated with a particular IPSec tunnel or security association. The data within a session handle includes the following:

- The operation (cipher, hash or both, and if both, the order in which the algorithms should be applied).
- The cipher setup data, including the cipher algorithm and mode, the key and its length, and the direction (encrypt or decrypt).
- The hash setup data, including the hash algorithm, mode (plain, nested or authenticated), and digest result length (to allow for truncation).
  - Authenticated mode can refer to HMAC, which requires that the key and its length are also specified. It is also used for GCM and CCM authenticated encryption, in which case the AAD length is also specified.
  - For nested mode, the inner and outer prefix data and length are specified, as well as the outer hash algorithm.

3.2.1.2 In-Place and Out-of-Place Support

An In-Place operation means that the destination buffer is the same as the source buffer. An Out-of-Place operation means that the destination buffer is different from the source buffer.

3.2.1.3 Partial Support

Most of the examples in this chapter operate on full packets, as indicated by the packetType of `CPA_CY_SYM_PACKET_TYPE_FULL`. The API also supports operating in partial mode; where, for example, state (e.g. cipher state) needs to be carried forward from one packet/record to the next. In Section 3.2.4 there is an example of hashing a file that uses the partial API.

NOTES:

1. The size of the data to be hashed or ciphered must be a multiple of the block size of the algorithm for all partial packets.
2. For hash/authentication, the digest verify flag only applies to the last partial packet.
3. For algorithm chaining, only the cipher state is maintained between calls. The hash state is not maintained between calls; instead the hash digest will be generated/verified for each call. The size of the data to be ciphered must be a multiple of the block size of the algorithm for all partial packets. The size of the data to be hashed does not have this restriction. If both the cipher state and the hash state need to be maintained between calls, then algorithm chaining cannot be used.

3.2.2 Cipher

This example demonstrates the usage of the symmetric API, specifically using this API to perform a cipher operation. It encrypts some sample text using the 3DES algorithm in CBC mode.

These samples are located in

```
quickassist\lookaside\access_layer\src\sample_code\functional\sym
\cipher_sample
```

The following subsections describe the main functions in this file.

3.2.2.1 symCallback

In order to use the API in asynchronous mode, a callback function must be supplied. This function is called back (that is, invoked by the implementation of the API) when the asynchronous operation has completed. The context in which it is invoked depends on the implementation. For example, it could be invoked in the context of a Linux* interrupt handler's bottom half or in the context of a user created polling thread. The context in which this function is invoked places restrictions on what processing can be done in the callback function. On the API it states that this function should not sleep (since it may be called in a context which does not permit sleeping, for example, a Linux* bottom half).

This function can perform whatever processing is appropriate to the application. For example, it may free memory, continue processing of a decrypted packet, etc. In this example, the function only sets the complete variable to indicate it has been called, as illustrated below.

Listing 3. Callback Function

```
static void symCallback(void *pCallbackTag,
                        CpaStatus status,
                        const CpaCySymOp operationType,
                        void *pOpData,
                        CpaBufferList *pDstBuffer,
                        CpaBoolean verifyResult)
{
    PRINT_DBG("Callback called with status = %d.\n", status);

    if (NULL != pCallbackTag) {
        /* indicate that the function has been called */
        COMPLETE((struct COMPLETION_STRUCT *)pCallbackTag);
    }
}
```

3.2.2.2 cipherSample

This is the main entry point for the sample cipher code. It demonstrates the sequence of calls to be made to the API in order to create a session, perform one or more cipher operations, and then tear down the session. The following steps are performed:

- Call the instance discovery utility function, `sampleCyGetInstance`. This is a simplified version of instance discovery, in which exactly one instance of a crypto service is discovered. It does this by querying the API for all instances, and returning the first instance, as illustrated in Listing 4.

  This step is described in Section 2.2.1 but is repeated here for convenience.

Listing 4. Getting an Instance

```
#ifdef DO_CRYPTO
void
sampleCyGetInstance(CpaInstanceHandle* pCyInstHandle)
{
    CpaInstanceHandle cyInstHandles[MAX_INSTANCES];
    Cpa16U numInstances = 0;
    CpaStatus status = CPA_STATUS_SUCCESS;

    *pCyInstHandle = NULL;

    status = cpaCyGetNumInstances(&numInstances);
    if ((status == CPA_STATUS_SUCCESS) && (numInstances > 0))
    {
        status = cpaCyGetInstances(MAX_INSTANCES, cyInstHandles);
        if (status == CPA_STATUS_SUCCESS)
        {
            *pCyInstHandle = cyInstHandles[0];
        }
    }

    if (0 == numInstances)
    {
        PRINT_ERR("No instances found for 'SSL'\n");
        PRINT_ERR("Please check your section names in the config file.\n");
        PRINT_ERR("Also make sure to use config file version 2.\n");
    }

}
#endif
```

- Set the address translation function for the instance. This function will be used by the API to convert virtual addresses to physical addresses.

Listing 5. Set Address Translation Function

```
status = cpaCySetAddressTranslation(cyInstHandle, sampleVirtToPhys);
```

- Start the crypto service running as shown below.

Listing 6. Start up

```
status = cpaCyStartInstance(cyInstHandle);
```

- The next step is to create and initialize a session. First, populate the fields of the session initialization operational data structure. Note that the size required to store a session is implementation-dependent, so you must query the API first to determine how much memory to allocate, and then allocate that memory.

  One of two available queries can be used:

  cpaCySymSessionCtxGetSize(const CpaInstanceHandle instanceHandle_in,const CpaCySymSessionSetupData *pSessionSetupData, Cpa32U *pSessionCtxSizeInBytes)

  This will always return the maximum session context size (i.e., the full size of the session including padding and other session state information) (see Listing 7a below).

  cpaCySymSessionCtxGetDynamicSize(const CpaInstanceHandle instanceHandle_in, const CpaCySymSessionSetupData *pSessionSetupData, Cpa32U *pSessionCtxSizeInBytes)

  This query can be used instead to return a reduced memory size, based on whether the use case meets certain session setup criteria (see Listing 7b below).

  This query will return one of three values for **pSessionCtxSizeInBytes** as follows:

  - If partial packets are not being used and the Symmetric operation is Auth- Encrypt (i.e., the cipher and hash algorithms are either CCM or GCM), the size returned will be approximately half of the standard size.
  - If partial packets are not being used and the cipher algorithm is not ARC4, Snow3g_UEA2, AES_CCM or AES_GCM, and the hash algorithm is not Snow3G_UIA2, AES_CCM or AES_GCM, and Hash Mode is not Auth, the size returned will be between half and one third of the standard size.
  - In all other cases, the standard size is returned.

Note: The following parameter exists in the `CpaCySymSessionSetupData` structure:

CpaBoolean partialsNotRequired

This flag indicates if partial packet processing is required for the session. If partial packets are not being used and the preference is to use one of the reduced session memory sizes, set this flag to CPA_TRUE before calling the **cpaCySymSessionCtxGetDynamicSize()** function.

Note: The equivalent reduced memory context query for Data Plane API (see Section 3.2.10, Chained Cipher & Hash using the Symmetric Data Plane) is:

cpaCySymDpSessionCtxGetDynamicSize(const CpaInstanceHandle instanceHandle_in, const CpaCySymSessionSetupData *pSessionSetupData, Cpa32U *pSessionCtxSizeInBytes)

Listing 7. Create and Initialize Cipher Session

```
/* Populate the session setup structure for the operation required */
sessionSetupData.sessionPriority = CPA_CY_PRIORITY_NORMAL;
sessionSetupData.symOperation = CPA_CY_SYM_OP_CIPHER;
sessionSetupData.cipherSetupData.cipherAlgorithm = CPA_CY_SYM_CIPHER_3DES_CBC;
sessionSetupData.cipherSetupData.pCipherKey = sampleCipherKey;
sessionSetupData.cipherSetupData.cipherKeyLenInBytes = sizeof(sampleCipherKey);
sessionSetupData.cipherSetupData.cipherDirection =
    CPA_CY_SYM_CIPHER_DIRECTION_ENCRYPT;

/* Determine size of session context to allocate */
PRINT_DBG("cpaCySymSessionCtxGetSize\n");
status =
    cpaCySymSessionCtxGetSize(cyInstHandle, &sessionSetupData, &sessionCtxSize);
}

if (CPA_STATUS_SUCCESS == status) {
    /* Allocate session context */
    status = PHYS_CONTIG_ALLOC(&sessionCtx, sessionCtxSize);
}

/* Initialize the Cipher session */
if (CPA_STATUS_SUCCESS == status) {
    PRINT_DBG("cpaCySymInitSession\n");
    status = cpaCySymInitSession(cyInstHandle,
                                 symCallback,       /* callback function */
                                 &sessionSetupData, /* session setup data */
                                 sessionCtx);       /* output of the function*/
}
```

- Call the function `cipherPerformOp`, which actually performs the cipher operation. This in turn performs the following steps:
  - Memory Allocation: Different implementations of the API require different amounts of space to store metadata associated with buffer lists. Query the API to find out how much space the current implementation needs, and then allocate space for the buffer metadata, the buffer list, and for the buffer itself. You must also allocate memory for the initialization vector.

Listing 8. Memory Allocation

```
status = cpaCyBufferListGetMetaSize(cyInstHandle, numBuffers, &bufferMetaSize);

if (CPA_STATUS_SUCCESS == status) {
    status = PHYS_CONTIG_ALLOC(&pBufferMeta, bufferMetaSize);
}

if (CPA_STATUS_SUCCESS == status) {
    status = OS_MALLOC(&pBufferList, bufferListMemSize);
}

if (CPA_STATUS_SUCCESS == status) {
    status = PHYS_CONTIG_ALLOC(&pSrcBuffer, bufferSize);
}

if (CPA_STATUS_SUCCESS == status) {
    status = PHYS_CONTIG_ALLOC(&pIvBuffer, sizeof(sampleCipherIv));
}
```

  - The memory for the source buffer and initialization vector is populated with the required data.
  - Set up Operational Data: Populate the structure containing the operational data that is needed to run the algorithm as shown below.

Listing 9. Set up Cipher Operational Data

```
pOpData->sessionCtx = sessionCtx;
pOpData->packetType = CPA_CY_SYM_PACKET_TYPE_FULL;
pOpData->pIv = pIvBuffer;
pOpData->ivLenInBytes = sizeof(sampleCipherIv);
pOpData->cryptoStartSrcOffsetInBytes = 0;
pOpData->messageLenToCipherInBytes = sizeof(sampleCipherSrc);
```

– Perform Operation: Initialize the completion variable which is used by the callback function to indicate that the operation is complete, then perform the operation.

Listing 10. Perform Cipher Operation

```
COMPLETION_INIT(&complete);

status = cpaCySymPerformOp(
    cyInstHandle,
    (void *)&complete, /* data sent as is to the callback function*/
    pOpData,           /* operational data struct */
    pBufferList,       /* source buffer list */
    pBufferList,       /* same src & dst for an in-place operation*/
    NULL);
```

– Wait for completion: Because the asynchronous API is used in this example, the callback function must be handled. This example uses a macro which can be defined differently for different operating systems. In a typical real-world application, the calling thread would not block, and the callback would essentially re-inject the (decrypted, decapsulated) packet into the stack.

Listing 11. Wait for Completion

```
if (!COMPLETION_WAIT(&complete, TIMEOUT_MS)) {
    PRINT_ERR("timeout or interruption in cpaCySymPerformOp\n");
    status = CPA_STATUS_FAIL;
}
```

- In a normal usage scenario, the session would be reused multiple times to encrypt multiple buffers or packets. In this example, however, the session is torn down.

Listing 12. Remove Cipher Session

```
sessionStatus = cpaCySymRemoveSession(cyInstHandle, sessionCtx);
```

- You can query statistics at this point, which can be useful for debugging. Note that some implementations may also make the statistics available through other mechanisms, such as the `/proc` virtual filesystem.
- Finally, clean up by freeing up memory, stopping the instance, etc.

3.2.3 Hash

This example demonstrates the usage of the symmetric API, specifically using this API to perform a hash operation. It performs an MD5 hash operation on some sample data.

These samples are located in `\sym\hash_sample`

The example is very similar to the cipher example, so only the differences are highlighted:

- When creating and initializing a session, some of the fields of the session initialization operational data structure are different from the cipher case, as shown below.

Listing 13. Create and Initialize Hash Session

```
/* populate symmetric session data structure
 * for a plain hash operation */
sessionSetupData.sessionPriority = CPA_CY_PRIORITY_NORMAL;
sessionSetupData.symOperation = CPA_CY_SYM_OP_HASH;
sessionSetupData.hashSetupData.hashAlgorithm = CPA_CY_SYM_HASH_MD5;
sessionSetupData.hashSetupData.hashMode = CPA_CY_SYM_HASH_MODE_PLAIN;
sessionSetupData.hashSetupData.digestResultLenInBytes = DIGEST_LENGTH;
/* Place the digest result in a buffer unrelated to srcBuffer */
sessionSetupData.digestIsAppended = CPA_FALSE;
/* Generate the digest */
sessionSetupData.verifyDigest = CPA_FALSE;
```

- When calling the function to perform the hash operation, some of the fields of the operational data structure are again different from the cipher case, as shown below:

Listing 14. Set up Hash Operational Data

```
pOpData->sessionCtx = sessionCtx;
pOpData->packetType = CPA_CY_SYM_PACKET_TYPE_FULL;
pOpData->hashStartSrcOffsetInBytes = 0;
pOpData->messageLenToHashInBytes = sizeof(vectorData);
pOpData->pDigestResult = pDigestBuffer;
```

3.2.4 Hash a File

This example demonstrates the usage of the symmetric API for partial mode, specifically using this API to perform hash operations. It performs a SHA1 hash operation on a file.

These samples are located in `\sym\hash_file_sample`

The example is very similar to the cipher example, so only the differences are highlighted:

- When creating and initializing a session, some of the fields of the session initialization operational data structure are different from the cipher case, as shown below.

Listing 15. Hash Session Setup Data

```
/* populate symmetric session data structure
 * for a plain hash operation */
sessionSetupData.sessionPriority = CPA_CY_PRIORITY_NORMAL;
sessionSetupData.symOperation = CPA_CY_SYM_OP_HASH;
sessionSetupData.hashSetupData.hashAlgorithm = CPA_CY_SYM_HASH_SHA1;
sessionSetupData.hashSetupData.hashMode = CPA_CY_SYM_HASH_MODE_PLAIN;
sessionSetupData.hashSetupData.digestResultLenInBytes = DIGEST_LENGTH;
/* Place the digest result in a buffer unrelated to srcBuffer */
sessionSetupData.digestIsAppended = CPA_FALSE;
/* Generate the digest */
sessionSetupData.verifyDigest = CPA_FALSE;
```

- Memory is allocated for the source buffer in a similar way to the cipher case.
- To perform the operation data is read from the file to the source buffer and the symmetric API is called repeatedly with `packetType` set to `CPA_CY_SYM_PACKET_TYPE_PARTIAL`. When the end of the file is reached the API is called with `packetType` set to `CPA_CY_SYM_PACKET_TYPE_PARTIAL_LAST`. The digest is produced only on the last call to the API.

Listing 16. Hashing a File

```
while (!feof(srcFile)) {
    /* read from file into src buffer */
    pBufferList->pBuffers->dataLenInBytes =
        fread(pSrcBuffer, 1, SAMPLE_BUFF_SIZE, srcFile);

    /* If we have reached the end of file set the last partial flag */
    if (feof(srcFile)) {
        pOpData->packetType = CPA_CY_SYM_PACKET_TYPE_LAST_PARTIAL;
} else {
        pOpData->packetType = CPA_CY_SYM_PACKET_TYPE_PARTIAL;
    }
    pOpData->sessionCtx = sessionCtx;
    pOpData->hashStartSrcOffsetInBytes = 0;
    pOpData->messageLenToHashInBytes = pBufferList->pBuffers->dataLenInBytes;
    pOpData->pDigestResult = pDigestBuffer;

    PRINT_DBG("cpaCySymPerformOp\n");
    /** Perform symmetric operation */
    status = cpaCySymPerformOp(
        cyInstHandle,
        (void *)&complete, /* data sent as is to the callback function*/
        pOpData,           /* operational data struct */
        pBufferList,       /* source buffer list */
        pBufferList,       /* same src & dst for an in-place operation*/
        NULL);

    if (CPA_STATUS_SUCCESS != status) {
        PRINT_ERR("cpaCySymPerformOp failed. (status = %d)\n", status);
        break;
    }

    if (CPA_STATUS_SUCCESS == status) {
        /** wait until the completion of the operation*/
        if (!COMPLETION_WAIT((&complete), TIMEOUT_MS)) {
            PRINT_ERR("timeout or interruption in cpaCySymPerformOp\n");
            status = CPA_STATUS_FAIL;
            break;
        }
    }
}
```

3.2.5 Chained Cipher & Hash

This example demonstrates the usage of the symmetric API, specifically using this API to perform a "chained" cipher and hash operation. It encrypts some sample text using the 3DES algorithm in CBC mode, and then performs an MD5 HMAC operation on the ciphertext, writing the MAC to the buffer immediately after the ciphertext.

These samples are located in `\sym\alg_chaining_sample`

The example is very similar to the cipher and hash examples, above, so only the differences are highlighted:

- When creating and initializing a session, some of the fields of the session initialization operational data structure are different, as shown below.

Listing 17. Create and Initialize Session Cipher & Hash

```
sessionSetupData.symOperation = CPA_CY_SYM_OP_ALGORITHM_CHAINING;
sessionSetupData.algChainOrder = CPA_CY_SYM_ALG_CHAIN_ORDER_CIPHER_THEN_HASH;

sessionSetupData.cipherSetupData.cipherAlgorithm = CPA_CY_SYM_CIPHER_3DES_CBC;
sessionSetupData.cipherSetupData.pCipherKey = sampleCipherKey;
sessionSetupData.cipherSetupData.cipherKeyLenInBytes = sizeof(sampleCipherKey);
sessionSetupData.cipherSetupData.cipherDirection =
    CPA_CY_SYM_CIPHER_DIRECTION_ENCRYPT;

sessionSetupData.hashSetupData.hashAlgorithm = CPA_CY_SYM_HASH_MD5;
sessionSetupData.hashSetupData.hashMode = CPA_CY_SYM_HASH_MODE_AUTH;
sessionSetupData.hashSetupData.digestResultLenInBytes = DIGEST_LENGTH;
sessionSetupData.hashSetupData.authModeSetupData.authKey = sampleCipherKey;
sessionSetupData.hashSetupData.authModeSetupData.authKeyLenInBytes =
    sizeof(sampleCipherKey);

/* The resulting MAC is to be placed immediately after the ciphertext */
sessionSetupData.digestIsAppended = CPA_TRUE;
sessionSetupData.verifyDigest = CPA_FALSE;
```

When calling the function to perform the chained cipher & hash operation, some of the fields of the operational data structure are again different from the cipher case, as shown below:

Listing 18. Set up Operational Data Cipher & Hash

```
/** Populate the structure containing the operational data that is
 * needed to run the algorithm
 */
pOpData->sessionCtx = sessionCtx;
pOpData->packetType = CPA_CY_SYM_PACKET_TYPE_FULL;
pOpData->pIv = pIvBuffer;
pOpData->ivLenInBytes = sizeof(sampleCipherIv);
pOpData->cryptoStartSrcOffsetInBytes = 0;
pOpData->hashStartSrcOffsetInBytes = 0;
pOpData->messageLenToCipherInBytes = sizeof(sampleAlgChainingSrc);
pOpData->messageLenToHashInBytes = sizeof(sampleAlgChainingSrc);
```

- Notice digestIsAppended is set in the session therefore the MAC will be placed immediately after the region to hash and the pDigestResult parameter of the operational data is ignored.

3.2.6 Chained Cipher & Hash – IPSec like use case

This example demonstrates the usage of the symmetric API for IPSec-like use cases as described in Figure 5 and Figure 6. For the outbound direction, this example will use the symmetric API to perform a “chained” cipher and hash operation. It encrypts some plaintext using the AES algorithm in CBC mode, and then performs a SHA1 HMAC operation on the ciphertext, initialization vector and header, writing the ICV to the buffer immediately after the ciphertext. For the inbound direction, this example will again use the symmetric API to perform a “chained” hash and cipher operation. It performs a SHA1 HMAC operation on the ciphertext, initialization vector and header and compares the result with the input ICV. Then it decrypts the ciphertext using the AES algorithm in CBC mode.

Figure 5. IPSec Outbound

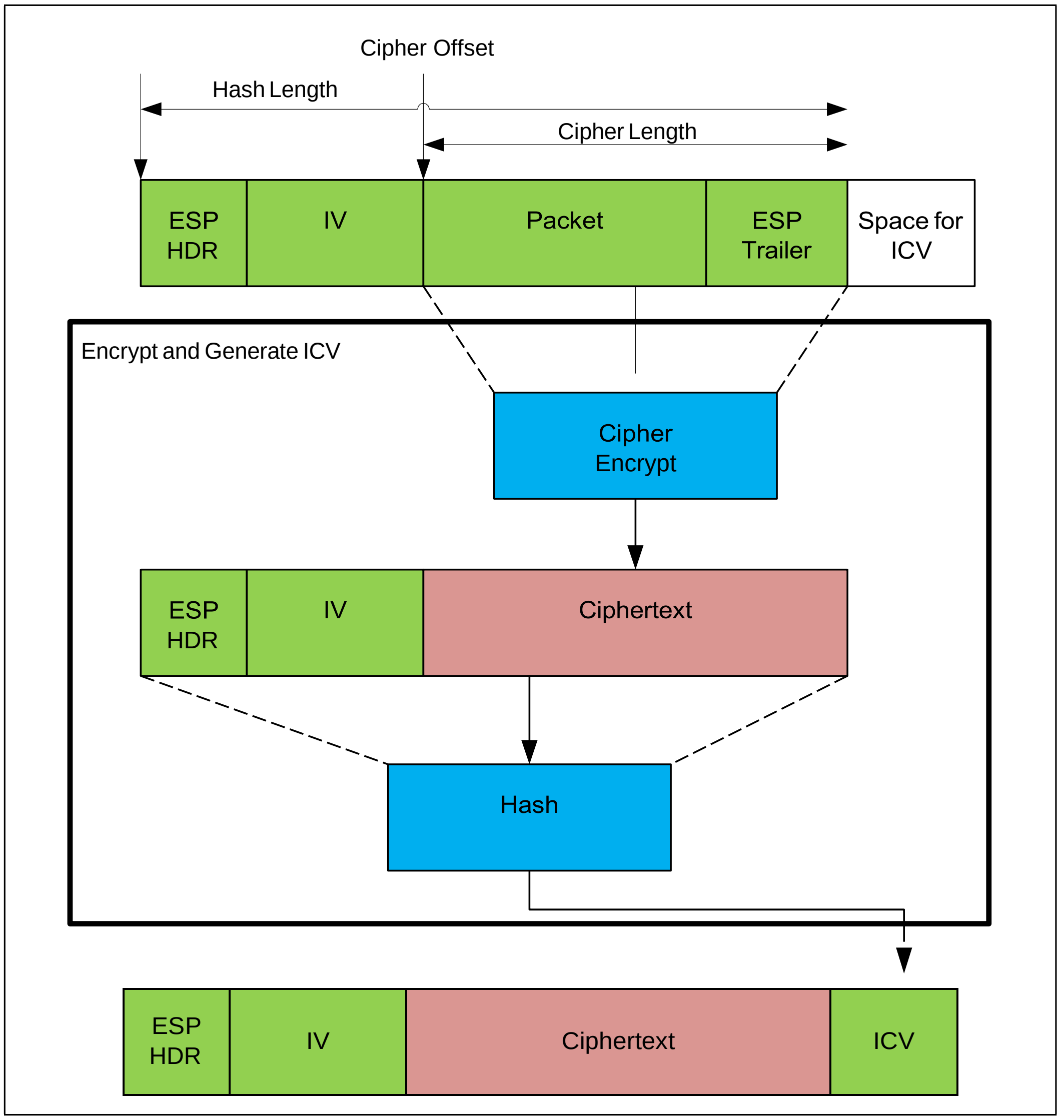
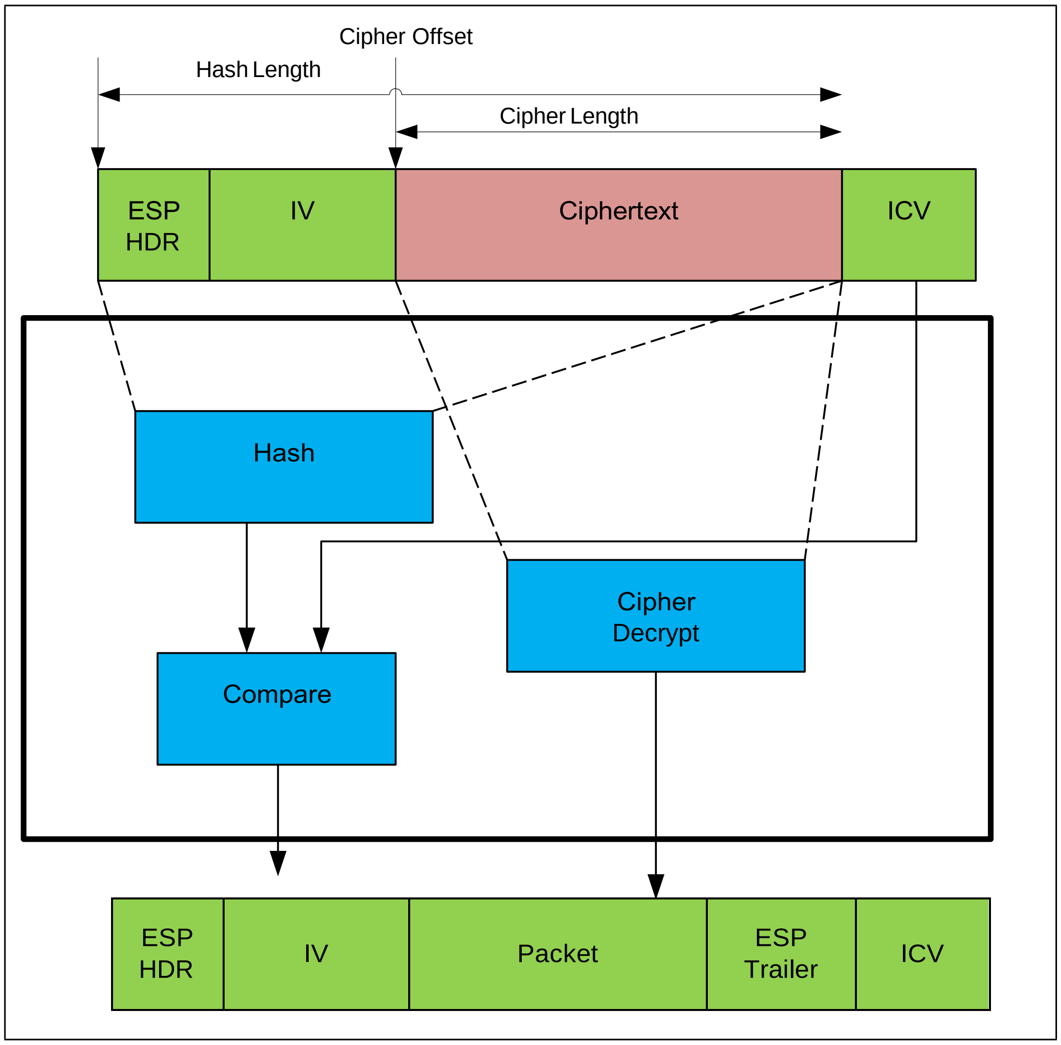


Figure 6. IPSec Inbound



These samples are located in `\sym\ipsec_sample`

Again, only the differences compared to previous examples are highlighted:

- When creating and initializing a session in the outbound direction, the session initialization operational data structure is shown below.

Listing 19. Session Setup Data IPSec Outbound

```
sessionSetupData.symOperation = CPA_CY_SYM_OP_ALGORITHM_CHAINING;
sessionSetupData.algChainOrder = CPA_CY_SYM_ALG_CHAIN_ORDER_CIPHER_THEN_HASH;

sessionSetupData.cipherSetupData.cipherAlgorithm = CPA_CY_SYM_CIPHER_AES_CBC;
sessionSetupData.cipherSetupData.pCipherKey = sampleCipherKey;
sessionSetupData.cipherSetupData.cipherKeyLenInBytes = sizeof(sampleCipherKey);
sessionSetupData.cipherSetupData.cipherDirection =
    CPA_CY_SYM_CIPHER_DIRECTION_ENCRYPT;

sessionSetupData.hashSetupData.hashAlgorithm = CPA_CY_SYM_HASH_SHA1;
sessionSetupData.hashSetupData.hashMode = CPA_CY_SYM_HASH_MODE_AUTH;
sessionSetupData.hashSetupData.digestResultLenInBytes = ICV_LENGTH;
sessionSetupData.hashSetupData.authModeSetupData.authKey = sampleAuthKey;
sessionSetupData.hashSetupData.authModeSetupData.authKeyLenInBytes =
    sizeof(sampleAuthKey);

/* Even though ICV follows immediately after the region to hash
   digestIsAppended is set to false in this case to workaround
   errata number IXA00378322 */
sessionSetupData.digestIsAppended = CPA_FALSE;
/* Generate the ICV in outbound direction */
sessionSetupData.verifyDigest = CPA_FALSE;
```

- When calling the function to perform the chained cipher & hash operation, the fields of the operational data structure are shown below:

Listing 20. Operational Data IPSec outbound

```
/** Populate the structure containing the operational data that is
 * needed to run the algorithm in outbound direction */
pOpData->sessionCtx = sessionCtx;
pOpData->packetType = CPA_CY_SYM_PACKET_TYPE_FULL;
pOpData->pIv = pIvBuffer;
pOpData->ivLenInBytes = sizeof(sampleCipherIv);
pOpData->cryptoStartSrcOffsetInBytes =
    sizeof(sampleEspHdrData) + sizeof(sampleCipherIv);
pOpData->messageLenToCipherInBytes = sizeof(samplePayload);
pOpData->hashStartSrcOffsetInBytes = 0;
pOpData->messageLenToHashInBytes =
    sizeof(sampleEspHdrData) + sizeof(sampleCipherIv) + sizeof(samplePayload);
/* Even though ICV follows immediately after the region to hash
digestIsAppended is set to false in this case to workaround
errata number IXA00378322 */
pOpData->pDigestResult =
    pSrcBuffer +
    (sizeof(sampleEspHdrData) + sizeof(sampleCipherIv) + sizeof(samplePayload));
```

- Note in this example `samplePayload` is the packet data plus the ESP trailer.

- When creating and initializing a session in the inbound direction, the session initialization operational data structure is shown below.

Listing 21. Session Setup Data IPSec Inbound

```
sessionSetupData.symOperation = CPA_CY_SYM_OP_ALGORITHM_CHAINING;
sessionSetupData.algChainOrder = CPA_CY_SYM_ALG_CHAIN_ORDER_HASH_THEN_CIPHER;

sessionSetupData.cipherSetupData.cipherAlgorithm = CPA_CY_SYM_CIPHER_AES_CBC;
sessionSetupData.cipherSetupData.pCipherKey = sampleCipherKey;
sessionSetupData.cipherSetupData.cipherKeyLenInBytes = sizeof(sampleCipherKey);
sessionSetupData.cipherSetupData.cipherDirection =
    CPA_CY_SYM_CIPHER_DIRECTION_DECRYPT;

sessionSetupData.hashSetupData.hashAlgorithm = CPA_CY_SYM_HASH_SHA1;
sessionSetupData.hashSetupData.hashMode = CPA_CY_SYM_HASH_MODE_AUTH;
sessionSetupData.hashSetupData.digestResultLenInBytes = ICV_LENGTH;
sessionSetupData.hashSetupData.authModeSetupData.authKey = sampleAuthKey;
sessionSetupData.hashSetupData.authModeSetupData.authKeyLenInBytes =
    sizeof(sampleAuthKey);

/* ICV follows immediately after the region to hash */
sessionSetupData.digestIsAppended = CPA_TRUE;
/* Verify the ICV in the inbound direction */
sessionSetupData.verifyDigest = CPA_TRUE;
```

- When calling the function to perform the chained hash & cipher operation, the fields of the operational data structure are:

Listing 22. Operational Data IPSec Inbound

```
/** Populate the structure containing the operational data that is
 * needed to run the algorithm in inbound direction */
pOpData->sessionCtx = sessionCtx;
pOpData->packetType = CPA_CY_SYM_PACKET_TYPE_FULL;
pOpData->pIv = pIvBuffer;
pOpData->ivLenInBytes = sizeof(sampleCipherIv);
pOpData->cryptoStartSrcOffsetInBytes =
    sizeof(sampleEspHdrData) + sizeof(sampleCipherIv);
pOpData->messageLenToCipherInBytes =
    bufferSize -
    (sizeof(sampleEspHdrData) + sizeof(sampleCipherIv) + ICV_LENGTH);
pOpData->hashStartSrcOffsetInBytes = 0;
pOpData->messageLenToHashInBytes = bufferSize - ICV_LENGTH;
```

- Note in this example `bufferSize` is the size of the data input (header, iv, ciphertext and ICV).

3.2.7 Chained Cipher & Hash – SSL like use case

This example demonstrates the usage of the symmetric API for SSL-like use cases as described in Figure 7 and Figure 8. For the outbound direction, this example will use the symmetric API to perform a “chained” hash and cipher operation. It performs a SHA1 HMAC[2] operation on a sequence number, part of the header and the plaintext. The resultant MAC is placed immediately after the plaintext. Then it encrypts the plaintext, MAC and padding using the AES algorithm in CBC mode.

For the inbound direction, this example will again use the symmetric API to perform a “chained” cipher and hash operation. It decrypts the ciphertext using the AES algorithm in CBC mode. Then it performs a SHA1 HMAC operation on the resultant plaintext, sequence number and part of the header and compares the result with the input MAC. Note for the inbound direction to use the “chained” API the length of the plaintext needs to be known before the ciphertext is decrypted to correctly set `messageLenToHashInBytes` and the length field in the header. For stream ciphers (e.g. ARC4) there is no padding added in the outbound direction so the length of the plaintext is simply the length of the ciphertext minus the length of the MAC. However, for block ciphers in CBC mode (as used in this example) the padlen is required to calculate the plaintext length. The final block of the ciphertext needs to be decrypted to discover the padlen. In this example before calling the “chained” API the final block of the ciphertext is decrypted to discover the padlen.

---

2. Note not all SSL use cases use HMAC. For example, SSLv3 (RFC 6106) does not use HMAC (in this case the nested hash functionality on the API can be used). However, TLSv1.2 (RFC 5246), for example, does use the HMAC algorithm.

Figure 7. SSL Outbound

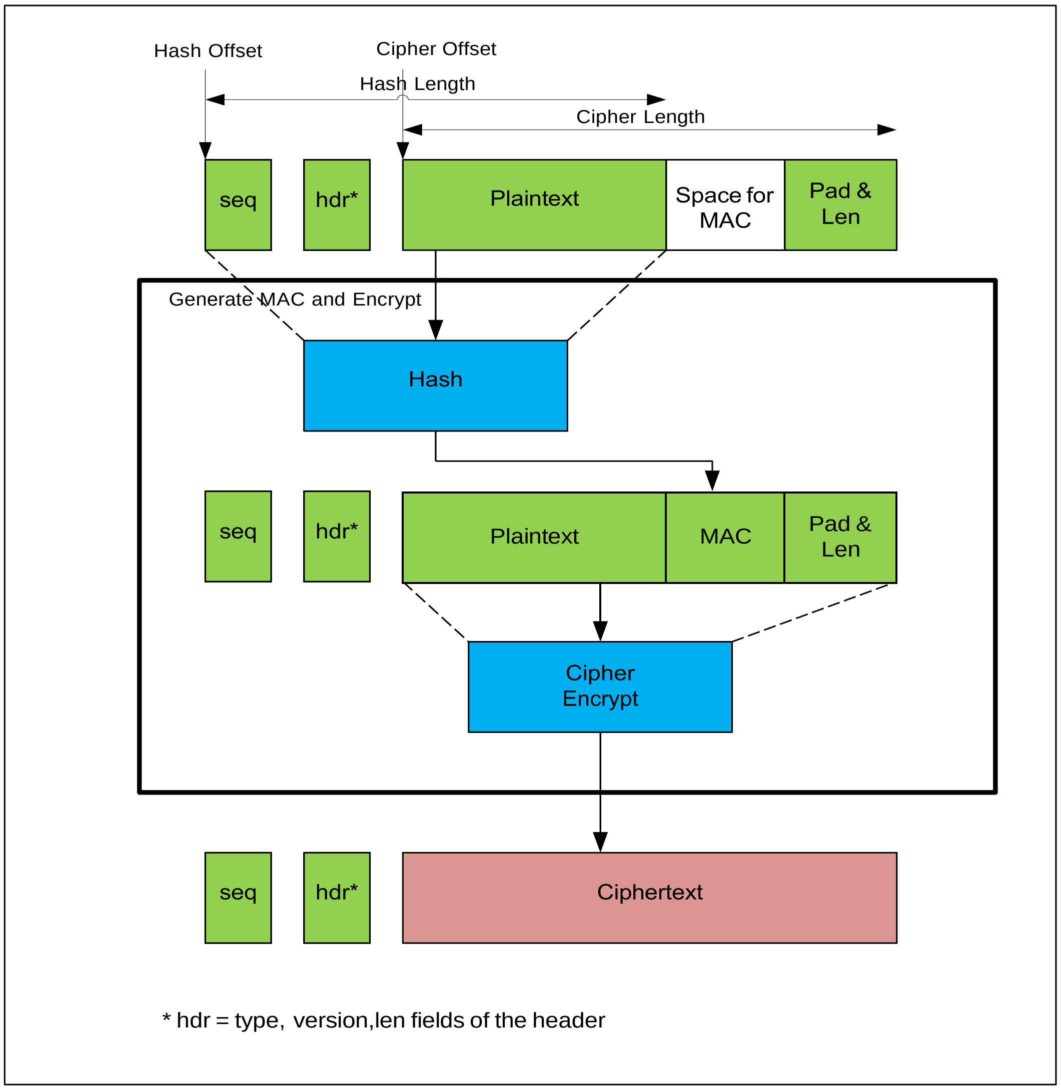
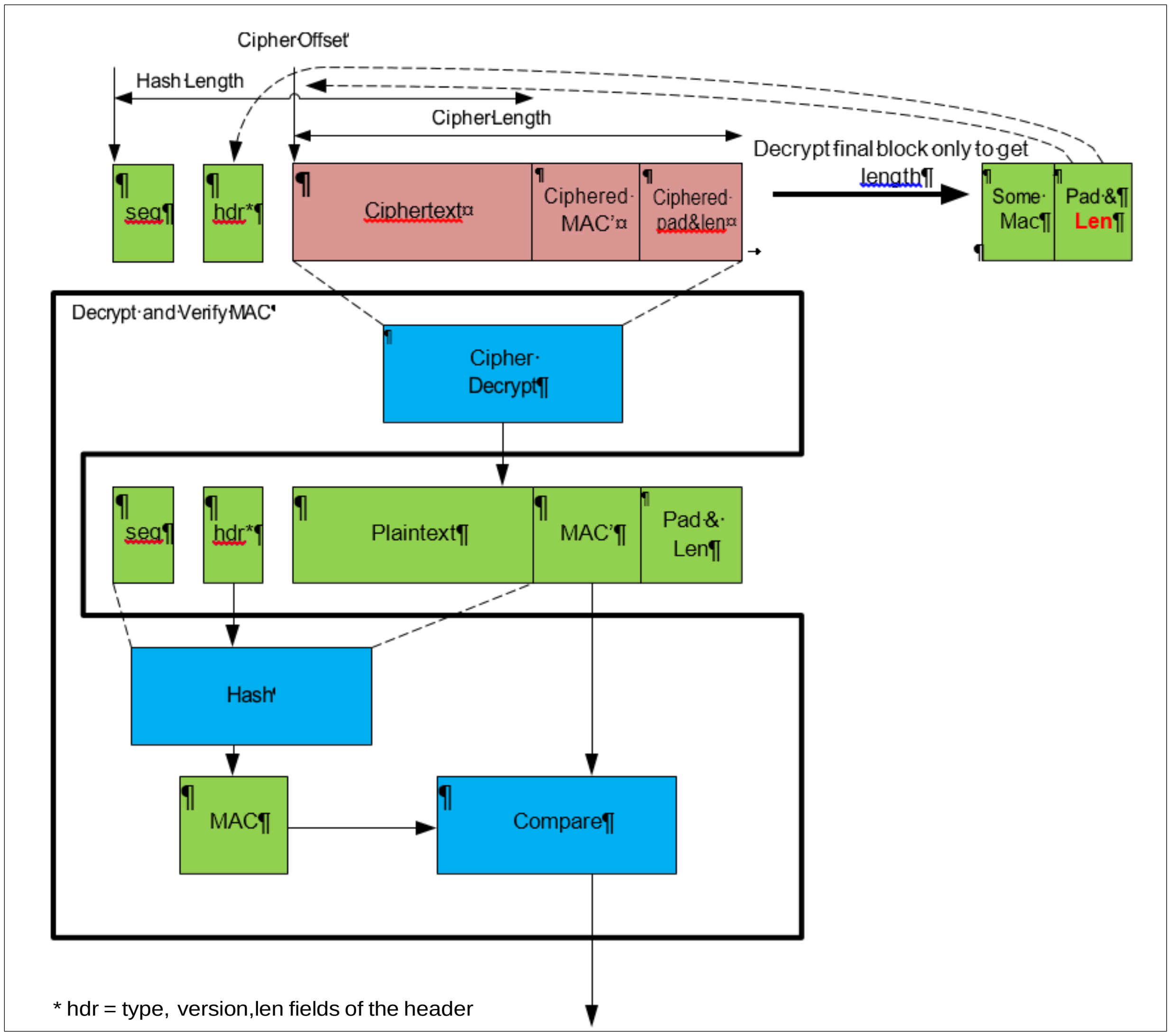


Figure 8. SSL Inbound



If using a block cipher in CBC mode then the last ciphertext block is used as the IV for subsequent packets (or records) in SSL and TLSv1.0 whereas in TLSv1.1 and 1.2 an explicit IV is used. However, if using a stream cipher that does not use a synchronization vector (such as ARC4) the stream cipher state from the end of one packet is used to process the subsequent packets. If using the QA API in this case then partial mode should be used to ensure the stream cipher state is maintained across multiple calls to the API.

Again these examples are very similar to previous examples so only the differences are highlighted:

- When creating and initializing a session in the outbound direction, the session setup data structure is shown below.

Listing 23. Session Data SSL outbound

```
sessionSetupData.symOperation = CPA_CY_SYM_OP_ALGORITHM_CHAINING;
sessionSetupData.algChainOrder = CPA_CY_SYM_ALG_CHAIN_ORDER_HASH_THEN_CIPHER;

sessionSetupData.cipherSetupData.cipherAlgorithm = CPA_CY_SYM_CIPHER_AES_CBC;
sessionSetupData.cipherSetupData.pCipherKey = sampleCipherKey;
sessionSetupData.cipherSetupData.cipherKeyLenInBytes = sizeof(sampleCipherKey);
sessionSetupData.cipherSetupData.cipherDirection =
    CPA_CY_SYM_CIPHER_DIRECTION_ENCRYPT;

sessionSetupData.hashSetupData.hashAlgorithm = CPA_CY_SYM_HASH_SHA1;
sessionSetupData.hashSetupData.hashMode = CPA_CY_SYM_HASH_MODE_AUTH;
sessionSetupData.hashSetupData.digestResultLenInBytes = MAC_LENGTH;
sessionSetupData.hashSetupData.authModeSetupData.authKey = sampleAuthKey;
sessionSetupData.hashSetupData.authModeSetupData.authKeyLenInBytes =
    sizeof(sampleAuthKey);

/* MAC follows immediately after the region to hash */
sessionSetupData.digestIsAppended = CPA_TRUE;
/* Generate the MAC in outbound direction */
sessionSetupData.verifyDigest = CPA_FALSE;
```

- A buffer large enough to hold the plaintext, MAC and padding is required. The size of this buffer will be:

Listing 24. Buffer Size SSL outbound

```
bufferSize = sizeof(samplePayload) + MAC_LENGTH;

/* bufferSize needs to be rounded up to a multiple of
   the AES block size */
padLen = 16 - bufferSize % 16;
bufferSize += padLen;
/* padLen excludes pad_length field */
padLen--;
```

- This buffer is filled with plaintext and padding leaving room for the “chained” API operation to add the MAC:

Listing 25. Buffer Setup SSL outbound

```
memcpy(pSrcBuffer, samplePayload, sizeof(samplePayload));
/* Leave space for MAC but insert padding data */
for (i = 0; i <= padLen; i++) {
    pSrcBuffer[(sizeof(samplePayload) + MAC_LENGTH + i)] = padLen;
}
```

- The session sequence number, the header and the buffer with the plaintext are described using a `CpaBufferList`:

Listing 26. BufferList Setup SSL outbound

```
pBufferList->pBuffers = pFlatBuffer;
pBufferList->numBuffers = numBuffers;
pBufferList->pPrivateMetaData = pBufferMeta;

/* Seq number */
pFlatBuffer->dataLenInBytes = SSL_CombinedHeadSize;
pFlatBuffer->pData = pCombinedHeadBuffer;
pFlatBuffer++;
memcpy((char *)pCombinedHeadBuffer + SESSION_SEQ_START,
       &sessSeqNum,
       sizeof(sessSeqNum));
memcpy((char *)pCombinedHeadBuffer + HDR_START,
       sampleHdrData,
       sizeof(sampleHdrData));

/* Data */
pFlatBuffer->dataLenInBytes = bufferSize;
pFlatBuffer->pData = pSrcBuffer;
```

- When calling the function to perform the chained hash & cipher operation, the fields of the operational data structure are:

Listing 27. Operational Data SSL outbound

```
pOpData->sessionCtx = sessionCtx;
pOpData->packetType = CPA_CY_SYM_PACKET_TYPE_FULL;
pOpData->pIv = pIvBuffer;
pOpData->ivLenInBytes = sizeof(sampleCipherIv);
pOpData->cryptoStartSrcOffsetInBytes = SSL_CombinedHeadSize;
pOpData->messageLenToCipherInBytes = bufferSize;
pOpData->hashStartSrcOffsetInBytes = SESSION_SEQ_START;
pOpData->messageLenToHashInBytes = sizeof(sessSeqNum) + sizeof(sampleHdrData) +
                                   bufferSize - MAC_LENGTH - padLen;
```

- When creating and initializing a session in the inbound direction, the session setup data structure is shown below.

Listing 28. Session Data SSL inbound

```
sessionSetupData.symOperation = CPA_CY_SYM_OP_ALGORITHM_CHAINING;
sessionSetupData.algChainOrder = CPA_CY_SYM_ALG_CHAIN_ORDER_HASH_THEN_CIPHER;

sessionSetupData.cipherSetupData.cipherAlgorithm = CPA_CY_SYM_CIPHER_AES_CBC;
sessionSetupData.cipherSetupData.pCipherKey = sampleCipherKey;
sessionSetupData.cipherSetupData.cipherKeyLenInBytes = sizeof(sampleCipherKey);
sessionSetupData.cipherSetupData.cipherDirection =
    CPA_CY_SYM_CIPHER_DIRECTION_ENCRYPT;

sessionSetupData.hashSetupData.hashAlgorithm = CPA_CY_SYM_HASH_SHA1;
sessionSetupData.hashSetupData.hashMode = CPA_CY_SYM_HASH_MODE_AUTH;
sessionSetupData.hashSetupData.digestResultLenInBytes = MAC_LENGTH;
sessionSetupData.hashSetupData.authModeSetupData.authKey = sampleAuthKey;
sessionSetupData.hashSetupData.authModeSetupData.authKeyLenInBytes =
    sizeof(sampleAuthKey);

/* MAC follows immediately after the region to hash */
sessionSetupData.digestIsAppended = CPA_TRUE;
/* Generate the MAC in outbound direction */
sessionSetupData.verifyDigest = CPA_FALSE;
```

- In this case the length of the ciphertext is `bufferSize` to calculate the padLen the final block is decrypted.

Listing 29. Calculating padLen SSL inbound

```
Cpa8U resBuff[16];

/* For decrypt direction need to decrypt the final block
   to determine the messageLenToHashInBytes */
status = sampleCodeAesCbcDecrypt(sampleCipherKey,
                                 sizeof(sampleCipherKey),
                                 (pSrcBuffer + (bufferSize - 32)), /* IV */
                                 (pSrcBuffer + (bufferSize - 16)), /* src */
                                 resBuff);                         /* dest */
/* padLen is the last byte decrypted incremented by one to
 * included the padLen block itself
 */
padLen = resBuff[15] + 1;
```

- When calling the function to perform the chained cipher & hash operation, the fields of the operational data structure are:

Listing 30. Operational Data SSL inbound

```
pOpData->sessionCtx = sessionCtx;
pOpData->packetType = CPA_CY_SYM_PACKET_TYPE_FULL;
pOpData->pIv = pIvBuffer;
pOpData->ivLenInBytes = sizeof(sampleCipherIv);
pOpData->cryptoStartSrcOffsetInBytes = SSL_CombinedHeadSize;
pOpData->messageLenToCipherInBytes = bufferSize;
pOpData->hashStartSrcOffsetInBytes = SESSION_SEQ_START;
pOpData->messageLenToHashInBytes = sizeof(sessSeqNum) + sizeof(sampleHdrData) +
                                   bufferSize - MAC_LENGTH - padLen;
```

3.2.8 Chained Cipher & Hash – CCM use case

This example demonstrates the usage of the symmetric API to perform a CCM operation as described in NIST publication SP800-38C (*Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality*).

This sample is located in `\sym\ccm_sample`

The example is very similar to the cipher example, so only the differences are highlighted:

- For the generation-encryption process the session setup data is shown below:

Listing 31. Session Data CCM Generate-Encrypt

```
sessionSetupData.symOperation = CPA_CY_SYM_OP_ALGORITHM_CHAINING;
sessionSetupData.algChainOrder = CPA_CY_SYM_ALG_CHAIN_ORDER_HASH_THEN_CIPHER;

sessionSetupData.cipherSetupData.cipherAlgorithm = CPA_CY_SYM_CIPHER_AES_CCM;
sessionSetupData.cipherSetupData.pCipherKey = sampleKey;
sessionSetupData.cipherSetupData.cipherKeyLenInBytes = sizeof(sampleKey);
sessionSetupData.cipherSetupData.cipherDirection =
    CPA_CY_SYM_CIPHER_DIRECTION_ENCRYPT;

sessionSetupData.hashSetupData.hashAlgorithm = CPA_CY_SYM_HASH_AES_CCM;
sessionSetupData.hashSetupData.hashMode = CPA_CY_SYM_HASH_MODE_AUTH;
sessionSetupData.hashSetupData.digestResultLenInBytes = DIGEST_LENGTH;

/* Notice for CCM authKey and authKeyLen are not required this
 * information is provided by the cipherKey in cipherSetupData
 */
sessionSetupData.hashSetupData.authModeSetupData.aadLenInBytes =
    sizeof(sampleAssocData);
/* For CCM digestAppended and digestVerify are not required. In
 * the encrypt direction digestAppended is CPA_TRUE and
 * digestVerify is CPA_FALSE
 */
```

- For the decryption-verification process the session setup data is:

Listing 32. Session Data CCM Decrypt-Verify

```
sessionSetupData.symOperation = CPA_CY_SYM_OP_ALGORITHM_CHAINING;
sessionSetupData.algChainOrder = CPA_CY_SYM_ALG_CHAIN_ORDER_CIPHER_THEN_HASH;

sessionSetupData.cipherSetupData.cipherAlgorithm = CPA_CY_SYM_CIPHER_AES_CCM;
sessionSetupData.cipherSetupData.pCipherKey = sampleKey;
sessionSetupData.cipherSetupData.cipherKeyLenInBytes = sizeof(sampleKey);
sessionSetupData.cipherSetupData.cipherDirection =
    CPA_CY_SYM_CIPHER_DIRECTION_DECRYPT;

sessionSetupData.hashSetupData.hashAlgorithm = CPA_CY_SYM_HASH_AES_CCM;
sessionSetupData.hashSetupData.hashMode = CPA_CY_SYM_HASH_MODE_AUTH;
sessionSetupData.hashSetupData.digestResultLenInBytes = DIGEST_LENGTH;
sessionSetupData.hashSetupData.authModeSetupData.aadLenInBytes =
    sizeof(sampleAssocData);
```

- The IV and AAD buffers are allocated as shown below:

Listing 33. CCM Allocate IV and AAD buffers

```
/* Allocate memory to store IV. For CCM this is the counter block
 * ctr0 (size equal to AES block size). The implementation will
 * construct the ctr0 block given the nonce. Space for ctr0 must be
 * allocated here  */
status = PHYS_CONTIG_ALLOC(&pIvBuffer, AES_BLOCK_SIZE);
}

if (CPA_STATUS_SUCCESS == status) {
    /* Allocate memory for AAD. For CCM this memory will hold the 16 byte
     * B0 block, the 2 bytes encoded length of associated data, the
     * assocaiated data itself and any padding to ensure total size is
     * a multiple of the AES block size
     */
    aadBuffSize = B0_BLOCK_SIZE + ALEN_ENCODING_SIZE + sizeof(sampleAssocData);
    if (aadBuffSize % AES_BLOCK_SIZE) {
        aadBuffSize += AES_BLOCK_SIZE - (aadBuffSize % AES_BLOCK_SIZE);
    }
    status = PHYS_CONTIG_ALLOC(&pAadBuffer, aadBuffSize);
}
```

- The operational data needed to perform the generate-encrypt or decrypt-verify operation is shown below:

Listing 34. CCM Operational Data

```
pOpData->sessionCtx = sessionCtx;
pOpData->packetType = CPA_CY_SYM_PACKET_TYPE_FULL;
pOpData->pIv = pIvBuffer;
/* Even though the iv buffer is 16 bytes the ivLenInBytes
 * is set to the length of the nonce. For CCM valid lengths
 * are in the range 7-13
 */
pOpData->ivLenInBytes = sizeof(sampleNonce);
pOpData->cryptoStartSrcOffsetInBytes = 0;
pOpData->messageLenToCipherInBytes = sizeof(samplePayload);
/* Notice for CCM hash offset and length are not required */
pOpData->pAdditionalAuthData = pAadBuffer;

/* Populate pIv and pAdditionalAuthData buffers with nonce and assoc data */
CPA_CY_SYM_CCM_SET_NONCE(pOpData, sampleNonce, sizeof(sampleNonce));
CPA_CY_SYM_CCM_SET_AAD(pOpData, sampleAssocData, sizeof(sampleAssocData));
```

3.2.9 Chained Cipher & Hash – GCM use case

This example demonstrates the usage of the symmetric API to perform a GCM operation as described in NIST publication SP800-38D [*Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*].

These samples are located in `\sym\gcm_sample`

An example of the session setup data and operational data for GCM authenticated encryption and decryption is shown below.

- For authenticated encryption the session setup data is:

Listing 35. Session Data GCM Auth-Encrypt

```
sessionSetupData.symOperation = CPA_CY_SYM_OP_ALGORITHM_CHAINING;
sessionSetupData.algChainOrder = CPA_CY_SYM_ALG_CHAIN_ORDER_CIPHER_THEN_HASH;

sessionSetupData.cipherSetupData.cipherAlgorithm = CPA_CY_SYM_CIPHER_AES_GCM;
sessionSetupData.cipherSetupData.pCipherKey = sampleKey;
sessionSetupData.cipherSetupData.cipherKeyLenInBytes = sizeof(sampleKey);
sessionSetupData.cipherSetupData.cipherDirection =
    CPA_CY_SYM_CIPHER_DIRECTION_ENCRYPT;

sessionSetupData.hashSetupData.hashAlgorithm = CPA_CY_SYM_HASH_AES_GCM;
sessionSetupData.hashSetupData.hashMode = CPA_CY_SYM_HASH_MODE_AUTH;
sessionSetupData.hashSetupData.digestResultLenInBytes = TAG_LENGTH;
/* For GCM authKey and authKeyLen are not required this information
   is provided by the cipherKey in cipherSetupData */
sessionSetupData.hashSetupData.authModeSetupData.aadLenInBytes =
    sizeof(sampleAddAuthData);
/* Tag follows immediately after the region to hash */
sessionSetupData.digestIsAppended = CPA_TRUE;
/* digestVerify is not required to be set. For GCM authenticated
   encryption this value is understood to be CPA_FALSE */
```

- For authenticated decryption the session setup data is:

Listing 36. Session Data GCM Auth-Decrypt

```
sessionSetupData.symOperation = CPA_CY_SYM_OP_ALGORITHM_CHAINING;
sessionSetupData.algChainOrder = CPA_CY_SYM_ALG_CHAIN_ORDER_HASH_THEN_CIPHER;

sessionSetupData.cipherSetupData.cipherAlgorithm = CPA_CY_SYM_CIPHER_AES_GCM;
sessionSetupData.cipherSetupData.pCipherKey = sampleKey;
sessionSetupData.cipherSetupData.cipherKeyLenInBytes = sizeof(sampleKey);
sessionSetupData.cipherSetupData.cipherDirection =
    CPA_CY_SYM_CIPHER_DIRECTION_DECRYPT;

sessionSetupData.hashSetupData.hashAlgorithm = CPA_CY_SYM_HASH_AES_GCM;
sessionSetupData.hashSetupData.hashMode = CPA_CY_SYM_HASH_MODE_AUTH;
sessionSetupData.hashSetupData.digestResultLenInBytes = TAG_LENGTH;

/* For GCM authKey and authKeyLen are not required this information
   is provided by the cipherKey in cipherSetupData */
sessionSetupData.hashSetupData.authModeSetupData.aadLenInBytes =
    sizeof(sampleAddAuthData);
/* Tag follows immediately after the region to hash */
sessionSetupData.digestIsAppended = CPA_TRUE;
/* digestVerify is not required to be set. For GCM authenticated
   decryption this value is understood to be CPA_TRUE */
```

- The IV and AAD buffers are allocated as shown below:

Listing 37. GCM Allocate IV and AAD buffers

```
/* Allocate memory to store IV. For GCM this is the block J0
 * (size equal to AES block size). If iv is 12 bytes the
 * implementation will construct the J0 block given the iv.
 * If iv is not 12 bytes then the user must contruct the J0
 * block and give this as the iv. In both cases space for J0
 * must be allocated. */
status = PHYS_CONTIG_ALLOC(&pIvBuffer, AES_BLOCK_SIZE);
}
if (CPA_STATUS_SUCCESS == status) {
    /* Allocate memory for AAD. For GCM this memory will hold the
     * additional authentication data and any padding to ensure total
     * size is a multiple of the AES block size
     */
    aadBuffSize = sizeof(sampleAddAuthData);
    if (aadBuffSize % AES_BLOCK_SIZE) {
        aadBuffSize += AES_BLOCK_SIZE - (aadBuffSize % AES_BLOCK_SIZE);
    }
    status = PHYS_CONTIG_ALLOC(&pAadBuffer, aadBuffSize);
}
```

- The operational data needed to perform the encrypt or decrypt operation is:

Listing 38. GCM Operational Data

```
pOpData->sessionCtx = sessionCtx;
pOpData->packetType = CPA_CY_SYM_PACKET_TYPE_FULL;
pOpData->pIv = pIvBuffer;
/* In this example iv is 12 bytes. The implementation
 * will use the iv to generation the J0 block
 */
memcpy(pIvBuffer, sampleIv, sizeof(sampleIv));
pOpData->ivLenInBytes = sizeof(sampleIv);
pOpData->cryptoStartSrcOffsetInBytes = 0;
pOpData->messageLenToCipherInBytes = sizeof(samplePayload);
/* For GCM hash offset and length are not required */
pOpData->pAdditionalAuthData = pAadBuffer;
```

GMAC is supported using the same API and similar data structures as the general GCM case shown above. However for GMAC, the `messageLenToCipherInBytes` will be set to 0.

3.2.10 Chained Cipher & Hash using the Symmetric Data Plane API

This example demonstrates the usage of the data plane symmetric API to perform a "chained" cipher and hash operation. It encrypts some sample text using the 3DES algorithm in CBC mode, and then performs an MD5 HMAC operation on the ciphertext, writing the MAC to the buffer immediately after the ciphertext. Note this example is simplified to demonstrate the basics of how to use the API and how to build the structures required. This example does not demonstrate the optimal way to use the API to get maximum performance for a particular implementation. Please refer to implementation specific documentation (for example, the *Intel® Communications Chipset 8900 to 8920 Series Software Programmer's Guide*) and performance sample code for a guide on how to use the API for best performance.

These samples are located in `\sym\symdp_sample`

Note that use of the data plane symmetric API follows some of the same basic steps as the traditional symmetric API:

- Discover and start up the cryptographic service instance.
- Register a callback function for the instance.
- Create and initialize a session.
- Enqueue the symmetric operation on the instance.
- Submit the symmetric operation for processing.
- Poll the instance for a response.
- Tear down the session.
- Stop the Cryptographic service instance.

The following are the steps in more detail:

- Cryptographic service instances are discovered and started in the same way and using the same API as the traditional symmetric use cases described in Listing 4, Listing 5, and Listing 6.
- The next step is to register a callback function for the cryptographic instance. The function is called back in the context of the polling function when an asynchronous operation has completed. This function can perform whatever processing is appropriate to the application. Note this differs from the traditional symmetric API where the callback function is registered for the session.

Listing 39. Register Callback function

```
status = cpaCySymDpRegCbFunc(cyInstHandle, symDpCallback);
```

- Create and initialize a session:

Listing 40. Create and Initialize Data Plane Session

```
sessionSetupData.sessionPriority = CPA_CY_PRIORITY_HIGH;
sessionSetupData.symOperation = CPA_CY_SYM_OP_ALGORITHM_CHAINING;
sessionSetupData.algChainOrder = CPA_CY_SYM_ALG_CHAIN_ORDER_CIPHER_THEN_HASH;

sessionSetupData.cipherSetupData.cipherAlgorithm = CPA_CY_SYM_CIPHER_3DES_CBC;
sessionSetupData.cipherSetupData.pCipherKey = sampleCipherKey;
sessionSetupData.cipherSetupData.cipherKeyLenInBytes = sizeof(sampleCipherKey);
sessionSetupData.cipherSetupData.cipherDirection =
    CPA_CY_SYM_CIPHER_DIRECTION_ENCRYPT;

sessionSetupData.hashSetupData.hashAlgorithm = CPA_CY_SYM_HASH_MD5;
sessionSetupData.hashSetupData.hashMode = CPA_CY_SYM_HASH_MODE_AUTH;
sessionSetupData.hashSetupData.digestResultLenInBytes = DIGEST_LENGTH;
sessionSetupData.hashSetupData.authModeSetupData.authKey = sampleCipherKey;
sessionSetupData.hashSetupData.authModeSetupData.authKeyLenInBytes =
    sizeof(sampleCipherKey);

/* Even though MAC follows immediately after the region to hash
   digestIsAppended is set to false in this case to workaround
   errata number IXA00378322 */
sessionSetupData.digestIsAppended = CPA_FALSE;
sessionSetupData.verifyDigest = CPA_FALSE;

/* Determine size of session context to allocate */
PRINT_DBG("cpaCySymDpSessionCtxGetSize\n");
status = cpaCySymDpSessionCtxGetSize(cyInstHandle,
                                     &sessionSetupData,
                                     &sessionCtxSize);
}

if (CPA_STATUS_SUCCESS == status) {
    /* Allocate session context */
    status = PHYS_CONTIG_ALLOC(&sessionCtx, sessionCtxSize);
}

if (CPA_STATUS_SUCCESS == status) {
    /* Initialize the session */
    PRINT_DBG("cpaCySymDpInitSession\n");
    status = cpaCySymDpInitSession(cyInstHandle, &sessionSetupData, sessionCtx);
}

#ifdef LAC_HW_PRECOMPUTES
if (CPA_STATUS_SUCCESS == status) {
    /* Poll for hw pre-compute responses. */
    do {
        status = icp_sal_CyPollDpInstance(cyInstHandle, 0);
    } while (CPA_STATUS_SUCCESS != status);
}
#endif
```

- In this example, data is stored in flat buffers (as opposed to scatter gather lists). The operational data in this case is:

Listing 41. Data Plane Operational Data

```
sessionSetupData.sessionPriority = CPA_CY_PRIORITY_HIGH;
sessionSetupData.symOperation = CPA_CY_SYM_OP_ALGORITHM_CHAINING;
sessionSetupData.algChainOrder = CPA_CY_SYM_ALG_CHAIN_ORDER_CIPHER_THEN_HASH;

sessionSetupData.cipherSetupData.cipherAlgorithm = CPA_CY_SYM_CIPHER_3DES_CBC;
sessionSetupData.cipherSetupData.pCipherKey = sampleCipherKey;
sessionSetupData.cipherSetupData.cipherKeyLenInBytes = sizeof(sampleCipherKey);
sessionSetupData.cipherSetupData.cipherDirection =
    CPA_CY_SYM_CIPHER_DIRECTION_ENCRYPT;

sessionSetupData.hashSetupData.hashAlgorithm = CPA_CY_SYM_HASH_MD5;
sessionSetupData.hashSetupData.hashMode = CPA_CY_SYM_HASH_MODE_AUTH;
sessionSetupData.hashSetupData.digestResultLenInBytes = DIGEST_LENGTH;
sessionSetupData.hashSetupData.authModeSetupData.authKey = sampleCipherKey;
sessionSetupData.hashSetupData.authModeSetupData.authKeyLenInBytes =
    sizeof(sampleCipherKey);

/* Even though MAC follows immediately after the region to hash
   digestIsAppended is set to false in this case to workaround
```

- This request is then enqueued on the instance.

Listing 42. Data Plane Enqueue

```
status = cpaCySymDpEnqueueOp(pOpData, CPA_FALSE);
```

- Other requests can now be enqueued before submitting all the requests to be processed. This allows the cost of submitting a request (which can be expensive, in terms of cycles, for some hardware-based implementations) to be amortized over all enqueued requests on the instance. Once sufficient requests have been enqueued they are all submitted for processing:

Listing 43. Data Plane Perform

```
status = cpaCySymDpPerformOpNow(cyInstHandle);
```

- An alternative to calling the `cpaCySymDpPerformOpNow` function is to set `performOpNow` to CPA_TRUE when calling the enqueue functions (`cpaCySymDpEnqueueOp` or `cpaCySymDpEnqueueOpBatch`). This is illustrated in the data compression data plane example.
- After submitting a number of requests and possibly doing other work (e.g. enqueuing and submitting more requests) the application can poll for responses which will invoke the callback function registered with the instance. See implementation specific documentation for information on the implementations polling functions.

- Once all requests associated with a session have been completed the session can be removed.

Listing 44. Data Plane Remove Session

```
sessionStatus = cpaCySymDpRemoveSession(cyInstHandle, sessionCtx);
```

- Finally, clean up by freeing up memory, stopping the instance, etc.

3.2.11 TLS Key and MGF Mask Generation

Refer to the API manual for full details of Key and Mask Generation operations.

1. Define a Flat Buffer callback function as per the API prototype, see the API manual. If synchronous operation is preferred, instead simply pass NULL to the API for the callback parameter.
2. Allocate memory for the operation.
3. Populate data for the appropriate operation data structure, see the API manual.
   - Fill in the Flat Buffers, pointer to data and length
   - Fill in the options for the operation required.
4. Call the appropriate key or Mask Generation API.
5. Complete the operation.

   The API for TLS key operations is based on the TLS 1.1 standard (RFC 4346). Backward compatibility is supported with the legacy TLS 1.0 standard (RFC 2246). The user-defined label should be used for backward compatibility with the client write key, server write key, and iv block. See the Cryptographic API manual for details of populating `CpaCyKeyGenTlsOpData`, the operation data structure. The following sections describe examples of the parameter mapping to the Cryptographic API.

3.2.11.1 Setting CpaCyKeyGenTlsOpData Structure Fields

In RFC 4346, Section 6.3 key_block is described as:

```
key_block = PRF(SecurityParameters.master_secret,
                 "key expansion",
                 SecurityParameters.server_random +
                 SecurityParameters.client_random);
```

This maps to the Cryptographic API's `CpaCyKeyGenTlsOpData` as follows:

```
TLS Key-Material Derivation:
    tlsOp = CPA_CY_KEY_TLS_OP_KEY_MATERIAL_DERIVE
    secret = master secret key
    seed = server_random + client_random
    userLabel = NULL
```

3.2.12.1 Setting CpaCyKeyGenTlsOpData Structure Fields for backward compatibility

1. In RFC 2246, Section 6.3 `final_client_write_key` is described as:

```
final_client_write_key    = PRF(client_write_key,
                              "client write key",
                              client_random +
                              server_random)[0..15]
```

   This maps to the Cryptographic API's `CpaCyKeyGenTlsOpData` as follows:

```
TLS User Defined Derivation:
    tlsOp = CPA_CY_KEY_TLS_OP_USER_DEFINED
    secret = client_write_key
    seed = client_random + server_random
    userLabel = "client write key"
```

2. In RFC 2246, Section 6.3 `final_server_write_key` is described as:

```
final_server_write_key    = PRF(server_write_key,
                              "server write key",
                              client_random +
                              server_random)[0..15]
```

   This maps to the Cryptographic API's `CpaCyKeyGenTlsOpData` as follows:

```
TLS User Defined Derivation:
    tlsOp = CPA_CY_KEY_TLS_OP_USER_DEFINED
    secret = server_write_key
    seed = client_random + server_random
    userLabel = "server write key"
```

3. In RFC 2246, Section 6.3 `iv_block` is described as:

```
iv_block = PRF("", "IV block",
               client_random + server_random)[0..15]
```

This maps to the Cryptographic API's `CpaCyKeyGenTlsOpData` as follows:

```
TLS User Defined Derivation:
    tlsOp = CPA_CY_KEY_TLS_OP_USER_DEFINED
    secret = NULL
    seed = client_random + server_random
    userLabel = "IV block"
```

Memory for the user label must be physically contiguous memory allocated by the user. This memory must be available to the API for the duration of the operation.

3.3 Using the Diffie-Hellman API

This example demonstrates the usage of the Diffie-Hellman API.

These samples are located in `\asym\diffie_hellman_sample`

The following steps are carried out:

- The example uses the API asynchronously, therefore, you must define a Diffie-Hellman callback function per the API prototype.
- Instance discovery and start up is done in a way similar to that defined for the symmetric examples above.
- The function `sampleDhPerformOp` is called which does the following:
  - Allocate memory for the operation, and populate data for the appropriate DH phase 1 operation data structure to generate the public value. The fields to be allocated and populated are the prime P, the base G, and the private value X. Space must also be allocated for the output, which is the public value (PV).

Listing 49. Allocate Memory and Populate Operational Data

```
status = OS_MALLOC(&pCpaDhOpDataP1, sizeof(CpaCyDhPhase1KeyGenOpData));

/*
 * Allocate input buffers for phase 1 and copy data. Input to DH
 * phase 1 includes the prime (primeP), the base g (baseG) and
 * a random private value (privateValueX).
 */
if (CPA_STATUS_SUCCESS == status) {
    memset(pCpaDhOpDataP1, 0, sizeof(CpaCyDhPhase1KeyGenOpData));

    pCpaDhOpDataP1->primeP.dataLenInBytes = sizeof(primeP_768);
    status =
        PHYS_CONTIG_ALLOC(&pCpaDhOpDataP1->primeP.pData, sizeof(primeP_768));

    if (NULL != pCpaDhOpDataP1->primeP.pData) {
        memcpy(pCpaDhOpDataP1->primeP.pData, primeP_768, sizeof(primeP_768));
    }
}

if (CPA_STATUS_SUCCESS == status) {
    pCpaDhOpDataP1->baseG.dataLenInBytes = sizeof(baseG1);
    status = PHYS_CONTIG_ALLOC(&pCpaDhOpDataP1->baseG.pData, sizeof(baseG1));

    if (NULL != pCpaDhOpDataP1->baseG.pData) {
        memcpy(pCpaDhOpDataP1->baseG.pData, baseG1, sizeof(baseG1));
    }
}

if (CPA_STATUS_SUCCESS == status) {
    pCpaDhOpDataP1->privateValueX.dataLenInBytes = sizeof(privateValueX);
    status = PHYS_CONTIG_ALLOC(&pCpaDhOpDataP1->privateValueX.pData,
                               sizeof(privateValueX));

    if (NULL != pCpaDhOpDataP1->privateValueX.pData) {
        memcpy(pCpaDhOpDataP1->privateValueX.pData,
               privateValueX,
               sizeof(privateValueX));
    }
}
```

– Invoke the phase 1 operation, which performs the modular exponentiation such that PV = (baseG ^ privateValueX) mod primeP. Note that in the case of phase 1, the operation is invoked synchronously, hence the NULL pointer for the callback function.

Listing 50. Perform Phase 1 Operation

```
status = cpaCyDhKeyGenPhase1(
    cyInstHandle,
    NULL,                 /* synchronous mode */
    pCallbackTagPh1,      /* Opaque user data; */
    pCpaDhOpDataP1,       /* Structure containing p, g and x*/
    pLocalOctetStringPV); /* Public value (function output) */
```

– In a real-world implementation of a key exchange protocol, the public value generated above would now be shared with another party, B. This example uses this public value to go on and invoke the second phase operation. First allocate memory for the secret value, set up the operational data for the phase 2 operation, and then perform that operation. This operation is invoked asynchronously, taking the callback function defined earlier as a parameter:

Listing 51. Perform Phase 2 Operation

```
status = cpaCyDhKeyGenPhase2Secret(
    cyInstHandle,
    (const CpaCyGenFlatBufCbFunc)asymCallback, /* CB function*/
    pCallbackTagPh2,        /* pointer to the complete variable*/
    pCpaDhOpDataP2,         /* structure containing p, the public value & x*/
    pOctetStringSecretKey); /* private key (output of the function)*/
```

- Finally, clean up by freeing up memory, stopping the instance, etc.

3.4 Prime Number Testing

This example demonstrates the usage of the prime number testing API.

These samples are located in `\asym\prime_sample`

The following steps are carried out:

- The API is used asynchronously, therefore, a callback function is defined as per the API prototype.
- Instance discovery and start up is done in a way similar to that defined for the symmetric examples above.
- The function `primePerformOp` is called which does the following:
  - Allocate memory for the operation
  - Populate data for the appropriate input fields and perform the operation. The fields populated include the following:
  - Prime Candidate
  - Whether to perform GCD test
  - Whether to perform Fermat test
  - Number of Miller-Rabin rounds
  - Whether to perform Lucas test

Listing 52. Setup Operational Data and Test Prime

```
pPrimeTestOpData->primeCandidate.pData = pPrime;
pPrimeTestOpData->primeCandidate.dataLenInBytes = sizeof(samplePrimeP_768);
pPrimeTestOpData->performGcdTest = CPA_TRUE;
pPrimeTestOpData->performFermatTest = CPA_TRUE;
pPrimeTestOpData->numMillerRabinRounds = NB_MR_ROUNDS;
pPrimeTestOpData->millerRabinRandomInput.pData = pMR;
pPrimeTestOpData->millerRabinRandomInput.dataLenInBytes = sizeof(MR);
pPrimeTestOpData->performLucasTest = CPA_TRUE;

status =
    cpaCyPrimeTest(cyInstHandle,
                   (const CpaCyPrimeTestCbFunc)primeCallback, /* CB function */
                   (void *)&complete,                         /* callback tag */
                   pPrimeTestOpData, /* operation data */
                   &testPassed); /* return value: true if the number is probably
                                    a prime, false if it is not a prime */
}
```

- Finally, statistics are queried and the service stopped.

4 Intel® QuickAssist Technology Data Compression API

This chapter describes the sample code for the Intel® QuickAssist Technology Data Compression API, beginning with an API overview, and followed by descriptions of various scenarios to illustrate usage of the API.

4.1 Overview

The Intel® QuickAssist Technology Data Compression API can be categorized into three broad areas as follows:

- Common: This includes functionality for initialization and shutdown of the service.
- Instance Management: A given implementation of the API can present multiple instances of the compression service, each representing a logical or virtual "device". Request ordering is guaranteed within a given instance of the service.
- Transformation:
  - Compression functionality
  - Decompression functionality

These areas of functionality are defined in cpa_dc.h and cpa_dc_dp.h

The Intel® QuickAssist Technology Data Compression API uses the "base" API (cpa), which defines base data types used across all services of the Intel® QuickAssist Technology API.

4.1.1 Session

Similar to the symmetric cryptography API, the data compression API has the concept of a session. In the case of the compression API, a session is an object that describes the compression parameters to be applied across a number of requests. These requests might submit buffers within a single file, or buffers associated with a particular data stream or flow. A session object is described by the following:

- The compression level. Lower levels provide faster compression and the cost of compression ratio, whereas, higher levels provide a better compression ratio as the cost of performance.
- The compression algorithm to use (e.g. deflate) and what type of Huffman trees to use (static or dynamic).

- The session direction. If all requests on this session will be compression requests, then the direction can be set to compress (and similarly, for decompress). A combined direction is also available if both compression and decompression requests will be called using this session.
- The session state. A session can be described as stateful or stateless. Stateful sessions maintain history and state between calls to the API, stateless sessions do not.
  - Stateless compression does not require history data from a previous compression/decompression request to be restored before submitting the request. Stateless sessions are used when the output data is known to be constrained in size. An overflow condition (when the output data is about to exceed the output buffer) is treated as an error condition and an explicit error code is returned to the client. On receiving the error, the client application is required to resubmit the job in its entirety with a larger output buffer. Requests are treated independently; state and history are not saved and restored between calls.
  - Stateful sessions are required when the data to be compressed or decompressed is larger than the buffers being used. This is a normal mode of operation for applications such as gzip where the size of the uncompressed data is not known prior to execution and therefore the destination buffer may not be large enough to hold the resultant output. Requests to stateful sessions are not treated independently and state and history can be saved and restored between calls. The amount of history and state carried between calls depends on the compression level. For stateful compression/decompression, only one outstanding request may be in-flight at any one time for that session.

4.2 Sample – Stateful Data Compression

This example demonstrates the usage of the synchronous API, specifically using this API to perform a compression operation. It compresses a file via a stateful session using the deflate compress algorithm with static Huffman trees and using gzip style headers and footers.

These samples are located in `\dc\stateful_sample`

4.2.1 Session Establishment

This is the main entry point for the sample compression code. It demonstrates the sequence of calls to be made to the API in order to create a session, perform one or more compress operations, and then tear down the session. At this point, it is assumed that the instance has been discovered and started, and that the capabilities of the instance have been queried and found to be suitable.

A session is established by describing a session, determining how much session memory is required, and then invoking the session initialization function `cpaDcInitSession`.

Listing 53. Create and Initialize Stateful Session

```
sd.compLevel = CPA_DC_L4;
sd.compType = CPA_DC_DEFLATE;
sd.huffType = CPA_DC_HT_STATIC;
sd.sessDirection = CPA_DC_DIR_COMBINED;
sd.sessState = CPA_DC_STATEFUL;
#if (CPA_DC_API_VERSION_NUM_MAJOR == 1 && CPA_DC_API_VERSION_NUM_MINOR < 6)
sd.deflateWindowSize = 7;
#endif
sd.checksum = CPA_DC_CRC32;

/* Determine size of session context to allocate */
PRINT_DBG("cpaDcGetSessionSize\n");
status = cpaDcGetSessionSize(dcInstHandle, &sd, &sess_size, &ctx_size);
}

if (CPA_STATUS_SUCCESS == status) {
    /* Allocate session memory */
    status = PHYS_CONTIG_ALLOC(&sessionHdl, sess_size);
}

if ((CPA_STATUS_SUCCESS == status) && (ctx_size != 0)) {
    /* Allocate context bufferlist */
    status = cpaDcBufferListGetMetaSize(dcInstHandle, 1, &buffMetaSize);

    if (CPA_STATUS_SUCCESS == status) {
        status = PHYS_CONTIG_ALLOC(&pBufferMeta, buffMetaSize);
    }

    if (CPA_STATUS_SUCCESS == status) {
        status = OS_MALLOC(&pBufferCtx, bufferListMemSize);
    }

    if (CPA_STATUS_SUCCESS == status) {
        status = PHYS_CONTIG_ALLOC(&pCtxBuf, ctx_size);
    }

    if (CPA_STATUS_SUCCESS == status) {
        pFlatBuffer = (CpaFlatBuffer *)(pBufferCtx + 1);

        pBufferCtx->pBuffers = pFlatBuffer;
        pBufferCtx->numBuffers = 1;
        pBufferCtx->pPrivateMetaData = pBufferMeta;

        pFlatBuffer->dataLenInBytes = ctx_size;
        pFlatBuffer->pData = pCtxBuf;
    }
}
/* Initialize the Stateful session */
if (CPA_STATUS_SUCCESS == status) {
    PRINT_DBG("cpaDcInitSession\n");
    status = cpaDcInitSession(dcInstHandle,
                              sessionHdl, /* session memory */
                              &sd,        /* session setup data */
                              pBufferCtx, /* context buffer */
                              NULL); /* callback function NULL for sync mode */
}
```

Source and destination buffers must be established.

Listing 54. Stateful Compression Memory Allocation

```
numBuffers = 1; /* only using 1 buffer in this case */
/* allocate memory for bufferlist and array of flat buffers in a contiguous
 * area and carve it up to reduce number of memory allocations required. */
bufferListMemSize =
    sizeof(CpaBufferList) + (numBuffers * sizeof(CpaFlatBuffer));

status = cpaDcBufferListGetMetaSize(dcInstHandle, numBuffers, &bufferMetaSize);

/* Allocate source buffer */
if (CPA_STATUS_SUCCESS == status) {
    status = PHYS_CONTIG_ALLOC(&pBufferMetaSrc, bufferMetaSize);
}
if (CPA_STATUS_SUCCESS == status) {
    status = OS_MALLOC(&pBufferListSrc, bufferListMemSize);
}
if (CPA_STATUS_SUCCESS == status) {
    status = PHYS_CONTIG_ALLOC(&pSrcBuffer, SAMPLE_BUFF_SIZE);
}

/* Allocate destination buffer the same size as source buffer */
if (CPA_STATUS_SUCCESS == status) {
    status = PHYS_CONTIG_ALLOC(&pBufferMetaDst, bufferMetaSize);
}
if (CPA_STATUS_SUCCESS == status) {
    status = OS_MALLOC(&pBufferListDst, bufferListMemSize);
}
if (CPA_STATUS_SUCCESS == status) {
    status = PHYS_CONTIG_ALLOC(&pDstBuffer, SAMPLE_BUFF_SIZE);
}

if (CPA_STATUS_SUCCESS == status) {
    /* Build source bufferList */
    pFlatBuffer = (CpaFlatBuffer *)(pBufferListSrc + 1);

    pBufferListSrc->pBuffers = pFlatBuffer;
    pBufferListSrc->numBuffers = 1;
    pBufferListSrc->pPrivateMetaData = pBufferMetaSrc;

    pFlatBuffer->dataLenInBytes = SAMPLE_BUFF_SIZE;
    pFlatBuffer->pData = pSrcBuffer;

    /* Build destination bufferList */
    pFlatBuffer = (CpaFlatBuffer *)(pBufferListDst + 1);

    pBufferListDst->pBuffers = pFlatBuffer;
    pBufferListDst->numBuffers = 1;
    pBufferListDst->pPrivateMetaData = pBufferMetaDst;

    pFlatBuffer->dataLenInBytes = SAMPLE_BUFF_SIZE;
    pFlatBuffer->pData = pDstBuffer;
```

At this point, the application has opened an instance, established a session and allocated buffers. It is time to start some compress operations. To produce gzip style compressed files, the first thing that needs to be performed is header generation.

Listing 55. Create Header

```
/* Write RFC1952 gzip header to destination buffer */
status = cpaDcGenerateHeader(sessionHdl, pFlatBuffer, &hdr_sz);
}
if (CPA_STATUS_SUCCESS == status) {
    /* write out header */
    fwrite(pFlatBuffer->pData, 1, hdr_sz, dstFile);
```

`cpaDcGenerateHeader` produces a gzip style header (compliant with RFC 1952) when the session setup data is set such that compType is CPA_DC_DEFLATE and checksum is CPA_DC_CRC32. Note, alternatively, a zlib style header (compliant with RFC 1950) can be produced if the session setup data is set such that compType is CPA_DC_DEFLATE and checksum is CPA_DC_ADLER32.

Perform Operation: This listing demonstrates looping through a file, reading the data, invoking the data compress operation and writing the results to the output file.

Listing 56. Perform Stateful Compression Operation

```
pBufferListSrc->pBuffers->dataLenInBytes = 0;
while ((!feof(srcFile)) && (CPA_STATUS_SUCCESS == status)) {
    /* read from file into src buffer */
    pBufferListSrc->pBuffers->pData = pSrcBuffer;
    pBufferListSrc->pBuffers->dataLenInBytes +=
        fread(pSrcBuffer + pBufferListSrc->pBuffers->dataLenInBytes,
              1,
              SAMPLE_BUFF_SIZE - pBufferListSrc->pBuffers->dataLenInBytes,
              srcFile);
    if (pBufferListSrc->pBuffers->dataLenInBytes < SAMPLE_BUFF_SIZE) {
        flush = CPA_DC_FLUSH_FINAL;
    } else {
        flush = CPA_DC_FLUSH_SYNC;
    }
    do {

        PRINT_DBG("cpaDcCompressData\n");
        status = cpaDcCompressData(dcInstHandle,
                                   sessionHdl,
                                   pBufferListSrc, /* source buffer list */
                                   pBufferListDst, /* destination buffer list */
                                   &dcResults,     /* results structure */
                                   flush,          /* Stateful session */
                                   NULL);

        if (CPA_STATUS_SUCCESS != status) {
            PRINT_ERR("cpaDcCompressData failed. (status = %d)\n", status);
            break;
        }
```

Finally, a gzip footer is generated. Similar to the call to `cpaDcGenerateHeader` a gzip footer (compliant with RFC 1952) is produced because the session setup data is set such that `compType` is `CPA_DC_DEFLATE` and `checksum` is `CPA_DC_CRC32`. The call to `cpaDcGenerateFooter` increments the `produced` field of the `CpaDcRqResults` structure by the size of the footer added. In this example the data produced so far has already been written out to the file so the `produced` field of the `CpaDcRqResults` structure is cleared before calling the `cpaDcGenerateFooter` function.

Listing 57. Create Footer

```
dcResults.produced = 0;
/* Write RFC1952 gzip footer to destination buffer */
status = cpaDcGenerateFooter(sessionHdl, pFlatBuffer, &dcResults);
}
if (CPA_STATUS_SUCCESS == status) {
    /* write out footer */
    fwrite(pFlatBuffer->pData, 1, dcResults.produced, dstFile);
}
```

Because this session was created with `CPA_DC_DIR_COMBINED` it can be used to decompress data also. The next listing demonstrates looping through a file, reading the compressed data, invoking the data decompress operation and writing the results to the output file. In this case the overflow condition has to be taken into account.

Listing 58. Perform Stateful Decompression Operation

```
pBufferListSrc->pBuffers->dataLenInBytes = 0;
while ((!feof(srcFile)) && (CPA_STATUS_SUCCESS == status)) {
    /* read from file into src buffer */
```

Once all operations on this session have been completed, the session is torn down.

Listing 59. Remove Stateful Session

```
sessionStatus = cpaDcRemoveSession(dcInstHandle, sessionHdl);
```

You can query statistics at this point, which can be useful for debugging. Note that some implementations may also make the statistics available through other mechanisms, such as the `/proc` virtual filesystem.

Finally, clean up by freeing up memory, stopping the instance, etc.

4.3 Sample – Stateless Data Compression

This example demonstrates the usage of the asynchronous API, specifically using this API to perform a compression operation. It compresses a data buffer via stateless session using the deflate compress algorithm with dynamic Huffman trees.

- The example below compresses a block of data into a compressed block.

These samples are located in `\dc\stateless_sample`

In this example dynamic Huffman trees are used. The instance can be queried to ensure dynamic Huffman trees are supported and if an instance specific buffer is required to perform a dynamic Huffman tree deflate request.

Listing 60. Querying and Starting a Compression Instance

```
status = cpaDcQueryCapabilities(dcInstHandle, &cap);
if (status != CPA_STATUS_SUCCESS) {
    return status;
}

if (!cap.statelessDeflateCompression || !cap.statelessDeflateDecompression ||
    !cap.checksumAdler32 ||
    !cap.dynamicHuffman) {
    PRINT_DBG("Error: Unsupported functionality\n");
    return CPA_STATUS_FAIL;
}

if (cap.dynamicHuffmanBufferReq) {
    status = cpaDcBufferListGetMetaSize(dcInstHandle, 1, &buffMetaSize);

    if (CPA_STATUS_SUCCESS == status) {
        status =
            cpaDcGetNumIntermediateBuffers(dcInstHandle, &numInterBuffLists);
    }
    if (CPA_STATUS_SUCCESS == status && 0 != numInterBuffLists) {
        status = PHYS_CONTIG_ALLOC(&bufferInterArray,
                                   numInterBuffLists * sizeof(CpaBufferList *));
    }
    for (bufferNum = 0; bufferNum < numInterBuffLists; bufferNum++) {
        if (CPA_STATUS_SUCCESS == status) {
            status = PHYS_CONTIG_ALLOC(&bufferInterArray[bufferNum],
                                       sizeof(CpaBufferList));
        }
        if (CPA_STATUS_SUCCESS == status) {
            status = PHYS_CONTIG_ALLOC(
                &bufferInterArray[bufferNum]->pPrivateMetaData, buffMetaSize);
        }
        if (CPA_STATUS_SUCCESS == status) {
            status = PHYS_CONTIG_ALLOC(&bufferInterArray[bufferNum]->pBuffers,
                                       sizeof(CpaFlatBuffer));
        }
        if (CPA_STATUS_SUCCESS == status) {
            /* Implementation requires an intermediate buffer approximately
                       twice the size of the output buffer */
            status =
                PHYS_CONTIG_ALLOC(&bufferInterArray[bufferNum]->pBuffers->pData,
                                  2 * SAMPLE_MAX_BUFF);
            bufferInterArray[bufferNum]->numBuffers = 1;
            bufferInterArray[bufferNum]->pBuffers->dataLenInBytes =
                2 * SAMPLE_MAX_BUFF;
        }

    } /* End numInterBuffLists */
}

if (CPA_STATUS_SUCCESS == status) {
    /*
     * Set the address translation function for the instance
     */
    status = cpaDcSetAddressTranslation(dcInstHandle, sampleVirtToPhys);
}

if (CPA_STATUS_SUCCESS == status) {
    /* Start DataCompression component */
    PRINT_DBG("cpaDcStartInstance\n");
    status =
        cpaDcStartInstance(dcInstHandle, numInterBuffLists, bufferInterArray);
}
```

The next listing demonstrates the sequence of calls to be made to the API in order to create a session. A session is established by describing the session, determining how much session memory is required, and then invoking the session initialization function `cpaDcInitSession`.

Listing 61. Create and Initialize Stateless Session

```
sd.compLevel = CPA_DC_L4;
sd.compType = CPA_DC_DEFLATE;
sd.huffType = CPA_DC_HT_FULL_DYNAMIC;
/* If the implementation supports it, the session will be configured
 * to select static Huffman encoding over dynamic Huffman as
 * the static encoding will provide better compressibility.
 */
if (cap.autoSelectBestHuffmanTree) {
    sd.autoSelectBestHuffmanTree = CPA_TRUE;
} else {
    sd.autoSelectBestHuffmanTree = CPA_FALSE;
}
sd.sessDirection = CPA_DC_DIR_COMBINED;
sd.sessState = CPA_DC_STATELESS;
#if (CPA_DC_API_VERSION_NUM_MAJOR == 1 && CPA_DC_API_VERSION_NUM_MINOR < 6)
sd.deflateWindowSize = 7;
#endif
sd.checksum = CPA_DC_ADLER32;

/* Determine size of session context to allocate */
PRINT_DBG("cpaDcGetSessionSize\n");
status = cpaDcGetSessionSize(dcInstHandle, &sd, &sess_size, &ctx_size);
}

if (CPA_STATUS_SUCCESS == status) {
    /* Allocate session memory */
    status = PHYS_CONTIG_ALLOC(&sessionHdl, sess_size);
}

/* Initialize the Stateless session */
if (CPA_STATUS_SUCCESS == status) {
    PRINT_DBG("cpaDcInitSession\n");
    status = cpaDcInitSession(
        dcInstHandle,
        sessionHdl,  /* session memory */
        &sd,         /* session setup data */
        NULL,        /* pContexBuffer not required for stateless operations */
        dcCallback); /* callback function */
}
```

Source and destination buffers are allocated in a similar way to the stateful example above.

Perform Operation: This listing demonstrates invoking the data compress operation, in the stateless case.

Listing 62. Data Plane Remove Compression Session

```
sessionStatus = cpaDcRemoveSession(dcInstHandle, sessionHdl);
```

4.4 Sample – Stateless Data Compression Using Multiple Compress Operations

This example demonstrates the use of the asynchronous API, specifically using this API to perform a compression operation. It compresses a data buffer using multiple stateless compression API requests and maintains length and checksum information across the multiple requests without the overhead of maintaining full history information as used in a stateful operation.

The samples are located in: `\dc\stateless_multi_op_checksum_sample`

In this sample, session creation is the same as for a regular stateless operation. Refer to the previous sample described in 4.3 for details.

Perform Operation: This listing demonstrates the invoking of the data compress operation in the stateless case, while maintaining checksum information across multiple compress operations. The key points to note are:

- The initial value of `dcResults.checksum` is set to `0` for CRC32, or set to `1` for Adler32 when invoking the first compress or decompress operation for a data set.

Listing 63. Setting the Initial Value of the Checksum

```
if (sd.checksum == CPA_DC_ADLER32) {
    /* Initialize checksum to 1 for Adler32 */
    dcResults.checksum = 1;
} else {
    /* Initialize checksum to 0 for CRC32 */
    dcResults.checksum = 0;
}
```

- The value of `dcResults.checksum` when invoking a subsequent compress operation for a data set is set to the `dcResults.checksum` value returned from the previous compress operation on that data set.

4.5 Sample – Data Compression Data Plane API

This example demonstrates the usage of the data plane data compression API to perform a compression operation. It compresses a data buffer via stateless session using the deflate compress algorithm with dynamic Huffman trees. Again, this example is simplified to demonstrate the basics of how to use the API and how to build the structures required. This example does not demonstrate the optimal way to use the API to get maximum performance for a particular implementation. Please refer to implementation specific documentation (for example, the Intel® Communications Chipset 8900 to 8920 Series Software Programmer's Guide) and performance sample code for a guide on how to use the API for best performance.

These samples are located in `\dc\dc_dp_sample`

The data plane data compression API is used in a similar way to the data plane symmetric cryptographic API:

- Data compression service instances are queried and started in the same way and using the same functions as before (see Listing 1 and Listing 59).
- The next step is to register a callback function for the data compression instance.

Listing 64. Register Compression Callback function

```
status = cpaDcDpRegCbFunc(dcInstHandle, dcDpCallback);
```

- Create and initialize a session:

Listing 65. Create and Initialize Compression Data Plane Session

```
if (CPA_STATUS_SUCCESS == status) {
    sd.compLevel = CPA_DC_L4;
    sd.compType = CPA_DC_DEFLATE;
    sd.huffType = CPA_DC_HT_FULL_DYNAMIC;
    /* If the implementation supports it, the session will be configured
     * to select static Huffman encoding over dynamic Huffman as
     * the static encoding will provide better compressibility.
     */
    if (cap.autoSelectBestHuffmanTree) {
        sd.autoSelectBestHuffmanTree = CPA_TRUE;
    } else {
        sd.autoSelectBestHuffmanTree = CPA_FALSE;
    }
    sd.sessDirection = CPA_DC_DIR_COMBINED;
    sd.sessState = CPA_DC_STATELESS;
#if (CPA_DC_API_VERSION_NUM_MAJOR == 1 && CPA_DC_API_VERSION_NUM_MINOR < 6)
    sd.deflateWindowSize = 7;
#endif
    sd.checksum = CPA_DC_CRC32;

    /* Determine size of session context to allocate */
    PRINT_DBG("cpaDcGetSessionSize\n");
    status = cpaDcGetSessionSize(dcInstHandle, &sd, &sess_size, &ctx_size);
}

if (CPA_STATUS_SUCCESS == status) {
    /* Allocate session memory */
    status = PHYS_CONTIG_ALLOC(&sessionHdl, sess_size);
}

/* Initialize the Stateless session */
if (CPA_STATUS_SUCCESS == status) {
    PRINT_DBG("cpaDcDpInitSession\n");
    status = cpaDcDpInitSession(dcInstHandle,
                                sessionHdl, /* session memory */
                                &sd);       /* session setup data */
}
```

- In this example, input and output data is stored in a scatter gather list. The source and destination buffers are described using the `CpaPhysBufferList` structure. In this example the allocation (which needs to be 8-byte aligned) and setup of the source buffer is shown. The destination buffers can be allocated and set up in a similar way.

Listing 66. Setup Source Buffer

```
numBuffers = 2;
/* Size of CpaPhysBufferList and array of CpaPhysFlatBuffers */
bufferListMemSize =
    sizeof(CpaPhysBufferList) + (numBuffers * sizeof(CpaPhysFlatBuffer));

/* Allocte 8-byte alligned source buffer List */
status = PHYS_CONTIG_ALLOC_ALIGNED(&pBufferListSrc, bufferListMemSize, 8);
if (CPA_STATUS_SUCCESS == status) {
    /* Allocate first data buffer to hold half the data */
    status = PHYS_CONTIG_ALLOC(&pSrcBuffer, (sizeof(sampleData)) / 2);
}
if (CPA_STATUS_SUCCESS == status) {
    /* Allocate second data buffer to hold half the data */
    status = PHYS_CONTIG_ALLOC(&pSrcBuffer2, (sizeof(sampleData)) / 2);
}
if (CPA_STATUS_SUCCESS == status) {
    /* copy source into buffer */
    memcpy(pSrcBuffer, sampleData, sizeof(sampleData) / 2);
    memcpy(pSrcBuffer2,
           &(sampleData[sizeof(sampleData) / 2]),
           sizeof(sampleData) / 2);

    /* Build source bufferList */
    pBufferListSrc->numBuffers = 2;
    pBufferListSrc->flatBuffers[0].dataLenInBytes = sizeof(sampleData) / 2;
    pBufferListSrc->flatBuffers[0].bufferPhysAddr =
        sampleVirtToPhys(pSrcBuffer);
    pBufferListSrc->flatBuffers[1].dataLenInBytes = sizeof(sampleData) / 2;
    pBufferListSrc->flatBuffers[1].bufferPhysAddr =
        sampleVirtToPhys(pSrcBuffer2);
```

- The operational data in this case is:

Listing 67. Compression Data Plane Operational Data

```
/* Allocate memory for operational data. Note this needs to be
 * 8-byte aligned, contiguous, resident in DMA-accessible
 * memory.
 */
status = PHYS_CONTIG_ALLOC_ALIGNED(&pOpData, sizeof(CpaDcDpOpData), 8);
}

if (CPA_STATUS_SUCCESS == status) {

    pOpData->bufferLenToCompress = sizeof(sampleData);
    pOpData->bufferLenForData = sizeof(sampleData);
    pOpData->dcInstance = dcInstHandle;
    pOpData->pSessionHandle = sessionHdl;
    pOpData->srcBuffer = sampleVirtToPhys(pBufferListSrc);
    pOpData->srcBufferLen = CPA_DP_BUFLIST;
    pOpData->destBuffer = sampleVirtToPhys(pBufferListDst);
    pOpData->destBufferLen = CPA_DP_BUFLIST;
    pOpData->sessDirection = CPA_DC_DIR_COMPRESS;
    pOpData->thisPhys = sampleVirtToPhys(pOpData);
    pOpData->pCallbackTag = (void *)0;
```

- This request is then enqueued and submitted on the instance.

Listing 68. Data Plane Enqueue and Submit

```
status = cpaDcDpEnqueueOp(pOpData, CPA_TRUE);
```

- After possibly doing other work (e.g. enqueuing and submitting more requests) the application can poll for responses which will invoke the callback function registered with the instance. See implementation specific documentation for information on the implementations polling functions.
- Once all requests associated with a **s**ession have been completed the session can be removed.

Listing 69. Data Plane Remove Compression Session

```
sessionStatus = cpaDcDpRemoveSession(dcInstHandle, sessionHdl);
```

- Finally, clean up by freeing up memory, stopping the instance, etc.