Programmer's Guide

Intel® QuickAssist Technology

Hardware Version 2.0

March 2024

Document Number: 743912-004

Performance varies by use, configuration and other factors. Learn more on the Intel's Performance Index site.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure.

Your costs and results may vary.

Intel technologies may require enabled hardware, software or service activation.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Code names are used by Intel to identify products, technologies, or services that are in development and not publicly available. These are not "commercial" names and not intended to function as trademarks.

See Intel's Legal Notices and Disclaimers.

Contents

List of Tables

1 About this Document

This programmer's guide provides information on the architecture of the software and usage guidelines.

Information on the use of Intel® QuickAssist Technology (Intel® QAT) APIs, which provide the interface to the acceleration services (cryptographic and data compression), is documented in the related Intel® QAT software library documentation referenced in the Release Notes.

In this document, for convenience:

- *Software package* is used as a generic term for the Intel® QAT Software Package for Hardware Version 2.0.
- *Acceleration driver* is used as a generic term for the software that allows the Intel® QAT Software Library APIs to access the Intel® QAT Endpoint(s).

Note: Refer to the Release Notes for a list of supported platforms.

Note: Current version of this document covers the out-of-tree acceleration driver. Future version of this document will be updated to cover in-tree driver as well.

For additional details on in-tree driver refer to https://github.com/intel/qatlib.

1.1 Conventions and Terminology

The following conventions are used in this manual:

- `Code text` - code examples, command line entries, Application Porgramming Interface (API) names, parameters, filenames, directory paths, and executables.
- **Bold text** - graphical user interface entries, buttons, and actions in instructions.
- *Italic text* - key terms and publication titles.

The following terms and acronyms are used in this manual.

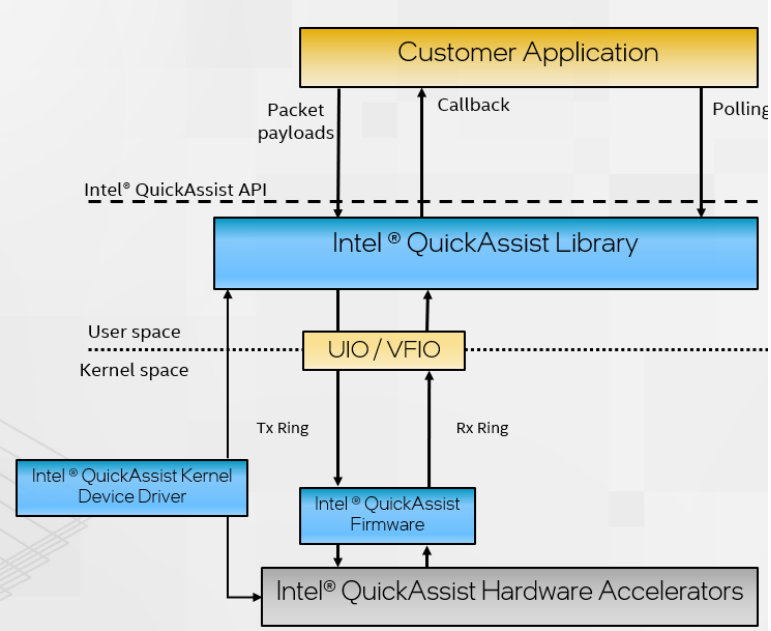
Table 1: Terminology

Term	Description
API	Application Programming Interface
asym	Asymmetric Cryptography
BDF	Bus Device Function
BOM	Bill of Materials
CBC	Cipher Block Chaining
cy	Cryptography
dc	Data Compression
GRUB	Grand Unified Bootloader
OS	Operating System
PCI	Peripheral Component Interconnect
PF	PCIe Physical Function
Intel® QAT	Intel® QuickAssist Technology
SKU	Stock Keeping Unit
sIOV	Scalable IOV
SR-IOV	Single Root-I/O Virtualization
VF	Virtual Function

2 Architecture

Because the hardware is accessed via the Intel® QAT APIs, it is not necessary to know all of the hardware and software architecture details, but some knowledge of the underlying hardware and software is helpful for performance optimization and debug purposes.

A simplified view of the hardware/software stack is shown in the following figure.



The flow can be broken down as:

1. Application submits payloads via the Intel® QuickAssist API as part of the request. The user space library converts these requests into descriptors and places these in the Transmit (Tx) *hardware-assisted queues* (aka ring).
2. Firmware parses the descriptor and configures the accelerators accordingly. Upon a job completion firmware returns the processed payload (either encrypted or compressed or both) and generates a response message. This response message is inserted in the response ring.
3. A polling thread owned by the application queries the response ring via the Intel® QuickAssist Library. If the application choses non-blocking calls the user space library will issue a callback to the application to inform that the operation is complete.

Note: The UIO (to be replaced with VFIO) layer is a framework present in both Linux Kernel and user space library libudev. This framework enables exchanging data between Kernel and user space. It offers better latency performance than IOCTL.

3 Infrastructure

The following sections describe the building blocks of the Intel® QAT Endpoints' architecture.

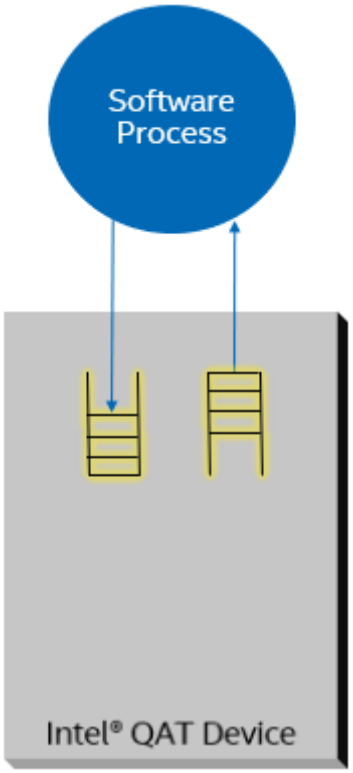
3.1 Queues and Queue Pairs

Communication between CPU and Intel® QuickAssist Technology hardware is via hardware-assisted queues (aka rings):

- Queues are *circular buffers*.
- Memory is in System DRAM.
- Device is configured with base address, entry size and number of entries via device CSRs.
- *Head* and *Tail* pointers are in device CSRs (MMIO space).

3.1.1 Queues Pairs

- To send a request, software writes request descriptor to next available entry in the request queue, and updates the tail pointer.
- Device firmware reads request descriptor from request queue, updating the head pointer. It then processes the request, writes response descriptor onto response queue, and updates the tail pointer.
- Response queues can be configured to generate an interrupt when device firmware updates the tail pointer, or can be polled.



3.1.2 Queue Bundles

Queues are grouped into bundles of 8 queues (4 Queue Pairs (QPs)).

- When SR-IOV is enabled, each bundle shows up as a separate Virtual Function.
- When s-IOV is enabled each QP is exposed as a separate Assignable Device Interface (ADI).

Within each bundle, by default, a separate QP is used for each of the three possible services:

1. Public Key Crypto
2. Symmetric Crypto
3. Data Compression

- Public Key Crypto
- Symmetric Crypto
- Data Compression

Max of 2 service types per QAT device at a time. Each QP can be allocated to a specific service, in a bare metal environment.

3.2 Service Instances

At the Intel® QuickAssist Technology API, we abstract queue pairs using the concept of service instances.

- To use a service, an application must first get a handle to a service instance.
- Corresponds to one or more queue pairs:
  - Data compression instance contains 1 queue pair.
  - Cryptographic instance:
    - QAT Gen2: contains 2 queue pairs, one for each sub-service of crypto (symmetric crypto, public key crypto).
    - QAT Gen4: crypto instances can be specified as either sym (symmetric) or asym (asymmetric) cryptography and contain 1 queue pair.

3.2.1 Configurable Items (via config file)

- Queue depth (for each queue).
- Number of service instances per process for a given device (limited by available rings), for example:
  - One per address space (e.g. user space processes).
  - One per software or hardware thread (logical core), to avoid contention.
- Number of queue pairs per service, per bundle/VF, will be configurable in future.

3.3 Memory Management

This section describes memory management requirements for submitting buffers to the QAT hardware.

3.3.1 Shared Virtual Memory

Shared Virtual Memory (SVM) is a new feature in QAT 2.0 hardware. In QAT 1.x hardware, memory needs to be submitted to the hardware as pinned and physically contiguous memory. In QAT 2.0, SVM allows direct submission of an applications buffer, thus removing the memcpy cycle cost, cache thrashing, and memory bandwidth. The SVM feature enables passing virtual addresses to the QAT hardware for processing acceleration requests.

With SVM:

- Virtually contiguous (can also deal with Scatter Gather Lists of virtually addressed buffers).
- Virtually addressed.

- Can tolerate page faults but Pinning (i.e. locked, guaranteed resident in physical memory) is recommended for performance.

3.3.1.1 SVM Kernel Requirements

In order to use SVM, ensure that kernel version v6.1 or higher is used. Alternatively verify the following kernel patches are applied.

- 81c95fbaebfa5990c3c786c8c3e87426a33106fe
- e65a6897be5e4939d477c4969a05e12d90b08409

Verification can be done with the following steps:

```
git tag --contains 81c95fbaebfa5990c3c786c8c3e87426a33106fe
git tag --contains e65a6897be5e4939d477c4969a05e12d90b08409
```

This requirement provides mitigation for the issue QAT20-23616 described in the Release Notes.

The following kernel boot parameters need to be defined in order to utilize SVM.

```
intel_iommu=on,sm_on
```

Refer to *Shared Virtual Memory Parameters* for details on QAT configuration files updates required to support SVM.

3.3.2 DMA-able Memory

If SVM is not enabled, Memory passed to Intel® QuickAssist Technology hardware must be DMA'able.

- Physically contiguous (can also deal with Scatter Gather Lists).
- Physically addressed.
  - If VT-d is enabled (e.g. in virtualized system), then Intel IOMMU will translate to host physical addresses as needed.
- Pinned (i.e. locked, guaranteed resident in physical memory).

Intel provides a User Space DMA-able Memory (USDM) component (kernel driver and corresponding user space library) which allocates/frees DMA-able memory, mapped to user space, performs virtual to physical address translation on memory allocated by this library

This component is used by the sample code supplied with the user space library.

3.3.3 Memory Type Determination

QAT 2.0 hardware offers the application to use virtual memory directly to sending the acceleration requests and saving the memory copy overhead. However, different SVM configurations will result in different memory types. The QAT package offers memory management library called User Space DMAable Memory(USDM) to help user space applications using the pinned memory.

SVMEnabled	ATEnabled	Memory Type
FALSE(0)	FALSE(0)	Pinned Memory (USDM)
TRUE(1)	FALSE(0)	Pinned Memory (USDM)
FALSE(0)	TRUE(1)	Invalid configuration
TRUE(1)	TRUE(1)	Pinned Memory (USDM) or Dynamic Memory (malloc/ zalloc/mmap...)

3.3.4 Buffer Formats

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

- Flat Buffers represent a single region of physically contiguous memory.
- Scatter-Gather Lists (SGL) are essentially an array of flat buffers, for cases where the memory is not all physically contiguous.

3.3.4.1 Flat Buffers

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

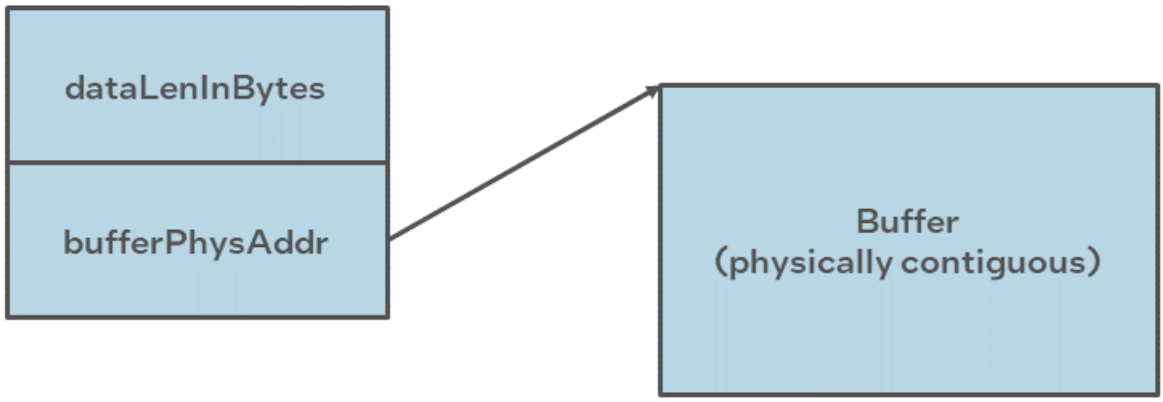
- Data pointer `pData`: points to the start address of the data or payload. The data pointer is a virtual address; however, the actual data pointed to is required to be in contiguous and DMAable physical memory. This buffer type is typically used when simple, unchained buffers are needed.
- Length of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

3.3.4.2 Scatter-Gather List (SGL) Buffers

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:

- The number of buffers in the list.
- `pBuffers`: 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.
- `pMetaData`: pointer to metadata required by the API:
  - The metadata 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 metadata 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 metadata pointer are required to reference physically contiguous DMAable memory.
  - There is a performance impact when using scatter-gather lists instead of flat buffers. Refer to the Performance Optimization Guide for additional information.
  - Scatter-Gather list (SGL) buffers should not have more than 256 entries.

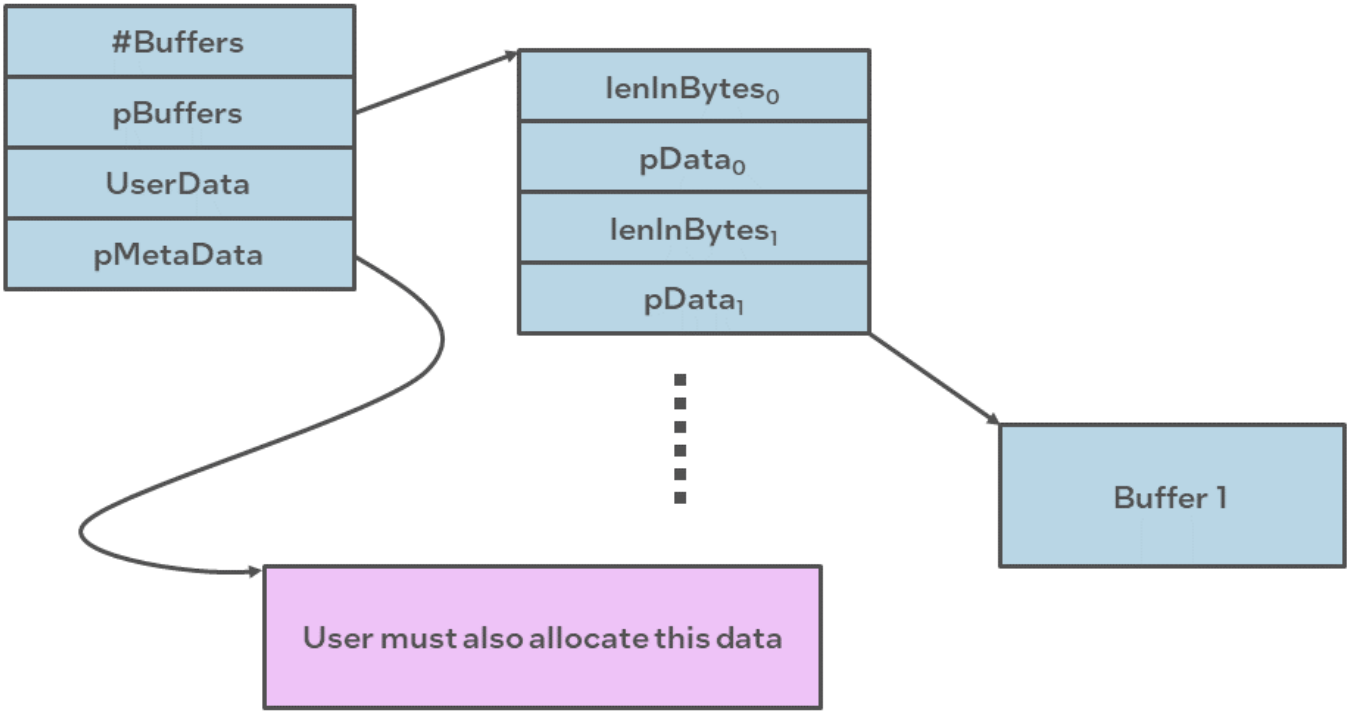
For data plane APIs (`cpa_sym_dp.h` and `cpa_dc_dp.h`) a region of memory that is not physically contiguous is described using the `CpaPhysBufferList` structure. This is similar to the `CpaBufferList` structure; the difference, in this case, the individual flat buffers are represented using physical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

3.3.5 Huge Pages

The included User space DMAable Memory driver `usdm_drv.ko` supports 2MB pages. This allows direct access to main memory by devices other than the CPU and the actual supported maximum memory size in one individual allocation when huge pages is enabled is 2MB - 5KB. Where the 5KB is used for memory management for the memory driver. The use of 2MB pages provides benefits, but also requires additional configuration. Use of this capability assumes that a sufficient number of huge pages are allocated in the operating system for the particular use case and configuration.

Here are some example use cases:

- Default settings applied:




```
modprobe usdm_drv.ko
```

- Set maximum amount of Non-uniform Memory Access (NUMA) type memory that the User Space DMAable Memory (USDM) driver can allocate to 32MB for all processes. Huge pages are disabled:

```
modprobe usdm_drv.ko max_mem_numa=32768
```

- Set maximum number of huge pages that the USDM can allocate to 50 in total and 5 per process:

```
modprobe usdm_drv.ko max_huge_pages=50 max_huge_pages_per_process=5
```

Note: This configuration works for up to the first 10 processes.

Here are examples of invalid use cases to avoid:

- This is erroneous configuration, maximum number of huge pages that USDM can allocate is 3 totals: 3 for a first process, 0 for the next processes:

```
insmod ./usdm_drv.ko max_huge_pages=3 max_huge_pages_per_process=5
```

- This command results in huge pages being disabled because `max_huge_pages` is 0 by default:

```
insmod ./usdm_drv.ko max_huge_pages_per_process=5
```

- This command results in huge pages being disabled because `max_huge_pages_per_process` is 0 by default:

```
insmod ./usdm_drv.ko max_huge_pages=5
```

Note: The use of huge pages may not be supported for all use cases. For instance, depending on the driver version, some limitations may exist for an Input/Output Memory Management Unit (IOMMU).

3.4 Modes of Operation

3.4.1 Calling Semantics

3.4.1.1 Asynchronous (Polled)

Hardware "request/response" interface is inherently asynchronous (non-blocking).

- Calling function returns once request submitted.
- Callback invoked when response available (polled).

3.4.1.2 Asynchronous (Interrupts)

Hardware "request/response" interface is inherently asynchronous (non-blocking).

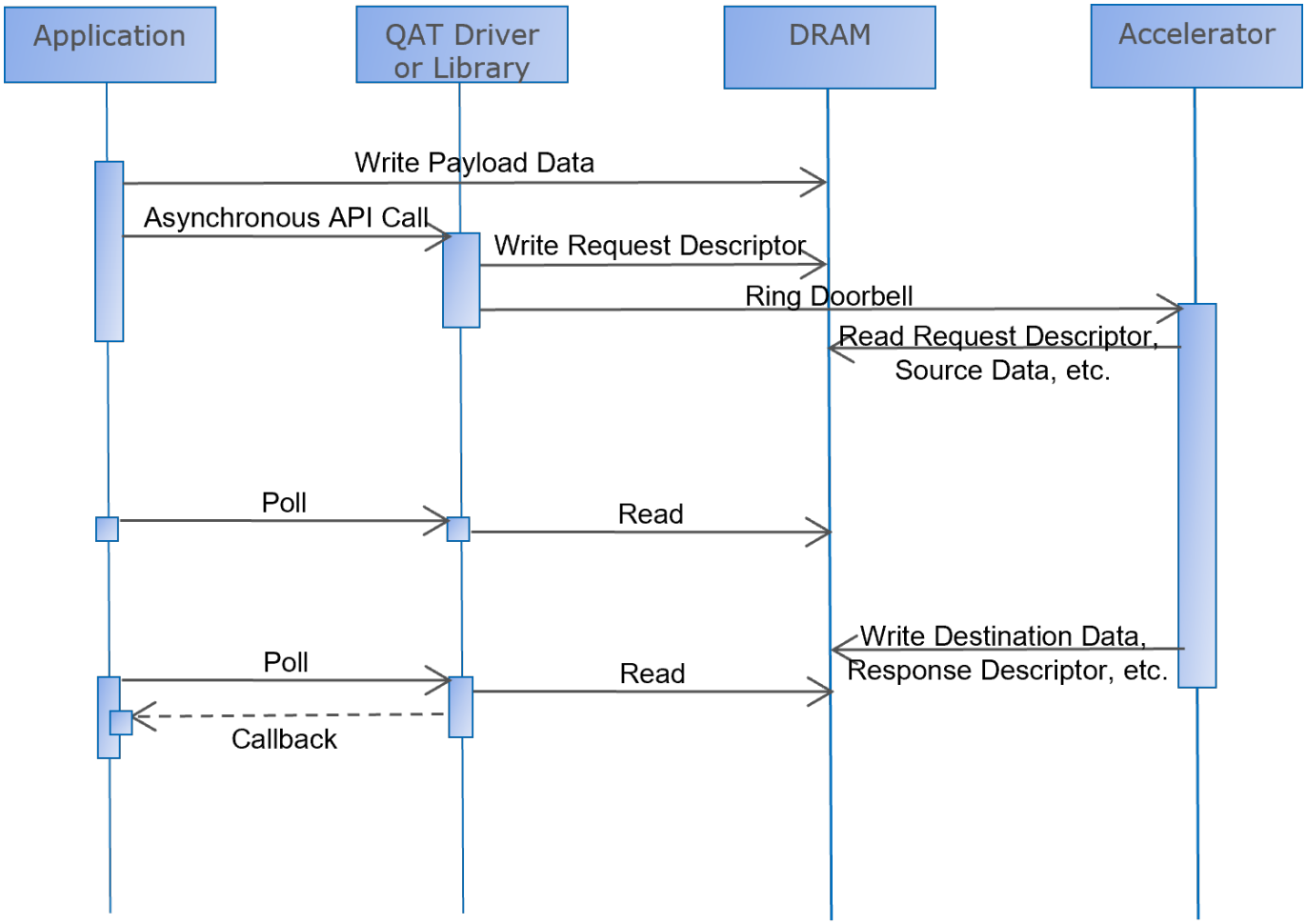
- Calling function returns once request submitted.
- Callback invoked when response available (interrupt-driven).

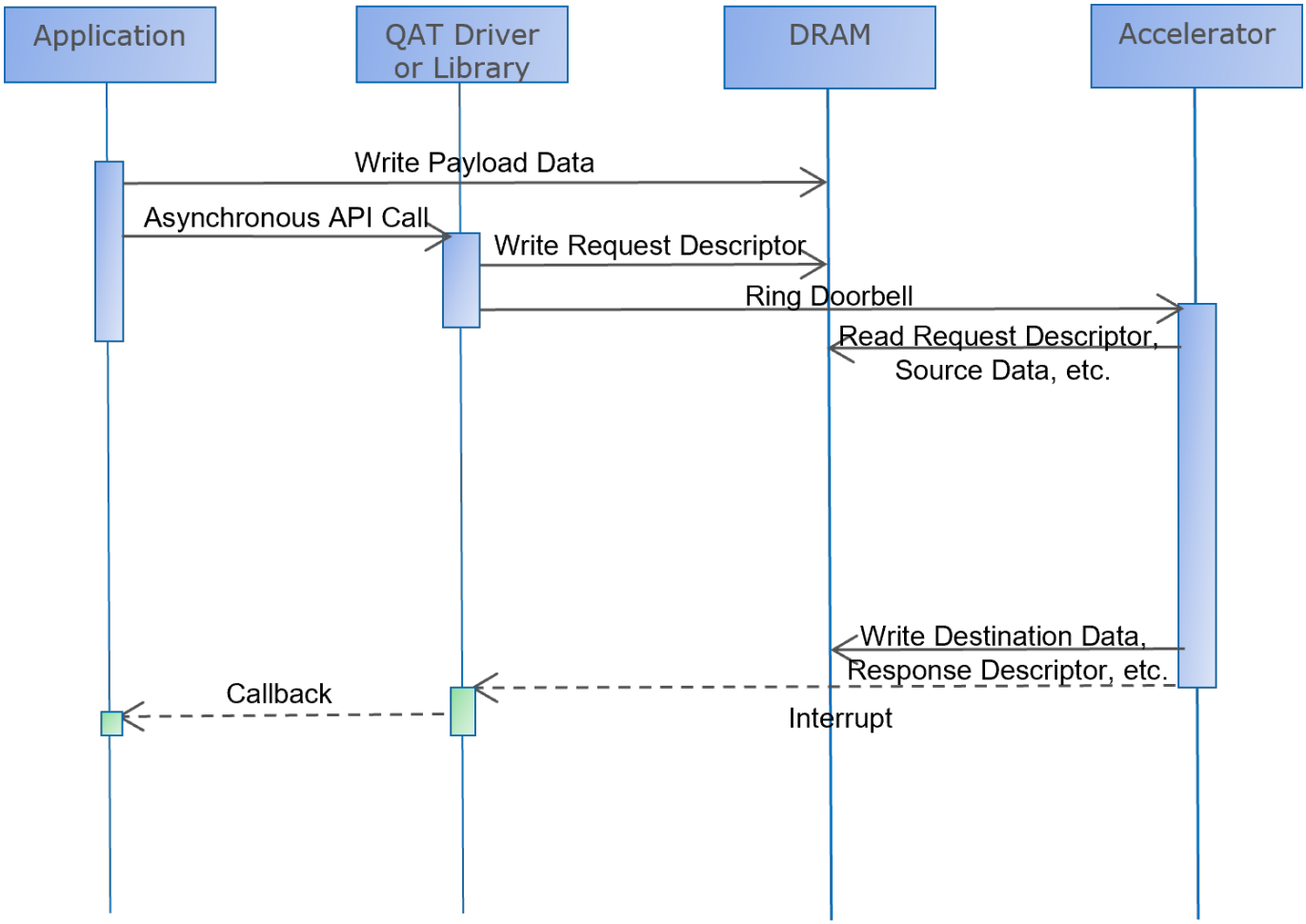
3.4.1.3 Synchronous

Software interface is traditionally synchronous (blocking).

- Calling function blocks until response available.
- Can be implemented "on top of" asynchronous hardware semantics.

3.4.1.4 Pros And Cons


Application
QAT Driver or Library
DRAM
Accelerator
Write Payload Data
Asynchronous API Call
Write Request Descriptor
Ring Doorbell
Read Request Descriptor, Source Data, etc.
Poll
Read
Write Destination Data, Response Descriptor, etc.
Poll
Read
Callback


Application
QAT Driver or Library
DRAM
Accelerator
Write Payload Data
Asynchronous API Call
Write Request Descriptor
Ring Doorbell
Read Request Descriptor, Source Data, etc.
Write Destination Data, Response Descriptor, etc.
Interrupt
Callback

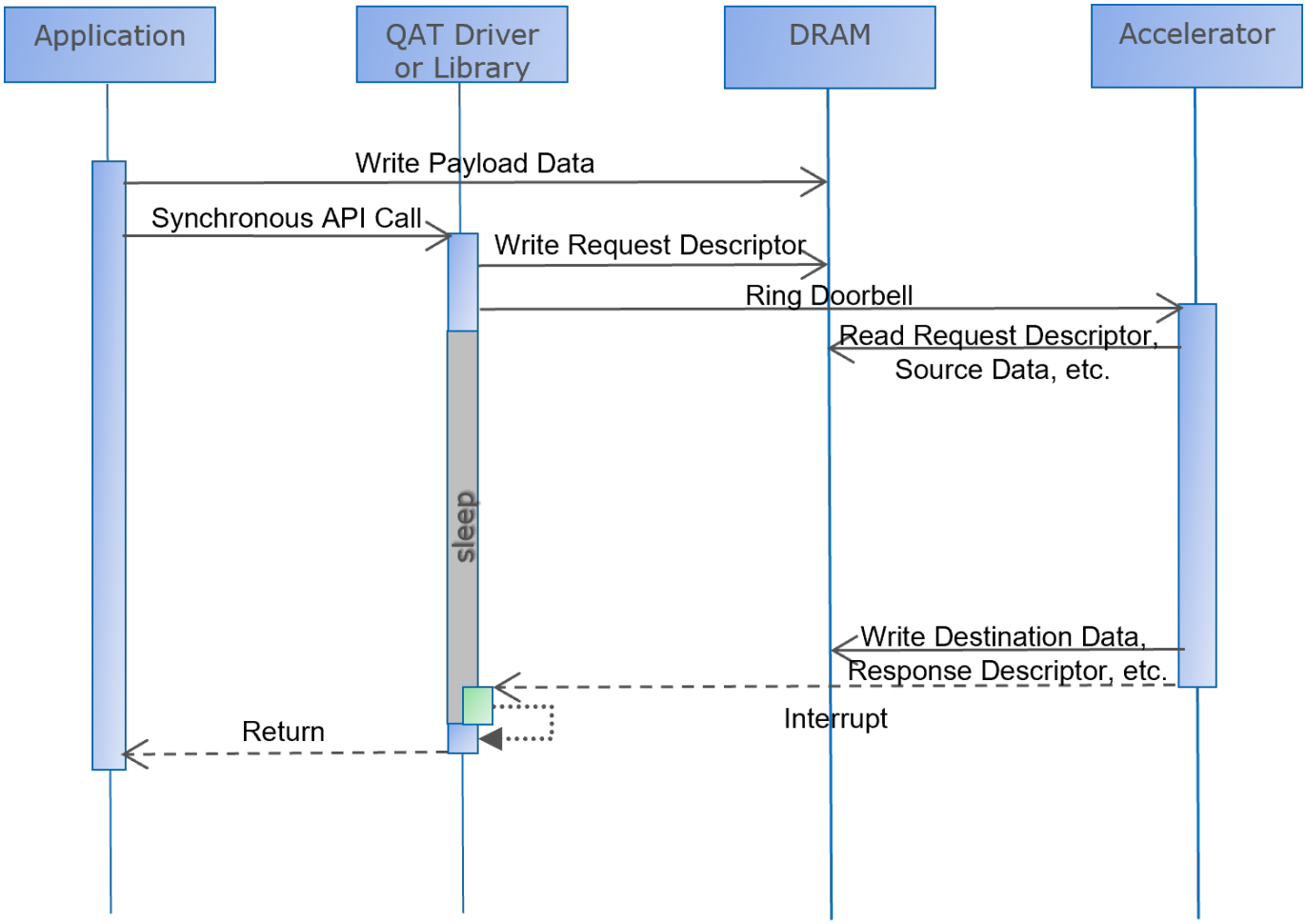

Application
QAT Driver or Library
DRAM
Accelerator
Write Payload Data
Synchronous API Call
Write Request Descriptor
Ring Doorbell
Read Request Descriptor, Source Data, etc.
sleep
Write Destination Data, Response Descriptor, etc.
Interrupt
Return

Table 2: Pros and cons of modes of operation

	Asynchronous	Synchronous
CPU Utilization	Software thread can do other things while hardware processes job, without need for expensive context switch.	Software thread blocked or idle awaiting response. Can use multi-threading, but context switching can be expensive.
Acceleration Utilization	A single software thread can have multiple requests outstanding, keeping multiple accelerator engines.	Hardware has at most one request outstanding per CPU/software thread, remaining threads are idle.
Ease of Use	Can be difficult if application is designed to use synchronous APIs.	Easier to integrate if application is designed to use synchronous APIs.

Note: Asynchronous API tends to be optimal for performance, but harder to integrate.

3.5 Load Balancing

3.5.1 Per Endpoint

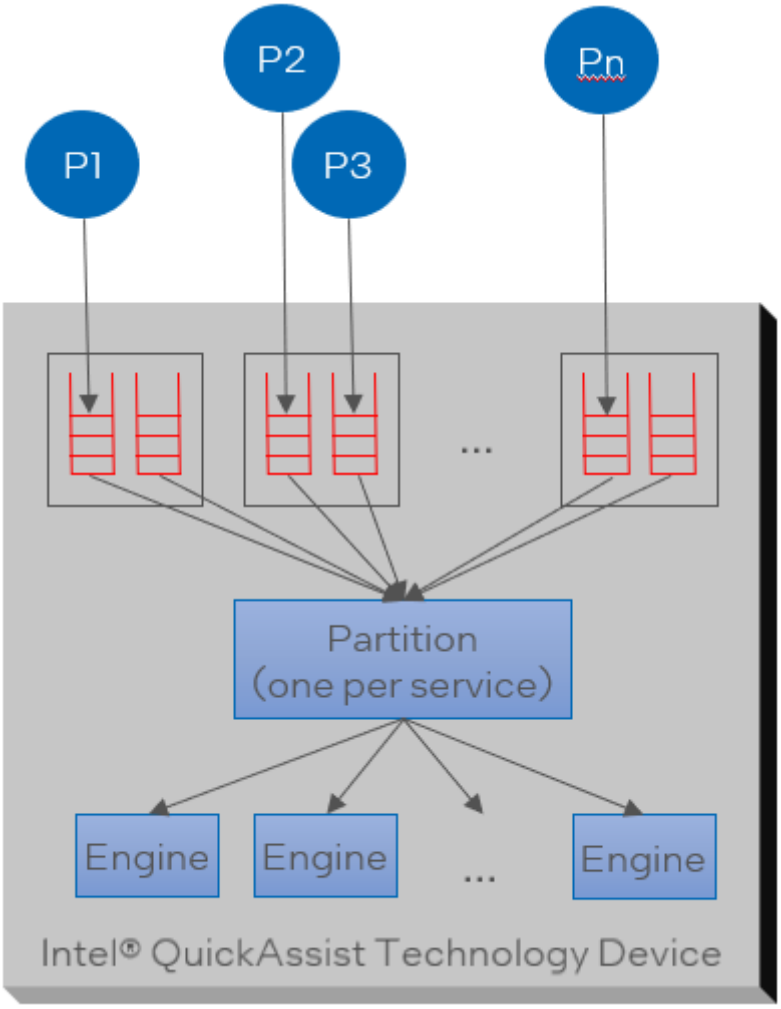
There are four arbiters, which by default are used for the different services (with one spare/unused).

Each partition:

- Arbitrates across two request queues per bundle/VF, to pick a request.
- Load balances all of these requests across all available “engines”.

Within a partition, arbitration uses round robin.

- Ensures fairness (in terms of number of requests) across queue pairs and guests


P1
P2
P3
Pn
...
Partition
(one per service)
Engine
Engine
...
Engine
Intel® QuickAssist Technology Device

3.5.2 Across Endpoints

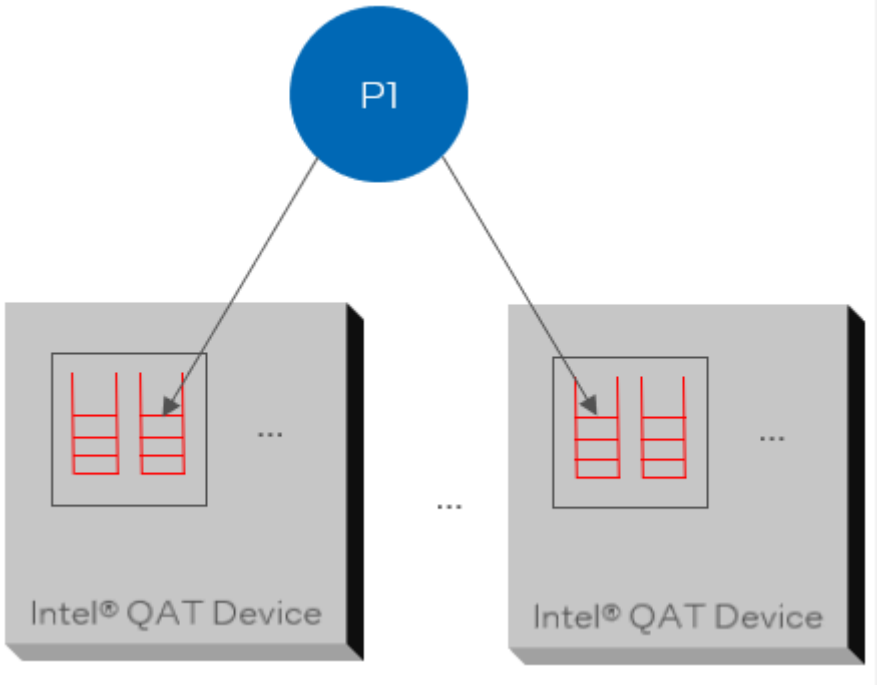
In a platform or CPU with multiple Intel® QAT devices, software is responsible for load sharing across devices/endpoints. Sapphire Rapids has up to four Intel® QAT devices/endpoints in a single CPU package (depending on SKU) PCIe card may have multiple (QAT 1.x) devices across one or more chipsets.

Software-based Load Sharing can be implemented at various layers:

- For applications using the Intel® QuickAssist Technology API, the application must implement load balancing.
- For applications using a framework (e.g. OpenSSL), the framework implements load balancing.

3.5.2.1 Load Sharing Criteria

- Simple round-robin scheme recommended.
- May want to consider "locality" in a multi-socket (NUMA) platform.



3.5.3 Dimensions

Table 3: Dimensions Gen 1 & Gen 2

	Gen 1	Gen 2	
	Intel® Communication Chipset 8925 to 8955 Series	Intel® C62x Chipset	Intel Atom® Processor C3000
Number PCIe End-points	1	3	1
Number of Bundles /VFs per Endpoint	32	16	16
Number of Queue Pairs per Bundle	8		

Table 4: Dimensions Gen 3 & Gen 4

	Gen 3	Gen 4
	Intel® Atom P5000 Processor/ Ice Lake D	Intel® 4th Gen Intel® Xeon® Scalable Processor (per socket)
Number PCIe End-points	1	4
Number of VFs per Endpoint	128	16
Number of Queue Pairs per Bundle	8	4
Number of s-IOV ADIs per Endpoint	N/A	64

3.6 Debugability

3.6.1 Overview of Intel® QAT debugfs entries

Some useful debugging information for the driver and configuration is available via the Linux debugfs file system, with the entries `/sys/kernel/debug/qat_*` and `/sys/kernel/debug/qae_mem_dbg/qae_mem_slabs`.

3.6.1.1 Entries in /sys/kernel/debug/qat_*

This includes:

Table 5: Intel® QuickAssist Technology /sys/kernel/debug Entries

Entry	Description	Supported Platforms
`cnv_errors`	Indicates number of *compressAndVerify* errors. Refer to *Compress and Verify Error log in Sysfs*.	All
`dev_cfg`	Displays internal device configuration information.	All
`frequency`	Displays frequency of Acceleration Engines.	All
`fw_counters`	Displays Acceleration Engine firmware requests/responses.	All
`heartbeat`, `heartbeat_failed`, `heartbeat_sent`	Refer to *System Virtual Files*.	All
`pm_status`	Displays power management status. Refer to *Power Management* for additional information.	QAT 2.0
`transport`	Contains firmware request/response data. Available only for kernel space instances.	All
`version`	Includes package version information.	All
`vqat`	Contains sIOV Virtual QAT device details. Refer to *Scalable IOV* for additional information.	QAT 2.0

3.6.1.2 Memory driver queries (qae_mem_slabs)

Debug features are also available by reading and writing the file `/sys/kernel/debug/qae_mem_dbg/qae_mem_slabs` . When *reading* the virtual/physical `address`, `size` and `slab id` together with the `pid` of the allocating process are shown. *Writing* a string to the file will start executing debug commands.

For example:

```
# cat /sys/kernel/debug/qae_mem_dbg/qae_mem_slabs
Pid 78854, Slab Id 10550771712
Virtual address 000000000b39412d, Physical Address 274e00000, Size 2097152
Pid 78854, Slab Id 10309599232
Virtual address 000000003670dd45, Physical Address 266800000, Size 2097152
...
```

There are three commands supported, and the below table shows their output:

Table 6: Read/Write to /sys/kernel/debug/qae_mem_dbg/qae_mem_slabs

Writing these strings...	...will output this when the file is read
d <pid> <virtual or physical address>	The 256 byte in hex and ascii from the start address
"c <pid> <slab id>" (pid should be the process id that can be obtained by a previous read)	The allocation bit map for the given slab identifier
"t"	Total size of NUMA memory allocated in kernel space

For example, by combining a write to the file and a subsequent read, you can see the total allocated NUMA memory, e.g.:

```
# echo "t" > /sys/kernel/debug/qae_mem_dbg/qae_mem_slabs ; cat /sys/kernel/debug/qae_mem_dbg/
↪qae_mem_slabs
Total allocated NUMA memory: 142606336 bytes
```

As above, the "d" and "c" commands will output their respective information.

3.7 Heartbeat

Under some circumstances, firmware in the Intel® QAT devices could become unresponsive, requiring a device reset to recover. The Intel® QAT Heartbeat feature provides a mechanism for the customer application to detect and reset unresponsive devices. It also notifies the application processes of the start and end of the reset operation and suspends all Intel® QAT instances between the events.

3.7.1 Heartbeat Operation

A Heartbeat-enabled Intel® QAT device firmware periodically writes counters to a specified physical memory location. A pair of counters per thread is incremented at the start and end of the main processing loop within the firmware. Checking for Heartbeat consists of checking the validity of the pair of counter values for each thread. Stagnant counters indicate a firmware hang.

3.7.1.1 Initialization

At startup, the Intel® QAT device driver allocates memory for the counter pairs to be written by the firmware and then sends a message to the firmware to start the Heartbeat functionality.

3.7.1.2 Heartbeat Monitoring

Heartbeat check/monitoring refers to invocation of one of the two API calls that checks if the device is responsive. Heartbeat failure refers the API returning failure.

The Intel® QAT driver does not monitor for Heartbeat. It should be initiated by a Heartbeat management thread calling one of the following APIs periodically:

- `icp_sal_check_device(Cpa32U accelId)`
- `icp_sal_check_all_devices(void)`

A failure return code implies the device has failed or hung.

The Heartbeat management thread should satisfy the following conditions:

- For any given device, only one such process/thread should monitor.
- One process can monitor one or more devices.
- It can be a user application that uses Intel® QAT services, or a separate management/control plane process.
- In virtualized environment, monitoring process(es)/thread(s) must run in the context of the host or hypervisor.

3.7.1.3 Resetting a Failed Device

A device can be configured for automatic reset by the Intel® QAT framework or manually reset by the application by using the `AutoResetOnError` field in the device configuration file `/etc/<device>.conf`, as shown below.

Table 7: AutoResetOnError Values

AutoResetOnError Value	Action on Heartbeat Failure
0 (default)	Do not reset the device
1	Reset the device automatically

If an Intel® QAT device is not configured for automatic reset, the management thread should reset it using the `icp_sal_reset_device(Cpa32U accelId)` API.

The `icp_sal_reset_device()` function starts an asynchronous reset sequence and returns immediately. The reset function should not be called again until the device has completed the reset to avoid a reset storm. The `icp_sal_check_device(<device id>)` function could be called in a loop to check if the device reset is still in progress.

If the application devices are all configured for automatic reset then the `icp_sal_check_all_devices()` function could be used; otherwise, the function should not be used because it does not return the identity of the failed device, which is a required parameter for the `icp_sal_reset_device()` function.

Function Signatures

The details of the above functions, parameters, and return values can be found in *Supported APIs > Additional APIs*.

3.7.2 Incorporating Heartbeat into Intel® QAT Applications

A typical Intel® QAT user application consists of two tasks:

- The first task is typically an application thread that initializes Intel® QAT instances and sessions, and then submits service requests for Intel® QAT crypto or compression.
- If an application employs polling to receive Intel® QAT service responses, then this task is also an application thread. Alternatively, responses are received as an interrupt handler.

Two more tasks are required to support Heartbeat:

- The first is a management task to monitor the devices for failure or hang and then resets them, when required. As discussed earlier, this could be an application thread of an independent management process.
- The second task is an application thread that polls for device reset events:
  - Device is restarting: `CPA_INSTANCE_EVENT_RESTARTING`
  - Device restart is complete: `CPA_INSTANCE_EVENT_RESTARTED`

If the application employs polling to receive Intel® QAT service responses, then this task could be included in the same polling loop.

The polling for device events is done using the API: `icp_sal_poll_device_events()`.

The two callback functions for crypto and compression are registered using the following APIs:

- `cpaCyInstanceSetNotificationCb`
- `cpaDcInstanceSetNotificationCb`

The details of the above functions, parameters, and return values can be found in *Supported APIs > Additional APIs*.

3.7.3 Restart Sequence

During the restart sequence, the user space library releases the memory used for rings and other data structures as part of the shutdown and reallocates them when the restart is completed. This is transparent to the user application, so it can continue to use the same logical instance after reset to submit Intel® QAT service requests. Any memory allocated by the user application for the Intel® QAT service is untouched during device reset.

A typical Heartbeat error use-case is as follows:

1. The driver and the firmware is loaded, initialized and started.

2. The user-space application registers to receive instance notifications by calling `cpaCyInstanceSetNotificationCb` and `cpaDcInstanceSetNotificationCb`.
3. The management thread monitors for the device's heartbeat. When a device is unresponsive, a device reset is initiated by this thread or by the Intel® QAT framework depending on the device configuration.
4. The kernel-space process sends the restarting event to the user-space process.
5. The user-space driver passes the device restarting event to all the registered application instances. It also frees memory and rings associated with the registered instances.
6. The kernel-space driver triggers the device reset.
7. During reset, the Intel® QAT service request made by the user application returns one of:
   - `CPA_STATUS_FAIL`
   - `CPA_STATUS_RETRY`
   - `CPA_STATUS_RESTARTING`
8. When the device reset is complete, the kernel-space driver sends a device restarted event to the user space driver.
9. The user space driver allocates the memory and rings and then forwards the device Restarted event to each of the registered instances.

3.7.4 Status of Packets in Flight (Crypto Applications Only)

When a device has fatal errors, the application ordinarily cannot determine whether or not inflight requests have been processed successfully.

The current Intel® QAT release includes a dummy response feature that creates mock responses to all requests submitted during a fatal error condition, so the application can detect them and, therefore, know which requests need to be resubmitted to the available devices or to the software.

Note: The sequence of dummy responses will match the sending request sequence for all requests submitted during a fatal error.

Since the dummy response feature only supports Public Key Encryption (PKE), dummy responses may be generated only when the `icp_sal_CyPollInstance()` function is called, since it is the function for crypto services.

The `icp_sal_poll_device_events()` function should also be called by the application, so that the application get a notification when the device encounters a failure and dummy responses are generated when calling `icp_sal_CyPollInstance()` for the inflight requests.

3.7.5 Determining Device ID

The *<device id>* that is passed as a parameter to several Heartbeat API is the numeric suffix of the device name displayed by the following command. (device name: *qat_dev0*):

```
service qat_service status
```

The output will look like:

```
There is 1 QAT acceleration device(s) in the system: qat_dev0 - type: c3xxx, inst_id: 0, node_
↪id: 0, bsf: 01:00.0, #accel: 3 #engines: 6 state: up
```

The Intel® QAT library has no API to discover the device number easily. However, an application can use the IOCTLs `IOCTL_GET_NUM_DEVICES` and `IOCTL_STATUS_ACCEL_DEV` to find the device_id of a particular device if they know the Bus Device Function (BDF). Refer to `perform_query_dev()` in `./adf_ctl.cpp`.

3.7.6 Testing Heartbeat

Two debug capabilities are available to assist the developers incorporating Heartbeat into their applications:

- Simulation of Heartbeat failure.
- System virtual files under `/sys/kernel/debug/`.

3.7.6.1 Simulated Heartbeat Failure Configuration

The Heartbeat feature is always enabled in the package. However, a debug capability that simulates device failure can be enabled during the configure step as follows:

```
./configure --enable-icp-hb-fail-sim
```

3.7.6.2 Simulating Heartbeat Failure

Simulating Heartbeat failure can be accomplished using two methods:

- Using the API `icp_sal_heartbeat_simulate_failure(<device id>)`.
- Executing the command:

```
cat /sys/kernel/debug/<device>/heartbeat_sim_fail
```

System Virtual Files

Note: The heartbeat `/sys/kernel/debug` files are associated with the QAT Physical Function (PF).

The Heartbeat feature implements the following system virtual files under the `/sys/kernel/debug/qat_<device>_<your_device_BDF>/` directory.

Table 8: Heartbeat System Virtual Files

File	Content
`heartbeat`	0: Device is responsive. -1: Device is NOT responsive.
`heartbeat_failed`	Number of times the device became unresponsive.
`heartbeat_sent`	Number of times the control process checked if the device is responsive.

A developer could simulate the Heartbeat management process by running the following script in the background:

```
#!/bin/bash
while : do
   cat /sys/kernel/debug/<device>/heartbeat > /dev/null sleep 1
done
```

Heartbeat Polling Frequencies

The application developer should decide on the following two Heartbeat polling frequencies:

- Device Heartbeat monitoring.
- Checking for device reset events.

Device Heartbeat Monitoring

Consider the following points when determining the frequency of Heartbeat monitoring:

- Increasing Heartbeat monitoring frequency will minimize the customer's system downtime.
- However, since device unresponsiveness should be an infrequent event, high frequency Heartbeat monitoring wastes CPU cycles.
- Also, if there are large Intel® QAT service requests that take some time to complete, high frequency Heartbeat monitoring could result in false reports of unresponsiveness.
- With QAT Gen4 devices, heartbeat update timer in firmware is a constant value of 200ms (unconfigurable). With QAT Gen2 devices this value is configurable with configuration item **HeartbeatTimer** (the default value is 500ms and the minimal allowed value is 200ms)
- For both QAT Gen2 and Gen4 monitoring interval should be larger or equal than the Heartbeat update interval. (e.g. if user configure HeartbeatTimer=300, polling interval should be >=300ms)

Checking for Device Reset Events

If the application uses polling for reading Intel® QAT service responses, there is no value in checking for resets more frequently. Since device unresponsiveness is an infrequent occurrence, frequency of checking for reset events could be a fraction of the frequency of polling for Intel® QAT service responses.

3.7.7 Handling Device Failures in a Virtualized Environment

The Heartbeat feature in the acceleration software can be used in a virtualized environment. Refer to the *Using Intel® Virtualization Technology (Intel® VT) with* Intel® *QuickAssist Technology Application Note* for more details on enabling SR-IOV and the creation of Virtual Functions (VFs) from a single Intel® QuickAssist Technology acceleration device to support acceleration for multiple Virtual Machines (VMs).

The following sequence describes a possible use case for using the Heartbeat feature in a virtualized environment.

1. The Intel® QAT Physical Function driver (PF driver) isloaded, initialized and started.
2. The Intel® QAT Virtual Function driver (VF driver) is loaded, initialized and started in the Guest OS in the VM.
3. The PF driver detects that the firmware is unresponsive (using either of the following methods: User Proc Entry Read (not Enabled by Default) on page 47 or User Application Heartbeat APIs (not Enabled by Default) on page 48).
4. The PF driver sends the "Restarting" event message to the VF via the internal PFto-VF communication messaging mechanism.
5. The VF driver sends the "Restarting" event to the application's registered callback. The callback is registered using either of the Intel® QAT API functions `cpaDcInstanceSetNotificationCb()` or `cpaCyInstanceSetNotificationCb()` in the Guest OS. (The application's callback function may perform any application-level cleanup.)
6. The PF driver starts the reset sequence (save state, initiate reset, and restore state).
7. The user restarts the Guest OS and loads the VF driver and application in the Guest OS.

Note:

- If the Heartbeat feature in the acceleration software is not enabled, the PF driver will not notify the VF driver that the firmware is unresponsive.
- The error detection mechanisms are not available on the VF driver in the VM, but device errors caused by any of the software running on the VM will be detected by the PF driver using the above mechanisms.

3.7.8 Incorporating Dummy Responses into an Intel® QAT Application

The dummy response feature has been incorporated in a scenario with the Intel® QAT engine and Nginx*. Figure below illustrates how it works. This can be used as a reference to so-called "software fallback."

The Intel® QAT engine is a shim layer between OpenSSL* libcrypto* and Intel® QAT Library. The Intel® QAT Library will generate failover responses.

The Heartbeat Monitoring Daemon, a single process, is a daemon which is used to check the device status periodically and trigger the driver the reset the device when heartbeat failure happens. Its only activity is calling `icp_sal_check_device()` or `icp_sal_check_all_devices()` periodically.

The Intel® QAT Engine polls for and handles "device error" and "device ok" events (via udev). It keeps track of the number of devices which are active.

- If some, but not all, Intel® QAT devices encounter errors, switch to remaining available devices by resubmitting the inflight requests, which are responded to with dummy responses and new requests to the available devices.
- If the number of active Intel® QAT devices goes to zero, switch to software and resubmit the inflight requests which are responded to with dummy responses and new requests to the software.
- If the number of active Intel® QAT devices goes positive again, switch back to hardware.

3.8 Telemetry

The telemetry feature is a tool to view the performance and utilization of an acceleration device. Telemetry data can be viewed on a per device and a per ring pair (also known as *queue pair*) basis.

3.8.1 Telemetry Usage

The telemetry feature is configured and queried using sysfs files in the Linux filesystem.

The telemetry sysfs folder is located at `/sys/devices/pciAAAA:BB/AAAA:BB:CC.D/telemetry` where:

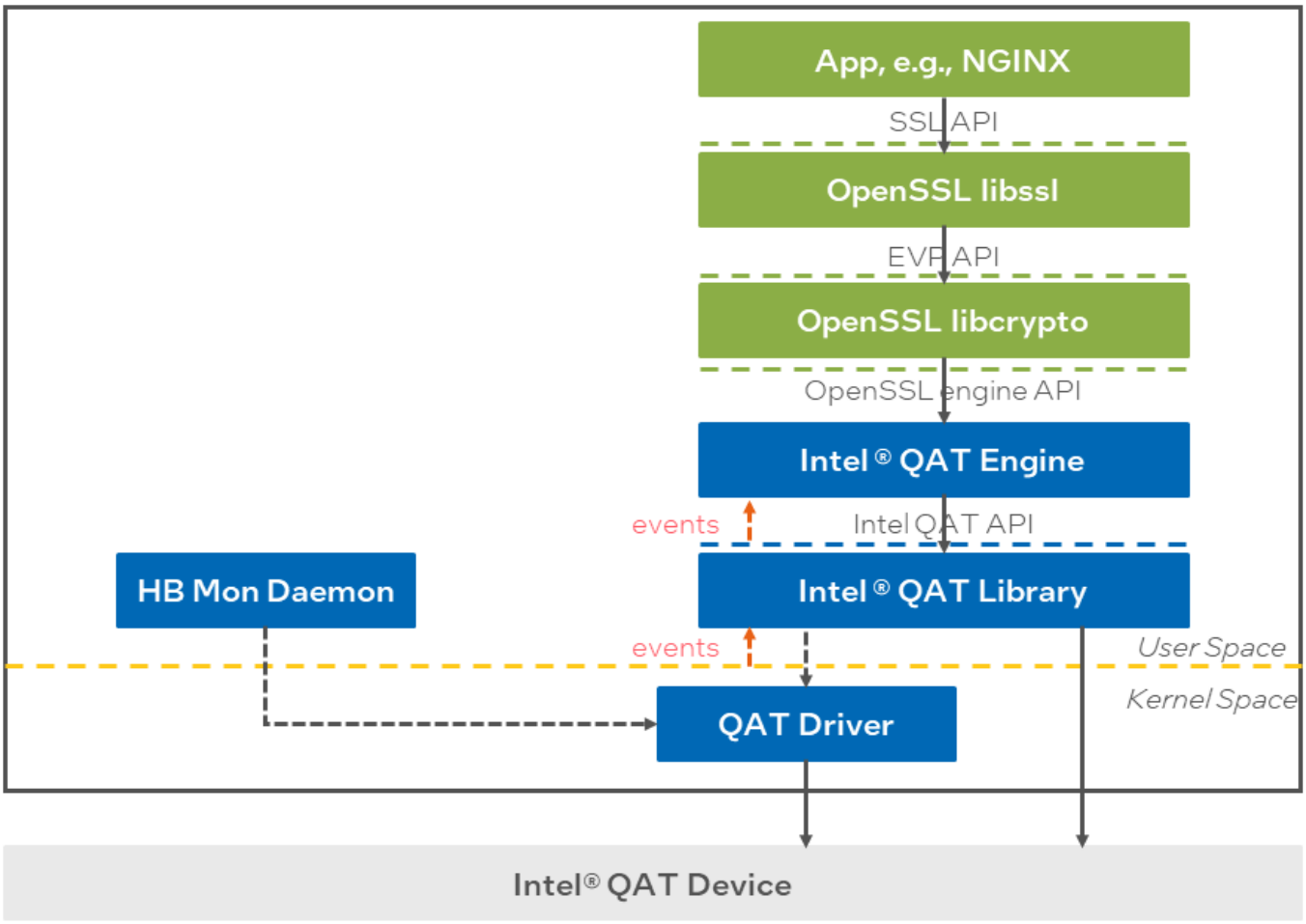
- `AAAA:BB:CC.D` is the *Domain:BDF* of the target Intel® QAT Endpoint.

Example:

```
ls /sys/devices/pciAAAA:BB/0000:6b:00.0/telemetry
```

The telemetry feature is controlled with standard linux file commands into the control file as outlined below. The telemetry data is accessed through the `device_data` or `rp_<X>_data` file depending on what data is required.

The telemetry data for device level and ring pair level is updated each second.


App, e.g., NGINX
SSL API
OpenSSL libssl
EVP API
OpenSSL libcrypto
OpenSSL engine API
Intel® QAT Engine
events
Intel QAT API
HB Mon Daemon
Intel® QAT Library
events
User Space
Kernel Space
QAT Driver
Intel® QAT Device

3.8.2 Telemetry Control

Device level telemetry is enabled by echoing `1` into the control file and disabled by echoing `0`. Reading the control file will tell whether the feature is currently enabled or disabled.

Ring Pair level telemetry is enabled when device level telemetry is enabled. However the ring pairs need to be selected. Only 4 ring pairs can be shown at any given time. By echoing the number of the ring pair (0-63) into a `rp_<X>_data` file it can be selected. Where X is A,B,C or D.

3.8.2.1 Telemetry Commands

Table 9: Telemetry Commands

Operation	Command
Enable Telemetry	`echo 1 > control`
Disable Telemetry	`echo 0 > control`
Query Telemetry data	`cat device_data`
Select Ring Pairs	`echo Num > rp_<X>_data, Num is the ring pair to be selected`
Query Ring Pair data	`cat rp_<X>_data`

Selecting Ring Pairs

This section provides guidance on the mapping of ring pairs to the VFs for the PF. There are 4 Ring Pairs per VF. The Ring Pairs for a PF looks like the following:

Table 10: Ring Pairs

VF	Ring Pairs			
1	0	1	2	3
2	4	5	6	7
3	8	9	10	11
4	12	13	14	15
5	16	17	18	19
6	20	21	22	23
7	24	25	26	27
8	28	29	30	31
9	32	33	34	35
10	36	37	38	39
11	40	41	42	43
12	44	45	46	47
13	48	49	50	51
14	52	53	54	55
15	56	57	58	59
16	60	61	62	63

The `ServicesEnabled` defined for the PF control the mapping of the Ring Pairs:

- If only one workload is enabled (`dc/sym/asym`), the first two columns are used for this service.
- If `dc` and `sym` or `asym` is enabled, the first two columns are for `sym` or `asym` and the second two columns are for `dc`
- If `sym` and `asym` is enabled, the first and third columns are for `asym` and second and fourth columns are for `sym`.

3.8.2.2 Device Level Telemetry Values

Table 11: Device Level Telemetry Values

Value	Meaning
`sample_cnt`	Message count, counter.
`pci_trans_cnt`	PCIe Partial Transactions, counter.
`max_rd_lat`	Max Read Latency, nanoseconds.
`rd_lat_acc_avg`	Average Read Latency, nanoseconds.
`max_lat`	Max Get To Put latency, nanoseconds.
`lat_acc_avg`	Average Get To Put latency, nanoseconds.
`bw_in`	PCIe write bandwidth, Mbps.
`bw_out`	PCIe read bandwidth, Mbps.
`at_page_req_lat_acc_avg`	Average Page Request Latency, nanoseconds.
`at_trans_lat_acc_avg`	Average Translation Latency, nanoseconds.
`at_max_tlb_used`	Maximum uTLB Consumed, counter.
`util_cpr<x>`	Compression Slice Utilization On Slice X, percentage execution cycles.
`util_dcpr<x>`	Decompression Slice Utilization On Slice X, percentage execution cycles.
`util_xlt<x>`	Translator Slice Utilization On Slice X, percentage execution cycles.
`util_cph<x>`	Cipher Slice Utilization On Slice X, percentage execution cycles.
`util_ath<x>`	Authentication Slice Utilization On Slice X, percentage execution cycles.
`util_ucs<x>`	UCS Slice Utilization On Slice X, percentage execution cycles.
`util_pke<x>`	PKE Slice Utilization On Slice X, percentage execution cycles.

3.8.2.3 Ring Pair Level Telemetry Values

Table 12: Ring Pair Level Telemetry Values

Value	Meaning
`sample_cnt`	Message count, counter.
`rp_num`	Number of the ring pair returning data.
`pci_trans_cnt`	PCIe Partial Transactions, counter.
`lat_acc_avg`	Average Get To Put latency, nanoseconds.
`bw_in`	PCIe write bandwidth, Mbps.
`bw_out`	PCIe read bandwidth, Mbps.

continues on next page

Table 12 – continued from previous page

Value	Meaning
`at_glob_devtlb_hit`	Descriptor DevTLB hit rate per ring, counter.
`at_glob_devtlb_miss`	Descriptor DevTLB miss rate per ring, counter.
`tl_at_payld_devtlb_hit`	Payload DevTLB hit rate per ring, counter.
`tl_at_payld_devtlb_miss`	Payload DevTLB miss rate per ring, counter.

3.8.3 Monitoring Telemetry - Text Based

The following example Python script highlights how telemetry data can be monitored at the command line. The script first enables telemetry service for each QAT endpoint that supports telemetry and is in the **up** state. It then queries the telemetry data on a periodic basis collecting the data and formatting the display.

```
Intel(R) QuickAssist Device Utilization

Device        %Comp   %Decomp %PKE    %Cipher %Auth   %UCS    Latency(ns)
================================================================================
qat_dev0      100     45      0       0       0       0       174332
qat_dev1      100     45      0       0       0       0       174276
qat_dev2      100     45      0       0       0       0       174266
qat_dev3      100     45      0       0       0       0       174264
qat_dev4      100     42      0       0       0       0       182140
qat_dev5      100     43      0       0       0       0       181970
qat_dev6      100     42      0       0       0       0       181984
qat_dev7      100     43      0       0       0       0       181892
================================================================================
```

Important: When running script as non-root User, ensure **adf_ctl** is added to **qat** group.

```
sudo chgrp qat /usr/local/bin/adf_ctl
```

Script can be downloaded from **here**

```
#!/usr/bin/env python

import time
import curses
import subprocess
import re


devices=[]
```

(continues on next page)

(continued from previous page)

```
paths=[]
set_paths=[]

def EnableTelemetry():

        devices.clear()
        command = "adf_ctl status"
        sp = subprocess.Popen(command,shell=True,stdout=subprocess.PIPE,stderr=subprocess.
↪PIPE,universal_newlines=True)

        # Store the return code in rc variable
        rc=sp.wait()

        # Separate the output and error
        # This is similar to Tuple where we store two values to two different variables
        out,err=sp.communicate()

        # Split string into list of strings
        output_adf = out.split()

        paths.clear()
        command = 'find /sys/devices/ -name "telemetry"'
        sp = subprocess.Popen(command,shell=True,stdout=subprocess.PIPE,stderr=subprocess.
↪PIPE,universal_newlines=True)

        # Store the return code in rc variable
        rc=sp.wait()

        # Separate the output and error.
        # This is similar to Tuple where we store two values to two different variables
        out,err=sp.communicate()

        # Split string into list of strings
        original_array= out.split()
        output_telem = sorted(original_array, key=lambda x: (x.split(':')[1], 16))

        i = 0
        state = "down"
        name = None
        bus = None
        telemetry_supported = False

        # Build device list from adf_status output
        while i < len(output_adf):
                if "qat_dev" in output_adf[i]:
                        name = output_adf[i]
                elif "type:" == output_adf[i]:
                        if "4xxx," == output_adf[i+1]:
                                telemetry_supported = True
                elif "bsf:" == output_adf[i]:
                        bus = output_adf[i+1][5:7]
```

(continues on next page)

(continued from previous page)

```
                elif "state:" == output_adf[i]:
                        if "up" == output_adf[i+1]:
                                if telemetry_supported == True:
                                        devices.append((name, bus))

                        # Reset variables to ensure we only attempt to enable telemetery on
↪devices that support telemetry and are in up state
                        state = "down"
                        name = None
                        telemetry_supported = False
                i += 1

        # Build path list from Telemetry search
        i = 0
        for i in range(len(output_telem)):
                paths.append(output_telem[i])

        # Verify Telemetry paths are part of enabled QAT endpoints
        set_paths.clear()
        i = 0
        for path in paths:
                while i < len(devices):
                        if devices[i][1] in path:
                                set_paths.append(path)
                        i += 1
                i = 0

        if len(set_paths) == 0:
           print("No telemetry supported QAT endpoints found... exiting.")
           quit()

        # Enable Telemetry for QAT endpoints
        for path in set_paths:
                control_file_name= path + "/control"
                command = "echo 1 > " + control_file_name

                try:
                        str(subprocess.check_output(command, shell=True))
                except:
                        break

def pbar(window):
        refresh_counter = 0

        while True:

                try:
                        refresh_counter += 1

                        window.addstr(0, 10, "Intel(R) QuickAssist Device Utilization")
                        window.addstr(2, 10, "Device\t%Comp\t%Decomp\t%PKE\t%Cipher\t%Auth\t
↪%UCS\tLatency(ns)")
```

(continues on next page)

(continued from previous page)

```
                        window.addstr(3, 10,
↪"================================================================================")

                        count = 0
                        for device in devices:

                                command = "cat " + set_paths[count] + "/device_data"
                                sp = subprocess.Popen(command,shell=True,stdout=subprocess.
↪PIPE,stderr=subprocess.PIPE,universal_newlines=True)

                                # Store the return code in rc variable
                                rc=sp.wait()

                                # Separate the output and error
                                # This is similar to Tuple where we store two values to two
↪different variables
                                out,err=sp.communicate()

                                # Split string into list of strings
                                output = out.split()

                                i = 0
                                while i < len(output):

                                        if "lat_acc_avg" == output[i]:
                                                latency = output[i+1]
                                        elif "util_cpr0" == output[i]:
                                                compression = output[i+1]
                                        elif "util_dcpr0" == output[i]:
                                                decompression0 = output[i+1]
                                        elif "util_dcpr1" == output[i]:
                                                decompression1 = output[i+1]
                                        elif "util_dcpr2" == output[i]:
                                                decompression2 = output[i+1]
                                        elif "util_pke0" == output[i]:
                                                pke0 = output[i+1]
                                        elif "util_pke1" == output[i]:
                                                pke1 = output[i+1]
                                        elif "util_pke2" == output[i]:
                                                pke2 = output[i+1]
                                        elif "util_pke3" == output[i]:
                                                pke3 = output[i+1]
                                        elif "util_pke4" == output[i]:
                                                pke4 = output[i+1]
                                        elif "util_pke5" == output[i]:
                                                pke5 = output[i+1]
                                        elif "util_cph0" == output[i]:
                                                cph0 = output[i+1]
                                        elif "util_cph1" == output[i]:
                                                cph1 = output[i+1]
                                        elif "util_cph2" == output[i]:
```

(continues on next page)

(continued from previous page)

```
                                        cph2 = output[i+1]
                                elif "util_cph3" == output[i]:
                                        cph3 = output[i+1]
                                elif "util_ath0" == output[i]:
                                        ath0 = output[i+1]
                                elif "util_ath1" == output[i]:
                                        ath1 = output[i+1]
                                elif "util_ath2" == output[i]:
                                        ath2 = output[i+1]
                                elif "util_ath3" == output[i]:
                                        ath3 = output[i+1]
                                elif "util_ucs0" == output[i]:
                                        ucs0 = output[i+1]
                                elif "util_ucs1" == output[i]:
                                        ucs1 = output[i+1]
                                i += 1

                        decompress_utilization = int(decompression0) +
↪int(decompression1) + int(decompression2)
                        if decompress_utilization > 0:
                                decompress_utilization = decompress_utilization / 3
                                decompress_utilization = round(decompress_utilization)
                        pke_utilization = int(pke0) + int(pke1) + int(pke2) +
↪int(pke3) + int(pke4)+ int(pke5)
                        if pke_utilization > 0:
                                pke_utilization = pke_utilization / 6
                                pke_utilization = round(pke_utilization)
                        cph_utilization = int(cph0) + int(cph1) + int(cph2) +
↪int(cph3)
                        if cph_utilization > 0:
                                cph_utilization = cph_utilization / 4
                                cph_utilization = round(cph_utilization)
                        ath_utilization = int(ath0) + int(ath1) + int(ath2) +
↪int(ath3)
                        if ath_utilization > 0:
                                ath_utilization = ath_utilization / 4
                                ath_utilization = round(ath_utilization)
                        usc_utilization = int(ucs0) + int(ucs1)
                        if usc_utilization > 0:
                                usc_utilization = usc_utilization / 2
                                usc_utilization = round(usc_utilization)
                        if int(latency) == 0:
                                window.addstr(4+count, 10, device[0] + '\t0\t0\t0\t0\
↪t0\t00                 ')

                        window.addstr(4+count, 10, device[0] + '\t' + compression + '\
↪t' + str(decompress_utilization) + '\t' + str(pke_utilization) + '\t' + str(cph_
↪utilization) + '\t' + str(ath_utilization) + '\t' + str(usc_utilization) + '\t'+ latency)
                        count += 1
```

(continues on next page)

(continued from previous page)

```
                        window.addstr(4+count, 10,
↪"================================================================================")
                        window.refresh()
                        time.sleep(2)
                        if refresh_counter % 5 == 0:
                                window.clear()
                                EnableTelemetry()

                except KeyboardInterrupt:
                        break
                except:
                        break


if __name__ == "__main__":
        EnableTelemetry()
        curses.wrapper(pbar)
```

3.9 Rate Limiting

Rate Limiting is implemented by monitoring the utilization of the device on a per-VF, per-service basis and comparing that to the SLA allocated to that VF and service. Resources are shared across guests and the resource utilization of each guest is measured relative to the capacity of the physical function.

The feature is supported only for SYM and ASYM services.

To enable the Rate Limiting feature:

1. **Install** the driver package on the host with Single-Root Input/Output Virtualization (SR-IOV) enabled.
2. **Set** `ServicesEnabled` to `asym` or `sym`.
3. **Perform** `qat_service shutdown` and `qat_service start`.

3.9.1 Service Level Agreement (SLA)

Service Level Agreement enforcement allocates a specified amount of capacity for a specified service to a specified VF: *max SLA enforced = (number of VFs) X (number of services)* where:

- Number of VFs varies based on device type
- Number of services = 2 (asymmetric or symmetric)

3.9.2 SLA Units

SLA units are measured as follows:

- Symmetric Crypto – 1Mbps of throughput.
- Asymmetric Crypto – 1 operation (ops) of reference operation.

3.9.3 SLA Manager Application

The `sla_mgr` tool is used to create, update, delete, list and get SLA capabilities. The SLA Manager executable is available in `$ICP_ROOT/build/sla_mgr` after the package is built and installed using `./configure; make install` commands.

3.9.3.1 SLA Commands

Table 13: Rate Limiting SLA Commands

Operation	Command
Create SLA	`./sla_mgr create <vf_addr> <rate_in_sla_units> <service>`
Update SLA	`./sla_mgr update <pf_addr> <sla_id> <rate_in_sla_units>`
Delete SLA	`./sla_mgr delete <pf_addr> <sla_id>`
Delete all SLAs	`./sla_mgr delete_all <pf_addr>`
Query SLA capabilities	`./sla_mgr caps <pf_addr>`
Query list of SLAs	`./sla_mgr list <pf_addr>`

Options:

- `pf_addr` - Physical address in domain:bus:device.function(xxxx:xx:xx.x) format.
- `vf_addr` - Virtual address in domain:bus:device.function(xxxx:xx:xx.x) format.
- `Service` - Asym(=0) or Sym(=1).
- `rate_in_sla_units` - [0-MAX]. MAX is found by querying the capabilities.
- `sla_id` - Value returned by `create` command.

One `rate_in_sla_units` is equal to:

- 1 operation per second - for asymmetric service.
- 1 Megabits per second - for symmetric service/compression service.

3.10 Power Management

The goal of power management is to manage and save power consumed by the device in the following states:

- idle => whenever no request is sent, power state is minimum.
- initialized or reset.
- active => whenever there are requests to be handled, power state is max.

3.10.1 Configuration

Power management configuration is included in the device configuration file (i.e. `/etc/4xxx_devX.conf` where `X` is the 0-based index of the device.)

Power management configurations parameters include:

Table 14: Power Management Configuration

Parameter	Description
`PmIdleInterruptDelay`	Configure power management interrupt delay from the system to QAT driver in millisecond(s). The default value is 512 milliseconds.
`PmIdleSupport`	Configure the device to enable/disable power management idle supporting. Power management idle support is enabled by default.

3.10.2 Usage

The information of power management status are exposed in debug sysfs file `/sys/kernel/debug/qat_4xxx_AAAA:BB:CC.D/pm_status` where:

- `AAAA:BB:CC.D` is the *Domain:BDF* of the target Intel® QAT Endpoint.

Example:

```
cat /sys/kernel/debug/qat_4xxx_0000:6b:00.0/pm_status
```

The QAT device is statically configured, so any change in device configuration file will only be effective after the device is rebooted.

3.10.3 Considerations

While power management is an important feature in reducing power consumed, it can affect the internal components' clocks of QAT devices, and that can affect example telemetry work. It also can impact latency numbers.

Important: It is recommended to disable the power management feature if either of the following is true:

- Using feature dependent on clock speed, such as telemetry, or
- Supporting latency-sensative workload.

3.11 Reliability, Availability, and Stability (RAS)

The RAS feature goal is to support the acceleration devices Reliability, Availability and Stability by handling the error interrupts initiated by the device.

Additionally the types of errors are counted and the counters made available via sysfs.

3.11.1 RAS Usage

Following PCIe specifications, errors are categorized as follows:

Table 15: RAS Error Types

Error Type	Description
Correctable	Device can recover on its own, no software involvement. The `ras_correctable` counter is incremented in sysfs.
Uncorrectable	Software intervention is needed to resolve the error. This may require the application to reset the session or resend the request to the device. The `ras_uncorrectable` counter is incremented in sysfs.
Fatal	Device unable to recover on its own even with software help. Restarting the device is required. The `ras_fatal` counter is incremented in sysfs.

The RAS sysfs files are located at `/sys/devices/pciAAAA:BB/AAAA:BB:CC.D/ras_X` where:

- `AAAA:BB:CC.D` is the *Domain:BDF* of the target Intel® QAT Endpoint.
- `ras_X` is the error type (`ras_correctable/ras_uncorrectable/ras_fatal`).

Example:

```
cat /sys/bus/pci/devices/0000\:6b\:00.0/ras_fatal
```

Note: RAS is enabled by default when the device is initialised.

3.11.2 AER Errors

The Linux kernel implements an AER driver for each PCIe device to handle errors reported through the AER mechanism.

AER error counters for each device are exposed through sysfs files categorized as follows:

Table 16: RAS AER Errors

Error Type	Description
AER Correctable	Device can recover on its own, no software involvement. The `aer_dev_correctable` counter is incremented in sysfs.
AER Uncorrectable	Software intervention is needed to resolve the error. In the case of an error caused by a transaction failure or for instance a packet memory buffer that can't be restored by ECC, then the device will need to reset in order to retry the transaction and attempt recovery. The `aer_dev_uncorrectable` counter is incremented in sysfs.
AER Fatal	In the case of a fatal error, the AER driver will additionally reset the PCIe link in an attempt to recover. The `aer_dev_fatal` counter is incremented in sysfs.

AER errors counters are exposed at `/sys/bus/pci/devices/AAAA:BB:CC.D/aer_dev_X` where:

- `AAAA:BB:CC.D` is the *Domain:BDF* of the target Intel® QAT Endpoint.
- `aer_dev_X` is the error type (`aer_dev_correctable/aer_dev_uncorrectable/aer_dev_fatal`).

Example:

```
cat /sys/bus/pci/devices/0000\:6b\:00.0/aer_dev_correctable
```

Important: AER reporting must be enabled in the BIOS to have errors reported through AER.

4 Acceleration Driver

Intel® QAT can accelerate the following services:

- Symmetric cryptography
- Public key cryptography
- Data compression/decompression

The Intel® QAT Endpoints are exposed as PCI devices. Applications running in user space typically access these services via the Intel® QAT APIs. Applications that run in the Linux* kernel can also access some services via the Linux* Kernel Cryptographic Framework (LKCF) API.

4.1 Controlling the Driver

Two methods are provided to manage the acceleration driver. They include:

- `qat_service`: script to manage the Intel® QAT Endpoints.
- `adf-ctl`: Utility for loading configuration files and sending events to the driver.

4.1.1 qat_service

The `qat_service` script is installed with the software package in the `/etc/init.d/` directory. The script allows a user to start, stop, or query the status (up or down) of a single Intel® QAT Endpoint or all Intel® QAT Endpoints in the system.

4.1.1.1 qat_service Usage

To view all Intel® QAT Endpoints in the system, use:

```
service qat_service status
```

If for example, there are two Intel® QAT Endpoints in the system, the output will be similar to the following:

```
qat_dev0 - type: c6xx, inst_id: 0, bsf: 06:00:0, #accel: 5 #engines: 10 state: up
qat_dev1 - type: c6xx, inst_id: 1, bsf: 83:00:0, #accel: 5 #engines: 10 state: up
```

Other options are also available:

```
service qat_service start||stop||status||restart||shutdown
```

For a system with multiple Intel® QAT Endpoints, you can start, stop or restart each individual device by passing the Intel® QAT Endpoint to be restarted or stopped as a parameter `qat_dev<N>`, for example:

```
service qat_service stop qat_dev0
service qat_service stop qat_dev1
```

The shutdown qualifier enables the user to bring down all Intel® QAT Endpoints and unload driver modules from the kernel. This contrasts with the stop qualifier, which brings down one or more Intel® QAT Endpoints, but does not unload kernel modules, so other Intel® QAT Endpoints can still run.

4.1.2 adf_ctl

The `adf_ctl` user space utility is separate to the driver and provides a mechanism for:

- Loading configuration file data to the kernel driver. The kernel space driver uses the data and also provides the data to the user space driver.
- Sending events to the driver to bring devices up and down.

The `adf_ctl` provided with the Intel® QAT 2.0 driver can also be used to interface with Intel® QAT 1.6 and 1.7 devices.

4.1.2.1 adf_ctl Usage

To bring up, down, restart or reset device(s):

```
adf_ctl [-c|--config] [qat_dev] [up|down|restart|reset]
```

To print device(s) status:

```
adf_ctl [qat_dev] status
```

To use the specified configuration file:

```
-c (--config) [config/file/path]
```

Note: If no device (physical or virtual) is selected, this file is used against all existing devices.

4.1.2.2 Examples

To bring device 0 down:

```
adf_ctl qat_dev0 down
```

To load device configuration from default path (e.g. `/etc/4xxx_dev1.conf`), then bring device 1 up:

```
adf_ctl qat_dev1 up
```

To load device configuration from specified path `/etc/4xxx_dev1.conf` and bring device 1 up:

```
adf_ctl -c /etc/user_4xxx_dev1.conf qat_dev1 up
```

To restart all devices with default configuration files:

```
adf_ctl restart
```

To restart all devices with specified configuration file `/etc/user_c4xxx_dev1.conf`:

```
adf_ctl -c /etc/user_4xxx_dev1.conf restart
```

To restart device 0 with specified configuration file `~/user_4xxx_dev1.conf`:

```
adf_ctl -c ~/user_c4xxx_dev1.conf qat_dev0 restart
```

To restart device 0:

```
adf_ctl qat_dev0 reset
```

4.2 Application Payload Memory Allocation

When performing offload operations through the Intel® QAT API, it is required that the payload data be placed in a buffer that is resident, physically contiguous, and DMA accessible from the acceleration hardware. It is the application's responsibility to provide buffers with these constraints.

Buffers are passed to the API with virtual addresses. The API translates these addresses to the address information required by the hardware.

4.2.1 Services

Table 17: Acceleration Driver Services

Service	API	Reference
Cryptographic service	`cpaCySetAddressTranslation()`	See the Intel® QuickAssist Technology Cryptographic API Reference Manual (refer to Table 2) for details.
Data Compression service	`cpaDcSetAddressTranslation()`	See the Intel® QuickAssist Technology Data Compression API Reference Manual (refer to Table 2) for details.

When the software requires the physical address, it calls the registered function.

Note: This address translation function is called at least once per request. Consequently, for optimal performance, the implementation of this function should be optimized.

If using the Intel® QAT Data Plane API, buffers are passed to the Intel® QAT API as physical addresses. The library passes this directly to the hardware, without the need for translation.

4.2.2 Thread Specific USDM

By default, memory allocation uses the USDM slab allocator, which gives 2MB contiguous memory. The allocation has locks in the library to prevent a race condition in getting the memory from the slab.

This lock has an impact on some multi-threaded applications and use cases, like HAProxy, causing a drop in performance.

To mitigate this issue, thread specific USDM is implemented which allocates and handles memory specific to threads. (For multi-thread apps, allocated memory information will be maintained separately for each thread).

This feature can be enabled by configuring with the configure flag:

```
--enable-icp-thread-specific-usdm
```

In some use cases with thread specific USDM, using a 128K slab allocator instead of the default 2MB allocator could improve performance and reduce memory consumption for a large number of threads. This can be enabled by configuring with the configure flag

```
--enable-128k-slab
```

Note: There is a limitation with thread specific USDM: memory allocated in one thread should be freed only by the thread which allocates it.

Incorrect cleanup can lead to a segmentation fault (segfault).

Also, memory allocated in a thread is freed automatically when the thread exits/terminates, even if the user does not explicitly free the memory.

See the ./configure flags section of the Getting Started Guide for more information on these flags.

Important: We have observed poor multithreaded performance with QAT_Engine using OpenSSL* at higher thread counts.

Unfortunately, these issues appear to stem from the way OpenSSL* implements its `engine_table_select` and locks. For relevant issues on the OpenSSL* github pages, see the two issues below:

- OpenSSL* 1.1.1.x: Performance bottleneck with locks in `engine_table_select()` function #18509, https://github.com/openssl/openssl/issues/18509
- OpenSSL* 3.0: 3.0 performance degraded due to locking #20286, https://github.com/openssl/openssl/issues/20286

4.3 Return Codes

This table shows the return codes used by various components of the acceleration driver, defined in `$ICP_ROOT/quickassist/include/cpa.h`.

Table 18: Return Codes

Return Type	Return Code	Description
`CPA_STATUS_SUCCESS`	0	Requested operation was successful.
`CPA_STATUS_FAIL`	-1	General or unspecified error occurred. Refer to the console log user space application or to `/var/log/messages` in kernel space for more details of the failure.
`CPA_STATUS_RETRY`	-2	Recoverable error occurred. Refer to relevant sections of the API for specifics on what the suggested course of action.
`CPA_STATUS_RESOURCE`	-3	Required resource is unavailable. The resource that has been requested is unavailable. Refer to relevant sections of the API for specifics on what the suggested course of action.
`CPA_STATUS_INVALID_PARAM`	-4	Invalid parameter has been passed in.
`CPA_STATUS_FATAL`	-5	Fatal error has occurred. A serious error has occurred. Recommended course of action is to shut down and restart the component.

continues on next page

Table 18 – continued from previous page

Return Type	Return Code	Description
CPA_STATUS_UNSUPPORTED	-6	The function is not supported, at least not with the specific parameters supplied. This may be because a particular capability is not supported by the current implementation.
CPA_STATUS_RESTARTING	-7	The API implementation is restarting. This may be reported if, for example, a hardware implementation is undergoing a reset.

4.4 Linux* Device Driver Operations Return Codes

This table shows the return codes used by the driver to handle Linux* device driver operations.

Table 19: Linux* Device Driver Operations Return Codes

Return Type	Return Code	Description
SUCCESS	0	Operation was successful.
FAIL	1	General error occurred. Refer to the console log user space application or to `/var/log/messages` in kernel space for more details of the failure.
-EPERM	-1	Operation is not permitted. Used during ioctl operations.
-ENOENT	-2	No such file or directory.
-EINTR	-4	Interrupted system call.
-EIO	-5	Input/Output error occurred. Used when copying configuration data to and from user space.
-EBADF	-9	Bad File Number. Used when an invalid file descriptor is detected.
-EAGAIN	-11	Try Again. Used when a recoverable operation occurred.
-ENOMEM	-12	Out of Memory. Memory resource that has been requested is not available.
-EACCES	-13	Permission Denied. Used when the operation failed to connect to a process or open a device.
-EFAULT	-14	Bad Address. Used when an operation detects invalid parameter data.
-EBUSY	-16	Device or resource is busy.
-EEXIST	-17	File exists.
-ENODEV	-19	No Such Device. Used when an operation detects invalid device id.
-EINVAL	-22	Invalid argument.
-ENOTTY	-25	Invalid Command Type. Used when an ioctl operation detects an invalid command type.
-ENOSPC	-28	No space left on device.
-ERANGE	-34	Math result not representable.

continues on next page

Table 19 – continued from previous page

Return Type	Return Code	Description
-ENOSYS	-38	Function not implemented.
-EL3HLT	-46	Level 3 Halted.
-ETIME	-62	Timer expired.
-EBADMSG	-74	Not a data message.
-EOVERFLOW	-75	Value too large for defined data type.
-EOPNOTSUPP	-95	Operation not supported on transport endpoint.
-EINPROGRESS	-115	Operation now in progress.

5 Configuration Files

This section describes the configuration file(s) that allows customization of runtime operation. The configuration file(s) must be tuned to meet the performance needs of the target application. There is a single configuration file for each Intel® QAT Endpoint in the system.

If Single-Root Input/Output Virtualization (SR-IOV) is enabled, a separate configuration file is used for each virtual function.

Note: The software package includes default configuration file(s), which may not provide optimal performance on all platforms. Consider performance implications as well as the configuration details provided in this section if your system requires modifications to the default configuration file.

5.1 Configuration File Overview

There is a single configuration file for each Intel® QAT Endpoint and there may be multiple Intel® QAT Endpoints.

Note: Depending on the model number, a device may also contain no Intel® QAT Endpoints.

The configuration file is split into a number of different sections: a General section and one or more Logical Instance sections.

The *General* section includes parameters that allow the user to specify:

- Which services are enabled.
- Concurrent requests default configuration.
- Interrupt coalescing configuration (optional).
- Statistics gathering configuration.

Additional details are included in *General Section*.

Logical Instances sections (there may be one or more) include parameters that allow the user to set:

- The number of cryptography or data compression instances being managed.

- For each instance, the name of the instance, whether or not polling is enabled, and the core to which an instance is affinitized.

Additional details are included in *Logical Instances Section*.

Sample configuration files are included in the package in the `quickassist/utilities/adf_ctl/conf_files` directory.

5.2 General Section

The general section of the configuration file contains general parameters and statistics parameters.

Note: *Default* denotes the value in the configuration file when shipped or the value used if not specified in the configuration file.

This table describes the other parameters that can be included in the General section.

Table 20: General Section Parameters

Parameter	Description	Default	Range
`ServicesEnabled`	Defines the service(s) available (cryptographic [cy], symmetric [sym], asymmetric [asym], data compression [dc]). Refer to *ServicesEnabled* for additional details.	<varies>	sym, asym, cy, dc Note: Multiple values permitted, use; as the delimiter.
`ServicesProfile`	Specifies the services that are available when the driver loads.	Default	See *ServicesProfile* for additional details.
`CyNumConcurrentSymRequests`	Specifies the number of cryptographic concurrent symmetric requests for cryptographic instances in general. Refer to *Concurrent Requests* for additional details.	512	64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, or 65536
`CyNumConcurrentAsymRequests`	Specifies the number of cryptographic concurrent asymmetric requests for cryptographic instances in general. Refer to *Concurrent Requests* for additional details.	64	64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, or 65536

continues on next page

Table 20 – continued from previous page

Parameter	Description	Default	Range
`DcNumConcurrentRequests`	Specifies the number of data compression concurrent requests for data compression instances in general. Refer to *Concurrent Requests* for additional details.	512	64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, or 65536
`DcIntermediateBufferSizeInKB`	Specifies the size in KB of each intermediate buffer in on-chip memory for dynamic compression.	64	32 or 64
`HeartbeatTimer`	Default heartbeat timer.	1000	> 200
`AutoResetOnError`	Automatically resets the device in case of fatal error or heartbeat failure.	0	0 or 1
`PmIdleInterruptDelay`	Default value for power management idle interrupt delay. Refer to *Power Management Parameters* for additional details.	512	
`PmIdleSupport`	This flag is to enable power management idle support. Refer to *Power Management Parameters* for additional details.	1	0 or 1
`SVMEnabled`	This flag is to enable SVM support. Refer to *Shared Virtual Memory Parameters* for additional details.	0	0 or 1
`ATEnabled`	This flag is to enable Address Translation Service(ATS). Refer to *Shared Virtual Memory Parameters* for additional details.	0	0 or 1

5.2.1 ServicesEnabled

Additional details on the `ServicesEnabled` parameter:

- This parameter is valid for all QAT devices.
- Default value varies depending on the underlying QAT Endpoint.
- `cy` is not valid value for QAT2.0 devices. `asym` and `sym` are used.
- Only two of the three services (`asym`, `sym`, and `dc`) may be enabled on an individual QAT2.0 Endpoints.

5.2.1.1 Performance Considerations

Important: The following is applicable to QAT2.0 devices only.

In order to maximize QAT throughput performance for a given service type, one should specify ONLY that service type parameter for `ServicesEnabled`.

By design, two Acceleration Engine clusters are available, each containing four Acceleration Engines. Each of these two clusters are limited to using a single service. Therefore, the possible split options are 4|4 or 8 for a given service type.

Configuration examples:

- While using VFs on a system configured for `ServicesEnabled = sym;dc`, 4 acceleration engines will be dedicated to SYM and 4 acceleration engines will be dedicated to DC, so only 2 resource providers per child VF can be used for SYM. Here, we should expect some performance degradation for SYM (even if DC is not used).
- While using VFs on a system configured for `ServicesEnabled = sym`, all 8 acceleration engines will be dedicated to SYM, so all 4 resource providers of a child VF can be used for SYM only. Here, we will see the best SYM performance.

Note: Packet size will also modulate the impact of the above configuration settings.

5.2.2 ServicesProfile

Important: This parameter is valid for QAT1.7x devices.

The `ServicesProfile` parameter defines the services that are available when the driver loads. For example, if `ServicesProfile = COMPRESSION` is in the General section, the compression and decompression are available, along with service chaining, but not cryptography.

Note: When a `ServicesProfiles` parameter value is used that supports rate limiting is defined, internal resources are reallocated to administrating Rate Limiting/Device Utilization. This reduces performance by roughly 5%.

5.2.2.1 General Default Configuration Parameters

Table 21: General Default Configuration Parameters

Service	DEFAULT	CRYPTO	COMPRESSION	CUSTOM1
Asymmetric Crypto	YES	YES		YES
Symmetric Crypto	YES	YES		YES
Hash	YES	YES	YES	YES
Cipher	YES	YES		YES
MGF KeyGen	YES	YES		
SSL/TLS KeyGen	YES	YES		YES
HKDF		YES		YES
Compression	YES		YES	YES
Decompression (stateless)	YES		YES	YES
Decompression (stateful)	YES		YES	
Service Chaining			YES	
Device Utilization		YES	YES	YES
Rate Limiting		YES	YES	YES

Note: Set the service profile to determine available features.

5.2.3 Concurrent Requests

Additional details on the concurrent request parameters:

- This parameter is valid for all QAT devices.
- The concurrent request parameters include both Transmit (Tx) and Receive (Rx) requests.
- For each service enabled, `NumConcurrentRequests` must be set to value from the range.
- The number of concurrent requests registered by the Intel® QuickAssist driver is set to `NumConcurrentRequests - 2`.

This implementation guarantees that the request ring will never be full and avoids the need for a Memory Mapped IO (MMIO) read. This implementation maximizes throughput performance.

5.2.4 Power Management Parameters

Important: This parameter is valid for QAT2.0 devices.

Power management configuration is included in the device configuration file (i.e. `/etc/4xxx_devX.conf` where `X` is the 0-based index of the device.)

Power management configurations parameters include:

- `PmIdleInterruptDelay` - Configure power management interrupt delay from the system to QAT driver in millisecond(s). The default value is 512 milliseconds.
- `PmIdleSupport` - Configure the device to enable/disable power management idle supporting. Power management idle support is enabled by default.

Refer to the *Power Management* section for additional details.

5.2.5 Shared Virtual Memory (SVM) Parameters

Important: This parameter is valid for QAT2.0 devices.

SVM configuration parameters are included in the device configuration file (i.e. `/etc/4xxx_devX.conf` where `X` is the 0-based index of the device.)

5.2.5.1 SVMEnabled

When this flag is set in the driver configuration, it indicates that the guest virtual address (GVA) to host physical address (HPA) translation will use IOMMU hardware based translation table instead of using the software based address translation. With SVMEnabled set, it is not required to submit buffers that are physically contiguous.

Details

- This parameter is disabled by default. Refer to *SVM Kernel Requirements* section for additional details.
- The parameter is valid for both PF and VF configuration files.
- It **is** possible for the VF to enable this parameter even if the parameter is disabled in the corresponding PF configuration file.

5.2.5.2 ATEnabled

When this flag is set in the driver configuration, the Address Translation Service (ATS) is enabled. IOMMU and QAT have the ability to handle page faults using Page Request Service (PRS) when using dynamic virtual memory allocated by systemcall such as `malloc`.

Details

- This parameter is disabled by default. Refer to *SVM Kernel Requirements* section for additional details.
- SVMEnabled **must** be enabled in order to enable ATEnabled.
- The parameter is valid for both PF and VF configuration files.
- It is **not** possible for the VF to enable the service if the parameter is disabled in the corresponding PF configuration file.

5.3 Logical Instances Section

This section allows the configuration of logical instances in each address domain (kernel space and individual user space processes).

The address domains are in the following format:

- For the kernel address domain: `[KERNEL]` targeted to Linux* Kernel Crypto Framework (LKCF).
- For user process address domains: `[xxxxx]`, where `xxxxx` may be any ASCII value that uniquely identifies the user mode process.

In user space, to allow the driver to configure the logical instances associated with a user process correctly, the process must call the function `icp_sal_userStart` passing the `xxxxx` string during process initialization. When the user space process is finished, it must call the function `icp_sal_userStop` to free resources. Refer to *User Space Access Configuration Functions* for more information.

A single VF configured for the SR-IOV use case cannot have both user space instances and kernel space instances. Separate VFs must be created for user space and kernel space.

The `NumProcesses` parameter (in the User Process section) indicates the max number of user space processes within that section name with access to instances on this device. Refer to *icp_sal_userStart* for more information.

The items that can be configured for a logical instance are:

- The name of the logical instance.
- The polling mode.
- The core to which the instance is affinitized (optional).

5.3.1 [KERNEL] Section

In the `[KERNEL]` section of the configuration file, information about the number and type of kernel instances can be defined. This table describes the parameters that determine the number of kernel instances for each service.

Note: The maximum number of cryptographic and data compression instances supported per Intel® QAT Endpoint is 32. For exceptions refer to *Increasing the Maximum Number of Processes/Instances.*

Table 22: [KERNEL] Section Parameters

Parameter	Description	Default	Range
`NumberCyInstances`	Specifies the number of cryptographic instances. Note: Depends on the number of allocations to other services.	0	0 to 32
`NumberDcInstances`	Specifies the number of data compression instances. Note: Depends on the number of allocations to other services.	0	0 to 32

5.3.2 User Process [xxxxx] Sections

There is one [xxxxx] section of the configuration file for each Intel® QAT Endpoint to be configured. In each [xxxxx] section of the configuration file, user space access to the Intel® QAT Endpoint can be configured. Parameters for each user process instance can also be defined. Common names for this section are `[SSL]` or `[SHIM]`

Note: Check the SKU information for your specific device to determine how many Intel® QAT Endpoints the device contains.

This table shows the parameters in the configuration file that can be set for user process [xxxxx] sections.

Table 23: [User Process] Section Parameters

Parameter	Description	Default	Range
`NumProcesses`	The number of user space processes with section name [xxxxx] that have access to this device. The maximum number of processes that can call `icp_sal_userStart` and be active at any one time. See *icp_sal_userStart* for additional information. *Caution*: Resources are pre-allocated. If this parameter value is set too high, the driver fails to load.	1	For constraints, see *Maximum Number of Process Calculations*. For exceptions, see *Increasing the Maximum Number of Processes/Instances*.
`LimitDevAccess`	Indicates if the user space processes in this section are limited to only access instances on this Intel® QAT Endpoint.	0	0 (disabled, processes in this section can access multiple Intel® QAT Endpoints), or 1 (enabled, processes in this section can only access this Intel® QAT Endpoint). For additional information, see *Configuring Multiple Processes on a System with Multiple Intel® QAT Endpoints*.
`NumberCyInstances`	Specifies the number of cryptographic instances. Note: Depends on the number of allocations to other services.	6	0 to 32. For exceptions, see *Increasing the Maximum Number of Processes/Instances*.
`NumberDcInstances`	Specifies the number of data compression instances. Note: Depends on the number of allocations to other services.	2	0 to 32. For exceptions, see *Increasing the Maximum Number of Processes/Instances*.

5.3.3 Cryptographic Logical Instance Parameters

The following table shows the parameters that can be set for cryptographic logical instances.

Note: *Default* denotes the value in the configuration file when shipped.

Table 24: Cryptographic Logical Instance Parameters

Parameter	Description	Default	Range
`CyXName`	Specifies the name of cryptographic instance number X.	IPSec0 for `KERNEL` section. SSL0 for user section	String (max. 64 characters)
`CyXIsPolled`	Specifies if cryptographic instance number x works in poll mode, interrupt mode or epoll mode.	0 for kernel space instances 1 for user space instance	0 (interrupt mode) for instances in the `KERNEL` section. 1 (poll mode) for instances in user space sections. 2 (epoll mode event based polling mode) for instances in user space section.
`CyXCoreAffinity`	Specifies the core to which the instance should be affinitized.	Varies depending on the value of X.	0 to max. number of cores in the system.

5.3.4 Data Compression Logical Instance Parameters

This table shows the parameters in the configuration file that can be set for data compression logical instances.

Table 25: Data Compression Logical Instance Parameters

Parameter	Description	Default	Range
`DcXName`	Specifies the name of data compression instance number X.	IPComp0	String (max. 64 characters)
`DcXIsPolled`	Specifies if data compression instance number x works in poll mode, interrupt mode or epoll mode.	0 for kernel space instances 1 for user space instances	0 (interrupt mode) for instances in the `KERNEL` section. 1 (poll mode) for instances in the `KERNEL_QAT` and user space sections. 2 (epoll mode event based polling mode) for instances in user space section.
`DcXCoreAffinity`	Specifies the core to which the data compression instance should be affinitized.	Varies depending on the value of X.	0 to max. number of cores in the system.

Note:

- The maximum number of data compression instances supported is 64.
- *Default* denotes the value in the configuration file when shipped.

5.3.5 Setting the Core Affinity Parameter for a Logical Instance

When instances are configured with `IsPolled = 1` (Polling mode), the parameter `CoreAffinity` does not have any impact.

Although not used, it is a valid parameter and applications can query the value using `cpaCyInstanceGetInfo2` (see `coreAffinity` bitmask in `CpaInstanceInfo2`). For example, the sample code affinitizes the thread that uses an instance to the core indicated in `CoreAffinity` the config file for that instance.

For instances configured in Interrupt Mode (`IsPolled = 2` in user space (epoll) and `IsPolled = 0` in kernel space), the value of `CoreAffinity` is used to affinitize the interrupt handler to that core.

5.4 Maximum Number of Process Calculations

The `NumProcesses` parameter is the number of user space processes per service within the [xxxx] section domain with access to this Intel® QAT Endpoint.

The value to which this parameter can be set is determined by a number of factors including the number of cryptography instances and/or data compression instances in the process section along with `ServicesEnabled` and potentially `ServicesProfile`. The total number of processes, per service, created by the driver is given by the expression (e.g., for cryptography):

```
(NumProcesses) x (NumberCyInstances)
```

The maximum number of processes that can be supported is dependant upon the underlying hardware.

5.4.1 Increasing the Maximum Number of Processes/Instances

Note:

- One bank is used per Intel® QAT virtual function (VF).
- This section only applies when the instances that make use of polled mode.

The maximum number of instances can be increased with the careful selection of the `ServiceEnabled` parameter.

Compression, symmetric cryptography, and asymmetric cryptography each require two rings out of the 16 possible rings for a ring bank. By selecting only, the services needed, the number of instances can be increased.

Here are the variations including the maximum number of processes that can be supported for given configuration:

Note: The `ServicesProfile` parameter value may also need to be changed. See *Services Profile* for additional information.

Table 26: Configuration Variations

ServicesEnabled	QAT 1.7x	QAT 2.0	Notes
`sym`	128	64	Compression and asymmetric crypto service not available.
`asym`	128	64	Compression and symmetric crypto service not available.
`cy`	128	Invalid	Compression and symmetric crypto service not available.
`dc`	128	64	Asymmetric and symmetric crypto service not available.
`dc;sym`	64	32	Asymmetric crypto services will not be available.
`dc;asym`	64	32	Symmetric crypto services will not be available.
`sym;asym`	Invalid	32	Compression services will not be available.

5.4.1.1 Invalid Configurations

If maximum number of processes is exceeded, the acceleration software will fail to load. The error message will be similiar to:

```
service qat_service restart qat_dev0
```

```
Stopping device qat_dev0
Starting device qat_dev0
Processing /etc/4xxx_dev0.conf
Ioctl failed
QAT Error: Failed to load config data to device
```

And `dmesg` output will look similar to:

```
[116378.383041] Don't have enough rings for instance SSL0 in process SHIM_DEV0_INT_32
[116378.391976] 4xxx 0000:6b:00.0: Failed to create rings for cy
[116378.398881] 4xxx 0000:6b:00.0: Failed to process user section SHIM
[116378.406484] 4xxx 0000:6b:00.0: Failed to config device
```

5.4.1.2 Configuring Instances for Virtual Functions

To configure the number of instances for a virtual function:

1. **Install** the driver package on the host with SR-IOV enabled.
2. **Update** the physical function configuration file to set `ServicesEnabled` (refer to *Increasing the Maximum Number of Processes/Instances*).
3. **Perform** `qat_service shutdown` and `qat_service start`.
4. **Update** the virtual function configuration file to set `ServicesEnabled` (refer to *Increasing the Maximum Number of Processes/Instances*).
5. **Restart** the `qat_service`.

The value of `ServicesEnabled` in the VF configuration file should be the same as the value of `ServicesEnabled` in the PF configuration file, or a subset of that value as shown in this table. For instance, if a PF is configured as `cy`, allowable VF configurations related to that PF can only be `cy`, `asym`, or `sym`. VF device restart will fail if a VF configuration is not allowed for that related PF.

If a VF service is configured to a subset of PF service, the number of VF instances is limited to the number allowed for that PF service as described in *Increasing the Maximum Number of Processes/Instances.* For example, if the PF configuration file has `ServicesEnabled=dc;asym`, only four (not eight) dc instances are enabled if the VF is configured for dc only.

Note: Valid Physical Function for each supported platform is described in the *Configuration Variations* table.

Table 27: Configuring Physical Functions and Virtual Functions

Configured PF Service	Available VF Services
dc;asym	dc;asym
	asym
	dc
dc;sym	dc;sym
	sym
	dc
asym;sym	asym;sym
	sym
	asym
asym	asym
sym	sym
dc	dc

5.5 Configuring Multiple Intel® QuickAssist Technology Endpoints in a System

A platform may include more than one Intel® QAT Endpoint. Each device must have its own configuration file. When the acceleration software is installed, default configuration files are installed to the `/etc` folder. The format and structure of the configuration file is exactly the same for all devices.

Warning: If a configuration file does not exist for an Intel® QAT Endpoint, that endpoint will not start, and an error is displayed indicating that a configuration file was not found.

To determine the number of Intel® QAT Endpoints in a system, use the `lspci` utility:

```
lspci -nn | egrep -e '8086:37c8|8086:19e2|8086:0435|8086:6f54|8086:4940|8086:4942'
```

The output from a high-end 4th Gen Intel® Xeon® Scalable Processor is similar to the following:

```
6b:00.0 Co-processor [0b40]: Intel Corporation Device [8086:4940] (rev 40)
70:00.0 Co-processor [0b40]: Intel Corporation Device [8086:4940] (rev 40)
75:00.0 Co-processor [0b40]: Intel Corporation Device [8086:4940] (rev 40)
7a:00.0 Co-processor [0b40]: Intel Corporation Device [8086:4940] (rev 40)
e8:00.0 Co-processor [0b40]: Intel Corporation Device [8086:4940] (rev 40)
ed:00.0 Co-processor [0b40]: Intel Corporation Device [8086:4940] (rev 40)
f2:00.0 Co-processor [0b40]: Intel Corporation Device [8086:4940] (rev 40)
f7:00.0 Co-processor [0b40]: Intel Corporation Device [8086:4940] (rev 40)
```

The output from a system with a high-end Intel® C62x Chipset SKU is similar to the following:

```
88:00.0 Co-processor [0b40]: Intel Corporation Device [8086:37c8] (rev 03)
8a:00.0 Co-processor [0b40]: Intel Corporation Device [8086:37c8] (rev 03)
8c:00.0 Co-processor [0b40]: Intel Corporation Device [8086:37c8] (rev 03)
```

Then, after the driver is loaded, the user can use the `qat_service` script to determine the name of each Intel® QAT Endpoint and its status. For example:

```
service qat_service status
```

The output from a high-end 4th Gen Intel® Xeon® Scalable Processor is similar to the following:

```
Checking status of all devices.
There is 8 QAT acceleration device(s) in the system:
    qat_dev0 - type: 4xxx,  inst_id: 0,  node_id: 0,  bsf: 0000:6b:00.0,  #accel: 1
↪#engines: 9 state: up
    qat_dev1 - type: 4xxx,  inst_id: 1,  node_id: 0,  bsf: 0000:70:00.0,  #accel: 1
↪#engines: 9 state: up
```

(continues on next page)

(continued from previous page)

```
    qat_dev2 - type: 4xxx,  inst_id: 2,  node_id: 0,  bsf: 0000:75:00.0,  #accel: 1
↪#engines: 9 state: up
    qat_dev3 - type: 4xxx,  inst_id: 3,  node_id: 0,  bsf: 0000:7a:00.0,  #accel: 1
↪#engines: 9 state: up
    qat_dev4 - type: 4xxx,  inst_id: 4,  node_id: 1,  bsf: 0000:e8:00.0,  #accel: 1
↪#engines: 9 state: up
    qat_dev5 - type: 4xxx,  inst_id: 5,  node_id: 1,  bsf: 0000:ed:00.0,  #accel: 1
↪#engines: 9 state: up
    qat_dev6 - type: 4xxx,  inst_id: 6,  node_id: 1,  bsf: 0000:f2:00.0,  #accel: 1
↪#engines: 9 state: up
    qat_dev7 - type: 4xxx,  inst_id: 7,  node_id: 1,  bsf: 0000:f7:00.0,  #accel: 1
↪#engines: 9 state: up
```

The output from a system with a high-end Intel® C62x Chipset SKU is similar to the following:

```
qat_dev0 - type: c6xx, inst_id: 0, bsf: 06:00:0, #accel: 5 #engines: 10 state: up
qat_dev1 - type: c6xx, inst_id: 1, bsf: 85:00:0, #accel: 5 #engines: 10 state: up
qat_dev2 - type: c6xx, inst_id: 2, bsf: 87:00:0, #accel: 5 #engines: 10 state: up
```

The `qat_service` can start, stop, restart and shutdown each device separately or all Intel® QAT Endpoints together. Refer to *Managing Intel QuickAssist Technology Endpoints Using qat_service* for more information.

Some important configuration file information when using multiple Intel® QAT Endpoints:

- When specifying kernel and user space instances in the configuration file, the `Cy<Number>Name` and `Dc<Number>Name` parameters must be unique in the context of the section name only.
  - For example, it is valid to have a parameter called `Cy0Name` in both a kernel instance section (if supported) and a user instance section in the same configuration file without issue. Also, the parameter names do not need to be unique at a system-wide level. For example, it is valid to have a parameter called `Cy0Name` in both the configuration file for `dev0` and the configuration file for `dev1` without issue.
- For Intel® QAT Endpoints with configuration files that have the same section name (for example, `[SSL]` and the same data in that section), it is necessary to use the `cpaCyInstanceGetInfo2()` function to distinguish between Intel® QAT Endpoints. The `cpaCyInstanceGetInfo2()` allows the user of the API to query which Intel® QAT Endpoint a cryptography instance handle belongs to. In addition, for any application domain defined in the configuration files (e.g. `[SSL]`), a call to `cpaCyGetNumInstances()` returns the number of cryptography instances defined for that domain across all configuration files. A subsequent call to `cpaCyGetInstances()` obtains these instance handles.

5.6 Configuring Multiple Processes on a System with Multiple Intel® QAT Endpoints

As an example, consider a system with two Intel® QAT Endpoints where it is necessary to configure two user space sections. One section is identified as SSL and the other is identified as Internet Protocol Security (IPSec).

- For the SSL section, configure eight processes, where each process has access to one acceleration instance.
- For the IPSec section, configure one process, with access to eight acceleration instances, four per Intel® QAT Endpoint.

In this scenario, the user space section of the configuration file would look like the following for the first Intel® QAT Endpoint.

```
[SSL] #User space section name
NumProcesses=4 # There are 4 user space process with section name SSL with access
↪to this device
LimitDevAccess=1 # These 4 SSL user space processes only use this device
NumCyInstances=1 # Each process has access to 1 Cy instance on this device
NumDcInstances=0 # Each process has access to 0 Dc instances on this device

# Crypto - User instance #0
Cy0Name = "SSL0"
Cy0IsPolled = 1
Cy0CoreAffinity = 0 # Core affinity not used for polled instance

[IPsec] #User space section name
NumProcesses=1 # There is 1 user space process with section name IPSec with access
↪to this device
LimitDevAccess=0 # This IPSec user space process may have access to other devices
NumCyInstances=4 # The IPSec process has access to 4 Cy instances on this device
NumDcInstances=0 # The IPSec process has access to 0 Dc instances on this device

# Crypto - User instance #0
Cy0Name = "IPSec0"
Cy0IsPolled = 1
Cy0CoreAffinity = 0 # Core affinity not used for polled instance
# Crypto - User instance #1

Cy1Name = "IPSec1"
Cy1IsPolled = 1
Cy1CoreAffinity = 0 # Core affinity not used for polled instance

# Crypto - User instance #2
Cy2Name = "IPSec2"
Cy2IsPolled = 1
Cy2CoreAffinity = 0 # Core affinity not used for polled instance
```

(continues on next page)

(continued from previous page)

```
# Crypto - User instance #3
Cy3Name = "IPSec3"
Cy3IsPolled = 1
Cy3CoreAffinity = 0 # Core affinity not used for polled instance
```

The second Intel® QAT Endpoint configuration looks like:

```
[SSL] #User space section name
NumProcesses=4 # There are 4 user space process with section name SSL with access
↪to this device
LimitDevAccess=1 # These 4 SSL user space processes only use this device
NumCyInstances=1 # Each process has access to 1 Cy instance on this device
NumDcInstances=0 # Each process has access to 0 Dc instances on this device

# Crypto - User instance #0
Cy0Name = "SSL0"
Cy0IsPolled = 1
Cy0CoreAffinity = 0 # Core affinity not used for polled instance

[IPSec] #User space section name
NumProcesses=1 # There is 1 user space process with section name IPSec with access
↪to this device
LimitDevAccess=0 # This IPSec user space process may have access to other devices
NumCyInstances=4 # The IPSec process has access to 4 Cy instances on this device
NumDcInstances=0 # The IPSec process has access to 0 Dc instances on this device

# Crypto - User instance #0
Cy0Name = "IPSec0"
Cy0IsPolled = 1
Cy0CoreAffinity = 0 # Core affinity not used for polled instance

# Crypto - User instance #1
Cy1Name = "IPSec1"
Cy1IsPolled = 1
Cy1CoreAffinity = 0 # Core affinity not used for polled instance

# Crypto - User instance #2
Cy2Name = "IPSec2"
Cy2IsPolled = 1
Cy2CoreAffinity = 0 # Core affinity not used for polled instance

# Crypto - User instance #3
Cy3Name = "IPSec3"
Cy3IsPolled = 1
Cy3CoreAffinity = 0 # Core affinity not used for polled instance
```

Eight processes (with section name SSL) can call the `icp_sal_userStart("SSL")` function to get access to one crypto instance each. One process (with section name IPSec) can call the `icp_sal_userStart("IPSec")` function to get access to eight crypto instances.

Internally in the driver, this works as follows:

1. When the driver is configured (that is, the `service qat_service` is called), the driver reads the configuration file for the device and populates an internal configuration table.
2. Reading the configuration file for `dev0`:
    a. For the section named `[SSL]`, the driver determines that four processes are required and that these processes limit access to this device only. In this case, the driver creates four internal sections that it labels `SSL_DEV0_INT_0`, `SSL_DEV0_INT_1`, `SSL_DEV0_INT_2` and `SSL_DEV0_INT_3`. Each section is given access to one crypto instance as described.
    b. For section name `[IPSec]`, the driver determines that one process is required and that this process does not limit access to this device only (that is, it may access instances on other devices). In this case, the driver creates one internal section that it labels `IPSec_INT_0` and gives this access to four crypto instances on this device.
3. Reading the configuration file for `dev1`:
    a. For the section named `[SSL]`, the driver determines that four processes are required and that these processes are limited to access this device only. In this case, the driver creates four internal sections that it labels `SSL_DEV1_INT_0`, `SSL_DEV1_INT_1`, `SSL_DEV1_INT_2` and `SSL_DEV1_INT_3`. Each section is given access to one crypto instance as described.
    b. For the section named `[IPSec]`, the driver determines that one process is required and that this process may have access to instances on other devices. In this case, the driver creates one internal section that it labels `IPSec_INT_0` and gives this access to four crypto instances on this device.

    Note: This section name now appears in both devices' internal configuration and, therefore, the process that gets assigned this section name will have access to instances on both devices.

4. In total, there are nine separate sections (`SSL_DEV0_INT_0`, `SL_DEV0_INT_1`, `SSL_DEV0_INT_2`, `SSL_DEV0_INT_3`, `SSL_DEV1_INT_0`, `SSL_DEV1_INT_1`, `SSL_DEV1_INT_2`, `SSL_DEV1_INT_3` and `IPSec_INT_0`) with access to crypto instances.

When a process calls the `icp_sal_userStart ("SSL")` function, the driver locates the next available section of the form `SSL_DEV<m>_INT<....>` (of which there are eight in total in this example) and assigns this section to the process. This gives the process access to corresponding crypto instances.

When a process calls the `icp_sal_userStart ("IPSec")` function, the driver locates the next available section of the form `IPSec_INT_<....>` (of which there is only one in total for this example) and assigns this section to the process. This gives the process access to the corresponding crypto instances.

Note: The `icp_sal_userStartMultiProcess()` function has been deprecated. The API still exists, but it simply calls `icp_sal_userStart()`.

5.7 Sample Configuration Files

Sample configuration files are available in `quickassist/utilities/adf_ctl/conf_files`. Depending on the product and configuration, one or more of these will be copied to `/etc` during the package installation.

6 Services

Intel® QAT can accelerate the following services:

- Data compression/decompression.
- Symmetric cryptography.
- Public key cryptography.

Details on the services are included in the following sections.

6.1 Data Compression

6.1.1 Compresion Features

- Deflate Compression Algorithm.
- LZ77 and Huffman Encoding.
- LZ4/LZ4s Compression Algorithm.
- Compress and Verify.
- Checksums.
- Programmable CRC64.

6.1.2 Compression Limitations

- Stateful compression is not supported.
- Stateful decompression is not supported on QAT2.0 hardware devices.
- Batch and Pack (BnP) compression is not supported.

6.1.3 Compression Session Setup

The following table lists the properties that should be configured in the `CpaDcSessionSetupData` structure depending on the compression algorithm requested.

Table 28: Compression CpaDcSessionSetupData Properties

Property	Details
`CpaDcCompLvl compLevel`	Properties common to all algorithms
`CpaDcCompType compType`	
`CpaDcAutoSelectBest autoSelectBestHuffmanTree`	
`CpaDcSessionDir sessDirection`	
`CpaDcSessionState sessState`	
`CpaDcCompWindowSize windowSize`	
`CpaDcChecksum checksum`	
`CpaDcHuffType huffType`	Deflate specific
`CpaDcCompLZ4BlockMaxSize lz4BlockMaxSize`	LZ4 specific
`CpaBoolean lz4BlockChecksum`	
`CpaBoolean lz4BlockIndependence`	
`CpaBoolean accumulateXXHash` (currently unsupported)	
`CpaDcCompMinMatch minMatch`	LZ4s specific

Note: Should the application use no-session API `cpaDcNsCompressData()`, the properties listed above are available in the `CpaDcNsSetupData` data structure.

6.1.4 Decompression Session Setup

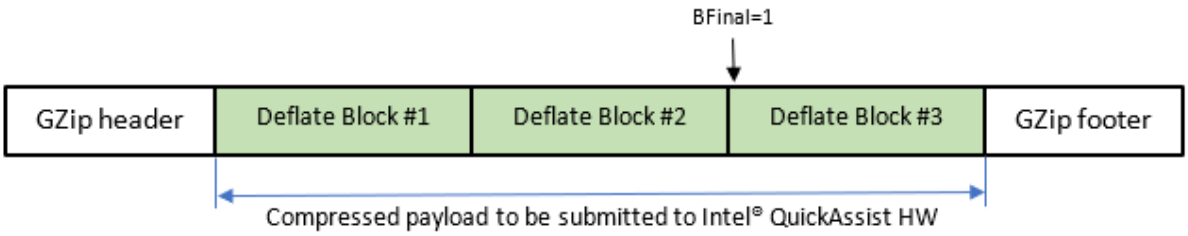
The following table lists the properties that should be configured in the `CpaDcSessionSetupData` structure depending on the format of the payload to decompression.

Table 29: Decompression CpaDcSessionSetupData Properties

Property	Deflate	LZ4
`CpaDcCompType compType`	Yes	Yes
`CpaDcSessionDir sessDirection`	Yes	Yes
`CpaDcSessionState sessState`	Yes	Yes
`CpaDcCompWindowSize windowSize`	Yes	Yes
`CpaDcChecksum checksum`	Yes	Yes

6.1.4.1 Deflate Decompression

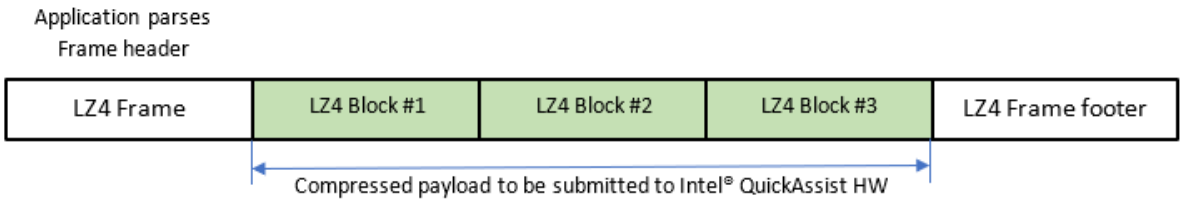
With deflate based format such as Gzip the application is required to skip the GZip header and present to Intel® QAT the first byte of the deflate block.



When the *Deflate Block #3* is processed, the property `endOfLastBlock` in the `CpaDcRqResults` structure will be set to `CPA_TRUE`. This notifies the application that no more data can be decompressed. At this point, the session must be either removed and re-initialized, or reset. With an application handling multi-gzip data, the both the Gzip footer and Gzip header must be skipped.

6.1.4.2 LZ4 Decompresion

When decompressing LZ4 frames, the application is required to parse the frame header to extract `B.Checksum` flag. This flag is used to set the configuration parameter `lz4BlockChecksum` in `CpaDcSessionSetupData`. When decompressing LZ4 frames, the application should not include the frame header nor the frame footer in the source buffer to be processed by Intel® QAT hardware.



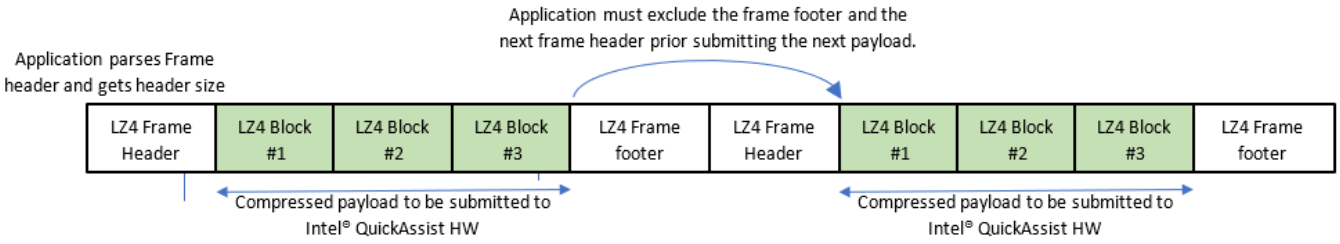
Note: In decompression direction, the property `endOfLastBlock` removes the need for the application to know where the last block ends. The QAT hardware will stop after processing the last block. This applies to both GZIP and LZ4 formats.

6.1.4.3 LZ4 Decompression Limitations

When decompressing LZ4 data compressed without Intel® QAT hardware, it is important to ensure that the compressor limits the history buffer to 32KB. Data compressed using a history buffer larger than 32KB will result with a `CPA_DC_INVALID_DIST (-10)` error code.

6.1.4.4 Multi-frame decompression support

Intel® QAT hardware can decompress a payload that includes multiple frames. This applies to both Gzip and LZ4 formats. The figure below shows how the application must behave to decompress LZ4 multi frame payloads.



6.1.5 Performance Considerations

To enable the application benefiting from the QAT2.0 HW maximal performance, it is recommended to populate all the DIMMs around the CPU sockets in use.

6.1.6 Flush Flags

The table below shows the flush flags that should be used depending on the application's use case. The application programming model should follow this table.

Table 30: Flush Flags

Algorithm	Use Case	Intermediate Request	Last Request
Deflate	Stateless compression	`CPA_DC_FLUSH_FULL`	`CPA_DC_FLUSH_FINAL` to set BFinal
	Stateless decompression	`CPA_DC_FLUSH_FINAL`	
	Stateful decompression (QAT1.X devices only)	`CPA_DC_FLUSH_SYNC`	`CPA_DC_FLUSH_FINAL`
LZ4	Stateless compression with accumulateXxHash = CPA_FALSE	`CPA_DC_FLUSH_FINAL`	`CPA_DC_FLUSH_FINAL`

continues on next page

Table 30 – continued from previous page

Algorithm	Use Case	Intermediate Request	Last Request
	Stateless compression with accumulateXxHash = CPA_TRUE	`CPA_DC_FLUSH_FULL`	`CPA_DC_FLUSH_FINAL`
	Stateless decompression	`CPA_DC_FLUSH_FINAL`	`CPA_DC_FLUSH_FINAL`
LZ4s	Stateless compression	`CPA_DC_FLUSH_FINAL`	`CPA_DC_FLUSH_FINAL`

Note: QAT1.X hardware devices still support stateful decompression operations. QAT2.0 hardware device only supports stateless operations.

6.1.7 Checksums

With the addition of LZ4 algorithm support on QAT2.0, the compression hardware accelerators are now capable to generate XXash32 checksums. QAT2.0 device is now supporting the following checksums:

Table 31: Checksums

Checksum Type	Usage
CRC32	Required for GZip support
Adler32	Required for Zlib support
XXhash32	Required for LZ4 support

Note: In the event the *XXHash32* checksum should be reset, it must be done calling the API `cpaDcResetXXHashState()`.

6.1.8 LZ4s Compressed Data Block format

LZ4s is a variant of LZ4 block format. LZ4s should be considered as an intermediate compressed block format. The LZ4s format is selected when the application sets the `compType` to `CPA_DC_LZ4S` in `CpaDcSessionSetupData`. The LZ4s block returned by the Intel® QAT hardware can be used by an external software post-processing to generate other compressed data formats.

The following table lists the differences between LZ4 and LZ4s block format. LZ4s block format uses the same high-level formatting as LZ4 block format with the following encoding changes:

Table 32: Differences between LZ4 and LZ4s block format

Feature	LZ4	LZ4s
Sequence Header	4-bit copy length 4-bit literal length	Same as LZ4
Copy Length	Length 4-19 bytes (encoding values 0-15), 19 bytes adds an extra byte with value 0x00.	Copy length value of 0 means no copy with this sequence. For Min Match of 3 bytes, Copy length value 1-15 means length 3-17 with 17 bytes adding an extra byte with value 0x00. For Min Match of 4 bytes, Copy length value 1-15 means length 4-18 with 18 bytes adding an extra byte with value 0x00.

Note:

- LZ4 requires a copy/token in every sequence, except the last sequence.
- The last sequence in block does not contain a copy in both LZ4 and LZ4s.

LZ4s encoding creates a single block of compressed data per request. This is different from LZ4 which creates multiple blocks defined by the LZ4 max block size setting. An LZ4s block is only made of LZ4s sequences.

A sequence in LZ4s can contain:

- Only a token.
- Only literals.
- A token and literals.

Any of the sequence types can exist anywhere in the LZ4s block. The last LZ4s sequence in the LZ4s block shall satisfy the end-of-block restrictions outlined in the LZ4 specification.

6.1.8.1 LZ4 Compression Support

With the QAT 2.0 API, the application can create LZ4 frames. This is achieved using APIs `cpaDcGenerateHeader()` and `cpaDcGenerateFooter()`. These APIs are also able to generate GZip and Zlib formats. More information is available in the API reference manual.

The `cpaDcGenerateHeader()` creates a 7 byte LZ4 frame header which includes:

- Magic number 0x184D2204.
- The LZ4 max block size defined in the `CpaDcSessionSetupData`.
- Flag byte as:
  - Version = 1
  - Block independence = 0
  - Block checksum = 0

  - Content size present = 0
  - Content checksum present = 1
  - Dictionary ID present = 0
- Content size = 0
- Dictionary ID = 0
- Header checksum = 1 byte representing the second byte of the XXH32 of the frame descriptor field.

The `cpaDcGenerateFooter()` API must be used after processing all the requests. This API must be called last to append the frame footer. The `cpaDcGenerateFooter()` API creates an 8 byte frame footer adding both the end marker (4 bytes set to 0x00) and the XXHash32 checksum computed by Intel® QAT hardware.

6.1.9 Compress-and-Verify

The Compress and Verify (CnV) feature checks and ensures data integrity in the compression operation of the Data Compression API. This feature introduces an independent capability to verify the compression transformation.

Refer to Intel® QuickAssist Technology Data Compression API Reference Manual.

Note:

- CnV is always enabled via the compression APIs (`cpaDcCompressData()`, `cpaDcCompressData2()`, `cpaDcNsCompressData()`, `cpaDcDpEnqueueOp()`).
- CnV supports compression operations only.
- The `compressAndVerify` flag in the `CpaDcDpOpData` structure should be set to `CPA_TRUE` when using the `cpaDcDpEnqueueOp()` or `cpaDcDpEnqueueOpBatch()` API.

6.1.9.1 Compress and Verify Error log in Sysfs

The implementation of the Compress and Verify solution keeps a record of the CnV errors that have occurred since the driver was loaded. The error count is provided on a per Acceleration Engine basis.

The path to the CnV error log is: `/sys/kernel/debug/qat_<qat_device>_<Bus>\:<device>.<function>/cnv_errors`

Each Acceleration Engine keeps a count of the CnV errors. The CnV error counter is reset when the driver is loaded. The tool also reports the last error type that caused a CnV error.

6.1.9.2 Compress and Verify and Recover (CnVnR)

The Compress and Verify and Recover (CnVnR) feature allows a compression error to be recovered in a seamless manner. It is supported in both the Traditional and in the Data Plane APIs.

The CnVnR feature is an enhancement of the Compress and Verify (CnV) solution. When a compress and verify error is detected, the Intel® QAT software will do a correction without returning a CnV error to the application.

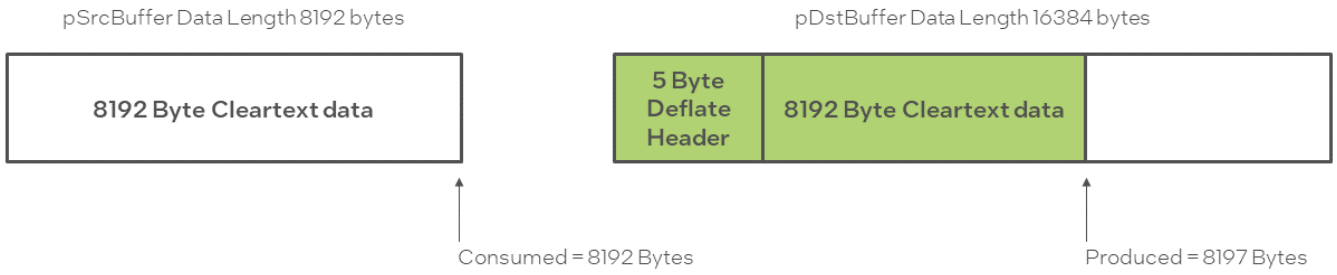
When a recovery occurs, `CpaDcRqResults.status` will return `CPA_DC_OK` or `CPA_DC_OVERFLOW` and the destination buffer will hold valid deflate data.

The application can find out if CnVnR is supported by querying the instance capabilities via the `cpaDcQueryCapabilities` API. On completion, the `compressAndVerifyAndRecover` property of the `CpaDcInstanceCapabilitie: ..\image:: ./s` structure will be set to `CPA_TRUE` if the feature is supported.

Table 33: Compress and Verify and Recover (CnVnR) Behaviors

API	CnVnR Behavior
`cpaDcCompressData`	Enabled by default, no option to disable.
`cpaDcCompressData2`	CnVnR is enabled when `compressAndVerifyAndRecover` property is set to `CPA_TRUE` in `CpaDcDpOpData` structure.
`cpaDcNsCompressData`	CnVnR is enabled by default.
`cpaDcDecompressData`	Not applicable
`cpaDcDecompressData2`	Not applicable
`cpaDcNsDecompressData`	Not applicable
`cpaDcDpEnqueueOp`	CnVnR is enabled when `compressAndVerifyAndRecover` property is set to `CPA_TRUE` in `CpaDcDpOpData` structure.
`cpaDcDpEnqueueOpBatch`	CnVnR is enabled when `compressAndVerifyAndRecover` property is set to `CPA_TRUE` in `CpaDcDpOpData` structure.

When a CnV recovery takes place, the Intel® QAT software creates a stored block out of the input payload that could not be compressed. The maximal size of a stored block allowed by the deflate standard is 65,535 bytes.



When a stored block is created, the DEFLATE header specifies that the data is uncompressed so that the decompressor does not attempt to decode the cleartext data that follows the header. The size of a stored block can be defined as: *Stored block size = Source buffer size + 5 Bytes* (used for the deflate header).

The recovery behaves differently on QAT2.0 than on QAT1.X devices. With QAT1.X devices, the recovery creates only one stored block, If the stored block size exceeds 65,535 bytes, the Intel® QAT solution creates one stored block of 65,535 bytes and `CpaDcRqResults.status` returns `CPA_DC_OVERFLOW`. On QAT2.0 device, when the recovery takes place, multiple stored blocks are created. This improvement was added to avoid the application having to handle the overflow.

CnV Recovery with LZ4 compression

When LZ4 compression is used, QAT software will generate an uncompressed LZ4 block in the event of a recovery. The LZ4 uncompressed block will have bit <31> set in the block header followed by the cleartext in the data section of the block.

CnV Recovery with LZ4s compression

LZ4s algorithm is an Intel® specific format. LZ4s payloads do not have a block header like LZ4. When a CnV recovery occurs, the source data will be copied to the destination and `dataUncompressed` property will be set to `CPA_TRUE` in `CpaDcRqResults` structure.

Counting Recovered Compression Errors

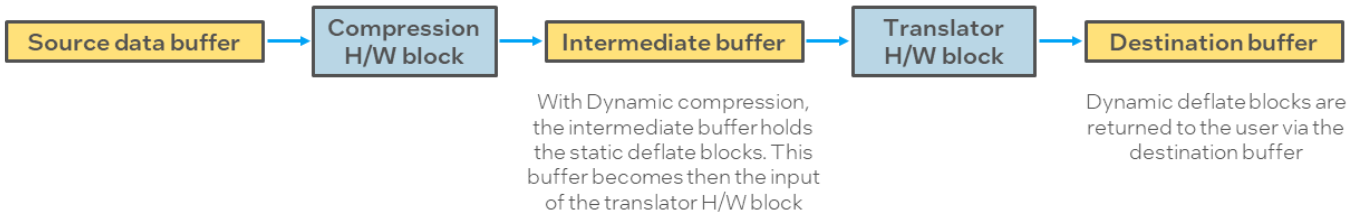
The Intel® QAT API has been updated to allow the application to track recovered compression errors. The `CpaDcStats` data structure has a new property called `numCompCnvErrorsRecovered` that is incremented every time a compression recovery happens.

The compression recovery process is agnostic to the application.

`CpaDcRqResults.status` returns `CPA_DC_OK` when a compression recovery takes place. The only way to know if a compression recovery took place on the current request is to call the `cpaDcGetStats()` API and to monitor `CpaDcStats.numCompCnvErrorsRecovered`.

6.1.10 Dynamic Compression

Dynamic compression involves feeding the data produced by the compression hardware block to the translator hardware block.



When the application selects the Huffman type to `CPA_DC_HT_FULL_DYNAMIC` in the session and auto-select best feature is set to `CPA_DC_ASB_DISABLED`, the compression service may not always produce a deflate stream with dynamic Huffman trees.

When using QAT2.0 device, it is no longer required to allocate intermediate buffers. The API `cpaDcGetNumIntermediateBuffers()` returns 0. As a good programming practise, the application should still call `cpaDcGetNumIntermediateBuffers()` and ensure that the number of intermediate buffers returned is 0.

6.1.11 Maximum Expansion with Auto Select Best Feature (ASB)

Compressing some input data may lead to a lower-than-expected compression ratio. This is because the input data may not be very compressible. To achieve a maximum compression ratio, the acceleration unit provides an auto select best (ASB) feature.

With QAT1.X devices, the Intel® QuickAssist Technology hardware will first execute static compression followed by dynamic compression and then select the output that yields the best compression ratio.

With QAT2.0 devices, the behaviour is different. ASB features chooses between a static and a stored block. ASB features with choose the block type that offers the best compression ratio.

Regardless of the ASB setting selected, dynamic compression will only be attempted if the session is configured for dynamic compression.

Table 34: ASB Settings

Setting	Description
`CPA_DC_ASB_DISABLED`	ASB mode is disabled.
`CPA_DC_ASB_ENABLED`	ASB mode is enabled.

Note:

- Setting ASB mode to `CPA_DC_ASB_ENABLED`, corresponds to the setting `CPA_DC_ASB_UNCOMP_STATIC_DYNAMIC_WITH_STORED_HDRS`.
- These ASB modes have been deprecated:
  - `CPA_DC_ASB_STATIC_DYNAMIC`
  - `CPA_DC_ASB_UNCOMP_STATIC_DYNAMIC_WITH_STORED_HDRS`
  - `CPA_DC_ASB_UNCOMP_STATIC_DYNAMIC_WITH_NO_HDRS`
- Based on the ASB settings, the produced data returned in the `CpaDcRqResults` structure will vary.

6.1.12 Maximum Compression Expansion

To facilitate the programming model of the application, Intel® added a new set of APIs that return the size of the destination according to the algorithm used.

These APIs are:

- `cpaDcDeflateCompressBound()`
- `cpaDcLZ4CompressBound()`
- `cpaDcLZ4SCompressBound()`

This new set of APIs will return to the application the size of the destination buffer that must be allocated to avoid an overflow exception. Each function initializes the `outputSize` parameter. The `outputSize` parameter takes into account the maximal expansion that the compressed data size can reach. The returned `outputSize` value must be used both to allocate the size of the `pData` and to initialize the `dataLenInBytes` in the `CpaBufferList` structure.

Note: Each one of the `compressBound()` APIs accepts as a parameter the instance handle. The instance handle is used internally by the library to determine on which hardware version the instance lives on. The size of the destination buffer to be allocated depends on the hardware generation the instance lives on.

6.1.13 No Session API

The no session API is a simplification of the existing compression and decompression APIs that does not require the application to create/remove a session. Instead the parameters that would normally be set when creating a session are passed into the compress/decompress APIs via a `CpaDcNsSetupData` structure. The no session API is especially useful to simplify existing applications as sessions no longer need to be created/tracked/removed. The no session API can be thought of as a "one shot" API and is intended for use cases where all the data to be compressed or decompressed for the current job is being submitted as one request. In addition to the simpler protocol, the API has a smaller memory footprint.

The no session API consists of the following API calls:

- `cpaDcNsCompressData()`
- `cpaDcNsDecompressData()`
- `cpaDcNsGenerateHeader()`
- `cpaDcNsGenerateFooter()`

The `cpaDcNsCompressData()` and `cpaDcNsDecompressData()` functions are very similar to the `cpaDcCompressData2()` and `cpaDcDecompressData2()` functions. Instead of passing in a `CpaDcSessionHandle`, a `CpaDcNsSetupData` structure and a `CpaDcCallbackFn` are passed in. The `CpaDcCallbackFn` is the user callback to be called on request completion when running asynchronously. For synchronous operation `CpaDcCallbackFn` must be set to `NULL`.

The no session API will work with all versions of QAT hardware but does not support stateful operation as without a session no state can be maintained between requests.

It is still possible to seed checksums for CRC32 and Adler32 by setting the `checksum` field of the `CpaDcRqResults` to the seed checksum value before submission. This will allow an overall checksum to be maintained across multiple submissions. For LZ4, checksum seeding is not supported. If checksums need to be maintained between LZ4 requests then the session based API must be used. The no session API supports data integrity checksums but as stateful operation is not supported the integrity checksums will always be for the current request only.

The no session API does support stateless overflow in the compression direction only like the session based API. In that case `consumed`, `produced` and `checksum` fields within the `CpaDcRqResults` structure will be valid when a status of `CPA_DC_OVERFLOW` is returned. It is the application's responsibility to arrange data

buffers for the next submission, ensure the checksum is seeded and maintain an overall count of the bytes consumed if footer generation is required. For performance reasons it is recommended that the compressBound API is used to size the destination buffer correctly to avoid overflow. It is necessary for an application to seed the checksum whether it wishes to continue the series of requests or to start a new one, only in the latter case, the seed is 0 for CRC32 and 1 for Adler32.

The no session API does not support setting the `sessDirection` field of the `CpaDcNsSetupData` structure to `CPA_DC_DIR_COMBINED`. In addition, it does not support setting the `sessState` field to `CPA_DC_STATEFUL`, or the `flushFlag` field of the `CpaDcOpData` structure to `CPA_DC_FLUSH_NONE` or `CPA_DC_FLUSH_SYNC`.

The `cpaDcNsGenerateHeader()` and `cpaDcNsGenerateFooter()` functions are also very similar to the session based equivalents but take a `CpaDcNsSetupData` instead of a `CpaDcSessionHandle`. For `cpaDcNsGenerateFooter()` an additional parameter is required called `count` that contains the overall length of the uncompressed input data. For most cases this will be the `consumed` value from the single submission contained in the `CpaDcRqResults` structure but in cases where multiple submissions represent the overall file then it is the application's responsibility to maintain the overall count of consumed bytes.

6.1.14 Compression Levels

Table 35: Compression Levels

Level	lvl_enum	QAT 2.0 (Deflate, iLZ77, LZ4, LZ4s)	QAT 1.7/1.8 (Deflate)
1	CPA_DC_L1	2 (HW_L1)	DEPTH_1
2	CPA_DC_L2		DEPTH_4
3	CPA_DC_L3		DEPTH_8
4	CPA_DC_L4		DEPTH_16
5	CPA_DC_L5		
6	CPA_DC_L6	8 (HW_L6)	
7	CPA_DC_L7		
8	CPA_DC_L8		
9	CPA_DC_L9	16 (HW_L9)	
10	CPA_DC_L10		Unsupported. Will be rejected at the API.
11	CPA_DC_L11		
12	CPA_DC_L12		
> 12		Unsupported. Will be rejected at the API.	

6.1.15 Compression Status Codes

The `CpaDcRqResults` structure should be checked for compression status codes in the `CpaDcReqStatus` data field. The mapping of the error codes to the enums is included in the `quickassist/include/dc/cpa_dc.h` file.

6.1.16 Intel® QuickAssist Technology Compression API Errors

The Intel® QuickAssist Technology Compression APIs that send requests to the compression hardware can return the error codes shown in *Compression API Errors*.

These APIs are:

- `cpaDcCompressData()`
- `cpaDcCompressData2()`
- `cpaDcNsCompressData()`
- `cpaDcDecompressData()`
- `cpaDcDecompressData2()`
- `cpaDcNsDecompressData()`
- `cpaDcDpEnqueueOp()`
- `cpaDcDpEnqueueOpBatch()`

Note: Decompression issues in may also apply to the compression use case due to potential issues encountered during a Compress-and-Verify operation. In this case, the file(s) `/sys/kernel/debug/qat_*/cnv_errors` may show these nested errors. In some cases, the suggested corrective action may need to be to store the block uncompressed or to compress the block with software.

6.1.16.1 Compression API Errors

Table 36: Compression API Errors

Error Code	Error Type	Description	Suggested Corrective Action(s)
0	`CPA_DC_OK`	No error detected by compression hardware.	None.
-1	`CPA_DC_INVALID_BLOCK_TYPE`	Invalid block type (type = 3); invalid input stream detected for decompression	Decompression error. Discard output. For a stateless session, resubmit affected request. For a stateful session, abort the session calling `CpaDcRemoveSession()`.

continues on next page

Table 36 – continued from previous page

Error Code	Error Type	Description	Suggested Corrective Action(s)
-2	CPA_DC_BAD_STORED_BLOCK_LEN	Stored block length did not match one's complement; invalid input stream detected	Decompression error. Discard output. For a stateless session, resubmit affected request. For a stateful session, abort the session calling `CpaDcRemoveSession()`.
-3	CPA_DC_TOO_MANY_CODES	Too many length or distance codes; invalid input stream detected	Decompression error. Discard output. For a stateless session, resubmit affected request. For a stateful session, abort the session calling `CpaDcRemoveSession()`.
-4	CPA_DC_INCOMPLETE_CODE_LENS	Code length codes incomplete: invalid input stream detected	Decompression error. Discard output. For a stateless session, resubmit affected request. For a stateful session, abort the session calling `CpaDcRemoveSession()`.
-5	CPA_DC_REPEATED_LENS	Repeated lengths with no first length; invalid input stream detected	Decompression error. Discard output. For a stateless session, resubmit affected request. For a stateful session, abort the session calling `CpaDcRemoveSession()`.
-6	CPA_DC_MORE_REPEAT	Repeat more than specified lengths; invalid input stream detected	Decompression error. Discard output. For a stateless session, resubmit affected request. For a stateful session, abort the session calling `CpaDcRemoveSession()`.

continues on next page

Table 36 – continued from previous page

Error Code	Error Type	Description	Suggested Corrective Action(s)
-7	CPA_DC_BAD_LITLEN_CODES	Invalid literal/length code lengths; invalid input stream detected	Decompression error. Discard output. For a stateless session, resubmit affected request. For a stateful session, abort the session calling `CpaDcRemoveSession()`.
-8	CPA_DC_BAD_DIST_CODES	Invalid distance code lengths; invalid input stream detected	Decompression error. Discard output. For a stateless session, resubmit affected request. For a stateful session, abort the session calling `CpaDcRemoveSession()`.
-9	CPA_DC_INVALID_CODE	Invalid literal/length or distance code in fixed or dynamic block; invalid input stream detected	Decompression error. Discard output. For a stateless session, resubmit affected request. For a stateful session, abort the session calling `CpaDcRemoveSession()`.
-10	CPA_DC_INVALID_DIST	Distance is too far back in fixed or dynamic block; invalid input stream detected	Decompression error. Discard output. For a stateless session, resubmit affected request. For a stateful session, abort the session calling `CpaDcRemoveSession()`. If the error is observed with LZ4 decompression, ensure that the compressor has a history buffer limited to 32KB.

continues on next page

Table 36 – continued from previous page

Error Code	Error Type	Description	Suggested Corrective Action(s)
-11	CPA_DC_OVERFLOW	Overflow detected. This is not an error, but an exception. Overflow is supported and can be handled.	Resubmit with a larger output buffer when appropriate. With decompression executed on QAT2.0, the application is required to resubmit the compressed data with a larger destination buffer.
-12	CPA_DC_SOFTERR	Other non-fatal detected.	Discard output. For a stateless session, resubmit affected request. For a stateful session, abort the session calling `CpaDcRemoveSession()`.
-13	CPA_DC_FATALERR	Fatal error detected.	Discard output and abort the session calling `CpaDcRemoveSession()`.
-14	CPA_DC_MAX_RESUBMITERR	On an error being detected, the firmware attempted to correct and resubmitted the request, however, the maximum resubmit value was exceeded. Maximal value is internally set in the firmware to 10 attempts. This is a QAT1.6 error only. This error code is considered as a fatal error.	Discard output and abort the session calling `CpaDcRemoveSession()`.

continues on next page

Table 36 – continued from previous page

Error Code	Error Type	Description	Suggested Corrective Action(s)
-15	CPA_DC_INCOMPLETE_FILE_ERR	QAT1.X device can report this error with Deflate decompression. However, it is not exposed to the application. The input file is incomplete. This indicates that the request was submitted with a CPA_DC_FLUSH_FINAL. However, a BFINAL bit was not found in the request. QAT2.0 can return this error code to the application during LZ4 decompression. This error is returned when a LZ4 block is incomplete.	No corrective action is required as it is not exposed to the application.
-16	CPA_DC_WDOG_TIMER_ERR	The request was not completed as a watchdog timer hardware event occurred. With QAT2.0 this error can be triggered by an internal parity error.	Discard output and resubmit the affected request.
-17	CPA_DC_EP_HARDWARE	This is a recoverable error. Request was not completed as an end point hardware error occurred (for example, a parity error).	Discard output and abort the session calling `CpaDcRemoveSession()`.
-18	CPA_DC_VERIFY_ERR	Compress and Verify (CnV). This is a compression direction error only. During the decompression of the compressed payload, an error was detected and the deflate block produced is invalid.	Discard output; resubmit affected request.

continues on next page

Table 36 – continued from previous page

Error Code	Error Type	Description	Suggested Corrective Action(s)
-19	CPA_DC_EMPTY_DYM_BLK	Decompression request contained an empty dynamic stored block (not supported).	Discard output.
-20	CPA_DC_CRC_INTEG_ERR	A data integrity CRC error was detected.	Discard output.
-93	CPA_DC_LZ4_MAX_BLOCK_SIZE_EXCEEDED	LZ4 max block size exceeded.	Discard output.
-95	CPA_DC_LZ4_BLOCK_OVERFLOW_ERR	LZ4 block overflow.	Discard output.
-98	CPA_DC_LZ4_TOKEN_IS_ZERO_ERR	LZ4 Decoded token offset or token length is zero.	Discard output.
-100	CPA_DC_LZ4_DISTANCE_OUT_OF_RANGE_ERR	LZ4 distance out of range for the len/ distance pair.	Discard output.

Note:

- Except for the errors CPA_DC_OK, CPA_DC_OVERFLOW, CPA_DC_FATALERR, CPA_DC_MAX_RESUBMITERR, CPA_DC_WDOG_TIMER_ERR, CPA_DC_VERIFY_ERR, and CPA_DC_EP_HARDWARE_ERR, the rest of the error codes can be considered as invalid input stream errors.
- When the suggested corrective action is to discard the output, it implies that the application must also ignore the consumed data, the produced data, and the checksum values.

6.1.17 Overflows Errors

This table describes the behavior of the Intel® QAT compression service when an overflow occurs during a compression or decompression operation. It also describes the expected behavior of an application when an overflow occurs.

Table 37: Overflows Errors

	Operation	Overflow Supported	Input Data Consumed	Valid Data Produced?	Status Returned in Results	Note
Traditional API	Stateless compression	YES	Possible - indicated in results consumed field	Possible - indicated in results produced field	-11	Overflow is considered as an exception
	Stateless decompression	NO	NO	NO	-11	Overflow is considered as an error
	Stateful decompression	YES on QAT1.x devices NO on QAT2.x devices	Possible - indicated in results consumed field	Possible - indicated in results produced field	-11	Overflow is considered as an exception on QAT1.x devices. QAT2.x does not support stateful decompression.
Data Plane API	Stateless compression	NO	NO	NO	-11	Overflow is considered as an error
	Stateful decompression	NO	NO	NO	-11	Overflow is considered as an error

6.1.17.1 Traditional API Overflow Exception

Stateless sessions support overflow as an exception for traditional API in the compression direction only. This means that the application can rely on the `cpaDcRqResults.consumed` to resubmit from where the overflow occurred. An overflow in the decompression direction must be treated as an error.

In this case, the application must resubmit the request with a larger buffer as described in the procedure for handling overflow errors. For stateful sessions, overflow is supported only in the decompression direction.

6.1.17.2 Data Plane API Overflow Error

The Data Plane API considers overflow status as an error. If an overflow occurs with the data plane API, the driver will output the following error message to the user:

```
Unrecoverable error: stateless overflow. You may need to increase the size of
your destination buffer
```

In this case, `cpaDcRqResults.consumed`, `.produced` and `.checksum` should be ignored. If length and checksum are required, they must be tracked in the application, because they are not maintained in the session.

6.1.17.3 Handling Overflow Errors

Resubmit the request with the following data:

- Use the same source buffer.
- Allocate a bigger destination buffer. It is recommended to use the `compressBound()` APIs in compression direction.
- If the overall checksum needs to be maintained, insert the checksum from the previous successful request into the `cpaDcRqResults` struct.

6.1.17.4 Compression Overflows in a Virtual Environment

In a virtual environment, the guest does not download the firmware. Only the host downloads the firmware. As a consequence, if the guest runs a newer Intel® QAT driver than the host, the guest application might experience false CNV errors. The correct course of action would be to update the host with the latest Intel® QAT driver.

6.1.17.5 Avoiding Compression Overflow Exceptions

Overflow exceptions happen for 2 reasons:

1. The application allocated a destination buffer that was too small to receive the compressed data.
2. A recovery occurred after a compress and verify error with an input payload greater than 65,535 bytes if the instance lives on a QAT1.X device.

To minimize the impact of resubmitting data after and overflow exception, the API `cpaDcDeflateCompressBound()` has been added to the Intel® QAT driver. A detailed explanation of *compressBound* APIs is provided in the *Maximum Compression Expansion* section.

6.1.18 Integrity Checksums

Integrity checksums are an additional method for payload verification throughout the compression/decompression lifecycle. They may be used to verify corruption has not happened when sending data to and from the Intel® QuickAssist HW, or for example the integrity checksums may be stored by an application along with the compressed data and used to detect corruption in the future without needing to decompress the data.

They should not be confused with the Gzip/Zlib/LZ4 footer checksums of CRC32, Adler32 and Xxhash32 that are calculated over the uncompressed input data only.

Integrity checksums use an additional structure that is the application's responsibility to allocate, maintain, and free. The structure is `cpaCrcData` and contains the following fields:

Table 38: Integrity Checksums

cpaCrcData Fields	Description
`Cpa32U crc32`	This is the existing CRC32 for the footer calculated across the uncompressed data in either the source or dest buffer depending whether it is a compress or decompress operation. This is the same as the value returned in the cpaDcRqResults checksum field.
`Cpu32U adler32`	This is the existing Adler32/Xxhash32 for the footer calculated across the uncompressed data in either the source or dest buffer depending whether it is a compress or decompress operation. This is the same as the value returned in the cpaDcRqResults checksum field.
`CpaIntegrityCrc integrityCrc`	This field contains the QAT 1.8 integrity checksums that consist of two 32bit CRC32's. These are calculated on the input data to the request within the HW and on the output data from the request within the HW.
`CpaIntegrityCrc64b integrityCrc64b`	This field contains the QAT 2.0 integrity checksums that consist of two 64 bit checksums. CPM 2.0 uses CRC64 by default for these checksums. The checksums are calculated on the input data to the request within the HW and on the output data from the request within the HW.

Once allocated, a pointer to the `cpaCrcData` structure must be assigned to the `pCrcData` field of the requests `CpaDcOpData` structure. The `cpaCrcData` structure assigned to the `pCrcData` pointer should be treated in the same way as the destination buffer, not freed until the request has completed, and not shared across requests if running asynchronously.

The integrity checksum feature itself is enabled on a per request basis by setting the `integrityCrcCheck` field contained in the `CpaDcOpData` structure to `CPA_TRUE`. Integrity checksums are available on the Traditional API including No Session requests, but are not available on the Data Plane API. Integrity checksums are calculated across only the current request in QAT 2.0. With QAT 1.8 it is possible to seed the integrity checksums on stateful decompression requests by reusing the same `cpaCrcData` structure on the subsequent request without resetting the contents. For QAT 1.8 stateful decompression requests it is the application's responsibility to allocate the `cpaCrcData` structure and keep it allocated for the lifetime of the session. Integrity checksums are not available on devices prior to QAT 1.8.

6.1.18.1 Verify HW Integrity CRC's

There is an additional feature to integrity checksums that can be enabled to automatically check that no corruption to data buffers has occured during transport to and from the Intel® QAT HW. This works by calculating integrity checksums across the source and destination buffers within the Intel® QAT API, and comparing the checksums with those generated within the Intel® QAT HW. Any discrepencies will result in a status of `CPA_DC_INTEG_ERR` being returned within the `cpaDcRqResults` structure. These additional checksums are calculated in SW using the CPU and have a cost in terms of performance. In order to enable the *Verify HW Integrity CRC* feature on a per request basis the `verifyHwIntegrityCrcs` field contained in the `cpaDcOpdata` structure needs to be set to `CPA_TRUE`. Additionally the `integrityCrcCheck` field must be enabled and a `cpaCrcData` structure allocated and a pointer to it must be assigned to the `pCrcData` field.

6.1.19 Data Compression Applications

Data compression can be used as part of application delivery networks, data de-duplication, as well as in a number of crypto applications, for example, VPNs, IDS/IPS and so on.

6.1.19.1 Compression for Storage

In a time when the amount of online information is increasing dramatically, but budgets for storing that information remain static, compression technology is a powerful tool for improved information management, protection and access.

Compression appliances can transparently compress data such that clients can keep between two- and five-times more data online and reap the benefit of other efficiencies throughout the data lifecycle. By shrinking the primary data, all subsequent copies of that data, such as backups, archives, snapshots, and replicas are also compressed. Compression is the newest advancement in storage efficiency.

Storage compression appliances can shrink primary online data in real time, without performance degradation. This can significantly lower storage capital and operating expenses by reducing the amount of data that is stored, and the required hardware that must be powered and cooled.

Compression can help slow the growth of storage, reducing storage costs while simplifying both operations and management. It also enables organizations to keep more data available for use, as opposed to storing data offsite or on harder-to-access media (such as tape). Compression algorithms are very compute-intensive, which is one of the reasons why the adoption of compression techniques in mainstream applications has been slow.

As an example, the DEFLATE Algorithm, which is one of the most used and popular compression techniques today, involves several compute-intensive steps: string search and match, sort logic, binary tree generation, Huffman Code generation. Intel® QAT devices in the platforms described in this manual provide acceleration capabilities in hardware that allow the CPU to offload the compute-intensive DEFLATE algorithm operations, thereby freeing up CPU cycles for other networking, encryption, or other value-add operations.

6.1.19.2 Data Deduplication and WAN Acceleration

Data Deduplication and WAN Acceleration are coarse-grain data compression techniques centered around the concept of single-instance storage. Identical blocks of data (either to be stored on disk or to be transferred across a WAN link) are only stored/moved once, and any further occurrences are replaced by a reference to the first instance.

While the benefits of deduplication and WAN acceleration obviously depend on the type of data, multi-user collaborative environments are the most suitable due to the amount of naturally occurring replication caused by forwarded emails and multiple (similar) versions of documents in various stages of development.

Deduplication strategies can vary in terms of inline vs post-processing, block size granularity (file-level only, fixed block size or variable block-size chunking), duplicate identification (cryptographic hash only, simple CRC followed by byte-level comparison or hybrids) and duplicate look-up (for example, Bloom filter based index).

Cryptographic hashes are the most suitable techniques for reliably identifying matching blocks with an improbably low risk for false positives, but they also represent the most compute-intensive workload in the application. As such, the cryptographic acceleration services offered by the hardware through the Intel® QAT Cryptographic API can be used to considerably improve the throughput of deduplication/WAN acceleration applications. Additionally, the compression/decompression acceleration services can be used to further compress blocks for storage on disk, while optionally encrypting the compressed contents.

6.2 Cryptographic Services

6.2.1 Introduction

Intel® QuickAssist Technology (Intel® QAT) accelerates cryptographic workloads by offloading the data to hardware capable of optimizing those functions. This makes it easier for developers to integrate built-in cryptographic accelerators into network and security applications.

Symmetric cryptography algorithms include:

- Cipher operations (AES, DES, 3DES, ARC4, CHA-CHA, SM4).
- Wireless (Kasumi, Snow, 3G).
- Hash/Authenticate operations (SHA-1, MD5, SHA-2, SHA-3, SHAKE).
- Authentication (HMAC, AES-XCBC, AES-CCM).

Public key algorithms include:

- RSA operation.
- Diffie-Hellman operation.
- Digital signature standard operation.

- Key derivation operation.
- Elliptic curve cryptography (ECDSA and ECDH).
- Prime number testing.

6.2.1.1 Supported Cipher Algorithms

The following table provides details on supported cipher algorithms for each platform.

Note:

- `cpaCySymInitSession()` returns error status of `CPA_STATUS_UNSUPPORTED` if cipher algorithm is not supported.
- The QAT2.0 driver has not been updated to enable the Opt-In functionality. This will be added in a future release.

Table 39: Supported Cipher Algorithms

Algorithm	QAT 1.7x	QAT 1.8	QAT 2.0
NULL	Yes	Yes	Yes
ARC4	Opt-in	Opt-in	No
AES-ECB	Opt-in	Opt-in	Opt-in
AES-CBC	Yes	Yes	Yes
AES-CTR	Yes	Yes	Yes
AES-CCM	Yes	Yes	Yes
AES-GCM	Yes	Yes	Yes
AES-F8	Opt-in	Opt-in	Opt-in
AES-XTS	Yes	Yes	Yes
DES-ECB	Opt-in	Opt-in	No
DES-CBC	Opt-in	Opt-in	No
3DES-ECB	Opt-in	Opt-in	No
3DES-CBC	Opt-in	Opt-in	No
3DES-CTR	Opt-in	Opt-in	No
KASUMI-F8	Yes	Yes	No
SNOW3G-UEA2	Yes	Yes	No
ZUC-EEA3	Yes	Yes	No
CHACHA	No	Yes	Yes
SM4-ECB	No	Opt-in	Opt-in
SM4-CBC	No	Yes	Yes
SM4-CTR	No	Yes	Yes

6.2.1.2 Supported Hash/Authenticate Algorithms

The following table provides details on supported hash algorithms for each platform.

Note:

- `cpaCySymInitSession()` returns error status of `CPA_STATUS_UNSUPPORTED` if hash algorithm is not supported.
- The QAT2.0 driver has not been updated to enable the Opt-In functionality. This will be added in a future release.

Table 40: Supported Hash/Authenticate Algorithms

Algorithm	QAT 1.7x	QAT 1.8	QAT 2.0
MD5	Opt-in	Opt-in	No
SHA1	Opt-in	Opt-in	Opt-in
SHA224	Opt-in	Opt-in	Opt-in
SHA256	Yes	Yes	Yes
SHA384	Yes	Yes	Yes
SHA512	Yes	Yes	Yes
SHA3-224	No	Opt-in	Opt-in
SHA3-256	Yes	Yes	Yes
SHA3-384	No	Yes	Yes
SHA3-512	No	Yes	Yes
AES-XCBC	Yes	Yes	Yes
AES-CBC_MAC	Yes	Yes	Yes
AES-CCM	Yes	Yes	Yes
AES-GCM	Yes	Yes	Yes
AES-GMAC	Yes	Yes	Yes
AES-CMAC	Yes	Yes	Yes
KASUMI-F9	Yes	Yes	No
SNOW3G-UIA2	Yes	Yes	No
ZUC-EIA3	Yes	Yes	No
POLY	No	Yes	Yes
SM3	No	Yes	Yes

6.2.1.3 Supported Public Key Algorithms

The following table provides details on supported asymmetric algorithms for each platform.

Note: QAT Public Key functions will return error status of `CPA_STATUS_UNSUPPORTED` if algorithm is not supported.

Table 41: Supported Public Key Algorithms

Algorithm	QAT 1.7x	QAT 1.8	QAT 2.0
RSA-1024	Opt-in	Opt-in	Opt-in
RSA-2048	Yes	Yes	Yes
RSA-3072	Yes	Yes	Yes
RSA-4096	Yes	Yes	Yes
RSA-8192	No	No	Yes
SM2	No	Yes	Yes
ECDH Point Multiply	Yes	Yes	Yes
ECDSA Sign	Yes	Yes	Yes
ECDSA Verify	Yes	Yes	Yes
x25519	Yes	Yes	Yes
x448	Yes	Yes	Yes

6.2.2 Cryptography Applications

Cryptography applications supported by the platforms described in this manual include, but are not limited to:

- *Virtual Private Networks*
- *Encrypted Storage*
- *Web Proxy Appliances*

6.2.2.1 IPsec and SSL VPNs

Virtual Private Networks (VPNs) allow for private networks to be established over the public Internet by providing confidentiality, integrity and authentication using cryptography. VPN functionality can be provided by a standalone security gateway box at the boundary between the trusted and untrusted networks. It is also commonly combined with other networking and security functionality in a security appliance, or even in standard routers.

VPNs are typically based on one of two cryptographic protocols, either IPsec or Datagram Transport Layer Security (DTLS). Each has its advantages and disadvantages.

One of the most compute-intensive aspects of a VPN is the cryptographic processing required to encrypt/decrypt traffic for confidentiality, to perform cryptographic hash functionality for authentication

and to perform public key cryptography, based on modular exponentiation of large numbers or elliptic curve cryptography as part of key negotiation and exchange. The accelerator provides cryptographic acceleration that can offload this computation from the CPU, thereby freeing up CPU cycles to perform other networking, encryption, or other value-add applications.

The Intel® QAT Endpoint offers its acceleration services through an API, called the Intel® QAT Cryptographic API. This can be invoked from the Linux* kernel or from Linux* user space as well as from other operating systems. Intel® also provides plugins to enable many of the PCH's cryptographic services to be accessed through open source cryptographic frameworks, such as the Linux* kernel crypto framework/API (also known as the `scatterlist` API) and OpenSSL* libcrypto* (through its EVP API). This facilitates ease of integration with certain open source implementations of protocol stacks, such as the Linux* kernel's native IPsec stack (called `NETKEY`) or with OpenVPN* (an open source SSL VPN implementation).

6.2.2.2 Encrypted Storage

In recent years, cases of lost laptops containing sensitive information have made the headlines all too frequently. Full disk encryption has become a standard procedure for many corporate PCs. Safe-guarding critical data however is not just a necessity in the client space, it is also a necessity in the data center.

Enterprise-class storage appliances achieve throughput rates in excess of 50 Gbps. Several high-profile cases of data theft have triggered updates to government regulations and industry standards. These regulations/standards now require protection of data-at-rest for applications involving sensitive data such as medical and financial records, typically using strong encryption. The high computational cost of adding encryption to storage appliances makes offload solutions an attractive value proposition.

Several complimentary standards exist for the encryption of data-at-rest, which, when combined with traditional network security protocols such as IPsec or SSL/TLS, provide an end-to-end encrypted storage solution, even for data-in-flight.

The IEEE* Security in Storage working group is developing the IEEE 1619 series of standards that deal with cipher algorithms for disk and tape storage devices (AES in CCM and GCM modes). The cryptographic acceleration services of platforms that use the Intel® QAT Endpoints are ideally suited for long-term encrypted storage solutions implementing the IEEE 1619.1 standard, by providing acceleration of the AES-256 cipher in CBC, CCM, and GCM modes and HMAC authentication using SHA-1, SHA-256 and SHA-512 hashes.

The Trusted Computing Group's (TCG) Storage Working Group does not prescribe a particular set of algorithms for the disk encryption. Instead, it defines several Storage Subsystem Classes (SSC) for various usage models, which define services such as enrollment and connection, protected storage (an extension of Trusted Platform Module (TPM)), locking, logging, cryptographic services, authorization, and firmware updates. The cryptographic acceleration services of the platform can help by providing the highest level of encryption for authenticating the host to trusted peripherals implementing the TCG storage standards.

6.2.2.3 Web Proxy Appliances

Historically, Web Proxy appliances have evolved to present a public or intermediary interface for clients seeking resources from other servers, providing services such as web page caching and load balancing. These appliances are located at the edge of the network, typically at network gateways. Due to their centralized presence in the network, Web Proxy appliances today (referred to with several different names, such as Application Delivery Controllers, Reverse Proxy, and so on) have become a collection of services that include:

- Application Load Balancing (L4-L7)
- SSL Acceleration
- WAN Acceleration
- Caching
- Traffic Management
- Web Application Firewall

SSL and WAN acceleration have become common place capabilities of the Web Proxy appliance, requiring compute intensive algorithms for cryptography (SSL) and compression (WAN acceleration). Intel® QAT devices on the platforms described in this manual provide acceleration of asymmetric cryptography (RSA is the most commonly used key negotiation algorithm in SSL), symmetric cryptography (all algorithms defined in the TLS RFCs can be accelerated with the PCH) and compression (DEFLATE algorithm). With the prominence of Web Proxy appliances in typical networks, this use case has applications from cloud computing to small web server deployments.

7 Supported APIs

The supported APIs are classified in two categories:

- Intel QuickAssist Technology APIs
- Additional APIs

Details on the APIs are included in the following sections.

7.1 Intel QuickAssist Technology APIs

The platforms described in this manual support the following Intel® QAT API libraries:

- **Cryptographic**: API definitions are located in: `$ICP_ROOT/quickassist/include/lac`, where `$ICP_ROOT` is the directory where the Acceleration software is unpacked. See the Intel QuickAssist Technology Cryptographic API Reference Manual for details.
- **Data Compression**: API definitions are located in: `$ICP_ROOT/quickassist/include/dc`. See the Intel QuickAssist Technology Data Compression API Reference Manual for details.

7.1.1 Cryptographic and Data Compression API Descriptions

Full descriptions of the Intel® QAT APIs are contained in the *Intel QuickAssist Technology Cryptographic API Reference Manual* and the *Intel QuickAssist Technology Data Compression API Reference Manual*.

In addition to the Intel® QAT Data Plane APIs, there are a number of Data Plane Polling APIs that are described in the *Polling Functions* section.

7.1.1.1 Data Plane APIs Overview

The *Intel QuickAssist Technology Cryptographic API Reference Manual* and the *Intel QuickAssist Technology Data Compression API Reference Manual* contain information on the APIs that are specific to data plane applications.

The APIs are recommended for applications that are executing in a data plane environment where the cost of offload (that is, the cycles consumed by the driver sending requests to the hardware) needs to be minimized. To minimize the cost of offload, several constraints have been placed on the APIs. If these constraints are too restrictive for your application, the traditional APIs can be used instead (at a cost of additional IA cycles).

The definition of the Cryptographic Data Plane APIs are contained in: `$ICP_ROOT/quickassist/include/lac/cpa_cy_sym_dp.h`

The definition of the Data Compression Data Plane APIs are contained in: `$ICP_ROOT/quickassist/include/dc/cpa_dc_dp.h`

7.1.1.2 IA Cycle Count Reduction When Using Data Plane APIs

From an IA cycle count perspective, the Data Plane APIs are more performant than the traditional APIs. The majority of the cycle count reduction is realized by the reduction of supported functionality in the Data Plane APIs and the application of constraints on the calling application.

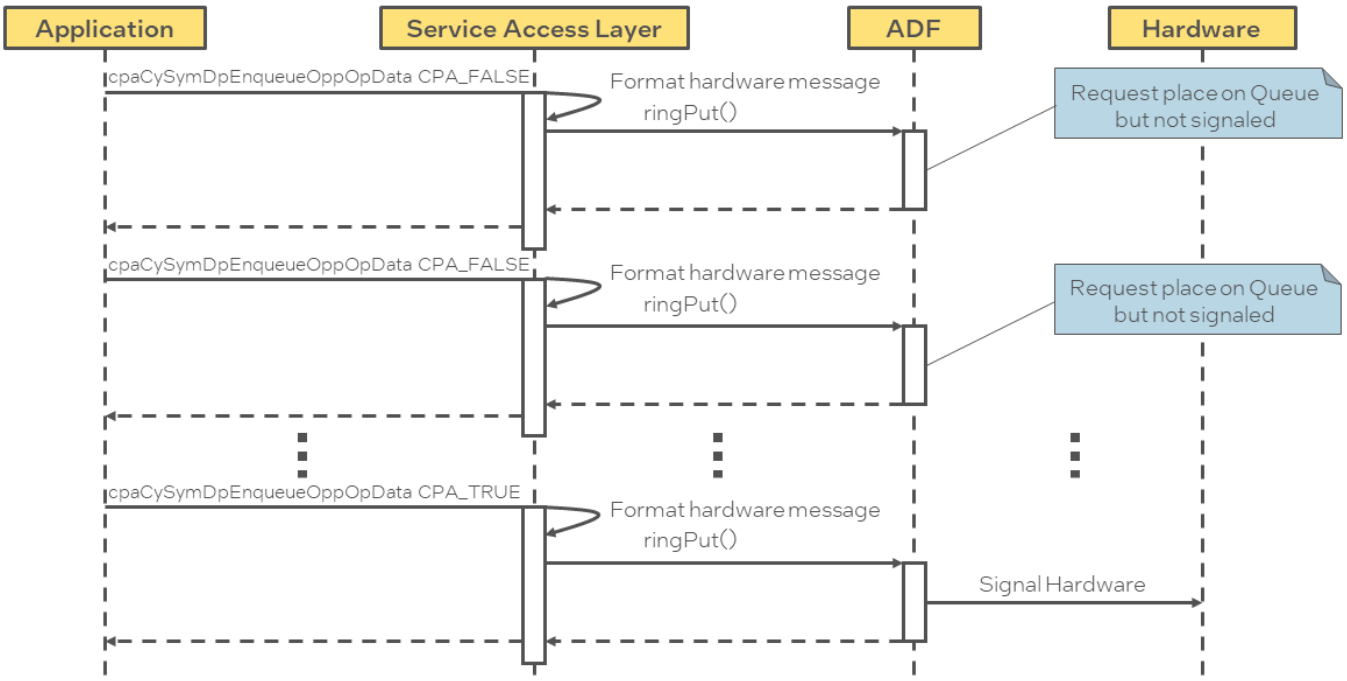
In addition, to further improve performance, the Data Plane APIs attempt to amortize the cost of an MMIO access when sending requests to, and receiving responses from, the hardware.

A typical usage is to call the `cpaCySymDpEnqueueOp()` or the `cpaDcDpEnqueueOp()` function multiple times with requests to process and the `performOpNow` flag set to `CPA_FALSE`. Once multiple requests have been enqueued, `cpaCySymDpEnqueueOp()` or `cpaDcDpEnqueueOp()` may be called with the `performOpNow` flag set to `CPA_TRUE`. This sends the requests to the Intel® QAT Endpoint for processing.

The Intel® QAT API returns a `CPA_STATUS_RETRY` when the ring becomes full.

The number of requests to place on the ring is application dependent and it is recommended that performance testing be conducted with tunable parameter values.

Two functions, `cpaCySymDpPerformOpNow()` and `cpaDCDpPerformOpNow()`, are also provided that allow queued requests to be sent to the hardware without the need for queuing an additional request. This is typically used in the scenario where a request has not been received for some time and the application would like the enqueued requests to be sent to the hardware for processing.



7.1.1.3 Usage Constraints on the Data Plane APIs

The following constraints apply to the use of the Data Plane APIs. If the application can handle these constraints, the Data Plane APIs can be used:

- Thread safety is not supported. Each software thread should have access to its own unique instance (`CpaInstanceHandle`) to avoid contention on the hardware rings.
- For performance, polling is supported, as opposed to interrupts (which are comparatively more expensive).
- *Polling functions* are provided to read responses from the hardware response queue and dispatch callback functions.
- Buffers and buffer lists are passed using physical addresses to avoid virtual-to- physical address translation costs.
- Alignment restrictions are placed on the operation data (that is, the `CpaCySymDpOpData` structure) passed to the Data Plane API. The operation data must be at least 8-byte aligned, contiguous, resident, DMA-accessible memory.
- Only asynchronous invocation is supported, that is, synchronous invocation is not supported.
- There is no support for cryptographic partial packets. If support for partial packets is required, the traditional Intel® QAT APIs should be used.
- Since thread safety is not supported, statistic counters on the Data Plane APIs are not atomic.
- The default instance (`CPA_INSTANCE_HANDLE_SINGLE`) is not supported by the Data Plane APIs. The specific handle should be obtained using the instance discovery func-

tions (`cpaCyGetNumInstances()`, `cpaCyGetInstances()`, `cpaDcGetNumInstances()`, `cpaDcGetInstances()`).

- The submitted requests are always placed on the high-priority ring.
- The data plane APIs are supported in both user space and polling mode in kernel space, but not supported in interrupt mode in kernel space.

7.1.2 Intel® QAT API Limitations

The following limitations apply when using the Intel® QAT APIs on the platforms described in this manual:

- For all services, the maximum size of a single perform request is 4 GB.
- For all services, data structures that contain data required by the Intel® QAT Endpoint should be on a 64-byte-aligned address to maximize performance. This alignment helps minimize latency when transferring data from DRAM to an Intel® QAT Endpoint integrated in the PCH device.
- For the key generation cryptographic API, the following limitations apply:

Table 42: Key Generation Cryptographic API Limitations

Secure Sockets Layer (SSL) key generation op-data:	Maximum secret length is 512 bytes
	Maximum **`userLabel`** length is 136 bytes
	Maximum **`generatedKeyLenInBytes`** is 248
Transport Layer Security (TLS) key generation op-data:	Secret length must be <128 bytes for TLS v1.0/1.1;
	Secret length must be <512 bytes for TLS v1.2
	Secret length must be <512 bytes for TLS v1.3
	`userLabel` length must be <256 bytes
	Maximum seed size is 64 bytes
	Maximum **`generatedKeyLenInBytes`** is 248 bytes
Mask Generation Function (MGF) op-data:	Maximum seed length is 255 bytes
	Maximum **`maskLenInBytes`** is 65528

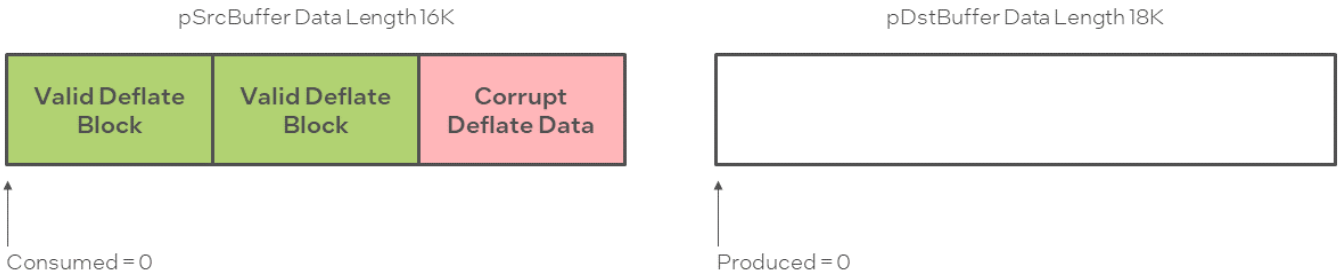
- For the cryptographic service, SNOW 3G and KASUMI* operations are not supported when **`CpaCySymPacketType`** is set to **`CPA_CY_SYM_PACKET_TYPE_PARTIAL`**. The error returned in this case is **`CPA_STATUS_INVALID_PARAM`**.
- For the cryptographic service, when using the asymmetric crypto APIs, the buffer size passed to the API should be rounded to the next power of 2, or the next 3- times a power of 2, for optimum performance.
- For the data compression service, the size of all stateful decompression requests have to be a multiple of two with the exception of the last request.
- For the data compression service, the **`CpaDcFileType`** field in the **`CpaDcSessionSetupData`** data structure is ignored (previously this was considered for semi-dynamic compression/decompression).

- For static compression, the maximum expansion during compression is ceiling *(9xTotal_Input_Byte/8)+7* bytes. If `CPA_DC_ASB_UNCOMP_STATIC_DYNAMIC_WITH_STORED_HDRS` or `CPA_DC_ASB_UNCOMP_STATIC_DYNAMIC_WITH_NO_HDRS` is selected, the maximum expansion during compression is the input buffer size plus up to ceiling *(Total_Input_Byte/65535)x5* bytes, depending on whether the stored headers are selected.

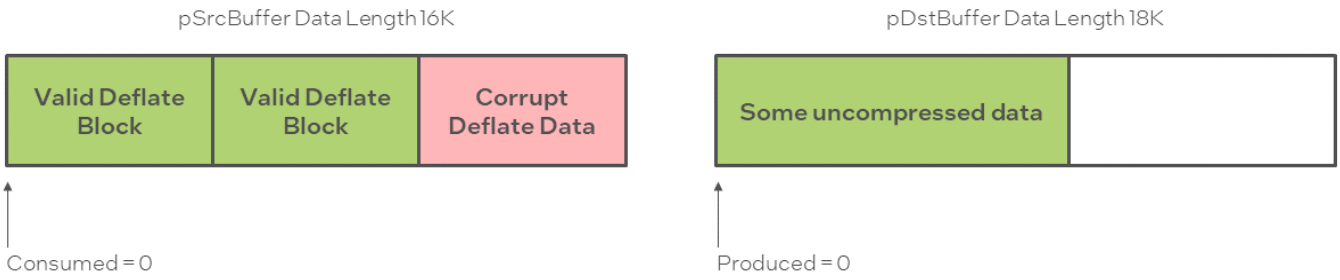
  Note: Due to the need for a skid pad and the way the checksum is calculated in the stored block case to prevent compression overflow, an output buffer size of ceiling (9*Total_Input_Byte/8) + 55 bytes needs to be supplied (even though the stored block output size might be less).

- The decompression service can report various error conditions, most of which arise from processing dynamic Huffman code trees that are ill-formed. These soft error conditions are reported at the Intel® QuickAssist Technology API using the `CpaDcReqStatus` enumeration. At the point of soft error, the hardware state will not be accurate to allow recovery. Therefore, in this case, the Intel® QuickAssist Technology software rolls back to the previous known good state and reports that no input has been processed and no output produced. This allows an application to correct the source of the error and resubmit the request.

  For example, if the following source and destination buffers were submitted to the Intel® QuickAssist Technology:



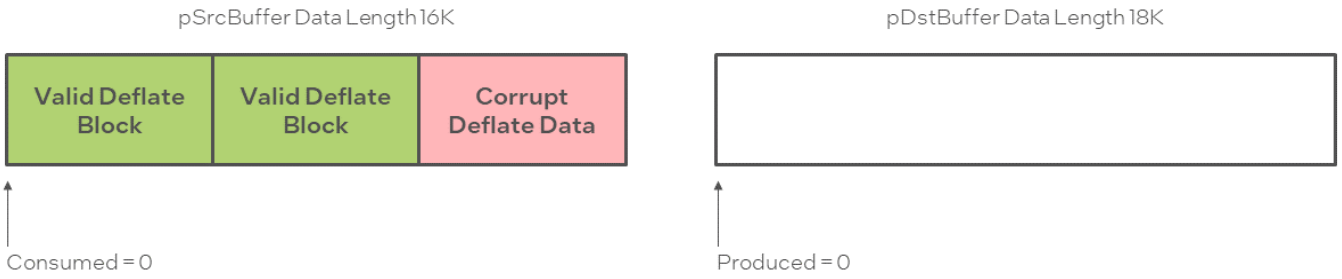
  The result would be:



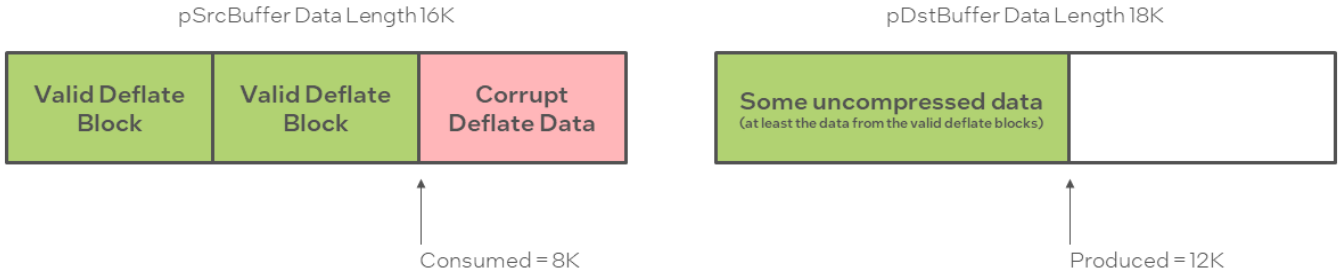
- Behavior when build flag `ICP_DC_RETURN_COUNTERS_ON_ERROR` is defined. In some specialized applications, when a decompression soft error occurs, the application has no way of correcting the source of the error and resubmitting the request. The session will need to be invalidated and terminated. In this case it is more useful to the application to output the uncompressed data

up to the point of soft error before terminating the session. There is a compile time build flag (`ICP_DC_RETURN_COUNTERS_ON_ERROR`) to select this mode of operation. This is the behavior of de-compression in case of soft error when this build flag is used.

If the following source and destination buffers were submitted to the Intel® QuickAssist Technology API:



The result would be:



Warning: It is important to note in this case:

- The consumed value returned in the `CpaDcRqResults` structure is not reliable.
- No further requests can be submitted on this session.

7.2 Additional APIs

Note: Not all *Additional APIs* are supported with all versions of the software package/hardware configuration.

There are a number of additional APIs that can serve for optimization and other uses outside of the Intel® QuickAssist Technology services.

7.2.1 Dynamic Instance Allocation Functions

These functions are intended for the dynamic allocation of instances in user space. The user can use these functions to allocate/free instances defined in the `[DYN]` section of the configuration file.

These functions are useful if the user needs to dynamically allocate/free cryptographic (CY) or Data Compression (DC) instances at runtime. This is in contrast to statically specifying the number of CY or DC instances at configuration time, where the number of instances cannot be changed unless the user modifies the `.conf` file and restarts the acceleration service.

The advantage of using these functions is that the number of CY/DC instances can be changed on-demand at runtime. The disadvantage is that runtime performance is impacted if the number of CY/DC instances is changed frequently.

If the user space application knows the number of instances to be used before starting, then the user can define `Number<Service>Instances` in the `[User Process]` section of the configuration file.

If the user space application can only know the number of instances at runtime, or wants to change the number at runtime, then the user can call the Dynamic Instance Allocation functions to allocate/free instances dynamically. The `Number<Service>Instances` in the `[DYN]` section of the .conf file(s) defines the maximum number of instances that can be allocated by user processes.

This can be useful when sharing instances among multiple applications at runtime. The maximum number of instances in a system is known in advance and it is possible to distribute them statically between applications using the configuration files. Once the driver is started, however, this cannot be changed. If, for example, there are 32 CY instances and we need to provision 16 processes, we can statically assign two CY instances per process. This can be a problem when a process needs more instances at any given time. With dynamic instance allocation, we can create a pool of instances that can be "shared" between the processes.

Continuing the example above with 32 CY instances and 16 processes, we can assign statically one CY instance to each process and create a pool of 16 `[DYN]` instances from the remainder. If at runtime one process needs more acceleration power, it can allocate some more instances from the pool, say, for example, eight, use them as appropriate and free them back to the pool when the work has been completed. Thereafter, other processes can use these instances as needed.

All dynamic instance allocation function definitions are located in: `$ICP_ROOT/quickassist/lookaside/access_layer/include/icp_sal_user.h`

Important: *Dynamic Instance Allocation Functions* are not currently supported with the QAT2.0 driver.

7.2.1.1 icp_sal_userCyGetAvailableNumDynInstances

Get the number of cryptographic instances that can be dynamically allocated using the `icp_sal_userCyInstancesAlloc` function.

Syntax

```
CpaStatus icp_sal_userCyGetAvailableNumDynInstances (Cpa32U *pNumCyInstances);
```

Parameters

`*pNumCyInstances`	A pointer to the number of cryptographic instances available for dynamic allocation.

Return Value

The `icp_sal_userCyGetAvailableNumDynInstances` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	A pointer to the number of cryptographic instances available for dynamic allocation.
`CPA_STATUS_FAIL`	Indicates a failure.

7.2.1.2 icp_sal_userDcGetAvailableNumDynInstances

Get the number of data compression instances that can be dynamically allocated using the `icp_sal_userDcInstancesAlloc` function.

Syntax

```
CpaStatus icp_sal_userDcGetAvailableNumDynInstances (Cpa32U *pNuDcInstances);
```

Parameters

`*pNumDcInstances`	A pointer to the number of data compression instances available for dynamic allocation.

Return Value

The `icp_sal_userDcGetAvailableNumDynInstances` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	A pointer to the number of data compression instances available for dynamic allocation.
`CPA_STATUS_FAIL`	Indicates a failure.

7.2.1.3 icp_sal_userCyInstancesAlloc

Allocate the specified number of Cryptographic (CY) instances from the amount specified in the `[DYN]` section of the configuration file. The `numCyInstances` parameter specifies the number of CY instances to allocate and must be less than or equal to the value of the `NumberCyInstances` parameter in the `[DYN]` section of the configuration file.

Syntax

```
CpaStatus icp_sal_userCyInstancesAlloc(Cpa32U numCyInstances, CpaInstanceHandle *pCyInstances);
```

Parameters

`numCyInstances`	The number of CY instances to allocate.
`*pCyInstances`	A pointer to the CY instances.

Return Value

The `icp_sal_userCyInstancesAlloc` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	Successfully allocated the specified number of CY instances.
`CPA_STATUS_FAIL`	Indicates a failure.

7.2.1.4 icp_sal_userDcInstancesAlloc

Allocate the specified number of Data Compression (DC) instances from the amount specified in the `[DYN]` section of the configuration file. The `numDcInstances` parameter specifies the number of dc instances to allocate and must be less than or equal to the value of the `NumberDcInstances` parameter in the `[DYN]` section of the configuration file.

Syntax

```
CpaStatus icp_sal_userDcInstancesAlloc(Cpa32U numDcInstances, CpaInstanceHandle *pDcInstances);
```

Parameters

`numDcInstances`	The number of DC instances to allocate.
`*pDcInstances`	A pointer to the DC instances.

Return Value

The `icp_sal_userDcInstancesAlloc` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	Successfully allocated the specified number of DC instances.
`CPA_STATUS_FAIL`	Indicates a failure.

7.2.1.5 icp_sal_userCyFreeInstances

Free the specified number of Cryptographic (CY) instances from the amount specified in the `[DYN]` section of the configuration file. The `numCyInstances` parameter specifies the number of CY instances to free.

Syntax

```
CpaStatus icp_sal_userCyFreeInstances(Cpa32U numCyInstances, CpaInstanceHandle *pCyInstances);
```

Parameters

`numCyInstances`	The number of CY instances to free.
`*pCyInstances`	A pointer to the CY instances.

Return Value

The `icp_sal_userCyFreeInstances` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	Successfully freed the specified number of CY instances.
`CPA_STATUS_FAIL`	Indicates a failure.

7.2.1.6 icp_sal_userDcFreeInstances

Free the specified number of Data Compression (DC) instances from the amount specified in the `[DYN]` section of the configuration file. The `numDcInstances` parameter specifies the number of DC instances to free.

Syntax

```
CpaStatus icp_sal_userDcFreeInstances(Cpa32U numDcInstances, CpaInstanceHandle *pDcInstances);
```

Parameters

`numDcInstances`	The number of DC instances to free.
`*pDcInstances`	A pointer to the DC instances to free.

Return Value

The `icp_sal_userDcFreeInstances` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	Successfully freed the specified number of DC instances.
`CPA_STATUS_FAIL`	Indicates a failure.

7.2.1.7 icp_sal_userCyGetAvailableNumDynInstancesByDevPkg

Get the number of cryptographic instances that can be dynamically allocated.

Syntax

```
CpaStatus icp_sal_userCyGetAvailableNumDynInstancesByDevPkg(Cpa32U *pNumCyInstances,
Cpa32U devPkgID);
```

Parameters

pNumCyInstances	A pointer to the number of cryptographic instances available for dynamic allocation.
devPkgID	The device ID of the device of interest (same as accelID in other APIs) If -1 then selects from all devices.

Return Value

The `icp_sal_userCyGetAvailableNumDynInstancesByDevPkg` function returns one of the following codes:

CPA_STATUS_SUCCESS	Successfully retrieved the number of cryptographic instances available for dynamic allocation.
CPA_STATUS_FAIL	Indicates a failure.

7.2.1.8 icp_sal_userDcGetAvailableNumDynInstancesByDevPkg

Get the number of data compression instances that can be dynamically allocated.

Syntax

```
CpaStatus icp_sal_userDcGetAvailableNumDynInstancesByDevPkg(Cpa32U *pNumDcInstances,
Cpa32U devPkgID);
```

Parameters

*pNumDcInstances	A pointer to the number of data compression instances available for dynamic allocation.
devPkgID	The device ID of the device of interest (same as accelID in other APIs) If -1 then selects from all devices.

Return Value

The `icp_sal_userDcGetAvailableNumDynInstancesByDevPkg` function returns one of the following codes:

CPA_STATUS_SUCCESS	Successfully freed the specified number of DC instances.
CPA_STATUS_FAIL	Indicates a failure.

7.2.1.9 icp_sal_userCyInstancesAllocByDevPkg

Allocate the specified number of Cryptographic (CY) instances from the amount specified in the `[DYN]` section of the configuration file. The `numCyInstances` parameter specifies the number of CY instances to allocate and must be less than or equal to the value of the `NumberCyInstances` parameter in the `[DYN]` section of the configuration file.

Syntax

```
CpaStatus icp_sal_userCyInstancesAllocByDevPkg(Cpa32U numCyInstances, CpaInstanceHandle
*pCyInstances, devPkgID);
```

Parameters

`numCyInstances`	The number of CY instances to allocate.
`*pCyInstances`	A pointer to the CY instances.
`devPkgID`	The device ID of the device of interest (same as accelID in other APIs) If -1 then selects from all devices.

Return Value

The `icp_sal_userCyInstancesAllocByDevPkg` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	Successfully allocated the specified number of CY instances.
`CPA_STATUS_FAIL`	Indicates a failure.

7.2.1.10 icp_sal_userDcInstancesAllocByDevPkg

Allocate the specified number of Data Compression (DC) instances from the amount specified in the `[DYN]` section of the configuration file. The `numDcInstances` parameter specifies the number of DC instances to allocate and must be less than or equal to the value of the `NumberDcInstances` parameter in the `[DYN]` section of the configuration file.

Syntax

```
CpaStatus icp_sal_userDcInstancesAllocByDevPkg(Cpa32U numDcInstances, CpaInstanceHandle
*pDcInstances, Cpa32U devPkgID;
```

Parameters

`numDcInstances`	The number of DC instances to allocate.
`*pDcInstances`	A pointer to the DC instances.
`devPkgID`	The device ID of the device of interest (same as accelID in other APIs) If -1 then selects from all devices.

Return Value

The `icp_sal_userDcInstancesAllocByDevPkg` function returns one of the following codes:

CPA_STATUS_SUCCESS	Successfully allocated the specified number of DC instances.
CPA_STATUS_FAIL	Indicates a failure.

7.2.1.11 icp_sal_userCyGetAvailableNumDynInstancesByPkgAccel

Get the number of cryptographic instances that can be dynamically allocated.

Syntax

```
CpaStatus icp_sal_userCyGetAvailableNumDynInstancesByPkgAccel(Cpa32U *pNumCyInstances,
Cpa32U devPkgID, Cpa32U accelerator_number);
```

Parameters

pNumCyInstances	A pointer to the number of cryptographic instances available for dynamic allocation.
devPkgID	The device ID of the device of interest (same as accelID in other APIs) If -1 then selects from all devices.
accelerator_number	Accelerator Engine to use. As 0 is the only valid value on C62x device, this API is same as as icp_sal_userCyInstancesAllocByDevPkg

Return Value

The icp_sal_userCyGetAvailableNumDynInstancesByPkgAccel function returns one of the following codes:

CPA_STATUS_SUCCESS	Successfully retrieved the number of cryptographic instances available for dynamic allocation.
CPA_STATUS_FAIL	Indicates a failure.

7.2.1.12 icp_sal_userCyInstancesAllocByPkgAccel

Allocates the specified number of Cryptographic (CY) instances from the amount specified in the [DYN] section of the configuration file. The numCyInstances parameter specifies the number of CY instances to allocate and must be less than or equal to the value of the NumberCyInstances parameter returned by a call to the icp_sal_userCyInstancesAllocByPkgAccel function.

Syntax

```
CpaStatus icp_sal_userCyInstancesAllocByPkgAccel(Cpa32U numCyInstances, CpaInstanceHan-
dle *pCyInstances, Cpa32U devPkgID, Cpa32U accelerator_number);
```

Parameters

numCyInstances	The number of CY instances to allocate.
*pCyInstances	A pointer to the CY instances.
devPkgID	The device ID of the device of interest (same as accelID in other APIs) If -1 then selects from all devices
accelerator_number	Accelerator Engine to use. As 0 is the only valid value on C62x device, this API is same as as icp_sal_userCyInstancesAllocByDevPkg

Return Value

The icp_sal_userCyInstancesAllocByPkgAccel function returns one of the following codes:

CPA_STATUS_SUCCESS	Successfully allocated the specified number of CY instances.
CPA_STATUS_FAIL	Indicates a failure.

7.2.2 IOMMU Remapping Functions

These functions are intended for IOMMU remapping operations.

All IOMMU remapping function definitions are located in: $ICP_ROOT/quickassist/lookaside/access_layer/include/icp_sal_iommu.h

7.2.2.1 icp_sal_iommu_get_remap_size

Returns the page_size rounded for IOMMU remapping.

Syntax

```
size_t icp_sal_iommu_get_remap_size(size_t size);
```

Parameters

size	The minimum required page size.

Return Value

The icp_sal_iommu_get_remap_size function returns the page_size rounded for IOMMU remapping.

7.2.2.2 icp_sal_iommu_map

Adds an entry to the IOMMU remapping table.

Syntax

```
CpaStatus icp_sal_iommu_map(Cpa64U phaddr, Cpa64U iova, size_t size);
```

Parameters

phaddr	Host physical address.
iova	Guest physical address.
size	Size of the remapped region.

Return Value

The `icp_sal_iommu_map` function returns one of the following codes:

CPA_STATUS_SUCCESS	Successful operation.
CPA_STATUS_FAIL	Indicates a failure.

7.2.2.3 icp_sal_iommu_unmap

Removes an entry from the IOMMU remapping table.

Syntax

```
CpaStatus icp_sal_iommu_unmap(Cpa64U iova, size_t size);
```

Parameters

iova	Guest physical address.
size	Size of the remapped region.

Return Value

The `icp_sal_iommu_unmap` function returns one of the following codes:

CPA_STATUS_SUCCESS	Successful operation.
CPA_STATUS_FAIL	Indicates a failure.

7.2.2.4 IOMMU Remapping Function Usage

These functions are required when the user wants to access an acceleration service from the Physical Function (PF) when SR-IOV is enabled in the driver. In this case, all I/O transactions from the device go through DMA remapping hardware.

This hardware checks:

1. If the transaction is legitimate
2. What physical address the given I/O address needs to be translated to.

If the I/O address is not in the transaction table, it fails with a DMA Read error shown as follows:

- **DRHD**: Handling fault status reg 3.
- **DMAR**:[DMA Read] Request device [02:01.2] fault addr <ADDR> DMAR:[fault reason 06] PTE Read access is not set.

To make this work, the user must add a 1:1 mapping as follows:

1. **Get** the size required for a buffer.

   ```
   int size = icp_sal_iommu_get_remap_size(size_of_data);
   ```

2. **Allocate** a buffer.

   ```
   char *buff = malloc(size);
   ```

3. **Get** a physical pointer to the buffer.

   ```
   buff_phys_addr = virt_to_phys(buff);
   ```

4. **Add** a 1:1 mapping to the IOMMU tables.

   ```
   icp_sal_iommu_map(buff_phys_addr, buff_phys_addr, size;
   ```

5. **Use** the buffer to send data to the Intel® QAT Endpoint.
6. Before freeing the buffer, **remove** the IOMMU table entry.

   ```
   icp_sal_iommu_unmap(buff_phys_addr, size);
   ```

7. **Free** the buffer.

   ```
   free(buff);
   ```

The IOMMU remapping functions can be used in all contexts that the Intel® QAT APIs can be used, that is, kernel and user space in a Physical Function (PF) Domain 0, as well as kernel and user space in a Virtual Machine (VM).

In the case of VM, the APIs will do nothing. In the PF Domain 0 case, the APIs will update the hardware IOMMU tables.

7.2.3 Polling Functions

These functions are intended for retrieving response messages that are on the rings and dispatching the associated callbacks.

All polling function definitions are located in: `$ICP_ROOT/quickassist/lookaside/access_layer/include/icp_sal_poll.h`

7.2.3.1 icp_sal_pollBank

Poll all rings on the given Intel® QAT Endpoint on a given bank number to determine if any of the rings contain response messages from the Intel® QAT Endpoint. The `response_quota` input parameter is per ring.

Syntax

```
CpaStatus icp_sal_pollBank(Cpa32U accelId, Cpa32U bank_number, Cpa32U response_quota);
```

Parameters

`accelId`	The device number associated with the Intel® QAT Endpoint. valid range is 0 to number of Intel® QAT Endpoints in the system.
`bank_number`	The number of the memory bank on the Intel® QAT Endpoint that will be polled for response messages. The valid range is 0 to 31.
`response_quota`	The maximum number of responses to take from the ring in one call.

Return Value

The `icp_sal_pollBank` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	Successfully polled a ring with data.
`CPA_STATUS_RETRY`	There is no data on any ring on any bank or the banks are already being polled.
`CPA_STATUS_FAIL`	Indicates a failure.

7.2.3.2 icp_sal_pollAllBanks

Poll all banks on the given Intel® QAT Endpoint to determine if any of the rings contain response messages from the Intel® QAT Endpoint. The `response_quota` input parameter is per ring.

Syntax

```
CpaStatus icp_sal_pollAllBanks(Cpa32U accelId, Cpa32U response_quota);
```

Parameters

`accelId`	The device number associated with the Intel® QAT Endpoint. valid range is 0 to number of Intel® QAT Endpoints in the system.
`response_quota`	The maximum number of responses to take from the ring in one call.

Return Value

The `icp_sal_pollAllBanks` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	Successfully polled a ring with data.
`CPA_STATUS_RETRY`	There is no data on any ring on any bank or the banks are already being polled.
`CPA_STATUS_FAIL`	Indicates a failure.

7.2.3.3 icp_sal_CyPollInstance

Poll the Cryptographic (CY) logical instance associated with the `instanceHandle` to retrieve requests that are on response rings associated with that instance and dispatch the associated callbacks. The `response_quota` input parameter is the maximum number of responses to process in one call.

Note: The `icp_sal_CyPollInstance()` function is used in conjunction with the `CyXIsPolled` parameter in the acceleration configuration file.

Syntax

```
CpaStatus icp_sal_CyPollInstance(CpaInstanceHandle instanceHandle, Cpa32U response_quota);
```

Parameters

`instanceHandle`	The logical instance to poll for responses on the response ring.
`response_quota`	The maximum number of responses to take from the ring in one call. When set to 0, all responses are retrieved.

Return Value

The `icp_sal_CyPollInstance` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	The function was successful.
`CPA_STATUS_RETRY`	There are no responses on the rings associated with the specified logical instance.
`CPA_STATUS_FAIL`	Indicates a failure.

Note: A ring is only polled if it contains data.

7.2.3.4 icp_sal_DcPollInstance

Poll the Data Compression (DC) logical instance associated with the `instanceHandle` to retrieve requests that are on response rings associated with that instance and dispatch the associated callbacks. The `response_quota` input parameter is the maximum number of responses to process in one call.

Note: The `icp_sal_DcPollInstance()` function is used in conjunction with the `DcXIsPolled` parameter in the acceleration configuration file.

Syntax

```
CpaStatus icp_sal_DcPollInstance(CpaInstanceHandle instanceHandle, Cpa32U response_quota);
```

Parameters

`instanceHandle`	The logical instance to poll for responses on the response ring.
`response_quota`	The maximum number of responses to take from the ring in one call. When set to 0, all responses are retrieved.

Return Value

The `icp_sal_DcPollInstance` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	The function was successful.
`CPA_STATUS_RETRY`	There are no responses on the rings associated with the specified logical instance.
`CPA_STATUS_FAIL`	Indicates a failure.

Note: A ring is only polled if it contains data.

7.2.3.5 icp_sal_CyPollDpInstance

Poll a particular Cryptographic (CY) data path logical instance associated with the `instanceHandle` to retrieve requests that are on the high-priority symmetric ring associated with that instance and dispatch the associated callbacks. The `response_quota` input parameter is the maximum number of responses to process in one call.

Note: This function is a Data Plane API function and consequently the restrictions in *Usage Constraints on the Data Plane APIs* apply.

Syntax

```
CpaStatus icp_sal_CyPollDpInstance(CpaInstanceHandle instanceHandle, Cpa32U
response_quota);
```

Parameters

`instanceHandle`	The logical instance to poll for responses on the response ring.
`response_quota`	The maximum number of responses to take from the ring in one call. When set to 0, all responses are retrieved.

Return Value

The `icp_sal_CyPollDpInstance` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	The function was successful.
`CPA_STATUS_RETRY`	There are no responses on the rings associated with the specified logical instance.
`CPA_STATUS_FAIL`	Indicates a failure.

7.2.3.6 icp_sal_DcPollDpInstance

Poll a particular Data Compression (DC) data path logical instance associated with the `instanceHandle` to retrieve requests that are on the response ring associated with that instance. The `response_quota` input parameter is the maximum number of responses to process in one call.

Note: This function is a Data Plane API function and consequently the restrictions in *Usage Constraints on the Data Plane APIs* apply.

Syntax

```
CpaStatus icp_sal_DcPollDpInstance(CpaInstanceHandle instanceHandle, Cpa32U
response_quota);
```

Parameters

`instanceHandle`	The logical instance to poll for responses on the response ring.
`response_quota`	The maximum number of responses to take from the ring in one call. When set to 0, all responses are retrieved.

Return Value

The `icp_sal_DcPollDpInstance` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	The function was successful.
`CPA_STATUS_RETRY`	There are no responses on the rings associated with the specified logical instance.
`CPA_STATUS_FAIL`	Indicates a failure.

7.2.4 User Space Access Configuration Functions

Functions that allow the configuration of user space access to the Intel® QAT services from processes running in user space.

All user space access configuration function definitions are located in `$ICP_ROOT/quickassist/lookaside/access_layer/include/icp_sal_user.h`

7.2.4.1 icp_sal_userStart

Initializes user space access to an Intel® QAT Endpoint and starts the `ProcessName` section in the given section of the configuration file. This function needs to be called prior to any call to Intel® QAT API function from the user space process. This function is typically called only once in a user space process.

Note: The `icp_sal_userStartMultiProcess()` function is still supported, but the parameter `limitDevAccess` is ignored because its value is set once in the configuration file and is not allowed to be specified again in the function.

The configuration format allows the user to easily create a configuration for many user spaces processes. The driver internally generates unique process names and a valid configuration for each process based on the section name (`pSectionName`) and mode (`limitDevAccess`) provided.

For example, on a system with *M* number of devices, if all *M* configuration files contain:

```
[IPSec]
NumProcesses = N
LimitDevAccess = 0
```

Then *N* internal sections are generated (each with instances on all devices) and *N* processes can be started at any given time. Each process can call `icp_sal_userStart("IPSec")` and the driver determines the unique name to use for each process.

Similarly, on an *M* device system, if all *M* configuration files contain:

```
[SSL]
NumProcesses = N
LimitDevAccess = 1
```

Then *MxN* internal sections are generated (each with instances on one device only) and *MxN* processes can be started at any given time. Each process can call `icp_sal_userStart("SSL")` and the driver determines the unique name to use for each process.

Refer to *Configuring Multiple Processes on a System with Multiple Intel® QAT Endpoints* for detailed example.

Syntax

```
CpaStatus icp_sal_userStart(const char *pSectionName);
```

Parameters

pSectionName	The section name described in the configuration file.

Return Value

The `icp_sal_userStart` function returns one of the following codes:

CPA_STATUS_SUCCESS	Successfully started user space access to the Intel® QAT Endpoint as defined in the configuration file.
CPA_STATUS_FAIL	Operation failed.

7.2.4.2 icp_sal_userStop

Closes user space access to the Intel® QAT Endpoint; stops the services that were running and frees the allocated resources. After a successful call to this function, user space access to the Intel® QAT Endpoint from a calling process is not possible. This function should be called once when the process is finished using the Intel® QAT Endpoint and does not intend to use it again.

Syntax

```
CpaStatus icp_sal_userStop( void);
```

Parameters

None

Return Value

The `icp_sal_userStop` function returns one of the following codes:

CPA_STATUS_SUCCESS	Successfully stopped user space access to the Intel® QAT Endpoint.
CPA_STATUS_FAIL	Operation failed.

7.2.5 Version Information Function

A function that allows the retrieval of version information related to the software and hardware being used.

The version information function definition is located in: `$ICP_ROOT/quickassist/lookaside/access_layer/include/icp_sal_versions.h`.

7.2.5.1 icp_sal_getDevVersionInfo

Retrieves the hardware revision and information on the version of the software components being run on a given device.

Note: The `icp_sal_userStart` function must be called before calling this function. If not, calling this function returns `CPA_STATUS_INVALID_PARAM` indicating an error. The `icp_sal_userStart` function is responsible for setting up the ADF user space component, which is required for this function to operate successfully.

Syntax

```
CpaStatus icp_sal_getDevVersionInfo(Cpa32U devId, icp_sal_dev_version_info_t *pVerInfo);
```

Parameters

`devId`	The ID (number) of the device for which version information is to be retrieved.
`*pVerInfo`	A pointer to a structure that holds the version information.

Return Value

The `icp_sal_getDevVersionInfo` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	Operation finished successfully; version information retrieved.
`CPA_STATUS_INVALID_PARAM`	Invalid parameter passed to the function.
`CPA_STATUS_RESOURCE`	System resource problem.
`CPA_STATUS_FAIL`	Operation failed.

7.2.6 Reset Device Function

This API can only be called in user-space.

The device can be reset using this API call. This will schedule a reset of the device. The device can also be reset using the `adf_ctl` utility, e.g., by calling `adf_ctl qat_dev0 reset`.

7.2.6.1 icp_sal_reset_device

Resets the device.

Syntax

```
CpaStatus icp_sal_reset_device(Cpa32U accelid);
```

Parameters

accelid	The device number.

Return Value

The `icp_sal_reset_device` function returns one of the following codes:

CPA_STATUS_SUCCESS	Successful operation.
CPA_STATUS_FAIL	Indicates a failure.

7.2.7 Thread-Less APIs

These APIs can be used when the QAT acceleration driver has been configured not to spawn threads.

These APIs can be used in the user space application.

7.2.7.1 icp_sal_poll_device_events

This reads any pending device events from `icp_dev%d_csr` and forwards to interested subsystems.

Syntax

```
CpaStatus icp_sal_poll_device_events(void);
```

Parameters

None

Return Value

The `icp_sal_poll_device_events` function returns one of the following codes:

CPA_STATUS_SUCCESS	Successful operation.
CPA_STATUS_FAIL	Indicates a failure.

7.2.7.2 icp_sal_find_new_devices

This tries to connect to any available devices that the kernel driver has brought up and initialized for use in user space process.

Syntax

```
CpaStatus icp_sal_find_new_devices(void);
```

Parameters

None

Return Value

The `icp_sal_find_new_devices` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	Successful operation.
`CPA_STATUS_FAIL`	Indicates a failure.

7.2.8 Compress and Verify (CnV) Related APIs

APIs documented in this section are used for Compress and Verify. These APIs can be used in the user space application.

7.2.8.1 icp_sal_get_dc_error

This API allows the application to return the number of errors that occurred a particular number of times during the lifetime of a process.

Syntax

```
Cpa64U icp_sal_get_dc_error(Cpa8S dcError);
```

Parameters

`dcError`	Compression Error code exposed by `CpaDcReqStatus` enum in `cpa_dc.h`

Return Value

The `icp_sal_get_dc_error` function returns a 64 bit unsigned integer representing how many times the error type specified by `Cpa8S dcError` occurred in the current process.

7.2.8.2 icp_sal_dc_simulate_error

This API injects a simulated compression error for a defined number of compression or decompression requests. The simulated compression errors can only be applied to the traditional APIs. It must be called prior the APIs that perform the request. In the case of a simulated Compress and Verify error for a single request, the application would call `icp_sal_dc_simulate_error()` API as such: `icp_sal_dc_simulate_error(1, CPA_DC_VERIFY_ERROR);`

Syntax

```
CpaStatus icp_sal_dc_simulate_error(Cpa8U numErrors, Cpa8S dcError);
```

Parameters

`numErrors`	Number of simulated compression or decompression errors desired.
`dcError`	Desired error code to be returned by the compression or decompression API.

Return Value

The `icp_sal_dc_simulate_error` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	Successful operation.
`CPA_STATUS_FAIL`	Indicates that an invalid error type was assigned to dcError parameter.

7.2.9 Heartbeat APIs

These APIs check firmware/hardware status for a given device and are used as part of the Heartbeat functionality.

7.2.9.1 icp_sal_check_device

This function checks the status of the firmware/hardware for a given device and is used as part of the Heartbeat functionality.

Syntax

```
CpaStatus icp_sal_check_device(Cpa32U accelID);
```

Parameters

`accelID`	The device ID.

Return Value

The `icp_sal_check_device` function returns one of the following codes:

CPA_STATUS_SUCCESS	Successful operation.
CPA_STATUS_FAIL	Indicates a failure.

7.2.9.2 icp_sal_check_all_devices

This function checks the status of the firmware/hardware for all devices and is used as part of the Heartbeat functionality.

Syntax

```
CpaStatus icp_sal_check_all_devices(void);
```

Parameters

None

Return Value

The `icp_sal_check_all_devices` function returns one of the following codes:

CPA_STATUS_SUCCESS	Successful operation.
CPA_STATUS_FAIL	Indicates a failure.

7.2.9.3 icp_sal_heartbeat_simulate_failure

This function simulates heartbeat failure for a specific device.

Syntax

```
CpaStatus icp_sal_heartbeat_simulate_failure(Cpa32U accelID);
```

Parameters

accelID	The device ID.

Return Value

The `icp_sal_heartbeat_simulate_failure` function returns one of the following codes:

CPA_STATUS_SUCCESS	Successful operation.
CPA_STATUS_FAIL	Indicates a failure.

7.2.10 Device Polling APIs

APIs documented in this section are used for polling devices.

7.2.10.1 icp_sal_poll_device_events

This function polls for device reset events.

Syntax

```
CpaStatus icp_sal_poll_device_events(void);
```

Parameters

None

Return Value

The `icp_sal_poll_device_events` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	Successful operation.
`CPA_STATUS_FAIL`	Indicates a failure.

Note: The events are sent to each instance that has registered a callback function. The callbacks are registered using `cpaCyInstanceSetNotificationCb` and `cpaDcInstanceSetNotificationCb`.

7.2.10.2 cpaCyInstanceSetNotificationCb

Cryptographic instances use this function to register for device event notifications.

Syntax

```
CpaStatus cpaCyInstanceSetNotificationCb(const CpaInstanceHandle instanceHandle, const
CpaCyInstanceNotificationCbFunc pinstanceNotificationCb, void *pCallbackTag);
```

Parameters

`instanceHandle`	Instance handle.
`pinstanceNotificationCb`	Instance notification callback function pointer.
`*pCallbackTag`	Opaque value provided by user.

Return Value

The `cpaCyInstanceSetNotificationCb` function returns one of the following codes:

CPA_STATUS_SUCCESS	The function was successful.
CPA_STATUS_FAIL	Indicates a failure.
CPA_STATUS_INVALID_PARAM	Invalid parameter passed in.
CPA_STATUS_UNSUPPORTED	Function is not supported.

The signature for the callback function is:

```
typedef void (*CpaCyInstanceNotificationCbFunc)(
      const CpaInstanceHandle instanceHandle,
      void * pCallbackTag,
      const CpaInstanceEvent instanceEvent);
```

Parameter:

```
typedef enum _CpaInstanceEvent
{
   CPA_INSTANCE_EVENT_RESTARTING = 0,
   CPA_INSTANCE_EVENT_RESTARTED,
   CPA_INSTANCE_EVENT_FATAL_ERROR
} CpaInstanceEvent;
```

7.2.10.3 cpaDcInstanceSetNotificationCb

Data compression instances use this function to register for device event notifications.

Syntax

```
CpaStatus cpaDcInstanceSetNotificationCb(const CpaInstanceHandle instanceHandle, const
CpaDcInstanceNotificationCbFunc pinstanceNotificationCb, void *pCallbackTag);
```

Parameters

instanceHandle	Instance handle.
pinstanceNotificationCb	Instance notification callback function pointer.
*pCallbackTag	Opaque value provided by user.

Return Value

The `cpaDcInstanceSetNotificationCb` function returns one of the following codes:

CPA_STATUS_SUCCESS	The function was successful.
CPA_STATUS_FAIL	Indicates a failure.
CPA_STATUS_INVALID_PARAM	Invalid parameter passed in.
CPA_STATUS_UNSUPPORTED	Function is not supported.

The signature for the callback function is:

```
typedef void (*CpaDcInstanceNotificationCbFunc)(
      const CpaInstanceHandle instanceHandle,
      void * pCallbackTag,
      const CpaInstanceEvent instanceEvent);
```

Parameter:

```
typedef enum _CpaInstanceEvent
{
   CPA_INSTANCE_EVENT_RESTARTING = 0,
   CPA_INSTANCE_EVENT_RESTARTED,
   CPA_INSTANCE_EVENT_FATAL_ERROR
} CpaInstanceEvent;
```

7.2.11 Congestion Management APIs

Congestion Management or Back-pressure mechanism APIs are intended to handle the cases when the device is busy. These APIs ensures there is enough space on the ring before submitting a request.

Applications can query the appropriate ring on each instance and select any instance with enough space without creating any `OpData` structures.

All these API definitions are located in: `$ICP_ROOT/quickassist/lookaside/access_layer/include/icp_sal_congestion_mgmt.h`.

Important: *Congestion Management APIs* are not currently supported with the QAT2.0 driver.

7.2.11.1 icp_sal_SymGetInflightRequests

This function is used to fetch in-flight and max in-flight request counts for the given symmetric instance handle.

Syntax

```
CpaStatus icp_sal_SymGetInflightRequests(CpaInstanceHandle instanceHandle, Cpa32U *maxInflightRequests, Cpa32U *numInflightRequests);
```

Parameters

`instanceHandle`	Symmetric instance handle.
`*maxInflightRequests`	A pointer to the max in-flight request count.
`*numInflightRequests`	A pointer to the current in-flight request count.

Return Value

The `icp_sal_SymGetInflightRequests` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	Successfully retrieved the request counts.
`CPA_STATUS_INVALID_PARAM`	Invalid parameter passed to the function.
`CPA_STATUS_FAIL`	Indicates a failure.

7.2.11.2 icp_sal_AsymGetInflightRequests

This function is used to fetch in-flight and max in-flight request counts for the given asymmetric instance handle.

Syntax

```
CpaStatus icp_sal_AsymGetInflightRequests(CpaInstanceHandle instanceHandle, Cpa32U *maxInflightRequests, Cpa32U *numInflightRequests);
```

Parameters

`instanceHandle`	Asymmetric instance handle.
`*maxInflightRequests`	A pointer to the max in-flight request count.
`*numInflightRequests`	A pointer to the current in-flight request count.

Return Value

The `icp_sal_AsymGetInflightRequests` function returns one of the following codes:

`CPA_STATUS_SUCCESS`	Successfully retrieved the request counts.
`CPA_STATUS_INVALID_PARAM`	Invalid parameter passed to the function.
`CPA_STATUS_FAIL`	Indicates a failure.

7.2.11.3 icp_sal_dp_SymGetInflightRequests

This data plane function is used to fetch in-flight and max in-flight request counts for the given symmetric instance handle.

Syntax

```
CpaStatus icp_sal_dp_SymGetInflightRequests(CpaInstanceHandle instanceHandle, Cpa32U *maxInflightRequests, Cpa32U *numInflightRequests);
```

Parameters

instanceHandle	Symmetric instance handle.
*maxInflightRequests	A pointer to the max in-flight request count.
*numInflightRequests	A pointer to the current in-flight request count.

Return Value

The icp_sal_dp_SymGetInflightRequests function returns one of the following codes:

CPA_STATUS_SUCCESS	Successfully retrieved the request counts.
CPA_STATUS_INVALID_PARAM	Invalid parameter passed to the function.
CPA_STATUS_FAIL	Indicates a failure.

7.2.12 Service Specific Polling APIs

These service specific polling APIs are intended for retrieving response messages that are on the specific ring and dispatching the associated callback.

All these API definitions are located in: $ICP_ROOT/quickassist/lookaside/access_layer/include/icp_sal_poll.h

7.2.12.1 icp_sal_CyPollSymRing

Poll the symmetric logical instance associated with the instanceHandle to retrieve requests that are on the response rings associated with that instance and dispatch the associated callbacks. The response_quota input parameter is the maximum number of responses to process in one call.

Syntax

```
CpaStatus icp_sal_CyPollSymRing(CpaInstanceHandle instanceHandle, Cpa32U response_quota);
```

Parameters

instanceHandle	Instance handle to poll for responses on the response ring.
response_quota	The maximum number of messages that will be read in one polling. Setting the response quota to zero means that all messages on the ring will be read.

Return Value

The icp_sal_CyPollSymRing function returns one of the following codes:

CPA_STATUS_SUCCESS	Successfully polled a ring with data.
CPA_STATUS_INVALID_PARAM	Invalid parameter passed to the function.
CPA_STATUS_RETRY	There are no responses on the rings associated with the instance.
CPA_STATUS_FAIL	Indicates a failure.
CPA_STATUS_RESTARTING	Device restarting. Resubmit the request.

7.2.12.2 icp_sal_CyPollAsymRing

Poll the asymmetric logical instance associated with the `instanceHandle` to retrieve requests that are on the response rings associated with that instance and dispatch the associated callbacks. The `response_quota` input parameter is the maximum number of responses to process in one call.

Syntax

```
CpaStatus icp_sal_CyPollAsymRing(CpaInstanceHandle instanceHandle, Cpa32U response_quota);
```

Parameters

instanceHandle	Instance handle to poll for responses on the response ring.
response_quota	The maximum number of messages that will be read in one polling. Setting the response quota to zero means that all messages on the ring will be read.

Return Value

The `icp_sal_CyPollAsymRing` function returns one of the following codes:

CPA_STATUS_SUCCESS	Successfully polled a ring with data.
CPA_STATUS_INVALID_PARAM	Invalid parameter passed to the function.
CPA_STATUS_RETRY	There are no responses on the rings associated with the instance.
CPA_STATUS_FAIL	Indicates a failure.
CPA_STATUS_RESTARTING	Device restarting. Resubmit the request.

7.2.13 Check Device Availability APIs

7.2.13.1 icp_sal_userIsQatAvailable

This API allows an application to establish if there is any active QAT device present on system, without calling internal libadf APIs or without a dependency on `icp_sal_userStart()`

Syntax

```
CpaBoolean icp_sal_userIsQatAvailable(void);
```

Parameters

None

Return Value

The `icp_sal_userIsQatAvailable` function returns one of the following codes:

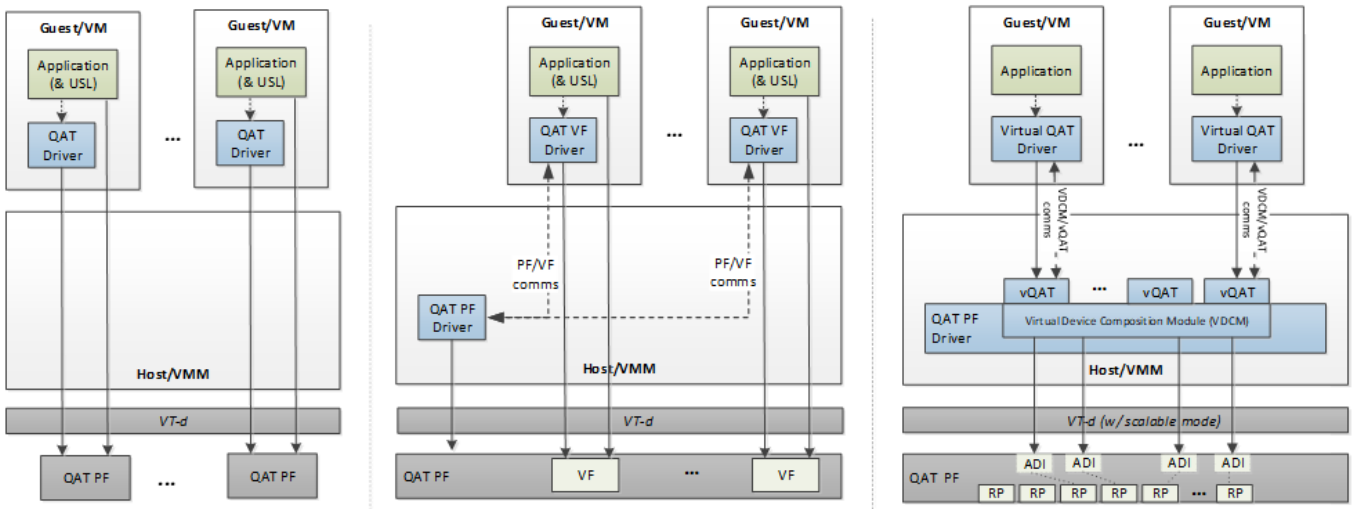
CPA_TRUE	Indicates that there is at least one active device.
CPA_FALSE	Indicates that there are no active devices.

8 Virtualization

8.1 Virtualization Deployment Model for Intel® QAT 2.0

Three different methods of virtualization are supported as shown in the below image.

Note: Single Root IOV (SR-IOV) and Scalable IOV (S-IOV) virtualization methods cannot be used simultaneously on the same Physical Function (PF).



8.2 Physical Device Direct Assignment

The hardware exposes one Physical Function (PF) per QAT Endpoint to the host. Number of QAT Endpoints per platform is included in the *Dimensions*.

One or more PFs may be passed to a single virtual machine.

There is no sharing of the PF.

Note: Hot plugging of Physical Functions (PFs) is not supported. To ensure the device functions correctly after being added, please restart the virtual machine.

8.3 Single Root IOV (SR-IOV)

When SR-IOV is enabled, the hardware exposes one Physical Function (PF) and **n** Virtual Functions(VFs) per QAT Endpoint to the host, where **n** is defined in *Dimensions*. Number of QAT Endpoints per platform is also included in the *Dimensions*.

One or more VFs can be passed through to different guests/VMs

For details on enabling SR-IOV refer to the Virtualization Deployment Guide.

8.4 Scalable IOV (S-IOV)

Scalable I/O Virtualization enables flexible composition of Virtual Functions by software from native hardware interfaces. Rather than implementing a complete SR-IOV virtual function (VF) interface, an S-IOV device exposes light-weight Assignable Device Interfaces (ADIs) that are optimized for fast-path (data-path) operations from the guest.

S-IOV uses *PASID* rather than BDF to identify unique address spaces which allow greater scalability. Number of supported ADIs is defined in *Dimensions*.

The public specification is available at Introducing Intel® Scalable I/O Virtualization.

Note: S-IOV is disabled in Linux Kernel after v5.16. Effort is underway to reenable in future kernel version.

For details on enabling S-IOV refer to the Virtualization Deployment Guide.

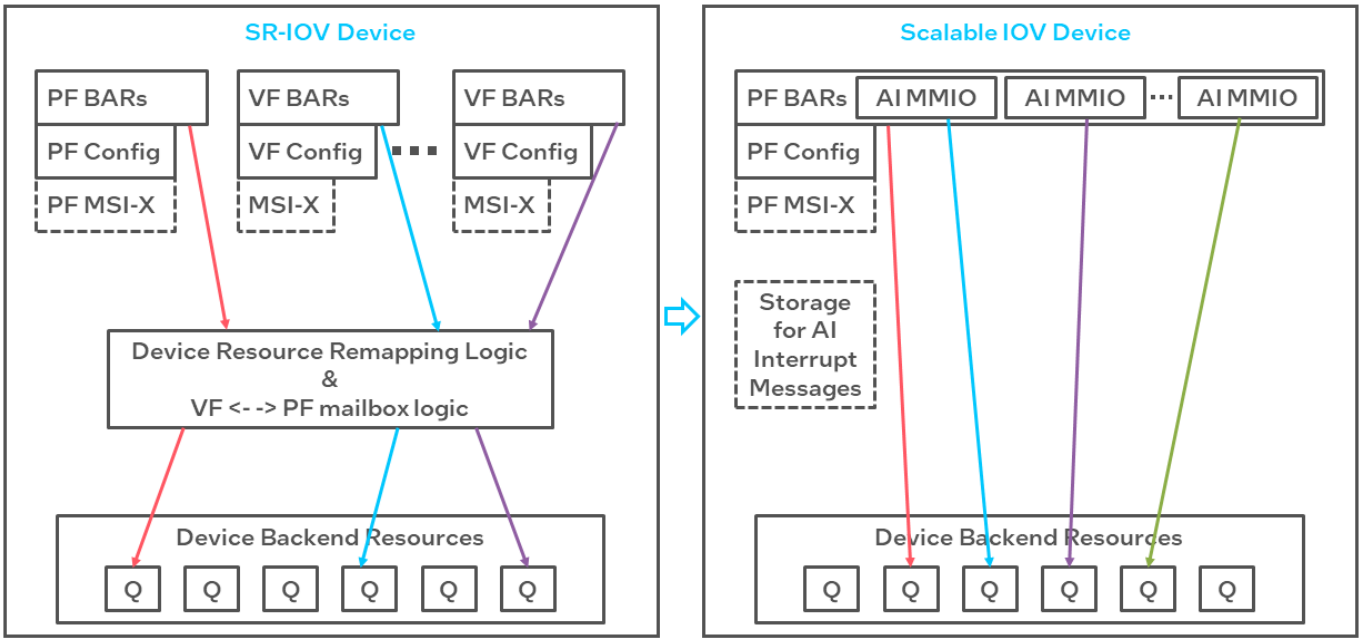
8.5 Reducing Number of VFs per Endpoint

Note: Reducing number of VFs per endpoint is supported starting with QAT Gen 4.

When the acceleration software is installed for SR-IOV use case, all VFs are enabled. In some instances, it is not desirable to enable all VFs.

The following commands can be used to limit VFs exposed per PF.

1. **Disable** all VFs on a specific device (6b:00.0 in this example):




```
echo 0 > /sys/bus/pci/devices/0000\:6b:00.0/sriov_numvfs
```

2. **Enable** number of desired VFs for specific device (4 VFs on 6b:00.0 in this example):

```
echo 4 > /sys/bus/pci/devices/0000\:6b:00.0/sriov_numvfs
```

Important:

- Restart the acceleration software for this change to take place.
- This change is not persistent. After a reboot, all VFs are exposed per PF.

After reducing the number of VFs per PF, it is possible that the mapping of QAT VF to configuration file has changed. This mapping is very important especially when there are different services enabled with each PF/VF configuration, remembering that to enable a service in a VF requires the same service to be enabled in the PF. The configuration file for the VF can be determined by examining the qat_service output.

For example, in the following output:

```
Checking status of all devices.
There is 100 QAT acceleration device(s) in the system:
qat_dev0 - type: 4xxx,  inst_id: 0,  node_id: 0,  bsf: 0000:6b:00.0,  #accel: 1
 ↪#engines: 9 state: up
qat_dev1 - type: 4xxx,  inst_id: 1,  node_id: 0,  bsf: 0000:70:00.0,  #accel: 1
 ↪#engines: 9 state: up
qat_dev2 - type: 4xxx,  inst_id: 2,  node_id: 0,  bsf: 0000:75:00.0,  #accel: 1
 ↪#engines: 9 state: up
```

(continues on next page)

(continued from previous page)

```
qat_dev3 - type: 4xxx,  inst_id: 3,  node_id: 0,  bsf: 0000:7a:00.0,  #accel: 1
↪#engines: 9 state: up
qat_dev4 - type: 4xxx,  inst_id: 4,  node_id: 1,  bsf: 0000:e8:00.0,  #accel: 1
↪#engines: 9 state: up
qat_dev5 - type: 4xxx,  inst_id: 5,  node_id: 1,  bsf: 0000:ed:00.0,  #accel: 1
↪#engines: 9 state: up
qat_dev6 - type: 4xxx,  inst_id: 6,  node_id: 1,  bsf: 0000:f2:00.0,  #accel: 1
↪#engines: 9 state: up
qat_dev7 - type: 4xxx,  inst_id: 7,  node_id: 1,  bsf: 0000:f7:00.0,  #accel: 1
↪#engines: 9 state: up
qat_dev8 - type: 4xxxvf,  inst_id: 80,  node_id: 0,  bsf: 0000:6b:00.1,  #accel: 1
↪#engines: 1 state: up
qat_dev9 - type: 4xxxvf,  inst_id: 81,  node_id: 0,  bsf: 0000:6b:00.2,  #accel: 1
↪#engines: 1 state: up
qat_dev10 - type: 4xxxvf,  inst_id: 82,  node_id: 0,  bsf: 0000:6b:00.3,  #accel: 1
↪#engines: 1 state: up
qat_dev11 - type: 4xxxvf,  inst_id: 83,  node_id: 0,  bsf: 0000:6b:00.4,  #accel: 1
↪#engines: 1 state: up
qat_dev12 - type: 4xxxvf,  inst_id: 84,  node_id: 0,  bsf: 0000:70:00.1,  #accel: 1
↪#engines: 1 state: up
qat_dev13 - type: 4xxxvf,  inst_id: 85,  node_id: 0,  bsf: 0000:70:00.2,  #accel: 1
↪#engines: 1 state: up
qat_dev14 - type: 4xxxvf,  inst_id: 86,  node_id: 0,  bsf: 0000:70:00.3,  #accel: 1
↪#engines: 1 state: up
qat_dev15 - type: 4xxxvf,  inst_id: 87,  node_id: 0,  bsf: 0000:70:00.4,  #accel: 1
↪#engines: 1 state: up
qat_dev16 - type: 4xxxvf,  inst_id: 88,  node_id: 0,  bsf: 0000:75:00.1,  #accel: 1
↪#engines: 1 state: up
qat_dev17 - type: 4xxxvf,  inst_id: 89,  node_id: 0,  bsf: 0000:75:00.2,  #accel: 1
↪#engines: 1 state: up
qat_dev18 - type: 4xxxvf,  inst_id: 90,  node_id: 0,  bsf: 0000:75:00.3,  #accel: 1
↪#engines: 1 state: up
qat_dev19 - type: 4xxxvf,  inst_id: 91,  node_id: 0,  bsf: 0000:75:00.4,  #accel: 1
↪#engines: 1 state: up
```

The configuration file name will be `/etc/4xxxvf_dev<x>.conf` where *x* is `inst_id`.

For *qat_dev9*, the configuration file is `/etc/4xxxvf_dev81.conf`

9 Secure Architecture Considerations

This section describes the potential threats identified as part of the secure architecture analysis of the Intel® Quick Assist Technology acceleration complex and the actions that can be taken to protect against these threats. This chapter concentrates on the acceleration complex. First, the terminology covering the main threat categories and mechanisms, attacker privilege and deployment models are presented. Then, some common mitigation actions that can be applied to many of these threat categories and mechanisms are discussed. Finally, more specific threat/attack vectors, including attacks against specific services of the PCH device are described.

9.1 Terminology

Each of the potential threat/attack vectors discussed may be described in terms of the following:

- *Threat Categories*
- *Attack Mechanism*
- *Attacker Privilege*
- *Deployment Models*

9.1.1 Threat Categories

System threats can be classified into the categories in the following table.

Table 43: Threat Categories

Category	Nature of Threat and Examples
Exposure of Data	Attacker reads data to which they should not have read access.
	Attacker reads cryptographic keys.
Modification of Data	Attacker overwrites data to which they should not have write access.
	Attacker overwrites cryptographic keys.
Denial of Service	Attacker causes application or driver software (running on an IA core) to fail or terminate.
	Attacker causes Intel® QuickAssist Accelerator firmware to hang, temporarily impeding service.

continues on next page

Table 43 – continued from previous page

Category	Nature of Threat and Examples
	Attacker causes excessive use of resource (IA core, Intel® QuickAssist Accelerator firmware thread, silicon slice, PCIe* bandwidth, and so on), thereby reducing availability of the service to legitimate client.

9.1.2 Attack Mechanism

Attack Mechanisms and Examples

Some of the mechanisms by which an attacker can carry out an attack are listed in the following table.

Table 44: Attack Mechanism

Mechanism	Examples
Contrived Packet Stream	Attacker crafts a packet stream that exploits known vulnerabilities in the software, firmware, or hardware. This could include vulnerabilities such as buffer overflow bugs, lack of parameter validation, and so on.
Compromised Application Software	Attacker modifies the application code calling the Intel® QuickAssist Technology API to exploit known vulnerabilities in the driver/hardware.
Application Malware	In an environment where an attacker may be able to run their own application, separate from the main application software, they may invoke the Intel® QuickAssist Technology API to exploit known vulnerabilities in the driver/hardware.
Compromised IA driver software	Attacker modifies the IA driver to exploit known vulnerabilities in the driver/hardware.
Defect	It is also possible that the attack is not malicious, but rather an unintentional defect.

9.1.3 Attacker Privilege

The following table describes the privileges that an attacker may have. The table describes the case of a non-virtualized system.

Table 45: Attacker Privilege

Privilege	Comments
Physical access	There is no attempt to protect against threats, such as signal probes, where the attacker has physical access to the system. Customers can protect their systems using physical locks, tamper-proof enclosures, Faraday cages, and so on.
Logged in as privileged user	There is no attempt to protect against threats where the attacker is logged in as a privileged user. Customers can protect their systems using strong, frequently changed passwords, and so on.

continues on next page

Table 45 – continued from previous page

Privilege	Comments
Logged in as unprivileged user	If the attacker is logged into a platform as an unprivileged user, it is important to ensure that they cannot use the services of the PCH to access (read or write) any data to which they would not otherwise have access.
Ability to send packets	In almost all deployments, attackers have the ability to send arbitrary packets from the network into the system. It is assumed that threats (for example, denial of service attacks) may arrive in this way.

9.1.4 Deployment Models

Some of the possible deployment models are given in the following table.

Table 46: Deployment Models

Deployment Model	Examples
System with no untrusted users	▪ Network security appliance ▪ Server in data center
System with potentially untrusted users	▪ Server in data center

9.2 Threat/Attack Vectors

A thorough analysis has been conducted by considering each of the threat categories, attack mechanisms, attacker privilege levels, and deployment models. As a result, the following threats have been identified. Also described are the steps a user of the PCH chipset can take to mitigate against each threat. Some general practices that mitigate many of the common threats are considered first. Thereafter, threats on specific services and mitigation against those threats are described.

9.2.1 General Mitigation

The following mitigation techniques are generic to different threats and attack vectors:

- Ensure that all software running on the platform that has access to Intel® Quick Assist Technology devices is within the trust boundary of the platform owner. This mitigation includes software running in virtual machines and containers.
- Intel® follows Secure Coding guidelines, including performing code reviews and running static analysis on its driver software and firmware, to ensure its compliance with security guidelines. It is

recommended that customers follow similar guidelines when developing application code. This should include the use of tools such as static analysis, fuzzing, and so on.

- Ensure each hardware component, including the PCH chipset, processor, and DRAM, is physically secured from attackers. This can include such examples as physical locks, tamper proofing, and Faraday cages (to prevent side-channel attacks via electromagnetic radiation).
- Ensure that network services not required on the module are not operating and that the corresponding network ports are locked down.
- Use strong passwords to protect against dictionary and other attacks on administrative and other login accounts.

9.2.2 General Threats

General threats include the following:

- *DMA*
- *Intentional Modification of IA Driver*
- *Modification of the QAT Configuration File*
- *Malicious Application Code*
- *Denial of Service*

9.2.2.1 DMA

Threat: The PCH can perform Direct Memory Access (DMA, the copying of data) between defined memory locations. Once an attacker has sufficient privilege to invoke the Intel® QuickAssist Technology API, or to write to/read from the hardware rings used by the driver to communicate with the device, they can send requests to the Intel® QuickAssist Accelerator to perform such DMA, passing arbitrary physical memory addresses as the source and/or destination addresses, thereby exposing or modifying regions of memory to which they would otherwise not have access.

Mitigation 1: Ensure that Intel® Input-Output Memory Management Unit (IOMMU) is enabled. This will force USDM to create QuickAssist IOMMU domain and all memory allocated by USDM will be mapped into this domain, hence malicious user or error in user application cannot read or write memory outside this domain which mitigates the risk. However because there is only single domain, there is no protection between individual Virtual Functions(VFs) or applications. This design is done for simplicity of memory manager and if needed, VFIO-PCI should be used to create individual domains per VF.

Mitigation 2: Ensure that only trusted users are granted permissions to access the Intel® QuickAssist Technology API, or to write to and read from the hardware rings. Specifically, the PCH configuration file describes logical instances of acceleration services and the set of hardware rings to be used for each such instance. User processes can ask the kernel driver to map these rings into their address spaces. To access a given device (identified by the number in the filenames below), the user must be granted read/write access to the following files, which may be in `/dev`:

- `uio<0..N>` (where `<0..N>` are the qat uio device numbers)
- `qat*`
- `usdm_drv`

9.2.2.2 Intentional Modification of IA Driver

Threat: An attacker can potentially modify the IA driver to behave maliciously. This may lead to a denial of service of Intel® Quick Assist Technology services.

Mitigation: The driver object/executable file on disk should be protected using the normal file protection mechanisms so that it is writable only by trusted users, for example, a privileged user or an administrator. Specifically, the Intel® QuickAssist Technology kernel objects and libraries should not be writeable by user. If the *qat* user group is being used to provide access to Intel® Quick Assist Technology services, then this group should not have write permission to the binaries.

9.2.2.3 Modification of the QAT Configuration File

Threat: The QAT configuration file is read at initialization time by the driver and specifies what instances of each service (cryptographic, data compression) should be created, and which rings each service instance will use. Modifying this file could lead to denial of service by deleting required instances or could be used to attempt to create additional instances that the attacker could subsequently attempt to access for malicious purposes.

Mitigation: The configuration file should be protected using the normal file protection mechanisms so that it is writable only by trusted users, for example, a privileged user or an administrator.

Note: By default, the configuration file is stored in the `/etc` directory and may be named something like, `c6xxx_dev0.conf` . Its default permissions are that it is readable and writeable only by *root* user and *qat* group.

9.2.2.4 Malicious Application Code

Threat: An attacker who can gain access to the Intel® QuickAssist Technology API may be able to exploit the following features of the API:

- Buffers passed to the API have a specified length of up to 32 bits. By specifying excessive lengths, an attacker may be able to cause denial of service by overwriting data beyond the end of a buffer.
- Buffer lists passed to the API consist of a scatter gather list (array of buffers). An attacker may incorrectly specify the number of buffers, causing denial of service due to the reading or writing of incorrect buffers.

Mitigation: Platform management can include the Rate Limiting feature to mitigate against Noisy Neighbors. Only trusted users and applications should be allowed to access the Intel® QuickAssist Technology API, as described in General Mitigations.

9.2.2.5 Denial of Service

Threat: An attacker may construct a service request that does not conform to the specification, resulting in low of service due to service timeouts, halting of Quick Assist service or undesired platform level conditions.

Mitigation: The current generation of Intel® Quick Assist Technology has been designed for performance, providing direct access to hardware via PCIe* MMIO space. Misuse of hardware registers is to be avoided, and the threat against intentional misuse must be mitigated by ensuring all software on the platform is trusted.

An attacker may attempt to contrive a packet stream that monopolizes the acceleration services, thereby denying service to legitimate users. This may consist of one or more of the following:

- Sending packets that are compressed (for example, using IPComp) or encrypted (for example, using IPsec), thereby reducing the availability of these services to legitimate traffic.
- Sending excessively large packets, causing some latency for legitimate packets.
- Sending small packets at a high packet rate, causing extra bandwidth utilization on the PCI Express* bus connecting the device to the processor.

Mitigation: Proper monitoring of Device Usage (DU) and the construction of Service Level Agreements (SLA) are now available as part of the Rate Limiting feature.

9.2.3 Threats Specific to Cryptographic Service

Threats against the cryptographic service include:

9.2.3.1 Reading Cryptographic Keys

Threat: Cryptographic keys are stored in DRAM. An attacker who can determine where these are stored could read the DRAM to get access to the keys or could write the DRAM to use keys known by the attacker, thereby compromising the confidentiality of data protected by these keys. Some cryptographic keys have long lives. The impact of an attacker obtaining the key may exist for the lifetime of the key itself.

Mitigation: DRAM is considered inside the cryptographic boundary (as defined by FIPS 140-2). The normal memory protection schemes provided by the Intel® architecture processor and memory controller, and by the operating system, prevent unauthorized access to these memory regions.

10 Revision History

Document Version	Description	Date
004	Updates for 1.1.40 Release	March 2024
003	RSA-1024 added as Opt-in.	June 2023
002	Note added about using SR-IOV and S-IOV simultaneously on same PF (not supported).	May 2023
001	Initial Release	February 2023