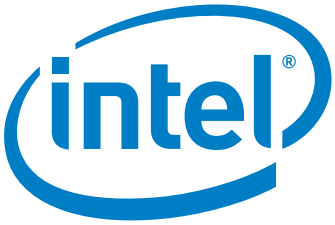


Intel® QuickAssist Technology Software for Linux*

Programmer's Guide - Hardware Version 1.7

October 2019

Document Number: 336210-010

Contents

Figures

Tables

Revision History

Date	Revision	Description
October 2019	010	Updates for 4.7.0 release: • Added virtual functions to list of configurable instances • Rate limiting and device utilization measurement features
July 2019	009	Updated configuration options for concurrent requests (Tables 10, 14, 15)
June 2019	008	Updates for 4.6.0 release: • Dummy responses added to heartbeat feature • Handling device failures in a virtualized environment
March 2019	007	Updates for 4.5.0 release: • Updated list of general parameters • Updated list of Intel® QuickAssist entries in `/sys/kernel/debug`
December 2018	006	Updates for 4.4.0 release: • Updated list of Compression API Errors
September 2018	005	Updates for 4.3.0 release: • Intel® QuickAssist API in kernel space • Added epoll content • Updates for the Compress and Verify and Recover feature • Other minor changes
June 2018	004	Added description of Compress and Verify and Recover (CnVnR) capability.
April 2018	003	Added Heartbeat description. Clarified explanations of stateless and stateful compression and decompression.
April 2018	002	Stateful compression is no longer supported by default.
August 2017	001	Initial public release.

§

1 Introduction

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 (refer to Table 2, Reference Documents and Resources).

1.1 Terminology

In this document, for convenience:

- Software package is used as a generic term for the Intel® QAT software package for Hardware Version 1.7.
- 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).

Table 1. Terminology

Term	Description
ADF	Acceleration Driver Framework
AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
BDF	Bus Device Function
CBC	Cipher Block Chaining mode
CCM	Counter with CBC-MAC mode
CnV	Compress and Verify
CnVnR	Compress and Verify and Recover
CY	Cryptography
DC	Data Compression
DID	Device ID
DMA	Direct Memory Access
DTLS	Datagram Transport Layer Security
DRAM	Dynamic Random Access Memory
DSA	Digital Signature Algorithm
ECC	Elliptic Curve Cryptography
EVP	Envelope (OpenSSL high-level cryptographic functions)
GCM	Galois/Counter Mode
GPL	General Public License
HMAC	Hash-based Message Authentication Mode
IA	Intel® Architecture
IDS/IPS	Intrusion Detection System/Intrusion Prevention System

Table 1. Terminology

Term	Description
IEEE	Institute of Electrical and Electronics Engineers
IKE	Internet Key Exchange
Intel® QAT	Intel® QuickAssist Technology
IOCTL	Input Output Control function
IOMMU	Input-Output Memory Management Unit
IPSec	Internet Protocol Security
LKCF	Linux* Kernel Cryptographic Framework
MGF	Mask Generation Function
MSI	Message Signaled Interrupts
NUMA	Non-uniform Memory Access
PCH	Platform Controller Hub. In this manual, a Platform Controller Hub device includes standard interfaces and Intel® QAT endpoint and I/O interfaces.
RSA	Rivest-Shamir-Adleman
SAL	Service Access Layer
SATA	Serial Advanced Technology Attachment
SGL	Scatter Gather List
SHA	Secure Hash Algorithm
SoC	System-on-a-Chip
SPI	Serial Peripheral Interconnect
SR-IOV	Single Root I/O Virtualization
SSC	Storage Subsystem Class
SSL	Secure Sockets Layer
TCG	Trusted Computing Group
TLS	Transport Layer Security
TPM	Trusted Platform Module
USDM	User Space DMA-able Memory
VF	Virtual Function
VPN	Virtual Private Network
WAN	Wide Area Network

1.2 Document Organization

This document is organized as follows:

- Part 1 provides an overview of the supported hardware and an overview of the software architecture.
- Part 2 describes the acceleration drivers included in the software package.

- Part 3 provides information on specific applications and software usage models.

Table 2. Reference Documents and Resources

Document	Document # / Location
Intel® QuickAssist Technology Software for Linux Release Notes (Hardware Version 1.7)*	336211
Intel® QuickAssist Technology Software for Linux Getting Started Guide (Hardware Version 1.7)*	336212
Intel® QuickAssist Technology API Programmer's Guide	330684
Intel® QuickAssist Technology Cryptographic API Reference Manual	330685
Intel® QuickAssist Technology Data Compression API Reference Manual	330686
Using Intel® Virtualization Technology (Intel® VT) with Intel® QuickAssist Technology Application Note	330689

1.3 Typographical Conventions

The following conventions are used in this manual:

- `Courier font`- file names, path names, executables, code examples, command line entries, API names, parameter names and other programming constructs
- *Italic text* – key terms and publication titles
- **Bold text** - graphical user interface entries, buttons, keyboard keys and Intel® software names

§

2 Software Overview

In addition to the hardware mentioned in Section 3.1, "Hardware/Software Overview", the respective platforms have critical software components that are part of the offering. The software includes drivers and acceleration code that runs on the Intel® Architecture (IA) CPUs and on Intel® QAT endpoints.

2.1 Intel® Communications Chipset 8925 to 8955 Series Compatibility

While the focus of this document is on Intel® QAT software for Hardware Version 1.7, the Intel® Communications Chipset 8925 to 8955 Series is also supported.

2.2 Logical Instances

A logical instance may be thought of as a channel to the hardware. A logical instance allows an address domain (that is, kernel space and individual user space processes) to configure the rings to be used by that address domain and to define the behavior of that ring.

2.2.1 Response Processing

In the kernel space, each logical instance can be configured to operate in one of the two modes:

- Interrupt mode
- Polled mode

In the user space, each logical instance can be configured to operate in one of the two modes:

- Polled mode
- Epolled mode

2.2.1.1 Interrupt Mode

The interrupt is only supported in Kernel space. In User space it is no longer supported; therefore, the user space instance can no longer be configured with interrupt enabled mode.

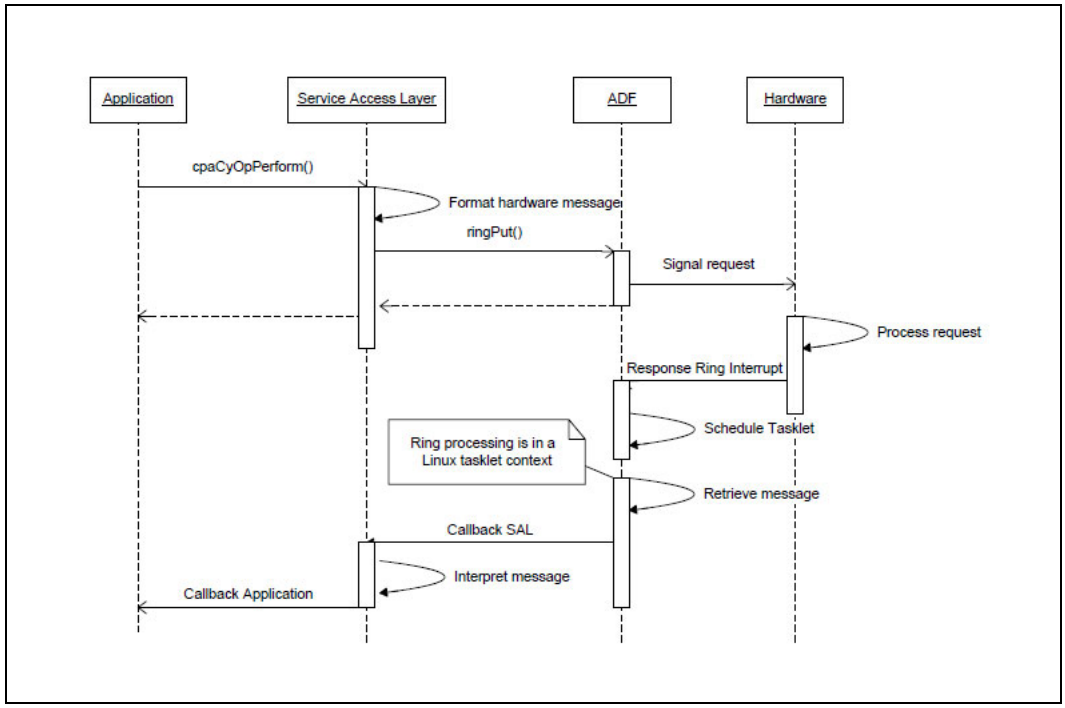
When configured in interrupt mode, the Accelerator Driver Framework (ADF) registers an interrupt handler for response ring processing.

As the latency in servicing an interrupt may be costly, the hardware assisted ring provides a mechanism to amortize the cost of an interrupt into a single interrupt that may service multiple responses. The interrupt coalescing section of the configuration file allows the user to select the mechanism to amortize response interrupts using either a time-based interrupt scheme or a number-of-responses-based scheme.

The ADF registers an interrupt handler to service the ring bank interrupt. When an interrupt fires, the ADF services the interrupt and creates an interrupt handler bottom half to consume the responses from the response ring. When MSI-X is supported, the bottom half of the interrupt handler is created and affinitized to the configured core. Callbacks to the application code occur in the context of this tasklet. This sequence is shown in the following figure (the full sequence has been reduced for clarity).

Note: Linux (and other operating systems) split an interrupt handler into two halves. The so-called "top half" is the routine that actually responds to the interrupt, that is, the one you register with request_irq. The "bottom half" is a routine that is scheduled by the top half to be executed later, at a safer time.

Figure 1. Kernel Space Response Ring Processing



2.2.1.2 Polled Mode

If the cost of servicing an interrupt and scheduling the interrupt handler bottom half is not desired, a user can choose to disable interrupts and poll for responses. This mechanism can be configured on a per logical instance basis by setting the or `Dc/CyXIsPolled` attribute of a logical instance in the configuration file to 1. When configured to 1, the ADF does not service interrupts for that logical instance.

The ADF provides a set of APIs to allow the client to poll a single bank or all banks on a given accelerator:

- `icp_sal_pollBank` - Poll the rings on the given bank number for a given accelerator.
- `icp_sal_pollAllBanks` - Poll the rings on all banks for a given accelerator.

The Service Access Layer (SAL) provides an API to poll on an individual logical instance:

- `icp_sal_CyPollInstance` - Poll a specific cryptographic (Cy) logical instance
- `icp_sal_DcPollInstance` - Poll a specific data compression (Dc) logical instance

Refer to Section 5.2.3, "Polling Functions" for details on all the polling functions.

2.2.1.3 Epolled Mode

The event-based poll mode is called "epoll mode". The Intel® QuickAssist Technology driver's new mode supports the Linux epoll interface. The Linux epoll is a scalable I/O event notification mechanism intended to replace the older select/poll system calls.

Note: There is a limit of one instance (and one process) per bank in epoll mode.

In order to use the Linux epoll, the user space application uses the following APIs:

- `epoll_create()/epoll_create1()` - creates an epoll instance and returns a file descriptor referring to that instance
- `epoll_ctl()` - registers the file descriptors which will be polled at
- `epoll_wait()` - waits for I/O events for the file descriptors registered via `epoll_ctl`, blocking the calling thread if no events are currently available

For more information, please consult the Linux epoll manuals, here:
http://man7.org/linux/man-pages/man7/epoll.7.html

The Intel® QuickAssist Technology driver's epoll mode is only used by the user space instances, it is not valid for the kernel space. The QAT driver's epoll mode consists of two parts: the kernel space part and the user space part.

The Coalescing fields expose the same behavior for the epoll mode. If the interrupt is delayed by changing the Coalescing fields, the event delivery to user space will be delayed too.

To enable epoll mode, please ensure the following steps are followed:

1. In the configuration file, please use the "IsPolled = 2" for the user space instance; for example:

   ```
   Cy0Name = "SSL0"
   Cy0IsPolled = 2
   ```

2. Whether the application uses the driver in a synchronous or asynchronous manner, it should create a thread to call the Intel® QuickAssist Technology driver's epoll API and the Linux standard epoll interface.

The Intel® QuickAssist Technology driver's epoll API:

Crypto: `icp_sal_CyGetFileDescriptor() / icp_sal_CyPutFileDescriptor()`
Compression: `icp_sal_DcGetFileDescriptor() / icp_sal_DcPutFileDescriptor()`

The Linux standard epoll interface:

`epoll_create() / epoll_ctl() / epoll_wait()`

There is just one limitation for the epoll mode: Only configure one user space instance for a bank. The instance can be a crypto or compression instance.

When a bank is used for the epoll mode, it means there is only one instance (crypto or compression) for this bank. When the instance is used by a process, it means the process is the only user for this bank. Other processes could not use this bank temporarily. But if the process releases this instance, other processes can use this bank. Since there is only one instance for this bank, no more than 16 user space instances are available to configure all the banks for the epoll mode. (For the Intel® Communications Chipset 8925-8955 series, up to 32 user space instances are available.)

If a process needs to provide compression service and crypto service at the same time, it will need two instances, which means the process needs two banks. In such a scenario, no more than eight processes can be used. (For the Intel® Communications Chipset 8925-8955 series, up to 16 processes can be used.)

For comparison purposes, when the CPU is in the idle state, for the user space instance, the standard poll mode ("`IsPolled = 1`") will poll the empty rings periodically and the polling will consume some CPU cycles (for instance, 2% usage may appear available when the CPU is in the idle state). But if epoll mode is used, the usage will stay at 0% when the CPU is in the idle state.

Note: The standard poll mode performs better when the CPU is in the high load state.

For user space instances, interrupt mode is no longer supported. Interrupt mode for the user space did not consume CPU cycles when there was no data in the response rings, unlike polling mode, which would continue to check at specified intervals. With the epoll support, standard Linux* epoll APIs, such as epoll_create()/epoll_ctl()/ epoll_wait(), can be used. Most web servers and socket-based applications, such as Nginx*, Apache*, etc., use one of epoll/select/poll to be notified when a socket is available for reading or writing, and then take appropriate action. With the epoll mode, the Intel® QuickAssist Technology driver will have more seamless integration into existing applications, such as Nginx*, as it will be using a standard notification mechanism.

§

3 Acceleration Drivers Overview

Selected products in the Intel® C620 Series Chipsets (also called Platform Controller Hub, or PCH), the Intel Atom® C3000 Processor Product Family (also called System-on-a-Chip, or SoC), and the Intel® Xeon® Processor D Family SoC support Intel® QuickAssist Technology. Depending on the product chosen, Intel® QAT accelerates both or either of two services: cryptography (both symmetric and public key) and data compression. 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.

3.1 Hardware/Software Overview

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. For example, to support customers with different acceleration performance requirements, the Intel® C62x Chipset is available in different SKUs, and also supports two different "fabric configurations". Figure 2 and Figure 3 show two possible configurations for the acceleration endpoints in one Intel® C62x Chipset die.

Figure 2. Intel® C62x Chipset (PCH) Acceleration Endpoint Configuration 1

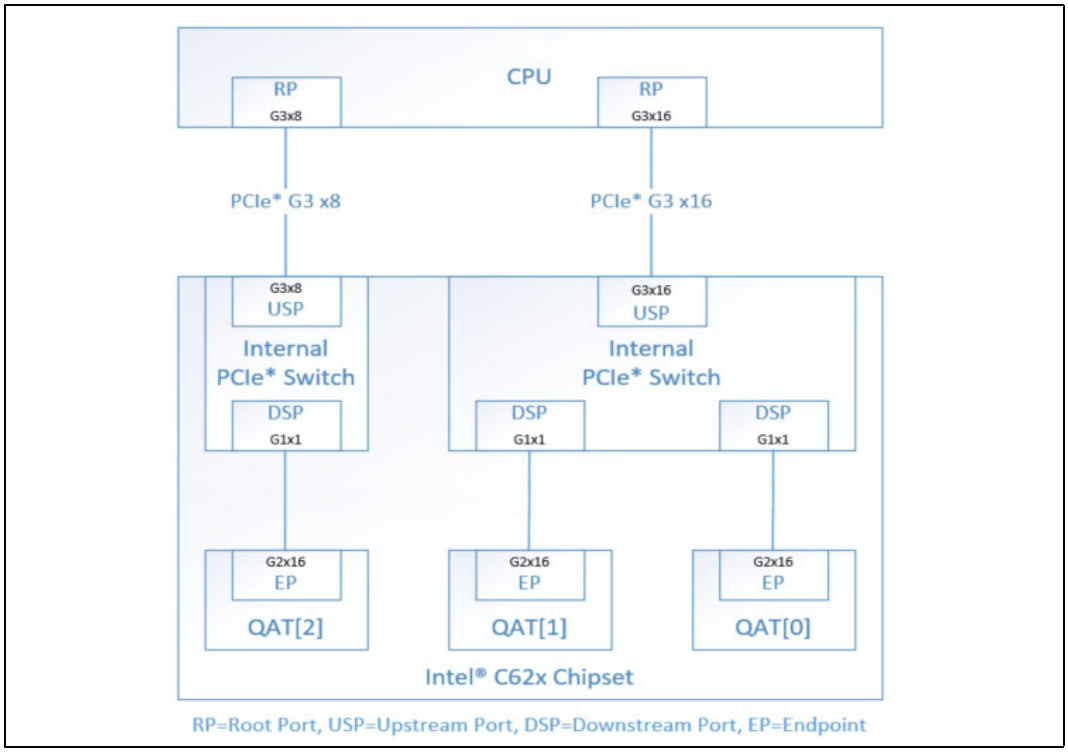
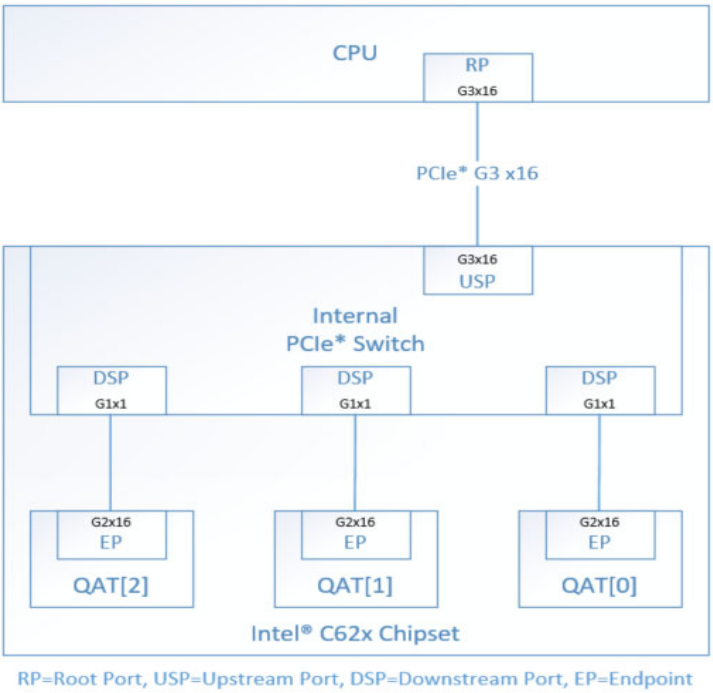


Figure 3. Intel® C62x Chipset (PCH) Acceleration Endpoint Configuration 2



For a given platform, the specific internal connections and number of Intel® QAT endpoints per die (for instance, up to three for Intel® C62x Chipset) is product-dependent, SKU-dependent, routing-dependent (i.e., how many lanes are routed), and configuration-dependent (e.g., with different fabric configuration softstraps). For each QAT endpoint (e.g., `QAT[0]`), hardware-assisted rings are used as the communication mechanism to transfer requests between the CPU and the QAT endpoint(s) and vice-versa. The hardware supports 256 rings (per QAT endpoint), each with head and tail Configuration Status Register (CSR) pointers that are mapped to PCIe* memory on the CPU. Rings are assigned by the provided software based on the cryptography (CY) and data compression (DC) instances declared in the configuration files. Refer to Section 3.2, Acceleration Driver Configuration File for more information.

Each Intel® QAT endpoint has multiple computation engines. For a given QAT endpoint, all rings associated with that endpoint are shared, and the hardware load balances requests from these rings.

A user can write directly to the Intel® QAT APIs, or the use of QAT can be done via frameworks that have been enabled by others including Intel (for example, zlib*, OpenSSL* libcrypto*, and the Linux* Kernel Crypto Framework).

The driver architecture supports simultaneous operation of multiple applications.

3.2 Acceleration Driver Configuration File

An acceleration driver has a configuration file that is used to configure the driver for runtime operation. 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, if applicable. The configuration file format is described in Section 4.1, Configuration File Overview.

3.3 Utility for Loading Configuration Files and Sending Events to the Driver - 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 1.7 driver can be used to interface with Intel® QAT 1.7 devices and Intel® QAT 1.6 devices.

Usage

`./adf_ctl [dev] [up|down|restart|reset]` - to bring up, down, restart or reset device(s)

or

`./adf_ctl status` - to print device(s) status

For instance:

```
# ./adf_ctl qat_dev0 down
# ./adf_ctl qat_dev1 up
```

3.4 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 is 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 (refer to Table 3).

Table 3. 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.

3.5 User Space Additional Functions

To allow a user space process access to the Intel® QAT rings, the service access layer must be configured to expose logical instances to the user space process. Logical instances are configured using the per device configuration file.

To allow each process to have separate logical instances, the configuration file groups a set of logical instances by name. The process then must call the `icp_sal_userStart` function (refer to Section 5.2.4.1) at initialization time with the name associated with the group of logical instances. Similarly, on process exit, to free the resources and make them available to other processes with the same name, the process must call the function `icp_sal_userStop` (refer to Section 5.2.4.2).

For example, the user can configure the driver to have two crypto logical instances available for the process called "SSL". The user space process may then access these logical instances by calling the `cpaCyGetInstances` function. The application may then initiate a session with these logical instances and perform a cryptographic operation. See the *Intel® QuickAssist Technology Cryptographic API Reference Manual* (refer to Table 2, Reference Documents and Resources) for more information on the API functions available for use.

For this example, the logical instances section of the configuration file is as follows:

```
[SSL]
NumberCyInstances = 2
NumberDcInstances = 0
NumProcesses = 1
LimitDevAccess = 0

# Crypto - User instance #0
Cy0Name = "SSL0"
Cy0IsPolled = 1
# List of core affinities
Cy0CoreAffinity = 1

# Crypto - User instance #1
Cy1Name = "SSL0"
Cy1IsPolled = 1
# List of core affinities
Cy1CoreAffinity = 2
```

In this example, the user process SSL configures two logical instances (called "SSL0" and "SSL1").

3.6 Managing Intel® QuickAssist Technology Endpoints Using 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.

Usage:

```
# ./qat_service start||stop||status||restart||shutdown
```

To view all QAT endpoints in the system, use:

```
# ./qat_service status
```

If there are two Intel® QAT endpoints in the system, for example, 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
```

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:

```
# ./qat_service stop qat_dev0
```

where the device number <N> is equal to 0 in this case.

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 endpoints can still run.

Note: In systems with more than three devices it might be necessary to change the `qat_service` timeout in `/etc/systemd/system/qat_service.service.d/startup-timeout.conf`.

3.7 Intel® QuickAssist Technology Entries in the /sys/kernel/debug Filesystem

Debug information for the driver and configuration is available with the entries `/sys/kernel/debug/qat_*`. This includes:

Table 4. Intel® QuickAssist Technology /sys/kernel/debug Entries

Entry	Description
`cnv_errors`	Indicates number of compressAndVerify errors. Refer to Section 5.1.5, Compress and Verify Error log in Sysfs:
`dev_cfg`	Displays internal device configuration information
`frequency`	Displays frequency of Acceleration Engines
`fw_counters`	Displays Acceleration Engine firmware requests/responses

`heartbeat`
`heartbeat_failed`
`heartbeat_sent`
	Refer to Section 3.17.3.3, System Virtual Files

`transport`	Contains firmware request/response data. Available only for kernel space instances.
`version`	Includes package version information

3.8 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.

3.8.1 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 the following table. These APIs are:

- `cpaDcCompressData()`
- `cpaDcDecompressData()`
- `cpaDcDpEnqueueOp()`
- `cpaDcDpEnqueueOpBatch()`

Table 5. Intel® QuickAssist Technology 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().
-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().

Table 5. Intel® QuickAssist Technology Compression API Errors (Continued)

Error Code	Error Type	Description	Suggested Corrective Action(s)
-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().
-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().
-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. Table 17 in Section 5.1.1.1 gives details on the various overflow exceptions.

Table 5. Intel® QuickAssist Technology Compression API Errors (Continued)

Error Code	Error Type	Description	Suggested Corrective Action(s)
-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().
-15	CPA_DC_INCOMPLETE _FILE_ERR	This decompression error can be reported only by QAT 1.7 devices. 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.	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.	Discard output and resubmit the affected request.
-17	CPA_DC_EP_HARDWARE _ERR	This is a recoverable error available only with QAT1.7 devices. 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_ERROR	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.
-19	CPA_DC_EMPTY_DYM_BLK	Decompression request contained an empty dynamic stored block (not supported).	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.

Note: 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.

3.9 Stateful Compression Unsupported

Stateful compression is no longer supported.

3.10 Stateless Compression Level Details

Throughput and compression ratio for stateless compression can be adjusted with the compression levels to achieve particular requirements. The most recent software packages now support four compression levels and the history buffer size is ignored. Compression levels 5 to 9 are retained for backwards compatibility, but map to level 4. Compression levels 1 to 4 translate to search depth 1, 4, 8, and 16, respectively.

3.11 Acceleration Driver Return Codes

Table 6 shows the return codes used by various components of the acceleration driver, defined in `quickassist/include/cpa.h`.

Table 6. Acceleration Driver 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 shutdown and restart the component.
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.

Table 7 shows the return codes used by the driver to handle Linux* device driver operations.

Table 7. Acceleration Driver Return Codes for Linux* Device Driver Operations

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.
-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.

Table 7. Acceleration Driver Return Codes for Linux* Device Driver Operations

-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.
-ENODEV	-19	No Such Device. Used when an operation detects invalid device id.
-ENOTTY	-25	Invalid Command Type. Used when an ioctl operation detects an invalid command type.

3.12 Batch and Pack Compression Unsupported

Batch and Pack (BnP) compression is no longer supported.

3.13 Compress and Verify Feature

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*.

Notes:

1. CnV is always enabled via the `cpaDcCompressData()` API.
2. CnV supports compression operations only.
3. The `compressAndVerify` flag in the `CpaDcDpOpData` structure should be set to `CPA_TRUE` when using the `cpaDcDpEnqueueOp()` or `cpaDcDpEnqueueOpBatch()` API. These API are declared in the API file `cpa_dc_dp.h`.
4. The `compressAndVerify` flag in the `CpaDcOpData` structure should be set to `CPA_TRUE` when using the `cpaDcCompressData2()` API. This API is declared in the API file `cpa_dc.h`.

The CnV functionality is implemented in the Data Compression APIs `cpaDcCompressData()`, `cpaDcCompressData2()`, `cpaDcDpEnqueueOp()` and `cpaDcDpEnqueueOpBatch()` for the compression path only.

These APIs are declared and documented in the API file `cpa_dc.h`.

Note: It is possible to recover from Compress and Verify errors in a seamless manner. Refer to the Compress and Verify and Recover discussion in Section 5.1.3.

3.14 Running Applications as Non-Root User

This section describes the steps required to run Intel® QuickAssist Technology userspace applications as non-root user. This section uses the user space performance sample code as an example.

Assumptions:

- Intel® QuickAssist Technology software is installed and running

- User space Acceleration Sample code (`cpa_sample_code`) compiled and the directory has read/write/execution permission for all the users
- User space DMAable Memory driver (`usdm_drv.ko`) compiled and installed

The following steps should be executed by users with root privilege or root user:

1. Export environmental variables.
   ```
   # export ICP_ROOT=/QAT
   ```
2. Create a linux group to provide access for all users in that group.
   ```
   # groupadd <group_name>
   ```
3. Add users to the new group. The group should only have users who need access to the application.
   ```
   # usermod -G <group_name> <yourusername>
   ```
4. Change group ownership of the following files. By default, the group ownership will be root.
   ```
   /dev/qat_*
   /dev/uio*
   /var/tmp/shm_key_uio_ctl_lock
   /dev/usdm_drv
   ```

   For /dev/qat_*:
   ```
   # chgrp <group_name> /dev/qat_*
   # chmod 660 /dev/qat_*
   ```
   And for /dev/usdm_drv:
   ```
   # chgrp <group_name> /dev/usdm_drv
   # chmod 660 /dev/usdm_drv
   ```
   And for /dev/uio*:
   ```
   # chgrp <group_name> /dev/uio*
   # chmod 660 /dev/uio*
   ```
   If it exists, for /var/tmp/shm_key_uio_ctl_lock:
   ```
   # chgrp <group_name> /var/tmp/shm_key_uio_ctl_lock
   # chmod 660 /var/tmp/shm_key_uio_ctl_lock
   ```
   If using huge pages with the included memory driver, also enter:
   ```
   # chgrp <group_name> /dev/hugepages
   # chmod 660 /dev/hugepages
   ```
5. Change the group ownership for the relevant *.so files:
   For 64-bit OS:
   ```
   # chgrp <group_name> /lib64/libqat_s.so
   # chgrp <group_name> /lib64/libusdm_drv_s.so
   ```
   For 32-bit OS:
   ```
   # chgrp <group_name> /usr/local/lib/libqat_s.so
   # chgrp <group_name> /usr/local/lib/libusdm_drv_s.so
   ```
6. Change the amount of max locked memory for the user name that is included in the group name (which is by default 64).
   This can be done by specifying the limit in `/etc/security/limits.conf`.
   Add the following line to `/etc/security/limits.conf`:
   ```
   <yourusername> - memlock 4096
   ```
7. At this point, switch to user name that is included in `<group_name>`.

```
# su <yourusername>
```

8. Launch the performance sample code.

```
# cd $ICP_ROOT/build/
# ./cpa_sample_code signOfLife=1
```

Note: The same basic steps can be followed to enable non-root access for customer applications accessing the acceleration software. These steps may need to be completed every time the memory driver is loaded or the acceleration software is restarted.

3.15 Random Number Generation

Starting with Intel® QuickAssist Technology Hardware version 1.7, Intel® QuickAssist Technology no longer includes random number generation capability, because this capability is already included in the CPU and is available via the RDRAND and RDSEED instructions.

3.16 Huge Pages with the Included Memory Driver

The included User space DMAable Memory driver (`usdm_drv.ko`) supports 2 MB pages. The use of 2 MB 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:

```
# insmod ./usdm_drv.ko
```
Default settings applied.

```
# insmod ./usdm_drv.ko max_mem_numa=32768
```
Maximum amount of Non-uniform Memory Access (NUMA) type memory that the User Space DMA-able Memory (USDM) driver can allocate is 32MB in total for all processes. Huge pages are disabled.

```
# insmod ./usdm_drv.ko max_huge_pages=50
max_huge_pages_per_process=5
```
Maximum amount of huge pages that the USDM can allocate is 50 in total and 5 per process (up to 10 processes, 0 for the next processes).

```
# insmod ./usdm_drv.ko max_huge_pages=3
max_huge_pages_per_process=5
```
An erroneous configuration, maximum amount of huge pages that USDM can allocate is 3 total; 3 for a first process, 0 for the next processes.

```
# insmod ./usdm_drv.ko max_huge_pages_per_process=5
```
An invalid configuration, huge pages are disabled because `max_huge_pages` is 0 by default.

```
# insmod ./usdm_drv.ko max_huge_pages=5
```
An invalid configuration, huge pages are disabled because `max_huge_pages_per_process` is 0 by default.

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).

3.17 Heartbeat

Under some circumstances, firmware in the Intel® QAT devices could become unresponsive, requiring a device reset to recover. The Intel® QuickAssist 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 QAT instances between the events.

3.17.1 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.

Initialization

At startup, the 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.

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
- Can be a user application that uses 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

Resetting a failed device

A device can be configured for automatic reset by the QAT framework or manually reset by the application by using the `AutoResetOnError` field in the device configuration file `/etc/<device>.conf`, as shown in Table 8.

Table 8. 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 Section 5.2, Additional APIs.

3.17.2 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 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:
  - `CPA_INSTANCE_EVENT_RESTARTING` (device is restarting)
  - `CPA_INSTANCE_EVENT_RESTARTED` (device restart is complete)

  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 Section 5.2, Additional APIs.

3.17.2.1 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 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 requests 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.

3.17.2.2 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.

3.17.2.3 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

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`.

3.17.3 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/`

3.17.3.1 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
```

3.17.3.2 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
  ```

3.17.3.3 System Virtual Files

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

Table 9. 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
```

3.17.3.4 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 QAT service requests that take some time to complete, high frequency Heartbeat monitoring could result in false reports of unresponsiveness.

Checking for device reset events

If the application uses polling for reading 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 QAT service responses.

3.18 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* (refer to Table 2) 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) is loaded, 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 PF-to-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.

Note: 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.

3.19 Incorporating Dummy Responses into a QAT Application

The dummy response feature has been incorporated in a scenario with the Intel® QAT engine and Nginx*. Figure 4 below illustrates how it works. This can be used as a reference to so-called "software fallback."

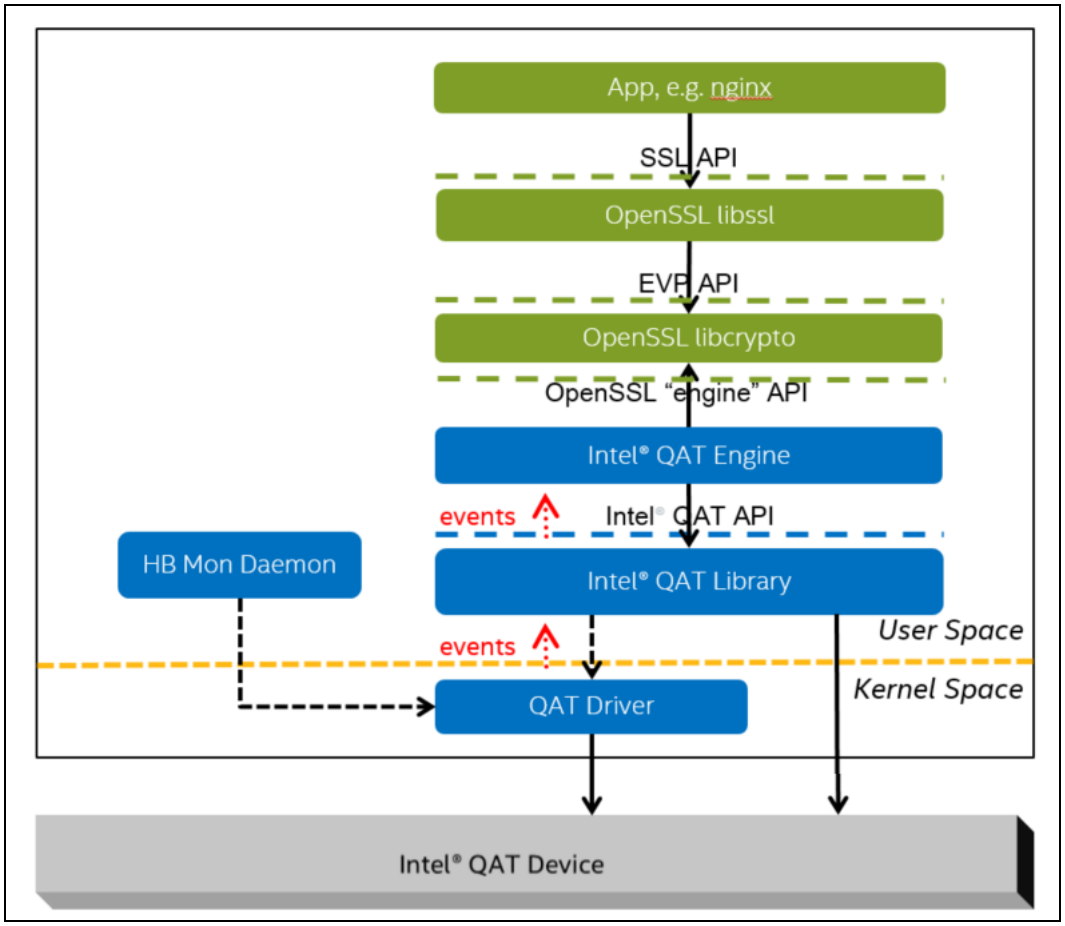
The QAT engine is a shim layer between OpenSSL libcrypto and QAT Library. The QAT Library will generate failover responses.

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 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, 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 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 QAT devices goes positive again, switch back to hardware.

Figure 4. Incorporating Dummy Responses in an Intel® QAT Operation



3.20 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 Rate Limiting feature is supported only in rate limiting firmware for cryptographic services.

To enable the Rate Limiting feature:

1. Install the driver package on the host with SRIOV enabled.
2. Update the physical function configuration file to set `RateLimitingEnabled=1` in the General section.
3. Set `ServicesEnabled` to `cy` or `sym` or `asym`.
4. Perform `qat_service shutdown` and `qat_service start`.

Note: This procedure also enables Device Utilization measurement (refer to Section 3.21).

3.20.1 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 cryptographic)

3.20.2 SLA Units

SLA units are measured as follows:

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

Note: Enforced SLAs are rounded up to the next multiple of 1000 units.

3.20.3 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.

3.20.3.1 Commands to Fetch Device Utilization

- 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 bus:device.function(xx:xx.x) format

`vf_addr` Virtual address in bus:device.function(xx:xx.x) format

`service` Asym(=0) or Sym(=1) cryptographic services

`rate_in_sla_units` [0-MAX]. MAX is found by querying the capabilities.

1 rate_in_sla_units is equal to:

1 operation per second – for asymmetric service

1 Megabits per second – for symmetric service

`sla_id` Value returned by `create` command

Note: An SLA is uniquely identified by `<pf_addr, sla_id>`.

For a given service, device would guarantee minimum `rate_in_sla_units` throughput.

Maximum throughput can be up to the maximum capacity of a device.

Note: Throughput measurement may not meet the 90 percent delivery standard when smaller packet sizes are used.

3.21 DU Manager Application

Device Utilization (DU) is a way to measure utilization of acceleration hardware that corresponds to the throughput of cryptographic services on a given physical or virtual function. This can vary between different device types and generations.

The `du_mgr` tool is used to measure the utilization of cryptographic services for a given physical or virtual function.

The DU execution tool is available in `$ICP_ROOT/build/du_mgr` after the package is built and installed using `./configure; make` install commands.

To enable the Device Utilization feature:

1. Install the driver package on the host with SRIOV enabled.
2. Update the physical function configuration file to set `RateLimitingEnabled=1` in the General section.
3. Set `ServicesEnabled` to `cy` or `sym` or `asym`.
4. Perform `qat_service shutdown` and `qat_service start`.

Note: This procedure also enables Rate Limiting (refer to Section 3.20).

3.21.1 Commands to Fetch Device Utilization

Start or Stop the device measurement: `./du_mgr ( start / stop ) <pf_addr>`

Query utilization for Physical function: `./du_mgr query <pf_addr> <service>`

Query utilization for Virtual function: `./du_mgr query_vf <pf_addr> <vf_addr> <service>`

Options:

`pf_addr` Physical address in bus:device.function(xx:xx.x) format

`vf_addr` Virtual address in bus:device.function(xx:xx.x) format

`service` Asym(=0) or Sym(=1) cryptographic services

3.21.2 Durations

Duration between start and stop commands should be between 5 to 10 seconds.

Duration of more than 10 seconds may give inconsistent query results.

Device utilization query and query_vf reports utilization between the last start and stop command.

For a given physical or virtual function, the device utilization reported would be in relation to the maximum device capacity.

3.21.3 Reference Algorithm

The Symmetric Crypto Algorithm for QAT 1.7 devices is AES128-CBC HMAC-SHA1 with Packet size 1024 bytes.

The Symmetric Crypto Algorithm for QAT 1.6 devices is AES128-CBC HMAC-SHA2-256.

The Asymmetric Crypto Algorithm for both systems is RSA with 2048 modulus size.

§

4 Acceleration Driver Configuration File

This chapter 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.

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

4.1 Configuration File Overview

There is a single configuration file for each Intel® QAT endpoint (and there may be multiple QAT endpoints on a single Intel® C62x Chipset).

Note: Depending on the model number, a device may also contain no 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 request default configuration.
- Interrupt coalescing configuration (optional).
- Statistics gathering configuration.

Additional details are included in Section 4.2, General Section.

Note: The concurrent request parameters include both transmit (Tx) and receive (Rx) requests.

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 Section 4.3, Logical Instances Section.

A sample configuration file is included in the package in the `quickassist/utilities/adf_ctl/conf_files` directory.

4.2 General Section

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

4.2.1 General Parameters

Table 10 describes the parameters that can be included in the General section.

Table 10. General Parameters

Parameter	Description	Default	Range
ServicesEnabled	Defines the service(s) available (cryptographic [cy], data compression [dc]).	cy;dc	cy, dc ***Note:*** Multiple values permitted, use ; as the delimiter. For exceptions, see Section 4.3.3.2, "Increasing the Maximum Number of Processes/Instances".
CyNumConcurrentSymRequests	Specifies the number of cryptographic concurrent symmetric requests for cryptographic instances in general.	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.	64	64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, or 65536
DcNumConcurrentRequests	Specifies the number of data compression concurrent requests for data compression instances in general.	512	64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, or 65536
StorageEnabled	When set to 1, the Storage Specific firmware image is loaded into the device. This is required to use advanced compression features like dc chaining. ***Note:*** If this parameter is set to 1, assymetric cryptography is not supported and the ServicesEnabled key can be only sym;dc or dc.	0	0 or 1
PkeServiceDisabled	Disables the Public Key Crypto service (asymmetric cryptography) and allows the full on-chip memory to be used for intermediate buffers when doing dynamic compression. ***Note:*** If this parameter is set to 1, the ServicesEnabled key can be only sym;dc, sym, or dc.	0	0 or 1
DcIntermediateBufferSizeInKB	Specifies the size in KB of each intermediate buffer in on-chip memory for dynamic compression.	64	32 or 64
AutoResetOnError	Automatically resets the device in case of fatal error or heartbeat failure.	0	0 or 1

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

For all the services enabled, `NumConcurrentRequests` must be set in the configuration file to one of the following values: 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768 and 65536.

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.

4.3 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 the Intel® QuickAssist API in Kernel address domain `[KERNEL_QAT]`
- 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 Section 5.2.4, "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 Section 5.2.4.2, "icp_sal_userStop" 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)

4.3.1 [KERNEL] Section

In the `[KERNEL]` section of the configuration file, information about the number and type of kernel instances can be defined.

Table 11 describes the parameters that determine the number of kernel instances for each service.

Note: The maximum number of cryptographic instances supported per QAT endpoint is 32; for exceptions, refer to Section 4.3.3.2, "Increasing the Maximum Number of Processes/Instances".

Note: The `NumberDcInstances` is ignored in this section and is set to 0.

Table 11. [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

4.3.2 [KERNEL_QAT] Section

The `[KERNEL_QAT]` section defines instances that can be used by the Intel® QuickAssist API in Kernel space domain.

This section is different from the `[KERNEL]` section. The `[KERNEL]` section in the configuration file defines instances to register the Intel® QuickAssist Acceleration with Linux Kernel Crypto Framework (LKCF) whilst the instances defined in the `[KERNEL_QAT]` section are exclusively targeted to be used with the Intel® QuickAssist API.

Table 12 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 QAT is 32 per endpoint; for exceptions, refer to Section 4.3.3.2, "Increasing the Maximum Number of Processes/Instances".

Table 12 describes the parameters that can be included in the `[GENERAL]` section.

Table 12. [KERNEL_QAT] Section Parameters

Parameter	Description	Default	Range
NumberCyInstances	Specifies the number of cryptographic instances. ***Note:*** Depends on the number of allocations to other services.	6	0 to 32
NumberDcInstances	Specifies the number of Data Compression instances. ***Note:*** Depends on the number of allocations to other services.	2	0 to 32

Notes:

1. `NumberCyInstances` depends on the number of allocations to other services
2. "Default" denotes the value in the configuration file when shipped.

4.3.3 User Process [xxxxx] Sections

There is one `[xxxxx]` section of the configuration file for each Intel® QAT endpoint to be configured.

Note: Check the SKU information for your specific device to determine how many QAT endpoints the device contains. There can be up to three endpoints per device.

In each `[xxxxx]` section of the configuration file, user space access to the QAT endpoint can be configured.

Table 13 shows the parameters in the configuration file that can be set for user process `[xxxxx]` sections.

Parameters for each user process instance can also be defined. The parameters that can be included for each specific user process instance are similar to those in Section 4.3, "Logical Instances Section".

Table 13. User Process [xxxxx] Sections 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. Refer to Section 5.2.4.1, "icp_sal_userStart" for more information. ***Caution:*** Resources are preallocated. If this parameter value is set too high, the driver fails to load.	1	For constraints, see Section 4.3.3.1, "Maximum Number of Process Calculations". For exceptions, see Section 4.3.3.2, "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 QAT endpoint.	0	0 (disabled, processes in this section can access multiple QAT endpoints) or 1 (enabled, processes in this section can only access this QAT endpoint). For additional information, see Section 4.5, "Configuring Multiple Processes on a System with Multiple 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 Section 4.3.3.2, "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

4.3.3.1 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, most significantly, the number of cryptography instances and/or data compression instances in the process section. The total number of processes, per service, created by the driver is given by the expression (e.g., for cryptography):

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

In Intel® QAT 1.7 devices, there are 16 ring banks per QAT endpoint and a maximum of two cryptography instances and two compression instances per bank. The maximum number of instances per device is 32 for cryptography and 32 for compression. For exceptions, refer to Section 4.3.3.2, "Increasing the Maximum Number of Processes/ Instances".

The following code example illustrates the maximum number of possible processes per device in polling mode:

```
NumProcesses = 32
NumCyInstances = 1
NumDcInstances = 1
```

4.3.3.2 Increasing the Maximum Number of Processes/Instances

Notes:

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

Under certain circumstances, it is possible to increase the number of processes supported by the software. In Intel® QAT 1.7 devices, there are 16 ring banks per QAT endpoint and a maximum of two cryptography instances and two compression instances per bank (or per VF) when the configuration file has `ServicesEnabled` equal to `cy;dc`. However, 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.

Note: Not all versions of the QAT software package support the ability to increase the number of processes.

Here are the variations:

- With `ServicesEnabled` equal to `sym`, only two rings are used for each instance, so eight instances can be used per bank (or per VF), or 128 instances per QAT endpoint. In this case, compression and asymmetric crypto services will not be available.
- With `ServicesEnabled` equal to `asym`, only two rings are used for each instance, so eight instances can be used per bank (or per VF), or 128 instances per QAT endpoint. In this case, compression and symmetric crypto services will not be available.
- With `ServicesEnabled` equal to `cy`, only four rings are used for each instance (two each for asymmetric and symmetric crypto), so four instances can be used per bank (or per VF), or 64 instances per QAT endpoint. In this case, compression services will not be available.
- With `ServicesEnabled` equal to `dc`, only two rings are used for each instance, so eight instances can be used per bank (or per VF), or 128 instances per QAT endpoint. In this case, asymmetric and symmetric crypto services will not be available.
- With `ServicesEnabled` equal to `dc;asym`, only four rings are used for each instance (two each for compression and asymmetric crypto), so four instances can be used per bank (or per VF), or 64 instances per QAT endpoint. In this case, symmetric crypto services will not be available.
- With `ServicesEnabled` equal to `dc;sym`, only four rings are used for each instance (two each for compression and symmetric crypto), so four instances can be used per bank (or per VF), or 64 instances per QAT endpoint. In this case, asymmetric crypto services will not be available.

4.3.3.3 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 Section 4.3.3.2)
3. Perform `qat_service shutdown` and `qat_service start`.
4. Update the virtual function configuration file to set `ServicesEnabled` (refer to Section 4.3.3.2)
5. Restart `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 Table 14. 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 Section 4.3.3.2. 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.

Table 14. Configuring Physical Functions and Virtual Functions

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

4.3.4 Cryptographic Logical Instance Parameters

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

Table 15. Cryptographic Logical Instance Parameters

Parameter	Description	Default	Range
CyXName	Specifies the name of cryptographic instance number X.	IPSec0 for KERNEL and KERNEL_QAT sections. SSL0 for user section	String (max. 64 characters)
CyXIsPolled	Specifies if cryptographic instance number x works in poll mode, interrrupt mode or epoll mode.	0 for kernel space instances 1 for user space instances	0 (interrupt mode) for instances in the KERNEL and KERNEL_QAT sections 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
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

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

4.3.5 Data Compression Logical Instance Parameters

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

Note: The maximum number of data compression instances supported is 64.

Table 16. 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, interrrupt mode or epoll mode.	0 for kernel space instances 1 for user space instances	0 (interrupt mode) for instances in the KERNEL and KERNEL_QAT sections 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: "Default" denotes the value in the configuration file when shipped.

4.3.6 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 = 1` in kernel space), the value of `CoreAffinity` is used to affinitize the interrupt handler to that core.

4.4 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. The format and structure of the configuration file is exactly the same for all devices. Consequently, the configuration file for QAT endpoint 0, (`c6xx_dev0.conf`, for the Intel® C62x Chipset; `c3xxx_dev0.conf`, for the Intel Atom® C3000 Processor Family SoC; `d15xx_dev0.conf`, for the Intel® Xeon® Processor D Family), can be cloned for use with other QAT endpoints.

All the configuration files are located in the `/etc` folder following the installation of the Intel® QuickAssist package.

Simply make a copy of the file and rename it by changing the `dev0` part of the file name. For example, for a second Intel® C62x Chipset QAT endpoint, change the file name to `c6xx_dev1.conf`; for a third QAT endpoint, change the file name to `c6xx_dev2.conf` and so on. Then, you can configure each QAT endpoint by editing the corresponding configuration file accordingly.

Note: 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'
```

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 QAT endpoint and its status. For example:

```
# service qat_service status

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 Section 3.6, 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 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.

4.5 Configuring Multiple Processes on a System with Multiple 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 QAT endpoint.

In this scenario, the user space section of the configuration files would look like the following.

For `/etc/c6xx_dev0.conf`:

```
[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
```

For `/etc/c6xx_dev1.conf`:

```
[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`, `SSL_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.

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

4.6 Sample Configuration File

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.

Note: The previous "v1" configuration file format is not supported.

§

5 Supported APIs

The supported APIs are described in two categories:

- Intel® QuickAssist Technology APIs
- Additional APIs

5.1 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 (refer to Table 2) for details.
- Data Compression - API definitions are located in: `$ICP_ROOT/quickassist/include/dc`. See the *Intel®* QuickAssist Technology Data Compression API Reference Manual (refer to Table 2) for details.

Base API definitions that are common to the API libraries are located in: `$ICP_ROOT/quickassist/include`. See also the *Intel®* QuickAssist Technology API Programmer's Guide (refer to Table 2) for guidelines and examples that demonstrate how to use the APIs.

5.1.1 Intel® QAT API Limitations

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

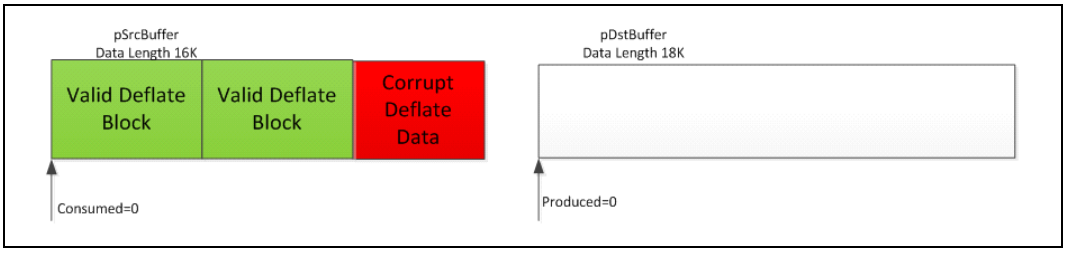
- For all services, the maximum size of a single perform request is 4 GB.
- For all services, data structures that contain data required by the QAT endpoint should be on a 64-byte-aligned address to maximize performance. This alignment helps minimize latency when transferring data from DRAM to an QAT endpoint integrated in the PCH device.
- For the key generation cryptographic API, the following limitations apply:
  - Secure Sockets Layer (SSL) key generation opdata:
    Maximum secret length is 512 bytes
    Maximum `userLabel` length is 136 bytes
    Maximum `generatedKeyLenInBytes` is 248
  - Transport Layer Security (TLS) key generation opdata:
    Secret length must be <128 bytes for TLS v1.0/1.1; <512 bytes for TLS v1.2
    `userLabel` length must be <256 bytes
    Maximum seed size is 64 bytes
    Maximum `generatedKeyLenInBytes` is 248 bytes
  - Mask Generation Function (MGF) opdata:
    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 (9*Total_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) * 5 bytes, depending on whether the stored headers are selected.

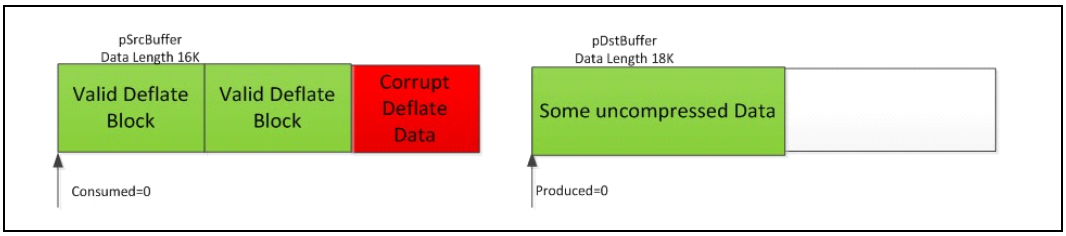
Note: Due to the need for a skid pad and the way the checksum is calculated in the stored block case to prevent compression overflow, an output buffer size of ceiling (9*Total_Input_Byte/8) + 55 bytes needs to be supplied (even though the stored block output size might be less).

- 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:



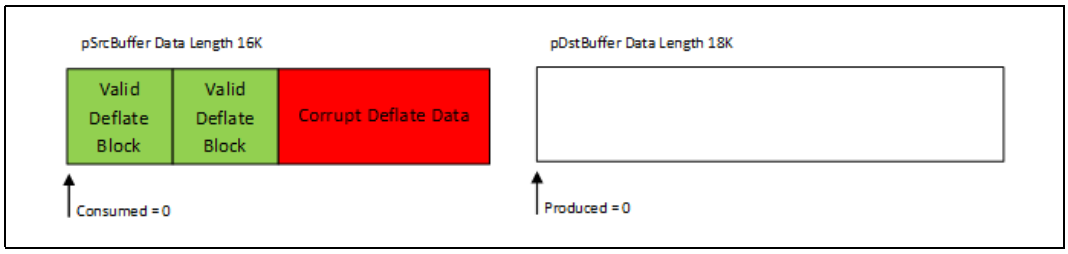
The result would be:



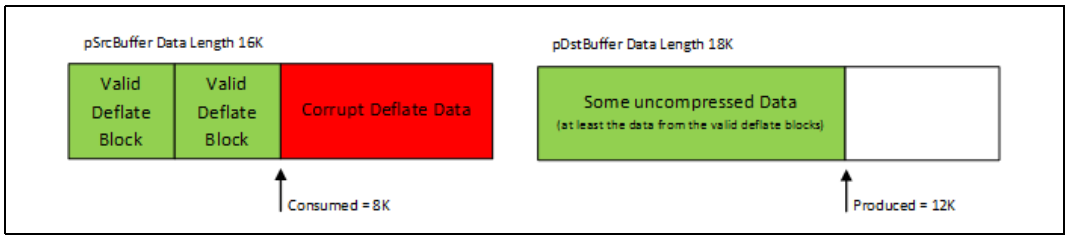
- Behavior when build flag ICP_DC_RETURN_COUNTERS_ON_ERROR is defined.
  In some specialized applications, when a decompression soft error occurs, the application has no way of correcting the source of the error and resubmitting the request. The session will need to be invalidated and terminated. In 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 decompression 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:



  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.

- For stateful decompression, the maximum output size is 4.29 GB (2^{32} bytes).

5.1.1.1 Resubmitting After Getting an Overflow Error

Table 17 describes the behavior of the Intel® QuickAssist Technology compression service when an overflow occurs during a compression or decompression operation.

It describes the expected behavior of an application when an overflow occurs.

Table 17. Compression/Decompression Overflow Behavior

	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	Possible - indicated in results consumed field	Possible - indicated in results produced field	-11	Overflow is considered as an exception
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

The Intel® QuickAssist releases enable the Compress and Verify feature by default for compression requests. The Compress and Verify feature implies that sessions can only be **Stateless** in the compression direction.

Overflow error in the Traditional API

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.

Overflow error in the Data Plane API

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.

Procedure for handling overflow errors

Resubmit the request with the following data:

- Use the same Source buffer.
- Allocate a bigger Destination buffer.
- Put the checksum from the previous successful request into the `cpaDcRqResults` struct.

Compression Overflow support in a Virtualized environment

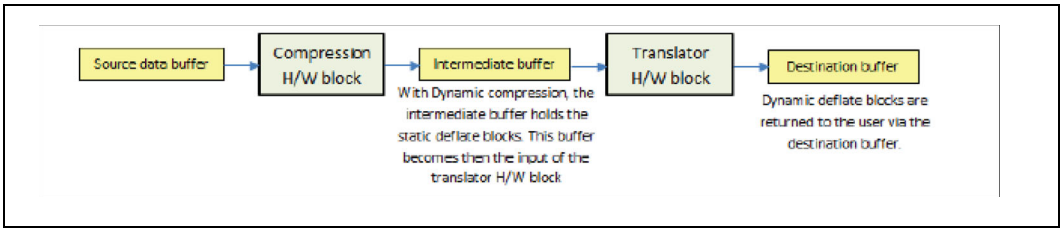
In a virtual environment, the guest does not download the firmware. Only the host downloads the firmware.

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.

5.1.1.2 Dynamic Compression for Data Compression Service

Dynamic compression involves feeding the data produced by the compression hardware block to the translator hardware block. Figure 5 shows the dynamic compression data path.

Figure 5. Dynamic Compression Data Path



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.

In the case of Stateful decompression requests, if the service returns an exception (e.g. overflow status in the results), it is recommended to examine the bytes consumed and returned in the `CpaDcRqResults` structure to verify if all the data in the source data buffer has been processed. Unprocessed data can be submitted in a subsequent request that uses the offset reported by the `consumed` field in the `CpaDcRqResults` structure.

5.1.1.3 Maximal Expansion with Auto Select Best Feature for Compression

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. In this mode, 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. To use the ASB feature, configure the `autoSelectBestHuffmanTree` enum during the session creation.

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

There are four possible settings available for the `autoSelectBestHuffmanTree` when creating a session. Based on the ASB settings described below, the produced data returned in the `CpaDcRqResults` structure will vary:

5.1.1.3.1 CPA_DC_ASB_DISABLED

ASB mode is disabled.

5.1.1.3.2 CPA_DC_ASB_STATIC_DYNAMIC

This setting is deprecated. Selecting `CPA_DC_ASB_STATIC DYNAMIC` has the same effect as selecting `CPA_DC_ASB_UCNCOMP_STATIC_DYNAMIC_WITH_STORED_HDRS` (refer to Section 5.1.1.3.3.)

5.1.1.3.3 CPA_DC_ASB_UNCOMP_STATIC_DYNAMIC_WITH_STORED_HDRS

Both a dynamic and a static compression operation are performed. However, if the produced data both for the dynamic and static operations return a greater value than the uncompressed source data and source block headers, the source data will be used as a stored block. With this ASB setting, a 5-byte stored block header is prepended to the stored block.

The worst-case produced data can be estimated to:

```
Produced data in bytes = Total input bytes + ceil (Total input
bytes / 65535) * 5
```

For example, for an input source size of 111261 bytes, the worst-case produced data will be:

```
Produced data = 111261 + ceil (111261 / 65535) * 5

   = 111261 + ceil (1.698) * 5

   = 111261 + 2 * 5

Produced data = 111271 bytes
```

5.1.1.3.4 CPA_DC_ASB_UNCOMP_STATIC_DYNAMIC_WITH_NO_HDRS

Note: The static test for this ASB setting has been deprecated.

With this ASB setting, if the produced data for the dynamic operation returns a greater value than the uncompressed source data, the uncompressed source data will be sent to the destination buffer though DMA transfer. This is the same behavior as with the ASB setting `CPA_DC_ASB_UNCOMP_STATIC_DYNAMIC_WITH_STORED_HDRS` except the stored block deflate headers are not prepended to the stored block. The produced data can be estimated via the following:

```
Produced data in bytes = Min(Dynamic, Uncompressed)
```

5.1.1.4 Maximal Expansion and Destination Buffer Size

For static compression operations, the worst-case possible expansion can be expressed as:

```
Max Static Produced data in bytes = ceil(9 * Total input bytes / 8)
+ 7
```

The memory requirement for the destination buffer is expressed by the following formula:

```
Destination buffer size in bytes = ceil(9 * Total input bytes / 8)
+ 55 bytes
```

The destination buffer size must take into account the worst-case possible maximal expansion + 55 bytes; e.g., for an input source size of 111261 bytes, the worst-case produced data will be:

```
Static Produced data = ceil(9 * 111261 / 8) + 7
   = ceil (125168.625) + 7
   = 125169 + 7
Worst case Static Produced data = 125176 bytes
Memory required for destination buffer = ceil(9 * 111261 / 8) + 55
   = ceil (125168.625) + 55
   = 125169 + 7
   = 125169 + 55
   = 125224 bytes to be allocated
```

Note: Regardless of the ASB settings, the memory must be allocated for the worst case. If an overflow occurs, either from static or dynamic compression, then the returned counters, status, and expected application behavior is as shown per Table 17.

5.1.2 Data Plane APIs Overview

The *Intel*® QuickAssist Technology Cryptographic API Reference Manual and the *Intel*® QuickAssist Technology Data Compression API Reference Manual (refer to Table 2) 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
```

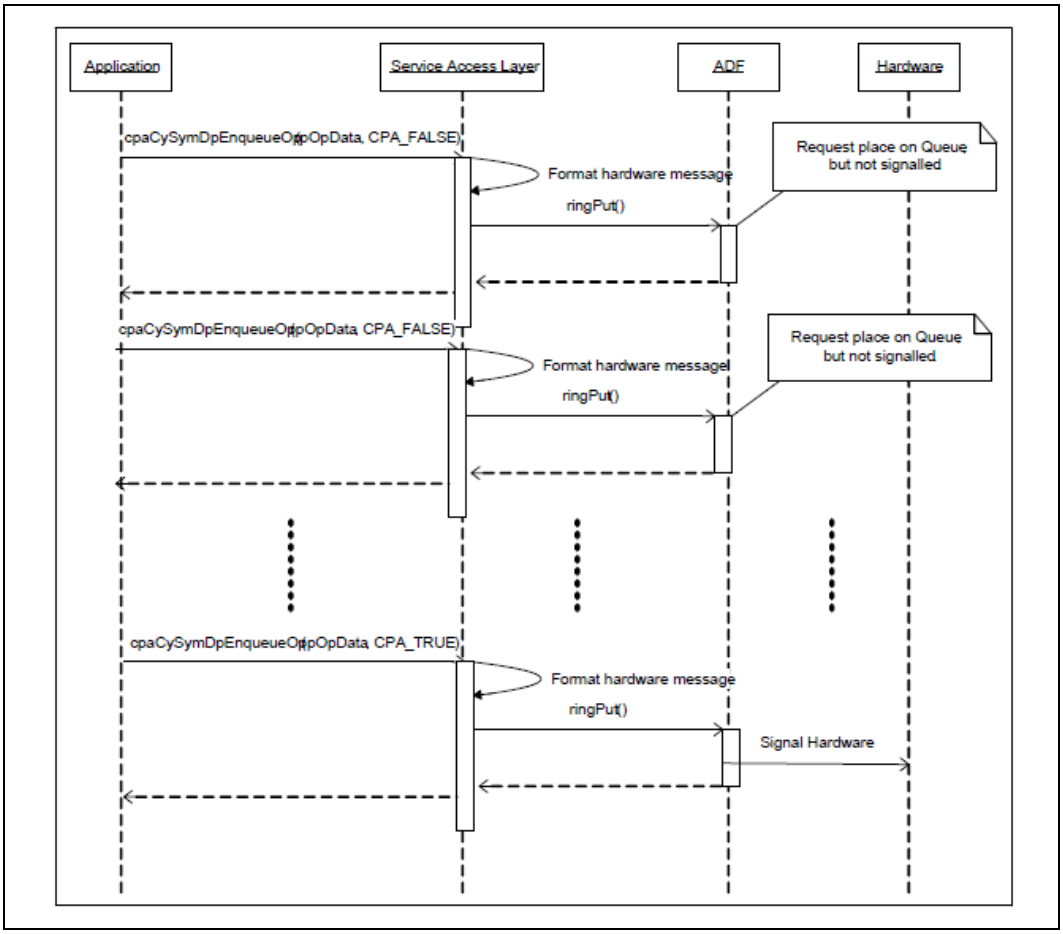
5.1.2.1 IA Cycle Count Reduction When Using Data Plane APIs

From an IA cycle count perspective, the Data Plane APIs are more performant than the traditional APIs (that is, for example, the symmetric cryptographic APIs defined in `$ICP_ROOT/quickassist/include/lac/cpa_cy_sym.h`). 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 (refer to Section 5.1.2.2, "Usage Constraints on the Data Plane APIs").

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, the `cpaCySymDpEnqueueOp()` or `cpaDcDpEnqueueOp()` function may be called with the `performOpNow` flag set to CPA_TRUE. This sends the requests to the QAT endpoint for processing. This sequence is shown in Figure 6.

Figure 6. Amortizing the Cost of an MMIO Across Multiple Requests



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 tuneable 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.

5.1.2.2 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 (refer to Section 5.2.3, "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 functions (`cpaCyGetNumInstances()`, `cpaCyGetInstances()`).
- 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.

5.1.2.3 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 (refer to Table 2). In addition to the Intel® QAT Data Plane APIs, there are a number of Data Plane Polling APIs that are described in Section 5.2.3, "Polling Functions".

5.1.3 Recovering from a Compress and Verify error

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 API.

The CnVnR feature is an enhancement of the existing 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 `CpaDcInstanceCapabilities` structure will be set to `CPA_TRUE` if the feature is supported.

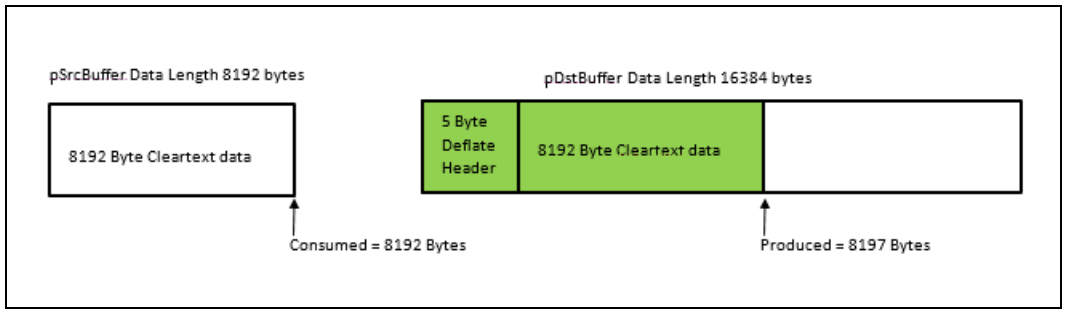
Table 18 provides details on the Intel® QuickAssist APIs supporting the CnVnR feature.

Table 18. API Support for Compress and Verify and Recover

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

When a CnV recovery takes place, the Intel® QuickAssist 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.

Figure 7. Maximum Stored Block Size



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*)

If a stored block needs to be created out of a cleartext payload size greater than 65,535 bytes, the Intel® QuickAssist solution creates one stored block of 65,535 bytes and `CpaDcRqResults.status` returns `CPA_DC_OVERFLOW`.

Note: If the application uses the Data Plane API, it is responsible for submitting request sizes smaller or equal to 65,530 bytes to avoid meeting the overflow error limit.

5.1.4 Counting Recovered Compression Errors

The Intel® QuickAssist 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`.

5.1.5 Compress and Verify Error log in Sysfs:

The implementation of the Compress and Verify and Recover 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:

```
cat /sys/kernel/debug/qat_dh895xcc_<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.

5.2 Additional APIs

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

Note: Not all additional APIs are supported with all versions of the software package.

These APIs are grouped into the following categories:

- Dynamic Instance Allocation Functions
- IOMMU Remapping Functions
- Polling Functions
- User Space Access Configuration Functions
- Version Information Function
- Acceleration Drivers Overview
- Compress and Verify (CnV) Related APIs

5.2.1 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 `*.conf` 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`.

The dynamic instance allocation functions include:

5.2.1.1 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

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

Return Value

The icp_sal_userCyGetAvailableNumDynInstances function returns one of the following codes:

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

5.2.1.2 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* pNumDcInstances);
```

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:

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

5.2.1.3 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:

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

5.2.1.4 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:

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

5.2.1.5 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 to free.

Return Value

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

Code Meaning

`CPA_STATUS_SUCCESS` Successfully freed the specified number of CY instances.

`CPA_STATUS_FAIL` Indicates a failure.

5.2.1.6 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_userDcInstancesAlloc` function returns one of the following codes:

Code Meaning

`CPA_STATUS_SUCCESS` Successfully freed the specified number of DC instances.

`CPA_STATUS_FAIL` Indicates a failure.

5.2.1.7 icp_sal_userCyGetAvailableNumDynInstancesByDevPkg

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

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:

Code Meaning

`CPA_STATUS_SUCCESS` Successfully retrieved the number of cryptographic instances available for dynamic allocation.

`CPA_STATUS_FAIL` Indicates a failure.

5.2.1.8 icp_sal_userDcGetAvailableNumDynInstancesByDevPkg

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

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:

Code Meaning

`CPA_STATUS_SUCCESS` Successfully retrieved the number of cryptographic instances available for dynamic allocation.

`CPA_STATUS_FAIL` Indicates a failure.

5.2.1.9 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:

Code Meaning

CPA_STATUS_SUCCESS Successfully allocated the specified number of CY instances.

CPA_STATUS_FAILIndicates a failure.

5.2.1.10 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:

Code Meaning

CPA_STATUS_SUCCESSSuccessfully allocated the specified number of DC instances.

CPA_STATUS_FAILIndicates a failure.

5.2.1.11 icp_sal_userCyGetAvailableNumDynInstancesByPkgAccel

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

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 `icp_sal_userCyGetAvailableNumDynInstancesByDevPkg`.

Return Value

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

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

5.2.1.12 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, CpaInstanceHandle *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 `icp_sal_userCyInstancesAllocByDevPkg`.

Return Value

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

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

5.2.2 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`.

The IOMMU remapping functions include:

- Section 5.2.2.1, "icp_sal_iommu_get_remap_size"
- Section 5.2.2.2, "icp_sal_iommu_map"
- Section 5.2.2.3, "icp_sal_iommu_unmap"

5.2.2.1 icp_sal_iommu_get_remap_size

Returns the `page_size` rounded for IOMMU remapping.

Syntax

```
size_ticp_sal_iommu_get_remap_size(size_t size);
```

Parameters

`size_t` The minimum required page size.

Return Value

The `icp_sal_iommu_get_remap_size` function returns the page_size rounded for IOMMU remapping.

5.2.2.2 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:

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

5.2.2.3 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 to be removed.

`size` Size of the remapped region.

Return Value

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

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

5.2.2.4 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 and 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 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) Domaine 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 Domaine 0 case, the APIs will update the hardware IOMMU tables.

5.2.3 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`.

The polling functions include:

- Section 5.2.3.1, "icp_sal_pollBank"
- Section 5.2.3.2, "icp_sal_pollAllBanks"
- Section 5.2.3.3, "icp_sal_CyPollInstance"
- Section 5.2.3.4, "icp_sal_DcPollInstance"
- Section 5.2.3.5, "icp_sal_CyPollDpInstance"
- Section 5.2.3.6, "icp_sal_DcPollDpInstance"

5.2.3.1 icp_sal_pollBank

Poll all rings on the given QAT endpoint on a given bank number to determine if any of the rings contain response messages from the 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 QAT endpoint. The valid range is 0 to the number of QAT endpoint devices in the system.

`bank_number` The number of the memory bank on the 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:

Code	Meaning
`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.

5.2.3.2 icp_sal_pollAllBanks

Poll all banks on the given QAT endpoint to determine if any of the rings contain response messages from the 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 QAT endpoint. The valid range is 0 to the number of 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:

Code	Meaning
`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.

5.2.3.3 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:

Code	Meaning
CPA_STATUS_SUCCESS	The function was successful.
CPA_STATUS_RETRY	There are no responses on the rings associated with the specified logical instance.

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

CPA_STATUS_FAIL	Indicates a failure.

5.2.3.4 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:

Code	Meaning
CPA_STATUS_SUCCESS	The function was successful.
CPA_STATUS_RETRY	There are no responses on the rings associated with the specified logical instance.

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

CPA_STATUS_FAIL	Indicates a failure.

5.2.3.5 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.

Syntax

Note: This function is a Data Plane API function and consequently the restrictions in Section 5.1.2.2, "Usage Constraints on the Data Plane APIs" apply.

```
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:

Code	Meaning
`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.

5.2.3.6 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.

Syntax

Note: This function is a Data Plane API function and consequently the restrictions in Section 5.1.2.2, "Usage Constraints on the Data Plane APIs" apply.

```
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:

Code	Meaning
`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.

5.2.4 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`.

The user space access configuration functions include:

- Section 5.2.4.1, "icp_sal_userStart"
- Section 5.2.4.2, "icp_sal_userStop"

5.2.4.1 icp_sal_userStart

Initializes user space access to a QAT endpoint and starts in the `pProcessName` 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 space 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 M*N internal sections are generated (each with instances on one device only) and M*N 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 Section 4.5, "Configuring Multiple Processes on a System with Multiple QAT Endpoints" for a detailed example.

Syntax

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

Parameters

`*pSectionName` The section name described in the simplified configuration file format.

`limitDevAccess` Deprecated/ignored.

Return Value

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

Code **Meaning**

`CPA_STATUS_SUCCESS` Successfully started user space access to the QAT endpoint as defined in the configuration file.

`CPA_STATUS_FAIL` Operation failed.

5.2.4.2 icp_sal_userStop

Closes user space access to the 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 QAT endpoint from a calling process is not possible. This function should be called once when the process is finished using the 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:

Code **Meaning**

`CPA_STATUS_SUCCESS` Successfully stopped user space access to the QAT endpoint.

`CPA_STATUS_FAIL` Operation failed.

5.2.5 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`.

There is only one version information function, that is, icp_sal_getDevVersionInfo.

5.2.5.1 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_userStartMultiProcess` (or `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_userStartMultiProcess` (or `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 Values

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

Code Meaning

`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.

5.2.6 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`.

5.2.6.1 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:

Code Meaning

`CPA_STATUS_SUCCESS`Successful operation.

`CPA_STATUS_FAIL`Indicates a failure.

5.2.7 Thread-Less APIs

These APIs can be used in the user space application.

The thread-less API functions include:

- Section 5.2.7.1, "icp_sal_poll_device_events"
- Section 5.2.7.2, "icp_sal_find_new_devices"

5.2.7.1 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:

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

5.2.7.2 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:

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

5.2.8 Compress and Verify (CnV) Related APIs

These APIs can be used in the user space application.

The CnV API functions include:

- Section 5.2.8.1, "icp_sal_dc_get_dc_error()"
- Section 5.2.8.2, "icp_sal_dc_simulate_error()"

5.2.8.1 icp_sal_dc_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

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

Return Value

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

5.2.8.2 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);
```

Followed by a call to:

`CpaDcCompressData()` or `CpaDcCompressData2()`.

To use this API, the driver must be configured and compiled with option `--enable-dc-error-simulation`.

Syntax

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

Parameters

`Cpa8U numErrors`: Number of simulated compression or decompression errors desired.

`Cpa8S dcError`: Desired error code to be returned by the compression or decompression API.

Return Value

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

Code	Meaning
`CPA_STATUS_SUCCESS`	Successful operation.

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

5.2.9 Heartbeat APIs

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

The Heartbeat API functions include:

- Section 5.2.9.1, "icp_sal_check_device()"
- Section 5.2.9.2, "icp_sal_check_all_devices()"
- Section 5.2.9.3, "icp_sal_heartbeat_simulate_failure()"

5.2.9.1 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:

Code Meaning

`CPA_STATUS_SUCCESS`Successful operation.

`CPA_STATUS_FAIL`Indicates a failure.

5.2.9.2 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:

Code Meaning

`CPA_STATUS_SUCCESS`Successful operation.

CPA_STATUS_FAIL	Indicates a failure.

5.2.9.3 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:

Code	Meaning
CPA_STATUS_SUCCESS	Successful operation.
CPA_STATUS_FAIL	Indicates a failure.

5.2.10 Device Polling APIs

5.2.10.1 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:

Code	Meaning
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`.

5.2.10.2 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 Values

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

Code Meaning

`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;
```

5.2.10.3 cpaDcInstanceSetNotificationCb

Cryptographic 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 Values

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

Code Meaning

`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;
```

§

6 Application Usage Guidelines

This chapter provides some usage guidelines and identifies some of the applications to which the platforms described in this manual are ideally suited.

6.1 Mapping Service Instances to Engines on the Intel® QAT Endpoint

A processor may be connected to one or more Intel® QAT endpoints. For example, an Intel Atom® C3000 Processor contains a single integrated QAT endpoint, while a single Intel® C620 Series Chipset contains up to three QAT endpoints.

Communication between software running on the processor and the QAT endpoint is via hardware-assisted rings. Rings are used in pairs; software writes requests onto a request ring, and reads responses back from a response ring. The QAT endpoint load balances requests from all rings of a given service type across all available hardware "engines" of the corresponding type.

A set of 16 ring banks provide the communication mechanism between a processor and the acceleration complex. Each ring bank contains 16 individual rings for communication.

Intel provides the software package that abstracts the communication between the host and the rings and presents the high-level Intel® QAT APIs.

6.1.1 Processor and Intel® QAT Endpoint Communication

An acceleration service uses different rings for request and response messages. Communication between the processor and QAT endpoint is achieved using the following operations:

- The processor uses a write (PUT) operation to place a request on the request ring.
- The QAT endpoint uses a read (GET) operation to retrieve the request from the request ring.
- Once the operation has been performed, the QAT endpoint uses a write (PUT) operation to put the response to the response ring.
- The processor uses a read (GET) operation to retrieve the response from the response ring.

6.1.2 Service Instances and Interaction with the Hardware

A ring bank supports two crypto instances and two compression instances. A service instance can be thought of as a channel between an QAT endpoint and a core/thread running on the processor, which uses the rings for communication. The rings are not exposed by an API, but are set up using configuration files (one for each QAT endpoint).

In general, a service instance uses a pair of rings, one for requests and one for responses. For cryptographic instances, separate request/response pairs are used.

6.1.3 Service Instance Configuration

The configuration of a service instance is done in the configuration file.

The following figure shows an example extract of the relevant section in the configuration file.

Figure 8. Service Instance Configuration

```
##########################################
# User Space Instances Section
##########################################
[proc0] (1)
NumberCyInstances = 1
NumberDcInstances = 0

# Crypto - user space instance #0
Cy0Name = "proc0_0" (2)
Cy0IsPolled = 1 (3)
Cy0CoreAffinity = 0 (4)
```

In the previous figure, the meaning of each numbered item is explained as follows:

1. Each named address domain (one domain for the kernel, any number of user space process domains) has its own service instances.
2. Specifies a name for the instance.
3. Specifies that the instance is using polling.
4. Specifies the core affinity for the instance.

6.1.4 Cryptographic Load Balancing Using Multiple Intel® QAT Instances

The application is responsible for load balancing/spreading requests across Intel® QAT endpoints. Load balancing across the engines computing instances within the QAT endpoint is performed by hardware.

In general, the device can be fully utilized from a single instance/ring pair. The main reasons for using multiple instances/ring pairs are:

- Separate software processes each benefit by having their own ring pair to enable the rings to be mapped into the address space of that process
- Separate threads within a process, possibly on different cores, avoid contention
- If using interrupts, they can be affinitized from different instances/ring pairs to different cores

6.2 Cryptography Applications

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

- Virtual Private Networks (VPNs, both IPsec and SSL). Both symmetric and public key cryptography can be offloaded for bulk transfer and key exchange (IKE, SSL handshakes and so on). Refer to Section 6.2.1, "IPsec and SSL VPNs" for more information.
- Encrypted Storage. See Section 6.2.2, "Encrypted Storage" for more information.
- Web Proxy Appliances. See Section 6.2.3, "Web Proxy Appliances".

6.2.1 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 PCH provides cryptographic acceleration that can offload this computation from the CPU, thereby freeing up CPU cycles to perform other networking, security or other value-add applications.

The 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 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 security to storage appliances makes offload solutions an attractive value proposition.

Several complimentary standards for the security of data-at-rest exist, which when combined with traditional network security protocols, such as IPsec or SSL/TLS, provide an end-to-end secure 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 QAT endpoints are ideally suited for secure long-term 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 security for authenticating the host to trusted peripherals implementing the TCG storage standards.

6.2.3 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 a number of 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.

6.3 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.

6.3.1 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, security or other value-add operations.

6.3.2 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 for data security.

§