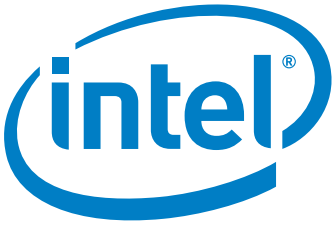


# Intel® QuickAssist Technology Software for Linux*

## Programmer's Guide - Hardware Version 1.7

***August 2017***

Document Number: 336210-001

# Revision History

| Date | Revision | Description |
|---|---|---|
| August 2017 | 001 | Initial public release. |

# Contents

## Figures



## Tables

# 1.0 Introduction

This programmer's guide provides information on the architecture of the software and usage guidelines. Information on the use of Intel® QuickAssist Technology APIs, which provide the interface to the acceleration services (cryptographic and data compression), is documented in the related Intel® QuickAssist Technology software library documentation (see Product Documentation on page 8).

## 1.1 Terminology

In this document, for convenience:

- Software package is used as a generic term for the Intel® QuickAssist Technology software package for Hardware Version 1.7.
- Acceleration driver is used as a generic term for the software that allows the Intel® QuickAssist Technology Software Library APIs to access the Intel® QuickAssist Technology endpoint(s).

**Table 1. Terminology**

| Term | Description |
|---|---|
| ADF | Acceleration Driver Framework |
| ASIC | Application Specific Integrated Circuit |
| BnP | Batch and Pack Compression |
| CnV | Compress and Verify (formerly MCA: Machine Check Architecture) |
| 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) |
| GPL | General Public License |
| MGF | Mask Generation Function |
| MSI | Message Signaled Interrupts |
| PCH | Platform Controller Hub. In this manual, a Platform Controller Hub device includes standard interfaces and Intel® QuickAssist Technology endpoint and I/O interfaces. |
| QAT | Intel® QuickAssist Technology |
| SAL | Service Access Layer |
| SATA | Serial Advanced Technology Attachment |
| SGL | Scatter Gather List |
| SoC | System-on-a-Chip |

**Table 1. Terminology**

| Term | Description |
|---|---|
| SPI | Serial Peripheral Interconnect |
| SR-IOV | Single Root I/O Virtualization |
| SSL | Secure Sockets Layer |
| TLS | Transport Layer Security |
| VPN | Virtual Private 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.

## 1.3 Product Documentation

Documentation supporting the software package includes:

- *Intel® QuickAssist Technology Software for Linux* Release Notes (Hardware Version 1.7)*
- *Intel® QuickAssist Technology Software for Linux* Getting Started Guide (Hardware Version 1.7)*
- *Intel® QuickAssist Technology Software for Linux* Programmer's Guide (Hardware Version 1.7)* (this document)

Related Intel® QuickAssist Technology software library documentation includes:

- *Intel® QuickAssist Technology API Programmer's Guide*
- *Intel® QuickAssist Technology Cryptographic API Reference Manual*
- *Intel® QuickAssist Technology Data Compression API Reference Manual*

Other related documentation:

- *Using Intel® Virtualization Technology (Intel® VT) with Intel® QuickAssist Technology Application Note*
- *Intel®Xeon®Processor (storage) - External Design Specification (EDS) Addendum* (document number: 503997)

## 1.4 Typographical Conventions

The following conventions are used in this manual:

- `Courier font`- file names, path names, 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 and buttons

# 2.0 Software Overview

In addition to the hardware mentioned in Hardware/Software Overview on page 10, 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 the Intel® QuickAssist Technology endpoints in the PCH.

## 2.1 Intel® Communications Chipset 8925 to 8955 Series Compatibility

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



# 3.0 Acceleration Drivers Overview

The Intel® C62x Chipset (a Platform Controller Hub, or PCH), the Intel Atom® C3000 Processor Product Family (a System-on-a-Chip, or SoC), and the Intel® Xeon® Processor D Family SoC support Intel® QuickAssist Technology, which accelerates two services: cryptography (both symmetric and public key) and data compression. The Intel® QuickAssist Technology endpoints are exposed as PCI devices. Applications running in user space typically access these services via the Intel® QuickAssist Technology APIs. Applications that run in the Linux* kernel can also access some services via the Linux* kernel cryptographic framework, also known as the LKCF API.

## 3.1 Hardware/Software Overview

Because the hardware is accessed via the Intel® QuickAssist Technology 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, in order to support customers with different acceleration performance requirements, the Intel® C62x Chipset will come in different SKUs, and also supports two different "fabric configurations". Figure 1 and Figure 2 show two possible configurations for the acceleration endpoints in one Intel® C62x Chipset die.

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

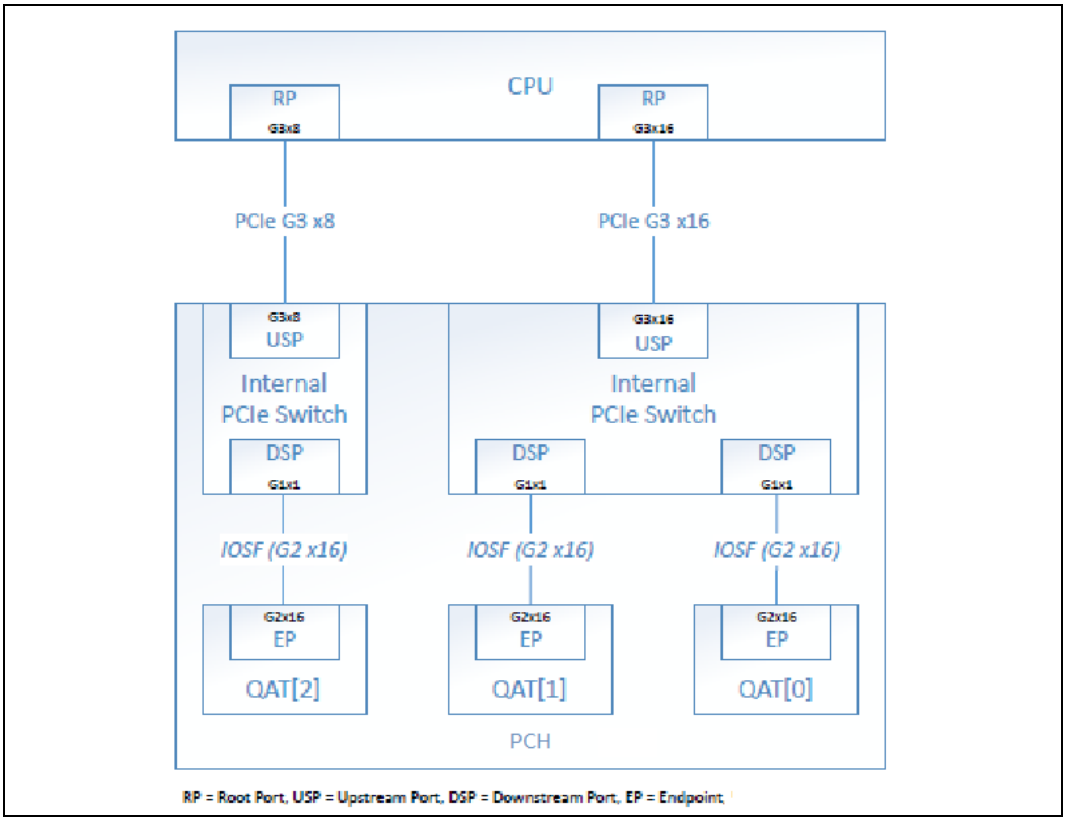
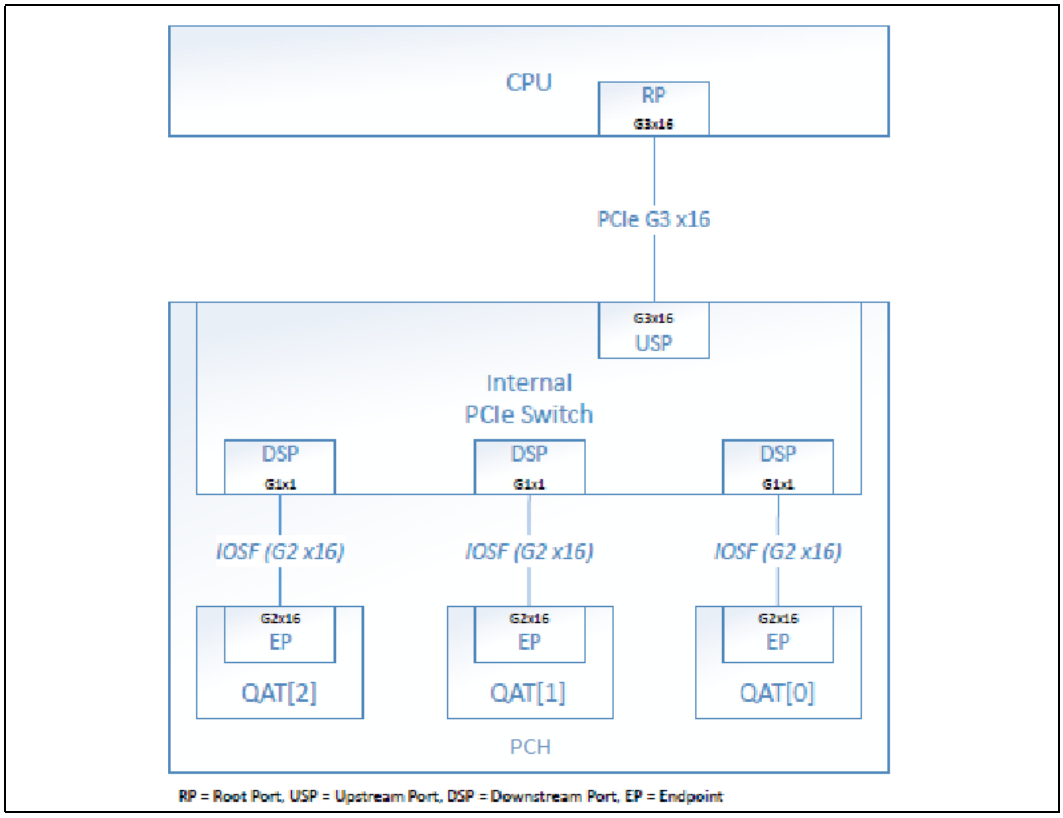


**Figure 2. Intel® C62x Chipset (PCH) Acceleration Endpoint Configuration 2**




For a given platform, the specific internal connections and number of 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 Cy (cryptography) and Dc (data compression) instances declared in the configuration files. See Acceleration Driver Configuration File on page 13 for more information.

Each 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® QuickAssist Technology 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).

*Note:* The Intel® QuickAssist Technology API is not exposed in the Linux* kernel. Kernel applications wishing to use QAT services must access them through the Linux* Kernel Cryptographic Framework. This will be extended over time to provide asynchronous access to all QAT services.

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 QAT endpoint in the system. If SRIOV is enabled, a separate configuration file is used for each virtual function, if applicable. The configuration file format is described in Configuration File Overview on page 23.

## 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 the 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 QAT1.7 driver can be used to interface with QAT1.7 devices and QAT1.6 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® QuickAssist Technology 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 (see Table 2).

**Table 2. Services**

| | | |
|---|---|---|
| Cryptographic service | `cpaCySetAddressTranslation` | See the *Intel® QuickAssist Technology Cryptographic API Reference Manual* for details. |
| Data Compression service | `cpaDcSetAddressTranslation` | See the *Intel® QuickAssist Technology Data Compression API Reference Manual* 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® QuickAssist Technology Data Plane API, buffers are passed to the Intel® QuickAssist Technology 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® QuickAssist Technology rings, the service access layer needs to 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 needs to call the icp_sal_userStart function 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 needs to call the function icp_sal_userStop.

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* 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 QAT endpoint or all 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 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 QAT endpoints, you can start, stop or restart each individual device by passing the 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 QAT endpoints and unload driver modules from the kernel. This contrasts with the *stop* qualifier which brings down one or more QAT endpoints, but does not unload kernel modules, so other endpoints can still run.

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

Limited debug information for the driver and configuration is available with the entries `/sys/kernel/debug/qat_*`. This includes the `fw_counters` entry, which can be used to verify that a specified QAT endpoint is being used. Additional functionality and information will be provided with future releases.

## 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 two traditional Intel® QuickAssist Technology Compression APIs, `cpaDcCompressData()` and `cpaDcDecompressData()`, that send requests to the compression hardware can return the error codes shown in the following table.

**Table 3. 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; for dynamic compression, corrupted intermediate data | Discard output; resubmit affected request or abort session. |
| -2 | CPA_DC_BAD_STORED_BLOCK_LEN | Stored block length did not match one's complement; invalid input stream detected | Discard output; resubmit affected request or abort session. |

## Table 3. Intel® QuickAssist Technology Compression API Errors

| 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; for dynamic compression, corrupted intermediate data | Discard output; resubmit affected request or abort session. |
| -4 | CPA_DC_INCOMPLETE _CODE_LENS | Code length codes incomplete; invalid input stream detected; for dynamic compression, corrupted intermediate data | Discard output; resubmit affected request or abort session. |
| -5 | CPA_DC_REPEATED_LEN S | Repeated lengths with no first length; invalid input stream detected; for dynamic compression, corrupted intermediate data | Discard output; resubmit affected request or abort session. |
| -6 | CPA_DC_MORE_REPEAT | Repeat more than specified lengths; invalid input stream detected; for dynamic compression, corrupted intermediate data | Discard output; resubmit affected request or abort session. |
| -7 | CPA_DC_BAD_LITLEN _CODES | Invalid literal/length code lengths; invalid input stream detected; for dynamic compression, corrupted intermediate data | Discard output; resubmit affected request or abort session. |
| -8 | CPA_DC_BAD_DIST _CODES | Invalid distance code lengths; invalid input stream detected; for dynamic compression, corrupted intermediate data | Discard output; resubmit affected request or abort session. |
| -9 | CPA_DC_INVALID_CODE | Invalid literal/length or distance code in fixed or dynamic block; invalid input stream detected; for dynamic compression, corrupted intermediate data | Discard output; resubmit affected request or abort session. |
| -10 | CPA_DC_INVALID_DIST | Distance is too far back in fixed or dynamic block; invalid input stream detected; for dynamic compression, corrupted intermediate data | Discard output; resubmit affected request or abort session. |
| -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. |
| -12 | CPA_DC_SOFTERR | Other non-fatal detected. | Discard output; resubmit affected request or abort session. |
| -13 | CPA_DC_FATALERR | Fatal error detected. | Discard output; restart or reset session. |
| -14 | CPA_DC_MAX _RESUBITE RR | On an error being detected, the firmware attempted to correct and resubmitted the request, however, the maximum resubmit value was exceeded. | Discard output; resubmit affected request or abort session. |
| -15 | CPA_DC_INCOMPLETE _FILE_ERR | The input file is incomplete. Note this is an indication that the request was submitted with a CPA_DC_FLUSH_FINAL, however, a BFINAL bit was not found in the request. | Continue with the session, if the file is not completed. Restart or reset the session if the following request is not related with the previous one. |
| -16 | CPA_DC_WDOG_TIMER _ ERR | The request was not completed as a watchdog timer hardware event occurred. | Discard output; resubmit affected request or abort session. |

**Table 3. Intel® QuickAssist Technology Compression API Errors**

| Error Code | Error Type | Description | Suggested Corrective Action(s) |
|---|---|---|---|
| -17 | CPA_DC_EP_HARDWARE _ERR | Request was not completed as an end point hardware error occurred (for example, a parity error). | Discard output; resubmit affected request or abort session. |
| -18 | CPA_DC_MCADECOMPERR | CnV (formerly known as MCA) decompress check error detected. | Discard output; resubmit affected request or abort session. |
| -19 | CPA_DC_EMPTY_DYM_BLK | Decompression request contained an empty dynamic stored block (not supported). | In a stateless session abort the session. In a stateful session decode the empty dynamic store block and continue. |

Except for the errors `CPA_DC_OK`, `CPA_DC_OVERFLOW` and `CPA_DC_FATALERR`, the rest of the error codes can be considered as invalid input stream errors.

## 3.9 Stateful Compression Level Details

Throughput and compression ratio for stateful 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. The context size is 32 KB for level 1, and 16 KB otherwise.

*Note:* For Intel® Communications Chipset 8925 to 8955 Series, the context size is 32kB for level 1, and 8kB otherwise.

## 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

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

**Table 4. 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. |

**Table 4. Acceleration Driver Return Codes**

| Return Type | Return Code | Description |
|---|---|---|
| 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. |

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

**Table 5. 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. |
| -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

The Batch and Pack (BnP) compression feature allows grouping multiple compression jobs into one request. The output data will represent each job packed into a single destination buffer. The BnP feature also outputs the metadata that the application knows where each job starts and finishes in the destination buffer.

The metadata output by the BnP feature are represented by:

- The status of each job
- The consumed value in bytes for each job in the batch
- The produced value in bytes for each job in the batch

*Notes:*

1. If the application configures the batch request with skip values either in the source buffers or in the destination buffer, the produced and consumed values returned for each job in the batch will include the length of these skip values. However, the checksum being reported does not include the skip values.
2. In the case of an overflow condition and when the application configures the skip region at the end of each job in the destination buffer, the skip regions will not be included in the destination buffer. This is implemented as such because the entire skip region cannot be guarantee to be available.
3. There are 4 modes available to define the operation of the skip regions in the input and output buffers. These are:
   `CPA_DC_SKIP_DISABLED`
   `CPA_DC_SKIP_AT_START`
   `CPA_DC_SKIP_AT_END`
   `CPA_DC_SKIP_STRIDE`
4. The mode `CPA_DC_SKIP_STRIDE` is not currently implemented.
5. Only static compression implementation of BnP is currently supported.
6. BnP compression requires storage-specific firmware; specify `StorageEnabled=1` in the configuration file.

The BnP functionality is implemented in the following Data Compression APIs:

```
cpaDcBnpBufferListGetMetaSize()

cpaDcBPCompressData()
```

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

## 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.

*Notes:*

1. To enable the CnV functionality, the application must call the above API with the `cnvDecompressCheck` set to `CPA_TRUE` in the CpaDcOpData structure declared in the API.
2. The API described above takes the CpaDcOpData structure as one of its parameters.
3. CnV is supported for static and dynamic stateless compression only. The `cpaDcDecompressData2()` API call is not supported for `cnvDecompressCheck` set to `CPA_TRUE`.
4. There is no support for the Data Plane API with the Compress and Verify feature.
5. The CnV functionality requires storage-specific firmware; specify `StorageEnabled=1` in the configuration file.
6. CnV is not supported with the Batch and Pack (BnP) compression feature.

The CnV functionality is implemented in the Data Compression API, `cpaDcCompressData2()`, for the compression path only.

This API is declared and documented in the API file `cpa_dc.h`.

## 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 hugepages 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
   ```

6. Change the amount of max locked memory for the user name that is include 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
   ```

The same basic steps can be followed to enable non-root access for customer applications accessing the acceleration software. Note that 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 Hugepages with the Included Memory Driver

The included User space DMAable Memory driver (`usdm_drv.ko`) supports hugepages. The use of hugepages provides benefits, but it also requires additional configuration. Use of this capability assumes that a sufficient number of hugepages 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 NUMA type memory that the USDM 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 in 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 that the use of hugepages may not be supported for all use cases. For instance, depending on the driver version, some limitations may exist for IOMMU.

# 4.0 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 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.

- **General** - Includes parameters that allow the user to specify:
  - — Which services are enabled.
  - — The configuration file format.
  - — Concurrent request default configuration.
  - — Interrupt coalescing configuration (optional).
  - — Statistics gathering configuration.

  Additional details are included in General Section on page 23.

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

- **Logical Instances** - one or more sections that include parameters that allow the user to set:
  - — The number of cryptography or data compression instances being managed.
  - — For each instance, the name of the instance, whether or not polling is enabled, and the core to which an instance is affinitized.

  Additional details are included in Logical Instances Section on page 24.

A sample configuration file is included in Sample Configuration File on page 32 and 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

The following table describes the parameters that can be included in the General section.

**Table 6. General Parameters**

| Parameter | Description | Default | Range |
|---|---|---|---|
| ServicesEnabled | Defines the service(s) available (cryptographic [cyX], data compression [dc]). | cy;dc | cy, dc<br>***Note:*** Multiple values permitted, use; as the delimiter.<br>For exceptions, see Section 4.3.2.2, "Increasing the Maximum Number of Processes/ Instances" on page 27. |
| CyNumConcurrentSymRequests | Specifies the number of cryptographic concurrent symmetric requests for cryptographic instances in general.<br>***Note:*** This parameter value can be overridden for a particular cryptographic instance if necessary. | 512 | 64, 128, 256, 512, 1024, 2048 or 4096 |
| CyNumConcurrentAsymRequests | Specifies the number of cryptographic concurrent asymmetric requests for cryptographic instances in general.<br>***Note:*** This parameter value can be overridden for a particular cryptographic instance if necessary. | 64 | 64, 128, 256, 512, 1024, 2048 or 4096 |
| DcNumConcurrentRequests | Specifies the number of data compression concurrent requests for data compression instances in general.<br>***Note:*** This parameter value can be overridden for a particular data compression instance if necessary. | 512 | 64, 128, 256, 512, 1024, 2048 or 4096 |
| ***Note:*** "Default" denotes the value in the configuration file when shipped or the value used if not specified in the configuration file. | | | |

## 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]`
- For user process address domains: `[xxxxx]`, where `xxxxx` may be any ASCII value that uniquely identifies the user mode process.

To allow a driver to correctly configure the logical instances associated with a user process, 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. See User Space Access Configuration Functions on page 57 for more information.

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. See Section 5.2.4.2, "icp_sal_userStop" on page 58 for more information.

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

- The name of the logical instance
- 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.

The following table 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, please see Configuration File Version 2 Differences on page 85.

**Table 7. [KERNEL] Section Parameters**

| Parameter | Description | Default | Range |
|---|---|---|---|
| NumberCyInstances | Specifies the number of cryptographic instances.<br>***Note:*** Depends on the number of allocations to other services. | 0 | 0 to 32 |
| ***Note:*** "Default" denotes the value in the configuration file when shipped. | | | |

### 4.3.2 User Process [xxxxx] Sections

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

*Note:* For the Intel® C62x Chipset, there are multiple (up to 3) QAT endpoints in a single device. For the Intel Atom® C3000 Processor Product Family, there is a single endpoint per device, and for the Intel® Xeon® Processor D Family [TBD]

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

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

**Table 8. 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.<br>The maximum number of processes that can call icp_sal_userStart and be active at any one time. See icp_sal_userStop on page 58 for more information.<br>***Caution:*** Resources are preallocated. If this parameter value is set too high, the driver fails to load. | 1 | For constraints, see Maximum Number of Process Calculations on page 26.<br>For exceptions, see Section 4.3.2.2, "Increasing the Maximum Number of Processes/ Instances" on page 27. |
| 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" on page 29. |
| NumberCyInstances | Specifies the number of cryptographic instances.<br>***Note:*** Depends on the number of allocations to other services. | 6 | 0 to 32. For exceptions, see Section 4.3.2.2, "Increasing the Maximum Number of Processes/ Instances" on page 27. |
| NumberDcInstances | Specifies the number of data compression instances.<br>***Note:*** Depends on the number of allocations to other services. | 2 | 0 to 32 |

***Notes:***
1. "Default" denotes the value in the configuration file when shipped.
2. The order of `NumProcesses` and `LimitDevAccess` parameters is important. The `NumProcess` parameter must appear **before** the `LimitDevAccess` parameter in the section.

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 Logical Instances Section on page 24.

### 4.3.2.1 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 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 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. This limits the maximum number of instances per device to 32 for cryptography and 32 for compression. For exceptions, see Section 4.3.2.2, "Increasing the Maximum Number of Processes/Instances" on page 27.

The following are examples that illustrate the maximum number of processes possible with a device:

- All processes/instances in polling mode:

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

### 4.3.2.2 Increasing the Maximum Number of Processes/Instances

Under certain circumstances, it is possible to increase the number of processes supported by the software. In 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 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, with 1 or 2 out of the three options, which will each require two rings per instance: "asym", "sym", and "dc".

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

Here are some examples:

- With "ServicesEnabled" equal to "asym", only two rings are used for each instance, so eight instances can be used per bank, or 128 instances per QAT endpoint. In this case, compression and symmetric crypto services will not be available.
- With "ServicesEnabled" equal to "asym;sym", only four rings are used for each instance (2 each for asymmetric and symmetric crypto), so four instances can be used per bank, 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 128 instances per QAT endpoint. In this case, asymmetric and symmetric crypto services will not be available.

Note that other combinations are possible.

*Note:* This section does not apply to the QAT virtual functions (VFs)

*Note:* This section only applies when the instances make use of polled mode.

## 4.4 Configuring Multiple Intel® QuickAssist Technology Endpoints in a System

A platform may include more than one 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.

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.

If a configuration file does not exist for an QAT endpoint, that QAT endpoint will not start at all and an error is displayed indicating that a configuration file was not found.

To determine the number of 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 user can also use the `qat_service` to start, stop, restart and shutdown each device separately or all QAT endpoints together. See Managing Intel® QuickAssist Technology Endpoints Using qat_service on page 14 for more information.

Some important configuration file information when using multiple 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 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 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.
- When using multiple configuration files, the `LimitDevAccess` parameter for a process must be enabled or disabled in all configuration files. The driver may not find the correct entries in the configuration file if the `LimitDevAccess` option is enabled in one configuration file and disabled in another.

## 4.5 Configuring Multiple Processes on a System with Multiple QAT Endpoints

As an example, consider a system with two QAT endpoints where it is necessary to configure two user space sections. One section is identified as `SSL` and the other is identified as `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 are limited to 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 is not limited to 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. Notice that 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

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

The following sample configuration file is provided in the software package.

```
################################################################
# This file is provided under a dual BSD/GPLv2 license.  When using or
#   redistributing this file, you may do so under either license.
#
#   GPL LICENSE SUMMARY
#
#   Copyright(c) 2007-2016 Intel Corporation. All rights reserved.
#
#   This program is free software; you can redistribute it and/or modify
#   it under the terms of version 2 of the GNU General Public License as
#   published by the Free Software Foundation.
#
#   This program is distributed in the hope that it will be useful, but
#   WITHOUT ANY WARRANTY; without even the implied warranty of
#   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
#   General Public License for more details.
#
#   You should have received a copy of the GNU General Public License
#   along with this program; if not, write to the Free Software
#   Foundation, Inc., 51 Franklin St - Fifth Floor, Boston, MA 02110-1301 USA.
#   The full GNU General Public License is included in this distribution
#   in the file called LICENSE.GPL.
#
#   Contact Information:
#   Intel Corporation
#
#   BSD LICENSE
#
#   Copyright(c) 2007-2016 Intel Corporation. All rights reserved.
#   All rights reserved.
#
#   Redistribution and use in source and binary forms, with or without
#   modification, are permitted provided that the following conditions
#   are met:
#
#     * Redistributions of source code must retain the above copyright
#       notice, this list of conditions and the following disclaimer.
#     * Redistributions in binary form must reproduce the above copyright
#       notice, this list of conditions and the following disclaimer in
#       the documentation and/or other materials provided with the
#       distribution.
#     * Neither the name of Intel Corporation nor the names of its
#       contributors may be used to endorse or promote products derived
#       from this software without specific prior written permission.
#
#   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
#   "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
#   LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
#   A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
#   OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
#   SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
#   LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
#   DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
#   THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
#   (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
#   OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#
#
#  version: QAT1.7.Upstream.L.1.0.3-22
################################################################
[GENERAL]
```

```
ServicesEnabled = cy;dc

ConfigVersion = 2

#Default values for number of concurrent requests*/
CyNumConcurrentSymRequests = 512
CyNumConcurrentAsymRequests = 64

#Statistics, valid values: 1,0
statsGeneral = 1
statsDh = 1
statsDrbg = 1
statsDsa = 1
statsEcc = 1
statsKeyGen = 1
statsDc = 1
statsLn = 1
statsPrime = 1
statsRsa = 1
statsSym = 1
KptEnabled = 0

# This flag is to enable SSF features (CNV and BnP)
StorageEnabled = 0

# Disable public key crypto and prime number
# services by specifying a value of 1 (default is 0)
PkeServiceDisabled = 0

# Specify size of intermediate buffers for which to
# allocate on-chip buffers. Legal values are 32 and
# 64 (default is 64). Specify 32 to optimize for
# compressing buffers <=32KB in size.
DcIntermediateBufferSizeInKB = 64

############################################
# Kernel Instances Section
############################################
[KERNEL]
NumberCyInstances = 1
NumberDcInstances = 1

# Crypto - Kernel instance #0
Cy0Name = "IPSec0"
Cy0IsPolled = 0
Cy0CoreAffinity = 0

# Data Compression - Kernel instance #0
Dc0Name = "IPComp0"
Dc0IsPolled = 0
Dc0CoreAffinity = 0

############################################
# User Process Instance Section
############################################
[SSL]
NumberCyInstances = 6
NumberDcInstances = 2
NumProcesses = 1
LimitDevAccess = 0

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



```
# Crypto - User instance #1
Cy1Name = "SSL1"
Cy1IsPolled = 1
# List of core affinities
Cy1CoreAffinity = 2

# Crypto - User instance #2
Cy2Name = "SSL2"
Cy2IsPolled = 1
# List of core affinities
Cy2CoreAffinity = 3

# Crypto - User instance #3
Cy3Name = "SSL3"
Cy3IsPolled = 1
# List of core affinities
Cy3CoreAffinity = 4

# Crypto - User instance #4
Cy4Name = "SSL4"
Cy4IsPolled = 1
# List of core affinities
Cy4CoreAffinity = 5

# Crypto - User instance #5
Cy5Name = "SSL5"
Cy5IsPolled = 1
# List of core affinities
Cy5CoreAffinity = 6


# Data Compression - User instance #0
Dc0Name = "Dc0"
Dc0IsPolled = 1
# List of core affinities
Dc0CoreAffinity = 1

# Data Compression - User instance #1
Dc1Name = "Dc1"
Dc1IsPolled = 1
# List of core affinities
Dc1CoreAffinity = 2
```

# 5.0 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 supports the following Intel® QuickAssist Technology API libraries:

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

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* for guidelines and examples that demonstrate how to use the APIs.

### 5.1.1 Intel® QuickAssist Technology API Limitations

The following limitations apply when using the Intel®QuickAssist Technology 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

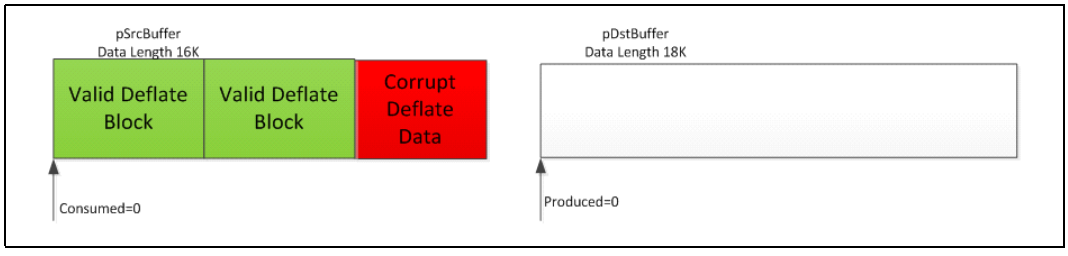
Maximum maskLenInBytes is 65528

- 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, only one outstanding compression request per stateful session is allowed.
- 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.

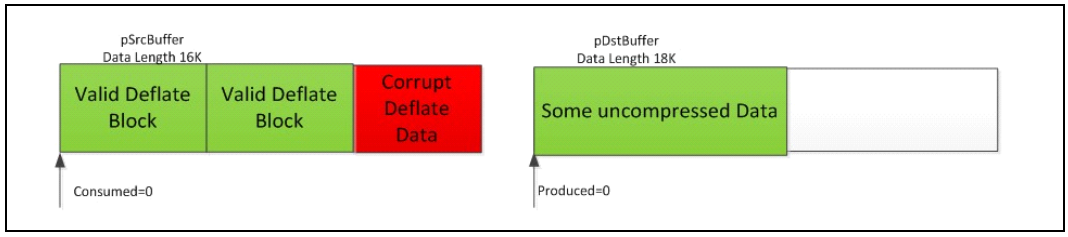
*Note:* Due to the need for a skid pad and the way the checksum is calculated in the stored block case to prevent compression overflow, an output buffer size of ceiling (9*Total_Input_Byte/8) + 55 bytes needs to be supplied (even though the stored block output size might be less).

- 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:



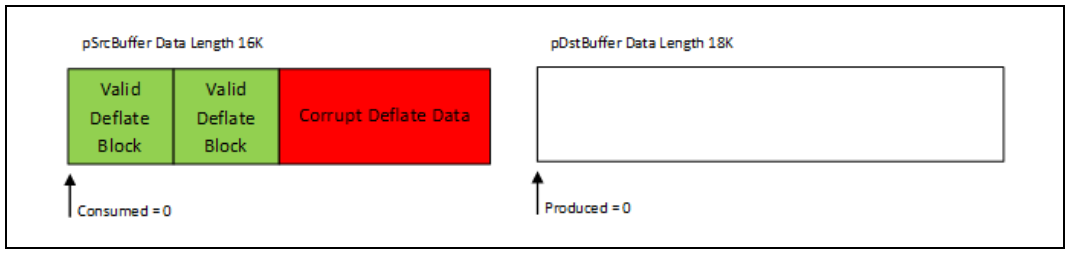
The result would be:



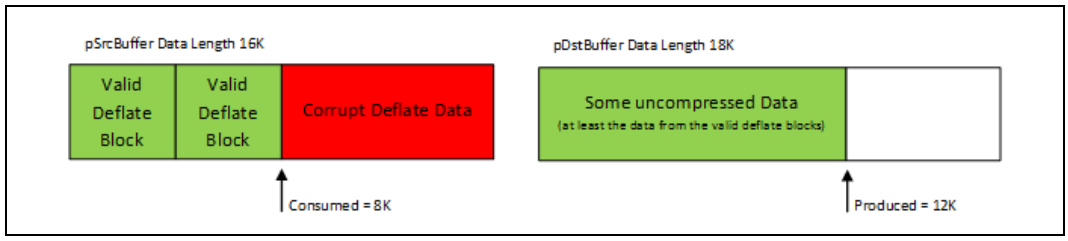

- Behavior when build flag ICP_DC_RETURN_COUNTERS_ON_ERROR is defined.
  In some specialized applications, when a decompression soft error occurs, the application has no way of correcting the source of the error and resubmitting the request. The session will need to be invalidated and terminated. In 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 compression, the maximum output size is 4 GB. Stateful compression requests that would generate an output size greater than 4.29 GB ($2^{32}$ bytes) will fail without an error.
- For stateful decompression, the maximum output size is 4.29 GB ($2^{32}$ bytes).

#### 5.1.1.1 Resubmitting After Getting an Overflow Error

The following table describes the behavior of the Intel® QuickAssist Technology compression service when an overflow occurs during a compress or decompress operation.

**Table 9. Compression/Decompression Overflow Behavior**

| Stateful/ Stateless | Static/ Dynamic | Overflow | Input data consumed? | Valid Data in Output Buffer? | Status Returned |
|---|---|---|---|---|---|
| Stateful (see details below) | Both | Yes | Possibly | Possibly | -11 |
| Stateless (see details below) | Both | Yes | No | No | -11 |

The following describes the expected behavior of an application when an overflow occurs.

**Stateful**

The produced and consumed values must be used to determine where the next request starts. Internally, the session stores `cumulativeConsumedBytes` and corresponding cumulative checksum based on these values and so expects the next request to continue after the valid data.

*Procedure*

Save the output data from the Destination buffer based on cpaDcRqResults.produced. Submit the next request with the following data:

- The first "cpaDcRqResults.consumed" bytes in the Source buffer have already been compressed, so rework the Source bufferList to start at the byte after this. Consumed = zero is a valid case; in this case, the full Source buffer must be resubmitted.
- The same Destination buffer can be re-used. It may now be big enough if part of the source data has been consumed already. Or increase if preferred.
- The results buffer can be re-used without change. In the Stateful case, the driver ignores everything in it and overwrites it on each API call.

**Stateless**

In the Stateless case, the entire compression request must be resubmitted with a larger output buffer. In this case, `cpaDcRqResults.consumed`, `.produced` and `.checksum` should be ignored. If length and checksum are required, these are not maintained in the session, and the responsibility to track these is passed up to the application.

*Procedure*

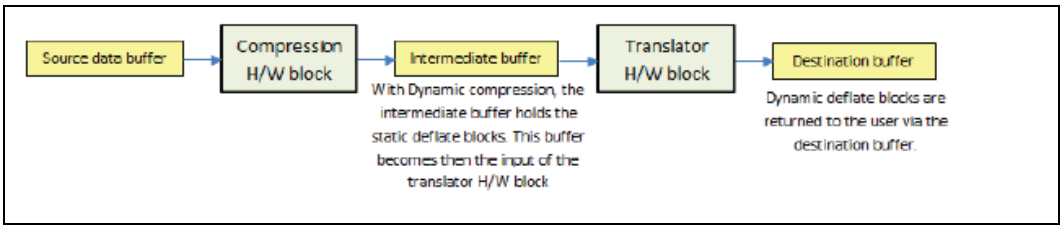
Resubmit the request with the following data:

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

#### 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. The following figure shows the dynamic compression data path.

**Figure 3. Dynamic Compression Data Path**




When the compression service returns an exception (e.g., overflow error) to the user, 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.

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. For example, in the case of an overflow during dynamic compression, static data will be returned in the destination buffer.

#### 5.1.1.3 Maximal Expansion with Auto Select Best Feature for Data Compression Service

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 which 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

Both dynamic and static compression operations are performed. The size of produced data returned in the CpaDcRqResults structure will be the minimal value of the two operations.

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

#### 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

With this ASB setting, both a dynamic and a static compression operation are performed. However, if the produced data both for the dynamic and static operation return 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(Static, 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 9.

## 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* mentioned previously 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
```

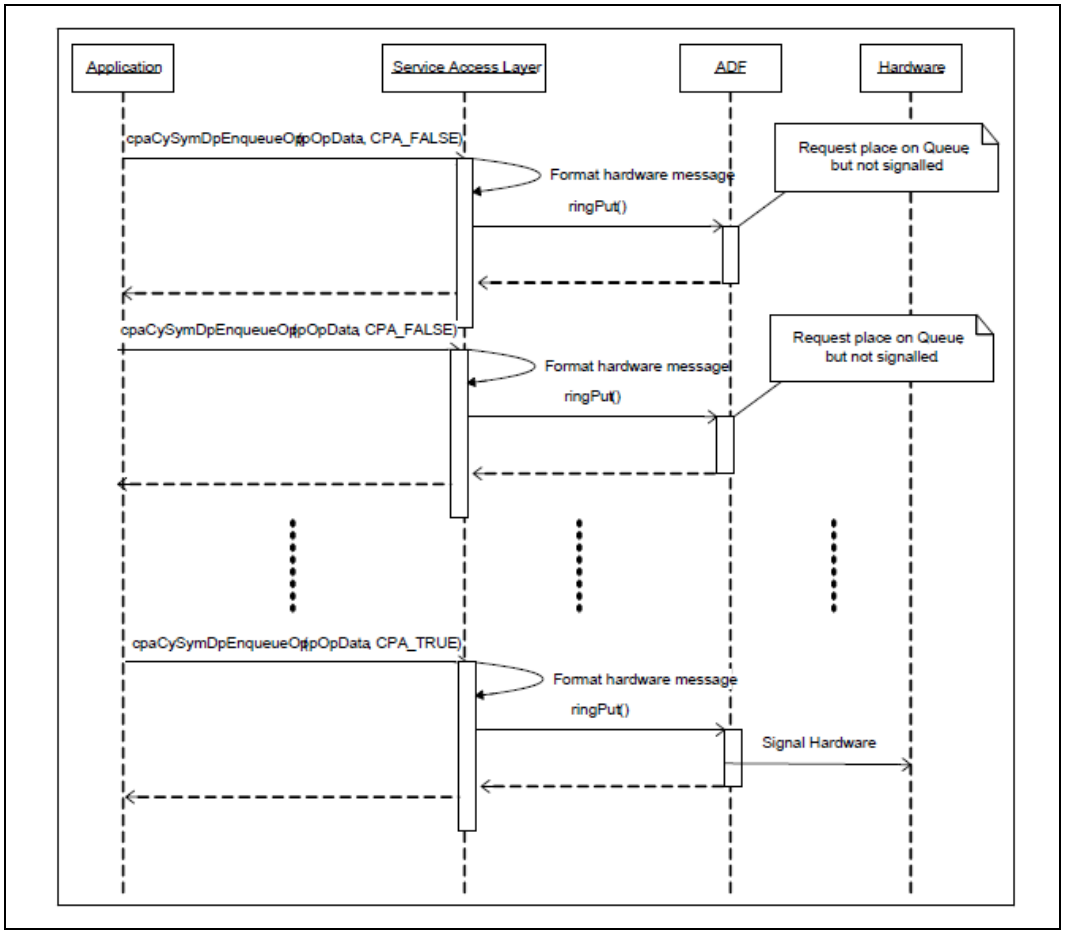
### 5.1.2.1 IA Cycle Count Reduction When Using Data Plane APIs

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 (see Usage Constraints on the Data Plane APIs on page 42).

In addition, to further improve performance, the Data Plane APIs attempt to amortize the cost of a Memory Mapped IO (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 the following figure.

**Figure 4. Amortizing the Cost of an MMIO Across Multiple Requests**




The Intel® QuickAssist Technology 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 (see Polling Functions on

page 53) 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® QuickAssist Technology 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.

#### 5.1.2.3 Cryptographic and Data Compression API Descriptions

Full descriptions of the Intel® QuickAssist Technology APIs are contained in the *Intel® QuickAssist Technology Cryptographic API Reference Manual* and the *Intel® QuickAssist Technology Data Compression API Reference Manual*. In addition to the Intel® QuickAssist Technology Data Plane APIs, there are a number of Data Plane Polling APIs that are described in Polling Functions on page 53.

## 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" on page 43
- "IOMMU Remapping Functions" on page 50
- "Polling Functions" on page 53
- "User Space Access Configuration Functions" on page 57
- "Version Information Function" on page 58
- **"Thread-Less APIs" on page 59**

### 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_FAIL | Indicates 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, 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_SUCCESS` | Successfully allocated the specified number of dc instances. |
| `CPA_STATUS_FAIL` | Indicates 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, 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:


- "icp_sal_iommu_get_remap_size" on page 51
- "icp_sal_iommu_map" on page 51
- "icp_sal_iommu_unmap" on page 51


### 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 (Cpa64Uphaddr, Cpa64U iova,
size_tsize);
```

**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® QuickAssist Technology APIs can be used, that is, kernel and user space in a Physical Function (PF) Dom0, 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 Dom0 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:



### 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 `cp_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" on page 42 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 inSection 5.1.2.2, "Usage Constraints on the Data Plane APIs" on page 42 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® QuickAssist Technology 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:


- "icp_sal_userStart" on page 57
- "icp_sal_userStop" on page 58


### 5.2.4.1 icp_sal_userStart

Initializes user space access to an 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® QuickAssist Technology 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 an M device system, 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" on page 29 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" on page 58.

### 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 isp_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:

- "icp_sal_poll_device_events" on page 60
- "icp_sal_find_new_devices" on page 60

#### 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 CpaStatus icp_sal_poll_device_events(void) ( Cpa32U
accelid);
```

**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 CpaStatus icp_sal_find_new_devices(void) ( Cpa32U
accelid);
```

**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. |

# 6.0 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® QuickAssist Technology Endpoint

A processor may be connected to one or more Intel® QuickAssist Technology 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® QuickAssist Technology APIs.

### 6.1.1 Processor and Intel® QuickAssist Technology 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:

1. The processor uses a write (put) operation to place a request on the request ring.
2. The QAT endpoint uses a read (get) operation to retrieve the request from the request ring.
3. Once the operation has been performed, the QAT endpoint uses a write (put) operation to put the response to the response ring.
4. 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 5. 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 Guidelines for Using Multiple Intel® QuickAssist Technology Instances for Load Balancing in Cryptography Applications

The application is responsible for load balancing/spreading requests across Intel® QuickAssist Technology 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). See IPsec and SSL VPNs on page 63 for more information.

- Encrypted Storage. See Encrypted Storage on page 63 for more information.
- Web Proxy Appliances. See Web Proxy Appliances on page 64.

See also the *Accelerating a Security Appliance White Paper*. This was first written to support the Intel® EP8057 Integrated Processor with Intel® QuickAssist Technology. Many of the concepts and ideas are applicable to the platforms described in this manual also.

### 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 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® QuickAssist Technology 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 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® QuickAssist Technology 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® QuickAssist Technology 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® QuickAssist Technology 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.