PENSANDO

SYSTEMS

SONIC API User Guide

Version 0.2.0

Dec 30, 2018

PENSANDO
SYSTEMS

Table of Contents

Legal

All information in this document is provided on a non-disclosure basis. Anyone reading this document implicitly agrees to be bound by Pensando Systems' non-disclosure terms.

Document Revision History

Revision	Author	Date	Status and Description
0.1	Roger	2018-04-16	Initial Version
0.2	Jeff	2018-11-19	Target for 11/30 deliverables.
0.2.1	Jeff	2019-01-01	Target for 1/2/19 deliverables

PENSANDO
SYSTEMS

NAPLES Offload Engine Overview

Introduction

Pensando NAPLES is a SmartNIC capable of handling both network traffic as well as providing hardware and CPU offload for Encryption/Decryption, Compression/Decompression, hash and checksum calculations, also known as accelerator services. To access these hardware accelerator services Pensando has developed the SONIC kernel driver that provides APIs for interaction with the offload services on NAPLES. "offload services" are defined as service requests that interact with the offload engines.

This document is intended for software engineers who need an understanding of the Pensando SONIC kernel driver and API for writing software that interacts with the offload services. This document contains the information needed to get started using and interacting with the SONIC kernel driver and the different accelerator services.

Prerequisites

If running on a FreeBSD-based kernel, this driver assumes that the kernel is compiled with `COMPAT_LINUXKPI`. Instructions are included in **Appendix A**.

The SONIC driver also requires that **`PCI ARI`** is disabled in the running kernel.
To verify:

```
# sysctl -a | grep hw.pci.enable_ari
hw.pci.enable_ari: 0
```

If **`enable_ari`** is not set to zero, then please run the command below and reboot:

```
# Disable PCI ARI
echo hw.pci.enable_ari="0" >> /boot/loader.conf
```

Offload Engines and Algorithms

The NAPLES adapter provides multiple offload engines and algorithms, as described by the block diagram and table below. The block diagram shows all the offload engines connected to the Network-on-chip (NOC) switch. Each accelerator engine supports data transfer bandwidth up to 100 Gbps.

These offload services can be chained and batched together as atomic operations, with all operations performed at wire speed.

Block Diagram of the NAPLES Adapter, including all offload engines

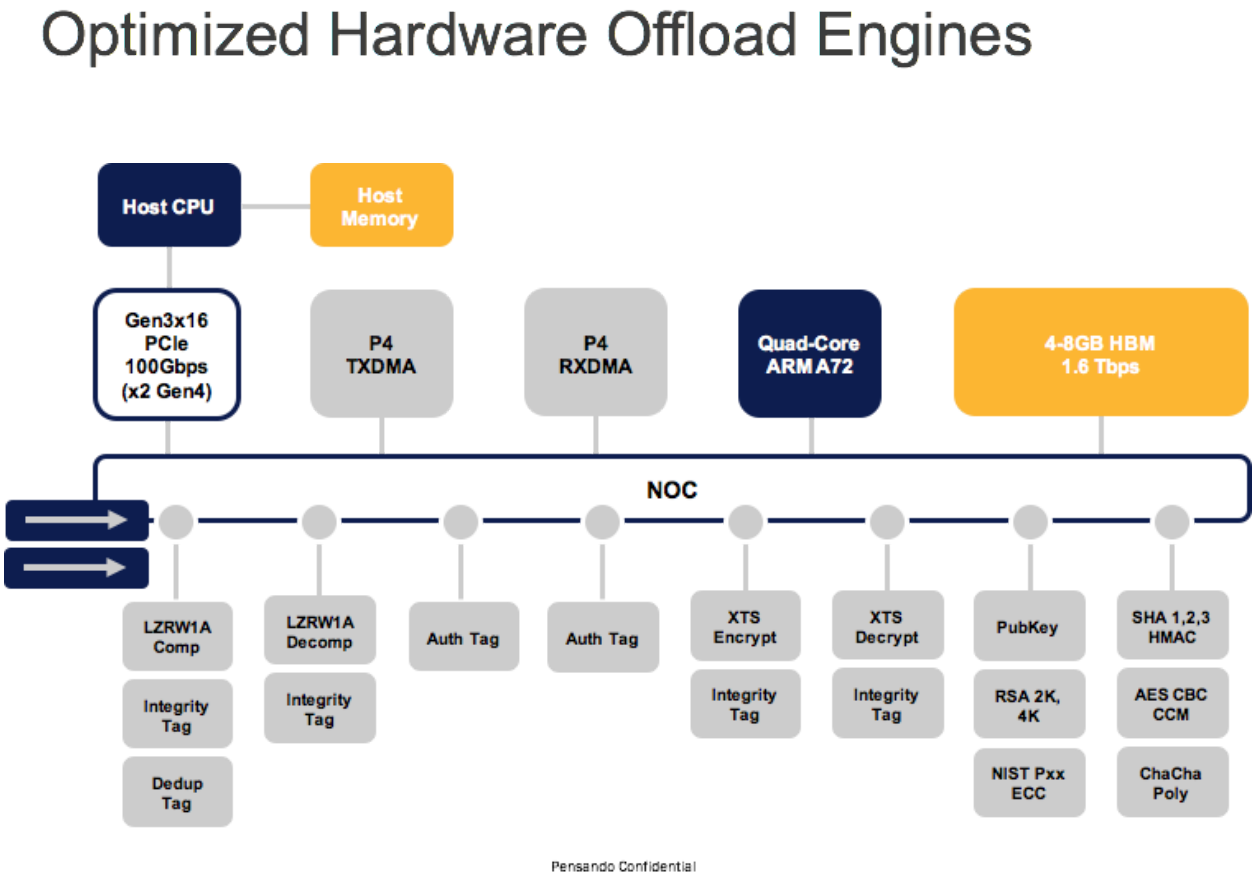


Table of Offload Engine Algorithms, supported by the current SONIC driver

Offload	Algorithm	Block Size	Info
Data Compression	LZRW1A	up to 64K	Insertion of an 8-Byte Compression Header (32-bit Checksum, 16-bit Length, 16-bit Version)
Data Decompression	LZRW1A	up to 64K	Removal of the 8-Byte compression Header
Data Encryption	XTS 256-bit	up to 4M and multiple of 16 Bytes	IV = LBA #
Data Decryption	XTS 256-bit	up to 4M and multiple of 16 Bytes	IV = LBA #
Deduplication	SHA-256/512	up to 4M	
Checksum	M-Adler-32		

Please note: The NAPLES SONIC driver currently under development. As a result, this document may describe features that have not currently been implemented. As of 11/30/18, the NAPLES SONIC driver supports only Data Compression/Decompression offload services.

Please note: The NAPLES adapter has additional offload engines that can be used by various networking protocols, that are currently not available in the SONIC driver.

SONIC Driver Overview

About the SONIC Driver

The Pensando SONIC Driver is a kernel device driver that supports all the necessary API's needed to interact with the offload services. As with any kernel module or device driver, the Pensando device driver can be loaded and unloaded to and from kernel at any time. [Ex: kldload/kldunload (FreeBSD) or insmod/rmmod (Centos).]

Driver initialization

The SONIC driver needs to be loaded into the kernel with root privileges. Below are the commands, depending on the specific OS:

OS	Command
FreeBSD	kldload sonic.ko
Linux	insmod sonic.ko

FreeBSD SONIC Driver Configuration

The SONIC Driver has the following tunable configuration variables that can set via "**kenv**":

- **compat.linuxkpi.sonic_log_level**="N"
  Specifies the logging level for the SONIC Driver. The standard Linux logging levels are used (See "Logging")

- **compat.linuxkpi.sonic_core_count**="N"
  Specifies the maximum number of cpu cores that can be used by the SONIC Driver.

  WARNING: The value of **compat.linuxkpi.sonic_core_count** should generally not be used, except to set it to the actual number of system cores.

Ex:

```
# kenv compat.linuxkpi.sonic_log_level="7"
# kenv compat.linuxkpi.sonic_core_count="16"
```

FreeBSD SONIC Driver Installation

The SONIC Driver is a binary file that is installed using the "kldload" command. Ex:

```
# kldload sonic.ko
```

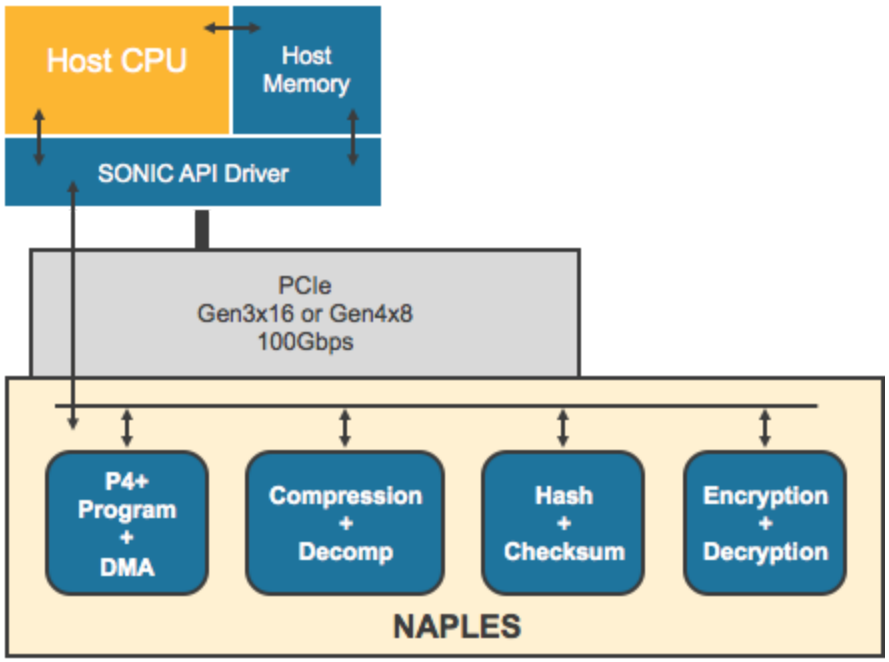
SONIC API Overview

About the SONIC API

The SONIC driver is a kernel driver available in both FreeBSD and LINUX.
Please note that decompression and XTS decryption API has been optimized for low latency and is therefore synchronous in nature. All other API's are optimized for high throughput and are therefore asynchronous in nature.

Architecture of the API

Host API is supported by P4 Programs and P4 DMA acting as an intermediary. P4+ programs are controlling the Storage Accelerator. See diagram below.



Service Requests

A service request includes a single data set to be processed by one or more offload engines. A service request can be sent as a single request (one service) or chained (multiple services). Please see below.

Single Service Request

A single request is used when a single dataset needs processing by a single service, such as encryption or compression. A single request invokes only a single accelerator service.

Chaining Service Requests

Chaining is used when the same dataset needs to be processed by multiple services, for example Encryption, Compression and Checksum calculation. The benefit with chaining is that a request is processed and forwarded among the different offload engines. A chaining request is processed as a single atomic operation.

Service Request Overview

Requests can be submitted to the offload engines in one of three different ways: Synchronous, Asynchronous or Poll.

Synchronous

Submitting a request synchronously will hold the calling thread until the result is returned. Please note this could affect the overall performance and might not be the optimal way to submit the requests. In general, synchronous requests should be avoided, except for data and meta-data updates that require the strictest serialization.

Asynchronous

Submitting a request asynchronously will execute the request as a separate thread and will not hold the calling thread. Rather than using interrupts, the asynchronous API callback functions will return the result via a callback function provided at the time of submission. Please note that the callbacks are invoked in the context of the submitting thread, and that the API does not allocate memory/buffers for the callback function.

Poll

Submitting a request through the Poll function is similar to Asynchronous requests. The Poll type also uses a callback function, but the user needs to poll to get the status of the request. Once the result is ready, the poll function will invoke the callback function.
Please note that the callbacks are invoked in the context of a polling thread, and that the API is not handling the creation and scheduling of these polling threads.

Processing of the Submitted Request

Requests can be processed in Non-Batch or Batch mode. Non-Bach requests contains one single service request (Single or chained). Batch mode is used to submit multiple different service requests (Single or chained), to be processed in a single call.

Non-Batch

Non-Batch is used to submit one dataset for processing, using one or more services (Single or Chained).

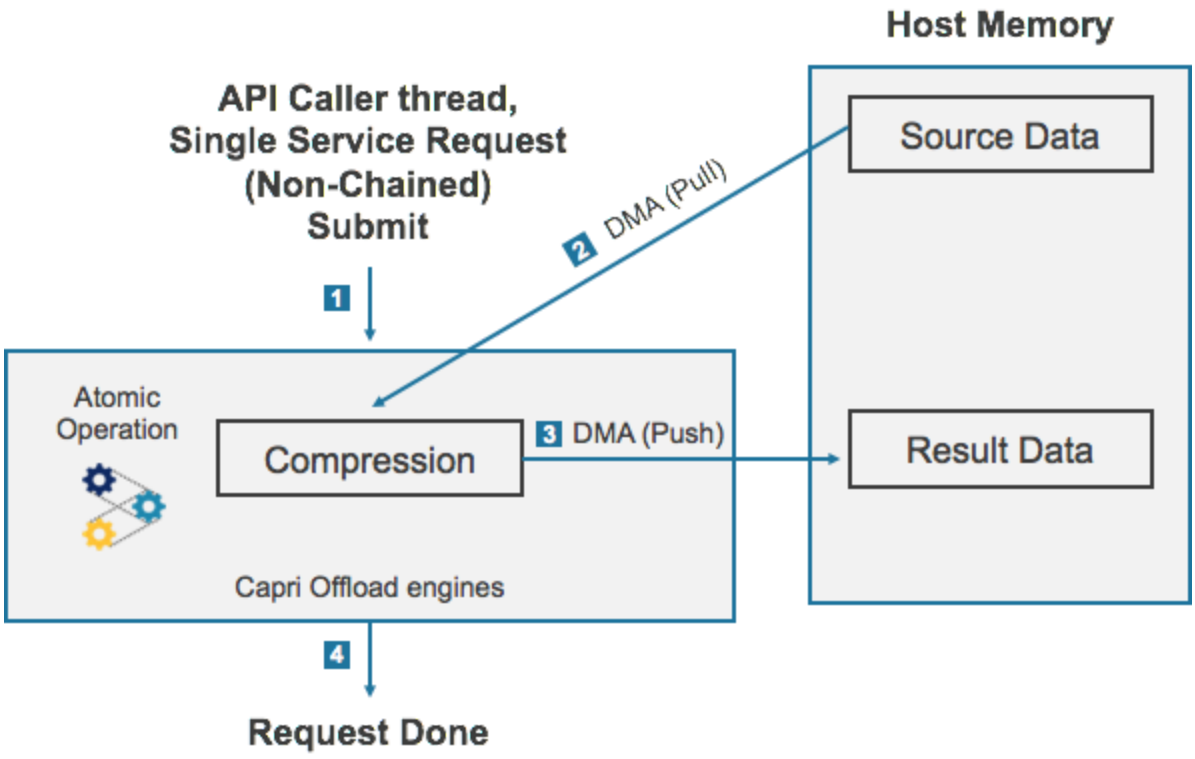
Batch

Batching is way to submit multiple requests with different service requests (Single or Chained) all to be processed in a single call. When submitting the request as a batch request, each service request is processed in parallel and atomicly, but the result is not returned until all processing of all data sets is completed.

Service Request Types

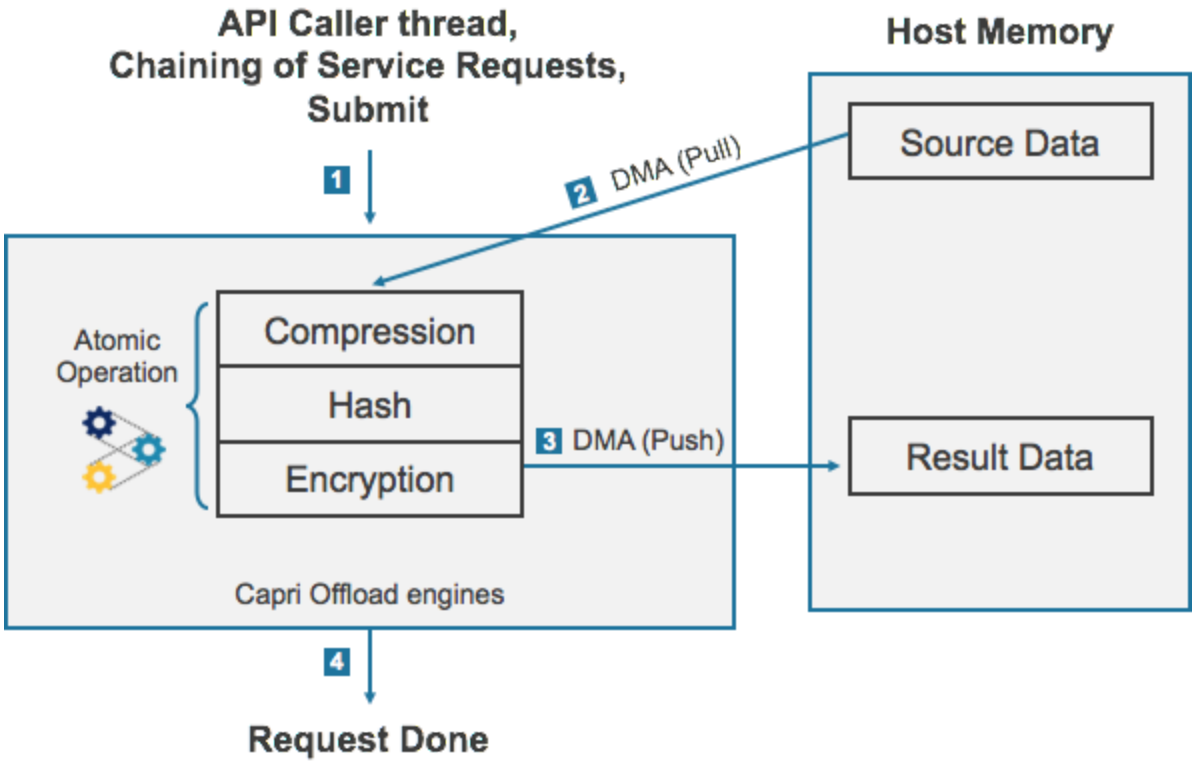
Non-Batched, Single Service Request

Below shows a single service compression request.



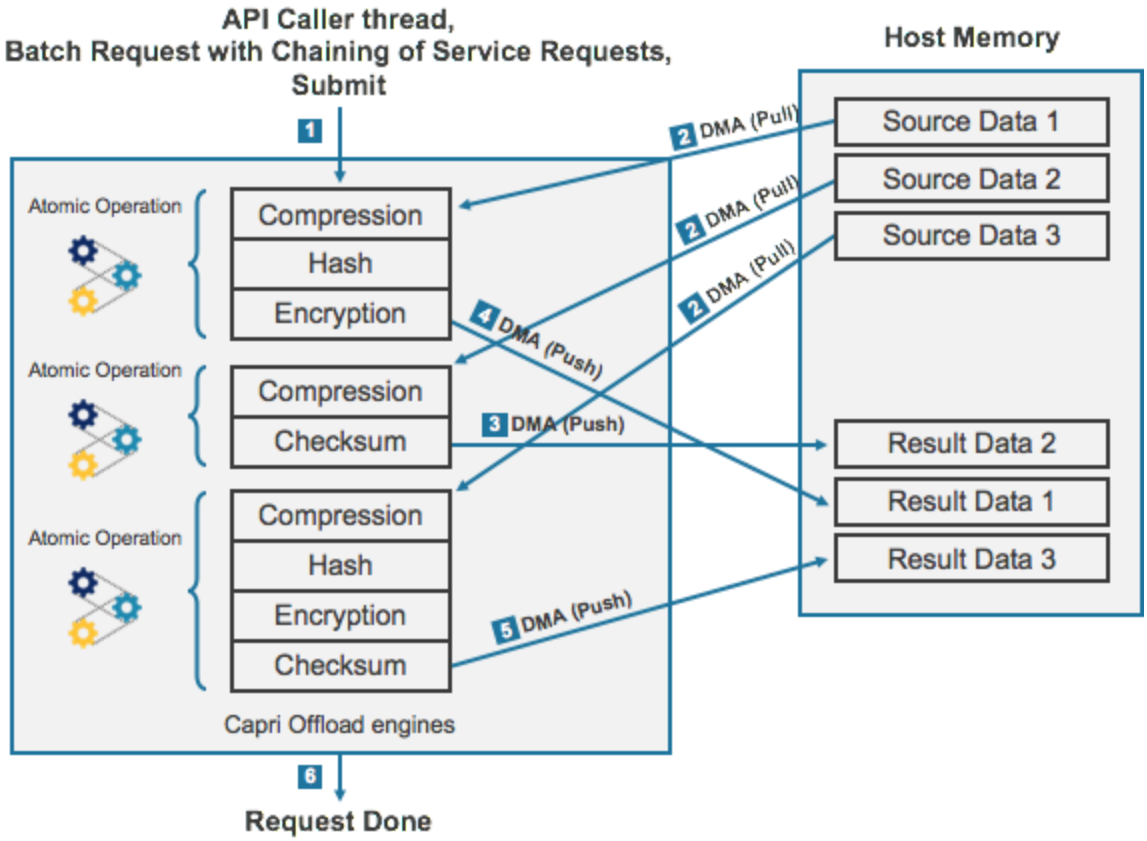
Non-Batched, Chained Service Request

Below shows a chained request on a single data set that includes multiple different services (Compression, Hash and Encryption).



Batched, Multiple Service Requests

Below shows three chained service requests that use multiple different offload services (Compression, Hash and Encryption) in various combinations. The batched request is considered complete once all processing of all service requests are completed.

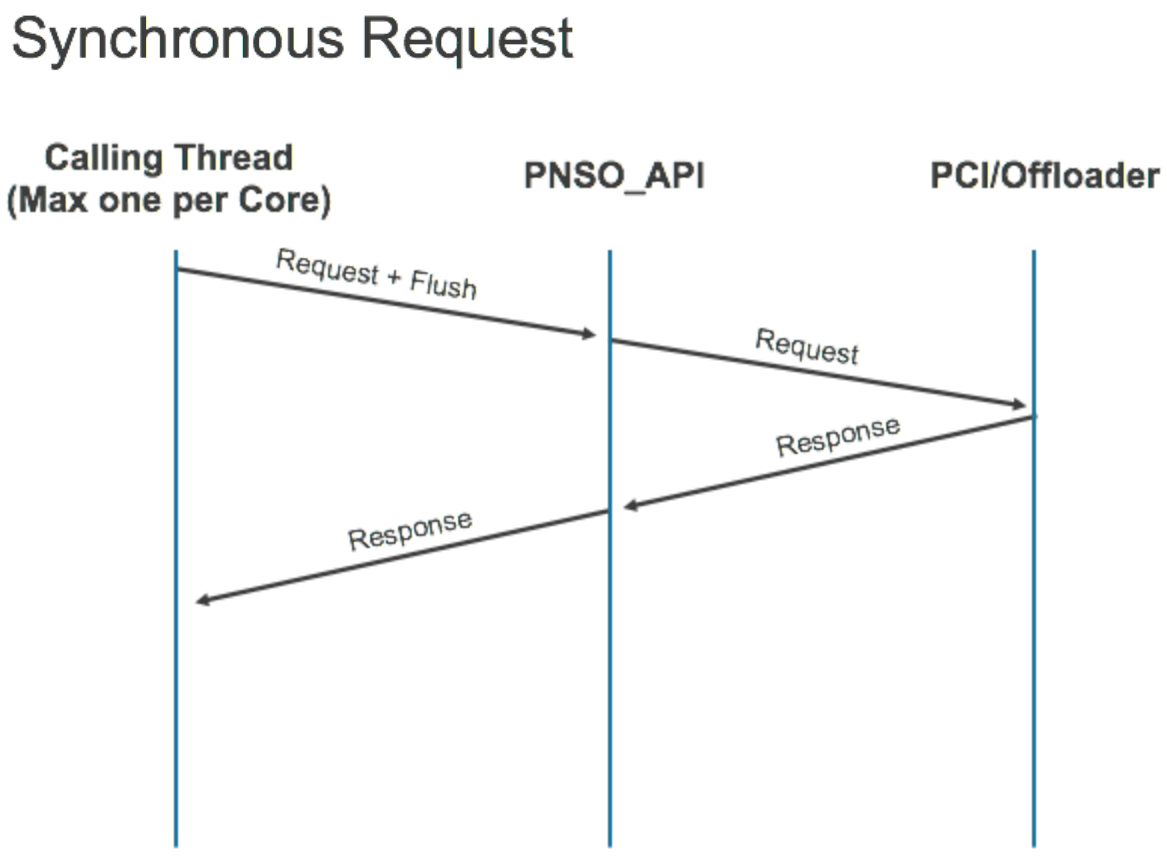


Submitting and Processing the Request

There are three different ways which service requests can be submitted, depending on the desired interaction with the offload services and the processing of the results. Requests can be submitted by one of three methods: Synchronous, Asynchronous or Poll. The three different methods will determine how the request is submitted and how the caller is notified upon completion. The different methods are described below.

Synchronous (Non-Batched)

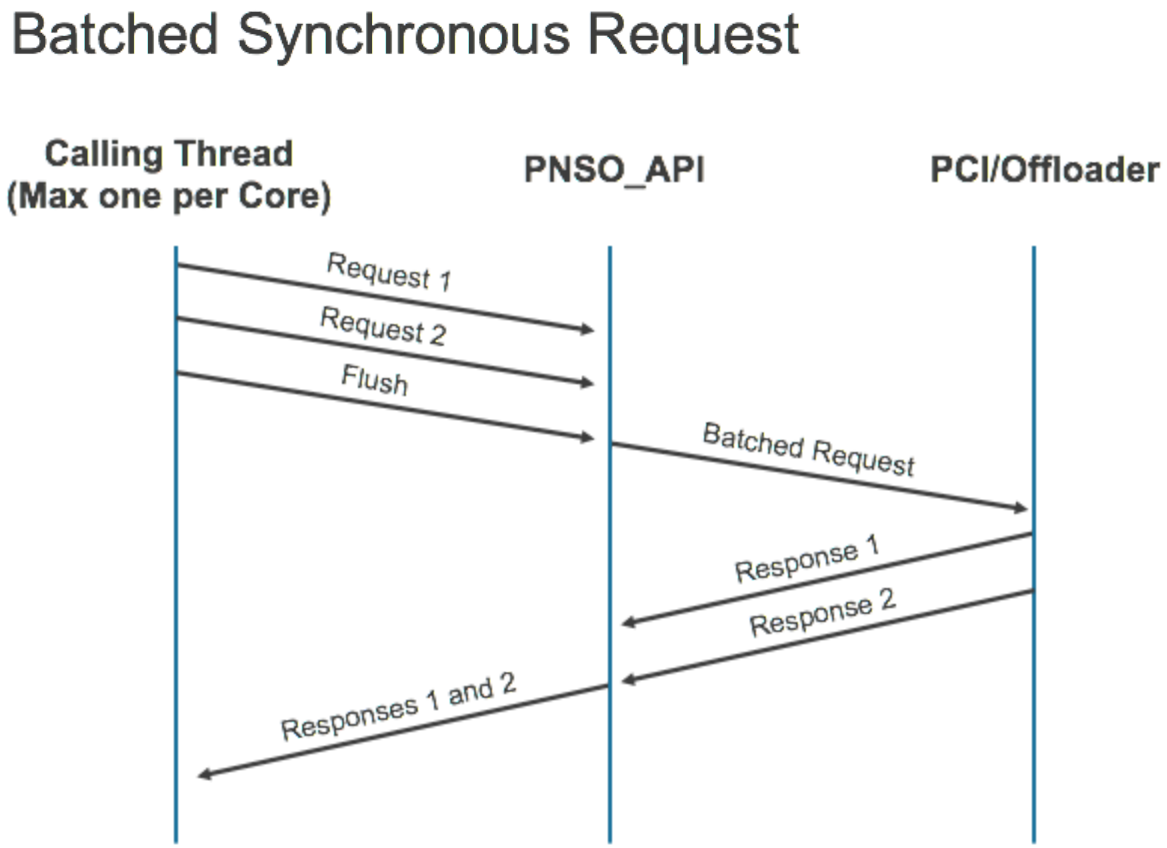
The **'pnso_submit_request'** function will complete the request and return with the result. The calling thread will wait synchronously for completion of the request. This request requires pointers to the request (*req) and response (*res) buffers.



Type	API Function Call	Description
Request + Flush	pnso_submit_request	Submit and process one request atomically. (Chained or Non-Chained) ***Note: Caller thread is blocked until the response is returned.***

Synchronous (Batched)

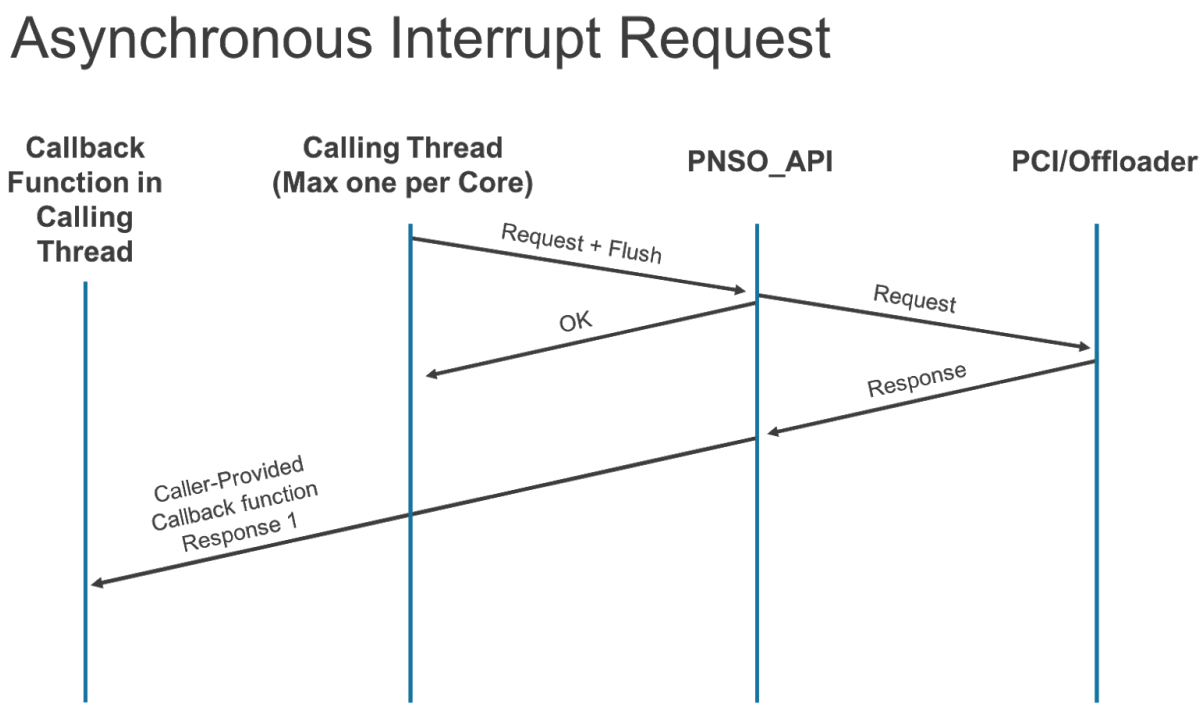
The **'pnso_add_to_batch'** and **'pnso_flush_batch'** functions will complete multiple batched requests and return with the batched result. The calling thread will be waiting synchronously for the completion of all requests in the batch. Synchronous requests require pointers to the request (*req) and response (*res) buffers.



Type	API Function Call	Description
Request	pnso_add_to_batch	Adds a request to batch buffer. (Chained or Non-Chained)
Flush	pnso_flush_batch	Processes all of the requests in the batch buffer atomically. Responses are available once all requests has been processed. ***Note: Caller thread is blocked until the response is returned.***

Asynchronous (Non-Batched)

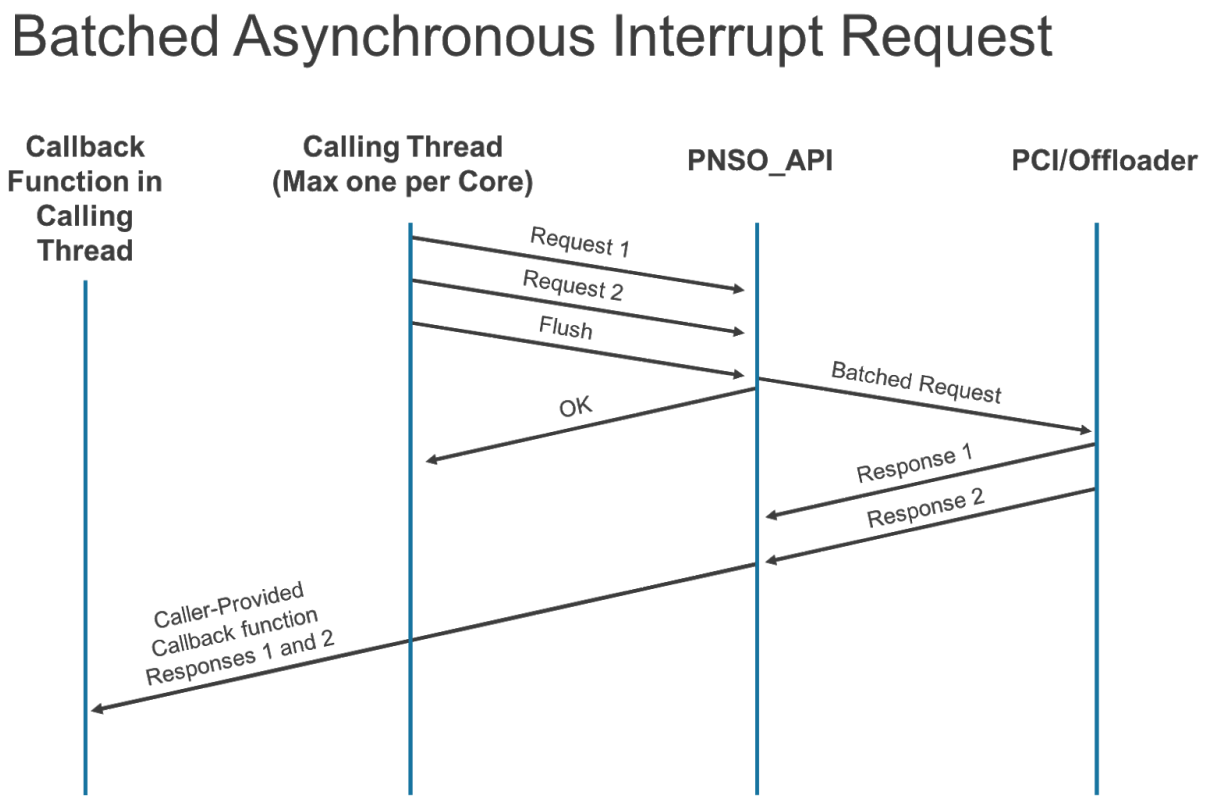
The **'pnso_submit_request'** function returns immediately, and completes the request in the background before invoking a caller-provided callback function. In this request, pointers are provided for the request (*req) and response (*res) buffers, the callback function (cb_func) and callback context (*cb_ctx). Once the request has been completed, the callback function will be invoked, indicating that the result is ready for processing.



Type	API Function Call	Description
Request + Flush	pnso_submit_request	Submit and process one request atomically. (Chained or Non-Chained) ***Note: Caller thread continues to execute. The response is returned by a caller-provided callback function.***

Asynchronous (Batched)

The **'pnso_add_to_batch'** and **'pnso_flush_batch'** functions return immediately. In this request, pointers are provided for the requests (*req) and response (*res) buffers, the callback function (cb_func) and callback context (*cb_ctx). Once all the requests have been completed, the callback function will be invoked, indicating that the results are ready for processing.

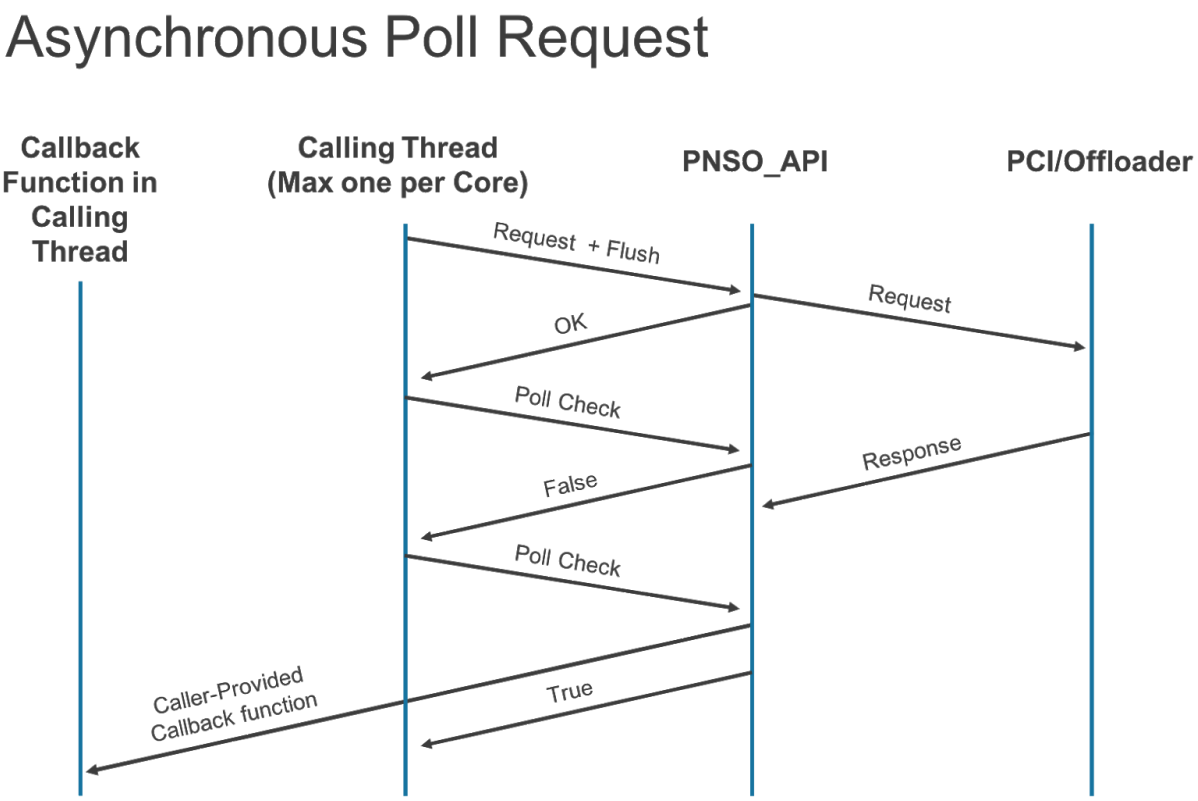


Type	API Function Call	Description
Request	pnso_add_to_batch	Adds a request to batch buffer. (Chained or Non-Chained)
Flush	pnso_flush_batch	Processes all of the requests in the batch buffer atomically. Responses are available once all requests has been processed. ***Note: Caller thread continues to execute. The responses are returned by a caller-provided callback function.***

Poll (Non-Batch)

The **'pnso_submit_request'** function returns immediately. The request is completed in the background before invoking a caller-provided callback function. In this request, pointers are provided for the request (*req) and response (*res) buffers, and poll function (*poll_func) and opaque poll context (**poll_ctx). Competition status is polled for, indicating that the result is ready for processing. The poll is done through the API-provided **'pnso_poll_fn'** polling function pointer, in combination with the API-provided **'pnso_poll_ctx'** for the polling function. The API provides both the polling function and the polling function context to use when calling the polling function. The caller has the responsibility for maintaining the corresponding polling context for each outstanding poll request.

Please note: The callback function is called AFTER a successful poll check call, please see below:



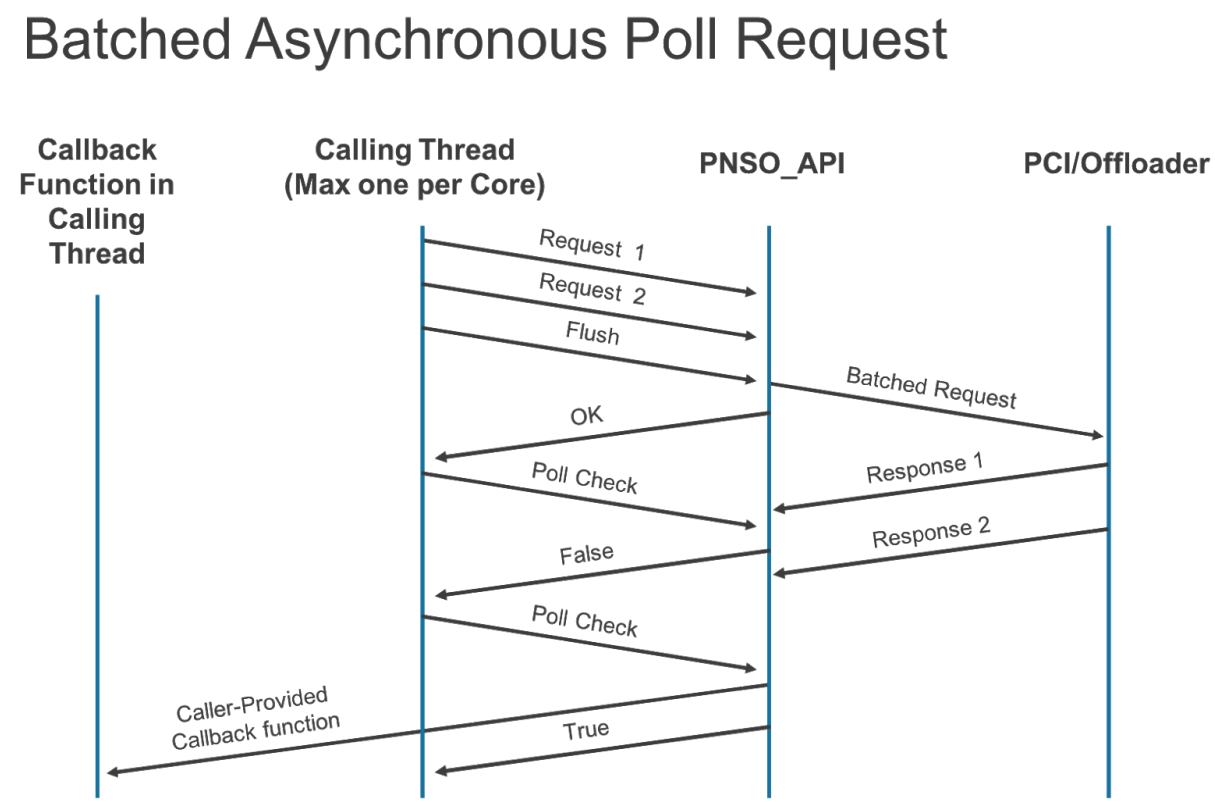
Type	API Function Call	Description
Request + Flush	pnso_submit_request	Submit and process one request atomically. (Chained or Non-Chained) ***Note: Caller thread continues to execute. The response is returned by a caller-provided callback function AFTER the caller thread performs a poll check***

		and the result is available.
Poll Check	pnso_poll_fn	Polling function to check for completion. Context is provided by 'pnso_poll_ctx'.

Poll (Batched)

The **'pnso_add_to_batch'** and **'pnso_flush_batch'** functions return immediately. In this request, pointers are provided for the request (*req) and response (*res) buffers, and poll function (*poll_func) and poll function context (**poll_ctx). Both the (*poll_func) and the (**poll_ctx) are returned/provided by the API driver. Completion status is polled for, indicating that the result is ready for processing. The poll is done through the **'pnso_poll_fn'** polling function pointer, in combination with the API-provided **'pnso_poll_ctx'** for the polling function. The API provides both the polling function and the polling function context to use when calling the polling function. The caller has the responsibility for maintaining the corresponding polling context for each outstanding poll request.

Please note: The callback function is called AFTER a successful poll check call. Please see below:



Type	API Function Call	Description
Request	pnso_add_to_batch	Adds a request to batch buffer (Chained or Non-Chained)
Flush	pnso_flush_batch	Processes all of the requests in the batch buffer atomically. Responses are available once all requests have been processed. ***Note: Caller thread continues to execute. The response is returned by a caller-provided callback function AFTER the caller thread performs a poll check and the result is available.***
Poll Check	pnso_poll_fn	Polling function to check for completion. Context is provided by **'pnso_poll_ctx'**.

Using the Storage API

Include Files

Callers of the SONIC API must include the following files:

```
#include "pnso_api.h"
```

Memory Allocation and Ownership

Please note that all host memory needs to allocated outside the API. The API assumes that the calling functions **must** provide pointers to allocated host memory. The API does not allocate memory.

Buffers and Lists

All buffers and buffer lists are passed using physical addresses to avoid virtual to physical address translation costs.

Flat Buffer

The smallest unit of buffer is **'pnso_flat_buffer'**, containing **'len'** which is the length of the buffer in bytes, and **'buf'** which is a pointer to a physical address where the data (buffer) resides.

```
struct pnso_flat_buffer {
    uint32_t len;
```

```
        uint64_t buf;
    };
```

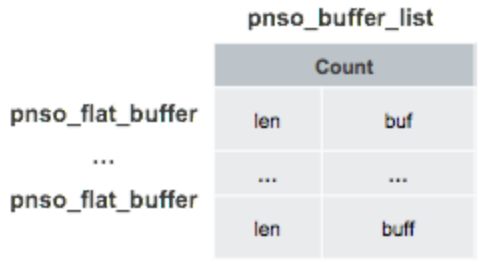
Scatter Gather List (SGL)

The **‘pnso_buffer_list’** defines a scatter/gather buffer list. This structure is used to represent a collection of physical memory buffers that are not contiguous. The **‘count’** specifies the numbers of buffers in the list and **‘buffers’** specifies an unbounded array of flat buffers as defined by **‘count**’. The buffers are used for offload engine data requests and results.

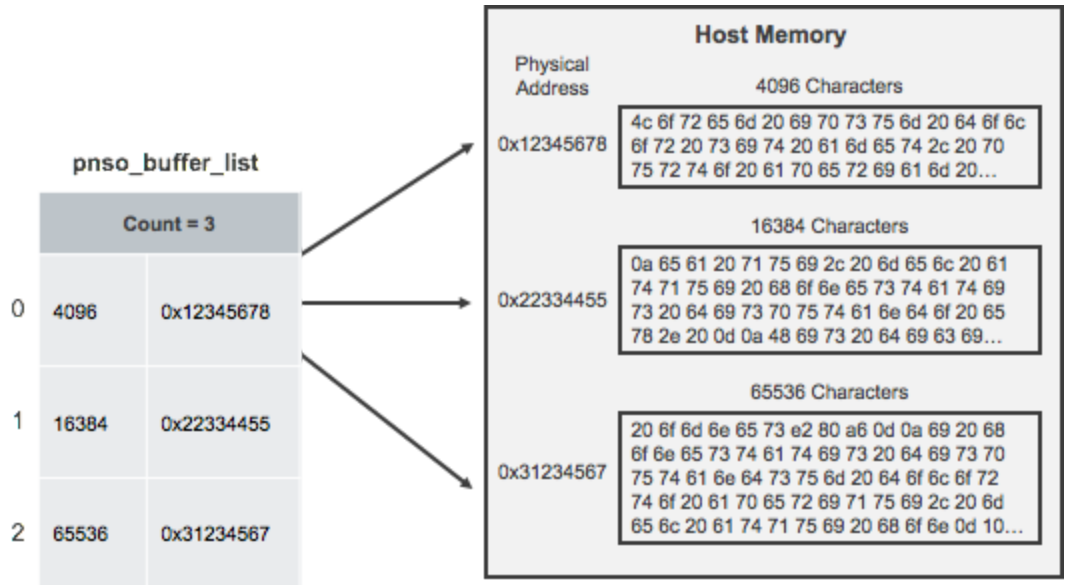
```
struct pnso_buffer_list {
    uint32_t count;
    struct pnso_flat_buffer buffers[0];
};
```

Flat Buffers and SGL Relationship

Below is a visualization of the **‘flat_buffer’** and **‘buffer_list’** relationship:



The image below is an example of a buffer_list with 3 x flat_buffer’s pointing to 3 different physical memory addresses where the buffers (data) reside.



Initialization and Service Descriptors

Before any of the offload services can be invoked, certain initialization is required, depending on which accelerator is invoked. Please see below:

- Driver Initialization
- API Initialization
- Offload service Initialization
- Offload service description
- Submit offload service request
- Access and process the result

API initialization

API initialization is required before the offload services can be invoked. Initialization is done by invoking the **'pnso_init'** function.

The **'pnso_init'** expect to be passed initialization parameters for the offload services. This is done through the struct **'pnso_init_params'**. The **'pnso_init'** function will return **'PNSO_OK'** indicating success, or **'-EINVAL'** if invalid parameters where passed.

The function **'pnso_init'** is defined as follows:

pnso_error_t pnso_init(struct pnso_init_params *init_params);

Please note: *Caller is responsible for allocation and deallocation of memory for input parameters.*

The **'*pnso_init_params*'** represents the initialization parameters for Pensando offload services. It is a struct and is defined as follow:

- per_core_qdepth: Specifies the maximum number of parallel outstanding requests per host CPU core.
- core_count: Specifies the number of CPU cores
- block_size: Specifies the native filesystem block size in bytes.

```
struct pnso_init_params {
        uint16_t per_core_qdepth;
        uint16_t core_count;
        uint32_t block_size;
};
```

Setting the "**per_core_qdepth**" should aim to balance request concurrency with system memory use. Setting the value too low may result in service request not being accepted by the API. Setting the value too high may consume memory unnecessarily.

Offload Service Initialization

Crypto Engine Initialization

The crypto accelerator service requires first registering the crypto key descriptor index, and Initialization Vector (IV).

XTS (XEX-based tweaked-codebook mode with ciphertext stealing) is a symmetric algorithm and requires a key, and a key index definition in the key index descriptor table.

The initialization is done by calling the **'pnso_set_key_dec_idx'** function and set the key for data encryption and a key for the descriptor index, please see below:

- **key1**: Specifies the key that will be used to encrypt the data
- **key2**: Specifies the key that will be used to encrypt initialization vector
- key_size: Specifies the size of the key in bytes -- 16 and 32 bytes for AES128 and AES256 respectively.
- **key_idx**: Specifies the key index in the descriptor table.

Return Value:

- PNSO_OK - on success
- -EINVAL - on invalid input parameters

```
pnso_error_t pnso_set_key_desc_idx(const void *key1,
                  const void *key2,
                  uint32_t key_size, uint32_t key_idx);
```

Please note: *The caller is responsible for allocation/deallocation of memory for input parameters.*

Compression Engine Initialization

Initialization of the compression accelerator requires registering a new header format, and adding a compression algorithm mapping. The mapping is the Pensando compression algorithm number to the customer's opaque algorithm identifier in the compression header. This allows customers to have their own list mapped to potentially different Pensando capabilities.

The registration is done by calling the '**pnso_register_compression_header_format**' function and providing the header format to be embedded at the beginning of the the compressed data. Please see below:

- **cp_hdr_fmt**: The header format to be embedded
- **hdr_fmt_idx**: Non-Zero index to uniquely identify the header format

Return Value:
- PNSO_OK - on success
- -EINVAL - on invalid input parameters

```
pnso_error_t pnso_register_compression_header_format(
        struct pnso_compression_header_format *cp_hdr_fmt,
        uint16_t hdr_fmt_idx);
```

Algorithm mapping is done by calling the **'pnso_add_compression_algo_mapping**' function and providing the compression algorithm number (Please see the API reference for a complete list of algorithms supported), and the compression header algorithm number. Please see below:

- **pnso_algo**: The compression algorithm number
- **header_algo**: The compression header algorithm number

Return Value:
- PNSO_OK - on success
- -EINVAL - on invalid input parameters

```
pnso_error_t pnso_add_compression_algo_mapping(
        enum pnso_compression_type pnso_algo,
        uint32_t header_algo);
```

Please Note: *Caller is responsible for managing the hdr_fmt_idx space and allocation/deallocation of memory for input parameters*

Offload Service Descriptors

Offload service requests require configuration of the service details. This configuration is done through service descriptors "**pnso_service_request**". As mentioned earlier, it is possible to chain multiple accelerator requests through service chaining. Chaining is done through a "svc[]" array.

A location must be provided for the result set through the "**pnso_service_result**" parameter.

Details for the different accelerator engines are provided below.

Crypto Engine

The crypto service is defined using the **'pnso_service'**. Please note that it is a 'union', and for the crypto accelerator the **'pnso_crypto_desc'** is used. The **'pnso_service'** is defined as follows:

- **svc_type**: specifies one of the enumerated values for the accelerator service type (for crypto, use the **pnso_crypto_desc**).
- **rsvd**: specifies a 'reserved' field meant to be used by Pensando.
- **crypto_desc**: struct that specifies the descriptor for encryption/decryption service.

The other services in this struct are described together with the corresponding accelerator service in this document.

```
struct pnso_service {
    uint16_t svc_type;
    uint16_t rsvd;
    union {
        struct pnso_crypto_desc crypto_desc;
        struct pnso_compression_desc cp_desc;
        struct pnso_decompression_desc dc_desc;
        struct pnso_hash_desc hash_desc;
        struct pnso_checksum_desc chksum_desc;
        struct pnso_decompaction_desc decompact_desc;
    } u;
};
```

The **'pnso_crypto_desc'** is the descriptor for encryption or decryption operation, it is a struct and is defined as follow:

- **algo_type**: Specifies one of the enumerated values of the crypto type (i.e. the enum **pnso_crypto_type**. See below).
- **rsvd**: Specifies a 'reserved' field meant to be used by Pensando.
- **key_desc_idx**: Specifies the key index in the descriptor table.
- **iv_addr**: Specifies the physical address of the initialization vector.

```
struct pnso_crypto_desc {
    uint16_t algo_type;
```

```
    uint16_t rsvd;
    uint32_t key_desc_idx;
    uint64_t iv_addr;
};
```

The **‘pnso_crypto_type’** is an enum and is defined as follow:

```
enum pnso_crypto_type {
    PNSO_CRYPTO_TYPE_NONE = 0,
    PNSO_CRYPTO_TYPE_XTS = 1,
    PNSO_CRYPTO_TYPE_MAX
};
```

This list allows capabilities to be extended with additional crypto types in future releases. For a complete list, please refer to the API Reference Guide. Currently, XTS is the only crypto service supported.

Compression/Decompression Engine

The compression service is defined using the **‘pnso_service’**. Please note that it is a ‘union’, and for the compression accelerator the **‘*pnso_compression_desc*’** or ‘***pnso_decompression_desc*’** are used. The **‘pnso_service’** is defined as follows:

- **svc_type**: specifies one of the enumerated values for the accelerator service type (for compression/decompression it would be defined as either ‘pnso_compression_desc’ or ‘pnso_decompression_desc’).
- **rsvd**: specifies a 'reserved' field meant to be used by Pensando.
- ***cp_desc/dc_desc***: struct that specifies the descriptor for compression/decompression services.

The other services in this struct are described together with the corresponding accelerator service in this document.

```
struct pnso_service {
    uint16_t svc_type;
    uint16_t rsvd;
    union {
        struct pnso_crypto_desc crypto_desc;
        struct pnso_compression_desc cp_desc;
        struct pnso_decompression_desc dc_desc;
        struct pnso_hash_desc hash_desc;
        struct pnso_checksum_desc chksum_desc;
        struct pnso_decompaction_desc decompact_desc;
    } u;
};
```

The **‘*pnso_compression_desc*’** is the descriptor for compression operation. It is a struct and defined as follow:

- **algo_type**: Specifies one of the enumerated values of the compressor algorithm (i.e. **pnso_compression_type**).
- **flags**: Specifies the following applicable descriptor flags to compression descriptor:

Flags	Description
PNSO_CP_DFLAG_ZERO_PAD	Zero fill the compressed output buffer aligning to block size.
PNSO_CP_DFLAG_INSERT_HEADER	Insert compression header defined by the format supplied in 'struct pnso_init_params'.
PNSO_CP_DFLAG_BYPASS_ONFAIL	Use the source buffer as input buffer to hash and/or checksum, services, when compression operation fails. This flag is effective only when compression, hash and/or checksum operation is requested.

- **threshold_len**: specifies the expected compressed buffer length in bytes. (This is to instruct the compression operation, upon its completion, to compress the buffer to a length that must be less than or equal to 'threshold_len').
- **hdr_fmt_idx**: specifies the index for the header format in the header format array.
- **hdr_algo**: specifies the value for header field PNSO_HDR_FIELD_TYPE_ALGO (This is the same value that is registered in '**pnso_add_compression_algo_mapping**').

```
struct pnso_compression_desc {
    uint16_t algo_type;
    uint16_t flags;
    uint16_t threshold_len;
    uint16_t hdr_fmt_idx;
    uint32_t hdr_algo;
};
```

The '***pnso_decompression_desc***' is the descriptor for the compression operation. It is a struct, defined as follows:

- **algo_type**: specifies one of the enumerated values of the compressor algorithm (i.e. **pnso_compression_type**) for decompression.
- **flags**: specifies the following applicable descriptor flags to decompression descriptor:

Flags	Description
PNSO_DC_DFLAG_HEADER_PRESENT	Indicates the compression header is present.

- **hdr_fmt_idx**: specifies the index for the header format in the header format array.
- **rsvd**: specifies a 'reserved' field meant to be used by Pensando.

```
struct pnso_decompression_desc {
    uint16_t algo_type;
    uint16_t flags;
    uint16_t hdr_fmt_idx;
    uint16_t rsvd;
};
```

The **‘*pnso_compression_type*’** is an enum and is defined as follows:

```
enum pnso_compression_type {
    PNSO_COMPRESSION_TYPE_NONE = 0,
    PNSO_COMPRESSION_TYPE_LZRW1A = 1,
    PNSO_COMPRESSION_TYPE_MAX
};
```

This list allows capabilities to be extended with additional crypto types in future releases. For a complete list, please refer to the API Reference Guide. Currently, LZRW1A is the only compression service supported.

Hash Engine

The hash service is defined using the **‘pnso_service’**. Please note that it is a ‘union’, and for the compression accelerator the **‘*pnso_hash_desc*’** is used. The **‘pnso_service’** is defined as follows:

- **svc_type**: specifies one of the enumerated values for the accelerator service type (for hash calculation it would be defined as ***‘pnso_hash_desc’***.
- **rsvd**: specifies a 'reserved' field meant to be used by Pensando.
- ***hash_desc***: struct that specifies the descriptor for data deduplication service.

The other services in this struct are described together with the corresponding accelerator service in this document.

```
struct pnso_service {
    uint16_t svc_type;
    uint16_t rsvd;
    union {
        struct pnso_crypto_desc crypto_desc;
        struct pnso_compression_desc cp_desc;
        struct pnso_decompression_desc dc_desc;
        struct pnso_hash_desc hash_desc;
        struct pnso_checksum_desc chksum_desc;
        struct pnso_decompaction_desc decompact_desc;
    } u;
};
```

The **‘pnso_hash_desc’** is the descriptor for hash calculation operation. It is a struct and defined as follow:

- **algo_type**: Specifies one of the enumerated values of the hash algorithm (i.e. pnso_hash_type) for data deduplication.

- **flags**: specifies the following applicable descriptor flag(s) to hash descriptor:

Flags	Description
PNSO_HASH_DFLAG_PER_BLOCK	Indicates to produce one hash per block. When this flag is not specified, hash for the entire buffer will be produced.

```
struct pnso_hash_desc {
    uint16_t algo_type;
    uint16_t flags;
};
```

The ***pnso_hash_type*** is an enum and is defined as follow:

```
enum pnso_hash_type {
    PNSO_HASH_TYPE_NONE = 0,
    PNSO_HASH_TYPE_SHA2_512 = 1,
    PNSO_HASH_TYPE_SHA2_256 = 2,
    PNSO_HASH_TYPE_MAX
};
```

Checksum Engine

The checksum service is defined using the **'pnso_service'**. Please note that it is a 'union', and for the checksum accelerator the **'*pnso_checksum_desc*'** is used. The **'pnso_service'** is defined as follows:

- **svc_type**: specifies one of the enumerated values for the accelerator service type (for hash calculation it would be defined as ***'pnso_checksum_desc'***.
- **rsvd**: specifies a 'reserved' field meant to be used by Pensando.
- ***chksum_desc***: struct that specifies the descriptor for the checksum calculation service.

The other services in this struct are described together with the corresponding accelerator service in this document.

```
struct pnso_service {
    uint16_t svc_type;
    uint16_t rsvd;
    union {
        struct pnso_crypto_desc crypto_desc;
        struct pnso_compression_desc cp_desc;
        struct pnso_decompression_desc dc_desc;
        struct pnso_hash_desc hash_desc;
        struct pnso_checksum_desc chksum_desc;
        struct pnso_decompaction_desc decompact_desc;
    } u;
};
```

The '***pnso_checksum_desc'*** is the descriptor for checksum calculation operation. It is a struct and defined as follow:

- **algo_type**: Specifies one of the enumerated values of the checksum algorithm (i.e. **pnso_chksum_type**).
- **flags**: Specifies the following applicable descriptor flag(s) to checksum descriptor:

Flags	Description
PNSO_CHKSUM_DFLAG_PER_BLOCK	Indicates to produce one checksum per block. When this flag is not specified, a checksum for the entire buffer will be produced.

```
struct pnso_checksum_desc {
    uint16_t algo_type;
    uint16_t flags;
};
```

Submitting an Offload Service Request

The table below describes the **'pnso_submit_request'** and the required parameters depending on request function:

Param	Type	Sync	Async	Poll	Description
*req	struct pnso_service_request	in	in	in	The set of service request structures to be used to submit the request
*resp	struct pnso_service_result	in/out	in/out	in/out	The set of service result structures to report the status of each service within a request upon its completion
cb_func	typedef completion_cb_t	NULL	valid	optional	The caller-supplied completion callback routine
*cb_ctx	Void *	NULL	valid	optional	The caller-supplied callback context information
*poll_func	typedef *pnso_poll_fn_t	NULL	NULL	valid	The polling function, which the caller will use to poll for completion of the request
**poll_ctx	void **	NULL	NULL	valid	The context to use when calling the polling function

The **'cb_func'** and '***cb_ctx'** are both caller-defined. '**cb_func'** is the function to call upon request completion, and "***cb_ctx**" can be used as the user-supplied context to identify which outstanding request has completed.

Correspondingly, '***poll_func'** and "****poll_ctx**" are both API-defined and opaque from the caller perspective. After submitting a poll request, the caller can poll for completion status by calling the **"*poll_func**" while passing in the **"**poll_ctx**" that corresponds to the given outstanding request.

***Please note:** The caller is responsible for allocation/deallocation of memory for both input and output parameters. Caller should keep the memory intact (ex: req/res) until the Pensando accelerator returns the result via completion callback.*

Access the Result

The **'*pnso_service_result*'** represents the result of the request upon completion for one or all services. It is a struct and is defined as follow:

- **err**: specifies the overall error code of the request. When set to '0', the request processing can be considered successful. Otherwise, one of the services in the request failed, and any output data should be discarded
- **num_services**: specifies the number of services in the request
- **svc**: specifies an array of service status structures to report the status of each service within a request upon its completion

***Please note:** When **'err'** is set to **'0'**, the overall request processing can be considered successful. Otherwise, one of the services in the request is failed, and any output data should be discarded.*

```
struct pnso_service_result {
    pnso_error_t err;
    uint32_t num_services;
    struct pnso_service_status svc[0];
};
```

The "pnso_service_status" represents the result for an individual element within a "pnso_service_result" set. It is a struct and is defined as follows:

- **err**: specifies the overall error code of the request. When set to '0', the request processing can be considered successful. Otherwise, one of the services in the request failed, and any output data should be discarded
- **svc_type**: specifies the service request type, corresponding to one of the "**pnso_service_type"** enum values
- **rsvd_1**: reserved for use by Pensando. Not to be used by caller.
- **u**: descriptor for output/result locations of the service requests. For the compression/decompression offload services (PNSO_SVC_TYPE_COMPRESS or PNSO_SVC_TYPE_DECOMPRESS) the **dst** structure will be used, representing a SGL for the service result set.

Please note: The caller is responsible for allocating all memory that is referenced by the SGL ("**pnso_buffer_list**" and all associated buffers)

```
struct pnso_service_status {
      pnso_error_t err;
      uint16_t svc_type;
      uint16_t rsvd_1;
      union {
            struct {
                  uint16_t num_tags;
                  uint16_t rsvd_2;
                  struct pnso_hash_tag *tags;
            } hash;
            struct {
                  uint16_t num_tags;
                  uint16_t rsvd_3;
                  struct pnso_chksum_tag *tags;
            } chksum;
            struct {
                  uint32_t data_len;
                  struct pnso_buffer_list *sgl;
            } dst;
      } u;
}
```

Coding Guidelines

- Avoid “Synchronous” service requests, except for the most critical meta-data updates that require the strictest serialization.

- Chain service requests whenever possible, rather than manually creating multiple single service request pipelines in software.

- Batched versus Non-Batched requests
    - In general, use of batching and batched requests with larger batch size can increase aggregate throughput. However, use of batched requests and large batch sizes will result in higher request latency. The caller must establish guidelines and policies that are in-line with expected service level requirements.
- Synchronous, Asynchronous and Poll Requests
    - Synchronous requests can be used when the lowest-possible latency is required
    - Asynchronous and Poll can be used when highest-possible throughput is required
- In general, the number of outstanding asynchronous requests per core should not exceed the pnso_init “per_core_qdepth” at any given time
- Input SGL buffers can be in 1 byte increments
- Output SGL buffers must be in “block size” increments, where “block size” corresponds to the “block_size” parameter given in the ***pnso_init_params*** function (e.g. 4096).

Logging

All logging is done through printk() and can be seen through:

- The system console
- syslog
- dmesg

Standard kernel logging levels are provided here for reference:

Name	String	Description
KERN_EMERG	“0”	System is unusable
KERN_ALERT	“1”	Action must be taken immediately
KERN_CRIT	“2”	Critical conditions
KERN_ERR	“3”	Error conditions
KERN_WARNING	“4”	Warning conditions

KERN_NOTICE	“5”	Normal but significant condition
KERN_INFO	“6”	Informational
KERN_DEBUG	“7”	Debug-level messages

Appendix A ： Compiling with COMPAT_LINUXKPI

The SONIC driver requires a FreeBSD-based kernel to be compiled with COMPAT_LINUXKPI. Below are the instructions:

```
# Install git and vim
env ASSUME_ALWAYS_YES=YES pkg install git vim

# Clone FreeBSD source.
git clone http://github.com/freebsd/freebsd /usr/src

# Checkout 11.2 branch
git checkout releng/11.2

# Create User ntpd
echo "test" | pw useradd -n ntpd -m -g wheel -s /sbin/nologin -d
/var/lib/ntpd -h -

# Create Group ntpd
pw groupadd ntpd

# Enable LINUXKPI option
cd /usr/src
echo "options COMPAT_LINUXKPI" >> sys/amd64/conf/GENERIC

# Enable OFED for RDMA
echo "options OFED" >> sys/amd64/conf/GENERIC

# Optional: Enable Journaling and Debugging Support.
echo "options GEOM_JOURNAL" >> sys/amd64/conf/GENERIC
echo "options KDB_UNATTENDED" >> sys/amd64/conf/GENERIC
echo "options KDB" >> sys/amd64/conf/GENERIC
echo "options DDB" >> sys/amd64/conf/GENERIC

# Build and Install the new Kernel
make buildworld buildkernel installworld installkernel
```

```
# Disable PCI ARI
echo hw.pci.enable_ari="0" >> /boot/loader.conf

# Optional: Enable Journaling and Disable background fsck
echo geom_journal_load="YES" >> /etc/rc.conf
echo fsck_y_enable="YES" >> /etc/rc.conf
echo background_fsck="NO" >> /etc/rc.conf

# Reboot
reboot
```