**Virtio-PDCP Security Accelerator**

**Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Version** | **Author** | **Reason** |
| 05/20/2015 | 1 | Freescale Semiconductor | Initial version |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[Table of Contents 3](#_Toc436903089)

[1 Introduction 4](#_Toc436903090)

[2 References 5](#_Toc436903091)

[3 Abbreviations 5](#_Toc436903092)

[4 Scope 6](#_Toc436903093)

[5 Virtio PDCP Security Device 6](#_Toc436903094)

[6 Virtio PDCP Device Definition 8](#_Toc436903095)

[6.1 Bus Details 8](#_Toc436903096)

[6.1.1 Virtio Device ID 8](#_Toc436903097)

[6.1.2 PCI Device Discovery 8](#_Toc436903098)

[6.1.3 Feature Bits 9](#_Toc436903099)

[6.1.4 Virtio PDCP Device Configuration Registers 9](#_Toc436903100)

[6.1.5 Virtqueues 11](#_Toc436903101)

[6.2 Device Emulation 11](#_Toc436903102)

[6.3 Invocation of device in QEMU 12](#_Toc436903103)

[7 API Messages between Virtio PDCP driver (Guest driver) and Vhost-User 14](#_Toc436903104)

[7.1 Command Message Format 14](#_Toc436903105)

[7.2 Queue 14](#_Toc436903106)

[7.3 Vhost-User 15](#_Toc436903107)

[7.3.1 Data Types 17](#_Toc436903108)

[7.3.2 CREATE PDCP SEC CONTEXT 20](#_Toc436903109)

[7.3.3 PROCESS PDCP SEC CONTEXT 21](#_Toc436903110)

[7.3.4 READ PDCP SEC CONTEXT 22](#_Toc436903111)

[7.3.5 DELETE PDCP SEC CONTEXT 23](#_Toc436903112)

[7.4 G API Details 24](#_Toc436903113)

[7.4.1 G Data Types 25](#_Toc436903114)

[7.4.2 CREATE PDCP SEC CONTEXT 30](#_Toc436903115)

[7.4.3 PROCESS PDCP SEC CONTEXT 30](#_Toc436903116)

[7.4.4 GET MAX PDCP VIRTIO QUEUES 32](#_Toc436903117)

[7.4.5 GET VIRTIO QUEUE 32](#_Toc436903118)

[7.4.6 Poll function 32](#_Toc436903119)

[7.4.7 READ PDCP SEC CONTEXT 33](#_Toc436903120)

[7.4.8 DELETE PDCP SEC CONTEXT 34](#_Toc436903121)

[8 Live Migration 34](#_Toc436903122)

[9 PDCP Packet Processing – Look Aside Accelerator Packet Flow 35](#_Toc436903123)

# Introduction

A recent uplink/downlink performance analysis of an LTE protocol stack on a representative virtual mobile platform [[2](http://www.hindawi.com/journals/vlsi/2013/369627/#B2), [3](http://www.hindawi.com/journals/vlsi/2013/369627/#B3)] has identified the Protocol Data Convergence Protocol (PDCP) as the most time-critical component within the Layer 2 software architecture. PDCP incorporates two computationally expensive tasks:

-The ciphering and/or integrity algorithms, responsible for user data protection and for providing a secure communication.

-The Robust Header Compression (ROHC) algorithm, which compresses IP packet headers.

While both protocol functions show long processing times, ciphering & integrity comes in the first place followed by ROHC.

CPU cycles consumed in software based PDCP Security processing envolves:

1. IV construction: Concatenation of negotiated key, HFN|SN, Direction and bearer.
2. Applying crypto algorithm on the PDU using IV computed above.
3. To perform confidentiality and integrity protection requires above 2 steps repeated.

With accelerators offering PDCP Protocol offload can save CPU cycles that makes it worth by boosting performance.

Some of network processors have H/W accelerators supporting PDCP Security acceleration, that can be leveraged to offload the most computationally extensive task i.e. security algorithms for data confidentiality and integrity protection.

On the other hand, virtualization of logically separated layers of LTE stack are in great demands by most of the service providers for well-known advantages of virtualized software.

Given that PDCP security accelerator is available for offload on the host processor, for PDCP running as VNF under guest Virtual Machine (VM), we have a challenge to leverage the accelerator on host processor from VM.

In this document, we are proposing a virtio-pci based PDCP Security driver and device as per the Virtio Standards, so that the VNF can use the Virtio PDCP Security driver to access the PDCP Security Accelerator available on host processor. Using the standard Virtio-pci based Driver the VNF can access any underlying vendor specific PDCP Security Accelerator.

# References

[1] [The 3rd Generation Partnership Project (3GPP)](http://www.3gpp.org)

[2] D. Szczesny, S. Hessel, A. Showk, A. Bilgic, U. Hildebrand, and V. Frascolla, “Performance analysis of LTE protocol processing on an ARM based mobile platform,” in Proceedings of the 11th International Symposium on System-on-Chip (SoC '09), pp. 56–63, Tampere, Finland, October 2009. [View at Publisher](http://dx.doi.org/10.1109/SOCC.2009.5335678)· [View at Google Scholar](http://scholar.google.com/scholar?q=http://dx.doi.org/10.1109/SOCC.2009.5335678) · [View at Scopus](http://www.scopus.com/scopus/inward/record.url?eid=2-s2.0-74549138321&partnerID=K84CvKBR&rel=3.0.0&md5=adcb924743fb3756a8a46ebe2befd992)

[3] D. Szczesny, S. Hessel, A. Showk, A. Bilgic, U. Hildebrand, and V. Frascolla, “Joint uplink and downlink performance profiling of LTE protocol processing on a mobile platform,” International Journal of Embedded and Real-Time Communication Systems, vol. 1, no. 4, pp. 21–39, 2010. [View at Publisher](http://dx.doi.org/10.4018/jertcs.2010100102) ·[View at Google Scholar](http://scholar.google.com/scholar_lookup?title=Joint+uplink+and+downlink+performance+profiling+of+LTE+protocol+processing+on+a+mobile+platform&author=D.+Szczesny&author=S.+Hessel&author=A.+Showk&author=A.+Bilgic&author=U.+Hildebr&author=V.+Frascolla&publication_year=2010)

[4] [Packet Data Convergence Protocol (PDCP ... - 3GPP)](http://www.3gpp.org/dynareport/36323.htm)

Virtio Specifications

[5] <http://docs.oasis-open.org/virtio/virtio/v1.0/virtio-v1.0.pdf>

[6] <http://ozlabs.org/~rusty/virtio-spec/virtio-0.9.5.pdf>

Virtio-net, Vhost-net, Vhost-user implementations in Linux 3.19, Qemu 2.2.0

# Abbreviations

VNF : Virtual Network Function

PDCP : Packet Data Convergence Protocol.

SRB : Signaling Radio Bearer.

DRB : Data Radio Bearer.

VM : Virtual Machine

KVM : Kernel Virtual Machine

MAC-I : Message Authentication Code – Integrity

SN : Sequence Number

PCI : Peripheral Component Interconnect

DL : Downlink

UL : Uplink

LCM : Least Common Multiple

# Scope

This document identifies a Virtio PDCP Security Accelerator which will perform PDCP Record Layer Acceleration. (e.g. Freescale SEC engine). In this case, the Guest VM can push PDCP Security Context into the hardware accelerator. Subsequently when buffers are submitted, the accelerator can perform Encap Security processing (clear packet to encrypted packet and/or add MAC-I for RBs) or Decap Security processing (encrypted packet to clear packet and/or authenticate for RBs) as required. This belongs to Look Aside class of accelerators as, the Guest VM submits packets to the accelerator and receives the processed packet from the accelerator before sending the packet out.

# Virtio PDCP Security Device

HARDWARE

QEMU

Guest VM

Virtio-pdcp FrdFrontend

IOMMU/SMMU

Host Kernel

KVM

Event-fd

PDCP Security Accelerator

Guest User

 Space

Guest Kernel Space

ODP/DPDK based

PDCP-GTPU

Vhost-user pdcp proxy

Memory mapped buffers, PCI space

Virtio-pdcp implementation

Virtio-pci driver

Figure 1: Virtio PDCP Device

Figure 1 shows the Virtio PDCP Device architectural overview. In the Guest VM, the Virtio PDCP Frontend Driver will drive the Virtio PDCP Device. The backend implementation of Virtio PDCP is a Virtio PDCP proxy, which interfaces with the PDCP Security Hardware Accelerator Engine. Virtio PDCP Frontend driver sends PDCP API messages through the Virtio/Vring Descriptors to the Vhost-User PDCP Proxy, which interfaces with the PDCP Hardware Accelerator. The API messages for Virtio PDCP is defined in this document.

# Virtio PDCP Device Definition

A default Virtio PDCP Device is expected to provide the following functionality:

1. Ciphering and/or integrity support for DRB and SRBs
2. COUNT Threshold Notification
3. Long SN Support
4. Asynchronous Security Processing

## Bus Details

The bus specific method used to discover this device is Virtio Over PCI bus. The device implements the following parts as per virtio specification

* Device status Field
* Feature bits
* Device Configuration Space
* One or more Virtqueues

The virtio header in the PCI configuration space shall be in PCI and the device header shall follow the endianness as defined by Virtio Standards.

The Virtqueue information (Vring descriptor, available ring, used ring) shall follow the endianness as defined by the Virtio Standards.

The PDCP commands, responses and buffers shall be in Guest Endian Order. The Guest shall inform the endianness to the Virtio PDCP device using a command.

### Virtio Device ID

The device ID used shall be 21

*Current specification shows allocated device id’s till 18. For experimental drivers, advise in the spec to choose 20 and work backwards.*

### PCI Device Discovery

The vendor ID and device ID used shall be as follows -

PCI Vendor ID – 0x1AF4

PCI Device ID – (0x1040 + 21) = 0x1055

### Feature Bits

An advanced Virtio PDCP device may provide features and functions in addition to a default virtio-pdcp device. In such a case the virtio-pdcp device shall exhibit its features using the Virtio PCI features bits

VIRTIO\_PDCP\_F\_SG\_BUFFERS (1) /\* Packet(s) to be PDCP Encrypted/Decrypted can span multiple buffers and hence multiple descriptors; Only the first buffer needs to have the virtio\_pdcp\_hdr; Rest can have the data \*/

VIRTIO\_PDCP\_F\_DRB (2) /\* Device supports DRB Security offload (7/12/15 bit SN)\*/

VIRTIO\_PDCP\_F\_SRB (3) /\* Device supports SRB Security Offload (5bit SN) \*/

VIRTIO\_PDCP\_COUNT\_THRESHOLD\_NOTIFY (4) /\* Support Notification when COUNT Value Exceeds Threshold \*/

VIRTIO\_PDCP\_REL\_COUNT\_NOTIFY (5) /\* Support Notification when number of buffers processed Exceeds Threshold \*/

VIRTIO\_PDCP\_F\_COUNT\_OVERRIDE (6) /\*Use COUNT provided along with the buffer rather than using HFN in SEC h/w context and SN from packet\*/

VIRTIO\_PDCP\_F\_LONG\_SN\_15BITS (7) /\*SN Size 15 bits supported in addition to 7, and 12 for DRBs and 5 bits for SRB\*/

VIRTIO\_PDCP\_F\_ROHC (8) /\*RoHC support\*/

The PCI feature bits part of Virtio Standards will be supported.

VIRTIO\_RING\_F\_INDIRECT\_DESC 28 /\* Support for indirect descriptor table; Refer to Appendix-B, virtio standards \*/

VIRTIO\_RING\_F\_EVENT\_IDX 29 /\* Usage of avail\_event and used\_event fields; Refer to Appendix-B, virtio-standards \*/

### Virtio PDCP Device Configuration Registers

1. Two 32 bit registers will be used for the same
	1. Register 1 – device queue features;  (Read by guest, write by device)
	2. Register 2 – guest queue features; (Write by guest, read by device)
2. Bit mapping for Register 1 and Register 2
	1. Bit 0-  Bit 15
		1. Should indicate maximum number of DL/UL processing queues.
		2. Maximum possible value = 16k-1 (one for control)
		3. Device will initialize value.
	2. Bit 16 – Bit 21 (Max value 64)
		1. Should indicate number of queues to achieve device side scaling
		2. e.g. if 4 hardware threads of host(proxy) assigned to one Virtual PDCP instance, then this number would read 4]; e.g. if the acceleration proxy has 4 threads, this number would read 4.
		3. Device will initialize value; Guest modifications in Register 2 will be ignored.
	3. Bit 22 – Bit 27 (Max value 64)
		1. Should indicate maximum number of queues to achieve guest side scaling.
		2. For example if Guest has 4 VCPUs assigned, then the value here would be 4].
		3. Device will not initialize a value in Register 1, Guest writes into register 2
		4. Guest will indicate the value
	4. Bit 28 – Bit 31: Reserved for future use.
3. Operation
	1. Qemu side
		1. Initializes the values in Register 1
	2. Guest side (upon probe)
		1. Reads the values in Register 1
		2. Writes back its selection values in Register 2
	3. Guest writes into Register 2 will be ignored or cause a device reset if written into after device is operational.
4. Number of Queues
	1. LCM of Total VM queues (Guest Side Scaling), Device side scaling
	2. E.g.: If there are 4 VCPUs assigned to the VM, the total VM queues = 4; If there are 2 Proxy threads assigned, the total number of Vrings would be LCM of 4, and 2, which is 4.
5. Implicit agreement between Guest driver/device on the ordering of Queues
	1. All data queues [DQs]
		1. For the following parameters, Guest Side Scaling (GS)=x, and Device Side Scaling (DS) = y,
		2. Total queues tq = LCM (x, y)
		3. Device side handling
			1. Each proxy thread or each core would handle tq/y queues
		4. Ordering
			1. For 4 CPUs, assuming the ordering is as follows:
				1. VCPU1 will own queue 1 for Input/Output
				2. VCPU2will own queue 2 for Input/Output
				3. VCPU3will own queue 3 for Input/Output
				4. VCPU4 will own queue 4 for Input/Output
		5. From device perspective
			1. If the device side scaling is 2, then for the above example, Device core 1 will handle ½ the queues (2) and Device core 2 will handle the remaining half.

### Virtqueues

The following queues shall be supported

1. Multiple data Queues will be supported.
	* For a given VM, the number of data Virtqueues calculation would depend on
		+ Number of VCPUs assigned to the VM
		+ Number of Proxy threads available to perform the acceleration operation on host (across VNFs or VMs)
		+ For example, if there are 4 VCPUs assigned to the Guest, 2 threads in Proxy (device backend) to perform acceleration function, the total number of virtqueues would be LCM of (4, 2) = 4

## Device Emulation

The virtio-pdcp device shall use a Vhost-User as a backend. The initialization sequence of the virtio PDCP device would be similar to a vhost-user/virtio-net. To summarize the steps here:

1. The device initialization and setup occurs as described in “Device Initialization” in virtio Spec.
2. Vrings are determined at initialization. The virtio-pdcp driver in Guest VM allocates the vrings and writes the Guest Physical Address corresponding to this page via iowrite to VIRTIO\_PCI\_QUEUE\_PFN.
3. Once “VIRTIO\_CONFIG\_STATUS\_DRIVER\_OK” status bit is set in the status register by the driver, the device is live. At this point, vhost-user backend would be hooked with this device.

## Invocation of device in QEMU



Figure 2: Virtio PDCP Device invocation

Specifying the following in the Qemu command line will add the device:

-device virtio-pdcp-pci,queues=<n>

queues is an optional parameter. By default, 2 queues shall be created.

The full invocation may look like:

# Qemu-kvm ….\

-chardev socket,path=/path/to/socket,id=chr0 \

-object <pdcp-dev>, type=vhost-user, chardev-chr0,id=pdcp0 \

-device virtio-pdcp-pci,pdcp-dev=pdcp0

-object memory-backend-file share=on

Multiple instances of the device virtio-pdcp-pci shall be supported in the QEMU invocation.

# API Messages between Virtio PDCP driver (Guest driver) and Vhost-User

The information sent as structures in the queue shall be packed, using C attribute \_\_packed\_\_.

## Command Message Format

Following the header there will be the descriptors pointing to the input buffer, followed by descriptors pointing to the output buffer.

Desc-1

Desc-2

Desc-3

Desc-4

Desc-5

struct virtio\_pdcp\_hdr

Input Buffer1

Input Buffer 2

Output Buffer 1

Output Buffer 2

Figure 3 Example descriptor chain

Figure 3 shows an example descriptor chain. The first descriptor points to the struct virtio\_pdcp\_hdr that contains the buffer information and results. The next 2 descriptors point to input buffers. The last 2 descriptors point to the Output buffers.

Note: As the descriptor queue size supported is 64k (16 bit space), there is no need to use the Indirect Descriptors. SG descriptors shall be made use of.

## Queue

PDCP driver have single Virtqueue, which handles both Data and Commands to avoid over head of polling on two queues.

##  Vhost-User

In Buffer1

enum **pdcp\_command**

{

 VIRTIO\_PDCP\_CTRL\_ADD\_SEC\_CONTEXT=1,

 VIRTIO\_PDCP\_CTRL\_DEL\_SEC\_CONTEXT,

 VIRTIO\_PDCP\_CTRL\_GET\_SEC\_CONTEXT,

 VIRTIO\_PDCP\_CTRL\_PROCESS\_PKT,

 VIRTIO\_PDCP\_CTRL\_GET\_STATS,

 VIRTIO\_PDCP\_CTRL\_CLEAR\_STATS,

};

struct **virtio\_pdcp\_hdr**{

 enum **pdcp\_command** cmd;

 uint32\_t frontendhandle[PDCP\_FRONTEND\_HANDLE\_SIZE];

 struct **g\_pdcp\_sec\_params** sec\_params;

 union{

 struct pdcp\_sec\_create\_in\_params add;

 struct pdcp\_sec\_delete\_in\_params del;

 struct pdcp\_sec\_get\_in\_params get;

 struct pdcp\_**sec\_process\_in\_params** pkt;

 struct pdcp\_sec\_get\_stats\_in\_params stats;

 struct pdcp\_sec\_clear\_stats\_in\_params stats\_clear;

 }in\_params;

 union {

 struct pdcp\_sec\_create\_out\_params add;

 struct pdcp\_sec\_get\_out\_params get;

 struct pdcp\_sec\_get\_stats\_out\_params stats;

 struct pdcp\_sec\_process\_out\_params pkt;

 }out\_params;

 enum g\_pdcp\_return\_codes status;

};

In Buffer2

struct **g\_pdcp\_sec\_params**

{

 enum g\_pdcp\_auth\_alg auth\_algo;

 uint8\_t auth\_key[AUTH\_KEY\_LEN];

 uint32\_t auth\_key\_len\_bits;

 enum g\_pdcp\_cipher\_alg cipher\_algo;

 uint8\_t cipher\_key[CIPHER\_KEY\_LEN];

 uint32\_t cipher\_key\_len\_bits;

 uint8\_t g\_sn\_size;

 enum g\_pdcp\_pkt\_dir pkt\_dir;

 enum g\_pdcp\_proto\_dir proto\_dir;

 uint32\_t hfn;

 uint32\_t bearerid;

 uint32\_t count\_threshold;

};

In Buffer3

Out Buffer1

Out Buffer2

Out Buffer3

3

struct **pdcp\_sec\_process\_in\_params**

{

 uint32\_t backendhandle[PDCP\_BACKEND\_HANDLE\_SIZE];

 uint32\_t pdcp\_count;

 uint32\_t n\_in\_bufs;

 uint32\_t n\_out\_bufs;

};

Figure 4: Vhost-user messaging

**Virtio PDPC Header**

**I Input + Output buffers**

**Virtio PDPC Header**

**Virtio PDPC Header**

**Virtio PDPC Header**

 CREATE PDCP SEC CONTEXT

 PROCESS PDCP SEC CONTEXT

 READ PDCP SEC CONTEXT

 DELETE PDCP SEC CONTEXT

Refer figure 4.

The **cmd** field in the **virtio\_pdcp\_hdr** identifies the command type (Add/Process/Read/Delete).

* VIRTIO\_PDCP\_CTRL\_ADD\_SEC\_CONTEXT
* VIRTIO\_PDCP\_CTRL\_DEL\_SEC\_CONTEXT
* VIRTIO\_PDCP\_CTRL\_GET\_SEC\_CONTEXT
* VIRTIO\_PDCP\_CTRL\_PROCESS\_PKT

**For Control commands (Add/Read/Delete)**

The first descriptor filled with virtio\_pdcp\_hdr information will be send to the Backend.

**For Data processing**

The data packets are encrypted/decrypted by placing them in the InputQ1…InputQn descriptor followed by the first descriptor ( virtio\_pdcp\_hdr) as shown in Figure3. When data is spread across a SG list, only the first buffer shall have the virtio\_pdcp\_hdr. The rest of the buffers will have only data.

### Data Types

#define PDCP\_FRONTEND\_HANDLE\_SIZE 8

#define PDCP\_BACKEND\_HANDLE\_SIZE 8

#define VIRTIO\_MAX\_RX\_QUEUES 128

#define VIRTIO\_MAX\_TX\_QUEUES 128

#define VIRTIO\_MIN\_RX\_BUFSIZE 64

#define VIRTIO\_MAX\_RX\_PKTLEN 9728

/\* Features desired/implemented by this driver. \*/

#define VIRTIO\_PDCP\_F\_SG\_BUFFERS (1) /\*Packet(s) to be PDCP Encrypted/Decrypted can span multiple buffers and hence multiple descriptors; Only the first buffer needs to have the virtio\_pdcp\_hdr; Rest can have the data\*/

#define VIRTIO\_PDCP\_F\_DRB (2) /\*Device supports DRB Security offload\*/

#define VIRTIO\_PDCP\_F\_SRB (3) /\* Device supports SRB Security Offload including authentication and ciphering \*/

#define VIRTIO\_PDCP\_F\_COUNT\_THRESHOLD\_NOTIFY (4) /\* Support Notification when COUNT Value Exceeds Threshold \*/

#define VIRTIO\_PDCP\_F\_REL\_COUNT\_NOTIFY (5) /\* Support Notification when number of buffers processed Exceeds Threshold \*/

#define VIRTIO\_PDCP\_F\_COUNT\_OVERRIDE (6) /\*Use COUNT provided along with the buffer rather than using HFN in SEC context and SN from packet\*/

#define VIRTIO\_PDCP\_F\_LONG\_SN\_15BITS (7) /\*SN Size 15 bits supported in addition to 7, and 12 for DRBs and 5 bits for SRB\*/

#define VIRTIO\_PDCP\_F\_ROHC (8) /\*RoHC support\*/

#define VIRTIO\_PDCP\_FEATURE\_TO\_BIT(f) (1<<(f-1))

#define VIRTIO\_PDCP\_SNOW\_F8 1<<0

#define VIRTIO\_PDCP\_SNOW\_F9 1<<1

#define VIRTIO\_PDCP\_AES\_ENC 1<<2

#define VIRTIO\_PDCP\_AES\_CMAC 1<<3

#define VIRTIO\_PDCP\_ZUC\_ENC 1<<4

#define VIRTIO\_PDCP\_ZUC\_AUTH 1<<5

#define VIRTIO\_PDCP\_SN\_SZ\_7 1<<6

#define VIRTIO\_PDCP\_SN\_SZ\_12 1<<7

#define VIRTIO\_PDCP\_SN\_SZ\_15 1<<8

#define VIRTIO\_PDCP\_COUNT\_THRESHOLD\_ABS 1<<9

#define VIRTIO\_PDCP\_COUNT\_THRESHOLD\_REL 1<<10

#define VIRTIO\_PDCP\_COUNT\_OVERRIDE 1<<11

struct pdcp\_sec\_process\_in\_params{

 uint32\_t backendhandle[PDCP\_BACKEND\_HANDLE\_SIZE];

 uint32\_t pdcp\_count;

 uint32\_t n\_in\_bufs;

 uint32\_t n\_out\_bufs;

};

struct pdcp\_sec\_process\_out\_params{

 uint32\_t n\_in\_bufs;

 uint32\_t n\_out\_bufs;

};

struct pdcp\_sec\_create\_in\_params{

/\* future use \*/

};

struct pdcp\_sec\_delete\_in\_params{

 uint32\_t backendhandle[PDCP\_BACKEND\_HANDLE\_SIZE];

};

struct pdcp\_sec\_get\_stats\_in\_params{

 uint32\_t backendhandle[PDCP\_BACKEND\_HANDLE\_SIZE];

};

struct pdcp\_sec\_clear\_stats\_in\_params{

 uint32\_t backendhandle[PDCP\_BACKEND\_HANDLE\_SIZE];

};

struct pdcp\_sec\_delete\_out\_params{};

struct pdcp\_sec\_create\_out\_params{

 uint32\_t backendhandle[PDCP\_BACKEND\_HANDLE\_SIZE];

};

struct pdcp\_sec\_get\_in\_params{

 uint32\_t backendhandle[PDCP\_BACKEND\_HANDLE\_SIZE];

 enum g\_pdcp\_sec\_get\_op operation; /\* Get First, Next or Exact \*/

 uint32\_t num; /\* Number of contexts to read \*/

};

struct pdcp\_sec\_get\_out\_params{

 struct g\_pdcp\_sec\_params sec\_params;

 struct g\_pdcp\_sec\_stats stats;

 uint32\_t backendnexthandle[PDCP\_BACKEND\_HANDLE\_SIZE];

};

struct pdcp\_sec\_get\_stats\_out\_params{

 struct g\_pdcp\_sec\_stats stats;

};

enum pdcp\_command

{

 VIRTIO\_PDCP\_CTRL\_ADD\_SEC\_CONTEXT=1,

 VIRTIO\_PDCP\_CTRL\_DEL\_SEC\_CONTEXT,

 VIRTIO\_PDCP\_CTRL\_GET\_SEC\_CONTEXT,

 VIRTIO\_PDCP\_CTRL\_PROCESS\_PKT,

 VIRTIO\_PDCP\_CTRL\_GET\_STATS,

 VIRTIO\_PDCP\_CTRL\_CLEAR\_STATS,

};

struct virtio\_pdcp\_hdr{

 enum pdcp\_command cmd;

 uint32\_t frontendhandle[PDCP\_FRONTEND\_HANDLE\_SIZE];

 struct g\_pdcp\_sec\_params sec\_params;

 union{

 struct pdcp\_sec\_create\_in\_params add;

 struct pdcp\_sec\_delete\_in\_params del;

 struct pdcp\_sec\_get\_in\_params get;

 struct pdcp\_sec\_process\_in\_params pkt;

 struct pdcp\_sec\_get\_stats\_in\_params stats;

 struct pdcp\_sec\_clear\_stats\_in\_params stats\_clear;

 }in\_params;

 union {

 struct pdcp\_sec\_create\_out\_params add;

 struct pdcp\_sec\_get\_out\_params get;

 struct pdcp\_sec\_get\_stats\_out\_params stats;

 struct pdcp\_sec\_process\_out\_params pkt;

 }out\_params;

enum g\_pdcp\_return\_codes status;

};

### [CREATE PDCP SEC CONTEXT](#_Toc435717532)

Virtio backend receives the only one descriptor. Which consist of PDCP header.

The **virtio\_pdcp\_hdr** holds the command type “**VIRTIO\_PDCP\_CTRL\_ADD\_SEC\_CONTEXT** “, **frontendhandle** and security context information (**g\_pdcp\_sec\_params)**.

**struct pdcp\_sec\_create\_out\_params{**

 uint32\_t **backendhandle**[PDCP\_BACKEND\_HANDLE\_SIZE];

};

 VIRTIO\_PDCP\_CTRL\_ADD\_SEC\_CONTEXT

struct **g\_pdcp\_sec\_params**

{

 enum g\_pdcp\_auth\_alg auth\_algo;

 uint8\_t auth\_key[AUTH\_KEY\_LEN];

 uint32\_t auth\_key\_len\_bits;

 enum g\_pdcp\_cipher\_alg cipher\_algo;

 uint8\_t cipher\_key[CIPHER\_KEY\_LEN];

 uint32\_t cipher\_key\_len\_bits;

 uint8\_t g\_sn\_size;

 enum g\_pdcp\_pkt\_dir pkt\_dir;

 enum g\_pdcp\_proto\_dir proto\_dir;

 uint32\_t hfn;

 uint32\_t bearerid;

 uint32\_t count\_threshold;

};

struct **virtio\_pdcp\_hdr**{

 enum **pdcp\_command** cmd;

 uint32\_t **frontendhandle**[PDCP\_FRONTEND\_HANDLE\_SIZE];

 struct **g\_pdcp\_sec\_params** sec\_params;

 union{

 struct pdcp\_sec\_create\_in\_params add;

 struct pdcp\_sec\_delete\_in\_params del;

 struct pdcp\_sec\_get\_in\_params get;

 struct pdcp\_sec\_process\_in\_params pkt;

 struct pdcp\_sec\_get\_stats\_in\_params stats;

 struct pdcp\_sec\_clear\_stats\_in\_params stats\_clear;

 }**in\_params**;

 union {

 struct **pdcp\_sec\_create\_out\_params** add;

 struct pdcp\_sec\_get\_out\_params get;

 struct pdcp\_sec\_get\_stats\_out\_params stats;

 struct pdcp\_sec\_process\_out\_params pkt;

 }**out\_params**;

 enum g\_pdcp\_return\_codes **status**;

};

After successful creation of the security context. Backend fills the backendhandle value to **out\_params.add. backendhandle** and update the **status** G\_PDCP\_SUCCESS (or) G\_PDCP\_FAILURE. Then puts into the used ring.

### [PROCESS PDCP SEC CONTEXT](#_Toc435717533)

Virtio backend receives the chain of descriptor. First one consist of PDCP header, followed by input/output buffers.

* The **virtio\_pdcp\_hdr** holds the command type “**VIRTIO\_PDCP\_CTRL\_PROCESS\_PKT**  and **pdcp\_sec\_process\_in\_params**

Backend gets its pdcp security context handle and process the input and output buffers based on the n\_in\_bufs and n\_out\_bufs value.

**struct pdcp\_sec\_process\_in\_params{**

 uint32\_t **backendhandle**[PDCP\_BACKEND\_HANDLE\_SIZE];

 uint32\_t pdcp\_count;

 uint32\_t n\_in\_bufs;

 uint32\_t n\_out\_bufs;

};

**struct pdcp\_sec\_process\_out\_params{**

 uint32\_t n\_in\_bufs;

 uint32\_t n\_out\_bufs;

};

VIRTIO\_PDCP\_CTRL\_PROCESS\_PKT

struct **virtio\_pdcp\_hdr**{

 enum **pdcp\_command** cmd;

 uint32\_t **frontendhandle**[PDCP\_FRONTEND\_HANDLE\_SIZE];

 struct g\_pdcp\_sec\_params sec\_params;

 union{

 struct pdcp\_sec\_create\_in\_params add;

 struct pdcp\_sec\_delete\_in\_params del;

 struct pdcp\_sec\_get\_in\_params get;

 **struct pdcp\_sec\_process\_in\_params pkt;**

 struct pdcp\_sec\_get\_stats\_in\_params stats;

 struct pdcp\_sec\_clear\_stats\_in\_params stats\_clear;

 }**in\_params**;

 union {

 struct pdcp\_sec\_create\_out\_params add;

 struct pdcp\_sec\_get\_out\_params get;

 struct pdcp\_sec\_get\_stats\_out\_params stats;

 **struct pdcp\_sec\_process\_out\_params pkt;**

 }**out\_params**;

 **enum g\_pdcp\_return\_codes status;**

};

After successful processing, Backend fills the backendhandle value to **out\_params.pkt. n\_in\_bufs** and  **out\_params.pkt.n\_out\_bufs** and update the **status** G\_PDCP\_SUCCESS (or) G\_PDCP\_FAILURE.

The command and input/output buffer descriptors chain are put into the used ring.

### [READ PDCP SEC CONTEXT](#_Toc435717534)

Virtio backend receives the only one descriptor. Which consist of PDCP header.

The **virtio\_pdcp\_hdr** holds the command type “VIRTIO\_PDCP\_CTRL\_GET\_SEC\_CONTEXT“, **pdcp\_sec\_get\_in\_params.**

In case of **G\_PDCP\_SEC\_GET\_FIRST\_N** operation, in\_params.get.backendhandle will be empty.

For **G\_PDCP\_SEC\_GET\_EXACT** and **G\_PDCP\_SEC\_GET\_NEXT\_N**, the in\_params.get.backendhandle holds the corresponding backend handle pointer.

enum g\_pdcp\_sec\_get\_op {

 G\_PDCP\_SEC\_GET\_FIRST\_N = 0,

 G\_PDCP\_SEC\_GET\_NEXT\_N,

 G\_PDCP\_SEC\_GET\_EXACT

};

**struct pdcp\_sec\_get\_out\_params{**

 struct g\_pdcp\_sec\_params sec\_params;

 struct g\_pdcp\_sec\_stats stats;

 uint32\_t backendnexthandle[PDCP\_BACKEND\_HANDLE\_SIZE];

};

**struct pdcp\_sec\_get\_in\_params{**

 uint32\_t backendhandle[PDCP\_BACKEND\_HANDLE\_SIZE];

 enum g\_pdcp\_sec\_get\_op operation; /\* Get First/Next/ Exact \*/

 uint32\_t num; /\* Number of contexts to read \*/

};

VIRTIO\_PDCP\_CTRL\_GET\_SEC\_CONTEXT

struct **virtio\_pdcp\_hdr**{

 enum **pdcp\_command** cmd;

 uint32\_t **frontendhandle**[PDCP\_FRONTEND\_HANDLE\_SIZE];

 struct g\_pdcp\_sec\_params sec\_params;

 union{

 struct pdcp\_sec\_create\_in\_params add;

 struct pdcp\_sec\_delete\_in\_params del;

 **struct pdcp\_sec\_get\_in\_params get;**

 struct pdcp\_sec\_process\_in\_params pkt;

 struct pdcp\_sec\_get\_stats\_in\_params stats;

 struct pdcp\_sec\_clear\_stats\_in\_params stats\_clear;

 }**in\_params**;

 union {

 struct pdcp\_sec\_create\_out\_params add;

 **struct pdcp\_sec\_get\_out\_params get;**

 struct pdcp\_sec\_get\_stats\_out\_params stats;

 struct pdcp\_sec\_process\_out\_params pkt;

 }**out\_params**;

 enum g\_pdcp\_return\_codes **status**;

};

After successful retrieval of the security context. Backend fills the **backendnexthandle** value (in case of **G\_PDCP\_SEC\_GET\_EXACT** and **G\_PDCP\_SEC\_GET\_NEXT\_N**) **g\_pdcp\_sec\_params** and the **status** G\_PDCP\_SUCCESS (or) G\_PDCP\_FAILURE. Then puts into the used ring.

### [DELETE PDCP SEC CONTEXT](#_Toc435717535)

Virtio backend receives the only one descriptor. Which consist of PDCP header.

The **virtio\_pdcp\_hdr** holds the command type “**VIRTIO\_PDCP\_CTRL\_DEL\_SEC\_CONTEXT**“,and **in\_params.del.backendhandle**

**struct pdcp\_sec\_delete\_in\_params{**

 uint32\_t backendhandle[PDCP\_BACKEND\_HANDLE\_SIZE];

};

VIRTIO\_PDCP\_CTRL\_DEL\_SEC\_CONTEXT

struct **virtio\_pdcp\_hdr**{

 enum **pdcp\_command** cmd;

 uint32\_t frontendhandle[PDCP\_FRONTEND\_HANDLE\_SIZE];

 struct g\_pdcp\_sec\_params sec\_params;

 union{

 struct pdcp\_sec\_create\_in\_params add;

 **struct pdcp\_sec\_delete\_in\_params del;**

 struct pdcp\_sec\_get\_in\_params get;

 struct pdcp\_sec\_process\_in\_params pkt;

 struct pdcp\_sec\_get\_stats\_in\_params stats;

 struct pdcp\_sec\_clear\_stats\_in\_params stats\_clear;

 }**in\_params**;

 union {

 struct pdcp\_sec\_create\_out\_params add;

 struct pdcp\_sec\_get\_out\_params get;

 struct pdcp\_sec\_get\_stats\_out\_params stats;

 struct pdcp\_sec\_process\_out\_params pkt;

 }**out\_params**;

 enum g\_pdcp\_return\_codes **status**;

};

After successful Deletion of the security context. Backend fills the **status** G\_PDCP\_SUCCESS (or) G\_PDCP\_FAILURE on the **virtio\_pdcp\_hdr** and puts into the used ring.

## G API Details

**virtio\_pdcp\_process\_cbk()** Receive processed native in/out buffer and it’s count, status.

Poll on specific Virtqueue to receive packets on registered callback function. g\_virtio\_pdcp\_poll(vq1)

**PDCP Application**

Pass security parameters, application handle and packet processing cbk function.

Store the pdcp context handle.

 Send native in/out buffer and it’s count

 Pass pdcp context handle

 to delete.

Check status (success/failure\_

In buf

Out buf

**g\_virtio\_pdcp\_poll**

Poll on selected Virtqueue’s

Used vring

**g\_pdcp\_sec\_create()** – Create virtio\_pdcp\_context\_block, fills the security params & Command type (VIRTIO\_PDCP\_CTRL\_ADD\_SEC\_CONTEXT) into PDCPhdr (virtio\_net\_hdr\_mz). Enqueue to avail vring and notify. call g\_virtio\_pdcp\_poll() & wait for the response

**Frontend** [Guest]

]

**g\_pdcp\_process\_packet()** – Make in/out buffers as single chain of buffers, **VIRTIO\_PDCP\_CTRL\_PROCESS\_PKT**.

 Enqueue to avail vring and notify.

Return to caller,unblocking call.

**g\_pdcp\_sec\_del()** –Pass the backend handle,command type (**VIRTIO\_PDCP\_CTRL\_DEL\_SEC\_CONTEXT)**

Avail vRing

Used vRing

 **Virtqueue - 1**

**Data Virtqeue (s)**

Parse Security params

**VIRTIO\_PDCP\_CTRL\_ADD\_SEC\_CONTEXT** – Create security context, return the backend handle and the status.

Parse the Header for commands

**VIRTIO\_PDCP\_CTRL\_PROCESS\_PKT** – Find the UL/DL sec context. Apply security, send processed out buffers as well as in buffers and status.

**Backend** [Host]

Fetch the backend handle

 **VIRTIO\_PDCP\_CTRL\_DEL\_SEC\_CONTEXT** – Delete the security context. Return the status.

Parse native input buffers

Figure 5: G-API functions

Refer Figure 5 for the G-API functional flows.

All the command G-API (CREATE/READ/DELETE) supports both SYNC/ASYNC modes of operation.

If **G\_PDCP\_CTRL\_FLAG\_ASYNC** flag is set, API call be asynchronous. Otherwise, API call will be synchronous.

### G Data Types

#define PDCP\_FRONTEND\_HANDLE\_SIZE 8

#define PDCP\_BACKEND\_HANDLE\_SIZE 8

#define CIPHER\_KEY\_LEN 16

#define AUTH\_KEY\_LEN 16

enum g\_pdcp\_auth\_alg {

 G\_PDCP\_AUTH\_ALG\_SNOWF9, /\* MD5 HMAC Authentication Algo. \*/

 G\_PDCP\_AUTH\_ALG\_AES, /\* SHA1 HMAC Authentication Algo. \*

 G\_PDCP\_AUTH\_ALG\_ZUC, /\* AES-XCBC Authentication Algo. \*/

 G\_PDCP\_AUTH\_ALG\_NONE, /\* No Authentication \*/

};

enum g\_pdcp\_cipher\_alg {

 G\_PDCP\_CIPHER\_ALG\_SNOWF8,

 G\_PDCP\_CIPHER\_ALG\_AES,

 G\_PDCP\_CIPHER\_ALG\_ZUC,

 G\_PDCP\_CIPHER\_ALG\_NULL /\* NULL Encryption algorithm \*/

};

enum g\_pdcp\_sn\_size

{

 G\_PDCP\_SN\_SIZE\_5=5,

 G\_PDCP\_SN\_SIZE\_7=7,

 G\_PDCP\_SN\_SIZE\_12=12,

 G\_PDCP\_SN\_SIZE\_15=15,

};

enum g\_pdcp\_proto\_dir

{

 G\_PDCP\_ENCAP,

 G\_PDCP\_DECAP

};

enum g\_pdcp\_pkt\_dir

{

 G\_PDCP\_UL,

 G\_PDCP\_DL

};

enum g\_pdcp\_notify\_type

{

G\_PDCP\_COUNT\_THRESHOLD=0,

G\_PDCP\_REL\_COUNT\_THRESOLD,

};

enum g\_pdcp\_control\_flags

{

 G\_PDCP\_CTRL\_FLAG\_ASYNC, /\* If Set, API call be asynchronous. Otherwise, API call will be synchronous \*/

 G\_PDCP\_CTRL\_FLAG\_NO\_RESP\_EXPECTED, /\* If set, no response is expected for this API call \*/

};

Application shall use the above data structure to pass the response requested – async or sync and whether a response is required or not. This structure is a parameter in most of the APIs.

enum g\_pdcp\_sa\_get\_op {

 G\_PDCP\_SEC\_GET\_FIRST\_N = 0,

 G\_PDCP\_SEC\_GET\_NEXT\_N,

 G\_PDCP\_SEC\_GET\_EXACT

};

enum g\_pdcp\_return\_codes

{

 G\_PDCP\_SUCCESS=1,

 G\_PDCP\_FAILURE,

};

struct g\_pdcp\_handle {

 void \*handle;

};

struct g\_pdcp\_sec\_handle {

 void \*handle; /\* context handle \*/

 };

struct g\_pdcp\_auth\_algo\_cap {

 uint32\_t snow\_f9:1,

 aes:1,

 zuc:1,

 none:1

};

struct g\_pdcp\_cipher\_algo\_cap {

 uint32\_t snowf8:1,

 aes:1,

 zuc:1,

 null:1;

};

struct g\_pdcp\_capabilities

{

 uint32\_t pdcp\_features; /\*subset of VIRTIO\_PDCP\_FEATURES negotiated with backend\*/

 struct g\_pdcp\_auth\_algo\_cap auth\_algo\_caps;

 struct g\_pdcp\_cipher\_algo\_cap cipher\_algo\_caps;

}

struct g\_dcp\_cap\_get\_outargs

{

 struct g\_pdcp\_capabilities caps; /\* Capabilities \*/

}

struct g\_pdcp\_sec\_params

{

 enum g\_pdcp\_auth\_alg auth\_algo;

 uint8\_t auth\_key[AUTH\_KEY\_LEN];

 uint32\_t auth\_key\_len\_bits;

 enum g\_pdcp\_cipher\_alg cipher\_algo;

 uint8\_t cipher\_key[CIPHER\_KEY\_LEN];

 uint32\_t cipher\_key\_len\_bits;

 uint8\_t g\_sn\_size;

 enum g\_pdcp\_pkt\_dir pkt\_dir;

 enum g\_pdcp\_proto\_dir proto\_dir;

 uint32\_t hfn;

 uint32\_t bearerid;

 uint32\_t count\_threshold;

}

typedef void(\***g\_pdcp\_add\_context\_resp\_cbfn**)(void \*cb\_arg, struct g\_pdcp\_sec\_add\_outargs);

- The callback function is used to pass response for the PDCP create command (asynchronous mode) from Backend.

typedef void(\***g\_pdcp\_pkt\_process\_resp\_cbfn**)(void \*cb\_arg, void \*pkt\_handle,

 void \*inbuf, void \*outbuf,

 uint32\_t n\_in\_bufs, uint32\_t n\_out\_bufs,

 int32\_t result);

- The callback function is similar to the input arguments, it return the result, no. of input buffer, no. of output buffer, in and out chained buffer pointer.

struct g\_pdcp\_sec\_add\_inargs

{

 enum g\_pdcp\_control\_flags flags;

 struct g\_pdcp\_sec\_params sec\_params;

 void \*cb\_arg; /\*for asynchronous response\*/

 **g\_pdcp\_add\_context\_resp\_cbfn add\_cb\_fn;**

 g\_pdcp\_pkt\_process\_resp\_cbfn pkt\_cb\_fn;

 void \*ctxt\_handle;

};

struct g\_pdcp\_sec\_add\_outargs {

 int32\_t result; /\* Non zero value: Success, Otherwise failure \*/

 struct g\_pdcp\_sec\_handle handle;

}

typedef void(\***g\_pdcp\_del\_context\_resp\_cbfn**)(void \*cb\_arg, struct g\_pdcp\_sec\_del\_outargs

);

- The callback function is used to pass response for the PDCP delete command (asynchronous mode) from Backend.

struct g\_pdcp\_sec\_del\_inargs

{

 enum g\_pdcp\_control\_flags flags;

 struct g\_pdcp\_sec\_handle handle; /\* SA Handle \*/

 void \*cb\_arg; /\*For Asynchronous response\*/

 **g\_pdcp\_del\_context\_resp\_cbfn resp\_fn; /\*** For Asynchronous response **\*/**

};

struct g\_pdcp\_sec\_del\_outargs

{

 int32\_t result;

};

struct g\_pdcp\_sec\_stats {

 uint64\_t packets\_processed; /\* Number of packets processed \*/

 uint64\_t bytes\_processed; /\* Number of bytes processed \*/

uint32\_t no\_tail\_room; /\* Number of packets with no tail room required for padding \*/

uint32\_t submit\_to\_accl\_failed; /\* Number of packets where submission to underlying hardware accelerator failed \*/

uint32\_t auth\_failures;

}

typedef void(\***g\_pdcp\_get\_sec\_resp\_cbfn**)(void \*cb\_arg, struct g\_pdcp\_sec\_get\_outargs);

- The callback function is used to pass response for the PDCP GET command (asynchronous mode) from Backend.

struct g\_pdcp\_sec\_get\_outargs {

{

 int32\_t result; /\* 0: Success: Non zero value: Error code indicating failure \*/

 int32\_t num\_out; /\*Number of output records\*/

 struct g\_pdcp\_sec\_patams \*params; /\* An array of params[] to hold ‘num\_out’ information \*/

 struct g\_pdcp\_sec\_stats \*stats; /\* An array of stats[] to hold the statistics \*/

 g\_pdcp\_sec\_handle \*handle; /\* handle returned to be used for subsequent Get Next N call \*/

};

struct g\_pdcp\_sec\_get\_inargs

{

enum g\_pdcp\_control\_flags flags;

 struct g\_pdcp\_sec\_handle \*handle; /\* Field is not applicable for get\_first \*/

 enum g\_pdcp\_sec\_get\_op operation; /\* Get First, Next or Exact \*/

 struct g\_pdcp\_sec\_params sec\_params;

 void \*cb\_arg; /\*For Asynchronous response\*/

 **g\_pdcp\_get\_sec\_resp\_cbfn resp\_fn; /\*** For Asynchronous response **\*/**

 uint32\_t num; /\* Number of contexts to read \*/

 uint32\_t get\_flags; /\* flags indicate to get complete information or only Statistics \*/

}

struct g\_pdcp\_process\_packet\_inargs {

 enum g\_pdcp\_control\_flags flags;

 struct g\_pdcp\_sec\_handle handle; /\* SA Handle \*/

 uint32\_t n\_in\_bufs;

 uint32\_t n\_out\_bufs;

 uint32\_t pdcp\_count;

 void \*inbuf; /\* Chain of input buffers \*/

 void \*outbuf; /\* Chain of output buffers \*/

 void \*pkt\_opaque;

};

### CREATE PDCP SEC CONTEXT

int32\_t g\_pdcp\_sec\_create(

 const struct g\_pdcp\_sec\_add\_inargs \*in,

 struct g\_pdcp\_sec\_add\_outargs \*out,

 struct g\_pdcp\_resp\_args \*resp);

/\* Function Name: g\_pdcp\_sec\_create

 \*Parameters:

 \*handle: Accelerator handle,

 \*Input Arguments: in {flags,sec\_params, pkt\_cb\_fn, add\_cb\_fn, ctxt\_handle}

 \* flags: Synchronous or asynchronous, Response required or not;

 \* sec\_params: security parameters,

 \* pkt\_cb\_fn: packet processing call back function

\* add\_cb\_fn: call back function when the response is expected asynchronously

 \* Out Argument: out - Result and Security Handle;

 \* resp: Response callback

 \* function and callback argument in case ASYNC response is

 \* requested

 \* Return Value: G\_PDCP\_SUCCESS (or) G\_PDCP\_FAILURE

 \* Description: Application uses this API to create PDCP Security

 \* context

\*/

Application can call this API to create PDCP Security context. This API returns G\_PDCP\_SUCCESS when the context has been successfully (if synchronous response is expected) created by the Virtual Accelerator. A Handle is returned by this API in the response. Application is expected to use this Handle in subsequent calls such as g\_pdcp\_sec\_del , or one of the Read context commands (g\_pdcp\_sec\_get ) or packet processing commands (g\_pdcp\_process\_packet).

This initiates VIRTIO\_PDCP\_CTRL\_ADD\_SEC\_CONTEXT command to backend driver.

The response can be synchronous or asynchronous depending on the flags in input arguments. After enqueue PDCP driver polls for the status message from the Host, updates the backend handle in corresponding context maintained by Frontend driver and return to the application.

### PROCESS PDCP SEC CONTEXT

 int32\_t g\_pdcp\_process\_packet(

struct g\_pdcp\_handle \*handle,

struct g\_pdcp\_process\_packet\_inargs \*in,

 struct g\_pdcp\_resp\_args \*resp);

/\*

 \* Function Name: g\_pdcp\_process\_packet

 \* Input: Accelerator handle,

 \* Input Arguments (

 \* handle – security handle

 \* n\_in\_bufs – no.of in buffers,

 \* n\_out\_bufs – no.of out buffers

 \* pdcp\_count

 \* \*inbuf – Chain of input buffers pointer

 \* \*outbuf - Chain of output buffers pointer

 \* \*pkt\_opaque – opaque pointer

 \* return :result

 \* Success or error code to indicate packet has been submitted

 \* to accelerator or not. Resp: includes the callback function

 \* that will be called on completion of packet processing.

 \*/

Application calls this API for PDCP security processing on the packet. When the application submits the security Handle, and the set of input and output(in resp args) buffers to the virtual accelerator, the application expects the virtual accelerator for PDCP security processing.

The response can be synchronous or asynchronous depending on the flags in input arguments. After enqueue PDCP driver polls for the status message from the Host and return to the application.

If asynchronous response is expected, after enqueue PDCP driver returns the status to the application. Application is expected to poll on the Queue by calling **g\_virtio\_pdcp\_poll** function to receive the processed buffers using the registered callback function during the g\_pdcp\_sec\_create.

### GET MAX PDCP VIRTIO QUEUES

uint16\_t **g\_virtio\_pdcp\_get\_maxq**(void);

This function returns the maximum queue supported by the PDCP device.

### GET VIRTIO QUEUE

void **g\_virtio\_pdcp\_get\_queue**(uint16\_t queue\_id, void \*\*qptr);

This function returns the virtqueue pointer to the qptrs, which corresponds to queue\_id (queue index).

### Poll function

void **g\_virtio\_pdcp\_poll**(void \*vq)

Application calls this API for polling on the used ring. On reception of the packets, the registered packet processing callback invoked. This is non-blocking call, it checks the used ring of the given virtio queue, and if some job is available in the used ring it invokes the corresponding call back function to pass the information back to application. If the response is for create command, it updates the backend handle in corresponding context maintained by Frontend driver.

### READ PDCP SEC CONTEXT

 int32\_t g\_pdcp\_sec\_get(

 struct g\_pdcp\_handle \*handle,

 const struct g\_pdcp\_sec\_get\_inargs \*in,

 struct g\_pdcp\_sec\_get\_outargs \*out,

 struct g\_pdcp\_resp\_args \*resp);

/\* Function Name: g\_pdcp\_sec\_get

 \* Input: Virtual Accelerator Handle (handle/group handle), Input

 \* arguments that includes sec\_handle (valid for get exact or

 \* get next calls) Operation Get First/Get First N/

 \* Get Next/Get Next N/Get Exact/, number of

 \* contexts to read

 \* flags: API control flags, out: contains required

 \* memory to hold the output information,

 \* result: SUCCESS or error code; resp: Optional response

 \* callback function and arguments, in case ASYNC flag is set.

 \* Return Value: Success or Error

 \* Description: Application/Sub-application can call this API to read

 \* Security Information or statistics

 \*/

Application can use this API to retrieve security context or statistics. For convenience several flags are available, such as ‘get first’, get first n number of contexts, get next, get next n number of contexts and get\_exact. Application has the flexibility to get either the security information or the statistics.

This is synchronous call. After enqueue PDCP driver polls for the status message from the Host and return to the application.

### DELETE PDCP SEC CONTEXT

 int32\_t g\_pdcp\_sec\_del(struct g\_pdcp\_handle \*handle,

 const struct g\_pdcp\_sec\_del\_inargs \*in,

 struct g\_pdcp\_sec\_del\_outargs \*out,

 struct g\_pdcp\_resp\_args \*resp);

/\* Function Name: g\_pdcp\_sec\_del

 \* Input: Accelerator Handle, Security context Handle

 \* Input/Output: Success or error code

 \* Description: Given the virtual accelerator handle and security handle, delete the context

 \*/

Application calls this API to delete the PDCP security context the virtual accelerator. This initiates VIRTIO\_PDCP\_CTRL\_ DEL\_SEC\_CONTEXT command to backend driver.

This is synchronous call. After enqueue PDCP driver polls for the status message from the Host and return to the application.

# Live Migration

One of the challenge for Virtio-PDCP device to support live migration of guest, is unavailability of H/W Security context on new host. Following description will provides a solution to this challenge.

On new host, the migrated guest will continue to pass the security parameters and backend handle along with process packet request to backend driver. The backend driver shall check the context in its database by validating the backend handle and if context is not available it can create new context and process the current request. In addition it shall also update the back handle in the command header received along with the process request from frontend.

# PDCP Packet Processing – Look Aside Accelerator Packet Flow

 Host User Space

Hardware

VNF

Virtio-net Frontend

Virtio-net Frontend

Virtio-pdcp Frontend

Guest User Space

ODP based

PDCP

QEMU

vRING

 Transport

Virtio-pdcp Backend

Host Kernel

KVM

vHost-net

PDCP Security Accelerator

NIC

Backplane Processing RLC/MAC/PHY

Backhaul Processing GTPU/IPSec/QoS

EPC

Guest Kernel Space



Figure 6 PDCP Packet Processing –Look Aside Accelerator Flow

Figure 4 shows the flow of packets when PDCP Look aside accelerator is used. This figure shows

Packet flow between EPC – eNodeB - RLC-MAC:

* Packets processed by Vhost-Net
* Packet announced to VNF through Virtio-Net driver
* Packets arrive at the PDCP module for Processing
* As packets are submitted by the PDCP Module to the Virtio-pdcp front end driver, the buffers are put in the Virtio Descriptor Vrings or Virt Qs to be transferred to the Virtio-pdcp Backend.
* The Virtio-pdcp Backend is responsible for translating the packets from Virt Q Descriptor to the actual hardware accelerator in a message that the accelerator understands and vice-versa.
* The Virtio-pdcp Backend is also responsible for picking up processed packets from the hardware accelerator, updating the VirtQ rings and notifying the Guest VNF.
* The processed packets are sent out through the Virtio-net interface for post security processing.