Public Key Accelerator: Users-Guide

Programming Document

Rev 1.3

Document Number: MLNX-15-XXXX

Check into portalx to obtain the doc number!!!!!

www.mellanox.com

Contents

Revision **5**
Purpose **5**
1 Overview **7**
2 Sample application **8**
3 Instance initialization **9**
3.1 Application mode 9
3.2 Enqueue/dequeue synchronization mode 10
3.3 Local initialization 10
3.4 Application termination 10
4 PK commands **11**
5 Result retrieval **11**

List of Figures

Figure 1 PKA Driver Components 8

List of Tables

Table 1: Revision 5
Table 2: Abbreviation 5
Table 3: Related Documentation 6

Revision

Table 1: Revision

Rev	Date	Author	Change Description

Purpose

This document is intended to guide a new PK crypto application developer. It offers programmers with the basic information required to code their own PKA-based application for BlueField.

Scope

This document aims to cover most of the PK software concepts in case customers wish to use it directly. It presents an overview of the PK components, and describes the API calls to guide the software implementation of user application based on the custom API.

Either the hardware specification or the software implementation details are not in the scope of this document.

Definitions/Abbreviation

Table 2: Abbreviation

Definitions/Abbreviation	Description

Related Documentation

This section lists the Related Documentation.

Table 3: Related Documentation

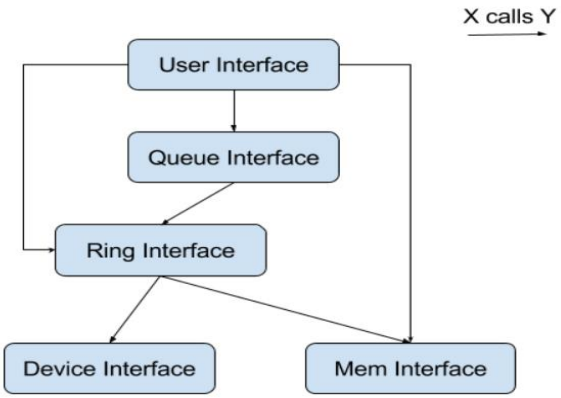
Document Title	Description
PKA Architecture Document	Overview of the architecture, design and implementation of the PK software package for BlueField.

1 Overview

The main goal of the BlueField Public Key Acceleration (PKA) software is to provide a simple, complete framework for crypto PK hardware offload. The PKA software consists of separate but related component parts.

The user API implements an interface that hides the hardware specifics from the applications. It provides direct access to PK hardware resources from user space, and makes available a number of arithmetic (basic operations, e.g., add and multiply, as well as complex operations, e.g., modular exponentiation and modular inversion) and high-level operations such as RSA, Diffie-Hallman, Elliptic Curve Cryptography, and the Federal Digital Signature Algorithm (DSA as documented in FIPS 186) public-private key systems.

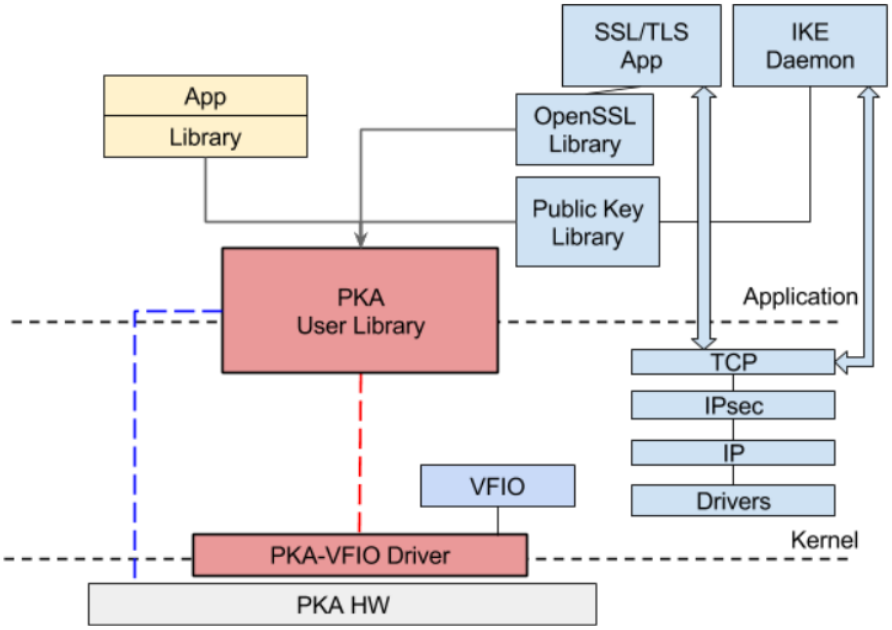
The API enables PK hardware acceleration through internal components, i.e., a set of libraries that provide all the elements needed for handling PK commands, dealing with multithreading, and managing hardware resources.



- **User Interface:** manages user context, wraps up PK commands and performs asynchronous calls.
- **Queue Interface:** leverages resource limitations and enables multithreading environment for hardware offload.
- **Ring Interface:** handles command/result descriptors within hardware rings and implements an interface to the EIP-154 master firmware.
- **Memory Interface:** allocates and frees memory to store and load application data vectors.
- **Device Interface:** abstracts and manages PK hardware resources.

To ensure the development of safe, high performance user-space drivers, a VFIO-based device driver implements an interface that might be supported by virtualization tools and some generic driver infrastructure. It defines a set of devices which are isolatable from all other devices in the system; in other words, it breaks the PK hardware resources among a set of devices that might be allocated to different user driver applications.

Figure 1 PKA Driver Components



2 Sample application

Below, we can see an illustrative example of library calls usage in order to process a PK operation, but do not provide user application function definition (e.g., from_hex_string(), to_hex_string()), and it omits macro definitions and error checking :

```
pka_instance_t    instance;
pka_handle_t      handle;
pka_operand_t     encrypt_key, n, msg;
pka_results_t     results;
uint8_t           result_buf[8];
char             *key_string     = "0x633649F8F2228670";
char             *modulus_string = "0x87C1F8442909789F";
char             *plaintext      = "0x4869207468657265"; // hex "Hi there"
char              ciphertext[20];

results.results[0].buf_ptr = result_buf;
results.results[0].buf_len = sizeof(result_buf);

from_hex_string(key_string,     &encrypt_key);
from_hex_string(modulus_string, &n);
from_hex_string(plaintext,      &msg);

// Global PKA initialization. This function must be called once per instance
// before calling any other PKA API functions.
instance = pka_init_global("pka_app", PKA_F_PROCESS_MODE_SINGLE |
                                      PKA_F_SYNC_MODE_DISABLE,
                                      PKA_RING_CNT, PKA_QUEUE_CNT,
                                      CMD_QUEUE_SIZE, RSLT_QUEUE_SIZE);

// Thread local PKA initialization. The instance parameter specifies which
// PKA instance the thread joins. It returns a valid handle for it.
handle = pka_init_local(instance);

pka_rsa(handle, NULL, &encrypt_key, &n, &msg);
```

```
//
// Here we can do other stuff
//

pka_get_rslt(handle, &results);

//
// Here we can process other PK commands
//

// Release the given handle and PK instance. Note that these calls will free
// rings related to the PK instance and will mark them as available again
pka_term_local(handle);
pka_term_global(instance);

to_hex_string(&results.results[0], ciphertext, 20);
printf("plaintext='%s'\nciphertext='%s'\n", plaintext, ciphertext);

// ciphertext should be '9F5E3B6F177E25B6'
```

3 Instance initialization

All resources must be allocated prior to performing PK operations. Thus, a ***pka_instance_t*** per application should be created. That instance describes the user execution context and holds the global status of assigned PK resources. In order to create an instance, you have to call ***pka_init_global(),*** and provide the instance name (e.g., program name), the number of rings needed by the application to process PK commands, the number and size of queues that will be created on top of the rings. You also have to specify the processing mode required by the application and whether the synchronization mode is enabled.

```
// Global PKA initialization. This function must be called once per instance
// before calling any other PKA API functions.
instance = pka_init_global("pka_app", PKA_F_PROCESS_MODE_SINGLE |
                                      PKA_F_SYNC_MODE_DISABLE,
                                      PKA_RING_CNT, PKA_QUEUE_CNT,
                                      CMD_QUEUE_SIZE, RSLT_QUEUE_SIZE);
```

The call to ***pka_init_global(),*** first, initializes the shared memory, i.e., the pool that is identified by name (usually the same name as the instance), a header and a set of queues to store application requests and replies. The header includes status and meta-data information as well as the list of assigned rings. The function also searches for available rings, i.e., checks whether the requested rings are available and allocates rings to the instance. Once allocated, rings cannot be used by other instances.

3.1 Application mode

Valid processing modes are either *'single-mode'* or *'multi-mode'* that might be selected using **PKA_F_PROCESS_MODE_SINGLE** flag and **PKA_F_PROCESS_MODE_MULTI** flag, respectively. In *'single-mode'*, the PK instance is expected to run over one process with

a single or multiple threads. On the other hand, in *'multi-mode'* the PK instance is expected to run over multiple processes with a single or multiple threads. In this mode, the processes have to share all of the hardware resources that are assigned to the application. The main process ID is stored as global information and might be retrieved when needed to access the instance parameters.

NOTE:

The 'multi-process' mode is not fully implemented. This mode requires additional mechanisms to leverage concurrency and shared resources.

3.2 Enqueue/dequeue synchronization mode

Besides, the library may offer to the application a mechanism to synchronize the accesses to PK rings, i.e., only one thread is allowed to add descriptors to a shared ring at a given time. To enable the synchronization mode, you have to use **PKA_F_SYNC_MODE_ENABLE** flag, otherwise use **PKA_F_SYNC_MODE_DISABLE** flag.

3.3 Local initialization

The ***pka_init_global()*** call returns a pointer to an initialized ***pka_instance_t***. This latter is used to request a ***pka_handle_t***. This handle is mandatory and is used to issue PK commands. Basically, a given instance has a set of running threads referred to as 'workers'; an application running multiple threads has many handles as threads. Typically, those workers run in parallel and can request PK commands anytime. The handle then, gives access to the ***pka_instance_t***. It also holds local information about the running worker such as the identifier, the number of PK pending requests.

To create a ***pka_handle_t***, you have to call ***pka_init_local()*** with a pointer to an initialized instance. The function will then, initialize the handle and will assign to it a request queue and a reply queue from the instance memory pool. Those queues are fully dedicated to the worker requesting the handle.

```
// Thread local PKA initialization. The instance parameter specifies which
// PKA instance the thread joins. It returns a valid handle for it.
handle = pka_init_local(instance);
```

3.4 Application termination

Note that the application must release all of the handles and close the instance before exiting. Otherwise the PK resources cannot be used. Each worker thread have to call ***pka_term_local()*** to release its handle, and finally the function ***pka_term_global()*** is called to free the instance resources.

```
// Release the given handle and PK instance. Note that these calls will free
// rings related to the PK instance and will mark them as available again
pka_term_local(handle);
pka_term_global(instance);
```

4 PK commands

All functions used to perform PK commands are asynchronous and require a *'handle'* argument to be passed to them. In addition to the '*handle'*, the function takes a pointer to user application data that might be associated with the PK operation, e.g., crypto session information.

```
pka_rsa(handle, NULL, &encrypt_key, &n, &msg);
```

For instance, when you wish to perform an RSA operation, you have to call ***pka_rsa()***. The function, first, processes the operands byte order, if needed. It also, apply few operand checks in order to prevent errors in hardware. Note that all of the operands passed via the PKA API calls are copied from the user context to the PKA instance context. After that the command is submitted, then processed.

5 Result retrieval

As mentioned before, all of the PK commands are asynchronous. Thus you have to explicitly call the library to retrieve the results of the previously processed commands. This is done using ***pka_get_result()***. You might also check whether there are available results by calling ***pka_has_avail_result()***. Results are located within the reply queue associated with the given '*handle'*. They are parsed and the result operands are returned to the user application.

```
pka_get_rslt(handle, &results);
```