Elliptic Driver SDK Guide

Document number: EDN-0510 Version: 1.35

Table of Contents

Index of Tables

1 Driver SDK Package

The driver SDK package contains various sub-modules that provide a programmable interface to Elliptic hardware. Each module is designed as a standalone SDK that comes with an example Linux kernel module (some of which are fieldable) to show how to integrate the SDK into a working system. There are modules for each piece of Elliptic hardware, all of which have a hierarchy that depends on how they are instantiated:

- PDU wrapper (provides basic IO to devices, DDT support)
  - SASPA wrapper
  - SPAcc device
  - SPAcc-PDU wrapper
    - SPAcc-EA IPSEC (with working Linux integration)
    - SPAcc-RE TLS/DTLS (with userspace integration)
    - SPAcc-KEP TLS/DTLS key engine
    - SPAcc-MPM packet engine
  - TRNG device
  - PKA device
  - EAPE IPsec device (with Linux integration)

Although devices such as the TRNG or PKA can be instantiated behind a SASPA or SPAcc-PDU wrapper the drivers themselves don't care about this facet. The resources are managed with one of the example PCI or memory based device modules. Located in the *src/pdu/linux/kernel* directory the following files can be used as templates for creating a platform specific device module.

- spacc_pci.c: Provides ability to bring up hardware typically a SPAcc or SPAcc-PDU (and it's related sub-cores), but supports bringing up standalone TRNG/PKA/EAPE devices
- spacc_mem.c: Similar to the PCI module but for direct memory mapped platforms
- saspa_pci.c: Provides ability to bring up SASPA wrapper and it's look-aside cores
- saspa_mem.c: Similar to previous but for direct memory mapped platforms

The device modules are what tell the kernel the resources of the devices. The device drivers themselves then bind to this information to provide an interface to the hardware. In this way the device drivers are platform independent.

1.1 Kernel Versions

Officially the drivers are tested on v3.6.0 and higher Linux kernels. They have been tested on older kernels but are not maintained there. Customers are strongly encouraged to use v3.2.23 or higher.

1.1.1 NIST CMAC Support

Officially as of **v3.7.0** the Linux kernel does not include CMAC support in either the CryptoAPI or IPsec stack. Elliptic has provided a patch in *elliptic/driver/patches/* to provide this to the user. It can be applied to a clean source tree and then configured with **CONFIG_CMAC** (under the Crypto menu).

The ESP/AH drivers will build regardless of this setting but users will not be able to program CMAC based IPsec SAs without it.

There is a document (**0001-Added-NIST-CMAC-to-the-CryptoAPI-library.patch.README**) which explains how to apply the patch to a **v3.7.0** kernel.

1.2 Building the SDK

As shipped the SDK is configured for a PCI lab environment where the *spacc_pci.c* driver looks for devices behind the PCI ID E117:59AE. Users must edit the provided *elliptic/driver/Makefile* file by performing the following steps

1. Comment out the line **PDU_USE_PCI=1** if PCI is not being used
2. (if not PCI) change the line with **PDU_BASE_ADDR** and **PDU_BASE_IRQ** to reflect the base memory address and IRQ of the SPAcc/SPAcc-PDU or SASPA.
    1. By default this is configured for the ARM Versatile Express platform

Next, by default only the SPAcc driver is configured to be built. If the device is a SASPA then comment out the line **ENABLE_SPACC=1** and uncomment the line **ENABLE_SASPA=1**.

If the device is a SPAcc-PDU you can enable the individual cores by uncommenting one (or several) of the following lines:

```
ENABLE_TRNG=1
ENABLE_PKA=1
ENABLE_RE=1  #requires ENABLE_SPACC
ENABLE_MPM=1 #requires ENABLE_SPACC
ENABLE_EA=1  #requires ENABLE_SPACC
ENABLE_EAPE=1 # cannot be enabled with ENABLE_EA=1
```

Note that uncommenting a driver for which there is no hardware will not cause build or runtime problems but will result in extra modules being built that will not provide any use.

If the device is a standalone TRNG, PKA, or EAPE the **ENABLE_SPACC** should still be defined as well as the following:

```
ENABLE_SINGLE_CORE=1
```

and also one of the following:

```
ENABLE_SINGLE_TRNG=1
ENABLE_SINGLE_TRNG3=1
ENABLE_SINGLE_EAPE=1
ENABLE_SINGLE_PKA=1
```

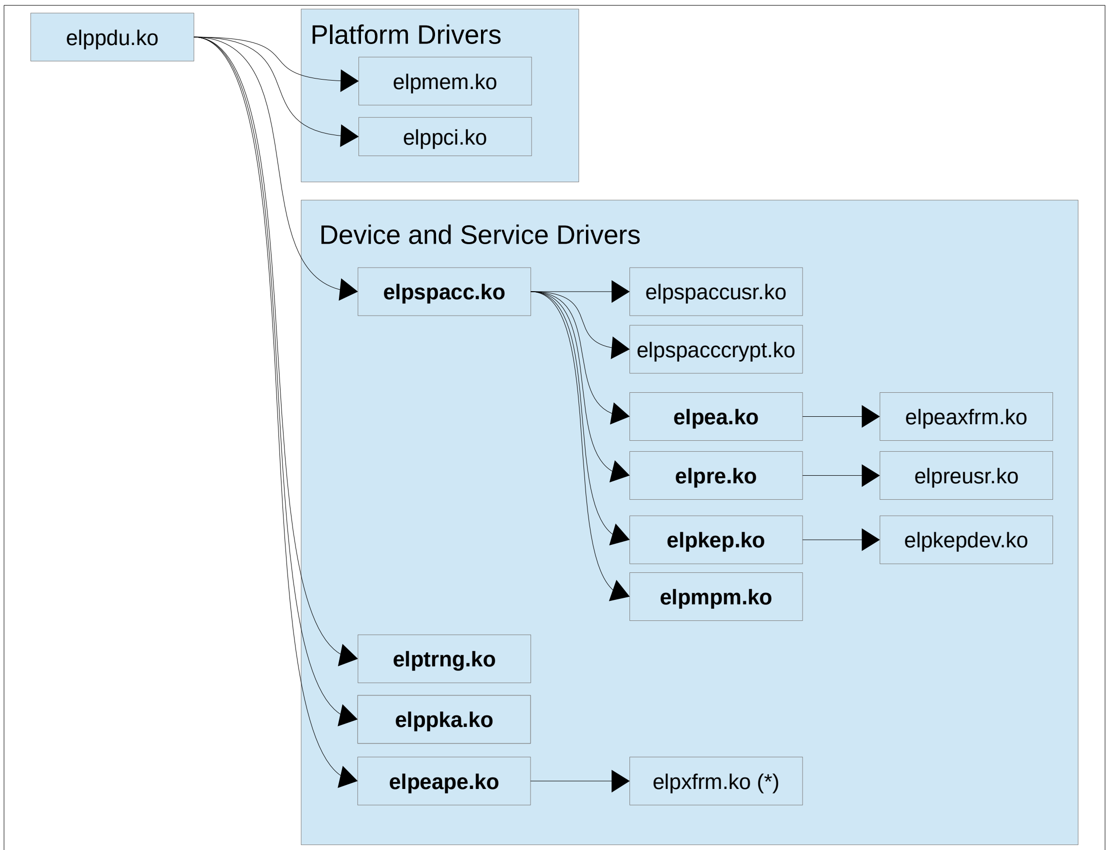
With single core devices the build requires one of the **ENABLE_*** flags to be set. For instance, for a single-core CLP-27 TRNG platform:

```
ENABLE_SPACC=1
ENABLE_TRNG=1
ENABLE_SINGLE_CORE=1
ENABLE_SINGLE_TRNG=1
```

should be set.

1.3 Kernel Modules

The build process produces a series of kernel modules listed below:



Drawing 1: Linux Kernel Modules

At the heart of the system is the *elppdu.ko* module which provides for the generic memory mapped I/O, DDT, locking, and configuration functionality required by the rest of the drivers.

From this the *platform drivers* are built. These drivers instruct the Linux kernel that devices are present at given memory locations with specified IRQ lines. The *elpmem.ko* module is a generic memory based platform driver which uses the environment variables **PDU_BASE_ADDR** and **PDU_BASE_IRQ** to specify to the kernel where the devices are. The *elppci.ko* is a PCI based platform driver that looks for a device with PCI ID E117:59AE to determine the memory address and IRQ. Both modules assume all cores are tied to the same IRQ and both are capable of enumerating a SPAcc or SPAcc-PDU (with EA, RE, TRNG, PKA, KEP, and MPM addons).

The rest of the modules are the *device* or *service* drivers which either add the ability to control and manage the hardware to the Kernel (in the case of device drivers) or add functionality for the user (in the case of service drivers).

- elpspacc.ko: This module provides access to the SPAcc, SPAcc-PDU, and SPAcc-HSM devices to other kernel modules. It handles low level slave register access, managing contexts (allocating/storing/loadings), responding to IRQs and issuing callbacks to the user.
  - elpspaccusr.ko: This module provides the /dev/spaccusr device which userspace tasks can use to perform cipher/hash jobs.
  - elpspacccrypt.ko: This module provides for Linux Kernel CryptoAPI support combined AEAD and individual hash jobs to process IPsec ESP and AH datagrams in the kernel. It is not intended to be a general purpose CryptoAPI driver. The driver supports all of the modes of the SPAcc (with respect to approved IPsec modes) with the following exceptions:
    - Ciphersuites **<u>not</u>** yet supported:
      - AH-GMAC (Linux Kernel does not support this mode as of 3.8-rc3)
      - ESP-CCM
  - elpea.ko: This module provides access to the SPAcc-EA addon (SPP-230/330) for performing IPsec transforms. The module handles context management (via the SPAcc module), responding to IRQs, issuing callbacks, and programming the low level slave ports.
    - elpeaxfrm.ko: This module uses the elpea functionality to implement a protocol driver for AH and ESP inside the kernel. Effectively routing all IPsec traffic through the device.
  - elpre.ko: This module provides access to the SPAcc-RE for performing SSL, TLS, and DTLS record transforms. The module handles context management (via the SPAcc module), responding to IRQs, issuing callbacks, and programming the low level slave ports.
    - elpreusr.ko: This module much like elpspaccusr.ko exports access to the record engine to userspace through the /dev/spaccreusr device.
  - elpkep.ko: This module provides access to the SPAcc-KEP for performing the hash computations required for SSL, TLS, and DTLS key exchange protocols. The module handles context management (via the SPAcc module), responding to IRQs, issuing callbacks, and programming the low level slave ports.
  - elpmpm.ko: This module provides access to the SPAcc-MPM device for performing in-memory queues of SPAcc related jobs. The module handles context management (via the SPAcc module), responding to IRQs, issuing callbacks, and programming the low level slave ports.
- elptrng.ko: This module provides access to the CLP-27 TRNG for random number generation. The module handles responding to IRQs and programming the low level slave ports. If **TRNG_CRYPTO_API** is defined in the top level makefile it will also export the TRNG functionality to the Kernel which will show up to the user as /dev/hwrng which they can read for random bits from the engine.

- elptrng3.ko: This module provides access to the CLP-800 Smart TRNG for random number generation.
- elppka.ko: This module provides access to the PKA for public key acceleration.
- elpeape.ko: This module provides access to the EAPE IPsec engine. The module handles context management (via the SPAcc module), responding to IRQs, issuing callbacks, and programming the low level slave ports.
  - elpxfrm.ko: This module uses the elpeape functionality to implement a protocol driver for AH and ESP inside the kernel. Effectively routing all IPsec traffic through the device. **Note**: That this driver differs from the elpeaxfrm.ko that is built when the SPAcc-EA is enabled as it makes calls to the EAPE SDK instead of the SPAcc-EA SDK.

1.4 Return Codes

All functions that return error codes return an *int*. A negative value indicates error, zero indicates success. Some functions (such as context and job allocation) return a positive integer to indicate the index of the context allocated.

1.5 Application Provided Functions

Some of the cores require functions to be provided to the SDK by the host application. These are for platform specific reasons to allow the SDK greater portability. The list of required functions are**:**

- `PDU_INIT_LOCK()`
- `PDU_LOCK()`
- `PDU_UNLOCK()`
- `pdu_mem_init()`
- `pdu_mem_deinit()`
- `pdu_ddt_init()`
- `pdu_ddt_add()`
- `pdu_ddt_reset()`
- `pdu_ddt_free()`
- `pdu_io_read32()`
- `pdu_io_write32()`
- `pdu_malloc()`
- `pdu_free()`
- `pdu_dma_alloc()`
- `pdu_dma_free()`
- `pdu_sync_single_for_cpu()`
- `pdu_sync_single_for_device()`
- `pdu_get_version()`
- `pdu_error_code()`

1.5.1 Porting Outline

The core of the SDKs (such as src/re/re/, src/ea/ea, src/eape/eape/, src/core/spacc/, etc.) are all relatively portable and dependent on the “pdu” functions to interact with the device. The non-portable platform specific code is placed in the respective Kernel directories (such as src/re/kernel, src/ea/kernel, etc.). These are Linux instantiations of the SDK in a live model (e.g., the devices are accessible on real hardware to the programmer).

A developer porting to a non-Linux platform will want to modify the following two files first.

src/pdu/bare/include/elppdu.h: This file needs to be updated with platform specific types and functions for operations such as locking and printing messages. The developer will want to remove any non-supported header include statements.

src/pdu/bare/pdu/pdu.c: This file has non-portable code at the top half for tasks such as memory based I/O reading and writing, allocating DDT structures, allocating virtual memory, etc.

Once those two are updated each of the individual “core” SDK directories should be useable with a few additions. Users will want to link the contents of src/pdu/bare/pdu/libpdu.a with the core SDK they are using to ensure the pdu functions are available.

To build the SDK libraries ensure that the definition of **PDU_USE_KERNEL** is undefined or removed in the top level makefile. This will build the files under src/pdu/bare/ and also force the individual device SDKs to build as libraries instead of kernel modules. After running “make” all of the library files will be available under bin/.

Building the bare libraries will give the developer access to control the hardware but work is required to put it in a live environment. The Linux kernel modules include additional code such as IRQ handlers and device initialization that cannot be part of a portable library.

After writing the PDU functions required under src/pdu/bare/pdu/pdu.c the developer will next need to:

1. Call the SDK init function (such as spacc_init()) with the appropriate base memory address of the HW registers. This may need to be memory mapped into the virtual address space if the system uses an MMU. Otherwise, it may be the physical address of the hardware.
2. Provide and register an IRQ handler for the OS to handle device interrupts
3. Call the SDK function provided to enable and configure interrupts
4. Integrate calls to the SDK into whatever stack (e.g., IPsec) being used.

1.5.2 Macros

The following macros are found in *elliptic/driver/src/pdu/linux/include/elppdu.h*.

```
#define PDU_MAX_DDT                 64
```

This denotes how many entries a single DDT can contain. By default, there are at most 64 (65 including the terminator entry) which requires 520 bytes of memory. The Linux version of the PDU functions use a DMA pool internally to help cut down on the memory usage. This value should not be set below 16 and generally no higher than 64.

```
#define PDU_DMA_ADDR_T              dma_addr_t
```

This is the type of address used for DMA addresses. Typically, it is merely a 32-bit value but in Linux kernel modules it is *dma_addr_t* and can change size even on 32-bit platforms (for instance, on x86 with PAE enabled).

The PDU implementation must provide functionality for locking to prevent race conditions in the case that multiple calling threads are trying to access the SDKs in parallel. It must define a lock data type, for instance, with Linux it is:

```
#define PDU_LOCK_TYPE                spinlock_t
```

Followed by functions which initialize, lock, and unlock the lock. For instance, with Linux they are:

```
#define PDU_INIT_LOCK(lock)          spin_lock_init(lock)
#define PDU_LOCK(lock, flags)        spin_lock_irqsave(lock, flags)
#define PDU_UNLOCK(lock, flags)      spin_unlock_irqrestore(lock, flags)
```

Where *flags* is an *unsigned long* variable that is presumably written to by the LOCK and read by UNLOCK. If it is not needed in a particular platform it can be ignored.

<u>**Note**</u>: that the SDKs will pass the address of whatever type specified by PDU_LOCK_TYPE to the functions. If they do not expect a pointer the macro should dereference the pointer passed by the SDK.

While these macros are called by the SDK in a general sense they might be part of an interrupt handler. They should map to functions suitable for interrupt handlers.

For non-interrupt based locks there are

```
#define PDU_LOCK_BH(lock,flags)            spin_lock(lock); (void)flags
#define PDU_UNLOCK_BH(lock,flags)          spin_unlock(lock); (void)flags
```

These macros are not meant for interrupt handlers.

1.5.3 DDT Support

The PDU implementation is platform specific but it must have this structure (in *src/pdu/include/elp-pdu.h*)

```
typedef struct {
   PDU_DMA_ADDR_T phys;
   unsigned long *virt;
   unsigned long  idx, limit, len;
} pdu_ddt;
```

The *phys* and *virt* pointers map to the **SAME** 32-bit array (the former being the physical address) that holds the pair of pointer/length DDT values. The *len* parameter holds the length in bytes of the contents pointed to by the DDT table. This structure is managed by the following four functions.

```
int pdu_ddt_init(pdu_ddt *ddt, unsigned long limit);
```

This function must initialize a DDT array with *limit* elements (not including the **NULL** terminator). It must populate the *phys* and *virt* pointers (corresponding to each other) as well as set the *len* to zero. Returns zero if the table has been initialized. In the Linux implementation the 31st bit of *lim-it* is used to indicate whether the driver should use **GFP_ATOMIC** or **GFP_KERNEL** for allocating memory.

```
int pdu_ddt_add(pdu_ddt *ddt, PDU_DMA_ADDR_T phys, unsigned long size);
```

This function must add a known physical address pointed to by *phys* of length *size* bytes to the DDT array, and increment the *len* parameter. The caller must perform any virtual to physical mapping required. Returns zero if the entry was added.

```
int pdu_ddt_reset(pdu_ddt *ddt);
```

This function should reset the DDT table (clear the entries and zero the *len* member).

```
int pdu_ddt_free(pdu_ddt *ddt);
```

This function frees the allocated DDT table.

1.5.4 I/O Support

```
void pdu_io_write32(void *addr, unsigned long val);
```

This function writes a 32-bit word *val* to the address pointed to by *addr*. The address is assumed to be a virtual address (or something the CPU can write to) already mapped to the physical address.

In the Linux kernel this maps to the *writel()* function.

```
unsigned long pdu_io_read32(void *addr);
```

This function reads a 32-bit word from the address pointed to by *addr*. The address is assumed to be a virtual address.

In the Linux kernel this maps to the *readl()* function.

Note: In the Linux versions of these functions writing to address **NULL** enables debugging support which then prints traces to the kernel log when the driver reads/writes from memory. The trace can be disabled by writing to address **NULL** a second time.

1.5.5 Heap Memory Support

```
void *pdu_malloc(unsigned long n);
```

This function allocates *n* bytes of memory (does not have to be page contiguous) and returns the pointer, or **NULL** on error.

In the Linux kernel this maps to the *vmalloc()* function. Note that *kmalloc()* could be used but is not optimal since code that calls *pdu_malloc*() does not require the allocated block to be aligned or even contiguous in physical memory.

```
void pdu_free(void *p);
```

This function frees a block of memory allocated with *pdu_malloc()*.

In the Linux kernel this maps to the *vfree()* function.

1.5.6 DMA Support

```
void pdu_sync_single_for_device(uint32_t addr, uint32_t size);
void pdu_sync_single_for_cpu(uint32_t addr, uint32_t size);
```

These functions take a physical address and size and synchronize the data either off CPU (*for_device*) or in CPU (*for_cpu*). In Linux for instance they map to calls to *dma_sync_single_for_*()*. On memory coherent devices these functions can map to empty functions.

```
void *pdu_dma_alloc(size_t bytes, PDU_DMA_ADDR_T *phys);
void *pdu_dma_free(size_t bytes, void *virt, PDU_DMA_ADDR_T phys);
```

These functions allocate and free DMA coherent memory which does not require manually synchronization calls. It effectively must allocate uncacheable memory. In the Linux kernel this maps to calls to the *dma_alloc_coherent()* functions.

The allocation function must return the *virtual* address (or **NULL** upon error) and store the physical address in *phys*. The free function must be passed the same size block as the allocation call.

1.5.7 Library Support

```
int pdu_error_code(int err);
```

This function maps the SDK error codes into system-specific error codes. In the Linux kernel this produces results such as -EINVAL, -ENOENT, etc. It is acceptable for this function to return its argument unchanged.

1.5.8 SPAcc-PDU Core Description Support

The SPAcc PDU contains registers that describe the PDU and the SPAcc devices. It is stored in the following structure (found in *src/pdu/include/elppdu.h*):

```
typedef struct {
   unsigned minor,
            major,
            qos,
            is_spacc,
            is_pdu,
            is_hsm,
            aux,
            vspacc_idx,
            partial,
            project;
} spacc_version_block;

typedef struct {
   unsigned num_ctx,
            num_rc4_ctx,
            num_vspacc,
            ciph_ctx_page_size,
            hash_ctx_page_size,
            dma_type,
            cmd0_fifo_depth,
            cmd1_fifo_depth,
            cmd2_fifo_depth,
            stat_fifo_depth;
} spacc_config_block;
```

```
typedef struct {
   unsigned minor,
            major,
            is_rng,
            is_pka,
            is_re,
            is_kep,
            is_ea,
            is_mpm;
} pdu_config_block;

typedef struct {
   unsigned minor,
            major,
            paradigm,
            num_ctx,
            ctx_page_size;
} hsm_config_block;

typedef struct {
   spacc_version_block spacc_version;
   spacc_config_block  spacc_config;
   pdu_config_block    pdu_config;
   hsm_config_block    hsm_config;
} pdu_info;
```

See the SPAcc-PDU User Guide documentation for more information on these fields.

```
int pdu_get_version(void *dev, pdu_info *inf);
```

This function fills out a *pdu_info* structure with the virtual memory pointer to the SPAcc-PDU core *dev*.

1.6 Offload Demos

The SPAcc and related cores all provide offload demos used to stress test the cores in a real system (running Linux). The demos are provided by the following modules

- mpmtraffic.ko, SPAcc-MPM
- elpeaoff.ko, SPAcc-EA
- elpredev.ko, SPAcc-RE

Along with the kernel modules are userspace applications intended to drive them. The EA and MPM cores are driven by the application *src/test/offload/offdemo.c* which is buildable as a stand alone C application. The RE is tested with *src/re/user/reuserspd.c*.

1.6.1 SPacc-MPM/SPAcc-EA Testing

These use the *offdemo* application to drive the cores respective kernel modules. The syntax of the command is

```
offdemo --device /dev/devicename [--size packetsize --cipher algo --ciphermode mode --hash algo
--hashmode mode --ciphersize keysize --hashsize keysize --epn epn]
```

Parameter	**Values**	**Parameter**	**Values**
size	1..16384	*cipher*	• null • des • aes • rc4 • multi2 • snow3g • zuc
ciphermode	• ecb • cbc • ctr • ccm • gcm • ofb • cfb • f8 • xts	*hash*	• null • md5 • sha1 • sha224 • sha256 • sha384 • sha512 • xcbc • cmac • kf9 • snow3g • crc32 • zuc • sha512_224 • sha512_256
hashmode	• raw • sslmac • hmac		

Table 1: offdemo command line options

For instance, to test AES128CBC + SHA1HMAC on the SPAcc with an EPN of 1800 the command would be

```
offdemo --device /dev/spacc --size 2048 --cipher aes --ciphermode cbc --hash sha1 --hashmode hmac \
--epn 0x1800
```

Which will then output a line resembling

```
3.53797,0,0.48,13.5671,151.745,9261.8,682.667
```

Which is the comma separate format of the following data

Wall time (sec)	**User time (sec)**	**System Time (sec)**	**CPU %**	**Mbit/sec**	**Jobs/sec**	**Jobs per 1% of CPU**
3.53797	0	0.48	13.5671	151.745	9261.8	682.667

Table 2: Example readout of offdemo tool

What this tells us is that the CPU was busy 13.56% of the time over 3.53 seconds. We processed 151 megabits of traffic or the equivalent of 9261 jobs per second which accounts for 682 jobs per percent of CPU consumed.

This data can then be read in other manners. For instance, this was over a 400MHz ARM A9 processor so 1% of CPU is 4 million cycles. 682 jobs per 1% amounts to 5865 cycles per job (where job in this case is a 2048-byte buffer) or 2.8 bits per ARM cycle processed (compared to the 0.112 bits per cycle native software would get on this architecture).

The SPAcc-EA can be tested in the same style except that the device is called */dev/spaccea*. Only valid ESP combinations of ciphers and hashes are allowed regardless of what the underlying SPAcc may support.

The SPAcc-MPM can be tested similarly with the device */dev/spaccmpm*. The MPM case has an additional parameter that can be optionally provided

```
--mpmchainsize size
```

Where the MPM chain size can be from 1 to whatever the elpmpm.ko module was loaded with for a max chain size (default is 32). The default (if omitted) is to use maximum size chains.

1.6.2 SPAcc-RE Testing

The SPAcc-RE testing emits data that is equivalent to the *offdemo* output but is a separate command because it works slightly differently. In the RE case we are feeding traffic through the userspace API to the kernel and reading it back. This is a realistic example of userspace traffic going through the SPAcc-RE device.

```
reuserspd [--size recordsize --cipher algo --hash algo --runs numberoftrials]
```

Parameter	Values
size	1...16384
cipher	• null • aes128cbc • aes256cbc • aes128gcm • aes256gcm • descbc • 3descbc • rc440 • rc4128
hash	• null • md5 • md580 • sha1 • sha180 • sha256

Table 3: reuserspd command line options

The defaults are for a 4096-byte record, 8192 runs, with AES128CBC+SHA1HMAC which is a fairly common ciphersuite. This command tries to open */dev/spaccreusr* which means the user running the command must have permissions to use it.

1.6.3 SPAcc Userspace Profiling

The SPAcc profiling uses the userspace driver to benchmark a realistic path from a userspace application through the kernel and device and back again. The tool is called *spaccdevoff* and is built by default when the SPAcc module is built.

```
spaccdevoff [--size jobsize --cipher ciphermode --ciphersize keysize --hash hashmode --hashsize key-
size --runs count --threads threadcount --inplace]
```

The cipher and hash algorithm choices are the same as in the RE benchmark program. The output style is essentially the same as all of the other tools with the exception that the first column is the number of threads being used.

2 SPAcc

2.1 Device Initialization

The SPAcc library operates on a *spacc_device* device context which is mapped to a single SPAcc core (whether a native SPAcc device or a virtual-SPAcc interface).

```
int spacc_init(void *baseaddr, spacc_device *spacc, pdu_info *info);
```

This will initialize the device based on the given base address in virtual memory *baseaddr*. It requires the PDU device information to be filled out in *info* with a prior call *to pdu_get_version()*. After this call there will be allocated memory inside the spacc structure which must be freed with a call to *spacc_fini()*.

2.2 Job Notification

2.2.1 IRQ Support

An IRQ handler for the SPAcc need only call *spacc_process_irq()*. Ideally, the call could take place from a tasklet (a non-hard IRQ context) but ideally should be quick enough to be called from a hard interrupt context. This function call requires that the user initialize the call back functions for the various interrupts (see the description of spacc_process_irq() later in the document).

The Linux kernel module by default enables a STAT_WD and STAT interrupt and operates in IRQ mode which means it is primarily driven by IRQs to collect terminated jobs. This is accomplished in the Linux kernel module with the following code:

```
/* register irq callback function */
priv->spacc.irq_cb_stat    = spacc_stat_process;
priv->spacc.irq_cb_stat_wd = spacc_stat_process;

/* set threshold and enable irq */
spacc_irq_stat_enable (&priv->spacc, priv->spacc.config.ideal_stat_level);
spacc_irq_stat_wd_enable (&priv->spacc);
spacc_irq_glbl_enable (&priv->spacc);

/* enable the wd */
spacc_set_wd_count(&priv->spacc, 250000); // 1msec @250MHz, adjust accordingly
```

This assigns the same callback function for both types of IRQ, the callback function simply schedules a soft-IRQ (outside the IRQ scope) call to *spacc_pop_packets()* which dequeues jobs off the STAT FIFO. This initialization code also assigns a STAT CNT value to trigger the STAT IRQ on when the STAT FIFO is filling up. Ideally, we don't want to wait for the FIFO to fill completely as it will stall the SPAcc but we don't want the value too low as to not get the benefit of a longer FIFO.

Finally, this code initializes the STAT and STAT_WD interrupts and then enables the global interrupts on the SPAcc device.

2.2.2 Watchdog Support

The default IRQ model for the SPAcc SDK is to use the STAT_WD and STAT interrupts to catch high volume traffic and occasional bursts. The STAT interrupt will fire when the STAT FIFO fills up (programmed with a STAT_CNT just below the maximum) and the STAT_WD interrupt will fire when the device idles for a given amount of time. The delay can be specified with this function.

```
int spacc_set_wd_count(spacc_device *spacc, uint32_t val);
```

This will set the SPAcc watchdog timer to the value specified by *val* it is indicated in core clock cycles. For example, with a 50MHz device a value of 50,000 indicates 1 millisecond.

As described in **EDN-0228** the watchdog works as follows:

1. A write to the CMD0 FIFO resets the timer value
2. Once the STAT FIFO has at least one entry the timer starts running
3. The timer stops running when the STAT FIFO empties.

When the timer matches the value programmed it will fire the STAT_WD interrupt. To enable the WD interrupt the function:

```
void spacc_irq_stat_wd_enable (spacc_device *spacc);
```

Can be called. This turns on the bit in the IRQ_EN register which allows STAT_WD interrupts to occur. They can be disabled with

```
void spacc_irq_stat_wd_disable (spacc_device *spacc);
```

When the STAT_WD interrupt fires the user should acknowledge the interrupt and then pull jobs off the STAT FIFO. The value programmed in the timer should be large enough such that the CPU can pull jobs off otherwise subsequent STAT_WD interrupts will be raised possibly causing a deadlock. A workaround is to disable the STAT_WD interrupt in your IRQ handler and then re-enable it from your soft-IRQ (equivalent) task after calling *spacc_pop_packets()*.

2.2.3 IRQ Management (Linux only)

For Linux users of the SPAcc driver there is a tool *bin/spaccirq* which allows root users to alter how the SPAcc IRQ system works. The command has the following options:

```
spaccirq: --irq_mode wd|step --epn 0x???? [--virt ??] [--wd nnnn] [--stat nnnn] [--cmd nnnn]
   --irq_mode  wd==watchdog, step==stepping IRQ
   --epn       EPN of SPAcc to change (in hex, e.g. 0x0605)
   --virt      Virtual SPAcc to change (default 0)
   --wd        Watch dog counter (in cycles) good values are typically >15000
   --stat      STAT_CNT IRQ trigger, see the print out from loading elpspacc.ko to
               see the size of the STAT FIFO
   --cmd       CMD_CNT IRQ trigger (for CMD0),  see the print out from loading
               elpspacc.ko to see the size of the CMD FIFO
```

All of the options default to zero which inside the kernel means use the defaults.

The IRQ modes are either “wd” which means to use the watchdog timer (users should specify --wd) or "step" which means to use the stepping CMD/STAT IRQ system (users should specify --stat and --cmd). The user **must** specify an --irq_mode option in this command.

The SPAcc device they which to manipulate is specified by --epn for the EPN and optionally --virt for a specific virtual SPAcc. The user **must** specify an --epn option in this command.

The stepping IRQ method uses the --cmd and --stat options to specify the CMD and STAT trigger levels. The values should be zero or higher and less than the total length of the FIFOs. When users load the *elpspacc.ko* module the kernel log will have detailed information about the FIFO depth of the core.

The watchdog IRQ method uses the --wd option which specifies the watchdog count. It is measured in cycles and should be high enough to prevent too many IRQs from triggering.

Examples:

- Setting watchdog mode with a count of 50000 cycles
  - `bin/spaccirq --epn 0x1234 --irq_mode wd --wd 50000`
- Setting stepping mode for reset defaults
  - `bin/spaccirq --epn 0x1234 --irq_mode step --cmd 0 --stat 1`

2.3 *Device Termination*

To clean up the memory allocated for a SPAcc device a call to the *spacc_fini()* function is needed.

```
void spacc_fini(spacc_device *spacc);
```

This will free up memory allocated for context and job management.

2.4 Context Manager

The SPAcc contains a context manager that arbitrates contexts between the SPAcc, RE, and KEP. It has a simple interface to request and release contexts. A number of contiguous contexts can be requested up to the maximum number of contexts in the engine. The RE/KEP must use virtual SPAcc 0 if more than one is enabled. The SPAcc SDK functions call these and are not required to be used externally. The functions are exposed for RE/KEP use.

2.4.1 spacc_ctx_init_all()

This function initializes the memory pointers internal in the hardware and resets the internal reference counts. This is called internally from spacc_open, but can also be called to reset context state at a later time.

```
void spacc_ctx_init_all (spacc_device *spacc);
```

2.4.2 spacc_ctx_request()

The caller to this function can specify an exact *ctx_idx* to use or pass -1 to request an unused one. The *ncontig* parameter is used to allocate a specific number of contiguous contexts. Generally a value of "1" is used. This function returns a context_index from 0 to the maximum number of contexts in the hardware, or a negative value if one is not found. Contexts can be reused, but it is up to the calling application to be aware of changing state.

```
int spacc_ctx_request (spacc_device *dev, int ctx_idx, int ncontig);
```

2.4.3 spacc_ctx_release()

This function releases a specific context for use. It will return a negative number if the ctx_idx is invalid and 0 if successful. The data contained in the context is not cleared. It is up to the application to clear this memory if required.

```
int spacc_ctx_release (spacc_device *dev, int ctx_idx);
```

2.5 Job Programming

Note: There are example job programming routines in *src/core/kernel/example.c* that demonstrate how to program simple cipher and hash jobs. Please consult that file if you need help using the SDK.

Programming a job with the SPAcc SDK always follows these general steps:

1. Open a SPAcc job index with *spacc_open()* passing it which cipher and hash modes are desired.
2. Calls to *spacc_write_context()* to set the hash and/or cipher context (keys, IV, salt).
    1. *optionally*, call *spacc_load_skp()* instead to load a key through the secure key port.
3. Optionally call *spacc_set_key_exp*() to either initialize the key or pre-compute the decrypt key (see the section for this function for more details).
4. Call to *spacc_set_operation()* to indicate which direction (encrypt/decrypt), ICV mode, ICV position method (append, offset, etc), and whether secure key port is used.
5. Construction of a *pdu_ddt* array for both input and output buffers (see the section about PDU)
6. Call to *spacc_packet_enqueue_ddt()* to push the job onto the command FIFO.
7. An interrupt should be tied to spacc_pop_packets() but that function may be polled as well. The calling application should poll *spacc_packet_dequeue()*.

Steps 1-4 may be omitted for repeated jobs, for example, processing bulk data through CBC mode, or partial processing support.

2.5.1 spacc_open()

The *spacc_open()* function allocates a single context from a given SPAcc device and assigns the cipher and hash modes.

```
int spacc_open (spacc_device *spacc, int enc, int hash, int ctx, int secure_mode,
                spacc_callback cb, void *cbdata);
```

Where *enc* is one of the enumerated cipher modes, *hash* is one of the enumerated hash modes, *ctxid* is used for resource allocation strategies (use -1 for a new context or pass a known one), and *secure_mode* indicates whether this job is executing in secure mode or not. Secure jobs use a secure key context and require the key to be loaded via *spacc_load_skp()*.

The callback *cb* is called when the job is popped off the status FIFO. It may be set to **NULL** to indicate no callback is desired. The form of the callback is:

```
typedef void (*spacc_callback)(void *spacc_dev, void *data);
```

Where *spacc_dev* is the *spacc_device* passed by the caller and *cbdata* is passed as *data*. This callback mechanism allows a user thread waiting on the job to be unblocked. **NOTE:** This callback may be called from an interrupt context and must be IRQ safe.

The allocated job is setup by default to run in complete packet mode, that is the message begin and end flags will be set. In order to run jobs in partial processing mode the caller must clear these flags and set them as appropriate. See *src/core/kernel/spaccdiag.c* for an example FSM that executes partial processing mode.

The supported cipher modes are as follows:

Encryption Enumeration	**Mode Description**
CRYPTO_MODE_NULL	NULL cipher
CRYPTO_MODE_RC4_40	RC4 40-bit key
CRYPTO_MODE_RC4_128	RC4 128-bit key
CRYPTO_MODE_RC4_KS	RC4 and the RC4 state is manually loaded in the RC4 key context (this allows non-standard key size support)
CRYPTO_MODE_AES_ECB	AES ECB mode
CRYPTO_MODE_AES_CBC	AES CBC mode
CRYPTO_MODE_AES_CTR	AES CTR mode
CRYPTO_MODE_AES_CCM	AES CCM mode
CRYPTO_MODE_AES_GCM	AES GCM mode (96-bit IV)
CRYPTO_MODE_AES_F8	AES F8 mode
CRYPTO_MODE_AES_XTS	AES XTS mode (with or without CTS)
CRYPTO_MODE_AES_CFB	AES CFB (s=128) mode
CRYPTO_MODE_AES_OFB	AES OFB (s=128) mode
CRYPTO_MODE_AES_CS1	NIST CBC-CS mode 1
CRYPTO_MODE_AES_CS2	NIST CBC-CS mode 2
CRYPTO_MODE_AES_CS3	NIST CBC-CS mode 3
CRYPTO_MODE_MULTI2_ECB	MULTI2 ECB mode
CRYPTO_MODE_MULTI2_CBC	MULTI2 CBC mode
CRYPTO_MODE_MULTI2_OFB	MULTI2 OFB mode
CRYPTO_MODE_MULTI2_CFB	MULTI2 CFB mode
CRYPTO_MODE_3DES_CBC	Triple DES CBC mode
CRYPTO_MODE_3DES_ECB	Triple DES ECB mode
CRYPTO_MODE_DES_CBC	DES CBC mode
CRYPTO_MODE_DES_ECB	DES ECB mode
CRYPTO_MODE_KASUMI_ECB	KASUMI ECB mode
CRYPTO_MODE_KASUMI_F8	KASUMI F8 mode
CRYPTO_MODE_SNOW3G_UEA2	SNOW3G UEA2 encryption mode
CRYPTO_MODE_ZUC_UEA3	ZUC UEA3 encryption mode

Table 4: SPAcc Encryption Modes

The supported hash modes are as follows:

Hash Enumeration	Mode Description
CRYPTO_MODE_HASH_MD5	MD5 hash mode
CRYPTO_MODE_HMAC_MD5	MD5 HMAC mode
CRYPTO_MODE_HASH_SHA1	SHA-1 hash mode
CRYPTO_MODE_HMAC_SHA1	SHA-1 HMAC mode
CRYPTO_MODE_HASH_SHA224	SHA-224 hash mode (not to be confused with SHA-512_224)
CRYPTO_MODE_HMAC_SHA224	SHA-224 HMAC mode
CRYPTO_MODE_HASH_SHA256	SHA-256 hash mode
CRYPTO_MODE_HMAC_SHA256	SHA-256 HMAC mode
CRYPTO_MODE_HASH_SHA384	SHA-384 hash mode
CRYPTO_MODE_HMAC_SHA384	SHA-384 HMAC mode
CRYPTO_MODE_HASH_SHA512	SHA-512 hash mode
CRYPTO_MODE_HMAC_SHA512	SHA-512 HMAC mode
CRYPTO_MODE_HASH_SHA512_224	SHA-512_224 hash mode
CRYPTO_MODE_HMAC_SHA512_224	SHA-512_224 HMAC mode
CRYPTO_MODE_HASH_SHA512_256	SHA-512_256 hash mode
CRYPTO_MODE_HMAC_SHA512_256	SHA-512_256 HMAC mode
CRYPTO_MODE_MAC_XCBC	AES XCBC MAC mode
CRYPTO_MODE_MAC_CMAC	AES CMAC MAC mode
CRYPTO_MODE_MAC_KASUMI_F9	KASUMI F9 MAC mode
CRYPTO_MODE_MAC_SNOW3G_UIA2	SNOW3G UIA2 MAC mode
CRYPTO_MODE_MAC_ZUC_UIA3	ZUC UIA3 MAC mode
CRYPTO_MODE_SSLMAC_MD5	SSL 3.0 MAC based on MD5
CRYPTO_MODE_SSLMAC_SHA1	SSL 3.0 MAC based on SHA1
CRYPTO_MODE_HASH_CRC32	CRC32 HASH (not cryptographically secure)

Table 5: SPAcc Hash Modes

2.5.2 spacc_write_context()

Once a SPAcc job index has been acquired that maps to a context on a device the keys (and other salient pieces of data such as IV and salt) can be loaded with the *spacc_write_context()* function.

```
int spacc_write_context (spacc_device *spacc, int handle, int op,
                       unsigned char * key, int ksz, unsigned char * iv, int ivsz);
```

The function assigns values to the context identified by *job_idx* and operates on the cipher (and RC4) or hash context pages depending on the value of *op*. When set to **SPACC_CRYPTO_OP-**

ERATION it sets the cipher/RC4 pages, when set to **SPACC_HASH_OPERATION** it sets the hash pages.

The key is passed as *key* of *ksz* bytes length, and the IV/salt as *iv* of *ivsz* bytes length. In certain modes the lengths are implicit. See the **EDN-0228 SPAcc User Guide** for more details.

When calling this function to load a fully scheduled RC4 key the key (cipher mode **CRYPTO_MODE_RC4_KS**) is assumed to be 258 bytes long. The first two bytes are *i* and *j* respectfully (see *spacc_write_rc4_context*()) and the remaining 256 bytes are the shuffle array. The calling application is responsible for scheduling the key as the SDK does not contain any cryptographic software.

In typical combined modes the caller must call this function twice to initialize the two different context pages. In some cases it may be useful to read back the context data. The function *spacc_read_context()* should be used.

```
int spacc_read_context (spacc_device *spacc, int job_idx, int op,
                        unsigned char *key, int ksz, unsigned char *iv, int ivsz);
```

Reading back the context allows for device context swapping. The typical recycling of a context would be:

1. Call *spacc_open*()
2. Call *spacc_write_context*() once or twice depending on the job
3. Call *spacc_set_operation*()
4. Perform SPAcc jobs
5. Call *spacc_read_context*()
6. Call *spacc_close*()
7. Start a new job doing steps 1-6
8. When you want to resume the first context simply write back the saved keys/ivs

2.5.3 spacc_load_skp()

To use the secure key port functionality of the SPAcc the key must be loaded with the *spacc_load_skp()* function.

```
int spacc_load_skp(spacc_device *spacc, uint32_t *key, int keysz, int idx, int alg,
                   int mode, int size, int enc, int dec);
```

This loads a key of *keysz* 32-bit words pointed to by *key* into the secure context indexed by *idx*. The *alg* and *mode* parameters are one of the mode enumerations for the SPAcc hardware listed in section 5.1 of EDN-0228 under the **CTRL** register. The *size* register is either 0, 1, or 2 for 64/128, 192, and 256-bit keys respectively. The *enc* and *dec* registers indicate whether the context can be used for encryption and decryption respectfully.

2.5.4 spacc_set_operation()

Once a SPAcc context has been allocated the operating mode can be assigned with the *spacc_set_operation()* function.

```
int spacc_set_operation (spacc_device *spacc, int job_idx, int op, uint32_t prot, uint32_t icvcmd,
                         uint32_t icvoff, uint32_t icvsz, uint32_t sec_key);
```

This function sets the encryption direction, ICV protocol, position, offset, and size. The *op* parameter should be set to **OP_ENCRYPT** for encryption or **OP_DECRYPT** for decryption.

The *prot* variable is set to one of the following three values: **ICV_HASH**, **ICV_HASH_ENCRYPT**, or **ICV_ENCRYPT_HASH** which either produce a hash (or MAC tag) on the plaintext, on the plaintext and then encrypt the tag, or on the ciphertext respectively.

The *icvcmd* parameter must be set to one of: **IP_ICV_OFFSET**, **IP_ICV_APPEND**, or **IP_ICV_IGNORE**. These tell the engine to store the ICV (hash or MAC tag) at a fixed location, append to the ciphertext, or ignore the setting (for jobs that do not have a hash component).

The ICV offset is set by the *icvoff* parameter and only used in **IP_ICV_OFFSET** mode. The size of the ICV is passed as *icvsz* and can be used to tell the engine to truncate the (or expect a truncated) ICV.

The *sec_key* parameter is used to indicate whether secure key port is to be used.

2.5.5 spacc_write_rc4_context()

The RC4 context can be initialized by either loading a short 40- or 128-bit key in the key context page, or by loading a fully scheduled RC4 context (258 bytes long) into the RC4 context page. The fully scheduled key can be loaded with *spacc_write_context*() or with the following function directly. When calling *spacc_write_context*() they key is assumed to be 258 bytes long where the first two bytes are *i* and *j* and the remaining 256 bytes are *ctxdata*.

```
int spacc_write_rc4_context (spacc_device *spacc, int job_idx, unsigned char i, unsigned char j,
                             unsigned char *ctxdata);
```

This sets the RC4 context page with the 256 bytes of state from *ctxdata* (the permutation) and the i and j pointers. This should only be called after opening a job_idx with the **CRYPTO_MODE_RC4_KS** mode, otherwise the engine will be programmed to expand a key context page key into an RC4 context page.

The RC4 context can be retrieved with the following function:

```
int spacc_read_rc4_context (spacc_device *spacc, int job_idx, unsigned char *i, unsigned char *j,
                            unsigned char *ctxdata);
```

Which retrieves the RC4 context page and stores the 256-byte permutation in *ctxdata* and the pointers in *i* and *j*.

2.5.6 spacc_set_key_exp()

This function sets the **KEY_EXP** bit of the SPAcc CTRL register. It is used in the following circumstances

- Using AES in decrypt mode
- Using RC4 in 40 or 128-bit key mode (but not **KS** mode)

```
int spacc_set_key_exp(spacc_device *spacc, int job_idx);
```

The function modifies the internal CTRL register of the job associated with *job_idx* to have the bit. By default, new jobs do **not** have this bit set.

2.5.7 spacc_set_auxinfo()

The *spacc_set_auxinfo()* function is used to set the AUXINFO register of the SPAcc (used by 3GPP modes).

```
int spacc_set_auxinfo(spacc_device *spacc, int job_idx, uint32_t direction, uint32_t bitsize);
```

This sets the direction bit and bits used in the last byte in the AUXINFO register.

2.5.8 spacc_packet_enqueue_ddt()

The *spacc_packet_enqueue_ddt* function is the only function in the SDK that writes directly to SPAcc FIFO registers (via memory mapped IO). The caller must call the various support functions such as *spacc_open()*, *spacc_set_context()*, *spacc_set_operation()*, and *spacc_set_auxinfo()* before calling this function.

```
int spacc_packet_enqueue_ddt (spacc_device *spacc, int job_idx,
                              pdu_ddt *src_ddt, pdu_ddt *dst_ddt,
                              uint32_t proc_sz,
                              uint32_t aad_offset, uint32_t pre_aad_sz, uint32_t post_aad_sz,
                              uint32_t iv_offset, uint32_t prio);
```

This function requires previously initialized DDT tables (using PDU DDT functions provided by the platform device driver). Passed as *src_ddt* and *dst_ddt*. The only thing that this function does with them is read out the *len* and *phys* members to program SPAcc FIFO registers.

The *proc_sz* parameter indicates the desired value for the PROCLEN register. It can be set to the special value **SPACC_AUTO_SIZE** to allow the function to compute the register on the fly. Setting it to a fixed value is useful when processing partial blocks of a larger mapped job.

The *aad_offset* parameter indicates the offsets for the source and destination buffers to the core. Note: it must be pre-formatted as per the SPAcc User Guide. Usually, this value is left as zero.

The *pre_aad_sz* and *post_aad_sz* parameters indicate the size of the pre- and post-aad respectively. By default, the SDK does not copy AAD data to the destination buffer. This can be changed by or'ing the value **SPACC_AADCOPY_FLAG** into the *pre_aad_sz* parameter. Without the flag set the *pre_aad_sz* is most commonly used by hashing jobs by setting it to *proc_sz* such that only the hash output is emitted to the destination buffer.

The *iv_offset* parameter indicates the offset of the IV (if to be read from the source DDT mapped data) or zero if to be read from the context page. The MSB (0x80000000) must be set if IV import is to be used (allowing for an IV offset of zero).

The *prio* parameter indicates which priority queue to fire the job in (if supported).

Once a job has been pushed on the command FIFO it can either be polled to be dequeued or the user can wait for an interrupt (depending on the platform setup). The user may not alter the contents of their DDT structures until the job is complete.

The function will return **CRYPTO_FIFO_FULL** if the CMD FIFO is full.

The function will return **CRYPTO_CMD_FIFO_INACTIVE** if the priority specified refers to a CMD FIFO that is not present.

2.5.9 spacc_packet_dequeue()

The *spacc_packet_dequeue()* function polls the SPAcc to determine if the specified job is complete. In reality, the function sees if *any* job is ready to be popped off the status FIFO and it marks that respective job as complete, upon returning it indicates to the caller it the job they inquired about is complete.

```
int spacc_packet_dequeue (spacc_device *spacc, int job_idx);
```

This function returns **CRYPTO_OK** if the job specified by *job_idx* is complete or **CRYPTO_IN-PROGRESS** if it is still pending.

2.5.10 spacc_pop_packets()

This function checks the status FIFO and pops all available jobs off. It marks the job complete and returns the number of jobs popped. The address to and integer must be passed in. The return code is **CRYPTO_OK** or **CRYPTO_INPROGRESS**.

```
int spacc_pop_packets (spacc_device * spacc, int *num_popped);
```

2.5.11 spacc_close()

Once a job has completed and a context is no longer required it should be freed as soon as possible to return the context to the device for other jobs/users use.

```
int spacc_close (spacc_device *spacc, int job_idx);
```

This closes the context and returns it to the available pool. Note that this will not wait for the job to pop off the status FIFO.

2.6 IRQ Functions

The following functions abstract the details and management of the IRQ registers.

2.6.1 spacc_irq_XXX_enable/disable()

The following functions abstract the details of managing the IRQ registers. Check with the HW documentation to understand when the IRQ signals are triggered.

The *cmdx* parameter represents the CMD0, CMD1, or CMD2 register in the hardware. The number of CMD options depends on enabling QOS in hardware. The *cmdx_cnt* is a threshold level for when to trigger the interrupt.

```
void spacc_irq_cmdx_enable (spacc_device *spacc, int cmdx, int cmdx_cnt);
void spacc_irq_cmdx_disable (spacc_device *spacc, int cmdx);
```

These function sets the threshold for triggering the IRQ at the same time as enabling the interrupt.

```
void spacc_irq_stat_enable (spacc_device *spacc, int stat_cnt);
void spacc_irq_stat_disable (spacc_device *spacc);
```

These functions enable and disable the STAT_WD IRQ.

```
void spacc_irq_stat_wd_enable (spacc_device *spacc);
void spacc_irq_stat_wd_disable (spacc_device *spacc);
```

These functions enable and disable the RC4 DMA IRQ.

```
void spacc_irq_rc4_dma_enable (spacc_device *spacc);
void spacc_irq_rc4_dma_disable (spacc_device *spacc);
```

These functions enable and disable the Global setting.

```
void spacc_irq_glbl_enable (spacc_device *spacc);
void spacc_irq_glbl_disable (spacc_device *spacc);
```

2.6.2 spacc_process_irq()

This function determines which IRQ was signaled and calls the registered callback function. The callback functions must be initialized in the spacc_device structure that is passed to the function. These parameters are:

```
        void (*irq_cb_cmdx)(struct _spacc_device *spacc, int x);
        void (*irq_cb_stat)(struct _spacc_device *spacc);
        void (*irq_cb_stat_wd)(struct _spacc_device *spacc);
        void (*irq_cb_rc4_dma)(struct _spacc_device *spacc);
```

and the function definition is:

```
uint32_t spacc_process_irq(spacc_device *spacc);
```

2.7 Other Functions

2.7.1 spacc_error_msg()

The SPAcc SDK returns CRYPTO_OK from most functions if successful, or a negative number to indicate failure. The error codes can be mapped to printable strings with the following function:

```
unsigned char *spacc_error_msg (int err);
```

2.7.2 spacc_set_secure_mode()

The *spacc_set_secure_mode()* function sets the secure mode register of the SPAcc which allows for restrictions on how the bus mastering and slave register map access behave.

```
void spacc_set_secure_mode (spacc_device *spacc, int src, int dst, int ddt, int global_lock);
```

The parameters all behave as booleans where *src* controls whether source data mastering asserts secure mode, *dst* for destination data, *ddt* for DDT structure access, and *global_lock* for the slave port registers.

2.7.3 spacc_dump_ctx()

The spacc_dump_ctx() function is used to get debugging information about a given SPAcc context.

```
void spacc_dump_ctx(spacc_device *spacc, int ctx);
```

This will print via ELPHW_PRINT the contents of the hash, cipher, and if appropriate the RC4 context for a given context specified. If the context is not accessible (secured context from normal port) it will print out all zeroes.

2.7.4 spacc_isenabled()

This function is provided to determine if a given cipher or hash mode is enabled in the core.

```
int spacc_isenabled(spacc_device *spacc, int mode, int keysize);
```

The user passes the CRYPTO_MODE_* mode they wish to test along with the keysize in bytes passed as *keysize*. It will return 0 if the mode is not enabled or non-zero if it is.

2.7.5 spacc_compute_xcbc_key()

This function is provided to initialize a 3-key XCBC MAC key from a single 16 byte key. This function is typically used in protocols such as IPsec where the SA supplied key is short. This function requires that either AES-128-ECB or AES-128-CBC be enabled to correctly compute the 48 byte key.

```
int spacc_compute_xcbc_key(
           spacc_device *spacc,
    const unsigned char *key, int keylen,
          unsigned char *xcbc_out);
```

This will use the n-byte input key pointed to by *key* of length *keylen* and process it into the 48-byte scheduled key *xcbc_out*.

2.8 SPAcc Userspace Interface

On top of the *src/core/kernel/example.c* example routines is a userspace interface that allows cipher, hash, and combined cipher/hash jobs to be performed from a userspace task. The module source code is *src/core/kernel/spacc_dev.c* and builds as *elpspaccusr.ko*. When loaded the driver provides the entry */dev/spaccusr* which the user space API can then use to perform SPAcc jobs.

The userspace side of the API resides in *src/core/user/spaccdev.c* and can be linked into any application that wishes to make use of the SPAcc. This source file depends on two additional files found in *src/core/include*: *elpspaccusr.h* and *elpspaccmodes.h* which can be copied with *spaccdev.c* into a user project.

Packets processed by the driver occur in a blocked mode in which the function will only return when the job is complete (or an error detected). It will re-try jobs internally if the command FIFO is full allowing more jobs to be on the fly than FIFO entries.

Note: Per default the userspace driver allows up to 32 segments per source and destination buffer (including user side scatter gather). This typically more than enough to handle up to 64Kbyte jobs using 4K pages. The user can change this by defining the variable **SPACC_DEFAULT_DDT_LENGTH** in their environment or editing the definition at the top of *elliptic/driver/src/core/kernel/spacc_dev.c*.

2.8.1 Opening a Userspace Handle

```
int spacc_dev_open(
   struct elp_spacc_usr *io,
   int cipher_mode, int hash_mode,    // cipher and hash (see elpspaccmodes.h)
   int encrypt,                       // non-zero for encrypt (sign) modes
   int icvmode,                       // ICV mode (see elpspaccmodes.h)
   int icvlen,                        // length of ICV, 0 for default length (algorithm dependent)
   int aad_copy,                      // non-zero to copy AAD to dst buffer
   unsigned char *ckey, int ckeylen, unsigned char *civ, int civlen,  // cipher key and IVs
   unsigned char *hkey, int hkeylen, unsigned char *hiv, int hivlen); // hash key and IVs (if any)
```

This function opens a SPAcc device handle. The cipher/hash modes are specified the same as if calling *spacc_open()* in the kernel space. The *encrypt* flag is set to non-zero when encrypting (generating a hash/hmac output) and zero when decrypting (verifying a hash/hmac input). The icvmode is set to one of the **ICV_** flags defined in *elpspaccmodes.h*. The desired length of the ICV may be specified as *icvlen* (bytes), where 0 indicates to use the default (maximum) length. The user may wish to indicate that AAD is to be copied via the *aad_copy* flag.

The ciphering key and IV are passed as *ckey* and *civ* with their respective lengths in bytes *ckeylen* and *civlen*.

The hashing key and IV are passed as *hkey* and *hiv* respectively (same with lengths).

When certain AES modes are used in decrypt mode the open call will cause a short SPAcc job to occur to ensure the cipher key has been properly initialized. This is done to allow multiple handles to be bound to this handle correctly. This should be transparent for the user but they will notice an extra IRQ during the open call.

The function returns 0 upon success and a negative number upon error.

2.8.2 Performing a SPAcc Job

A cipher, hash, or cipher/hash job may be performed with this API with the following function:

```
int spacc_dev_process(
   struct elp_spacc_usr *io,
   unsigned char *new_iv,
   int            iv_offset,
   int            pre_aad,
   int            post_aad,
   int            src_offset,
   int            dst_offset,
   unsigned char *src, unsigned long src_len,
   unsigned char *dst, unsigned long dst_len);
```

This function instructs the device driver to process a SPAcc job and does not return (it is a blocking call) until the SPAcc completes the job or there is an error.

The user may use a fresh cipher IV with the job by passing a pointer to the new IV in *new_iv*. If they wish to use IV import or the existing IV in the cipher key context they may pass it as **NULL**.

The user may instruct the device to use the IV import functionality to read an IV from the source packet by setting the *iv_offset* parameter. If not they may set it to the default of *-1* which means to ignore the IV import functionality.

Note: It is more efficient to use the IV import functionality over passing an IV pointer since it invokes fewer slave port accesses to the SPAcc core. If at all possible it is better to store the IV at the beginning or end of the source buffer (if there is room) and import it from there.

The length of the pre and post AAD data is specified in the *pre_aad* and *post_aad* parameters.

The offset from the start of the source and destination buffers to the first byte of *pre_aad* (or plaintext/ciphertext if *pre_aad* is zero) is specified in the *src_offset* and *dst_offset* parameters.

The length of the input packet is specified in *src_len*. It is much like the **PROC_LEN** register of the SPAcc in that it includes pre-AAD but not post-AAD. Unlike the **PROC_LEN** register it never includes the ICV data (for example when *icv_mode* is **ICV_HASH_ENCRYPT**). The *src_len* does not have to contain the IV if IV import is being used. For instance, if the IV is stored passed the end of the packet data the driver will know to map ***more*** than the specified bytes.

The *dst_len* parameter is how many bytes starting from *dst* to map into the kernel.

Note: It is more efficient to perform SPAcc jobs in-place where *src* == *dst*. In this mode the Kernel will only map the larger of the input/output buffers once from userspace into the kernel. Users can still use source/destination offsets but the destination offset cannot overlap with the source buffer (unless the offsets are equal).

If the function is successful it will return the number of bytes emitted by the SPAcc (which may be zero if there are no plaintext bytes and *aad_copy* is set to zero). It will return -1 upon error. The user may check the variable *io->io.err* for the SPAcc related error.

Note: The kernel side of this function will retry up to 100 times with a 10 millisecond delay[1] to program the job if the CMD FIFO is full. After the 100th retry it will return an error with *io->io.err* set to **CRYPTO_FIFO_FULL**.

See *src/core/user/spaccdevtest.c* for an example of calling this API in one of the three modes.

2.8.3 3GPP Ciphersuite Modes

With the 3G cipher suites (SNOW3G, ZUC, KASUMI F9) users will need to call the following macro to set the AUXINFO register.

```
spacc_dev_setaux(fd, align, dir)
```

Which assigns the alignment and direction flags to the state pointed to by *fd*.

2.8.4 Performing Jobs with Userspace Scattergather

The function *spacc_dev_process()* only allows the user to pass a single buffer for source and/or destination to the kernel side of the task. This is sufficient for many APIs but prevents users from gathering jobs from distinct sources. To support scattergather users from the userspace there is access to create DDT entries and pass them on to the kernel.

The following two macros are used to set DDT entries:

```
ELP_SPACC_SRC_DDT(fd, x, addr, len)
ELP_SPACC_DST_DDT(fd, x, addr, len)
```

Where *fd* is a *struct elp_spacc_ioctl* type, *x* is which entry to set and *addr*/*len* are the address/length of this segment. DDT lists must be terminated with a NULL pointer which can be accomplished with:

```
ELP_SPACC_SRC_TERM(fd, x)
ELP_SPACC_DST_TERM(fd, x)
```

Which sets the *x*'th entry to the terminator. The macro **ELP_SPACC_USR_MAX_DDT** indicates the maximum number of DDT entries and can be changed (provided the kernel module is recompiled as well). By default, it is set to 8. **Note**: that room is provided for the required terminator, so by default 9 DDT entries are allocated.

Once the DDTs are ready the source and destination lengths must be computed. These are the original *src_len*/*dst_len* passed to *spacc_dev_process()*. They can be set manually with:

```
ELP_SPACC_SRCLEN_SET(fd, x)
ELP_SPACC_DSTLEN_SET(fd, x)
```

1 It is ideal to have the clock tick set to at least 100Hz for this function to have a ceiling of 1sec timeout.

or set to be computed on the fly with:

```
ELP_SPACC_SRCLEN_RESET(fd)

ELP_SPACC_DSTLEN_RESET(fd)
```

Note: that if they are computed on the fly the caller must reset them every time they issue a job.

Unlike with *spacc_dev_process()*, the caller is required to include the IV (if using *iv_offset*) in one of the DDT segments passed. Users can still perform jobs in place (for higher performance) provided both the destination and source DDTs are equal (in pointers and in lengths).

Once the DDTs and source and destination lengths are ready the user can call the following function to issue the job:

```
int spacc_dev_process_multi(

   struct elp_spacc_usr *io,

   unsigned char *new_iv,

   int           iv_offset,

   int           pre_aad,

   int           post_aad,

   int           src_offset,

   int           dst_offset

   int           map_hint);
```

This will issue the job with the same semantics as *spacc_dev_process()*. The new parameter *map_hint* is used to help the kernel module determine if the mappings overlap. It may be one of the following values: **SPACC_MAP_HINT_TEST, SPACC_MAP_HINT_USESRC, SPACC_MAP_HINT_USEDST, SPACC_MAP_HINT_NOLAP**.

The **SPACC_MAP_HINT_TEST** hint instructs the module to test whether the mappings are compatible with a single mapping. The **SPACC_MAP_HINT_USESRC** and **SPACC_MAP_HINT_USEDST** tell the module to use the source or destination mappings explicitly. The caller must use the larger of the two mappings, otherwise the job could fail (and the application could segfault). The **SPACC_MAP_HINT_NOLAP** hint instructs the module that there is no overlap and it must map them separately.

For the general case use **SPACC_MAP_HINT_TEST** as that will safely test internally if the mapping is compatible (though it will be slower than an explicit hint).

Note: Calls to *spacc_dev_process()* and *spacc_dev_process_multi()* may be mixed on the same handle provided the DDT entries are initialized prior to each call to *spacc_dev_process_multi()*. Calling *spacc_dev_process()* **will overwrite** the DDT entries associated with the handle.

An example of this API can be found in the function *speed()* of the file *elliptic/driver/src/core/user/spaccdevtest.c*.

2.8.5 Context Sharing

It is possible to share SPAcc key context pages between jobs to allow parallel use of the SPAcc (via threading on the user side) via the register and bind API. First, the caller must register their state for sharing by calling:

```
int spacc_dev_register(struct elp_spacc_usr *io);
```

This assigns the state a random 128-bit key[2] that is later used to pair open handles. This function returns 0 upon success. Each time it is called a new 128-bit key is used for sharing. **Note:** that this key has nothing to do with whatever ciphering key is being used, it is only used to pair handles. Essentially, this key prevents other threads/processes from attempting to bind to an open handle and then using the key context without permission.

Next, a new handle can be opened and paired (bound) to a master by calling:

```
int spacc_dev_open_bind(struct elp_spacc_usr *master, struct elp_spacc_usr *new);
```

This will clone *master* and initialize *new*. When done they will both refer to the same SPAcc key context page despite using different SPAcc handles. Now both *master* and *new* can be used in parallel (in different threads) to allow filling the CMD FIFO of the SPAcc.

The SPAcc key context page is not freed until all references are released so the *master* handle can be closed while the *new* one is still operational.

NOTE: With shared contexts the IV import functionality must be used with every packet since the key context page cannot be changed. This is accomplished by passing *iv_offset* as a value above -1 to the function *spacc_dev_process()*. The use of the *new_iv* parameter is **not** allowed in this mode.

The program in *elliptic/driver/src/core/user/spaccdevtest.c* makes use of this functionality towards the end of the file in the function *speed()*. Another way to fit this into APIs is to use an offset where the application expects to see application data, for instance:

```
struct data_pointer {
   unsigned char *mem, // base where we would put the IV
                 *src; // where application data goes
};
```

Which can then be initialized as follows:

```
int init_dp(struct data_pointer *dp, int len)
{
   dp->mem = calloc(1, len + 16);
   dp->src = dp->mem + 16;
   return dp->mem == NULL;
}
```

Now the application can read/write data from *dp->src* and the provider can pass *dp->mem* to the SPAcc userspace API. Here we leave 16 bytes so we can store an IV and use the IV import functionality easily.

2.8.6 Partial Message Processing

For SPAcc cores that support partial message processing users may program jobs through the userspace API in partial steps. This allows large jobs to be processed through the SPAcc in steps that are more manageable. Jobs are programmed as normal with the exception of message flags that must be used.

```
#define ELP_SPACC_USR_MSG_DEFAULT(fd)
#define ELP_SPACC_USR_MSG_FIRST(fd)
#define ELP_SPACC_USR_MSG_MIDDLE(fd)
#define ELP_SPACC_USR_MSG_LAST(fd)
```

2 By reading /dev/urandom for 16 bytes.

The first block of data processed must be processed by calling **ELP_SPACC_USR_MSG_FIRST()** on the *elp_spacc_usr* handle that is open. If there are more than one block remaining **ELP_SPACC_USR_MSG_MIDDLE()** must be called (only once), and for the final block **ELP_SPACC_USR_MSG_END()**. For the rules of how to split up jobs for partial processing refer to the **EDN-0228 SPAcc user guide**.

To return an open handle to the default non-partial processing mode use **ELP_SPACC_USR_MSG_DEFAULT()**.

For a simple example of partial processing refer to *elliptic/driver/src/core/user/spaccdevtest.c*.

2.8.7 Feature Testing

The device driver may be queried to determine which cipher and hash modes are available to the user.

```
int spacc_dev_features(struct elp_spacc_features *features);
```

This will populate the structure *elp_spacc_features* with values from the kernel device driver. It will return non-zero upon error.

The structure has the following definition:

```
struct elp_spacc_features {
   unsigned
      project,
      partial,
      qos,
      max_msg_size;
   unsigned char
      modes[CRYPTO_MODE_LAST];
};
```

The *project* value denotes the SPAcc project number used to identify the configuration. The *partial* value denotes whether partial processing is supported or not. The *qos* flag indicates whether IV import functionality is available. The *max_msg_size* value denotes the maximum message size that is allowed by the core.

To determine if certain cipher or hash modes are available the following function may be used:

```
int spacc_dev_isenabled(struct elp_spacc_features *features, int mode, int keysize);
```

This function returns 1 if the mode is available. The mode is specified as one of the **CRYPTO_MODE_*** values define in *elliptic/driver/src/core/include/elpspaccmodes.h* and the key size in bytes.

For example, to determine if AES-128-CBC is available:

```
if (spacc_dec_isenabled(&features, CRYPTO_MODE_AES_CBC, 16)) {
   printf(“AES-128-CBC is available\n”);
}
```

When testing the presence of unkeyed HASH functions use the key size 0. For example,

```
if (spacc_dec_isenabled(&features, CRYPTO_MODE_HASH_MD5, 0)) {
   printf(“MD5 hash is available\n”);
}
```

would test the presence of the MD5 hash function.

2.8.8 Closing Handles

```
int spacc_dev_close(struct elp_spacc_usr *io);
```

This function closes the device handle.

3 RE

The SPAcc-PDU Record Engine (RE) is used to perform SSL 3.0 and TLS/DTLS record transforms. The engine is always attached to the first SPAcc interface of a device (if virtual SPaccs are present) and uses contexts out of that device.

3.1 Initialization

The initialization of the RE engine must occur *after* initializing the SPAcc SDK for the corresponding device. The RE makes calls to the SPAcc SDK to grab a device context and allocate handles. Once the SPAcc has been initialized the following function can be used to initialize an RE device:

```
int re_init(void *baseaddr, spacc_device *spacc, re_device *re);
```

Like the SPAcc initialization it requires a base address which represents the virtual address of the memory mapped I/O from the RE device. It allocates structures and data inside the *re* parameter which must be freed with *re_fini()* when finished.

After calling this function the parameter *re->sa_pool_size* is set to the size in bytes of memory that the caller must allocate (and map) setting the *re->sa_pool_virt* and *re->sa_pool_phys* pointers to the virtual and physical pointers directly. For example, from the RE kernel module:

```
   err = re_init(baseaddr, spacc, &re);
   if (err != CRYPTO_OK) {
      iounmap(baseaddr);
      return -1;
   }

   // after we call re_init we need to allocate a pool
   re.sa_pool_virt = pdu_dma_alloc(re.sa_pool_size, &re.sa_pool_phys);
   if (!re.sa_pool_virt) {
      printk("Cannot allocate SA pool\n");
      iounmap(baseaddr);
      return -1;
   }
```

The *pdu_dma_alloc* function allocates a consistent block of memory which retrieves both the virtual and physical address. The RE SDK needs both since it does not perform mappings (a platform specific operation) itself.

3.2 IRQ Support

An IRQ handler for the SPAcc-RE should simply call *re_packet_dequeue()* with a handle of -1. This will pop a job off the FIFO, mark it as done, and optionally call a callback (to wake a user thread for instance).

3.3 Termination

When the RE device is no longer needed it can be freed with the following function:

```
void re_fini(re_device *re);
```

This will free any internally allocated memory in the RE device structure. Note: The caller must free the SA pool allocated just after calling *re_init()* themselves. For instance:

```
// fini the SDK then free the SA pool
re_fini(&re);
dma_free_coherent(NULL, re.sa_pool_size, re.sa_pool_virt, (dma_addr_t)re.sa_pool_phys);
```

3.4 Job Programming

The usual method of programming a record processing job is as follows

1. Open an RE context with *re_get_context()*
    1. (Optionally), the caller can register a callback that will be called when the job terminates
2. Call to *re_reset_sa()* to clear the current SA
3. Write the read and write SA state with *re_set_next_read()* and *re_set_next_write()* programming the keys, IVs, RE parameters (cipher/hash/proto), and sequence numbers.
4. Initialize source and destination DDT structures with PDU DDT functions.
5. Call *re_start_operation()* to enqueue the job in the RE command FIFO
6. Wait for an interrupt or poll *re_finish_operation()* to pop the job off the status FIFO

Between successive records steps 1-2 can be omitted.

3.4.1 re_get_context()

The RE engine uses contexts in much the same way as the SPAcc engine does. Each context allocated with this function maps directly to a SPAcc context (on the first virtual spacc if enabled).

```
int re_get_context(re_device *re, re_callback cb, void *cbdata);
```

This returns the context handle, or negative upon error. The callback *cb* is of the form:

```
void re_callback(void *re_dev, void *data);
```

and is called when the job is popped off the FIFO (ideally by an IRQ handler calling *re_packet_dequeue()* with a handle of -1). The *re_dev* value is a pointer to the *re_device* structure being used in case the callback needs to access the device. The *cbdata* pointer is passed as *data* to the callback. This can be used as a pointer to a semaphore to unblock a user thread.

Note: The callback is called potentially from an interrupt context. It must be IRQ safe.

3.4.2 re_get_context_ex()

To allocate an RE context with a specific SPAcc context the following function can be used:

```
int re_get_context_ex (re_device * re, int ctxid, re_callback cb, void *cbdata);
```

This function allows independent callbacks to be registered with different RE contexts while they re-use the same SA and SPAcc key context page. In theory, a multiple threaded application could program jobs for the same SSL/TLS session in parallel (to keep the RE CMD FIFO full).

3.4.3 re_get_spacc_context()

To determine which SPAcc context has been allocated to a given RE context the following function may be used:

```
int re_get_spacc_context(re_device *re, int re_ctx);
```

This returns the SPAcc context allocated which can be passed to *re_get_context_ex()* to allocate another RE context with the same SPAcc context.

3.4.4 re_reset_sa()

The SA context page is where the RE engine stores and retrieves information such as cipher keys, IVs, and sequence numbers. It must be initialized to the all zero state with the following function.

```
int re_reset_sa (re_device *re, int handle, uint32_t version);
```

3.4.5 re_set_next_read() and re_set_next_write()

These functions write the the SA structures for the read and write channels. They rely on the SA having been reset previously (or at least initially). The function reads the SA_FLAGS field to determine which read/write channel (of the two) are to be written. See section 7.1 of EDN-0278 for the structure of the flags and the rest of the SA.

```
int re_set_next_read(re_device *re,
                     int handle,
                     unsigned char *iv,              uint32_t ivlen,
                     unsigned char *key,             uint32_t keylen,
                     unsigned char *mackey,          uint32_t mackeylen,
                     unsigned char *params,          uint32_t paramlength,
                     unsigned char *sequence_number, uint32_t seqlength);


int re_set_next_write(re_device *re,
                      int handle,
                      unsigned char *iv,              uint32_t ivlen,
                      unsigned char *key,             uint32_t keylen,
                      unsigned char *mackey,          uint32_t mackeylen,
                      unsigned char *params,          uint32_t paramlength,
                      unsigned char *sequence_number, uint32_t seqlength);
```

The caller must set both directions at least once to perform directional communication with this RE handle.

3.4.6 re_start_operation()

Once a context has been allocated and the SA programmed with the keys and IVs a record can now be processed in either encrypt or decrypt mode.

```
int re_start_operation(re_device *re, int handle, pdu_ddt *src_ddt, pdu_ddt *dst_ddt,
                       uint32_t type);
```

This will push a command on the FIFO getting the RE working on a record. The caller must initialize their source and destination DDT structures with functions from the PDU DDT SDK prior to calling this function. The *type* parameter reflects the **CMD** field of the **RE_CTRL** register (section 7.2 of EDN-0278). The defines in *elprehw.h* provided for the **CMD** field are: **RE_SND_CCS**, **RE_SND_ALERT**, **RE_SND_HANDSHAKE**, **RE_SND_DATA**, **RE_RCV_RECORD**.

Once a job has been successfully pushed on the command FIFO the caller may not modify the SA or DDT structures passed in.

3.4.7 re_finish_operation()

Once a job has been pushed onto the command FIFO the caller can poll the status FIFO with the following function.

```
int re_finish_operation(re_device *re, int handle, uint32_t * length, int * id);
```

This will return CRYPTO_OK if the specified handle is done, otherwise it returns CRYPTO_IN-PROGRESS. It is upto the IRQ handler to call *re_packet_dequeue()* to move jobs off the RE FIFO.

3.4.8 re_release_context()

When a caller is finished with a context they can call *re_release_context()* to release the SPAcc and RE context back to the pool.

```
int re_release_context(re_device *re, int handle);
```

Once this call is complete the associate SA state can no longer be assumed. The caller should call *re_retrieve_sa()* if they intend to continue where the context left off.

3.5 Other Functions

3.5.1 re_retrieve_sa() and re_write_sa()

When contexts are in short supply the caller may save their current SA state to an external buffer. This allows the caller to free their handle, and then when needed next restore the SA and resume working on the record stream. The *re_start_operation()* function sets the LOAD/STORE SA bit of the RE CTRL register instructing the engine to update the SA before and after RE commands are processed.

```
int re_retrieve_sa(re_device *re, int handle, unsigned char * sabuff, uint32_t bufflength);
```

This will retrieve the SA out of the context memory and store it in *sabuff*. You must pass it the length of the buffer (which must be at least 1024 bytes long, the size of an SA) in *bufflength*. After the caller has retrieved the SA they can call *re_release_context()* on the handle to free it back to the general pool of available contexts.

To restore the SA to an allocated handle the *re_write_sa()* function is available.

```
int re_write_sa(re_device *re, int handle, unsigned char * sa, uint32_t bufflength);
```

This will write the SA pointed to by *sa* to the context specified by the handle. The length of the source must be passed as *bufflength* and must be 1024 bytes long.

3.6 Userspace Library

The userspace library contained within *src/re/user/reuser.c* implements an interface to the */dev/spaccreusr* device driver provided by the *elpreusr.ko* kernel module. Like the SPAcc userspace driver this driver provides for zero-copy, scattergather, and multi-threading support to RE contexts.

Example usage of the driver can be found in *elliptic/driver/src/re/user/reuserdiag.c* and *reuserspd.c*.

Note: Per default the userspace driver allows up to 16 segments per source and destination buffer (including user side scatter gather). This typically more than enough to handle up to 16Kbyte jobs using 4K pages. The user can change this by defining the variable **RE_DEFAULT_DDT_LENGTH** in their environment or editing the definition at the top of *elliptic/driver/src/re/kernel/re_dev.c*.

3.6.1 Opening an RE userspace handle

```
int re_dev_open(struct elp_spacc_re_usr *io, int version);
```

This function will open a handle to the device requesting a specific version of SSL/TLS/DTLS. This call will allocate a SPAcc key context page handle. Returns zero or above upon success.

The version should be one of:

```
enum {
   RE_VERSION_SSL3_0,
   RE_VERSION_TLS1_0,
   RE_VERSION_TLS1_1,
   RE_VERSION_TLS1_2,
   RE_VERSION_DTLS
};
```

3.6.2 Sharing a SPAcc Context Page

As with the SPAcc userspace driver it is possible to share a SPAcc context page between multiple open RE handles. This allows multi-threaded applications to enqueue multiple jobs into the RE using the same session keys.

After a single RE context has been allocated with *re_dev_open()* it can be registered for sharing with:

```
int re_dev_register(struct elp_spacc_re_usr *io);
```

This reads a random 128-bit key (from /dev/urandom) to associate with the context which is used to prevent other processes from associating with it. Next, the handle can be cloned with:

```
int re_dev_open_bind(struct elp_spacc_re_usr *master, struct elp_spacc_re_usr *new);
```

After this call the handle *new* refers to its own RE context but shares the SPAcc context page with *master*. Now both *master* and *new* can be used in parallel.

3.6.3 Setting Read/Write Contexts

```
int re_dev_set_write_context(struct elp_spacc_re_usr * fd,
                   unsigned char *iv,             uint32_t ivlen,
                   unsigned char *key,            uint32_t keylen,
                   unsigned char *mackey,         uint32_t mackeylen,
                   unsigned char *params,          uint32_t paramlength,
                   unsigned char *sequence_number, uint32_t seqlength);
int re_dev_set_read_context(struct elp_spacc_re_usr * fd,
                   unsigned char *iv,              uint32_t ivlen,
                   unsigned char *key,             uint32_t keylen,
                   unsigned char *mackey,          uint32_t mackeylen,
                   unsigned char *params,          uint32_t paramlength,
                   unsigned char *sequence_number, uint32_t seqlength);
```

These functions set the write and read contexts of a given handle respectively. The iv/key/mackey parameters are the data values for the symmetric algorithms and their lengths depend on the cipher-suite being used.

The *params* field is a 2 byte field that the SPAcc-RE uses to describe the ciphersuite. The *params* field is left generic so as to make the API more flexible going forward. It can be set for the SPAcc-RE with the **ELP_RE_PARAMS** macro (see *src/re/include/elpreuser.h* for information).

After setting the context a Change CipherSuite (CCS) message must be passed to the SPAcc-RE.

3.6.4 Simple Sending/Receiving Records

```
int re_dev_do_record(struct elp_spacc_re_usr *io,
                     const unsigned char *in,        uint32_t  inlen,
                           unsigned char *out,       uint32_t  outlen,
                                    int src_offset,      int  dst_offset,
                                    int cmd);
```

This function is responsible for sending records through the device. The caller must set *outlen* to the max size of the output buffer. The *type* of the command must be one of:

```
enum {
  RE_DEV_CCS,        // command will set the cipher suite
  RE_DEV_HANDSHAKE,  // command will send a handshake
  RE_DEV_ALERT,      // command will send an alert
  RE_DEV_WRITE_MODE, // command will write plaintext and will return ciphertext
  RE_DEV_READ_MODE,  // command will write ciphertext and will return plaintext
                     // read mode is used to receive any record including CCS, HANDSHAKE, and ALERT
};
```

In order to perform a CCS command there must be at least 1 byte of data (typically just the 0x01 byte). The function blocks while the device is busy (see the registering/binding contexts). It returns a zero or positive number indicating the length of the output upon success. It returns a negative number upon error.

Note: The kernel side of this function will retry up to 100 times with a 10 millisecond delay[3] to program the job if the CMD FIFO is full. After the 100th retry it will return an error with a record engine error code set to **CRYPTO_FIFO_FULL**.

The record engine error code can be read from the **RE_USR_ERR** macro where caller passes the *elp_spacc_re_usr* structure as a pointer.

Note: The ideal usage of this function is to have *in* and *out* overlap. For sending records the

3 It is ideal to have the clock tick set to at least 100Hz for this function to have a ceiling of 1sec timeout.

src_offset should be set to the size of the SSL/TLS record header and *dst_offset* set to zero. For receiving records the *dst_offset* and *src_offset* should be equal (ideally set to zero).

3.6.5 Sending/Receiving Scattergather Records

The interface also supports the user using scattered source and destination buffers when calling the interface. By default upto **SPACC_RE_USR_MAX_DDT** segments are supported per direction. This can be changed in *elliptic/driver/src/re/include/elpreuser.h*. **Note:** that changing this value will require recompiling the kernel driver.

The following two macros are used to set DDT entries:

```
ELP_SPACC_RE_SRC_DDT(fd, x, addr, len)
ELP_SPACC_RE_DST_DDT(fd, x, addr, len)
```

Where *fd* is a *struct elp_spacc_re_ioctl* type, *x* is which entry to set and *addr*/*len* are the address/length of this segment. DDT lists must be terminated with a NULL pointer which can be accomplished with:

```
ELP_SPACC_RE_SRC_TERM(fd, x)
ELP_SPACC_RE_DST_TERM(fd, x)
```

Which sets the *x*'th entry to the terminator. Once the DDTs are ready the source and destination lengths must be computed.

They can be set manually with:

```
ELP_SPACC_RE_SRCLEN_SET(fd, x)
ELP_SPACC_RE_DSTLEN_SET(fd, x)
```

Where *x* is the length of the aggregate buffer. They can also be set to be computed on the fly with:

```
ELP_SPACC_RE_SRCLEN_RESET(fd)
ELP_SPACC_RE_DSTLEN_RESET(fd)
```

Note: that if they are computed on the fly the caller must reset them every time they issue a job.

To program these jobs the following function can be used:

```
int re_dev_do_record_multi(struct elp_spacc_re_usr *io,
                           int src_offset, int dst_offset, int hint, int cmd);
```

This will program the job to run the given *cmd* (see *re_dev_do_record()*) and return the final output size. The *hint* parameter has the same functionality as that from *spacc_dev_process_multi()*. It is used to indicate to the kernel how to map the user memory. It may be one of the following values: **RE_MAP_HINT_TEST, RE_MAP_HINT_USESRC, RE_MAP_HINT_USEDST, RE_MAP_HINT_NOLAP**.

The **RE_MAP_HINT_TEST** hint instructs the module to test whether the mappings are compatible with a single mapping. The **RE_MAP_HINT_USESRC** and **RE_MAP_HINT_USEDST** tell the module to use the source or destination mappings explicitly. The caller must use the larger of the two mappings, otherwise the job will fail (and potentially it could segfault the calling application). The **RE_MAP_HINT_NOLAP** hint instructs the module that there is no overlap and it must map them separately. If you are unsure use **RE_MAP_HINT_TEST** as that will test internally if the mapping is compatible (though it will be slower than an explicit hint).

For the general case use **RE_MAP_HINT_TEST** as that will safely test internally if the mapping is compatible (though it will be slower than an explicit hint).

Note: The optimal usage for sending records is to have the buffers overlap and set the *src_offset* to just past the SSL/TLS header size and the *dst_offset* to zero. At this point both the source and destination mappings can be the same and the user can pass the **RE_MAP_HINT_USEDST** hint. This will instruct the kernel to map the destination buffer and use that both for input and output (saving one costly memor mapping). For the receive path *src_offset* and *dst_offset* must be equal (ideally zero) and the **RE_MAP_HINT_USESRC** hint should be used.

3.6.6 Closing a handle

```
int re_dev_close(struct elp_spacc_re_usr *io);
```

Closes a handle on the RE device. Note: This call will not free the associated SPAcc handle until all references to the handle are closed.

4 KEP

The SPAcc-PDU KEP engine is used to perform key exchange and message signing symmetric operations in the SSL and TLS protocols. It attaches to the first SPAcc core (in the case of virtual SPAcc devices) and like the RE shares contexts with that device.

4.1 Initialization

The KEP SDK is initialized with a call to *kep_init()* which allocates memory as required internally.

```
int kep_init(void *baseaddr, kep_device *kep);
```

The caller must pass the virtual address of the memory mapped I/O for the KEP device as *baseaddr*. The routine then initializes the contents of the *kep* structure. The caller must initialize the SPAcc SDK first for the device attached to the first core.

4.2 IRQ Support

An IRQ handler for the SPAcc-KEP should simply call *kep_done()* with a handle of -1. This will pop a job off the FIFO, mark it as done, and optionally call a callback (to wake a user thread for instance).

4.3 Termination

When the KEP SDK is no longer needed the resources allocated inside the *kep_device* structure can be freed with this function.

```
void kep_fini(kep_device *kep);
```

4.4 Job Programming

The usual way of programming a KEP job is as follows:

1. Allocate a handle with *kep_open()* specifying which operation and relevant options needed.
    1. (optionally) Register a callback to allow waking user threads.
2. (optionally) load hash keys into the context(s) with *kep_load_keys()*
3. Send the job into the command FIFO with *kep_go()*
4. Wait for an interrupt or poll with the function *kep_done()*

Typically, KEP jobs are not repeated but if they must be the caller can omit steps 1 and 2.

4.4.1 kep_open()

KEP jobs require a handle that associate with contexts on the first SPAcc core (in a virtual SPAcc environment) and are shared between RE and normal SPAcc jobs.

```
int kep_open(kep_device *kep, int op, int option, kep_callback cb, void *cbdata);
```

This opens handle for a given operation *op* with options *option*. The callback *cb* is an optional pointer to a callback function (may be **NULL** to disable) which is called when a job is completed. It is of the form:

```
void kep_callback(void *kep_dev, void *data);
```

It is passed the *kep_device* and *cbdata* pointers passed when the handle was opened. **NOTE:** The callback may be called from an interrupt context and must be IRQ safe.

The operation must be one of the following enumerated values:

KEP Operation	**Description**
KEP_SSL3_KEYGEN	SSL3.0 Key Generation (as described in section 8.1 of the SSL 3.0 spec)
KEP_SSL3_SIGN	SSL3.0 Signing Hash (as described in section 7.6.8 of the SSL 3.0 spec)
KEP_TLS_PRF	TLS1.0/1.1 PRF (described in section 5 of the TLS 1.0 spec)
KEP_TLS_SIGN	TLS1.0/1.1 Signing Hash (described in section 7.4.9 of the TLS 1.0 spec)
KEP_TLS2_PRF	TLS1.2 PRF (described in section 5 of the TLS 1.2 spec)
KEP_TLS2_SIGN	TLS1.2 Signing Hash (described in section 7.4.9 of the TLS 1.2 spec)

Table 6: KEP Operation Modes

The option parameter has different meanings depending on which operation is being performed. The following table summarizes the valid options:

KEP Operation	**KEP Option**	**Usage**	**Notes**
KEP_SSL3_KEYGEN	N/A	No Options	
KEP_SSL3_SIGN	**KEP_OPT_SSL3_SIGN_MD5** **KEP_OPT_SSL3_SIGN_SHA1**	One of	
KEP_TLS_PRF	N/A	No Options	
KEP_TLS_SIGN	**KEP_OPT_TLS_SIGN_CLIENT** **KEP_OPT_TLS_SIGN_SERVER**	One of	
KEP_TLS2_PRF	**KEP_OPT_TLS2_PRF_MD5** **KEP_OPT_TLS2_PRF_SHA1** **KEP_OPT_TLS2_PRF_SHA224** **KEP_OPT_TLS2_PRF_SHA256** **KEP_OPT_TLS2_PRF_SHA384** **KEP_OPT_TLS2_PRF_SHA512**	One of	SHA-2 hashes may not be enabled in the SPAcc engine, consult your ICD for details.
KEP_TLS2_SIGN	**KEP_OPT_TLS2_SIGN_CLIENT** **KEP_OPT_TLS2_SIGN_SERVER**	One Of (binary OR with one of below)	
	KEP_OPT_TLS2_PRF_MD5 **KEP_OPT_TLS2_PRF_SHA1** **KEP_OPT_TLS2_PRF_SHA224** **KEP_OPT_TLS2_PRF_SHA256** **KEP_OPT_TLS2_PRF_SHA384** **KEP_OPT_TLS2_PRF_SHA512**	One of (binary OR with one of above)	SHA-2 hashes may not be enabled in the SPAcc engine, consult your ICD for details.

Table 7: KEP Option Values

4.4.2 kep_load_keys()

The TLS operations require hash keys to be loaded into SPAcc contexts. The *kep_load_keys()* function performs this operation.

```
int kep_load_keys(kep_device *kep, int handle, void *s1, uint32_t s1len, void *s2, uint32_t s2len);
```

This will load the key pointed to by *s1* of length *s1len* into the context indicated to by *handle*. If the operation requires a second key it will be loaded into the adjacent context with the contents of *s2* of length *s2len*. It is up to the caller to split the master secret (and pad if the original length is odd). Note that if the operation does not require a second key (such as TLS 1.2 operations) then *s2* may be **NULL**.

4.4.3 kep_go()

Once the context has been opened and the keys loaded (as required) the caller may start a KEP job by calling the *kep_go()* function.

```
int kep_go(kep_device *kep, pdu_ddt *src_ddt, pdu_ddt *dst_ddt, int handle);
```

The source and destination DDT structures (*src_ddt* and *dst_ddt*) must be setup by the caller prior this call using the PDU DDT SDK. Once the job has been pushed on the command FIFO the caller may not modify the DDT entries nor the SPAcc hash context page.

4.4.4 kep_done()

A job can be retrieved from the status FIFO using the *kep_done()* function.

```
int kep_done(kep_device *kep, int handle) ;
```

This will poll the status FIFO returning **CRYPTO_INPROGRESS** if the job is pending and **CRYPTO_OK** if it has completed. It may pop other jobs off the status FIFO which can be discovered when the correct handle is passed to the function.

Note: that the KEP engine may exhibit undefined behaviour (seemingly timeout) if an unsupported hash is used as part of the job.

4.4.5 kep_close()

When a context is no longer required it can be freed back to the pool of available contexts (shared by the RE and SPAcc) with the *kep_close()* function.

```
int kep_close(kep_device *kep, int handle);
```

5 MPM Engine

5.1 Overview of MPM SDK

The Multi-Packet Manager engine allows the developer to offload SPAcc PDU descriptors from FIFO space inside the SPAcc engine to host system memory. In effect, allowing more jobs to be queued up at once increasing the throughput of the SPAcc engine at relatively lower cost.

The MPM SDK organizes user jobs in the form of PDU chains which can be in various states of execution allowing job queues to be built up while the engine is busy. The IRQ handlers start ready chains keeping the engine busy.

To avoid overhead the SDK operates with pre-allocated pools of PDU descriptors and key buffers using parameters that are run-time configurable. There are pre-allocated arrays of PDU/key indecies called *chains* which have a fixed (run-time) length of *links*. That is, a chain may only hold as many jobs as there are links.

The operation of the device through the SDK requires users to assign jobs to chains and then enqueue the chain for execution when sufficient work has been assigned. Chains will only be in contention for execution once the user enqueues them. Otherwise, a chain is merely in a “building” phase and cannot be run.

Each job can be assigned a callback which notifies the user when the job completes. This is how jobs are then sent back down the appropriate communication stack. By default, jobs are only notified when an entire chain completes, this is called the End Of List (EOL) interrupt. Alternatively, jobs can be marked as “on demand” which means that as soon as that job completes the interrupt fires and the callback is called.

On demand jobs are useful when one or two jobs in a long pre-existing not yet enqueued chain are high priority. If the chain is empty it is just as effective to simply enqueue the on demand jobs on their own. To keep track of the on demand jobs and their callbacks a secondary shorter array is maintained for them. The secondary array has less space than the master PDU array and as such it is possible that an on demand job won't fit. On demand jobs can be lower priority for two reasons. The first is that the secondary array is filled. When this happens the job is placed in the normal queue and not marked as on demand (no IRQ will fire for just that job unless it's the end of the chain and an EOL will fire). The second way is if the IRQ is triggered while the IRQ is busy. This could happen if on demand jobs are too close to each other. When this happens the regular EOL interrupt will pick up the jobs and call the callback registered. In both cases no jobs should be lost but they will not be handled with the ideal lower latency path.

There are tradeoffs to be made with respect to how long chains should be. Longer chains yield higher throughput but since the engine is busier but result in a longer latency between EOL interrupts. Since the user is only told their job is finished when EOL triggers this yields higher latency jobs. A shorter chain minimizes latency but decreases throughput since the engine is idle during IRQ events. It is up to the driver developer to determine the ideal chain length and enqueue policy based on their real or simulated live traffic. To this end there are performance counters available that track engine use and on demand statistics (such as missed jobs).

5.1.1 Callbacks

The user is notified that their job is finished by the use of callbacks. There are two types of callbacks used in this SDK, the per-job callback and the per-chain callback.

The per-chain callback is used for normal EOL (end of list) and ON DEMAND interrupts and is called for every single job that is executed. This gives the finest grain of control of how jobs are cleared. A use for this for instance might be where a user has two stacks (say IPSEC and SRTP) and has different callbacks for passing messages down their respective stacks but wishes to enqueue jobs when possible (possibly on the same chain). The prototype is:

```
void mpm_callback(void *mpm_dev, void *data, int pdu_idx, int key_idx, uint32_t status);
```

This is passed the pointer of the *mpm_device* structure, the PDU and key indecies and the status code out of the PDU descriptor (containing the error bits, the done bit, etc.). The *data* parameter is what is passed when the job was inserted into a chain. This allows a per-job [or chain] piece of data allowing different threads to share a single callback.

The per-chain callback is used when an EOL interrupt occurs and overrides any per-job callback on any non ON DEMAND job. This callback mechanism is used when a group of related jobs are to be run at once. The savings here is there are fewer function calls to retire the chain lowering the latency of the system. The prototype is:

```
void mpm_chain_callback(void *mpm_dev, void *chain_id);
```

This is passed the *mpm_device* structure and the ID of the chain as passed (user selected) when calling *mpm_enqueue_chain()*.

5.1.2 Chain States

When the library is initialized all of the chains are set to a **CHAIN_FREE** state. This means they may be assigned a **CHAIN_BUILDING** state and have jobs inserted into them. There is only one **CHAIN_BUILDING** chain at a time. When the chain is filled to the desired capacity it is enqueued which marks it as **CHAIN_BUILT**, there can be multiple chains in this state. When the SDK puts a chain in the MPM engine it marks the chain as **CHAIN_RUNNING**, and when it finishes as **CHAIN_DONE**. There can only be one chain running at a time but multiple can be marked as done. The *mpm_clear_chain()* function will mark chains as **CHAIN_PROCESSING** when they are being cleared (callbacks being called) and finally they return to **CHAIN_FREE** when all of the callbacks have been called and the PDUs returned to the free PDU pool.

5.2 Library Initialization

The MPM initialization occurs in two steps. First, the user calls the function *mpm_init()* with the number of chains, pdus, keys, etc. Second, the user then allocates physical memory for the PDU and key buffers.

```
int mpm_init(mpm_device *mpm, void *regmap, spacc_device *spacc,
             int no_chains, int no_pdus, int no_keys, int no_chain_links, int no_demand);
```

The user calls this function with a pointer to a *mpm_device* structure in *mpm*, the base address of the device in *regmap*, the SPAcc device it is attached to (VSPAcc 0 in VSPAcc devices) in *spacc*.

The next parameters indicate the parameter for in flight jobs.

Parameter	Use
no_chains	The number of chains in flight. Should be at least 4 and ideally no more than 16-24.
no_pdus	The number of PDU descriptors to make available. Each PDU descriptor requires 128 bytes of physically contiguous memory.
no_keys	The number of key buffers to make available. Each key buffer requires 256 bytes of physically contiguous memory. Ideally, there should not be more key buffers than PDU descriptors but this is allowable.
no_chain_links	The number of links per chain. Should be at least 16. The product of *no_chains* and *no_chain_links* is the number of jobs possible to enqueue and should not be less than *no_pdus*. It is acceptable for their to be more total links than *no_pdus* since it is possible that not all chains are completely full.
no_demand	The number of on demand jobs allowed. Should be at least 2 and no more than 16. Larger values increase the IRQ latency and smaller values decrease the ability to enqueue on demand jobs.

Table 8: mpm_init() parameters

The number of PDU and key buffers depends on the traffic model. Since they must each be contiguous in physical memory there is usually pressure on keeping them small. It is typical to have more PDU buffers than key buffers but the ratio strictly depends on key-reuse of the underlying protocols.

Once this function returns it fills in two parameters from the *mpm_device* structure the following two fields:

```
/* amount of bytes required by both */
size_t pdus_mem_req,
       keys_mem_req;
```

Both of these contain the amount of bytes to allocate for the PDU and key buffers respectively. The pointers *mpm_device.pdu.pdus* and *mpm_device.pdu.pdus_phys* must be set to the virtual and physical address of the PDU buffer. Similarly, *mpm_device.key.keys* and *mpm_device.key.keys_phys* for the KEY buffers. If the two buffers cannot be allocated the user should then call *mpm_deinit()* to free any memory that *mpm_init()* allocated on its behalf.

A typical linux initialization may resemble:

```
// initialize the SDK
err = mpm_init(&mpm, baseaddr, spacc,
      mpm_config_chains, mpm_config_pdus, mpm_config_keys, mpm_config_links, mpm_config_demand);
if (err != CRYPTO_OK) {
   return -1;
}
printk(KERN_DEBUG "mpm_init::Requires %d bytes for PDU and %d bytes for keys\n",
       mpm.pdus_mem_req, mpm.keys_mem_req);

// after we call mpm_init we need to allocate a pools
mpm.pdu.pdus = pdu_dma_alloc(mpm.pdus_mem_req, &mpm.pdu.pdus_phys);
if (!mpm.pdu.pdus) {
   printk(KERN_DEBUG "Cannot allocate PDUs pool\n");
   mpm_deinit(&mpm);
```

```
      return -1;
   }
   mpm.key.keys = pdu_dma_alloc(mpm.keys_mem_req, &mpm.key.keys_phys);
   if (!mpm.key.keys) {
      printk(KERN_DEBUG "Cannot allocate KEYs pool\n");
      pdu_dma_free(mpm.pdus_mem_req, mpm.pdu.pdus, mpm.pdu.pdus_phys);
      mpm_deinit(&mpm);
      return -1;
   }
```

At this point the SDK has been initialized and the user can begin allocating PDU and keys, inserting PDUs into chains and enqueing chains.

5.2.1 Assigning SPAcc Contexts

For the MPM to operate it requires at least one SPAcc context assigned to it. Multiple contexts allow key caching to occur to increase throughput. Contexts are assigned with the following function

```
int mpm_req_spacc_ctx(mpm_device *mpm, int no_ctx, int *ctxs);
```

This will attempt to allocate up to *no_ctx* contexts from the SPAcc and assign them to the MPM. The array *ctxs* is populated with the SPAcc context numbers or -1 if the allocation failed. The array must be kept live for the release function call:

```
int mpm_free_spacc_ctx(mpm_device *mpm, int no_ctx, int *ctxs);
```

This will free the allocated SPAcc contexts from the MPM and allow normal SPAcc (or RE/KEP) jobs to use them.

5.3 Library Deinitialization

The library can be freed up provided there are no pending jobs by calling the following function:

```
int mpm_deinit(mpm_device *mpm);
```

After this is called the user can then free the PDU and key buffers allocated externally.

5.4 Running Jobs

5.4.1 Theory of Operation

The theory of execution for the MPM is the following

1. User allocates a PDU buffer with *mpm_alloc_pdu()*
2. User optionally allocates a key buffer (different PDUs may use the same key buffer) with *mpm_alloc_key()*
3. User assigns key if using a new key buffer with *mpm_set_key()*
4. User inserts PDU into chain with *mpm_insert_pdu()*
5. Repeats steps 1-4 for as many jobs the user wishes to group in one chain
6. Calls *mpm_enqueue_chain()* and then schedules the single-issue task which calls *mpm_clear_chains()* (see section 5.4.2 and 5.4.3)
7. Waits for all callbacks to be called before signaling lower level network stacks that the pay-

loads are available.

5.4.2 IRQ Handler

The SDK requires an IRQ handler the can handle the following events:

1. ON DEMAND interrupt
    1. Figure out which job(s) just finished and call callback
    2. Clear ON DEMAND stat bit
2. EOL interrupt
    1. Clear EOL stat bit
    2. Mark current chain as done
    3. Enqueue next chain in sequence by writing to MPM_START
    4. Schedule single-issue task (see section 5.4.3) which in-turn calls *mpm_clear_chains()*

A sample IRQ handler is available in the file src/mpm/kernel/mpm.c in the function named *mpm_irq_handler()*. Ideally, a non-linux IRQ handler should be modeled directly after this since many inner SDK variables have to be accessed in specific manners for the SDK to function correctly. To make development easier two functions are provided:

```
void mpm_process_int_demand(mpm_device *mpm);
```

This function handles ON DEMAND interrupts.

```
void mpm_process_int_eol(mpm_device *mpm);
```

This function handles EOL interrupts. **Note**: that the caller of this function must still call or schedule a call to *mpm_clear_chains()* to process the callbacks for each job/chain.

5.4.3 Single-Issue Tasks

The SDK requires a single-issue task (called tasklets in linux) in order to clear the finished chains and possibly enqueue the next chain. The entire SDK starts with the user scheduling the task which then fires the very first job. Subsequent jobs may be issued from the IRQ handler if they are pending otherwise the engine returns to an idle state and the task is what will start the engine again.

The task must simply call *mpm_clear_chains()* in a single-issue fashion since the function does not perform any form of mutex or blocking. Two or more threads cannot simultaneously call *mpm_clear_chains()* and the user should never call it directly (outside the task).

In linux this is accomplished with a tasklet process which operate in the SOFTIRQ space (hence they cannot block). On other platforms this can be accomplished with a semaphore at the top of a never ending for-loop statement which issues a call to *mpm_clear_chains()*. Instead of scheduling the tasklet the semaphore can be incremented.

5.4.4 mpm_alloc_pdu()

This function returns the index to an allocated PDU descriptor or -1 if none are available.

```
int mpm_alloc_pdu(mpm_device *mpm);
```

There is no free function for PDU descriptors. They are returned to the pool of available PDUs when the job associated with it terminates. If the PDU is never inserted into a chain it will be lost.

5.4.5 mpm_alloc_key()

This function returns the index to an allocated key buffer or -1 if none are available.

```
int mpm_alloc_key(mpm_device *mpm);
```

Keys are maintained with a reference count (starting at 0) which prevents the same key buffer from being used by multiple different job under different keys at once. Each time a key is used on a job being inserted the reference counter is incremented. Each time the job pops off it is decremented. Only when the count reaches -1 does the key return to the free pool. The count can be decremented with following function:

```
int mpm_free_key(mpm_device *mpm, int key_idx);
```

This decrements the count by 1 and returns it to the pool if the count reaches -1.

5.4.6 mpm_set_key()

This function assigns the key sizes and contents to a cipher and hash key pair of a key context.

```
int mpm_set_key(mpm_device *mpm,
                int key_idx, int ckey_sz, int hkey_sz, int no_cache, int inv_ckey,
                unsigned char *ckey, unsigned char *hkey);
```

This sets the key sizes (in bytes) of the cipher and hash components to *ckey_sz* and *hkey_sz* respectively. If *no_cache* is set the key will be marked to not be cacheable, this is useful for keys that will not used more than once. If *inv_ckey* is set the key expansion/inversion will be performed when the cipher is used.

The key bytes are pointed to by *ckey* and *hkey* either of which may be **NULL** to indicate that the key is not to be filled out. The contents of the array are not zeroed before copying so a NULL pointer means that the respective key has uninitialized contents.

5.4.7 mpm_insert_pdu()

This function fills out a PDU descriptor and inserts it into the current building chain. If there is no current building chain (a chain marked **CHAIN_BUILDING**) it will find a free chain and convert it. If the current building chain is full (no more links) it will attempt to enqueue the chain (with no per-chain callback) and then start a new chain. This latter fail-safe mode is not the desired operation mode since it typically represents a failure to enqueue frequently enough, but is nonetheless provided to make multi-threaded development easier.

```
int mpm_insert_pdu(mpm_device *mpm,
                   int pdu_idx, int key_idx, int ondemand,
                   mpm_callback cb,  void *cb_data,
                   pdu_ddt *src, pdu_ddt *dst,
                   uint32_t pre_aad_len, uint32_t post_aad_len, uint32_t proc_len,
                   uint32_t icv_len, uint32_t icv_offset, uint32_t iv_offset, uint32_t aux_info,
                   uint32_t ctrl,
                   unsigned char *ciph_iv, unsigned char *hash_iv);
```

The function parameters are:

Parameter	**Description**
mpm	The MPM device
pdu_idx	The index of the PDU descriptor to insert
key_idx	The index of the key buffer to associate with this job
ondemand[4]	Non-zero to mark the job as ON DEMAND
cb	The callback for the job (may be **NULL** if no callback is desired)
cb_data	The callback data pointer associated with this job (may be shared amongst other jobs)
src	The source DDT structure (filled like a normal SPAcc job)
dst	The destination DDT structure
pre_aad_len	The pre-AAD length in bytes
post_aad_len	The post-AAD length in bytes
proc_len	The SPAcc PROC_LEN value (see EDN-0228)
icv_len	The length of the ICV in bytes (usually left to 0 for default)
icv_offset	The offset of the ICV
iv_offset	The offset of the IV inside the source if the most significant bit is set (see EDN-0278 section 10.1)
aux_info	The AUX INFO register (see EDN-0228)
ctrl	The CTRL register (see EDN-0278 section 10.1)
ciph_iv	The IV for the cipher (job dependent) may be **NULL** to leave empty
hash_iv	The IV for the hash (job dependent) may be **NULL** to leave empty

Table 9: mpm_insert_pdu() Parameters

The CTRL register can be filled out with the **MPM_SET_CTRL** macro:

```
#define MPM_SET_CTRL(ciph, hash, ciphmode, hashmode, encrypt, aadcopy, icv_pt, icv_enc, icv_append)
```

With the following parameters

4 On demand jobs may be inserted as regular priority if there is no space available in the on demand queue.

Parameter	Value	Parameter	Value

ciph	Cipher Algorithm (bits 2:0 of CTRL)
	`C_NULL   = 0`
	`C_DES    = 1`
	`C_AES    = 2`
	`C_RC4    = 3 (not supported)`
	`C_MULTI2 = 4`
	`C_KASUMI = 5`
	`C_SNOW3G_UEA2 = 6`
	`C_ZUC_UEA3 = 7`
		hash	Hash Algorithm (bits 7:4 of CTRL)
			`H_NULL   = 0`
			`H_MD5    = 1`
			`H_SHA1   = 2`
			`H_SHA224 = 3`
			`H_SHA256 = 4`
			`H_SHA384 = 5`
			`H_SHA512 = 6`
			`H_XCBC   = 7`
			`H_CMAC   = 8`
			`H_KF9    = 9`
			`H_SNOW3G_UIA2 = 10`
			`H_CRC32_I3E802_3 = 11`
			`H_ZUC_UIA3 = 12`
			`H_SHA512_224 = 13`
			`H_SHA512_256 = 14`

ciphmode	Cipher Chaining Mode (bits 11:8 of CTRL)
	`CM_ECB = 0`
	`CM_CBC = 1`
	`CM_CTR = 2`
	`CM_CCM = 3`
	`CM_GCM = 5`
	`CM_OFB = 7`
	`CM_CFB = 8`
	`CM_F8  = 9`
	`CM_XTS = 10`
		hashmode	Hash Mode (bits 13:12 of CTRL)
			`HM_RAW    = 0`
			`HM_SSLMAC = 1`
			`HM_HMAC   = 2`

encrypt	1 for encrypt, 0 for decrypt	*aadcopy*	1 to copy AAD to destination, 0 to not
icv_pt	Perform ICV over plaintext	*icv_enc*	Encrypt ICV
icv_append	Append ICV to end of payload		

Table 10: MPM_SET_CTRL Macro Parameters

5.4.8 mpm_enqueue_chain()

When a chain is filled with the desired amount of jobs it can be marked as executable by calling this function.

```
int mpm_enqueue_chain(mpm_device *mpm, mpm_chain_callback cb, void *id);
```

This will enqueue the current chain marked as **CHAIN_BUILDING** if and only if there is at least one job inserted. If no jobs are inserted the call is ignored (including the callback parameters). This allows multiple threads that are inserting jobs to call *mpm_enqueue_chain()* without first checking how many jobs are placed in the chain.

A per chain callback may be registered by passing *cb* as non-**NULL** and the value *id* will be passed to it. *Note:* that enabling a per-chain callback disables per-job callbacks (exception being ON DEMAND jobs). *cb* may be **NULL** to indicate that no per-chain callback is desired.

The chain is officially going to run however until the single-issue task calls *mpm_clear_chains()* which is accomplished in linux by calling *tasklet_schedule()*. In other platforms it will be accomplished by incrementing the semaphore blocking the task which calls *mpm_clear_chains()*.

5.5 Performance Tracking

When **MPM_PERF_MON** is enabled in the top level Makefile the SDK builds with performance trackers that help collection information about events such as interrupts, jobs scheduled, etc.

The function *mpm_stats()* allocates a buffer and fills it with a description of various counters:

```
char *mpm_stats(mpm_device *mpm)
```

Which returns a buffer that resembles the below text. The buffer must be freed with *pdu_free()* when done.

```
MPM: Performance Monitoring Stats
MPM: EOL Interrupts        :     4096
MPM: DEMAND Interrupts     :     4083
MPM: Jobs Inserted         :   131072
MPM: Jobs Cleared          :   131072
MPM: KEY Full Cache Hit    :        0
MPM: KEY Part Cache Hit    :     4082
MPM: KEY Miss              :   126990
MPM: DEMAND jobs hit       :     4083
MPM: DEMAND downgrade      :        0 (job posted as non-demand)
MPM: DEMAND missed         :       13 (normal EOL IRQ picked it up)
MPM: DEMAND hitrate        :    99.68 %
MPM: active_total          :  1379221
MPM: spacc_total           :  1298139
MPM: idle_total            :     3446
MPM: IRQ latency           :     5945
MPM: active rate           :    99.75 %
MPM: spacc rate            :    94.12 %
MPM: IRQ rate              :   172.51 %
MPM: Active Cycles per EOL :      336 (average)
```

In this sample, 4096 chains of 32 jobs (AES-128-CTR with same key over 120 bytes of plaintext) were run where one job per chain was marked as ON DEMAND. We see the 4096 EOL (end of list) interrupts but only 4083 DEMAND interrupts. This means we hit some latency in which the DEMAND interrupt handler did not respond in time. This is further confirmed by the “DEMAND missed” line which states that 13 of the DEMAND jobs were actually picked up by the EOL IRQ handler.

There are no “KEY Full Cache Hits” since the IV must be reloaded from the PDU descriptor for every job. We see some partial hits (the AES key) is hit, and many misses. The misses occur because in this case only one context was provided and the MPM engine can't pre-load any data.

Adding a second SPAcc context to the run-time inverts the numbers (130,000 partial hits and 418 full misses).

The “DEMAND hitrate” measures the percentage of jobs inserted as DEMAND which are actually executed as demand and picked up by the correct IRQ. In this case, no jobs were demoted to a normal job but 13 IRQs were missed most likely during other interrupts and those jobs were forced to be picked up by a normal EOL handler.

The “active_total” counter indicates the number of clock cycles (divided by 256) that the MPM engine was busy. This includes time it takes to fetch keys and PDU descriptors and the time the SPAcc engine is busy. The “spacc_total” counter indicates how busy the SPAcc is (in cycles divided by 256). A high rate of key cache misses would show up as a much higher active_total than a spacc_total. In this case, they are divergent because there are many cache mixxes.

The “idle_total” indicates how long the MPM engine is sitting idle after the first job was fired (it does not count the time between reset and the first job). A high idle time means that the engine is not processing PDUs actively.

The “IRQ latency” indicates the length of time between an IRQ being raised and the handler writing back to the MPM_IRQ_STAT register.

The “active rate” measures how much of the time spent busy and idle is the engine busy. For example, here we see 99.75% which means over the entire time the jobs were being processed the MPM was busy 99.75% of the time either fetching keys/IVs or waiting for the SPAcc to complete jobs.

The “spacc rate” measures how much time is spent actually in the SPAcc doing cryptographic work. Here we see 94.12% which means that of all the MPM active cycles it was only busy ciphering data 94.12% of the time, the other 6% of the time was spent either loading keys or waiting for the command FIFO to clear a slot.

The “IRQ rate” measures how much of the idle time is spent waiting for an interrupt. It only makes sense if there are no on demand jobs since you can end up with larger than 100% values.

The “Active Cycles per EOL” measures how many MPM active cycles are spent per EOL interrupt, it is an average taken over all jobs performed thus far.

5.5.1 Parameters

Ultimately the choice of parameters such as the number of chains, links, and how often they are enqueued will affect performance. There are common trends that have tradeoffs to be aware of.

Number of chains. The number of chains matters only if there are jobs waiting on free chains to be populated. In a typical benchmark case of full loading a higher amount of chains means there is less delay between the chain being enqueued and processed by the MPM. In effect the chains are being executed from the IRQ and the MPM has more of a chance of keeping pace with the host once jobs start running. Both cases of 16 and 32 chains (fully loaded with 32 jobs each) resulted in a plateau once the chains were full (e.g., at 16 and 32 chains respectively) that was lower for the case with higher active chain counts. The overall latency was higher for the 16 chain variant by a non-trivial amount. Having more chains gives more flexibility to run many chains at once and generally does not consume much more memory (provided there are not too many links per chain). It is therefore ideal to have more chains by default.

Links per chain. The number of links per chain effects the maximal throughput of the MPM as it limits the amount of traffic per interrupt. Fewer jobs per chain (while performing the same total number of jobs) results in quicker latency per EOL interrupt changes. That is, the latency per EOL rises quicker with (say) 16 links compared to 32 links. Both cases peak at the comparable latency (assuming the same number of chains in total the latency favours the more densely packed chains) and the 16 link case decreases in latency quicker towards the end of the cycle. Having fewer jobs per chain makes the system more adaptive to latency changes, it's therefore advantageous to have called *mpm_init()* with a maximal link count and then tailor the jobs per chain to meet load. The ideal case would be only maximize the jobs per chain once all chains are busy, that is, as you run out of free chains you increase the workload per chain. This results in the same overall latency once the engine is fully busy but gives the first chains a lower overall latency.

When the links per chain surpasses the SPAcc's command FIFO depth the MPM engine can end up waiting on the SPAcc while in a “busy” state. This means that the SPAcc active rate is not going to be helped much. For example, on AES-128-CTR jobs of 120-byte payloads the following active rates were seen (over 131,072 jobs):

Chain Length	**MPM Active Rate (%)**	**SPAcc Active Rate (%)**	**MPM Cycles**	**Cycles Between EOLs (/256)**
1	74	55	267,553	2
8	94	92	215,385	13
16	96	96	210,925	26
24	98	98	209,757	38
32	98	98	210,165	51
64	99	99	216,640	106

Table 11: Chain Length vs. FIFO Depth

As can be seen from the table jumping from a chain length of 1 job to 16 jobs shaves 57,000 MPM cycles (actually divided by 256 so in effect 14,592,000 cycles were saved) off the entire time all 131,072 jobs take to finish. This particular SPAcc has a FIFO depth of 16. As we can see as we approach 64 jobs per chain the MPM ends up waiting for FIFO slots to push jobs into. As a result the MPM active cycle count increases. For as little as 1% less SPAcc efficiency we have reduced the latency of job notification by a third by dropping down to 24 jobs per chain. At 32 jobs the MPM cycle count begins turning around (increasing) and we stop seeing an improvement in throughput of the SPAcc core in a meaningful sense. Therefore, in this case an optimal setting appears to be around 24 jobs per chain (when optimizing for MPM active time).

6 ESP/AH Offload

6.1 *Overview of the ESP/AH Engine*

The ESP/AH (EA) engine uses the SPAcc to perform ESP/AH packet processing. It uses an external SA memory to store context information for each IPsec connection direction while using the SPAcc cipher/hash contexts to perform the raw transforms.

The EA SDK supports multiple parallel EA contexts at once and multiple packets per SA. The SDK supports having more SAs active than SPAcc contexts by only using permissive locking of SPAcc contexts. That is, when there are no pending jobs it will (by default) not hold a SPAcc context for a given EA context. If too many contexts are in use the user will get context allocation failures returned.

6.1.1 Callbacks and Interrupts

Like the other SPAcc based engines the EA is interrupt driven and uses callbacks as the primary means of notifying a user when their packet is done.

The interrupt handler need only call the function *ea_pop_packet()* which will dequeue any pending packets. Users can also call that function if an IRQ was missed (or polling is desired). The *ea_pop_packet()* function then calls (optionally) the callback associated per packet. Each packet has its own callback information. That is, even on a single EA context each packet can call a different callback.

The ideal configuration for most traffic patterns is to have the STAT IRQ fire when the count in the STAT FIFO hits the value specified in *ea.config.ideal_stat_limit*. In addition to that the STAT_WD IRQ can be triggered to fire when the device goes idle with jobs pending in the STAT FIFO.

6.1.2 IRQ Macros

To facilitate turning on and off interrupts the following macros are provided.

```
EA_IRQ_ENABLE_GLBL(ea)
```

This enables global interrupts and should be the last value enabled in the IRQ_EN register.

```
EA_IRQ_ENABLE_CMD(ea, count)
EA_IRQ_ENABLE_CMD1(ea, count)
EA_IRQ_ENABLE_STAT(ea, count)
EA_IRQ_ENABLE_STAT_WD(ea, count)
```

These turn on the CMD0, CMD1, STAT and STAT_WD interrupts respectively with programmed counts of *count* (to the IRQ_CTRL register). The count provided to the STAT_WD IRQ is the number of clock cycles the device is allowed to idle (with jobs in the STAT FIFO) before the IRQ is raised. For instance, 250,000 would be 1 millisecond at 250MHz.

To turn off interrupts the following functions are provided:

```
EA_IRQ_DISABLE_GLBL(ea)
EA_IRQ_DISABLE_CMD(ea)
EA_IRQ_DISABLE_CMD1(ea)
EA_IRQ_DISABLE_STAT(ea)
```

```
EA_IRQ_DISABLE_STAT_WD(ea)
```

For example, the following code enables interrupts in the Linux kernel module.

```
   EA_IRQ_ENABLE_STAT(&ea, ea.config.ideal_stat_limit);
   EA_IRQ_ENABLE_STAT_WD(&ea, timer);
   EA_IRQ_ENABLE_GLBL(&ea);
```

6.2 Library Initialization

The library is initialized with the following call (after the SPAcc SDK has been initialized):

```
int ea_init(ea_device *ea, spacc_device *spacc, void *regmap, int no_ctx);
```

This initializes the device structure *ea* for use. The *spacc* pointer is the pointer to the SPAcc the EA is attached to (vSPAcc 0). The *regmap* pointer is the base address of the EA hardware registers. The *no_ctx* parameter is the number of EA contexts desired. There can be more contexts than SPAcc contexts but each context takes 256 bytes of DMA'able memory and an additional 100 in virtual memory.

The *ea_device* structure has the following (relevant) definition:

```
typedef struct {
  spacc_device *spacc;
  void *regmap;

  struct ea_config {
     uint32_t cmd0_depth,
              cmd1_depth,
              sp_ddt_cnt,
              ipv6,
              ar_win_size,
              stat_depth,
              ideal_stat_limit,
              op_mode,
              tx_fifo,
              rx_fifo;
  } config;

  ea_ctx *ctx;
  int     num_ctx;

  void           *sa_ptr_virt;
  PDU_DMA_ADDR_T  sa_ptr_phys;

  PDU_LOCK_TYPE   lock;

  size_t sa_ptr_mem_req;
} ea_device;
```

The *ea_init()* function populates the *sa_ptr_mem_req* field with the amount of DMA'able memory required for the SA. Like the MPM and RE the caller must then set the pointers *sa_ptr_virt* and *sa_ptr_phys*. For instance:

```
err = ea_init(&ea, spacc, baseaddr, 8);
if (err) {
   return -1;
}

// after we call re_init we need to allocate a pool
printk("EA requires %zu bytes for SA\n", ea.sa_ptr_mem_req);
ea.sa_ptr_virt = pdu_dma_alloc(ea.sa_ptr_mem_req, &ea.sa_ptr_phys);
if (!ea.sa_ptr_virt) {
   printk("Cannot allocate SA pool\n");
   return -1;
}
```

Will allocate the required memory in the Linux kernel.

The *config* structure inside the *ea_device* structure contains relevant information about the EA device. The *cmd0_depth*, *cmd1_depth*, and *stat_depth* are the depths of the respective FIFOs in the device. This can be used to determine how to prioritize the FIFOs for job processing.

6.3 Deinitialization

To deinitialize the SDK all pending jobs must be finished, the *ea_deinit()* function is then called, and finally the SA memory freed.

```
int ea_deinit(ea_device *ea);
```

The SA memory can be freed in Linux with:

```
dma_free_coherent(NULL, ea.sa_ptr_mem_req, ea.sa_ptr_virt, ea.sa_ptr_phys);
```

6.4 Mode of Operation

The EA device supports up to two command FIFOs which may be asymmetric in size. This allows flexibility in how jobs are prioritized into the core. If CMD1 is not present the SDK can only run using a single CMD FIFO.

```
int ea_set_mode(ea_device *ea, int mode);
```

This function sets the SDK mode of operation where *mode* is one of the following:

- **EA_OP_MODE_FULL_DUPLEX_FAVTX**
  - Favour the longer CMD FIFO for outbound traffic
- **EA_OP_MODE_FULL_DUPLEX_FAVRX**
  - Favour the longer CMD FIFO for inbound traffic

In the first two modes the SDK will favour the longer of the two CMD FIFOs (or simply CMD0 if they are the same length) for their respective traffic. If either of the FIFOs are full when a job is being added the SDK will return **EA_FIFO_FULL**. These modes are useful when the traffic patterns are asymmetric (such as TCP transmit/receive with SYN/ACK packets). By default, the SDK will initialize favouring TX which for the cases of the CMD FIFOs being symmetric (or CMD1 not present) results in a balanced transmission case.

6.5 Theory of Operation

The typical theory of operation is:

1. Once an SA is loaded by the user allocate an EA handle with *ea_open()*
    1. You can optionally pass *lock_ctx* as 1 to ensure higher performance
2. Load the SA with *ea_build_sa()* with the keys and other parameters
3. For each packet:
    1. Lock a mutex-like variable (*like a completion in linux)*
    2. Fire the packet with *ea_go()*
        1. Set the callback to unlock the mutex
    3. Wait on the mutex
        1. Alternatively you can loop calling *ea_pop_packet()* and *ea_done()* in turn, this will consume more CPU time
    4. Process the packet further up or down (depending on direction) the IP stack
4. When finished with the SA call *ea_close()*
    1. This call may end up not freeing the EA context immediately if there are still jobs pending.

6.6 ea_open()

This function is used to open a new EA context.

```
int ea_open(ea_device *ea, int lock_ctx);
```

The *lock_ctx* parameter is used to indicate whether a SPAcc context should be allocated for this EA context at all times. This ensures priority when using the SPAcc resources. Normally this should be left as 0 to ensure better co-operation between other EA contexts.

The function returns a handle upon success or a negative value on error.

6.7 ea_build_sa()

This function is used to load an SA into an EA context.

```
int ea_build_sa(ea_device *ea, int handle, ea_sa *sa);
```

Which loads from the structure *ea_sa*:

```
typedef struct {
   uint32_t ctrl,
            spi,
            seqnum[2],
            hard_ttl[2],
            soft_ttl[2];
   unsigned char
            armask[32],
            ckey[32],
            csalt[4],
            mackey[64];
} ea_sa;
```

See EDN-0278 for more information about the SA fields.

6.7.1 CTRL Fields

The CTRL register inside the SA controls cipher and mac choices, key sizes, mac lengths, and a slew of other parameters. There are macros in *elpeahw.h* to deal with this field, the output of the macros can be OR'ed together to form the 32-bit CTRL register.

Macro Name	Uses	Parameters
SA_CTRL_ACTIVE	Marks the SA as active and ready to use. Should always be 1	
SA_CTRL_SEQ_ROLL	Allows the sequence number to rollover, otherwise marks the active bit as 0.	
SA_CTRL_TTL_EN	Enables TTL calculations	
SA_CTRL_HDR_TYPE	0=ESP,1=AH	
SA_CTRL_AR_EN	Enables anti-replay	
SA_CTRL_AR_WINSIZE(x)	Sets anti-replay window size	0=32pkts, 1=64pkts, 2=128pkts, 3=256pkts
SA_CTRL_IPV6	0=IPv4, 1=IPv6	
SA_CTRL_DST_OPT_MODE	Configures how IPv6 destination option headers are treated	
SA_CTRL_ESN	Enables extended sequence number processing	
SA_CTRL_CKEY_LEN(x)	Sets the cipher key length	**CKEY_LEN_128** => aes-128,des **CKEY_LEN_192** => aes-192,3des **CKEY_LEN_256** => aes-256
SA_CTRL_MAC_LEN(x)	Sets the AEAD mac length	**MAC_LEN_64** => 64-bits **MAC_LEN_96** => 96-bits **MAC_LEN_128** => 128-bits
SA_CTRL_CIPH_ALG(x)	Sets the cipher algorithm	**CIPH_ALG_NULL** => No cipher **CIPH_ALG_DES_CBC** => DES or 3DES **CIPH_ALG_AES_CBC** **CIPH_ALG_AES_CTR** **CIPH_ALG_AES_CCM** **CIPH_ALG_AES_GCM** **CIPH_ALG_AES_GMAC**
SA_CTRL_MAC_ALG(x)	Sets the mac algorithm	**MAC_ALG_NULL** => No hash **MAC_ALG_HMAC_MD5_96** **MAC_ALG_HMAC_SHA1_96** **MAC_ALG_HMAC_SHA256_128** **MAC_ALG_HMAC_SHA384_192** **MAC_ALG_HMAC_SHA512_256** **MAC_ALG_AES_XCBC_MAC_96**

Macro Name	Uses	Parameters
		MAC_ALG_AES_CMAC_96

Table 12: ESP/AH SA CTRL Field

6.8 ea_go()

This function is used to enqueue an IPsec packet in the engine. The SA must be setup with the *ea_build_sa()* function first.

```
int ea_go(ea_device *ea, int handle, int direction,
          pdu_ddt *src, pdu_ddt *dst,
          ea_callback cb, void *cb_data);
```

This will place a packet pointed to by *src* and destined to *dst* in the queue using a specific EA context indicated by *handle*. The *direction* bit is set to 1 for outbound and 0 for inbound packets. The callback is specified by *cb* which is a pointer to a callback function. It maybe **NULL** to indicate no callback is desired. The callback has the following prototype:

```
void ea_callback(void *ea_dev, void *data,
                 uint32_t payload_len, uint32_t retcode, uint32_t sttl, uint32_t swid);
```

The *cb_data* value is a pointer that will be passed by the EA SDK to the callback when the job completes. This is useful for passing a mutex pointer (or similar structure) to the callback to unblock the user. The *cb_data* value is passed as *data* to the callback. The *payload_len* receives the length of the output, *retcode* the EA error code (Table #17 in EDN-0278), *sttl* the software TTL trip value, and *swid* the software ID value of the job.

The *ea_go()* function returns the software ID of the job enqueued which can then be passed to *ea_done()* to check if the job is finished.

If the EA CMD FIFO is full the function will return **EA_FIFO_FULL**. If there are no available SPAcc contexts it will return **EA_NO_CONTEXT**. In both of these cases the user may retry the exact same job after waiting. If the function encounters a fatal error it will return **EA_ERROR**.

6.9 ea_close()

This function is used to close an EA context.

```
int ea_close(ea_device *ea, int handle);
```

This will close an EA context allocated by *ea_open()*. If there are jobs pending it will mark the context as to be dismissed but won't actually free it until the last job posts.

6.10 ea_done()

This function is used to indicate if a particular packet has terminated.

```
int ea_done(ea_device *ea, int handle, int swid);
```

This returns -1 on error, 0 if not finished (or not found), and 1 if finished. This function is not meant to be used routinely since polling for job completion is slow. It is merely available to round out the API.

Note: If an IRQ is missed this function will never see the job as finished. The user would have to then call *ea_pop_packet()* to fetch jobs out of the EA FIFO.

6.11 ea_pop_packet()

This function is used to pop finished jobs off the EA FIFO.

```
int ea_pop_packet(ea_device *ea);
```

This function pops any pending jobs off the EA FIFO and calls the appropriate (if any) callbacks for the jobs that are finished. This function is typically called from an interrupt handler but the user may call it as well if polling is desired.

7 CLP-27 (TRNG)

7.1 Library Initialization

The library is initialized with a call to *trng_init()*.

```
int32_t trng_init (trng_hw * hw, uint32_t reg_base, uint32_t enable_irq, uint32_t reseed);
```

Where *reg_base* is the address in virtual memory of the TRNG device. *enable_irq* set to 1 to enable interrupts and *reseed* set to 1 to enable a HW RNG reseed.

7.2 Library Termination

The library can be deinitialized with the following function.

```
void trng_close (trng_hw * hw);
```

This will disable interrupts and stop data generation.

7.3 Data Generation

Data generation with the TRNG device requires a reseed operation which can be performed either with the HW RNG source or with a user supplied nonce reseed value.

HW RNG reseeding is ideal since it initializes the TRNG device with truly random bits, however it is also quite slow. Nonce reseeding requires the user to have a supply of entropy to initialize the TRNG with and is very fast. Nonce reseeding is ideal for resuming the TRNG from a previously HW RNG reseeded state.

7.3.1 trng_reseed_random()

The random reseed function triggers the TRNG to reseed itself from the RNG rings.

```
int32_t trng_reseed_random (trng_hw * hw, uint32_t wait, int lock);
```

The caller may instruct the function to *wait* for the operation to finish before moving on. The re-seed operation can take a noticeable amount of time to complete which is why waiting is optional. The *lock* parameter is used to enable/disable locking, this is useful when calling this function from a previously locked state (for example part of a Linux RNG stack). Passed as zero the function will not lock, non-zero causes it to lock.

7.3.2 trng_reseed_nonce()

The nonce reseed function initializes the TRNG with a buffer supplied by the user.

```
int32_t trng_reseed_nonce (trng_hw * hw, uint32_t seed[TRNG_NONCE_SIZE_WORDS], int lock);
```

This allows initializing the TRNG with entropy collected by the user in the advent that waiting for the HW RNG to initialize is not acceptable. The seed is a set of 8 32-bit words. Neither the first 4 or last 4 may be all zeroes. The *lock* parameter is used to enable/disable locking, this is useful when calling this function from a previously locked state (for example part of a Linux RNG stack). Passed as zero the function will not lock, non-zero causes it to lock.

7.3.3 trng_rand()

To read from the TRNG the following function is available.

```
int32_t trng_rand (trng_hw * hw, uint8_t * randbuf, uint32_t size, int lock);
```

This fills the buffer pointed to by *randbuf* with *size* bytes of random data from the TRNG device. The TRNG device must be properly seeded before this function is called. The *lock* parameter is used to enable/disable locking, this is useful when calling this function from a previously locked state (for example part of a Linux RNG stack). Passed as zero the function will not lock, non-zero causes it to lock.

8 PKA

8.1 Library Initialization

The PKA is a slave-only device with no meaningful persistent state. Hence, no initialization or termination is required. Instead, all functions which access the hardware take a pointer to the base of the PKA memory map as the first argument.

8.2 Device Configuration

Some of the PKA's features may be affected by the hardware configuration. To determine the available features, use the function

```
int elppka_get_config(uint32_t *regs, struct pka_config *config);
```

which reads the global feature registers and fills out the structure pointed to by *config*. The structure members can then be inspected to influence software behaviour. The *pka_config* structure is defined as follows:

```
struct pka_config {
   unsigned fw_ram_size;
   unsigned fw_rom_size;
   unsigned rsa_size;
   unsigned ecc_size;
   unsigned alu_size;
};
```

The *fw_ram_size* and *fw_rom_size* members indicate the size of the firmware RAM and ROM memories, respectively, in 32-bit words. If either of these counts are zero, then there is no memory of the corresponding type. If *both* members are zero, then the PKA has been configured to use an external firmware memory and its behaviour is out of the scope of this SDK.

The *rsa_size* and *ecc_size* members specify the maximum supported operand size for RSA and ECC operations, respectively, in bits.

The *alu_size* member indicates the PKA's internal ALU width which is useful only for diagnostic purposes.

For compatibility with earlier versions of this SDK, another function is provided

```
void elppka_fw_size(uint32_t *regs, unsigned *ramsize, unsigned *romsize);
```

which only reads the firmware RAM and ROM sizes (as described above) and stores them in **ramsize* and **romsize*, respectively. New applications should use the *pka_get_config* function instead.

8.3 Operands

PKA operand memory can be accessed by the following functions:

```
int elppka_load_operand(uint32_t *regs, unsigned bank, unsigned index,
                                        unsigned size, const uint8_t *data);
int elppka_unload_operand(uint32_t *regs, unsigned bank, unsigned index,
                                          unsigned size, uint8_t *data);
```

These functions are identical except that the first copies *size* bytes from the buffer pointed to by *data* into the PKA, and the second copies *size* bytes from the PKA, storing the result into the buffer pointed to by *data*. The operand location is specified by *bank* and *index*. *Bank* must be set to one of PKA_OPERAND_A, PKA_OPERAND_B, PKA_OPERAND_C or PKA_OPERAND_D. *Index* specifies the number within the bank. For example, to access operand D3, set *bank* to PKA_OPER-AND_D and *index* to 3. Because of hardware requirements, *size* must be at least 20 (for 160-bit operations) and at most 512 (for 4096-bit operations).

The meaning of PKA operands depends on the firmware in use; see the documentation for your firmware image for details.

8.3.1 PKA Endian

Internally, the PKA stores operands as a sequence of 32-bit words, with the least significant word first. On the other hand, software convention is usually for large integers to be stored as a sequence of bytes, with the most significant byte first. The PKA SDK uses that software convention for loading and unloading operands, and the data buffer is assumed to be stored with the most significant byte first. The SDK will convert between this format and the PKA internal format automatically. However, the hardware byte swapper must be configured properly by the host software prior to loading any operands. This can be done by a call to the function

```
void elppka_set_byteswap(uint32_t *regs, int swap);
```

which enables the byte swapper if *swap* is set to a non-zero value. Otherwise, it disables the byte swapper. Little-endian hosts should *enable* the byte swapper; big-endian hosts should leave it disabled.

8.4 Running the PKA

An operation can be started on the PKA by calling

```
int elppka_start(uint32_t *regs, uint32_t entry, uint32_t flags, unsigned size);
```

where *entry* specifies the firmware entry point (as an offset in 32-bit words) and *size* specifies the initial operand size. The *flags* parameter provides an initial setting for the PKA flags register, which should normally be set to zero. See the documentation for your firmware image for details about the entry points. All relevant operands must be loaded prior to starting the PKA. The PKA signals completion of the operation by raising an interrupt. See EDN-0243 (PKA User Guide) for details on accessing the PKA interrupt status. The result can be obtained by calling

```
int elppka_get_status(uint32_t *regs, unsigned *code);
```

which returns 0 if the PKA is idle or CRYPTO_INPROGRESS if the PKA is still running. If this function returns 0 and *code* is not a null pointer, the PKA return status is stored in **code*. The meaning of the return status values are described in the PKA user guide and in the firmware documentation.

9 SASPA

The SASPA is essentially a collection of smaller cores sharing a common context and message data interface. Each core has its own register block. In the following section, different functions may take different register base addresses depending on which core they work with. The function prefix will determine which register block you should pass. Functions whose name starts with *elpsaspa* should be passed the global register block, functions whose name starts with *elpaes* should be passed the AES register block, and so on.

9.1 Device Configuration

Many of the SASPA's features are affected by the hardware configuration. To determine the available features, use the function

```
void elpsaspa_get_config(uint32_t *regs, struct saspa_config *config);
```

which reads the global feature registers and fills out the structure pointed to by *config*. This config structure may subsequently be used as a parameter for a few other SDK functions, or its members may be inspected directly.

9.2 Shared Memory

9.2.1 Shared Memory Allocation

Shared memory may be allocated and freed from the pool with the following functions:

```
long saspa_alloc_ctx(struct device *dev, size_t size);
long saspa_alloc_data(struct device *dev, size_t size);
void saspa_free(struct device *dev, long handle, size_t size);
```

The allocation functions return a handle (or -1 on error) to the memory. The *ctx* version is used to allocate key/hash context pages and the *data* version to allocate buffers for processing data. They both allocate from opposite ends of the shared memory pool. Operating under the theory that this will leave larger contiguous free blocks for the data side.

Both types of handles can be freed with the *saspa_free()* function.

9.2.2 Shared Memory Access

The SASPA shared memory can be *directly* accessed by the following functions:

```
void elpsaspa_write_mem(uint32_t *dst, const void *src, size_t size);
void elpsaspa_read_mem(void *dst, uint32_t *src, size_t size);
```

which operate in a manner similar to *memcpy,* writing to or reading from the SASPA shared memory, respectively. The shared memory consists of 32-bit words, however *size* need not be a multiple of 4 bytes, nor is any particular alignment for the host buffers required. If *size* is not divisible by 4, then (when writing to shared memory) the data will be padded out with zero bytes when writing the last word. When reading, only *size* bytes will be stored in the host buffer.

The memory can be indirectly accessed with the following functions:

```
int saspa_mem_write(struct device *dev, long handle, const void *src, size_t size);
int saspa_mem_read(struct device *dev, void *dst, long handle, size_t size);
int saspa_mem_addr(struct device *dev, long handle);
```

They use handles provided by the *saspa_alloc_*()* functions. The *addr* function retrieves the address relative to the start of the shared memory of where a handle is referencing. This is used to program the HMAC core with the address of the key.

These functions require the SASPA byte swapper to be configured appropriately for the host's endian, otherwise data will not be written or read correctly. To configure the byte swapper, use the function

```
void elpsaspa_set_byteswap(uint32_t *regs, int swap);
```

which enables the byte swapper if *swap* is non-zero. Otherwise, it disables the byte swapper. Little endian hosts should *enable* the byte swapper, big endian hosts should disable it.

9.3 AES

The AES core in the SASPA has a complicated context layout, and fields can move around depending on the core configuration. To help determine the context layout, one can use the function

```
void elpaes_get_ctx_layout(const struct saspa_config *config, struct saspa_aes_layout *layout);
```

which fills in the structure pointed to by *layout* according to the configuration parameters pointed to by *config*, which should be obtained by an earlier call to *elpsaspa_get_config*. The layout structure describes the variable components of the AES context, and does not describe fields which have fixed offsets.

The struct is defined as follows

```
struct saspa_aes_layout {
   unsigned char total_size;
   unsigned char tag_offset;
   unsigned char ctr_offset;
   unsigned char cbc_offset;
};
```

The *total_size* member is the size of the AES context page in bytes: each context requires that many bytes of shared memory. The *tag_offset* specifies the offset from the start of the context (in bytes) where MAC tags are stored. The *ctr_offset* specifies the offset from the start of the context of the initial counter value (for CTR and CCM modes), or the XCBC K1 value (for 1-key XCBC mode), or the XCBC K3 value (for 3-key XCBC mode). The *cbc_offset* specifies the offset from the start of the context of the initial value for CBC/CCM/XCBC and CMAC modes.

Not all configurations have all the offset parameters defined. If a context field does not exist for the core's configuration, then the corresponding offset member will be set to (unsigned char)-1. Normally, only the *total_size* member will be interesting, because the function

```
int elpaes_build_ctx(void *out, int mode, const struct saspa_aes_ctx *ctx,
                     const struct saspa_aes_layout *layout);
```

can be used to actually construct an AES context structure to pass to the engine. The *out* member must point to a buffer at least *layout->total_size* bytes long, and *mode* designates the cipher mode that this context will be used for. The *layout* member points to a the layout structure obtained by an earlier call to *elpaes_get_ctx_layout*. The actual context values are stored in *ctx*. This structure is used as if it were declared as follows:

```
struct saspa_aes_ctx {
   unsigned char saspa_aes_key[32];        /* 16, 24 or 32-byte AES key. */
   unsigned char saspa_aes_cbc_iv[16];     /* IV for CBC, CCM, CMAC and XCBC modes */
   unsigned char saspa_aes_ctr_iv[16];     /* Initial counter for CTR and CCM modes. */
   unsigned char saspa_aes_xts_key[16];    /* Tweak key for XTS. */
   unsigned char saspa_aes_xcbc_k1[16];    /* Only for 1-key XCBC mode. */
   unsigned char saspa_aes_xcbc_k2[16];    /* K2 for 3-key XCBC mode. */
   unsigned char saspa_aes_xcbc_k3[16];    /* K3 for 3-key XCBC mode. */
   unsigned char saspa_aes_f8_iv_s[16];    /* IV/Sj-1 */
   unsigned char saspa_aes_f8_salt_j[16];  /* SALT/j */
   unsigned char saspa_aes_f8_salt_iv[16]; /* SALT/IV' */
   unsigned char saspa_aes_tag[16];        /* MAC tag for CCM, CMAC and XCBC modes. */
};
```

except that to keep the structure size small, most of these fields are actually union members and the above member names are actually macros. Thus, one should only assign values to the fields relevant to the particular cipher mode being used. The resulting context will be written to the buffer specified by *out*, and may subsequently be written into the SASPA's shared memory.

A routine to invoke AES jobs has been provided in src/saspa/kernel/aes.c in the function *saspa_aes_run()*.

```
int saspa_aes_run(struct device *dev,
                  int encrypt, int mode, int keysize, const struct saspa_aes_ctx *ctx,
                  void *_msg, size_t msglen, size_t aadlen, uint32_t *stat);
```

This function will encrypt/decrypt a buffer of memory passed by the caller. It handles allocating context and data memory out of the SASPA buffer. An example usage of this function can be found in src/saspa/kernel/saspadiag.c.

9.4 DES

The DES core provides access to ECB and CBC chaining modes as well as single and triple DES key sizes. The contexts are stored in the *saspa_des_ctx* structure and are stored into memory by the *elpdes_build_ctx* function.

```
struct saspa_des_ctx {
   unsigned char key[24];
   unsigned char iv[8];
};
int elpdes_build_ctx(void *_out, const struct saspa_des_ctx *ctx);
```

For single DES modes the first 8 bytes of *key* are used (*key*[0...7]). For triple DES modes all 24 bytes are used in order of K0, K1, and K2.

The function *elpdes_build_ctx* stores the 32-byte DES context in the buffer *_out* which can then be written to the SASPA shared memory to be used in processing data through the DES core.

A routine to invoke DES jobs has been provided in src/saspa/kernel/des.c in the function *saspa_des_run()*.

```
int saspa_des_run(struct device *dev, int encrypt, int mode, int keysize,
                  const struct saspa_des_ctx *ctx, void *_msg, size_t msglen);
```

This function will encrypt/decrypt a buffer of memory passed by the caller. It handles allocating context and data memory out of the SASPA buffer. An example usage of this function can be found in src/saspa/kernel/saspadiag.c.

9.5 HMAC

The HMAC core provides access to hashing and HMACing of buffered data. The contexts are stored in the *saspa_hash_ctx* structure.

```
struct saspa_hash_ctx {
   unsigned char curiv[64],
                 key[64];
   uint32_t      sslmac_seq;
};
```

The state of the hash can be stored into and retrieved from the *curiv* field. They HMAC key can be stored in *key* from byte 0 onward for the length of the desired key. The lower 32-bits of the SSL sequence number can be stored in *sslmac_seq* for SSLMAC computations.

A routine to invoke HMAC jobs has been provided in src/saspa/kernel/hash.c in the function *saspa_hash_run()*.

```
int saspa_hash_run(struct device *dev, int mode, int sslmac, int keysize,
                  const struct saspa_hash_ctx *ctx,
                  const void *_msg, size_t msglen, void *_out, size_t outlen);
```

This function will hash or HMAC a buffer of memory passed by the caller. It handles allocating context and data memory out of the SASPA buffer. An example usage of this function can be found in src/saspa/kernel/saspadiag.c.

10 EAPE ESP/AH Engine

The EAPE core is a high performance multi-pipelined engine for performing IPsec packet processing. The SDK for controlling this core is similar to that of the SPAcc-EA (see section 6) where the significant difference from a programming point of view is the SA structure.

Like the SPAcc-EA the EAPE supports numerous contexts (active SAs) wherein each context may have numerous active (pending) packets being processed.

10.1 Callbacks and Interrupts

Like the other SPAcc based engines the EAPE is interrupt driven and uses callbacks as the primary means of notifying a user when their packet is done.

The interrupt handler need only call the function *eape_pop_packet()* which will dequeue any pending packets (from both pipelines). Users can also call that function if an IRQ was missed (or polling is desired). The *eape_pop_packet()* function then calls (optionally) the callback associated per packet. Each packet has its own callback information. That is, even on a single EAPE context each packet can call a different callback.

The callback function has the prototype

```
void eape_callback(void *eape_dev,
                    void *data, uint32_t payload_len, uint32_t retcode, uint32_t swid);
```

Which is called whenever a packet is popped off the inbound or outbound pipelines.

10.2 Library Initialization

The EAPE SDK is initialized with the following function:

```
int eape_init(eape_device *eape, void *regmap, int no_ctx);
```

Where *regmap* is the virtual address of the control registers and *no_ctx* is the number of desired contexts (at least 32 should be allocated).

This function fills in the *sa_ptr_mem_req* field of the *eape_device* structure. It indicates the number of bytes required for the SA pool for the contexts being used. The caller must then allocate the memory and set the virtual and physical pointers to *sa_ptr_virt* and *sa_ptr_phys* from the *eape_device* structure. An example initialization taken from *src/eape/kernel/eape.c* is

```
/* init the library */
 err = eape_init(&eape, baseaddr, 32);
 if (err) {
    return -1;
 }

 // after we call re_init we need to allocate a pool
 printk("EAPE requires %zu bytes for SA\n", eape.sa_ptr_mem_req);
 eape.sa_ptr_virt = pdu_dma_alloc(eape.sa_ptr_mem_req, &eape.sa_ptr_phys);
 if (!eape.sa_ptr_virt) {
    printk("Cannot allocate SA pool\n");
```

```
      return -1;
   }
```

Where this uses *pdu_dma_alloc* to allocate coherent memory for the device.

To deinitialize the library simply call *eape_deinit()* and then free the SA pool.

10.3 Theory of operation

The EAPE SDK is meant to be used in a fairly straightforward fashion. Processing IP datagrams is performed as follows:

1. Allocate a context with *eape_open()*
2. Initialized the SA with *eape_build_sa()* by filling out the fields in the *eape_sa* structure.
3. For each packet
    1. Initialize the source and destination DDTs
    2. Enqueue the packet with *eape_go()*
    3. When the IRQ fires it will call your callback which you can then use to retire the packet through the IP stack
4. When finished with the SA call *eape_close()* to release the context

10.4 eape_open()

To allocate a handle the *eape_open()* function is used.

```
int eape_open(eape_device *eape);
```

This function returns the handle of the context or -1 if an error occurred.

10.5 eape_build_sa()

To program an SA with the key material and flags required to process packets the *eape_build_sa()* function is used.

```
int eape_build_sa(eape_device *eape, int handle, eape_sa *sa);
```

This programs the SA page for the given context handle with the contents of the pointed to *eape_sa* structure. It returns 0 on success.

The *eape_sa* structure is defined as follows:

```
typedef struct {
   uint32_t spi,
            flags,
            seqnum[2],
            hard_ttl[2],
            soft_ttl[2];
   unsigned char
            alg,
            armask[8],
            ckey[32],
            csalt[4],
            mackey[64];
} eape_sa;
```

The fields have the following definitions:

Field	Use		Field	Use
spi	The SPI of the IPsec SA		flags	Controls the actions of the SA, has the following flags **EAPE_SA_FLAGS_ACTIVE** **EAPE_SA_FLAGS_SEQ_ROLL** **EAPE_SA_FLAGS_TTL_EN** **EAPE_SA_FLAGS_TTL_CTRL** **EAPE_SA_FLAGS_HDR_TYPE** **EAPE_SA_FLAGS_AR_EN** **EAPE_SA_FLAGS_IPV6** **EAPE_SA_FLAGS_DST_OP_MODE** **EAPE_SA_FLAGS_ESN_EN**
seqnum	Sequence number: should start at zero		hard_ttl	Hard TTL limit in which a failure will be indicated
soft_ttl	Soft TTL limit in which a soft error will be indicated.		alg	Controls which cipher/hash are used, has the following flags **EAPE_SA_ALG_AUTH_NULL** **EAPE_SA_ALG_AUTH_MD5_96** **EAPE_SA_ALG_AUTH_SHA1_96** **EAPE_SA_ALG_AUTH_SHA256** **EAPE_SA_ALG_AUTH_GCM_64** **EAPE_SA_ALG_AUTH_GCM_96** **EAPE_SA_ALG_AUTH_GCM_128** **EAPE_SA_ALG_AUTH_GMAC** **EAPE_SA_ALG_CIPH_NULL** **EAPE_SA_ALG_CIPH_DES_CBC** **EAPE_SA_ALG_CIPH_3DES_CBC** **EAPE_SA_ALG_CIPH_AES128_CBC** **EAPE_SA_ALG_CIPH_AES192_CBC** **EAPE_SA_ALG_CIPH_AES256_CBC** **EAPE_SA_ALG_CIPH_AES128_GCM** **EAPE_SA_ALG_CIPH_AES192_GCM** **EAPE_SA_ALG_CIPH_AES256_GCM** **EAPE_SA_ALG_CIPH_AES128_CTR** **EAPE_SA_ALG_CIPH_AES192_CTR** **EAPE_SA_ALG_CIPH_AES256_CTR**
armask	The anti-replay mask, should start at all zero		ckey	The cipher key
csalt	The cipher SALT (used in GCM/CTR modes)		mackey	The authentication key

Table 13: EAPE SA Fields

10.6 eape_go()

The *eape_go()* function is used to enqueue a packet in one of the pipelines.

```
int eape_go(
    eape_device *eape,
    int handle, int direction, pdu_ddt *src, pdu_ddt *dst, eape_callback cb, void *cb_data);
```

This will enqueue the packet pointed to by the *src* DDT structure into the pipeline indicated by *direction* (1 for outbound, 0 for inbound) using the SA indicated by *handle*. The return of this function is positive for success and negative for error. The return value when positive is the ID of the job (used when calling *eape_done()*).

The callback function is pointed to by *cb* with its associated data *cb_data*. If no callback is desired the *cb* pointer may be set to **NULL**. In this latter case the user may use the *eape_done()* function to poll whether their packet is finished or not.

10.7 eape_done()

The *eape_done()* function can be used (instead of callbacks/interrupts) to check if a job is finished. The *eape_go()* function returns a job ID which is used by this function to look up the job.

```
int eape_done(eape_device *eape, int handle, int swid);
```

This checks the status of the job indicated by *swid* issued on the SA indicated by *handle*. It returns 1 if the job is finished, 0 if not, and -1 on error (or if the *swid* is not valid).

10.8 eape_close()

To relinquish an SA context the *eape_close()* function can be called.

```
int eape_close(eape_device *eape, int handle);
```

This will mark the SA context as no longer in use. Note that it won't actually return the context back to the pool of free contexts until all jobs have posted against it. That is, you can call this function even if jobs are still in flight with the SA.

10.9 eape_pop_packet()

The *eape_pop_packet()* function is used to pull jobs off both inbound and outbound pipelines. It is typically called by the interrupt handler. It calls callback functions per job.

```
int eape_pop_packet(eape_device *eape);
```

11 CLP-800 (TRNG3)

This section describes the API for the CLP-800 Smart TRNG (also known as the TRNG3). This is a new hardware design with an entirely new programming model as compared with the older CLP-27 TRNG.

11.1 Library initialization

The TRNG3 SDK provides two main header files

```
#include “elptrng3_hw.h” /* Constants for the hardware register offsets & fields */
#include “elptrng3.h”    /* Declarations for this SDK functionality */
```

All TRNG3 functions take a pointer to the SDK state structure as their first argument. Each state structure tracks a particular hardware instantiation. Before any other SDK functions can be used, the structure must first be initialized by calling

```
void elptrng3_setup(struct elptrng3_state *trng3, uint32_t *regbase);
```

where

- `trng3` points to the allocated state structure
- `regbase` is the base address of the TRNG3's register map. This can be `NULL` when using custom register I/O as described below.

This function will put the device into a sane initial state and fill out the state structure (described below).

11.2 Error handling

Some of the TRNG3 SDK functions are documented to indicate successful completion by their return value. All such functions have failure conditions indicated (as usual) by a negative return. If the function description documents specific return codes, they are returned in the indicated situation. Nevertheless, a function may still fail for reasons not documented in its description by returning other error codes.

11.3 Hardware parameters

The `elptrng3_setup` function will query the hardware to determine what features are available in the design. The results are written to the state structure, and the application may inspect these values.

```
struct elptrng3_state {
     /* H/W features */
   unsigned short epn, stepping, output_len, diag_level;
   unsigned secure_reset:1, rings_avail:1;

   /* ... */
};
```

The meaning of the struct members is

- `epn`: Elliptic Project Number, an identifier associated with the particular design.
- `stepping`: Revision of this design configuration.
- `output_len`: Maximum supported output length of the core per random request, in bytes. In current designs, the core can be configured to support either 16 or 32-byte output.

- `diag_level`: Describes the amount of instrumentation in the configuration. Configurations with levels greater than zero allow some access to the internal state of the engine when it is configured in secure mode, via the indirect access port.
- `secure_reset`: 1 if the design resets in secure mode, 0 otherwise.
- `rings_avail`: 1 if the design supports true random reseed mode (TRNG configuration), 0 in PRNG-only configurations.

11.4 Register access

Two functions are provided to access TRNG3 registers directly:

```
uint32_t elptrng3_readreg(struct elptrng3_state *trng3, unsigned offset);
void elptrng3_writereg(struct elptrng3_state *trng3, unsigned offset, uint32_t val);
```

where

- `trng3` points to an initialized state structure
- `offset` is the register index relative to the base address (in units of 32-bit words)
- (for `elptrng3_writereg`) val is the value to write.

For details on the available registers, see the CLP-800 User Guide (EDN-0462).

11.4.1 Custom register wrappers

The above functions should be used for all register I/O as they allow the use of custom wrappers for access. This feature is useful when the TRNG3 is instantiated in another design that does not expose its register map as a flat region of memory. For most configurations, it is not necessary to use the wrappers described in this section.

When `elptrng3_setup` is called with a `NULL` value for regbase, the custom wrappers in the state structure will be used. There are 3 struct members which must be initialized manually prior to calling `elptrng3_setup`:

```
struct elptrng3_state {
   /* Register access */
   void     (*writereg)(void *base, unsigned offset, uint32_t val);
   uint32_t (*readreg) (void *base, unsigned offset);
   void *regbase;

   /* ... */
}
```

All register accesses in the SDK are done indirectly via the writereg and readreg members in the state structure. The function assigned to the `readreg` member shall read the specified register and return the resulting value, the function assigned to the `writereg` member shall write the specified register with the given value. The `regbase` member in the state structure may be assigned any value, and is passed verbatim as the `base` parameter to both functions. Note that as the `elptrng3_setup` function will call these register access functions, they may not access any members of the TRNG3 state structure or call any TRNG3 SDK functions.

The readreg and writereg parameters are

- `base` is the regbase member of the state structure
- `offset` is the register index (in units of 32-bit words)
- (for `writereg`) `val` is the value to write.

Example usage

The following shows how the functions might be written for accessing the TRNG3 via a hypothetical indirect interface similar to the TRNG3's own indirect access port:

```
void my_writereg(void *base_, unsigned offset, uint32_t val)
{
      uint32_t *base32 = base_;
      pdu_io_write32(&base32[MY_INDIRECT_ADDRESS], offset);
      pdu_io_write32(&base32[MY_INDIRECT_VALUE],   val);
      pdu_io_write32(&base32[MY_INDIRECT_CONTROL], MY_INDIRECT_WRITE);
}

uint32_t my_readreg(void *base_, unsigned offset);
{
      uint32_t *base32 = base_;
      pdu_io_write32(&base32[MY_INDIRECT_ADDRESS], offset);
      pdu_io_write32(&base32[MY_INDIRECT_CONTROL], MY_INDIRECT_READ);
      return pdu_io_read32(&base32[MY_INDIRECT_VALUE]);
}

struct elptrng3_state *setup_hardware(uint32_t *my_indirect_base)
{
      struct elptrng3_state *trng3;

      trng3 = malloc(sizeof *trng3);
      if (trng3) {
              trng3->readreg = my_readreg;
              trng3->writereg = my_writereg;
              trng3->regbase = my_indirect_base;
              elptrng3_setup(trng3, NULL);
      }

      return trng3;
}
```

11.5 Device configuration

After initializing the SDK with `elptrng3_setup`, the device can be configured using the functions in this section. Except for exceptional situations mentioned below, it is safe to perform engine reconfiguration concurrently with random number generation. However, concurrent threads must not attempt to simultaneously reconfigure the engine (by calling functions in this section) as the results are undefined.

11.5.1 Interrupts

If interrupts are being used, then the interrupt sources can be enaled or disabled by using the following functions:

```
void elptrng3_enable_irqs(struct elptrng3_state *trng3, uint32_t irq_mask);
void elptrng3_disable_irqs(struct elptrng3_state *trng3, uint32_t irq_mask);
```

where

- `trng3` points to an initialized state structure
- `irq_mask` is the bitwise-OR of the interrupt sources to enable or disable.

Macros are defined to make it easier to set irq_mask. They correspond to the bits in the IE and ISTAT registers defined in the CLP-800 User Guide (EDN-0462).

- `TRNG3_IRQ_GLBL_EN_MASK`: global interrupt enable
- `TRNG3_IRQ_RQST_ALARM_MASK`: request-based reseed reminder interrupts
- `TRNG3_IRQ_AGE_ALARM_MASK`: age-based reseed reminder interrupts

- `TRNG3_IRQ_SEED_DONE_MASK`: random reseed completion interrupts
- `TRNG3_IRQ_RAND_RDY_MASK`: random number generation interrupts.

Note that if the global interrupt enable is not set, the device will not generate any interrupts.

11.5.2 Reseeding

A reseed process may be initiated using the following function

```
int elptrng3_reseed(struct elptrng3_state *trng3, const void *nonce);
```

where

- `trng3` points to an initialized state structure
- `nonce` points to the 32-byte seed to use, or `NULL` to initiate a random reseed.

Reseeds using a nonce complete immediately. Random reseeds complete asynchronously. The TRNG3 remains fully functional during the reseed process and will continue to produce random numbers using the previous seed until the new one is ready. The availability of a new random seed is indicated by the `SEED_DONE` interrupt.

The function returns 0 if the reseed was successfully started. Defined errors are

- `CRYPTO_MODULE_DISABLED` if a random reseed was requested but the hardware does not include the ring functionality.

Note that the hardware also has an external input pin to initiate random reseeds.

11.5.3 Secure mode

The TRNG3 can be run in either *secure* or *promiscuous* modes. Running in promiscuous mode allows host access to some internal state of the engine, for diagnostic purposes. The mode can be changed by calling the function

```
void elptrng3_set_secure(struct elptrng3_state *state, int secure);
```

where

- `trng3` points to an initialized state structure
- `secure` specifies the mode; 0 sets promiscuous mode, any other value sets secure mode.

Care must be taken when changing the mode of the engine, as this will zeroize all internal state and cancel any outstanding operations. The device will revert to the unseeded state, so it is normally necessary to explicitly reseed the engine after changing the mode.

11.5.4 Output length configuration

If the TRNG3 was configured to support 256-bit output, then the output length may be changed at runtime between 128 and 256-bit modes. The maximum supported output size (in bytes) is stored in the `output_len` member of the state structure. Generally speaking, larger values perform better. Smaller sizes make individual random numbers generate faster. To change the output length of the engine, use the function

```
int elptrng3_set_output_len(struct elptrng3_state *trng3, unsigned outlen);
```

where

- `trng3` points to an initialized state structure
- `outlen` specifies the output length in bytes (either 16 or 32).

The function returns 0 on success. Defined errors are

- `CRYPTO_MODULE_DISABLED` if the request was valid but not supported by the hardware

Care must be taken when reconfiguring the output length, as it only affects random number generations that were started after setting the output length. It is therefore recommended that the output length be set once at initialization and left unchanged thereafter. If the output length must be changed at runtime, it may be necessary to discard the next generated value.

Example usage

The following shows how the output length may be changed at runtime safely. If the application has concurrent threads which may also access the device, it is necessary to block them from retrieving random numbers across the entire reconfiguration and subsequent polling loop (e.g. by using a mutex if numbers are retrieved in polling mode, or by masking the device's interrupt contribution for interrupt mode).

```
int my_set_output_len(struct trng3_state *trng3, unsigned len)
{
      int ret;

      ret = elptrng3_set_output_len(trng3, len);
      if (ret == 0) {
             while (elptrng3_get_random(&priv->trng3, NULL) == CRYPTO_INPROGRESS)
                    ;
      }

      return ret;
}
```

11.6 Mission mode

Once the TRNG3 is seeded, and the output length is (optionally) configured, it is ready to produce random output.

11.6.1 Generating random numbers

To start generating a new random value, write `TRNG3_CMD_GEN_RAND` to the control register, for example

```
elptrng3_writereg(state, TRNG3_CTRL, TRNG3_CMD_GEN_RAND);
```

This operation completes asynchronously. The engine indicates that the operation has completed by asserting the `RAND_RDY` interrupt. Since the engine collects data in the background, to achieve highest throughput it should be requested to generate a new random number prior to when that value is actually needed.

11.6.2 Retrieving random numbers

Output from the TRNG3 can be retrieved using the following function

```
int elptrng3_get_random(struct elptrng3_state *trng3, void *out);
```

where

- `trng3` points to an initialized state structure

- `out` points to a buffer large enough[5] to hold the output from the engine. If this parameter is `NULL` the output will be discarded (but all other behaviour, including status checks and clearing the `RAND_RDY` interrupt) will still be performed.

This function may be called from a `RAND_RDY` interrupt handler, but the engine is normally fast enough that polling mode is more efficient. Nevertheless, the TRNG3 uses the interrupt status indicator to signal availability of output, so this function must be called *without* acknowledging the `RAND_RDY` interrupt. If a random value is successfully retrieved, the `RAND_RDY` interrupt flag will be cleared automatically by this function.

A return value of 0 indicates that there is no data available and the engine is not currently generating a new random number. If data is available, it is written to the provided output buffer and the number of bytes written (either 16 or 32) is returned. Otherwise, the defined errors are

- `CRYPTO_NOT_INITIALIZED` if the engine is not currently seeded. The engine must be seeded before random numbers can be generated.
- `CRYPTO_INPROGRESS` if a random number is currently being generated.

When using polling mode with multiple concurrent threads, it is necessary to protect the hardware across the entire loop from simultaneous access by multiple threads (e.g. by using a mutex). Such protections are normally not necessary in interrupt service routines as appropriate mutual exclusion is typically provided automatically by the hardware or operating system.

Example usage

The following example shows a very simple polling loop to retrieve a random value. It uses the return value of `elptrng3_get_random` to send a generate command if the engine is idle.

```
int poll_random(struct elptrng3_state *trng3, void *out)
{
	int ret;

	do {
		ret = elptrng3_get_random(trng3, out);
		if (ret == 0) {
			/* Generate a new number */
			elptrng3_writereg(trng3, TRNG3_CTRL, TRNG3_CMD_GEN_RAND);
			ret = CRYPTO_INPROGRESS;
		}
	} while (ret == CRYPTO_INPROGRESS);

	return ret;
}
```

11.6.3 Reseed reminder alarms

The TRNG3 may be configured to raise an interrupt when a seed has been in use for a while, to remind software that it may be time to reseed the engine. The request-based reseed reminder may be set using the following function

```
int elptrng3_set_request_reminder(struct elptrng3_state *trng3, unsigned long val);
```

where

- `trng3` points to an initialized state structure
- `val` is the number of requests before the alarm is signalled, or 0 to disable the feature.

5 The buffer only needs to be as large as the configured output length, but it is safest to size the buffer to fit the largest possible output size (i.e. 32 bytes). For convenience, an object-like macro is defined to expand to this maximum size: `ELPTRNG3_MAXLEN`.

The function returns 0 on success. Note that the engine's internal counter has a resolution of 16 requests, so the specified request count will be rounded up to the nearest 16 requests.

The age-based reseed reminder may be set using the following function

```
int elptrng3_set_age_reminder(struct elptrng3_state *trng3, unsigned long long val);
```

where

- `trng3` points to an initialized state structure
- `val` is the number of clock cycles before the alarm is signalled, or 0 to disable the feature.

The function returns 0 on success. As with the request-based reminder, the engine's internal counter has a resolution of 2^{26} clock cycles, and the specified cycle count will be rounded up.

Changes to either reminder alarm do not take effect until the next time the engine is reseeded.

11.7 Diagnostics

When the engine is in promiscuous mode, it is possible to retrieve the seed material from the engine. This may be used to test the bit generation frontend. The seed material is retrieved using the function

```
int elptrng3_get_seed(struct elptrng3_state *trng3, void *out);
```

where

- `trng3` points to an initialized state structure
- `out` points to a 32-byte buffer to store the output.

Returns 0 on success. Defined errors are

- `CRYPTO_NOT_INITIALIZED` if the engine is unseeded.

If the engine is in secure mode, the result will always be all-bits zero.