Technical notes on using Analog Devices products and development tools
Visit our Web resources http://www.analog.com/ee-notes and http://www.analog.com/processors or e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

Using the Security Packet Engine to Protect Data

Contributed by Yi, Gabby. *Rev 1 – April 14, 2016*

Introduction

Both the ADSP-BF70x Blackfin+ processors and the ADSP-SC58x/ADSP-2158x SHARC+ processors contain a security packet engine (PKTE), which is a hardware accelerator that helps to offload some of the common cryptographic functions like symmetric ciphers, hash functions, and message authentication code (MAC) transforms from having to be performed in software on the host processor. The security packet engine also contains a pseudo-random number generator (PRNG). This EE-Note reviews how to configure and use the security packet engine to perform AES-128 encryption and decryption in various operational modes.

Security Packet Engine (PKTE) Overview

Supported Algorithms

The PKTE supports many cipher algorithms, as shown in Table 1.

Cipher Algorithm	Modes	Key length
DES	ECB, CBC	56-bit
Triple DES	ECB, CBC	3 x 56-bit
AES	ECB, CBC, ICM, CTR	128-, 192- and 256-bit
ARC4™	Stateful and Stateless Mode	Up to 128-bit

Table 1. Supported Ciphers in the Security Packet Engine

It can also perform the SHA-1, SHA-2 (224-bit and 256-bit) and MD5™ hash functions, as well as supporting the HMAC transforms for SHA-1, SHA-2 and MD5™.

Modes of Operation

In Direct Host Mode (DHM), jobs for the packet engine can be configured to be performed individually while the host processor waits for the results before configuring the packet engine for the next job. In Autonomous Ring Mode (ARM), a list of jobs can be configured in a queue and presented to the packet engine all at once.

Direct Host Mode (DHM)

In this mode, all the details of the job are configured by writing to the memory-mapped registers (MMRs) of the packet engine. Even the input data is written directly to the packet engine's input buffer, which is part of the packet engine's MMR space. The same is said for the output results, which need to be read out directly from the memory-mapped output buffer of the packet engine. In this mode, no Direct Memory Access (DMA) is used at all.

Autonomous Ring Mode (ARM)

In this mode, the segments of the packet engine's registers are mirrored in host memory structures. These structures, known as descriptors, are configured to define the job for the packet engine. Multiple descriptors can be defined as an array prior to starting the PKTE, and when this descriptor array is presented to the PKTE, it will run through each descriptor and perform the job as configured without any further assistance or interruption from the host processor.

Configuring the PKTE for AES Encryption/Decryption

Input and Output Data Arrangement

Figure 1 shows the expected arrangement of the input data. The packet engine consumes the data depending on how the job is configured.

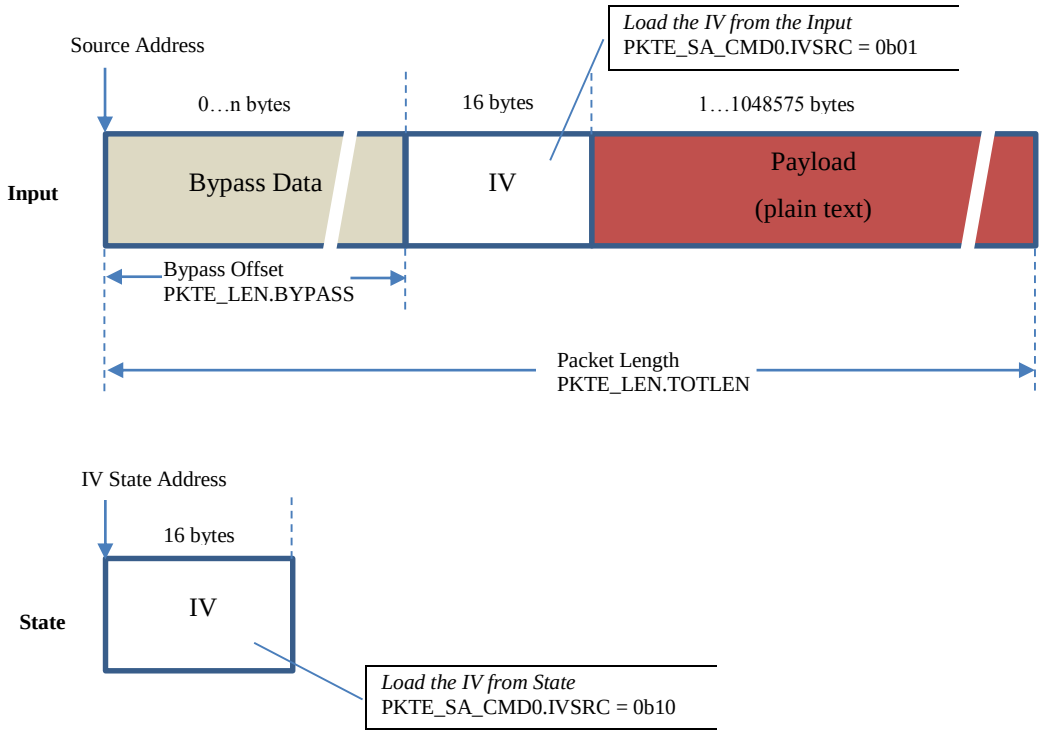

Figure 1. Input Data Formatting for the Security Packet Engine

As can be seen, the data to be encrypted is actually the third piece of data in the input, depending on how the job is configured. The first piece of input is the bypass data, which the PKTE directly copies from the input source to the output destination. An optional second piece of input is the Initialization Vector (IV). If the descriptor is configured to *load the IV from the input*, this IV will be the second piece of input. Otherwise,

if the descriptor is configured to *load the IV from state*, the IV will be read from the state structure in host memory (ARM) or from the state MMRs (DHM) instead.

Figure 2 shows how the PKTE consumes the input data and how the results are output. First, whether the IV comes from the state structure or as part of the input data (or even if it's generated from the PRNG), it is processed along with the input plaintext data. The PKTE will also generate the needed padding, depending on the descriptor configuration and the selection of the algorithm.

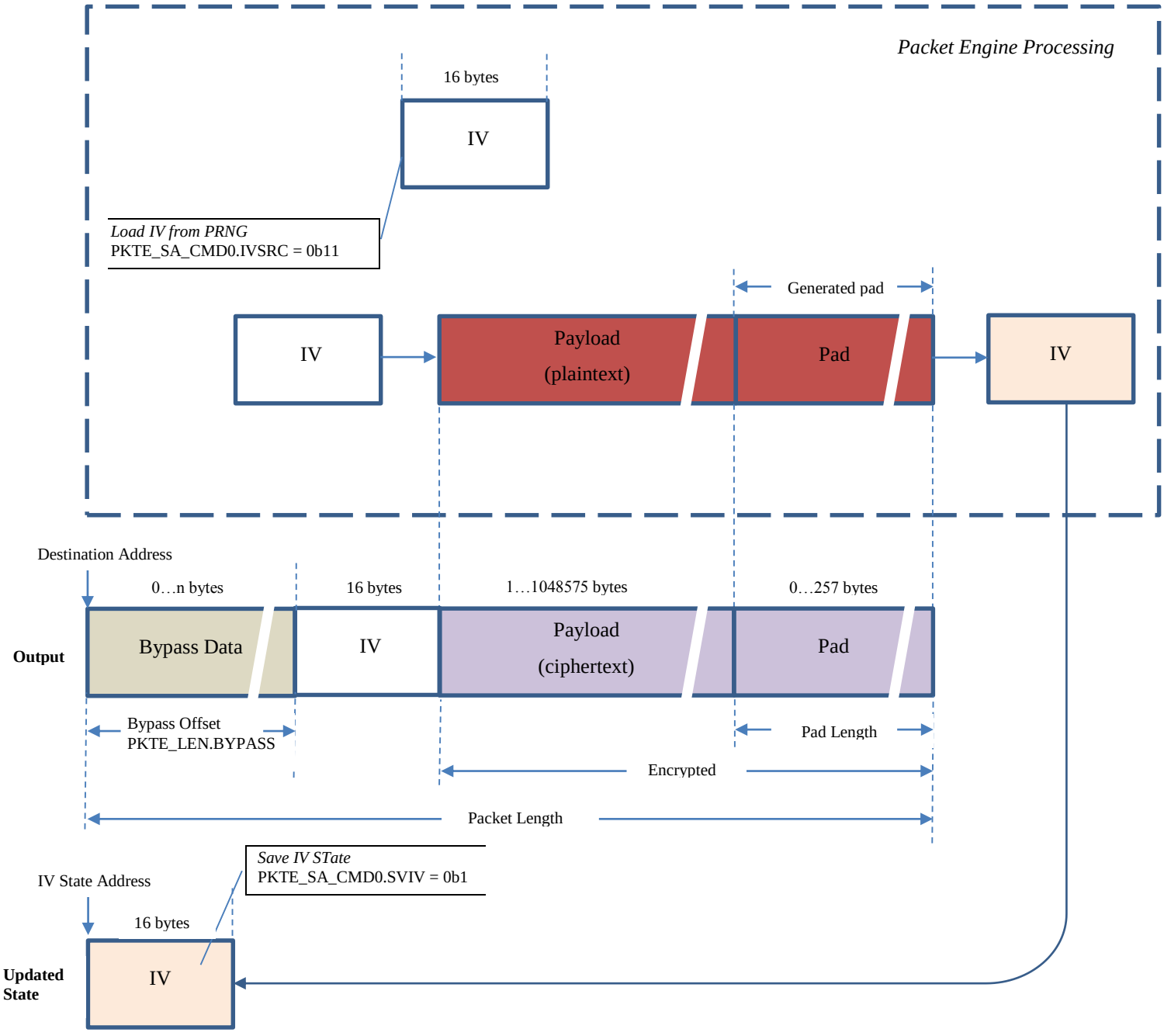

Figure 2. Data Consumption and Output Data Formatting by Security Packet Engine

For the output, the PKTE will directly copy any bypass data from the input source to the result destination. If the descriptor is configured to *Copy Header* (`PKTE_SA_CMD1.CPYHDR = 0b1`), the IV will also be copied to the destination after the bypass data from whatever source the IV is configured to come from. Following this is the encrypted plaintext, otherwise known as ciphertext, which is the encryption of the original input data and any generated padding that follows.

Finally, if the *Save IV State* is chosen (`PKTE_SA_CMD0.SVIV = 0b1`), the state structure buffer will be updated with the IV. Either *Copy Header* or *Save IV State* should be selected if the PRNG is used to generate an IV. Since the IV is needed to perform decryption, the IV should be saved using any of these methods.

AES Example Code

Included with this EE-Note is *example code*[1] to set up the PKTE to perform AES encryption and decryption with different configurations. The first example is a call to the function `aes_dhm_example1()`, passing the Boolean value of `false` as input. This example sets up the PKTE to encrypt input `0x00112233445566778899aabbccddeeff` using key `0x000102030405060708090a0b0c0d0e0f` in ECB mode. This input/output example is presented in the *AES Specification*[2]. If calculated correctly, the output is `0x69c4e0d86a7b0430d8cdb78070b4c55a`. After calculating the ciphertext, the example then sets up the PKTE to take in the ciphertext as input and decrypt it. If successful, the output is the original plaintext input.

Code Review of aes_dhm_example1() Example

The first thing this function does is configure the engine for DHM, as shown in. Listing 1.

```
/* Configure Engine */

    *pREG_PKTE0_CFG = 3;      // reset engine and release
    ssync();                  // ensure write propagates through before releasing
                              // engine out of reset
    *pREG_PKTE0_CFG = 0;      // release out of reset and put into DHM
```

Listing 1. aes_dhm_example1 Engine Configuration

`PKTE0_CFG` is set to 0x3 to reset the engine, and then a `ssync()` is performed to ensure that the write to reset the engine propagates through before releasing the engine from reset. The `ssync()` intrinsic decodes to a native system synchronization instruction (`SSYNC;`) on the Blackfin+ processor, and this function is redefined for SHARC+ and ARM compilations to those cores' respective synchronization instructions. The engine is then taken out of reset and set for DHM by setting `PKTE0_CFG` back to 0.

Next, the command descriptor registers are directly configured, as shown in Listing 2.

```
/* Write Command Descriptor Regs */

    *pREG_PKTE0_CTL_STAT = 0x1;                    // host ready
    *pREG_PKTE0_LEN      = SZ_INPUT1 | 0x400000;   // set length.  Set sync bits
                                                   // PE_READY=0, HOST_READY=1
    *pREG_PKTE0_CDSC_CNT = 1;                      // cmd desc cnt =1.  Trigger
                                                   // PKTE to validate cmd desc
```

Listing 2. aes_dhm_example1 Command Descriptor Configuration

For DHM, only a subset of the command descriptor registers is used. Specifically, the control/status register (PKTE0_CTL_STAT), the packet length register (PKTE0_LEN), and the command descriptor count register (PKTE0_CDSC_CNT) are used.

PKTE0_CTL_STAT is first set to 0b01 to indicate that the host is ready for the PKTE to process another job. This also resets the status from the PKTE, indicating that it has finished with any previous job.

Next, the packet length is programmed along with synchronization bits mirrored from PKTE0_CTL_STAT into the PKTE_LEN register. If there were any bypass data, its length would also be programmed into this register, but this example doesn't have any bypass data.

Finally, PKTE0_CDSC_CNT must be written with a '1' to indicate that the command descriptor for the packet engine is valid.

The next step is to program the Security Association (SA) Record registers. For this example, the command registers and key registers are all that's needed to be programmed, as shown in Listing 3.

```
/*
 * Configure SA regs to define operation and State regs
 * to provide any extra input
 */
    *pREG_PKTE0_SA_CMD0 =
         (((null_hash) << BITP_PKTE_SA_CMD0_HASH)    & BITM_PKTE_SA_CMD0_HASH)    |
         (((aes)       << BITP_PKTE_SA_CMD0_CIPHER)  & BITM_PKTE_SA_CMD0_CIPHER)  |
         (((zero)      << BITP_PKTE_SA_CMD0_PADTYPE) & BITM_PKTE_SA_CMD0_PADTYPE) |
         (((basic)     << BITP_PKTE_SA_CMD0_OPGRP)   & BITM_PKTE_SA_CMD0_OPGRP)   |
         (((outbound)  << BITP_PKTE_SA_CMD0_DIR)     & BITM_PKTE_SA_CMD0_DIR )    |
         (((encrypt)   << BITP_PKTE_SA_CMD0_OPCD)    & BITM_PKTE_SA_CMD0_OPCD );

    *pREG_PKTE0_SA_CMD1  =
         (((bits_128)<< BITP_PKTE_SA_CMD1_AESKEYLEN)&BITM_PKTE_SA_CMD1_AESKEYLEN) |
         (((ecb)     << BITP_PKTE_SA_CMD1_CIPHERMD)  & BITM_PKTE_SA_CMD1_CIPHERMD) ;

    /*
     * Program in the key for encryption
     */
    *pREG_PKTE0_SA_KEY0 = ((uint32_t *)key1)[0];
    *pREG_PKTE0_SA_KEY1 = ((uint32_t *)key1)[1];
    *pREG_PKTE0_SA_KEY2 = ((uint32_t *)key1)[2];
    *pREG_PKTE0_SA_KEY3 = ((uint32_t *)key1)[3];

    /*
     * trigger operation to begin and have it wait for input
     */
    *pREG_PKTE0_SA_RDY   = 1;
```

Listing 3-aes_dhm_example1 security association record configuration

The Command 0 (PKTE0_SA_CMD0) register is programmed to identify the type of function the PKTE is being configured for. In this case, the example code is doing a basic encryption (outbound) using the AES algorithm without the need for any padding. The Command 1 (PKTE0_SA_CMD1) register is programmed

with ancillary details about the function. In this case, the key length for AES is 128 bits, and ECB (Electronic Cookbook) mode is being used.

After this, the 128-bit key is written to the first four 32-bit key registers (PKTE0_SA_KEY0-3), which are also part of the SA Record, and a '1' is written to PKTE0_SA_RDY to trigger the operation.

At this point, the PKTE knows what operation to do, but it has no input to perform the operations on. So, the input data is copied from the source buffer to the PKTE's input buffer whose starting address is shared with the PKTE0_DATAIO_BUF MMR. Once the data is filled in, the input buffer counter register (PKTE0_INBUF_CNT) is updated with the number of bytes copied to the input buffer, rounded up to the next multiple of four.

Now, the PKTE will begin processing the data. The host processor can either wait for an interrupt or poll for status bits to see whether an error has occurred or to see if the operation has completed. In this example, the Operation Done (PKTE_STAT.OPDN) bit is polled.

Once the operation is done, the number of output words available is obtained from the Output Buffer Full Count field in the PKTE0_STAT register. Since this is a cipher example and the input size is within the size limitations of the packet engine's IO buffer, the output count should be the same as the input size. When reading the output, the same data buffer that the input was written to is read from. In other words, the PKTE input buffer and output buffer share the same memory. When written to, the data will go to its input buffer. When read from, the data will come from the output buffer.

Once the data is read out, the number of bytes read from the output buffer is written to PKTE0_OUTBUF_CNT by the host processor as an acknowledgement. A '1' is also written to the result descriptor count register (PKTE_RDSC_CNT) so that the PKTE can accept a new command descriptor.

For sanity, the result is compared against the known answer provided by the specification.

The second part of this example basically follows the same steps, but it configures the engine to perform a decryption. Since the engine was not reset, the key used for encryption still remains in the key registers and does not need to be reprogrammed unless the decryption key is to be used. In the AES algorithm, an input block of 128 bits is processed in multiple "rounds". The key used for encryption is actually a seed to spawn new keys for each round, which is known as the key schedule. When decrypting, the engine needs to recalculate this schedule to find the last key of the last round. If this is already known, the key registers can be programmed with this key instead, thus saving some cycles needed to compute the key schedule. When using this option, the AES Decryption Key bit (PKTE0_SA_CMD1.AESDECKEY) needs to be set. The example code runs through the aes_dhm_example1() function twice, once using only the cipher key for both encryption and decryption, and a second time using the cipher key for encryption and the decryption key (obtained from the example in the specification) for decryption.

Code Review of aes_dhm_example2() Example

In aes_dhm_example2(), AES encryption is performed in DHM as in the first example, but with two notable differences.

First, CBC mode is used instead of ECB mode, as programmed in the PKTE0_SA_CMD1 register. For CBC mode, an IV is needed. The IV is the same size as an AES block size of 128 bits. Since PKTE0_SA_CMD0 is programmed to load the IV from state and the engine is operating in DHM, the 128-bit IV is programmed into the first four 32-bit state IV registers (PKTE0_STATE_IV0-3).

The second difference is the size of the input, which is 320 bytes in this example. The data set used was taken from a FIPS Cryptographic Algorithm Validation Program test set and doubled. Using a host utility, OpenSSL, the expected output was calculated. Since the PKTE's data IO buffer is only 256 bytes, this example performs the encryption in two pieces. It loads the entire data IO buffer and programs the input buffer count (`PKTE0_INBUF_CNT`) to 256. The PKTE will perform the encryption on this portion of the input, and then the host processor will read out the results in the same manner as in the first example. The next time through the loop, the host will write in the rest of the input and then program `PKTE0_INBUF_CNT` with the 64 bytes that remains of the original 320-byte input so that it can also be processed.

Code Review of aes_arm_example1() Example

The `aes_arm_example1()` example demonstrates the same AES encryptions performed in the `aes_dhm_example1()` and `aes_dhm_example2()` examples, but it uses ARM instead of DHM. Additionally, the second encryption passes the IV in along with the input data instead of doing so in the SA state structure. Finally, the example also demonstrates bypass functionality.

Because the first two examples used DHM, the registers had to be written directly. When the PKTE uses ARM, the registers for the command descriptor, the SA record, and the SA state are mirrored in structures in memory, the `#typedefs` for which can be found in the `AesExamples.h` header file. The command and result descriptors are defined as `cd_t` and `rd_t`, respectively, while the SA record is defined as `sa_t` and the SA state is defined as `state_t`.

Using the command descriptor structure type, `cd_t`, and the result descriptor structure type, `rd_t`, an array was created. Since only two separate encryptions are being performed, the size of the array is two. These two arrays form the command and result descriptor ring, respectively. A developer doesn't need to know the number of jobs beforehand to define the size of the ring. The size should be big enough such that while the PKTE is processing current jobs, the host processor can define and add more jobs to the ring, as needed; however, this example only shows a simplified use case.

Figure 3 illustrates how all the structures are organized in the host processor's memory.

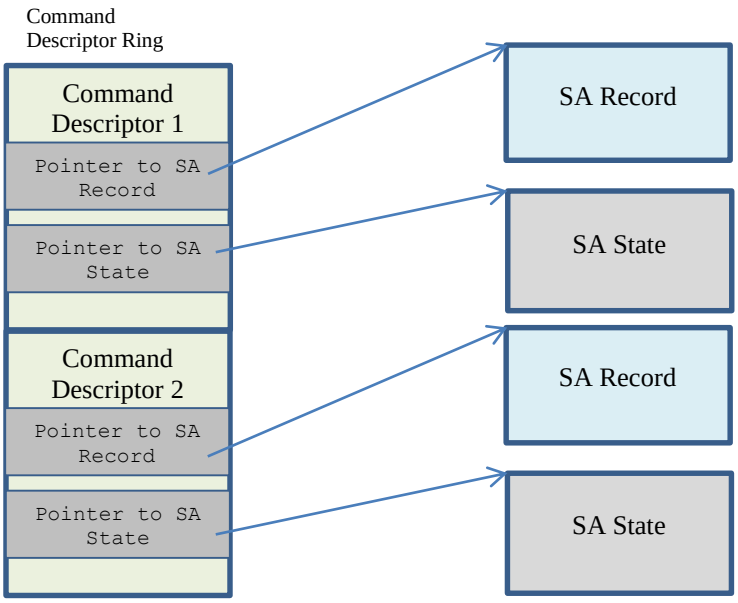

Figure 3. Representation of Command Descriptor Ring

As shown, the command descriptors are arranged as an array. Elements in the command descriptors include pointers to hold the address of where the SA record and SA state structures are in memory.

Separate SA record structures and SA state structures are also declared for each job. Depending on the use case, SA record structures and SA state structures can be reused and correlated to multiple command descriptors, though this example does not demonstrate this.

Listing 4 shows the configuration code for the `aes_arm_example1()` example.

```
    *pREG_PKTE0_CFG = 0x300;     // release out of reset and put into AR mode

/*
 * Configure rings
 */
    *pREG_PKTE0_RING_CFG = RING_SIZE-1;                // tell engine ring size
    *pREG_PKTE0_CDRBASE_ADDR = (uint32_t)(c_ring);     // tell engine where base of
                                                       // command desc ring is
    *pREG_PKTE0_RDRBASE_ADDR = (uint32_t)(r_ring);     // tell engine where base of
                                                       // result desc ring is
```

Listing 4-aes_arm_example1 Mode and Ring Configuration

As in the previous examples, the packet engine can be reset and then set again for the new mode of operation. This time, `PKTE0_CFG` is programmed with 0x300 to take the engine out of reset and configure it for ARM.

After this, the engine needs to know the size and location of the command and result descriptor rings. Note that `PKTE0_RING_CFG` is programmed with the size of the ring minus one, then the `PKTE0_CDRBASE_ADDR` and `PKTE0_RDRBASE_ADDR` registers are set to the base addresses of the command and result descriptor rings, respectively.

Next, the command descriptor and the SA record are configured for the first job, as shown in Listing 5.

```
/*
* Configure descriptor for 1st example
*/

/*
 * set up command descriptor
 */
    cmdDesc = &c_ring[0];

    cmdDesc->CTRL_STAT     = BITM_PKTE_CTL_STAT_HOSTRDY; // transfer ownership from
                                                         // host to PE of the desc
    cmdDesc->SRC_ADDR      = plaintext1;
    cmdDesc->DEST_ADDR     = output1;
    cmdDesc->SA_ADDR       = (unsigned long)(&saRec[0]);
    cmdDesc->SA_STATE_ADDR = (unsigned long)(&saState[0]);
    cmdDesc->USERID        = EXAMPLE1_ID;
    cmdDesc->LENGTH        = SZ_INPUT1 | BITM_PKTE_LEN_HSTRDY;
```

Listing 5-aes_arm_example1 Command Descriptor Configuration for 1st Encryption Job

There isn't much difference than before, except now the values are programmed into structures in host memory. All the elements in the descriptor need to be programmed, including indicating where the input data is stored, where the output is going to, the address of the SA record, and the address of the SA state structure. The command descriptor also includes a field for a User ID, which is applicable for advanced use cases such as network security protocols. The PKTE copies the `USERID` field to the result descriptor `USERID` field. This helps to identify what the result was associated with when there are multiple threads and jobs occurring in the system.

The set up for the second encryption is similar except that it uses bypass data and the IV is included in the input buffer. As shown in Listing 6, the length field in the command descriptor for the second job is the summation of the input size, the IV size, and the bypass data size. The mirrored synchronization bits are ORed in as well.

```
cmdDesc->LENGTH = (SZ_INPUT2 + AES_IV_SZ + BYPASS_SZ_WDS * sizeof(uint32_t)) |
                  ((BYPASS_SZ_WDS<<BITP_PKTE_LEN_BYPASS) & BITM_PKTE_LEN_BYPASS) |
                  BITM_PKTE_LEN_HSTRDY;
```

Listing 6-aes_arm_example1 Packet Length Configuration for 2nd Encryption Job

Once everything is configured, the PKTE can start, which is accomplished by incrementing the command descriptor count. In this case, two descriptors were set up, so `PKTE0_CDSC_INCR` is programmed to 2. Just as in the DHM examples, once the engine is started, the host processor can either poll or set up interrupts to trigger on either completion or errors. Here, the example polls on the count of the result descriptor ring (`PKTE0_RDSC_CNT`). Since the command descriptor ring count started with zero and was incremented by two, the result descriptor ring count will be two once both of the command descriptors have been processed.

The host processor can then analyze the result descriptors to see if any errors occurred and find where the results are stored.

Benchmarks

Table 2 shows the PKTE performance for the various algorithms it supports, and these measurements are identical between the ADSP-BF70x Blackfin+ processors and the ADSP-SC58x/ADSP-2158x SHARC+ processors.

Security Algorithm	Mode	Performance [bits/cycle]
Cipher Cores		
AES 128-bit key	ECB, CTR/ICM	2.46
AES 192-bit key	ECB, CTR/ICM	2.06
AES 256-bit key	ECB, CTR/ICM	1.78
AES 128-bit key	CBC with 128-bit Encrypt	2.42
AES 128-bit key	CBC with 128-bit Decrypt	2.46
AES 192-bit key	CBC with 192-bit Encrypt	2.03
AES 192-bit key	CBC with 192-bit Decrypt	2.06
AES 256-bit key	CBC with 256-bit Encrypt	1.75
AES 256-bit key	CBC with 256-bit Decrypt	1.78
DES	ECB, CBC	8
Triple-DES	ECB, CBC	2.67
ARC4[π]	-	3.2
Hash Cores		
SHA-1	-	6.23
SHA-2 (224bit and 256bit)	-	7.88
MD5[π]	-	7.88

[π]Only available on ADSP-SC58x/ADSP-2158x SHARC+ processors.

Table 2- Benchmarks for Security Packet Engine

The PKTE resides on different clock domains between the two architectures, so actual benchmark times may vary. On the ADSP-BF70x Blackfin+ processors, the PKTE runs on SCLK1, whereas SCLK0 is the correct clock domain for the ADSP-SC58x/ADSP-2158x SHARC+ processors. Please see the *Blackfin+ datasheet*[3] and the *SHARC+ datasheet*[4] for details regarding specific models and associated timing specifications.

Conclusion

This EE-Note explored some of the functionality and flexibility of the PKTE by reviewing code that sets up the packet engine to perform AES encryption and decryption in different operating modes and configurations.

References

[1] *Associated Code for Using the Security Packet Engine to Protect Data (EE-386).* Rev 1, April 2016. Analog Devices, Inc.

[2] *Federal Information Processing Standards Publication 197. Announcing the Advanced Encryption Standard (http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf).* Nov. 26, 2001.

[3] *ADSP-BF700/701/702/703/704/705/706/707 Blackfin+ Core Embedded Processor Datasheet*. Rev A. September 2015. Analog Devices, Inc.

[4] *ADSP-SC582/583/584/587/589/ADSP-21583/584/587 SHARC+ Dual-Core DSP with ARM Cortex-A5 Datasheet*. Rev PrF. February 2016. Analog Devices, Inc.

Readings

[1] *ADSP-BF70x Blackfin+ Processor Programming Reference.* Rev 0.2. May 2014. Analog Devices, Inc.

[2] *ADSP-BF70x Blackfin+ Processor Hardware Reference.* Rev 0.2, May 2014. Analog Devices, Inc.

[3] *ADSP-SC58x SHARC Processor Hardware Reference.* Rev 0.2, June 2015. Analog Devices, Inc.

Document History

Revision	Description
Rev 1 – April 14, 2016 *by G. Yi*	Initial Release