AURIX 2G HSM Introduction

Allen
IFCN ATV SMD GC SAE MC
2018/5/2

History List/ Changes

- Module Name: HSM
- Module Owner: Danilo Ciancitto
- Slide Target: HSM A1G/A2G Delta
- Slide Version: 2.1
- Slide Status: Released
- Last Modification: 01/20/2017
- Revision History:
  - 1.0 First version for TC2xx
  - 2.0 Second version for TC3xx
  - 2.1 Watchdog slides added

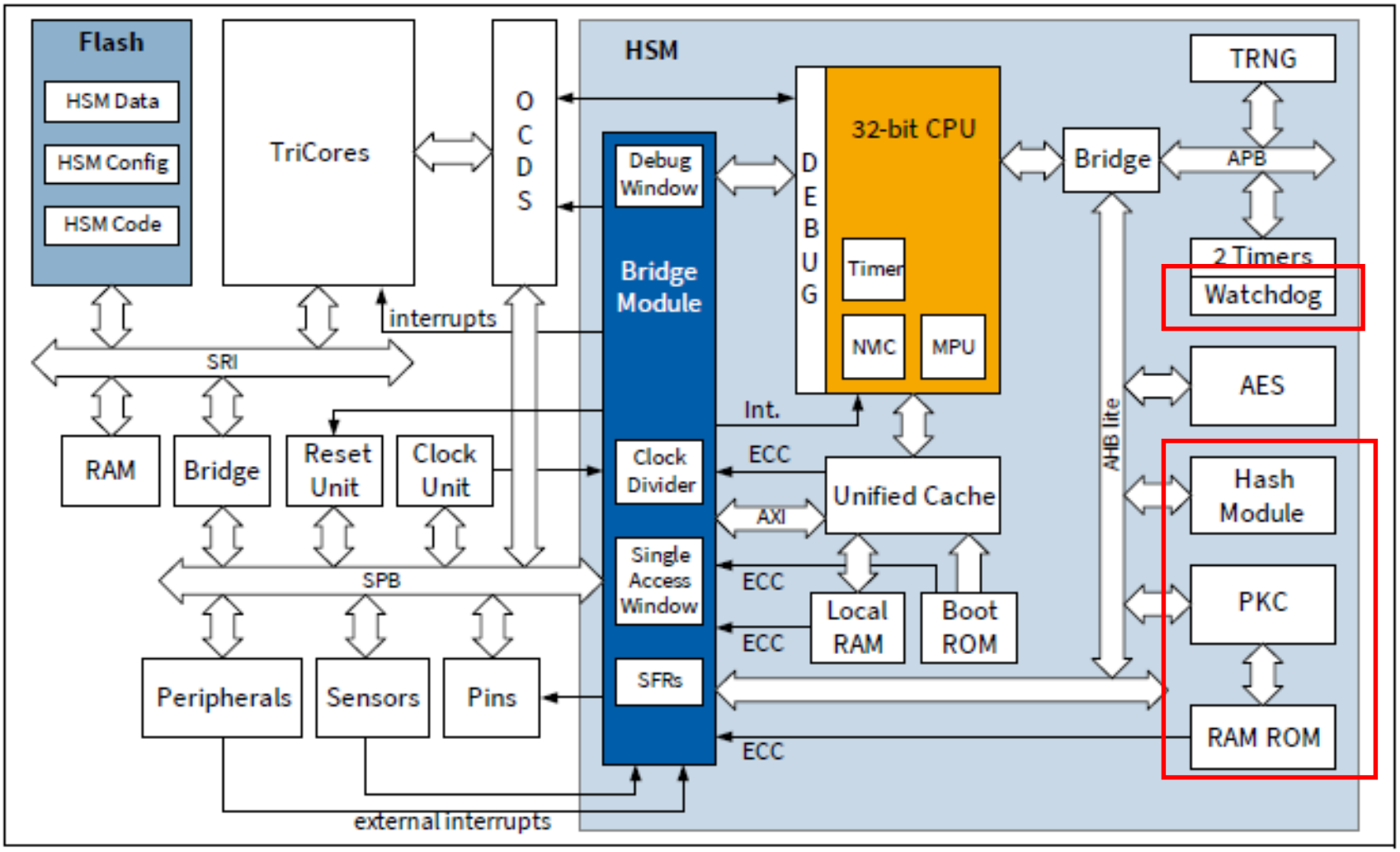
Agenda
HSM Introduction

- HSM Introduction and Architecture Overview
- True Random Number Generator
- AES 128 Encryption / Decryption Module
- Public Key Cryptography (PKC) Module
- SHA256- HASH Module
- Performance Figures
- Watchdog Timer
- Bridge Module

Agenda
HSM Introduction

1. HSM Introduction and Architecture Overview
2. True Random Number Generator
3. AES 128 Encryption / Decryption Module
4. Public Key Cryptography (PKC) Module
5. SHA256- HASH Module
6. Performance Figures
7. Watchdog Timer
8. Bridge Module

HSM Introduction
Hardware Security Module (HSM)

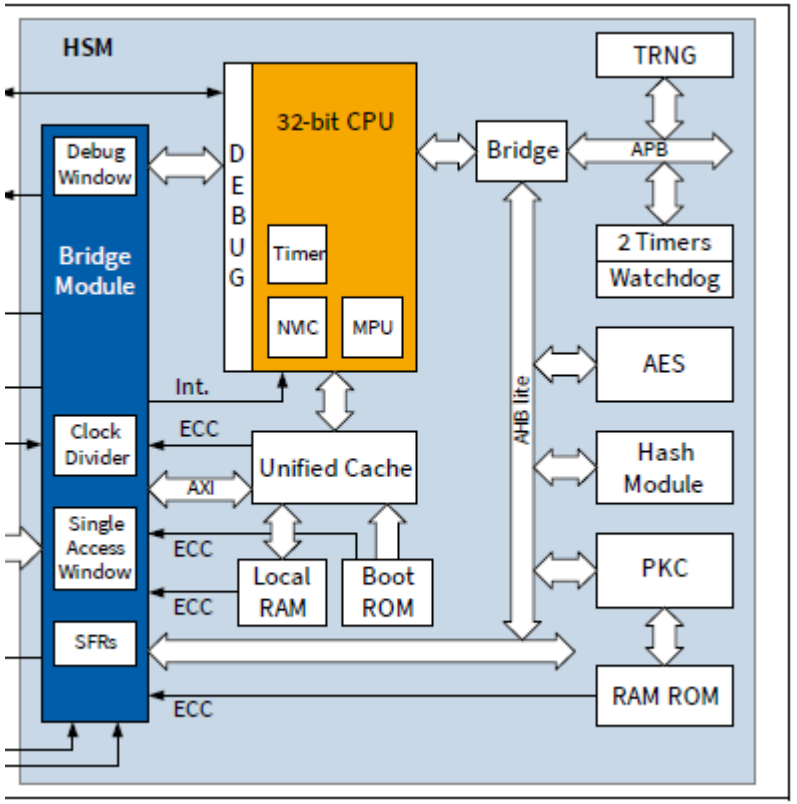


› Red boxed modules are new ones and available only for A2G

HSM Introduction
Hardware Security Module (HSM)

HSM offers protection against logical attacks. For this purpose it relies on the following features implemented in the Host system:

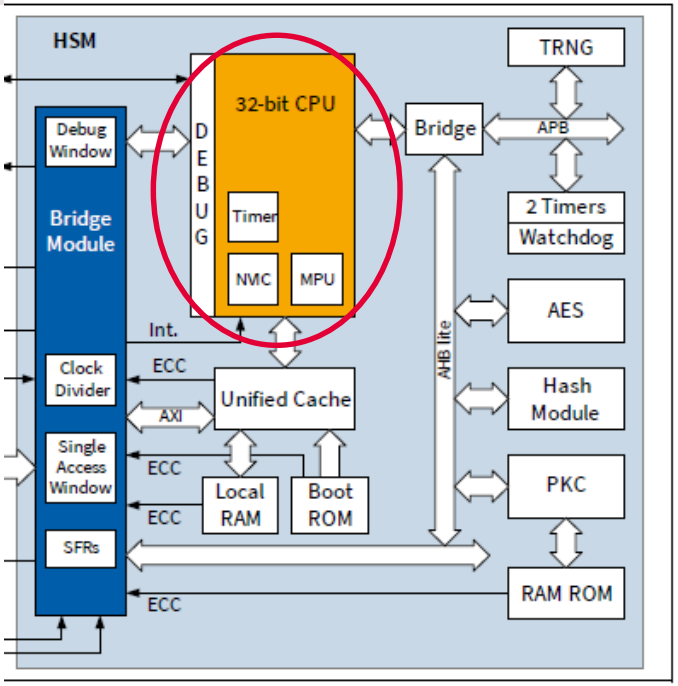
- Adequate protection and locking of flash memory areas that are reserved for the HSM
- One time programmable (OTP) Pages in flash config area
- A memory region in the config area that is reserved for the HSM.
  - Locking of this region against modifications (OTP)
  - HSM region in the config area can be switched off after the boot process
- Configuration of MBISTs in HSM module after reset, start of HSM after MBIST configuration.



The HSM contains no hardware countermeasures against physical attacks. Countermeasures against side-channel attacks and failure attacks have to be implemented by software".

HSM Architecture Overview

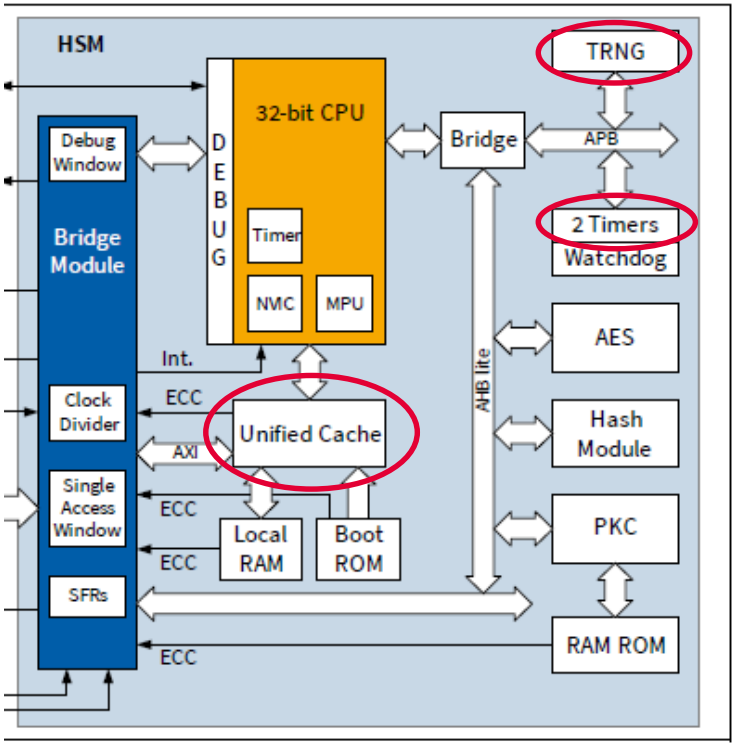
- CPU based on **ARM™ Cortex™ M3**
  - 24-bit **SysTick** Timer
    - Ref. Freq. fSPB/16 (i.e. 6.25Mhz)
  - Nested Vector Interrupt Controller:
    - 8 Interrupt nodes and 8 Priority level
  - Memory Protection Unit (MPU)
    - ARMv7-M compatible Protected Memory System Architecture
  - Debug support
    - HSM controls debug access to the system and separately to the HSM itself



Exception Number	Exception Type	Priority	Description
16	Timer 0	Programmable	Timer 0 overflow interrupt
17	Timer 1	Programmable	Timer 1 overflow interrupt
18	TRNG	Programmable	True Random Number Generator
19	Bridge Service	Programmable	HSM Bridge Service interrupt
20	Bridge Error	Programmable	HSM Bridge Error interrupt
21	Sensor Interrupt	Programmable	Sensor interrupt
22	External Interrupt	Programmable	External Peripheral interrupt

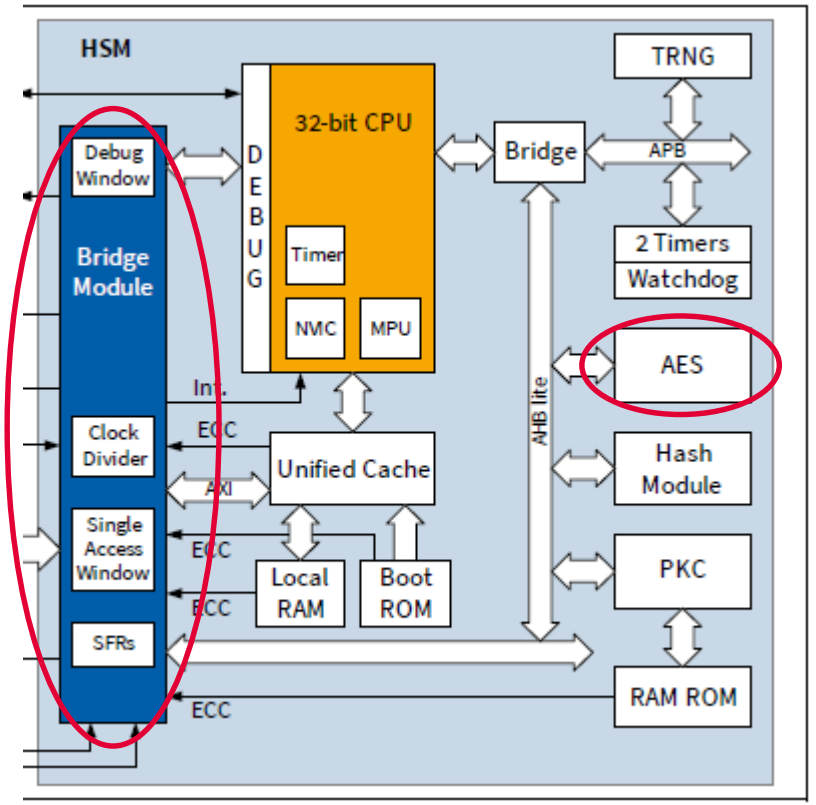
HSM Architecture Overview

- 4 Kbytes unified data and instruction **cache**
  - 4 ways – set associative
  - A cache hit -> Single Cycle Access
  - A cache miss ->up to 4 cycles delay
  - **Hit rate -> 90%**
- True Random Number Generator (TRNG)
  - provides random data for cryptographic algorithms (keys), protocols (challenges, blinding values, padding bytes, etc.).
- Timer module with 2 16-bit general purpose timers
  - Both timers are counting upwards and can be started and stopped individually
  - Each timer has own prescaler and reload value
  - Whenever the counter register overflows (transition from 0xFFFF to "reload value"), The timer x interrupt request is triggered and an interrupt will be generated if configured by the relevant node in the interrupt control module



HSM Architecture Overview

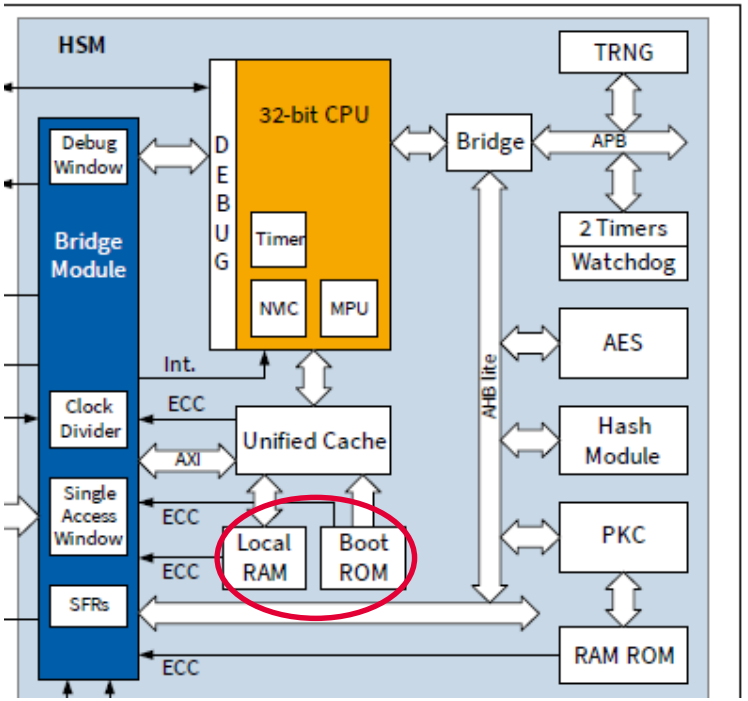
- AES module with local key and context storage
  - 8 128-bit keys (2 of them are non-alterable)
  - 5 contexts (including 1 for pseudo RNG)
- Bridge module connects HSM to the Host
  - "Firewall" functionality: HSM internals are protected from accesses by other masters.
  - Bus master on the SPB. Accesses possible to complete system memory map.
  - Communication SFRs for exchanging data.
  - SFRs for triggering an interrupt in the HSM CPU
  - 32 HSM external interrupt inputs
    - Mapped to one interrupt node of the NVIC
  - 2 interrupt signals from HSM to the system interrupt controller
  - Inputs for up to 10 sensors
  - Control of 2 pins
  - Option for triggering application and system reset of the chip

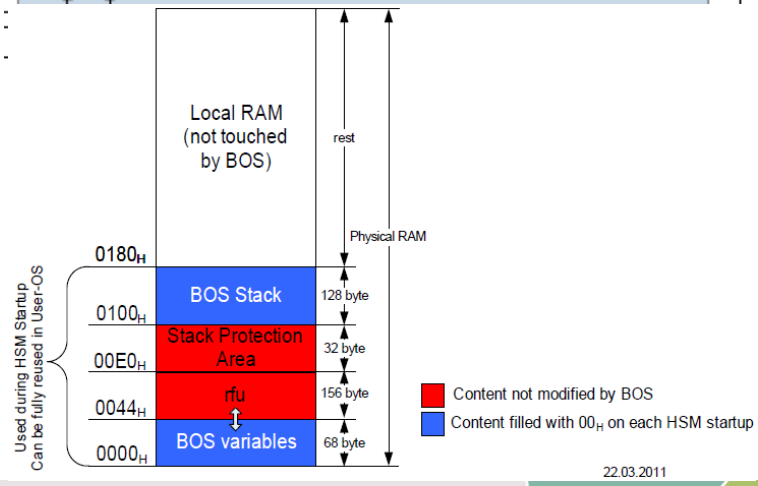


HSM Architecture Overview

- Boot ROM (4KB)
  - contains code and read-only data that is necessary for the start up of the HSM.
  - The ROM can **be switched off** when the boot software is finished via **DBGCTR (security protection** and **power consumption)**.
  - ROM data are protected with 1-bit error correction and 2-bit error detection
  - The ROM is tested by a **checksum**.

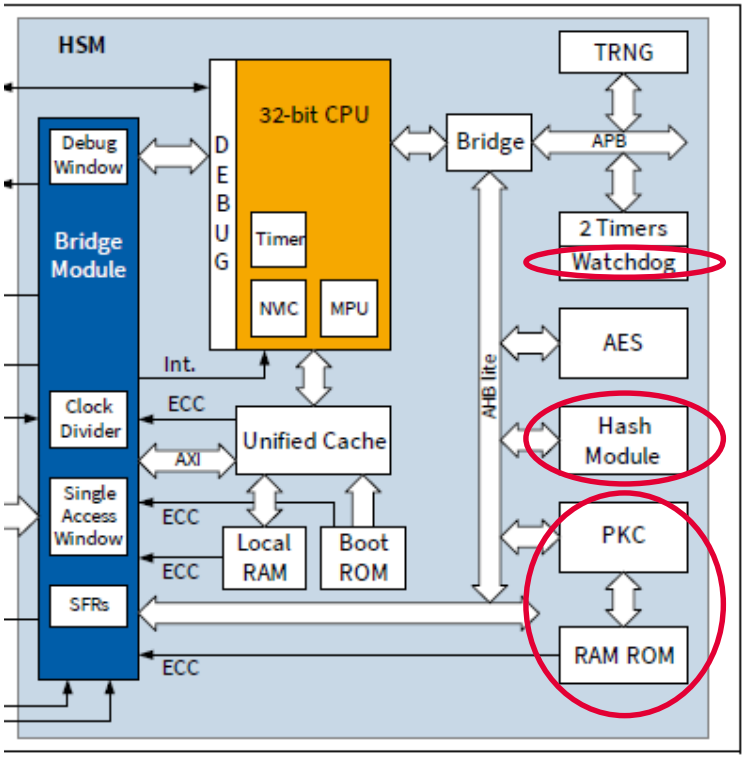
- Local RAM:
  - 24KB/40KB for A1G
  - 96KB for A2G
  - Contents is not preserved over a reset
  - The RAM is tested via **MBIST**
  - 1-bit error correction, 2-bit error detection
  - The first **384 bytes** of the physical local RAM are reserved for the BOS (fully reusable in user-OS)





HSM Architecture Overview

- Watchdog Timer
  - One 16-bit upcounting watchdog timer to monitor system operation for possible timeouts
- SHA224/256 HW accelerator
  - Intended to be used for signature generation, verification and generic data integrity checks
- PKC –ECC-256 Module
  - Hardware accelerator to support fast signature generation and verification with ECDSA



Those 3 modules are available only for A2G.

HSM Architecture Overview

- HSM Flash dedicated sectors in PFx for A1G:
- TC23x/TC27x/TC29
  - S6 -> 16KB
  - S16 & S17 -> 2 x 64KB = 128KB
- Only for TC27x/TC29x is available also Dflash
  - 8 x 8KB = 64KB

Logical Sector	Log. Sub-Sector	Size	Offset Address[1)]
HSM0	DF_HSM	8 KByte	$00'0000_H$
HSM1		8 KByte	$00'2000_H$
HSM2		8 KByte	$00'4000_H$
HSM3		8 KByte	$00'6000_H$
HSM4		8 KByte	$00'8000_H$
HSM5		8 KByte	$00'A000_H$
HSM6		8 KByte	$00'C000_H$
HSM7		8 KByte	$00'E000_H$

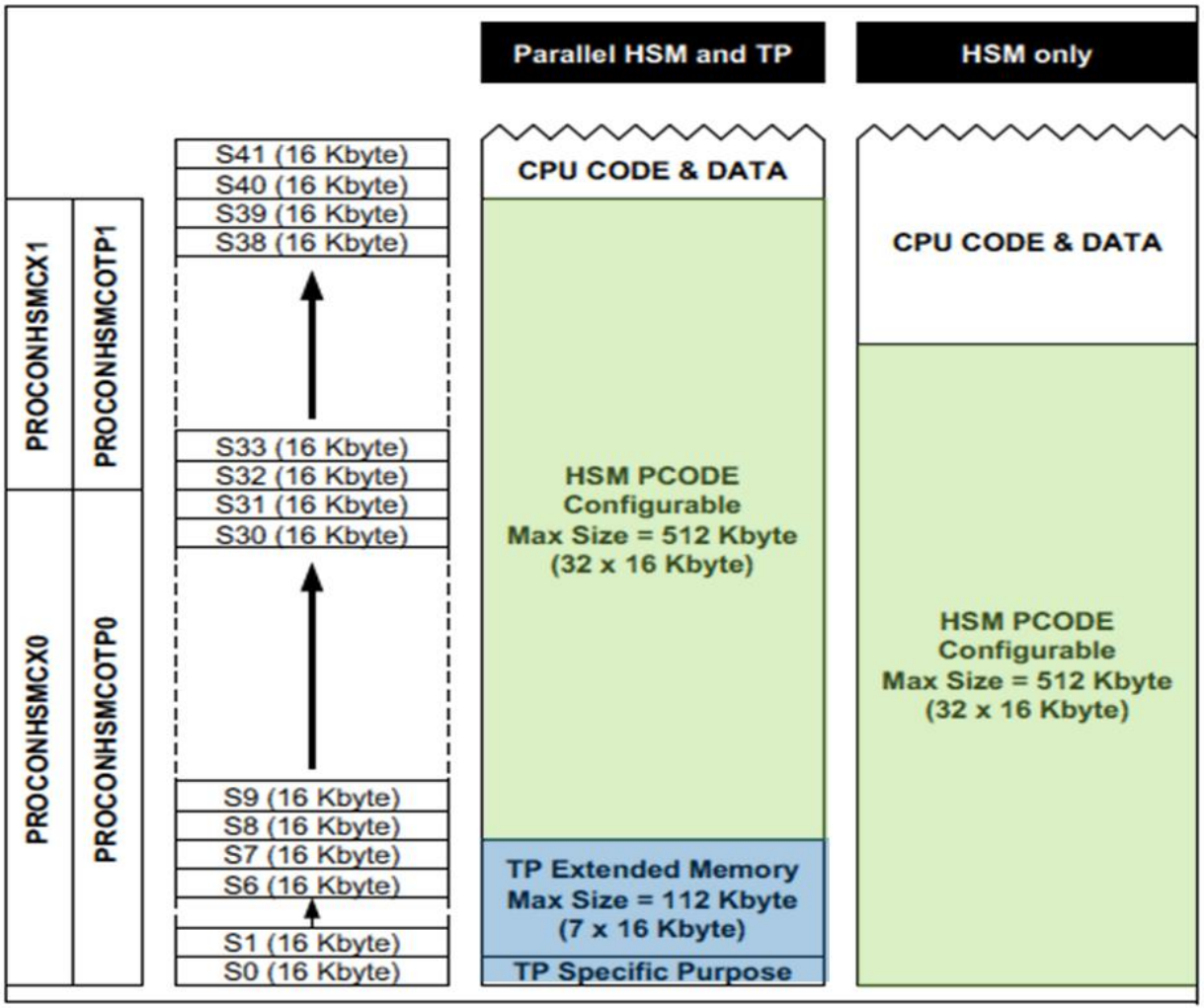
1) Offset with respect to AN_DFlash_B1 and AN_DFlash_B1F (see Table 11-1).

Logical Sector	Phys. Sub-Sector	Size	Offset Address[1)]
S0	PS0 (512 KB)	16 KB	$00'0000_H$
S1		16 KB	$00'4000_H$
S2		16 KB	$00'8000_H$
S3		16 KB	$00'C000_H$
S4		16 KB	$01'0000_H$
S5		16 KB	$01'4000_H$
S6		16 KB	$01'8000_H$
S7		16 KB	$01'C000_H$
S8		32 KB	$02'0000_H$
S9		32 KB	$02'8000_H$
S10		32 KB	$03'0000_H$
S11		32 KB	$03'8000_H$
S12		32 KB	$04'0000_H$
S13		32 KB	$04'8000_H$
S14		32 KB	$05'0000_H$
S15		32 KB	$05'8000_H$
S16		64 KB	$06'0000_H$
S17		64 KB	$07'0000_H$
S18	PS1 (512 KB)	64 KB	$08'0000_H$
S19		64 KB	$09'0000_H$
S20		128 KB	$0A'0000_H$
S21		128 KB	$0C'0000_H$
S22		128 KB	$0E'0000_H$
S23	PS2 (512 KB)	256 KB	$10'0000_H$
S24		256 KB	$14'0000_H$
S25	PS3 (512 KB)	256 KB	$18'0000_H$
S26		256 KB	$1C'0000_H$

1) Offset with respect to AC_PFx and AN_PFx (see Table 11-1).

HSM Architecture Overview

- HSM Flash dedicated sectors in PFx for A2G:
- Till 40 x 16KB = 640KB
- Configuration Options
  - If parallel TP and HSM operation are required then PF0 S0 to S39 my be configured for TP and HSM PCODE as follows:
    - PF0 S0: specific TP purpose
    - PF0 S1 to S7: TP extended memory
    - PF0 S8 to S39: HSM PCODE
    - TP, HSM PCODE and CPU address ranges should be contiguous



HSM Architecture Overview

› HSM DFlash dedicated sectors in DFLASH Bank 1 for A2G:

Table 6-13 Sector Structure of DFLASH Bank 1 in Single Ended Mode

Logical Sector	Physical Sector	Offset Address	Size	Total
EEPROM0	DF1_EEPROM	$00'0000_H$	4 Kbyte	4 Kbyte
EEPROM1		$00'1000_H$	4 Kbyte	8 Kbyte
EEPROM2		$00'2000_H$	4 Kbyte	12 Kbyte
EEPROM3		$00'3000_H$	4 Kbyte	16 Kbyte
...		...	...	...
EEPROM28		$01'C000_H$	4 Kbyte	116 Kbyte
EEPROM29		$01'D000_H$	4 Kbyte	120 Kbyte
EEPROM30		$01'E000_H$	4 Kbyte	124 Kbyte
EEPROM31		$01'F000_H$	4 Kbyte	128 Kbyte

Attention: For DF1_EEPROM configured in single ended mode the minimum erase size is 4 Kbyte aligned to the logical sector address boundary.

Table 6-14 Sector Structure of DFLASH Bank 1 in Complement Sensing Mode

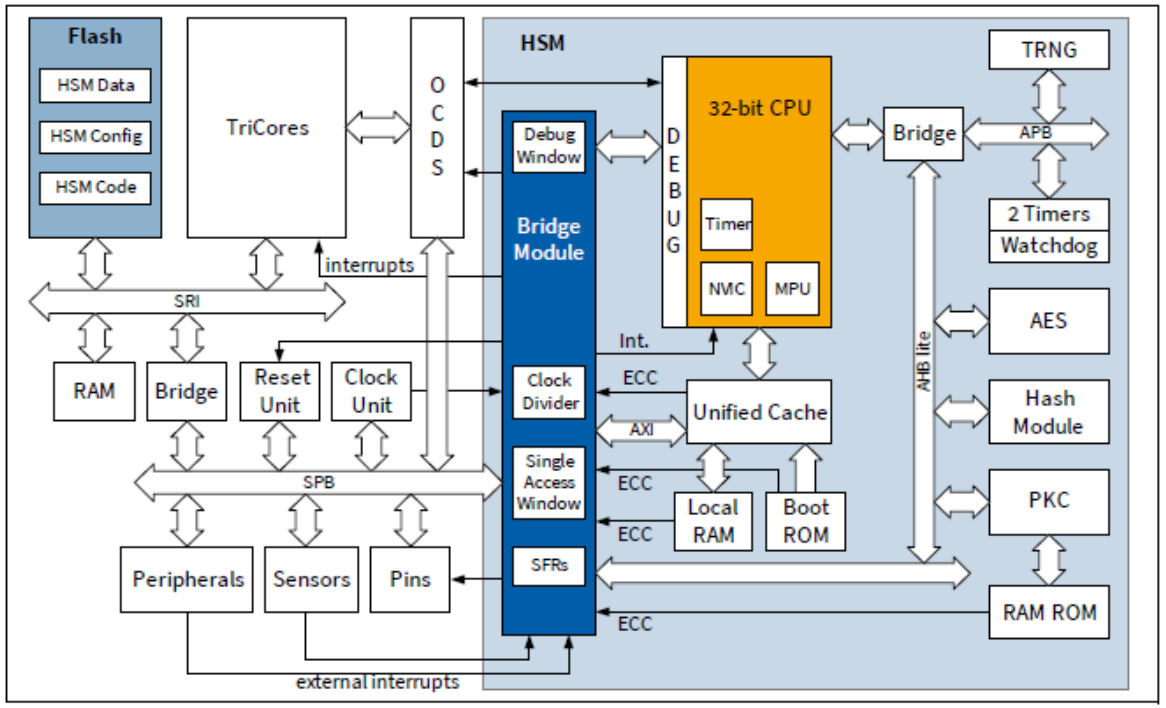
Logical Sector	Physical Sector	Offset Address	Size	Total
EEPROM0	DF1_EEPROM	$00'0000_H$	2 Kbyte	2 Kbyte
EEPROM1		$00'0800_H$	2 Kbyte	4 Kbyte
EEPROM2		$00'1000_H$	2 Kbyte	6 Kbyte
EEPROM3		$00'1800_H$	2 Kbyte	8 Kbyte
...		...	...	...
EEPROM28		$00'E000_H$	2 Kbyte	58 Kbyte
EEPROM29		$00'E800_H$	2 Kbyte	60 Kbyte
EEPROM30		$00'F000_H$	2 Kbyte	62 Kbyte
EEPROM31		$00'F800_H$	2 Kbyte	64 Kbyte

Attention: For DF1_EEPROM configured in complement sensing mode the minimum erase size is 2 Kbyte aligned to the logical sector address boundary.

HSM Introduction
Hardware Security Module (HSM)

Optional Module to implement *security relevant applications*:

- Secure boot
- Tuning protection
- Secure sensor communication
- Authentication
- Secure flash load
- Immobilizer (theft protection)
- Secure log and
- Secure debug authentication
- ….



Agenda
HSM Introduction

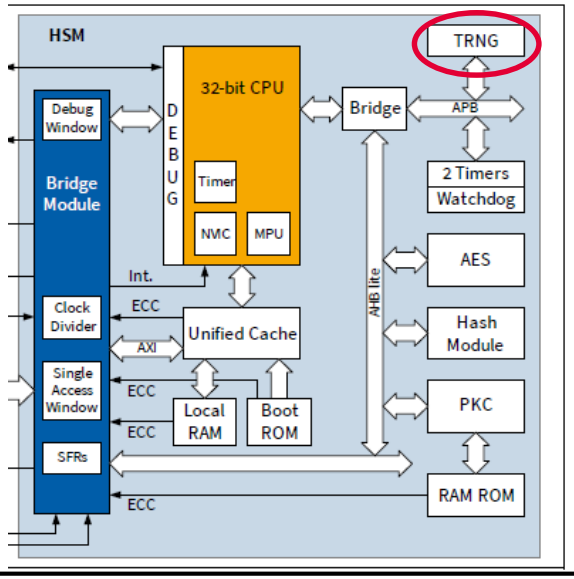
1. HSM Introduction and Architecture Overview
2. True Random Number Generator
3. AES 128 Encryption / Decryption Module
4. Public Key Cryptography (PKC) Module
5. SHA256- HASH Module
6. Performance Figures
7. Watchdog Timer
8. Bridge Module

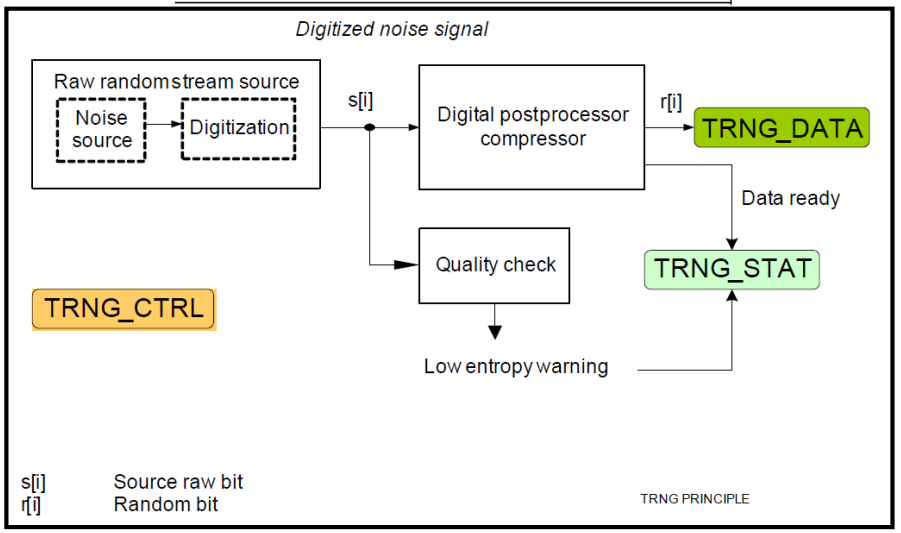
TRNG - True Random Number Generator

- To generate Random Numbers for:
  - Keys for cryptographic algorithms
  - Support protocols (challenges,..)

- High "entropy" is required
  - Uncertainty associated with a random variable
  - The lower the entropy, the more predictable the bit values
  - Post processing implemented to increase the entropy

- The Quality (min. entropy req.) is defined by:
  - AIS-31 publication by German BSI





TRNG Entropy

- What is the achievable Entropy with HSM implemented TRNG?
  - For a true random number generator according to the BSI AIS 20/31 functionality class PTG.2 the Shannon entropy per bit is at least H1 = 0.9991363
  - If 80 bits = 10 bytes of data are collected from the TRNG, this results in H1 = 80*0.9991363 = 79.93 bits of entropy.
  - If a PRNG based on the AES is seeded from this TRNG, one would anyway initialize the state with 16 byte of random data (i.e. 127.9 bit of entropy), where 16 byte is the block size of AES.

- Remark:
  - Note that the BSI document AIS 20/31 supersedes the AIS-31 and the old “P2 high” Is now called “PTG.2”. But this doesn’t change anything for the TRNG.

TRNG Throughput

- The time needed to generate a random number is not constant!!
  - Depends on the "entropy" level of the source
- The throughput R is given in units of kilobits per second with following formula

$$R\left[\frac{kbit}{s}\right] = \frac{f[MHz] \cdot 8000}{f[MHz] \cdot T[\mu s] + C} \qquad (3.1)$$

The constants T and C are given in the following table. They correspond to the constant and dynamic time components, respectively, for generating one byte. Hence, the number of clock cycles needed for generating one output byte is given by t = f×T+C.

Table 3-1 Generation Time for One Random Byte

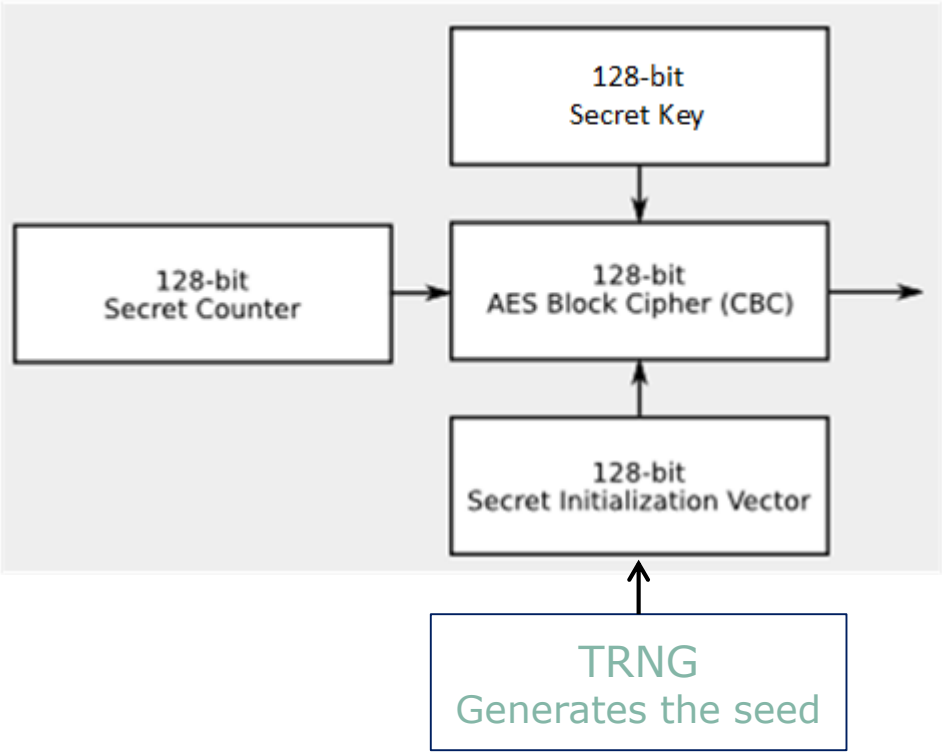
	Minimum	Typical	Maximum
Constant time T [μs]	9	18	71
Dynamic time C [clock cycles]	335	414	758

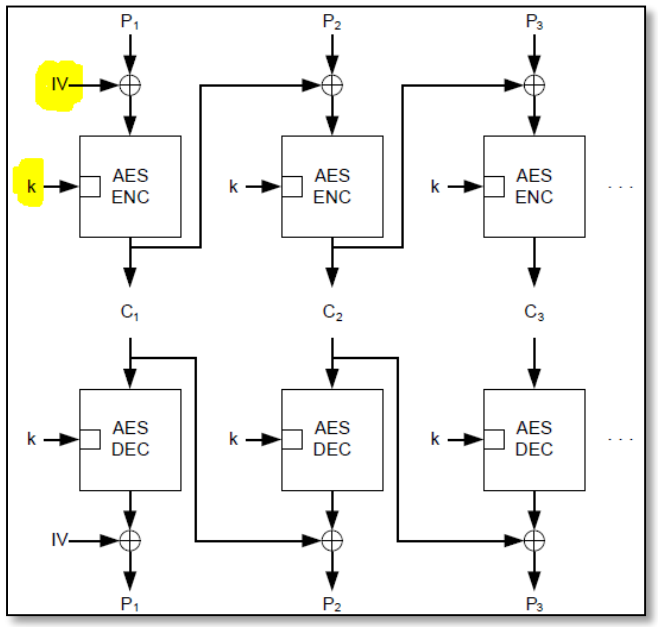
- For a clock frequency of 100MHz the typical throughput is approximately Rtyp = 360kb/s
- If the maximum number of clock cycles elapses before generation of a data block of a size specified by TRNG_CTRL.DBS, a warning is indicated (TRNG_STAT.WARN)

TRNG Operation

- The TRNG module continuously generates true random bytes for the system operation once enabled (TRNG_CTRL.DIS = 0).
- The availability of new true random data is signaled by TRNG_STAT.DTA_RDY bit and triggering the corresponding “output buffer not empty” interrupt
- The TRNG waits until the data has been read before recommencing generation.
- Once the data has been read (from TRNG_DATA) the flag TRNG_STAT.DTA_RDY is cleared by hardware.
- When the Data quality is too poor, no new number is generated (reported by FIPS_ERR and WARN bits + interrupt)
- The TRNG enters sleep mode when the module is disabled by setting the control bit TRNG_CTRL.DIS = 1

PRNG with TRNG + AES





- A pseudo RNG can be implemented using the TRNG + AES CBC (Cipher Block Chaining mode)
- PRNGs are much easier to construct in digital hardware than a TRNG
- Once a PRNG is properly seeded, it can generate unpredictable output very fast

Agenda
HSM Introduction

1. HSM Introduction and Architecture Overview
2. True Random Number Generator
3. AES 128 Encryption / Decryption Module
4. Public Key Cryptography (PKC) Module
5. SHA256- HASH Module
6. Performance Figures
7. Watchdog Timer
8. Bridge Module

AES 128 Module Features

- The AES module is a fast hardware device that supports encryption and decryption via a 128-bit key AES
- It enables plain/simple encryption and decryption of a single 128-bit data (i.e., plain text or cipher text) block as well as encryption or decryption of a multitude of data blocks of 128 bits each. For these, several so called modes of operation are implemented
  - ECB (electronic code book mode)
  - CBC (cipher block chaining mode)
  - CTR (32-bit counter mode)
  - OFB (output feedback mode)
  - CFB (cipher feedback mode)
- This enables also the additional modes
  - GCM (Galois counter mode)
  - XTS (XEX-based Tweaked Code Book mode (TCB) with Cipher Text Stealing (CTS))
- Furthermore, the AES module supports the following features:
  - Internal storage for 8 AES keys which are not readable
  - Key 0 and 1 are lockable, i.e., not writable nor readable
  - 5 context registers for different modes of operation
  - Maximal latency of 1 µs @ 100 MHz

AES 128 Module Registers

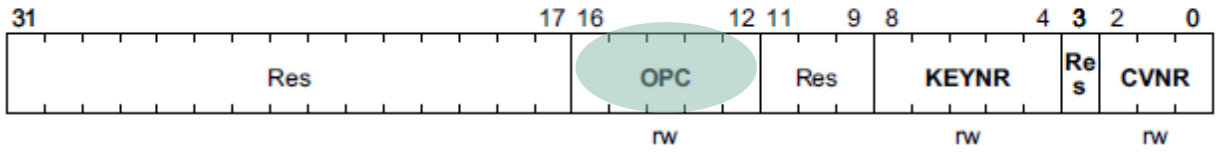
- HSM Contains **SFR Registers** and **Internal Registers**.
- The **SFR-registers** are directly accessible by the CPU
  - The control register **AESCTRL**
  - The status register **AESSTAT**
  - The input register **AESIN** which consists of **4x32-bit** word registers
  - The output register **AESOUT** which consists of **4x32-bit** word registers
    - not accessible by CPU during en-/decryption operation
  - The second output register **AESOUTSAVE** which consists of **4x32-bit** word registers. (used for pipelined usage)
- The internal registers are only accessible indirectly via the SFR-registers
  - **8 key registers** K0, K1, ..., K7 (128 bits).
  - The **5 chaining variable registers** CV0, CV1, CV2, CV3, CV4 which are each 128 bits

AES 128 Module AESCTRL Register (1/2)

- Following operations of the AES module are solely controlled by writing to the **AESCTRL** register:
  - Copying of a **key** value from **AESIN** to some K[x] (x=0,...,7) register.
  - Locking of the keys **K0** or **K1**, such that they are not writable any more.
  - Copying of an **IV (Initial Value)** value from **AESIN** to some chaining variable registers CV[y](y=0,..,4).
  - Copying of a CV value from some CVy to AESOUT.
  - Cryptographically processing of the input in AESIN, using at most one of the specified keys and chaining variables and outputting it in AESOUT.
  - Saving of the content of AESOUT into AESOUTSAVE. This is only an indirect process which will happen automatically if a process is triggered that will overwrite AESOUT.

AES 128 Module AESCTRL Register (2/2)

AESCTRL	Offset	Reset Value
AES Control Register	10_H	$0000\ 0000_H$



Field	Bits	Type	Description
OPC	16:12	rw	**Operation Code** This field defines the operation of the AES module. For an in-depth description of the commands, see the table below. *Note: An opcode with unspecified OPC will be ignored.* 00_H **ECB-ENC** encrypts input with ECB mode 01_H **ECB-DEC** decrypts input with ECB mode 02_H **CBC-ENC** encrypts input with CBC mode 03_H **CBC-DEC** decrypts input with CBC mode 04_H **CTR** en- and decrypts input with CTR mode 05_H **OFB** en- and decrypts input with OFB mode 06_H **CFB-ENC** encrypts input with CFB mode 07_H **CFB-DEC** decrypts input with CFB mode 08_H **GCM-ENC** encrypts input with GCTR mode and G-hashes OUT 09_H **GCM-DEC** decrypts input with GCTR mode and G-hashes IN $0A_H$ **GCM-MAC** G-hashes the input $0B_H$ **XTS-ENC** encrypts with XTS mode $0C_H$ **XTS-DEC** decrypts with XTS mode 10_H **WK** writes input to selected key register and resets IN 11_H **LK** locks the key registers K0 or K1 such that they are not writable anymore: Key is specified by KEYNR and works only with KEYNR=0 or 1, otherwise the command will be ignored. Furthermore LOCK0 or LOCK1 (in AESSTAT) is set to 1. 12_H **WCV** writes input to the selected chaining variable register 13_H **RCV** copies selected CV register to output register

Encryption/Decryption Operation Procedure

- Usually an en-/decryption operation is done in the following way:
  - The key is loaded into some K[y] register
  - Initial value IV is loaded into some CV[y] register.
  - Then the input register AESIN is filled with the input for the cryptographic operation.
  - En-/decryption operation is triggered by writing AESCTRL register fields:
    - OPC - Operation Code
    - KEYNR - Key Number
      - An opcode with invalid KEYNR will be ignored
    - CVNR - Chaining Variable Number
      - An opcode with invalid CVNR will be ignored
  - Waiting for AES (AESSTAT.BSY) to finish the operation
  - Reading the output out of AESOUT.

AES 128 Performance

- For encryption/decryption of 128-Bit block of data AES 128 CBC algorithm needs 14 clock cycles
- AES 128 performance seen on system level is not easy to derive since it is dependent on SW implementation and might differ due to:
  - Location of the AES 128 key
    - e.g. AES module, HSM SRAM or within HSM DFLASH
  - Location of HSM Code
    - HSM SRAM or HSM PFLASH
  - Utilization of implemented ARM-M3 Data Cache
    - e.g. interleaved parallel AES operations and data transfers
  - Efficiency of the used SW API from Host to HSM
    - e.g. AUTOSAR based protocol with Start/Update/Finish might add some additional latency
  - Additional Safety checks for the MAC operation
  - And last but not least the HSM operation frequency
    - 100Mhz are desired

Electronic Code Book Mode (ECB)

- **ECB** (electronic code book mode)
- Blocks of 128-bit are encrypted/decrypted using a 128-bit keyword

```
void AESencrypt(void) {
int i=0;
unsigned int *psrc, *pdst;
HSM_AES->AESIN0 = 0x16157e2b;
HSM_AES->AESIN1 = 0xa6d2ae28;
HSM_AES->AESIN2 = 0x8815f7ab;
HSM_AES->AESIN3 = 0x3c4fcf09;
HSM_AES->AESCTRL = AESCTRL_WK | AESCTRL_KEY5; // Load the key in K5

psrc = (unsigned int *) &pHOST2HSMbuf[0];
pdst = (unsigned int *) &pHSM2HOSTbuf[0];
while (i < 0x100) {
        HSM_AES->AESIN0 = hostread32_uncached((unsigned int) &(psrc++)[0]);
        HSM_AES->AESIN1 = hostread32_uncached((unsigned int) &(psrc++)[0]);
        HSM_AES->AESIN2 = hostread32_uncached((unsigned int) &(psr
        HSM_AES->AESIN3 = hostread32_uncached((unsigned int) &(ps
        // Encrypt using ECB with K5
        HSM_AES->AESCTRL = AESCTRL_ECB_ENC | AESCTRL_KEY5;
        // Do nothing loop while the AES is busy
        while ((HSM_AES->AESSTAT & AESSTAT_BSY_MASK) == AESSTAT_BSY)
                ;
                hostwrite32_uncached((unsigned int) &(pdst++)[0], HSM_AES->AESOUT0);
                hostwrite32_uncached((unsigned int) &(pdst++)[0], HSM_AES->AESOUT1);
                hostwrite32_uncached((unsigned int) &(pdst++)[0], HSM_AES->AESOUT2);
                hostwrite32_uncached((unsigned int) &(pdst++)[0], HSM_AES->AESOUT3);
                i += 4;
}
}
```

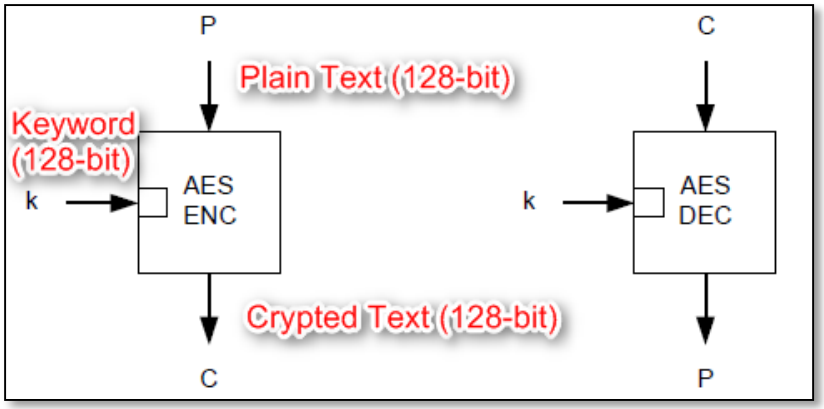
1 Load the Keyword

Load the Data (to encrypt)

2

3

Start the encryption



Cipher Block Chaining Mode (CBC)

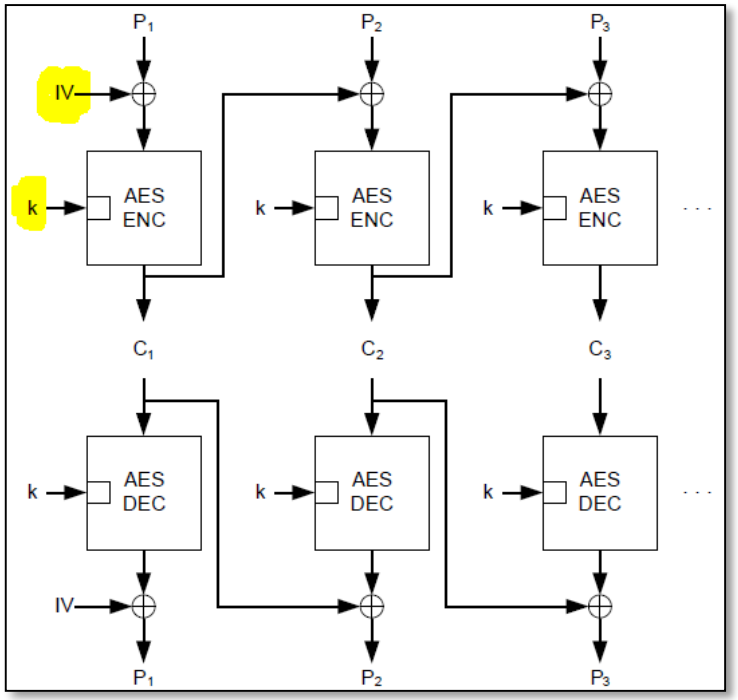
- **CBC** (cipher block chaining mode)
- Blocks of 128-bit are encrypted/decrypted using a 128-bit keyword + Initial Value (IV)

```
void CMAencrypt(void) {
int i;
unsigned int *psrc, *pdst;

HSM_AES->AESIN0 = 0x16157e2b; // Keyword
HSM_AES->AESIN1 = 0xa6d2ae28;
HSM_AES->AESIN2 = 0x8815f7ab;
HSM_AES->AESIN3 = 0x3c4fcf09;
// Load the key in K5
HSM_AES->AESCTRL = AESCTRL_WK | AESCTRL_KEY5;

HSM_AES->AESIN0 = 0x03020100; //Initial Value (IV)
HSM_AES->AESIN1 = 0x07060504;
HSM_AES->AESIN2 = 0x0B0A0908;
HSM_AES->AESIN3 = 0x0F0E0D0C;
// Load the key in var0
HSM_AES->AESCTRL = AESCTRL_WCV | AESCTRL_CV0; // IV--->CV0

psrc = (unsigned int *) &pHOST2HSMbuf[0];
pdst = (unsigned int *) &pHSM2HOSTbuf[0];
i = 0;
while (i < 0x4) {
         HSM_AES->AESIN0 = hostread32_uncached((unsigned int) &(psrc++)[0]);
         HSM_AES->AESIN1 = hostread32_uncached((unsigned int) &(psrc++)[0]);
         HSM_AES->AESIN2 = hostread32_uncached((unsigned int) &(psrc++)[0]);
         HSM_AES->AESIN3 = hostread32_uncached((unsigned int) &(psrc++)[0]);
         // Encrypt using ECB with K5 and V0
         HSM_AES->AESCTRL = AESCTRL_CBC_ENC | AESCTRL_KEY5 | AESCTRL_CV0;
         // Do nothing loop while the AES is busy
         while ((HSM_AES->AESSTAT & AESSTAT_BSY_MASK) == AESSTAT_BSY)
         ;
         hostwrite32_uncached((unsigned int) &(pdst++)[0], HSM_AES->AESOUT0);
         hostwrite32_uncached((unsigned int) &(pdst++)[0], HSM_AES->AESOUT1);
         hostwrite32_uncached((unsigned int) &(pdst++)[0], HSM_AES->AESOUT2);
         hostwrite32_uncached((unsigned int) &(pdst++)[0], HSM_AES->AESOUT3);
         i += 4;
         }
}
```



Agenda
HSM Introduction

1. HSM Introduction and Architecture Overview
2. True Random Number Generator
3. AES 128 Encryption / Decryption Module
4. Public Key Cryptography (PKC) Module
5. SHA256- HASH Module
6. Performance Figures
7. Watchdog Timer
8. Bridge Module

Public Key Cryptography (PKC) Module - Feature

- The PKC module is a hardware module that supports fast signature generation and verification with ECDSA.
  In particular, it enables modular and non-modular operations on integers and binary polynomials up to 256 bit length:
  - Multiplication
  - Modular addition and subtraction
  - Modular multiplication
  - Modular inversion and division
- It enables also complex algorithms on all common elliptic curves of bit length up to 256:
  - Addition of two points in affine coordinates
  - Doubling of a point in affine coordinates
  - Scalar multiplication
- The supported curves are all curves defined over finite fields of the type F_p and $GF(2^d) = F_2[X]/f$ of bit length up to 256 bit length:
  - This includes the **NIST curves** P-192, P-224, P-256, K-163, B-163, K-233, B-233, as well as the **Brainpool curves** brainpoolP160r1, brainpoolP192r1, brainpoolP224r1, brainpoolP256r1.
  - Additionally support for operations on Curve25519 and Ed25519 is included.
- Furthermore, the PKC module supports the following features:
  - Storage for 32 values (integers or binary polynomials) of up to 256 bit length.
  - **Generation of 200 ECDSA-signature/s @100MHz for elliptic curves of key-length 256.**
  - **Verification of 100 ECDSA-signature/s @100MHz for elliptic curves of key-length 256.**

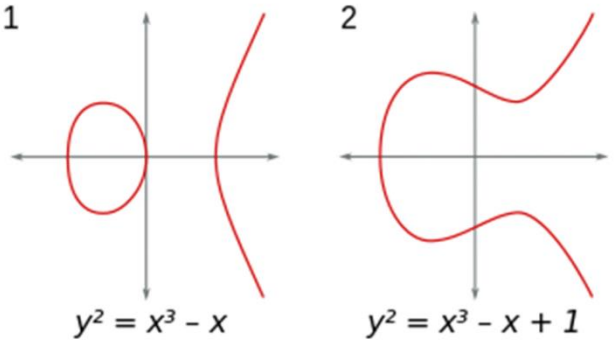
Elliptic Curves – Representation Variants

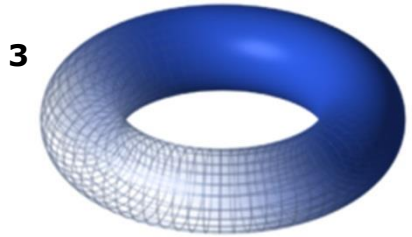
- There exist different possibilities to represent an elliptic curves as solution from algebraic notations:

- **Weierstrass-Notation [1]:**
  - $y^2 = x^3 + a*x + b$

- **Montgomery-Notation [2]:**
  - $B*y^2 = x^3 + A*x^2 + x$

- **(twisted) Edwards-Notation [3]:**
  - $a*x^2 + y^2 = 1 + d*x^2*y^2$





- The coefficient pairs (a,b), (A,B) or respective (a,d) do characterize/define the specific elliptic curve
- A point which belongs to the respective curve corresponds to a pair (x,y) from elements, which solves the corresponding equation above

Elliptic Curves – Comparison (1)

- The different algebraic notations supports different properties:
  - **Weierstrass-Equation:**
    - any elliptic curve can be represented by Weierstrass notation
    - NIST and Brainpool curves are implemented in Weierstrass notation
  - **Montgomery & Edwards-Equations:**
    - not all elliptic curves can be represented by Montgomery or Edward notations
    - the implementation of the formulas (in SW or HW) may offer some advantages for ‘smarter solutions’ which e.g. may lead to some speed (performance) advantage
      - curve ED25519 is represented in Edwards-Notation (indicated by ED)
        - e.g. in comparison to Weierstrass, a curve ED25519 HW accelerator implementation may offer a speed advantage of up to performance factor 1.3-1.5
  - Note:
    - one notation may be transformed from one form to another & vice versa

Elliptic Curves – Comparison (2)

- **NIST:**
  - defined in the 90-tees, but still not known to be compromised
  - different key-lengths available and correspondingly security protection levels (256 bit-key-length assumed to be adequately secure for the next 5-15 years)
  - represented in Weierstrass notation and optimized for SW implementation
    - this may impact on the other hand a side channel resistant HW implementation
  - politically not trusted (especially in Europe) because of involvement of NSA (in US)
    - generation algorithm very well known but the source of some parameters is still unclear

- **Brainpool:**
  - clean generated public curves without back door (with BSI involvement)
  - different key-lengths available (256 bit-key-length assumed to be adequately secure for the next 5-15 years)
  - represented in Weierstrass notation
    - not optimized for SW implementation, generic pseudo-random notation
      - good fit for side-channel resistant HW implementation
      - mainly used only in Europe today (rarely used in other parts of the world)
  - C2C Consortium in Europe decided to mandate as default

Elliptic Curves – Comparison (3)

Curve ED25519:

- created by D. Bernstein (from University of Chicago) & T. Lange
- very much hyped from several security consortiums (also within automotive community) (e.g. used today in commercial products like in Apple iOS >V9.0)
- exists in Montgomery and Edwards notations
- Curve ED25519 have a ***Cofactor >1*** (NIST and Brainpool have a Cofactor = 1)
  - Definition Cofactor:
    - In cryptography, an elliptic curve is a group based on a finite field F_n
      - this group has n elements on it, and we work on a prime-sized subgroup of size q
      - the value $h=n/q$ is denoted as the ***Cofactor*** of the curve
  - this might have some security impact on some protocols like e.g. Diffie-Hellman and most probably this will require some additional SW checks for security attack hardening (with the danger of SW patent infringement)
    - on the other hand, different SW solutions might differ in achieved security level on side channel resistance, which is not easy to determine
- Highly optimized for SW/HW performance with estimated increased performance factor from 1.3-1.5 versus NIST or Brainpool
  - Due to Cofactor > 1, a part of the performance advantage will be probably melted away by the necessity by additional SW hardening measures
  - no proven side channel resistance currently available from HW solutions (this risk is denied by inventors so far)

Elliptic Curves
Aurix TC3x HSM in context with ED25519 Curve

- The time being, when the ECC256 HSM HW accelerator IP was specified and demanded from TIER1 RfQs for AURIX TC3x, the curve **ED25519** was neither mentioned, nor demanded
  - in the recent time, we see an upcoming interest in the curve ED25519 (e.g. in Security WG from AUTOSAR or from OEM side)
  - therefore an internal feasibility study was performed by IFX and came to the following conclusions:
    - an upgrade of current ECC256 IP to support curve ED25519 will be included (earliest in TC39x, B-Step), but requires some IP modifications (additional microcode ROM and bigger Parameter SRAM) and correspondingly results in moderate ECC256 module chip size increase
    - estimated EdDSA performance results in increased approx. 130-150 ver/s (or correspondingly 260-300 sig/s) – excluding additional SW side channel hardening measures (as proposed one page before) - (an acceleration factor of 1.3-1.5 is assumed)

Agenda
HSM Introduction

1. HSM Introduction and Architecture Overview
2. True Random Number Generator
3. AES 128 Encryption / Decryption Module
4. Public Key Cryptography (PKC) Module
5. SHA256- HASH Module
6. Performance Figures
7. Watchdog Timer
8. Bridge Module

SHA256 - Features

- The hash module is capable of executing either the MD-5, SHA-1 or SHA-224/SHA-256 function using a common internal engine.
- The module is intended to be used for signature generation, verification and generic data integrity checks.
  - The time required to process one 512-bit input data block depends on the selected algorithm:

Hash Algorithms and Performance

Algorithm	Clock Cycles per 512-bit Data Block	Notes
MD-5	65	—
SHA-1	81	—
SHA-224	65	Special software handling needed,
SHA-256	65	—

65 cycles corresponds to
eoretical reachable 98MByte/s
SHA256 module performance

realistic system performance:
30-50MByte/s

- The module supports multi-tasking environments
  - however, for preemptive multi-tasking, it will be necessary to use a software abstraction layer as the module does not support stopping or resuming hash calculations at arbitrary points in time.

SHA256 – HASH Module

- Example code for HASH SHA256
  - SHA256 algorithm choosen with HASH order out MSW first

```
//Example taken over from Target Spec for HASH SHA256process
void AES_HASH_SHA256()
{
    int i;
    unsigned int *psrc, *pdst;

    // Start HASH Process using the HASH_ALGO SHA 256 and HASH ORDER OUT MSW_FIRST
    HSM_HASH->HASH_CFG = (HASH_CFG_ALGO_Msk | HASH_ALGO_SHA_256)|(HASH_CFG_ORDER_OUT_Msk | HASH_ORDER_OUT_MSW_FIRST);


    psrc = (unsigned int *) &pHOST2HSMbuf[0];
    pdst = (unsigned int *) &pHSM2HOSTbuf[0];
    for (i=0; i<16; i++)
    {
#ifdef TOMG_NO_CACHE
        HSM_HASH->HASH_DATA = pHOST2HSMbuf[i];
#else
        HSM_HASH->HASH_DATA = hostread32_uncached((unsigned int) &(psrc++)[0]);
#endif
    }

        // Do nothing loop while the HASH is busy
        while ((HSM_HASH->HASH_STAT & HASH_STAT_BSY_Msk) == HASHSTAT_BSY)
            ;
        while ((HSM_HASH->HASH_STAT & HASH_STAT_CNT_Msk) != 0x0){
            for(i=0; i<8; i++)
            {
#ifdef TOMG_NO_CACHE
                pHSM2HOSTbuf[i] = HSM_HASH->HASH_VAL;
#else
                hostwrite32_uncached((unsigned int) &(pdst++)[0], HSM_HASH->HASH_VAL);
#endif

            }
        }
}
```

Agenda
HSM Introduction

1. HSM Introduction and Architecture Overview
2. True Random Number Generator
3. AES 128 Encryption / Decryption Module
4. Public Key Cryptography (PKC) Module
5. SHA256- HASH Module
6. Performance Figures
7. Watchdog Timer
8. Bridge Module

ECC256 – SW versus HW Performance Figures

- ECC256 is estimated to get an approx performance from up to 3-6 sig/s
  - please refer to the table below

ECC256 [operations/s]	ARM Cortex M3 (100MHz – SW)	TC3xx (HW)	Speed Factor Delta (HW/SW)
Scalar Multiplication	≈ 3-6	≈ 228	≈ 28-76
Verify (ECDSA256)	≈ 1.5-4	≈ 100	≈ 25-66
Sign (ECDSA256)	≈ 3-6	≈ 200	≈ 33-66

- external company INVIA claims up to four ECDSA256 Verify operations with a ARM M3 100MHz processor with their SW solution
  - http://invia.fr/cryptography/software-ecc.aspx
    - all ANSI standard curves supported;
      - all NIST standard curves supported;
      - ECDSA key generation, signature and verification;
      - ECDH key generation and common key functions;
      - core functions optimized for the targeted processor;
      - SPA, DPA[1] and DFA resistant through state-of-the-art countermeasures;
      - configurable architecture:
      - – adjustable trade-off between performance and RAM footprint;
      - – dedicated hardware accelerator available separately (PK2C).
      - typical code size on Cortex-M3: 20 kbytes;
      - **less than 25 Mcycles to compute a secured 256-bit ECDSA signature verification;**
      - low RAM footprint (1.2 kbytes for a secured 256-bit ECDSA signature verification);
      - contactless protocols are supported down to 25 MHz (using PK2C accelerator).

RSA1024/RSA2048 – SW versus HW Performance Figures

Please check:
Please check the table.

- **some background info's to**
  http://realtimelogic.com/products/sharkssl/Cortex-M3/
  - RSA1024 @ 100MHz with Assembler optimized Code:
    - 13/2ms = 6.5 ms for encryption (153.8 verifications/s)
    - 311/2ms = 155.5ms for decryption (6.4 signatures/s)
  - RSA1024 encryption (verification) can be simplified by using smaller public keys for the verification process (e.g. Exp≤2^{16}+1 instead of 1024-bit)
    - e.g. Tier1 is using 17-bit public key length instead of 1024-bit key length for verification of the secure Flash-Bootloader

RSA1024 [operations/s]	**ARM Cortex M3 (100MHz – SW)**
Verify	1.6
Verify Exp≤2^{16}+1	154
Sign	1.6
Sign with CRT (Chinese Remainder Theorem)	6.4

RSA2048 [operations/s]	**ARM Cortex M3 (100MHz – SW)**
Verify	0.25
Verify Exp≤2^{16}+1	24
Sign	0.625
Sign with CRT (**C**hinese **R**emainder **T**heorem)	2.5

  - Assumption: Sign in SW done with CRT; Pre- and Post-Processing excluded
  - TC3xx (HW) : figures include Pre- and Post-Processing

OEM SW Benchmark Results
ARM M3 – 80MHz

- RSA3072 with Exp $^{16+1}$
  - RSA-PSS (**P**robabilistic **S**ignature **S**cheme):
    - 1 verification at 185ms (5.4ver/s)
    - **code size: 5kB**

- ECC256: NIST curve
  - ECDSA (**E**lliptic **C**urve **D**igital **S**ignature **A**lgorithm):
    - 1 verification at 365ms - (2.7ver/s)
    - **code size: 10kB**

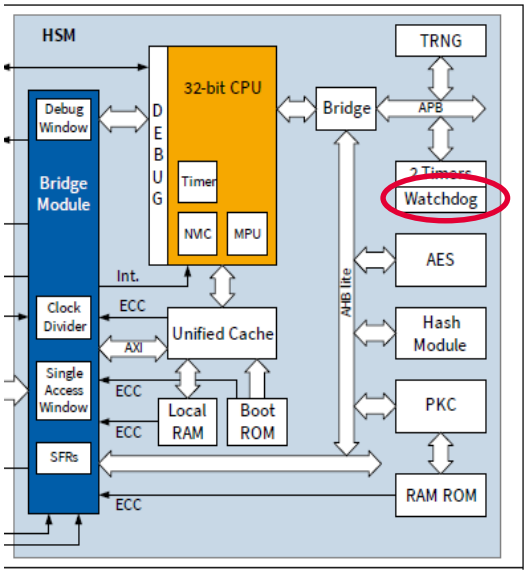
- ED25519: (Bernstein curve in Edwards notation)
  - EdDSA: 1 verification at 149ms – (6,7 ver/s)
    - **code size: 80kB**
      - is blocking for AURIX HSM in case for usage of secure FLASH Boot Loader
      - AURIX HSM SRAM size: 40kB / AURIX 2G: 96kB
    - or code size optimized: 17,5KB with 1 verification in **2s (0.5 ver/s)**

Agenda
HSM Introduction

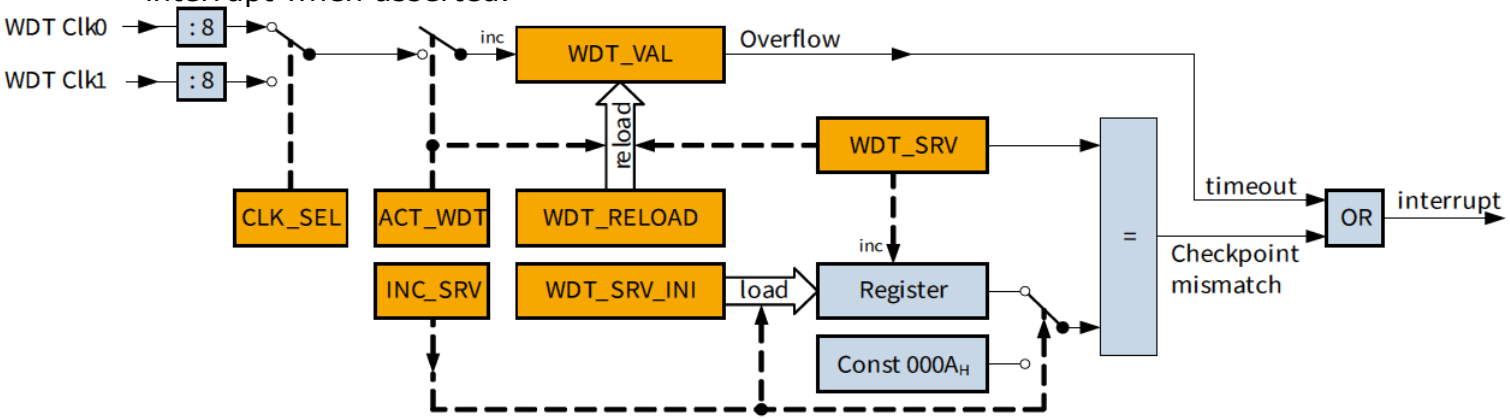
1. HSM Introduction and Architecture Overview
2. True Random Number Generator
3. AES 128 Encryption / Decryption Module
4. Public Key Cryptography (PKC) Module
5. SHA256- HASH Module
6. Performance Figures
7. Watchdog Timer
8. Bridge Module

Watchdog Timer - Feature

- The timer module features a watchdog timer to monitor system operation for possible time-outs and to check for the correct order of operations:
  - One 16-bit upcounting watchdog timer.
  - Two selectable clock sources, one fixed prescaler
  - Checkpoint functionality.
    - Separate watchdog time-out and checkpoint mismatch events.
  - The frequency of the selected input clock is divided by another fixed factor of "8"
    - Effectively, the HSM system clock is divided by a factor of 1024 (128 * 8).
    - Assuming a system frequency of 100 MHz the watchdog timer can realize time out periods up to ~ 670 ms.

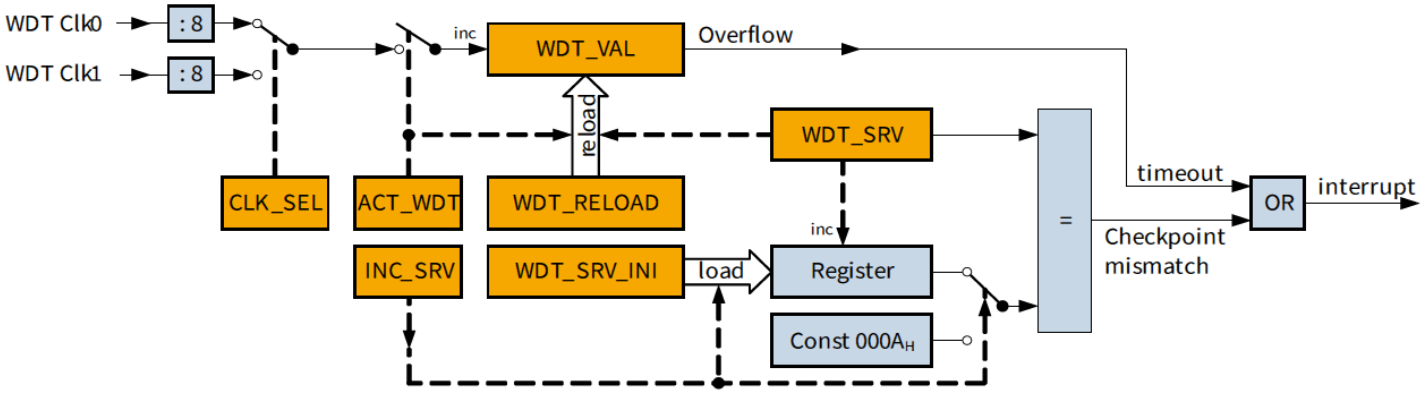


- Functional Description
  - The grey boxes are registers or bit fields which can be read or written by software.
  - The watchdog time-out and checkpoint mismatch outputs trigger an interrupt when asserted.



Watchdog Timer - Operation and Service

- The key elements of the watchdog timer are a counter register **WDT_VAL** and a reload register **WDT_RELOAD**.
  - The Reload register contain the initial value of the counter
  - The Watchdog timer is started by setting the **WDT_CRTL.ACT_WDT** bit.
- When the counter value switches from 0xFFFF to 0x0000 an overflow occurs. This time-out of the watchdog causes an interrupt.
- The WDT timer is serviced by writing to the service register **WDT_SRV**. Two modes can be distinguished depending on the control bit **WDT_CRTL.INC_SRV**:
  - If bit INC_SRV is cleared, the WDT timer has to be serviced by writing the value 0x000A to the WDT_SRV register.
  - If bit INC_SRV is set, the value which will be written to the WDT_SRV must be incremented by one each time the register is written, otherwise an interrupt will be triggered. The initial value is contained in the **WDT_SRV_INI** register and it has to be set by SW before enabling the watchdog.



Watchdog Timer - Programming model

- The watchdog timer supports two main applications:
  - Watchdog operation
  - Checkpoint operation
- Those two operations are independent of each other, then different modes can be generated:

WDT_CTRL		Effect
ACT_WDT	INC_SRV	
0	0	Watchdog timer is not running, the value $000A_H$ must be used whenever writing to WDT_SRV.
1	0	Watchdog timer operation: the value $000A_H$ must be used whenever writing to WDT_SRV.
0	1	Checkpoint operation: the value written to WDT_SRV must be incremented by one each time.
1	1	Combined Watchdog and Checkpoint operation: the watchdog timer is running, the value written to WDT_SRV must be incremented by one each time.

Watchdog Timer – Watchdog mode

- In „Watchdog mode" the basic application is to service the timer periodically such that a time out will be avoided.
- The timer is serviced by writing to the service register WDT_SRV.
- The watchdog is configured for this operation by the following steps:
  - Disable checkpoint operation by clearing the INC_SRV bit.
  - Stop the watchdog by clearing the ACT_WDT bit if not already stopped, otherwise the Reload value will not be copied in the counter register.
  - Select the appropriate clock source by the CLK_SEL bit.
  - Configure the reload value for the desired time.
  - Enable the Watchdog by setting the ACT_WDT.
- To serve the Watchdog, the constant value 0x000A has to be written in the WDT_SRV register.

Watchdog Timer – Checkpoint mode

- In „Checkpoint mode“ the watchdog is serviced by writing ascending values into the service register WDT_SRV.
- In this way it´s possible to check that the service points are handled in the correct order, providing a hardware flow control to the software.
- The watchdog is configured for this operation by the following steps:
  - Disable watchdog operation by clearing the ACT_WDT bit.
  - Clear the INC_SRV bit, because the initial value will be loaded from the WDT_SRV_INI only if INC_SRV changes from 0 to 1.
  - Select the desired initial value by the WDT_SRV_INI.
  - Enable the checkpoint operation by setting the INC_SRV: the first valid service value will be the WDT_SRV_INI + 1.
- To serve the Watchdog in „Checkpoint mode“ ascending values starting with the WDT_SRV_INI + 1 has to be written to WDT_SRV register.

Watchdog Timer – Resolution and Period

- Assuming a stable frequency of the input clock, the resolution of the timer is:
  - $t_{res}[\mu s] = \frac{128*8}{f[MHz]}$
- The timeout of the watchdog in clock cycle is:
  - $n_{\text{period}} = 128 * 8 * (65536 - RELOAD\_VALUE)$
    - Or:
  - $t_{period}[\mu s] = t_{res}[\mu s] * (65536 - \text{RELOAD}_{\text{VALUE}}) = \frac{128*8*(65536-RELOAD\_VALUE)}{f[MHz]}$

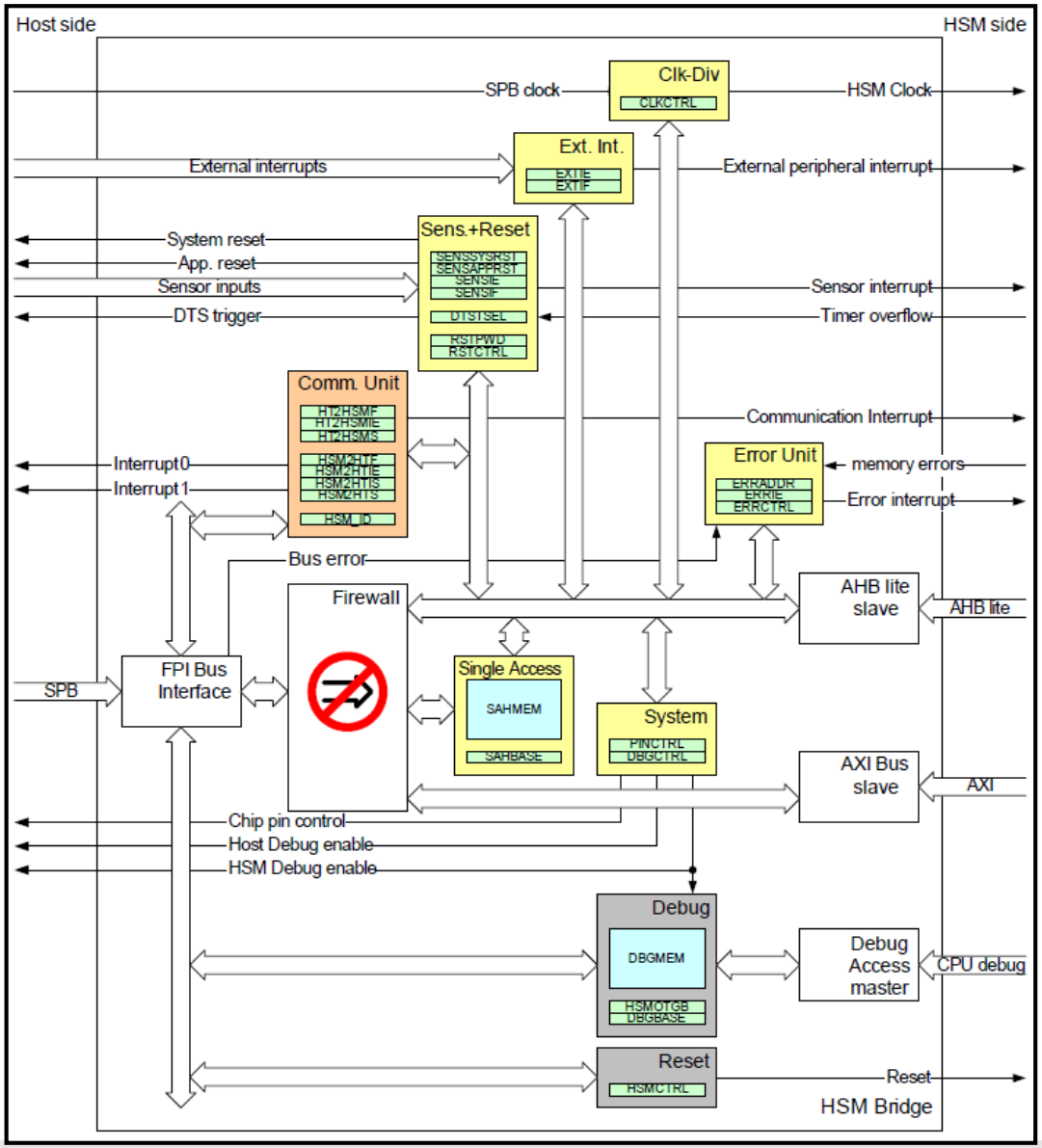
Agenda
HSM Introduction

1. HSM Introduction and Architecture Overview
2. True Random Number Generator
3. AES 128 Encryption / Decryption Module
4. Public Key Cryptography (PKC) Module
5. SHA256- HASH Module
6. Performance Figures
7. Watchdog Timer
8. Bridge Module

Bridge Module
Communication HSM-Host

HOST (TriCore)

Restricted Access to HSM resources (in normal mode)



HSM

Full Access to Host System

HSM accessible
HSM and host accessible
Host accessible

SFR
Memory window

Comm. Unit

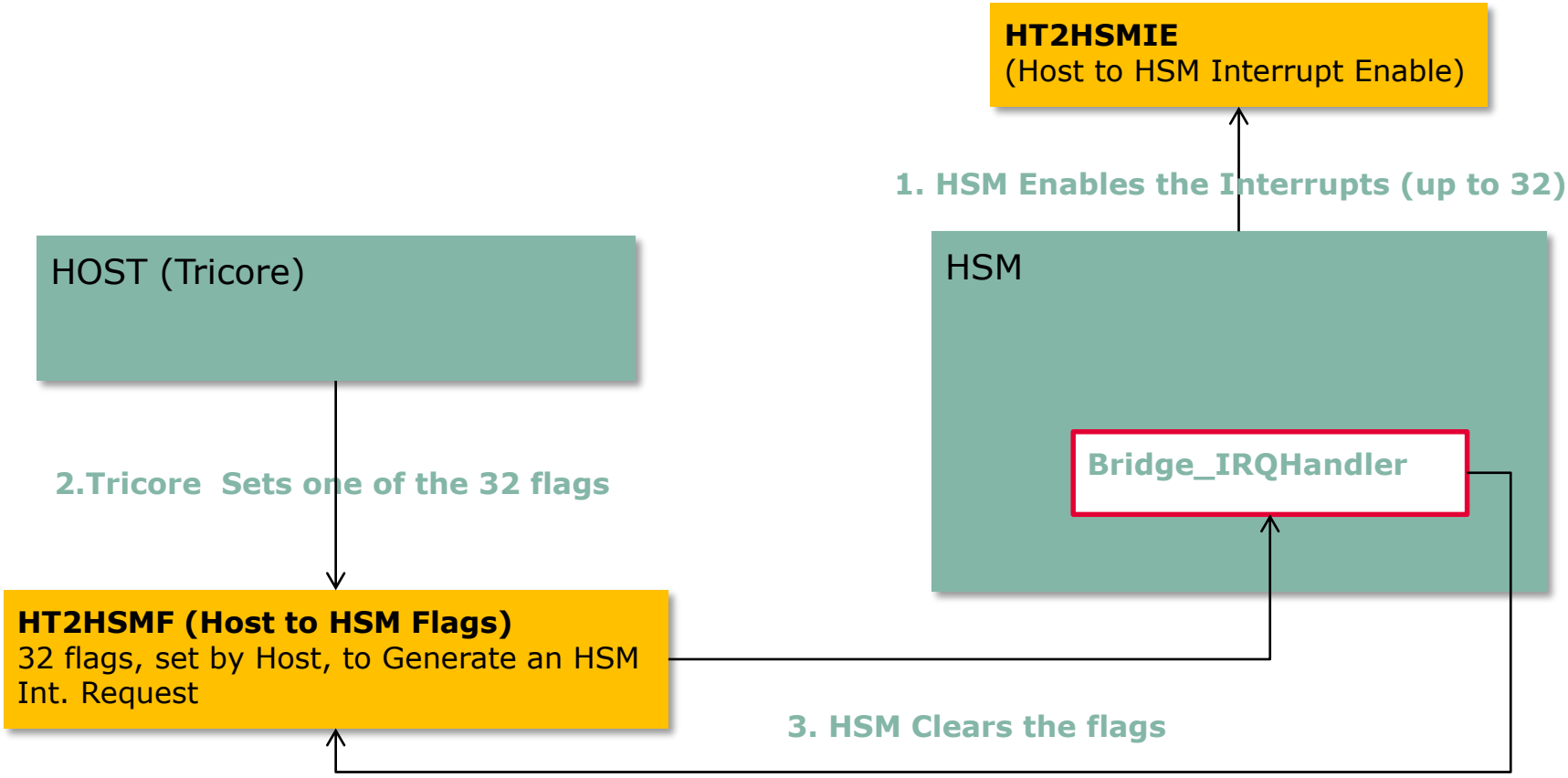
HT2HSMF
HT2HSMIE
HT2HSMS

HSM2HTF
HSM2HTIE
HSM2HTIS
HSM2HTS

HSM_ID

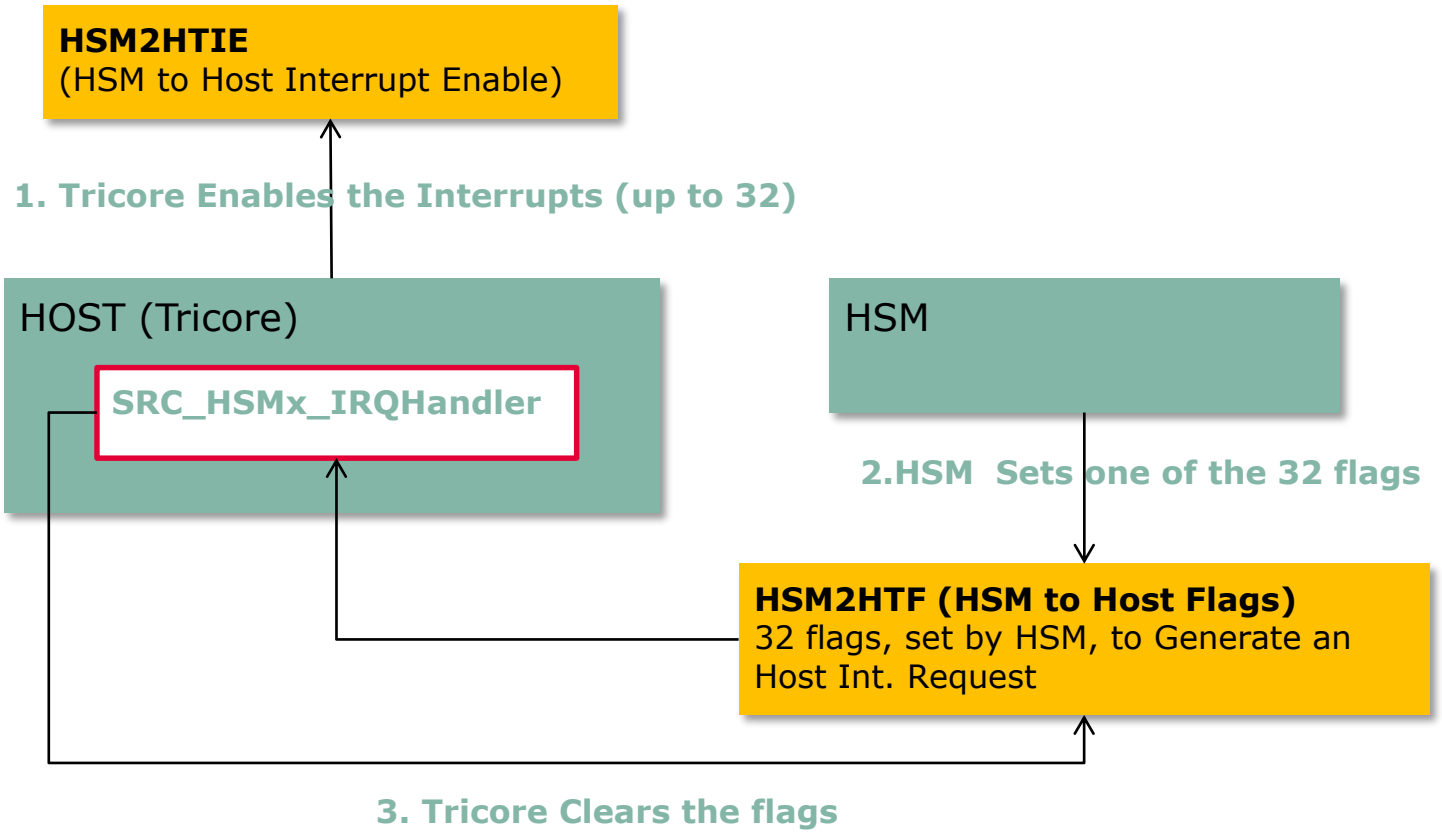
Bridge Module
Communication Unit Interrupts (1/2)

› Host to HSM Interrupts



Bridge Module
Communication Unit Interrupts (2/2)

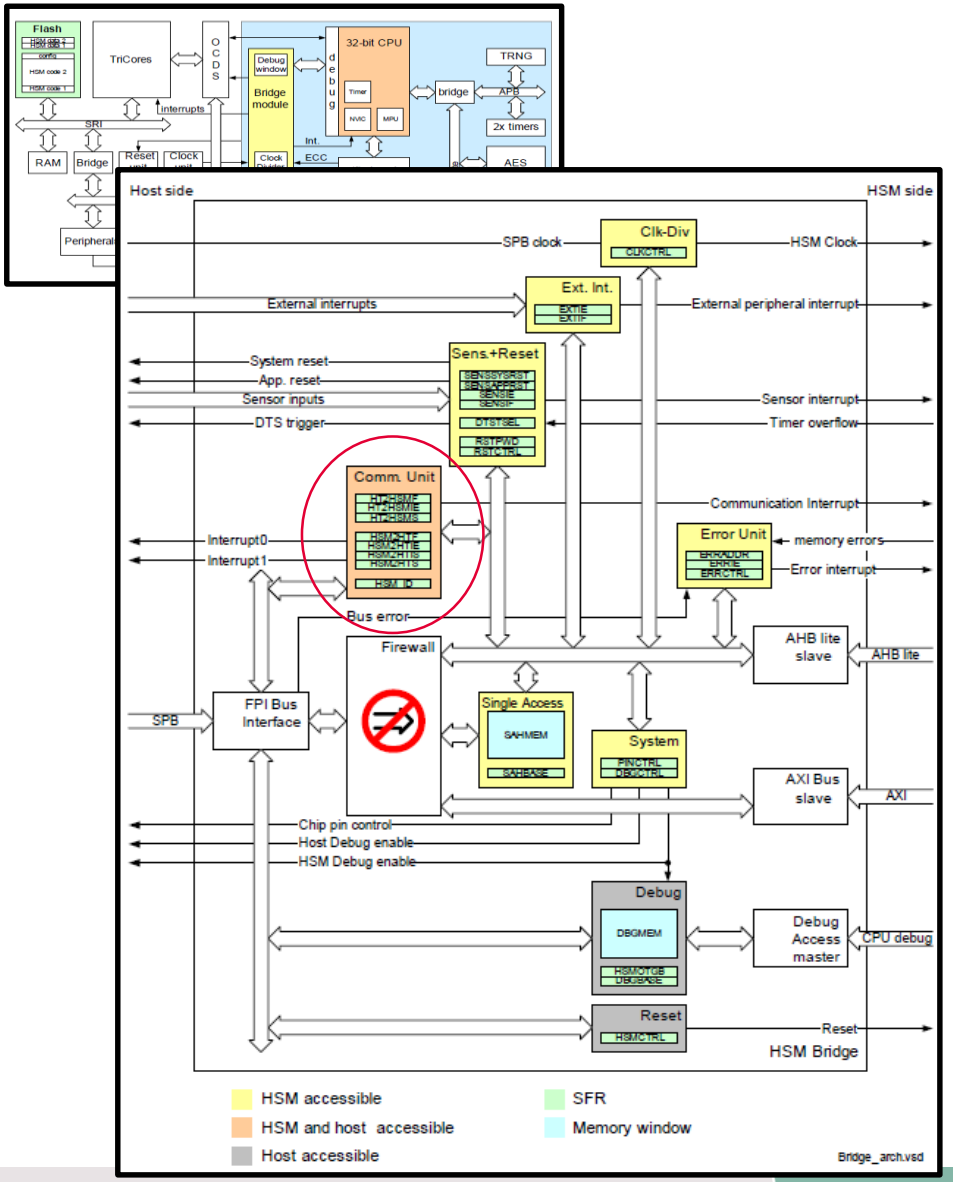
› HSM to Host Interrupts



Data Sharing between HSM and TriCore

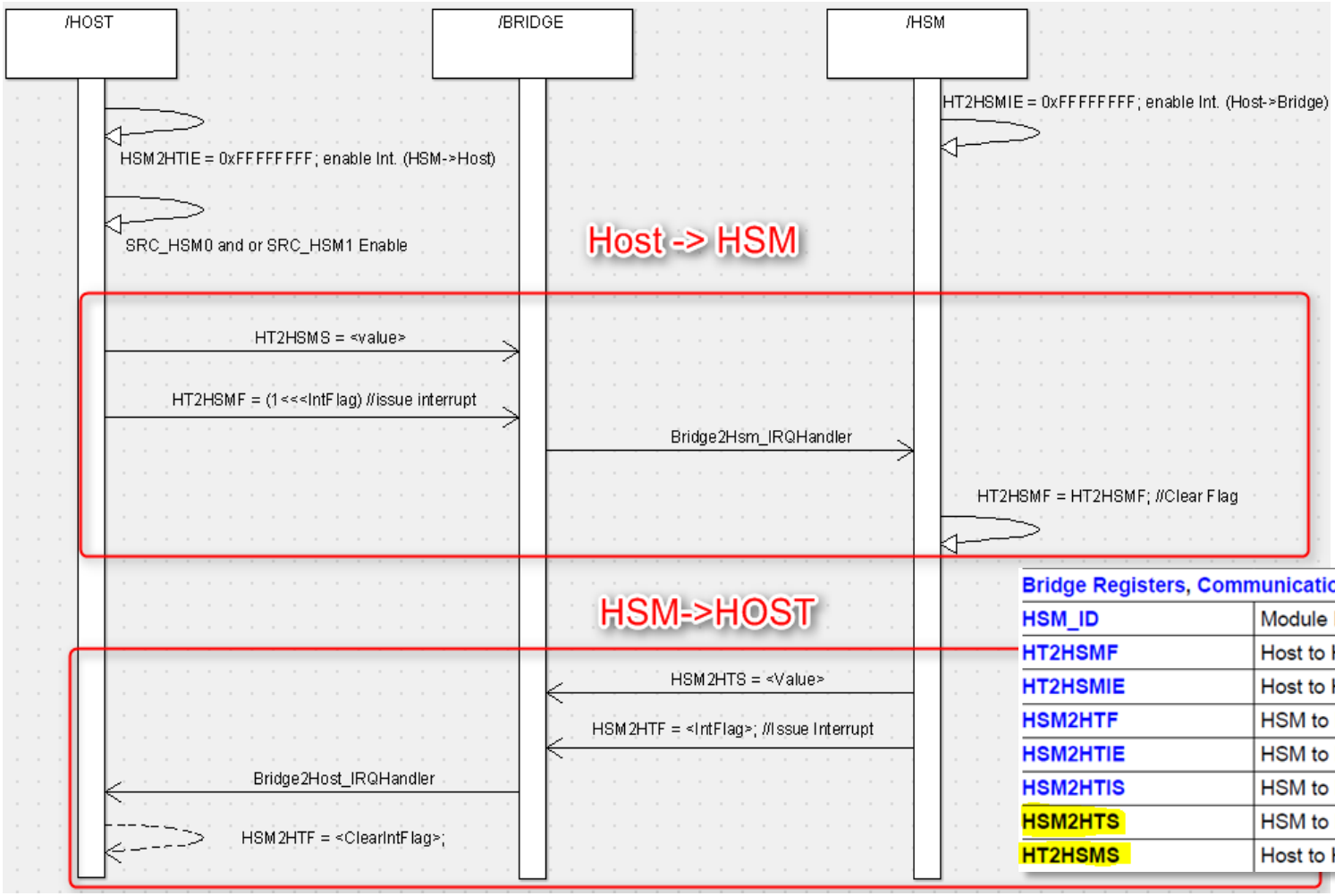
- The **Bridge Module** connects the HSM to the Host and enable the communication between them.

- The **Communication Unit** can be accessed by HSM and Host:
  - Information exchange between host and HSM
    - **Bridge Registers (symmetrical)**
    - **Shared Memory Region**
  - Joint usage of host peripherals
    - **No semaphore and mutex available**, bridge registers needs to be used.



Data Sharing between HSM and TriCore

- To share small information (32-bits) between HSM and Host
  - **HSM2HTS** and **HT2HSMS** registers can be used

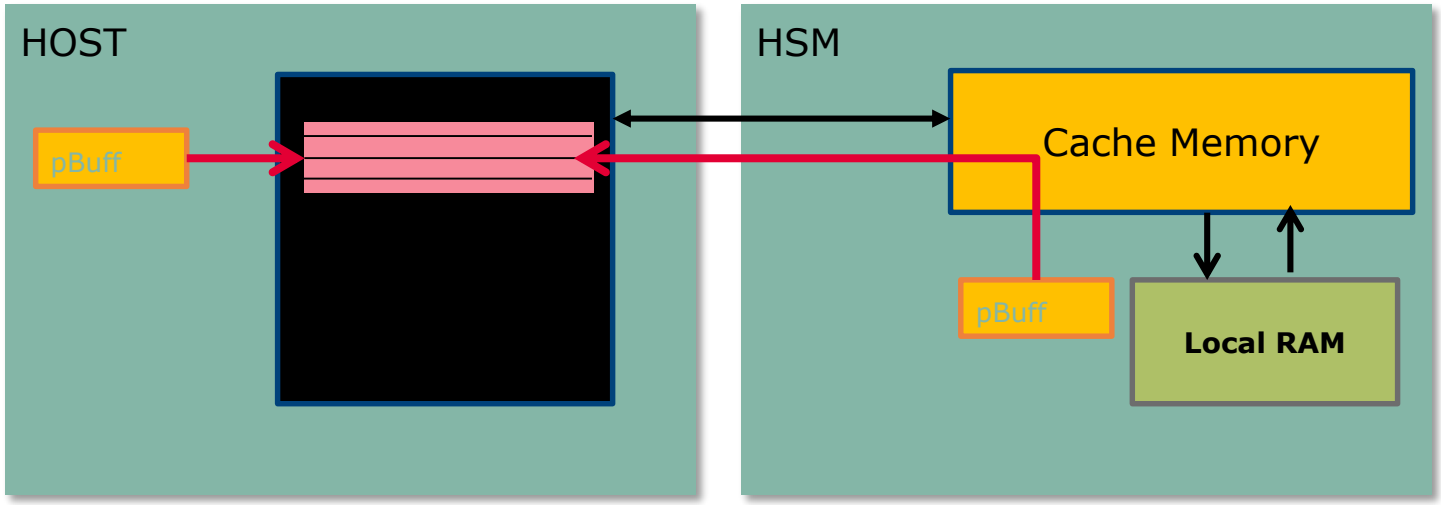


Bridge Registers, Communication Unit	
HSM_ID	Module Identifier Register
HT2HSMF	Host to HSM Flag Register
HT2HSMIE	Host to HSM Interrupt Enable
HSM2HTF	HSM to Host Flag Register
HSM2HTIE	HSM to Host Interrupt Enable
HSM2HTIS	HSM to Host Interrupt Select
HSM2HTS	HSM to Host Status
HT2HSMS	Host to HSM Status

Data Sharing between HSM and TriCore

Please check:
Changed a fill color to match style.

- To share bigger information, a shared memory could be used (HOST side):



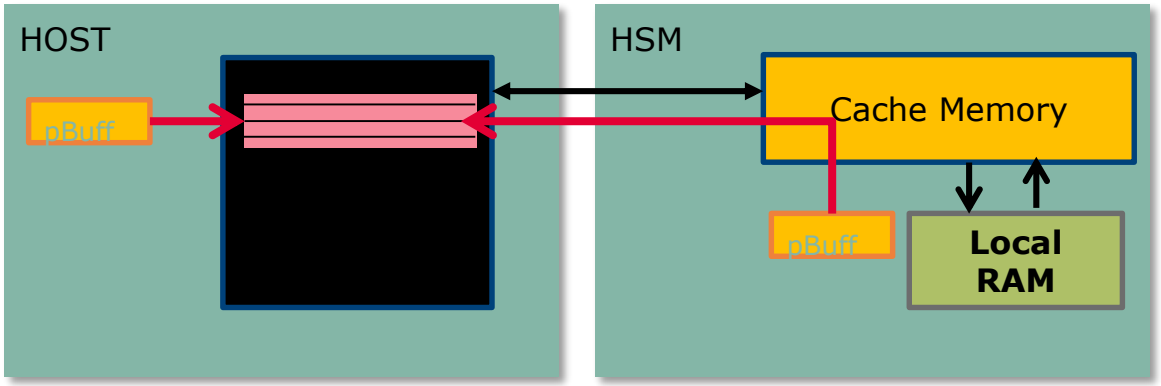
- The **Cache memory** has to be managed in the proper way

- The Cache memory can be by-passed using a user defined **64KB memory windows**

Data Sharing between HSM and TriCore

Please check:
Changed a fill color to match style.

- If the Cache is used:
  - The Host sends the &pBuff to the HSM using HT2HSMS register
    - HSM_HT2HSMS.U=(uint32)&HOST2HSMbuf[0]
  - The HSM defines a variable to point to the &pBuff
  - The pBuff is pointing not to the HOST RAM, but to the Cache
  - After the initialization, the values needs to be copied from the cache to the Host RAM



Cache SFRs, Configuration SFRs	
CACHE_CONFIG	Cache Configuration Register
CACHE_CTRL	Cache Control Register
Cache SFRs, Command SFRs	
CACHE_AC	Cache All Clean Register
CACHE_SC	Cache Set Clean Register
CACHE_BC	Cache Block Clean Register
CACHE_BT	Cache Block Touch Register
CACHE_BL	Cache Block Lock Register
CACHE_BU	Cache Block Unlock Register

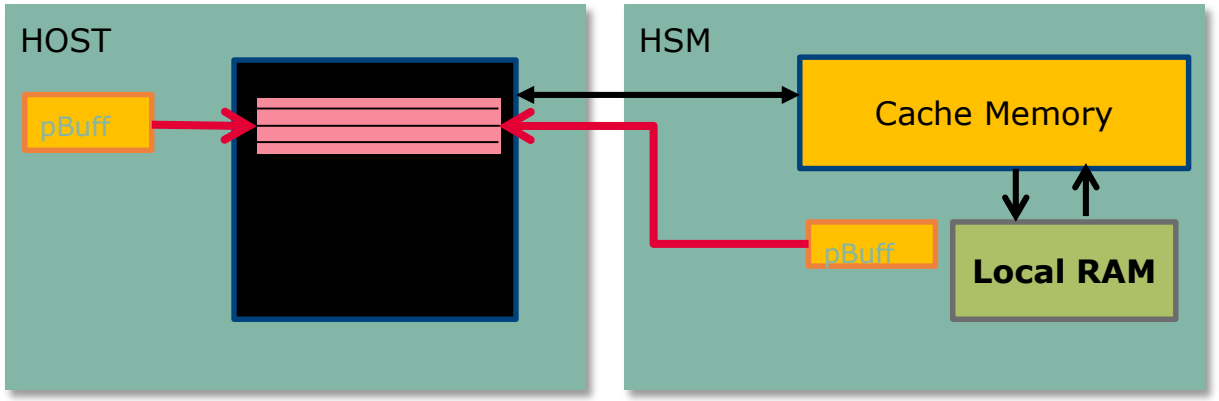
HSM Side Code

```
case CMD_SETADDR_HOST2HSMBUF: {
    pHOST2HSMbuf = (unsigned int *) HSM_BRIDGE->HT2HSMS; //pointer to buffer on Host
    for (i = 0; i < 0x100; i++)
        pHOST2HSMbuf[i] = 0x00010000 | i; //fill with a pattern
    //the data is now in cache, you have to clean the cache
    //for sure this is unefficient (does not consider cache line lenght),
    //but I want to demonstrate the pitfall if you are working with caches
    for (i = 0; i < 0x100; i++)
        HSM_CACHE->CACHE_BC = (unsigned int) &pHOST2HSMbuf[i];
    HSM_BRIDGE->HSM2HTS = 0xCCCC0000 | command; //confirm that the command was executed
    command = CMD_NOTHINGTODO;
    HSM_BRIDGE->HSM2HTF = 0x8; //Raise an Interrupt to the Host (Tricore)
    break;
}
```

Data Sharing between HSM and TriCore

Please check:
Changed a fill color to match style.

- Without the Cache:
  - A window (max 64Kbyte) can be addressed without using the cache
  - The **SAHBASE** and **SAHMEM** registers need to be used




```
Hostaddress = [SAHBASE]+(HSMadress-SAHMEM)& 0xFFFF
```

```
case CMD_SETADDR_HSM2HOSTBUF: {
    pHSM2HOSTbuf = (unsigned int *) HSM_BRIDGE->HT2HSMS;
    // Problem is here if buffer wraps 64Kbyte boundary
    HSM_BRIDGE->SAHBASE = ((unsigned int) &pHSM2HOSTbuf[0])
            & 0xFFFF0000;
    pHSM2HOSTbuf = (unsigned int *) ((HSM_BRIDGE->HT2HSMS & 0x0000FFFF)
            | (unsigned int) &HSM_BRIDGE->SAHMEM[0]);
    for (i = 0; i < 0x100; i++)
        pHSM2HOSTbuf[i] = 0x00020000 | i;
    HSM_BRIDGE->HSM2HTS = 0xCCCC0000 | command;
    pHSM2HOSTbuf = (unsigned int *) HSM_BRIDGE->HT2HSMS;
    command = CMD_NOTHINGTODO;
    HSM_BRIDGE->HSM2HTF = 0x8; //Raise an Interrupt to the Host (Tricore)
    break;
}
```

Data Sharing between HSM and TriCore

- Helper Functions for HSM non-cached read and write access to Host Address Room

```
void
hostwrite32_uncached (unsigned int addr, unsigned int value)
{
  unsigned int *pmem;
  HSM_BRIDGE->SAHBASE = addr & 0xFFFF0000;
  pmem = (unsigned int *) ((addr & 0x0000FFFF) | (unsigned int) &HSM_BRIDGE->SAHMEM[0]);
  pmem[0] = value;
}

//helper function to make uncached access to Host address room
unsigned int
hostread32_uncached (unsigned int addr)
{
  unsigned int *pmem;
  HSM_BRIDGE->SAHBASE = addr & 0xFFFF0000;
  pmem = (unsigned int *) ((addr & 0x0000FFFF) | (unsigned int) &HSM_BRIDGE->SAHMEM[0]);
  return (pmem[0]);
}
```

```
Hostaddress = [SAHBASE]+(HSMadress-SAHMEM)& 0xFFFF
```

Part of your life. Part of tomorrow.

Infineon