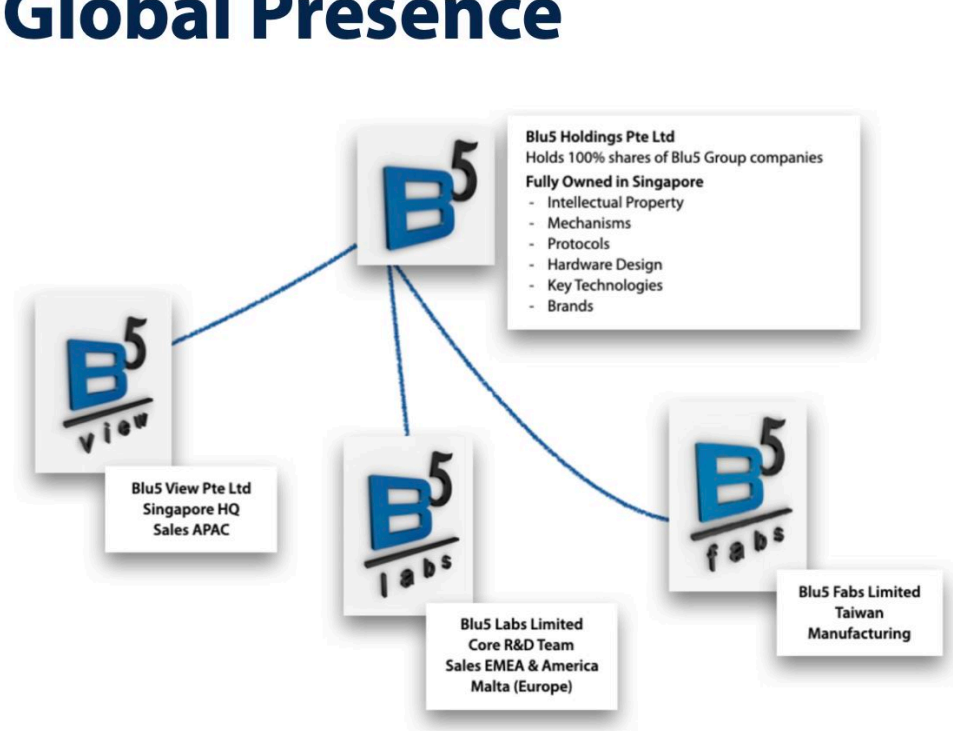
Hacking Hardware Devices

Antonio Varriale
Blu5 Group

Blu5 Group

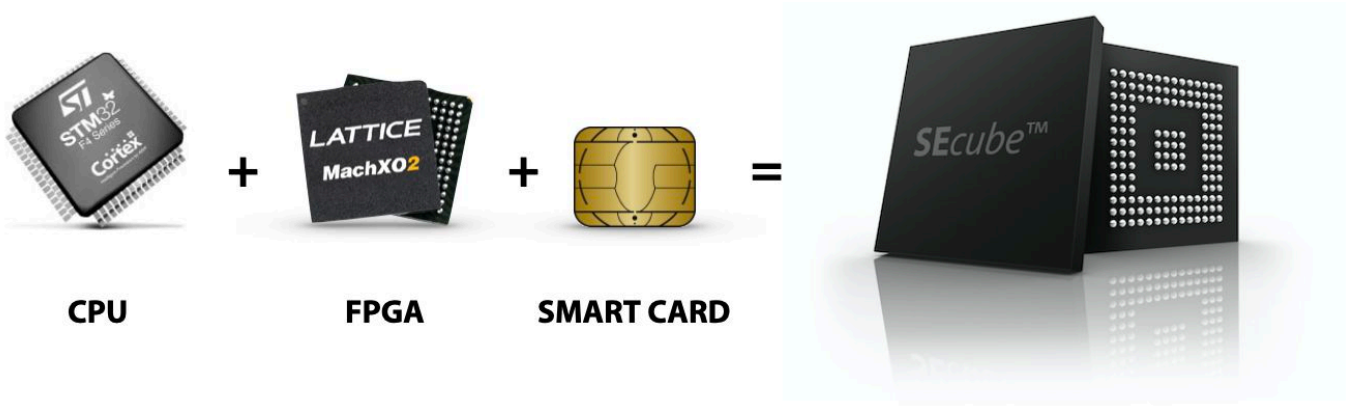
Global Presence

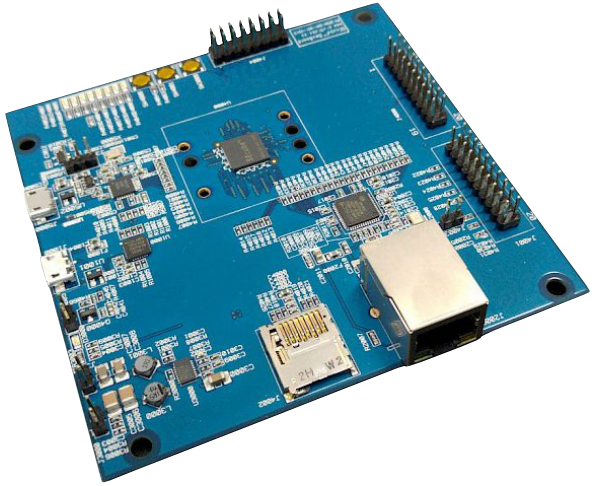


Blu5 Group solid international presence is based on three operational companies, **main office in Singapore**, **R&D facilities in Europe** and **manufacturing in Asia Pacific**.

Sales and Field Support network, managed by partners over 3 continents, is expanding to match the company growth.

The first Open Security Platform in a Single Chip





The first micro SD card with a Secure Environment onboard

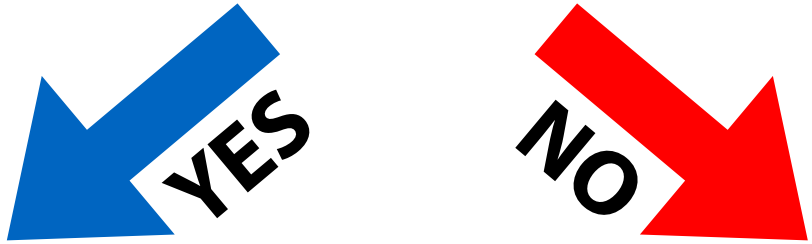
A myriad of micro devices are coming

But don't expect the same OS that you find on mobile devices, PCs and servers.

- ✓ Code once, run on 8-bit (e.g., Arduino Mega 2560), 16-bit (e.g., MSP430), and 32-bit platforms
- ✓ Robust runtime system
- ✓ Modular for flexible code-footprint
- ✓ Fosters energy-efficiency
- ✓ Real-time capable by limiting interrupt latency (~50 clock cycles) and priority-based scheduling
- ✓ Multi-threading with ultra-low overhead (<25 bytes per thread)

- ✓ 6LoWPAN, IPv6, RPL, UDP, TCP, QUIC
- ✓ MQTT-SN, CoAP, and CBOR
- ✓ BLE, LoRaWAN, 802.15.4, WLAN, CAN
- ✓ LwM2M client integration
- ✓ Static and dynamic memory allocation
- ✓ High resolution and long-term timers
- ✓ Tools and utilities (System shell, Crypto primitives, ...)
- ✓ Automated testing on various embedded hardware in the loop

Is you hardware target using Secure Programming Strategies?



Ready for DEFENCE

Ready for ATTACK

Hardware Features & Tools

Most Wanted Features for Performance/Security

- ✓ Full integration on single die (core, flash, sram, EEPROM)
- ✓ High Performances
- ✓ Floating Point
- ✓ DSP Libraries
- ✓ Low power consumption
- ✓ Dynamic Power scaling (100uA/MHz)
- ✓ HW Crypto Accelerators
- ✓ True Random Noise Generator
- ✓ Debug Lock
- ✓ Integrity and Tamper Protection
- ✓ Memory Protection Unit
- ✓ In-line Firmware Update (memory segmentation, flash technology)

Focus on Security Requirements

✓ Protection from Software Attacks

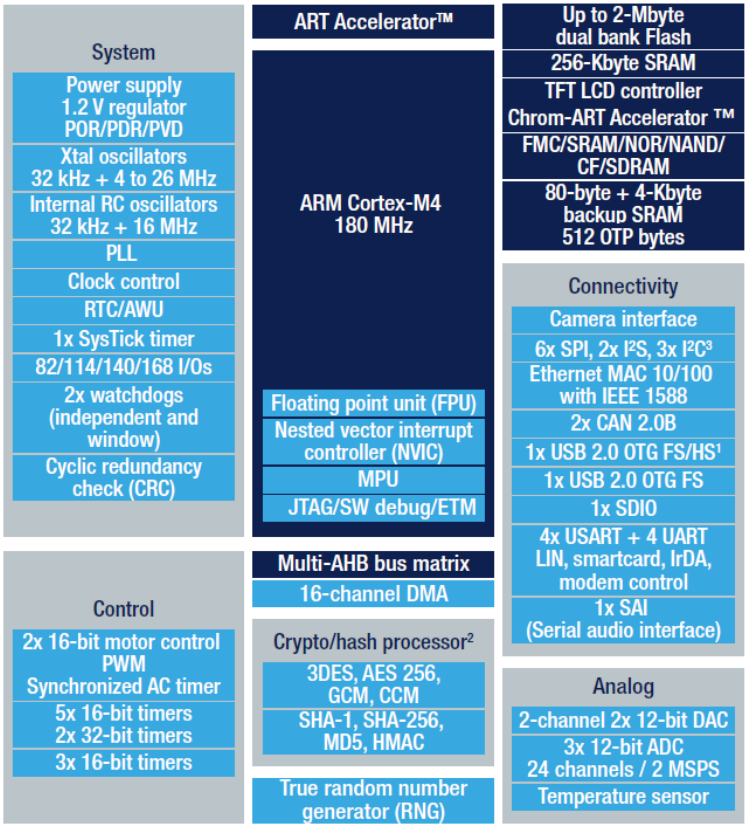
- Firmware Corruption (Buffer overflows, stack corruptions, …)
- Untrusted firmware update
- Debug activation
- Wrong execution, denial of service

✓ Protection from Hardware Attacks

- Fault injection by physical attacks on the system, external to package
  - Out of range usage
  - Glitch on supplies or clocks
  - Radiation exposure
  - Side channel attacks: spy product to get secrets (power supply, electromagnetic radiations, …)
  - Internal fault injection after decapsulation (Force nodes by probing, laser beam, …)
  - Reverse engineering (code/data extraction)
  - Circuit modification (fib: focused ion beam, …)

Example: STM32F4x Main Technical Specs

- Core: ARM 32-bit CortexTM-M4 CPU with FPU
- Adaptive real-time accelerator (ART AcceleratorTM) allowing 0-wait state execution from Flash memory
- frequency up to 168 MHz, memory protection unit, 210 DMIPS/ 1.25 DMIPS/MHz (Dhrystone 2.1), and DSP instructions
- Memories
  - Up to 1 Mbyte of Flash memory
  - Up to 192+4 Kbytes of SRAM including 64-Kbyte of CCM (core coupled memory) dataRAM
- Clock, reset and supply management
- 4-to-26 MHz crystal oscillator
  - Internal 16 MHz factory-trimmed RC (1% accuracy)
  - Internal 32 kHz RC with calibration
- Low power
  - Sleep, Stop and Standby modes
- Debug mode
  - Serial wire debug (SWD) & JTAG interfaces
  - Cortex-M4 Embedded Trace MacrocellTM • Up to 140 I/O ports with interrupt capability
- Up to 138 5 V-tolerant I/Os
- Up to 15 communication interfaces
- Up to 4 USARTs/2 UARTs (10.5 Mbit/s, ISO
- 8- to 14-bit parallel camera interface up to 54 Mbytes/s
- True random number generator



Notes:
1. HS requires an external PHY connected to the ULPI interface
2. Crypto/hash processor on STM32F415, STM32F417, STM32F437 and STM32F439
3. With digital filter feature

CMSIS-DSP

http://www.keil.com/pack/doc/CMSIS/DSP/html/index.html

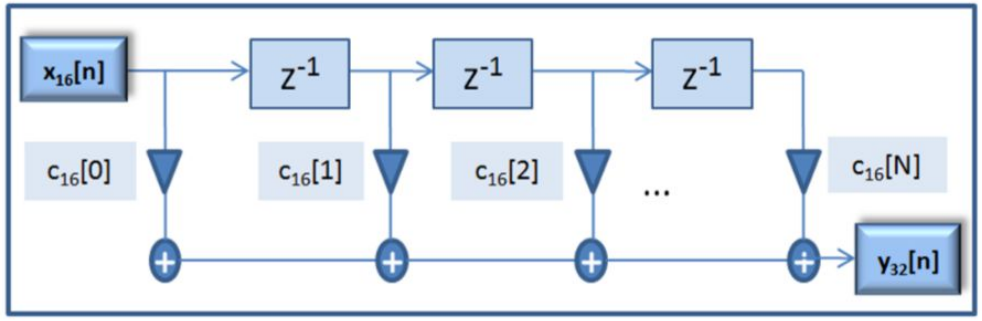
Cortex-M4 CMSIS-DSP

		Cycle counts		
CLASS	INSTRUCTION	ARM9E-S	CORTEX-M3	Cortex-M4
Arithmetic	ALU operation (not PC)	1 - 2	1	1
	ALU operation to PC	3 - 4	3	3
	CLZ	1	1	1
	QADD, QDADD, QSUB, QDSUB	1 - 2	n/a	1
	QADD8, QADD16, QSUB8, QSUB16	n/a	n/a	1
	QDADD, QDSUB	n/a	n/a	1
	QASX, QSAX, SASX, SSAX	n/a	n/a	1
	SHASX, SHSAX, UHASX, UHSAX	n/a	n/a	1
	SADD8, SADD16, SSUB8, SSUB16	n/a	n/a	1
	SHADD8, SHADD16, SHSUB8, SHSUB16	n/a	n/a	1
	UQADD8, UQADD16, UQSUB8, UQSUB16	n/a	n/a	1
	UHADD8, UHADD16, UHSUB8, UHSUB16	n/a	n/a	1
	UADD8, UADD16, USUB8, USUB16	n/a	n/a	1
	UQASX, UQSAX, USAX, UASX	n/a	n/a	1
	UXTAB, UXTAB16, UXTAH	n/a	n/a	1
	USAD8, USADA8	n/a	n/a	1
Multiplication	MUL, MLA	2 - 3	1 - 2	1
	MULS, MLAS	4	1 - 2	1
	SMULL, UMULL, SMLAL, UMLAL	3 - 4	5 - 7	1
	SMULBB, SMULBT, SMULTB, SMULTT	1 - 2	n/a	1
	SMLABB, SMLBT, SMLATB, SMLATT	1 - 2	n/a	1
	SMULWB, SMULWT, SMLAWB, SMLAWT	1 - 2	n/a	1
	SMLALBB, SMLALBT, SMLALTB, SMLALTT	2 - 3	n/a	1
	SMLAD, SMLADX, SMLALD, SMLALDX	n/a	n/a	1
	SMLSD, SMLSDX	n/a	n/a	1
	SMLSLD, SMLSLD	n/a	n/a	1
	SMMLA, SMMLAR, SMMLS, SMMLSR	n/a	n/a	1
	SMMUL, SMMULR	n/a	n/a	1
	SMUAD, SMUADX, SMUSD, SMUSDX	n/a	n/a	1
	UMAAL	n/a	n/a	1
Division	SDIV, UDIV	n/a	2 - 12	2 - 12

Single cycle MAC

Code Example: FIR

$$y_{32}[n] = \sum_{i=1}^{N} x_{16}[n-1] * c_{16}[i]$$



Cortex-M3 Code Segment:

```
FIR_LOOP:
LDR     R2,[R0],#4    ;(2) Load input x16
LDR     R3,[R1],#4    ;(2) Load coeff c16
SXTH    R4, R2        ;(1) Extract x16[n-i]
ASR     R2, R2,#16    ;(1) Extract x16[n-i-1]
SXTH    R5, R3        ;(1) Extract c16[i]
ASR     R3, R3,#16    ;(1) Extract c16[i+1]
MLA     R6, R4, R5    ;(2) y32 += x16[n-i]*c16[i]
MLA     R6, R2, R3    ;(2) y32 += x16[n-i-1]*c16[i+1]
SUBS    R7, R7, #2    ;(1) loop count -= 2
BNE     FIR_LOOP      ;(2)

Note:
1. In these examples, FIR_LOOP is unrolled by 2
2. This example assumes number of taps is even.
```

Cortex-M4 Code Segment:

```
FIR_LOOP:
LDR     R2,[R0],#4 ;(1) Load input x16[n-i],x16[n-i-1]
LDR     R3,[R1],#4 ;(2) Load coeff c16[i], c16[i+1]
SUBS    R5, R5, #2 ;(1) loop count -= 2
SMLAD   R4, R2, R3 ;(1) y32 +=x16[n-i,n-i-1]*C16[i,i+1]
BNE     FIR_LOOP   ;(2)
```

Processor	Kernel cycles	Total Cycles	Number of Instructions	Register usage
Cortex-M3	8	15	10	7
Cortex-M4	1	7	5	5
Advantage	**8x**	**~2.2x**	**2x**	**1.4x**

Low Power Modes

The ST32F4xx features 3 low power modes

- SLEEP, core stopped, peripheral running (**2mA @ 2 MHz, 38mA @ 120MHz**)
- STOP, clocks stopped, RAM, registers kept (**1mA**)
- STANBY, only backup domain kept, return via RESET (**150uA**)

Low power mode	Conditions	Wakeup time in µs
Sleep mode		1 Typ
Stop mode	regulator in Run mode	13 Typ
Stop mode	regulator in low power mode	17 Typ
Stop mode	regulator in low power mode and Flash in Deep power down mode	110 Typ
Standby mode		375 Typ

Privilege Levels: Privileged & Unprivileged

✓ **Privileged/System Level**

- The software can use all the instructions and has access to all resources
- Privileged software executes at the privileged level

✓ **Unprivileged/User Level**

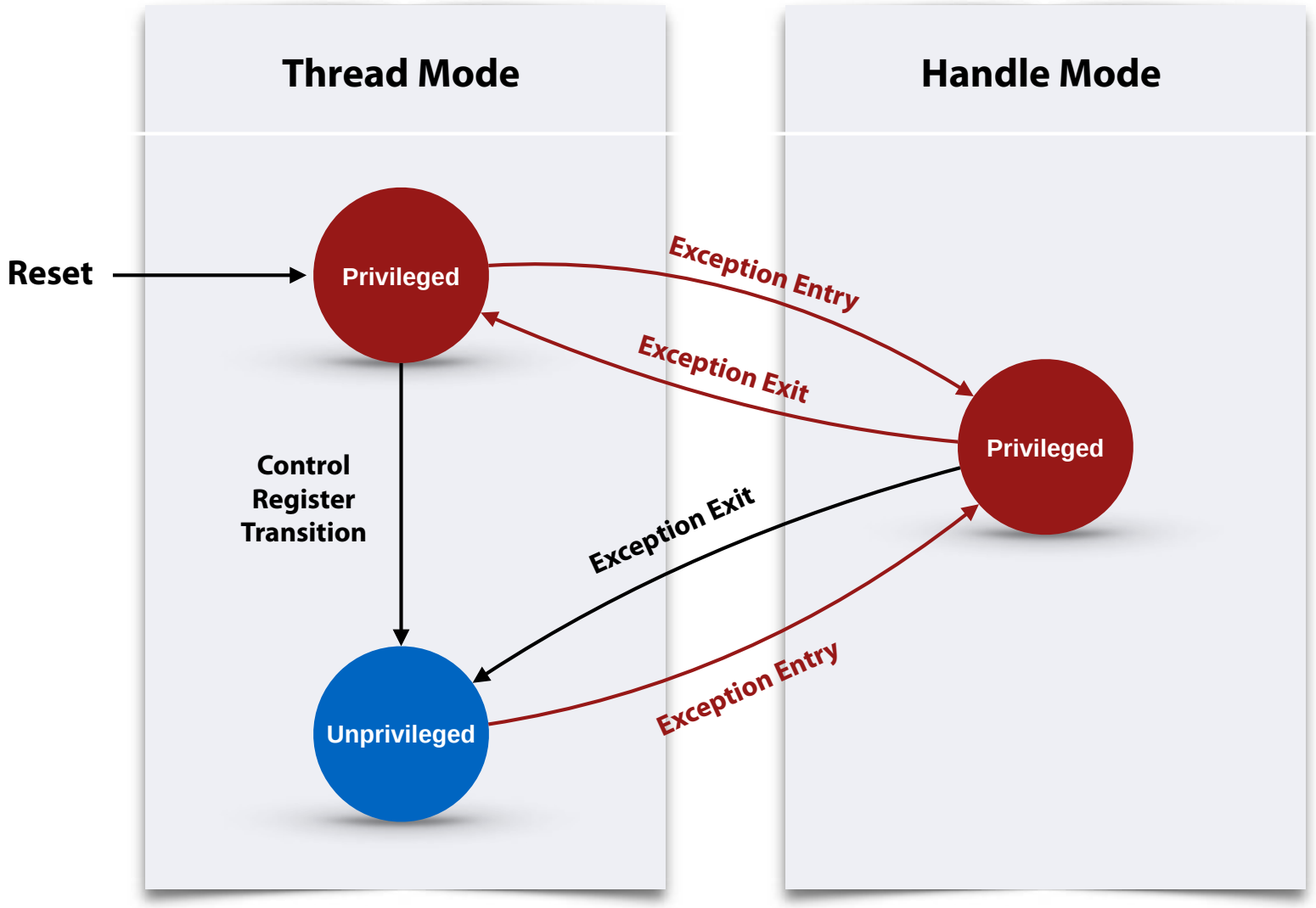
- The software has limited access to the MSR and MRS instructions (copy immediate values from/to Processor Status Register) and cannot use the CPS (Change Processor Status) instruction
- The software cannot access the system timer, NVIC or system control block
- might have restricted access to memory or peripherals (Through MPU)

Modes: Threads & Interrupts

- ✓ **Thread Mode**
  - Used to execute application Software
  - The processor enters Thread mode when it comes out of reset
  - Thread mode can be both Privileged and Unprivileged

- ✓ **Handle Mode**
  - Used to handle exceptions
  - The processor returns to Thread mode when it has finished exception processing
  - Handle mode is always Privileged

Execution State Diagram



Memory Protection Unit

The Memory Protection Unit (MPU) divides the memory map into a number of regions and defines the location, size, access permission and memory attributes of each region, supporting:

- Independent attribute settings for each region
- Overlapping regions
- Export of memory attributes to the system

The Memory attributes affect behaviour of memory access to the region

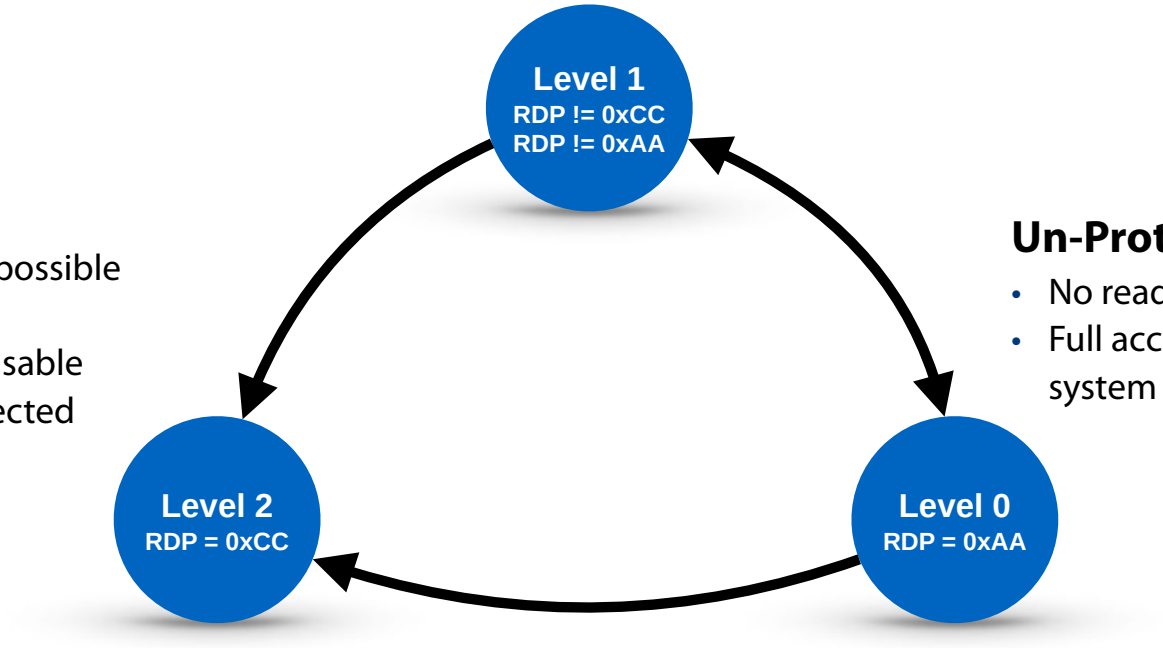
- Eight separate memory regions, 0-7
- background region (default)

The MPU register can be set in privileged level only

Readout/JTAG Protection

Readout protection

- BLOCKED access to memory from SRAM, system memory and JTAG
- Remove readout protection possible after full erase of the memory and its blank verification



JTAG Fuse

- No Un-Protecting possible
- JTAG disable
- System Memory disable
- User settings protected

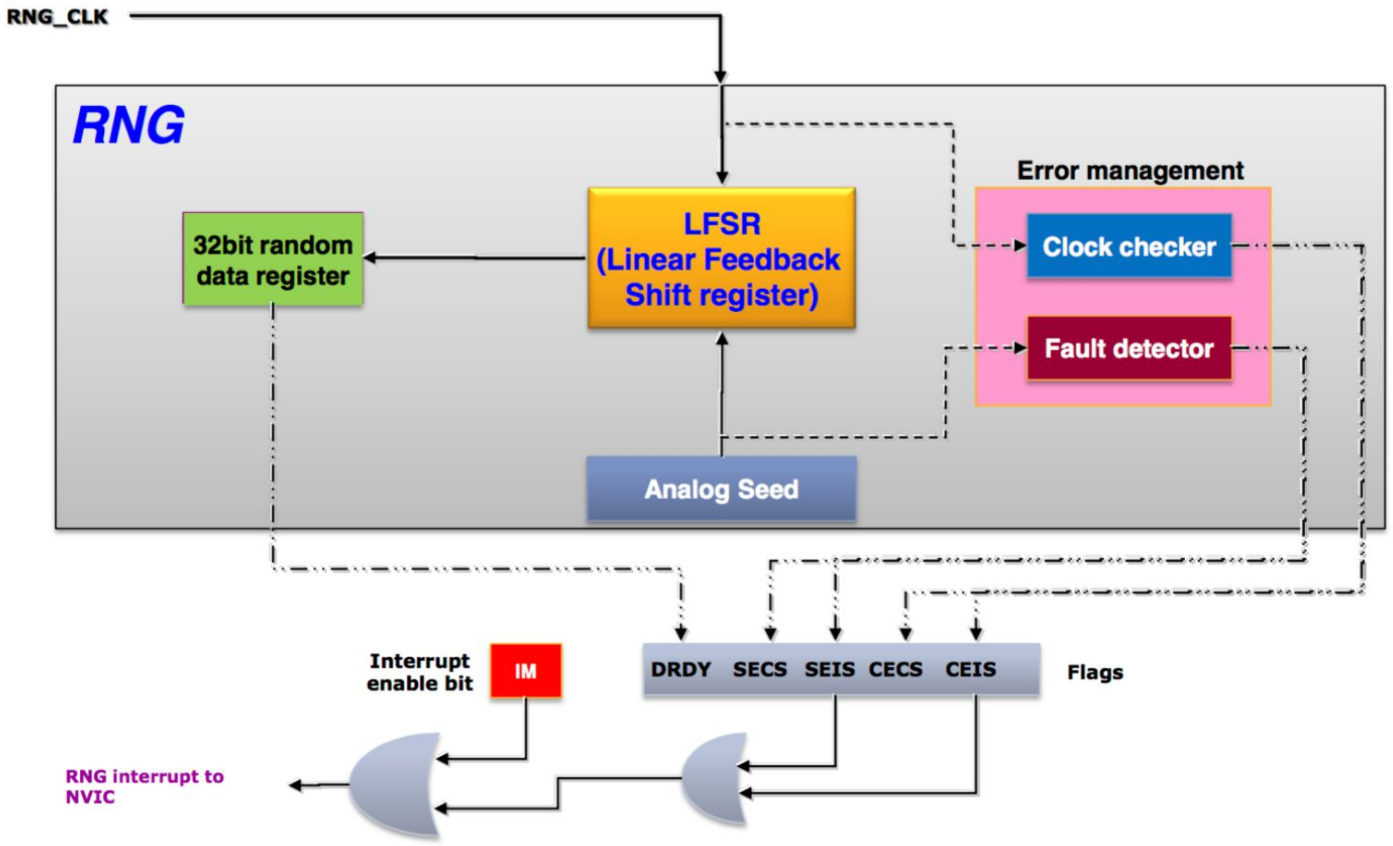
Un-Protected

- No readout protection
- Full access to memory from SRAM, system memory and JTAG

True Random Noise Generator Features

- ✓ **32-bit Random Numbers, produced by an analog generator Generator**
- ✓ **Clocked by a dedicated clock (PLL48CLK)**
- ✓ **40 periods of the PLL48CLK clock signal between two consecutive random numbers**
- ✓ **FIPS 140-2 compliant**
- ✓ **Hardware runtime check**
  - ✓ 5 flags (1 x valid data ready, 2 x abnormal sequence, 2 x frequency error)
  - ✓ 1 Interrupt (abnormal sequence or frequency error)

True Random Noise Generator Block Diagram



Integrity and Tamper Protection

Integrity & Safety

- ✓ CRC Calculation Unit
- ✓ Power Supply Integrity Monitoring
- ✓ Clock Security System
- ✓ Error Correction Code
- ✓ Parity Check
- ✓ Temperature Sensor

Tamper Protection

- ✓ Backup Domain
- ✓ RTC (alarm timestamp)
- ✓ RTC Register protection
- ✓ Backup Registers
- ✓ GPIO Configuration Locking
- ✓ Watchdogs

In-line firmware update

It is possible thanks to the permanent memory segmentation and to the use of modern flash technology instead of EEPROM

Flash Behaviour

Set 1 to 0
The single bit can be modified

Set 0 to 1
The whole sector must be modified

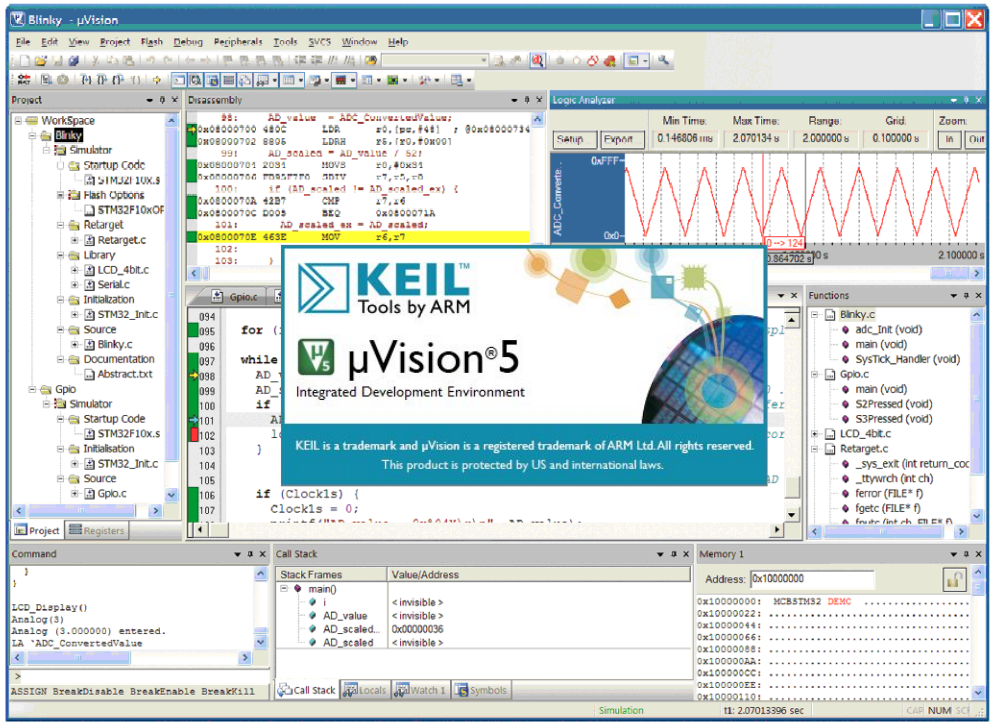
Example

Current Value:	1	0	1	1	0	1	0	0	0	1
Written Value:	0	0	0	1	1	1	0	1	1	1
Actual Value:	0	0	0	1	**0**	1	0	**0**	**0**	1

Development Tool

Keil™ Development Tool is an ARM® product which includes

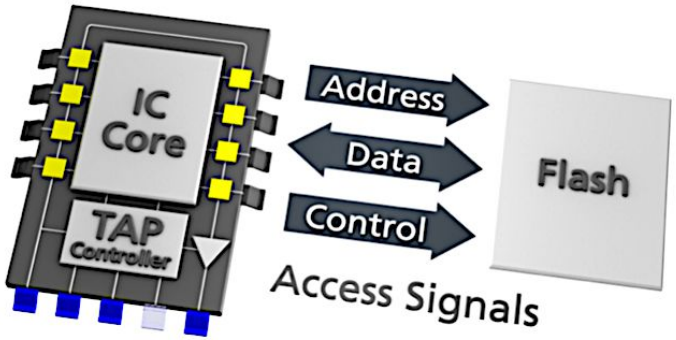
- C/C++ Compiler,
- Debugger,
- Integrated Environment,
- Simulation Models,
- Libraries,
- Debug and Trace adapters



Secure Programming Strategies

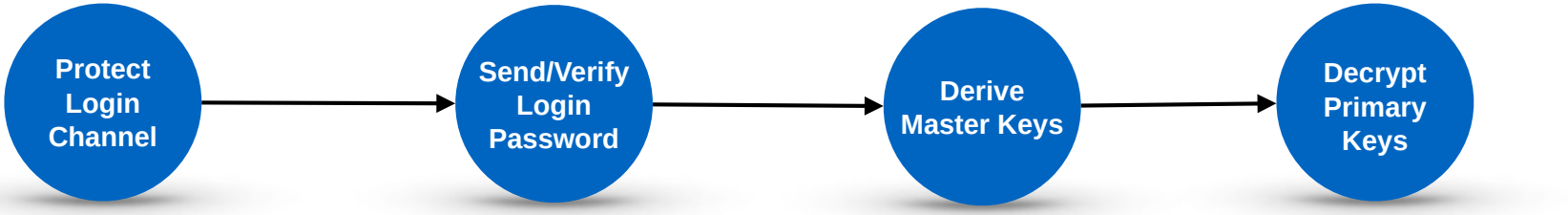
Secure Programming Strategies

- ✓ Chained Security
  - ✓ All Secrets encrypted by Primary Keys
  - ✓ Primary Keys Encrypted by Master Keys
  - ✓ Master Keys decrypted/derived by after successful login
- ✓ Multi-Layer Security (e.g. Factory/OEM/user, etc.)
  - ✓ Roles separation
  - ✓ Keys separation
- ✓ Memory Protection Unit
- ✓ Privileged/Unprivileged execution
- ✓ Software Interrupt Calls



Chained Security

- ✓ **No clear MASTER KEYS stored in the CPU FLASH**
  - ✓ Even if it is not accessible via JTAG!
- ✓ **No clear Login Password stored in the CPU FLASH**
  - ✓ Even if it is not accessible via JTAG!
- ✓ **Secrets available <u>only in MPU protected RAM areas after successful login</u>**



- **At least a Secondary pwd must be used**
- **Secondary Pwd must be updated**
- **Secondary Pwd is expendable**
- **Channel protected (e.g. with ECDH + Secondary Password)**

- **Login Password must be updated**
- **Login Password is not sent in clear on the channel**
- **Login Password is not stored in the CPU Flash**

- **Use Login Password and Secondary Password to Derive Master Keys**
- **Master Keys are not stored in the CPU Flash**

- **Master Keys are used to decrypt the Primary Keys**
- **Primary Keys are not stored in clear in the CPU Flash**

Protection "at rest"

Multi-Layer Security

In order to separate responsibilities and roles, it is strongly suggested to implement software compartments (or layers) owned by specific entities

For example, the following layers can be implemented to manage 3 entities/ ownerships:

1. Boot, owned by the FACTORY (HW maker)
2. Main Application, owned by the OEM (Integrator)
3. User Functions, owned by the USER (Final Customer)

Each layer must be protected by (at least) one secret (key) to operate on it (e.g. delete, update, etc.)

MPU and Privileged Modes

Region Number	Address	Size	Permission	Attributes
1	B5_POLICIES_ADDRESS	B5_MPU_REGION_SIZE_4KB	B5_MPU_PRI_RW_USR_RO	B5_MPU_ATTR_CACHE | B5_MPU_ATTR_SHARE
2	B5_KEYS_RAM_ADDRESS	B5_MPU_REGION_SIZE_128KB	B5_MPU_PRI_RW_USR_NOACCESS	B5_MPU_ATTR_CACHE | B5_MPU_ATTR_SHARE
3	...	...	...	...
4	...	...	...	...
5	...	...	...	...
6	...	...	...	...
7	...	...	...	...
8	...	...	...	...

Default regions are RW for Privileged and NO_ACCESS for Unprivileged

Interrupt Communication

There are 2 main ways to call subroutines

1. Standard Calls
2. Interrupt Calls (software interrupts)

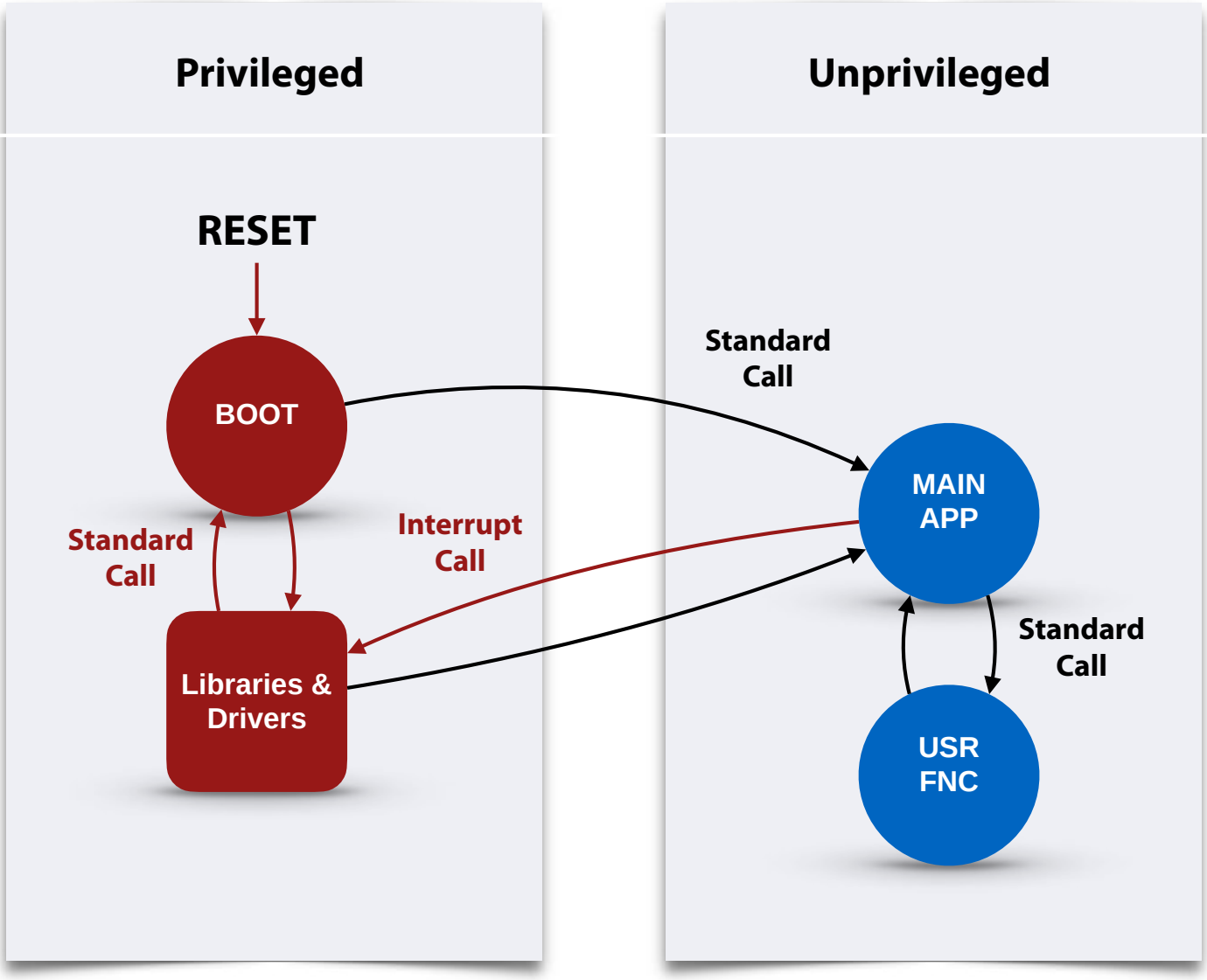
Standard Calls

- Execution Privilege **UNCHANGED**
- Usually Thread Mode to Thread Mode (subroutine which calls another subroutine)
- Sometimes Handle Mode to Handle Mode (interrupt which calls a subroutines)
- Implemented through a BL/BX ARM instruction

Interrupt Calls

- Execution Privilege temporarily changes to **PRIVILEGED**
- From Thread Mode to Handle Mode (subroutine which calls a software interrupt)
- When in Handle Mode, interrupt calls cannot be executed since interrupts are disabled under interrupt routines execution
- Implemented through a software interrupt call

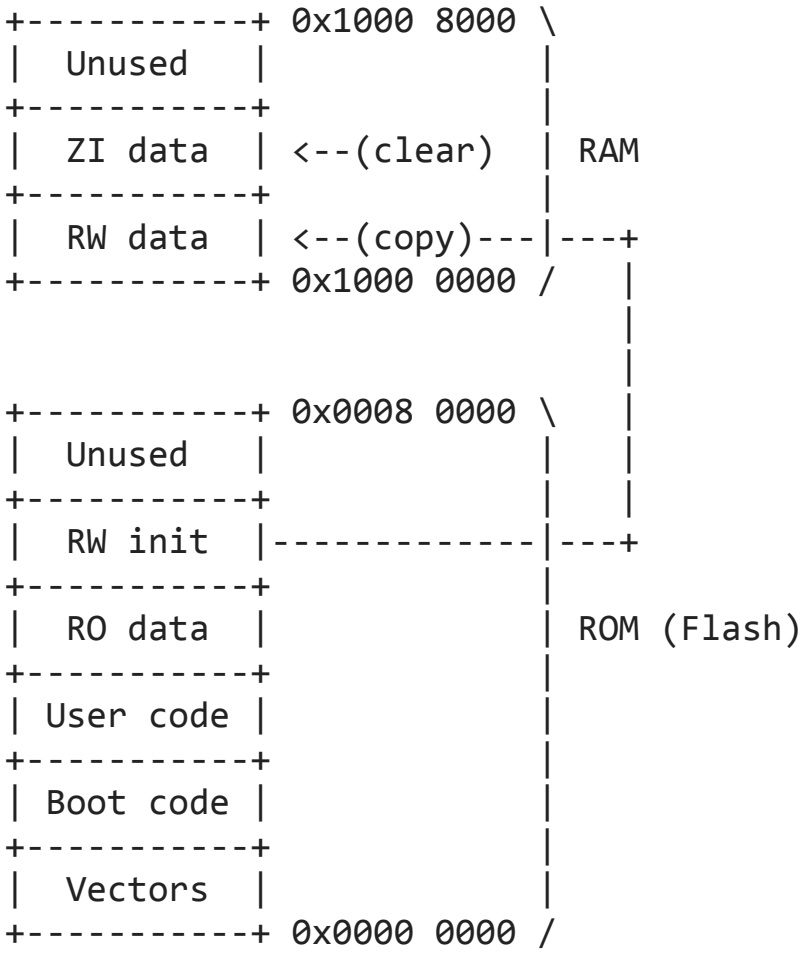
Usual Execution Flow



Advanced Linker Usage

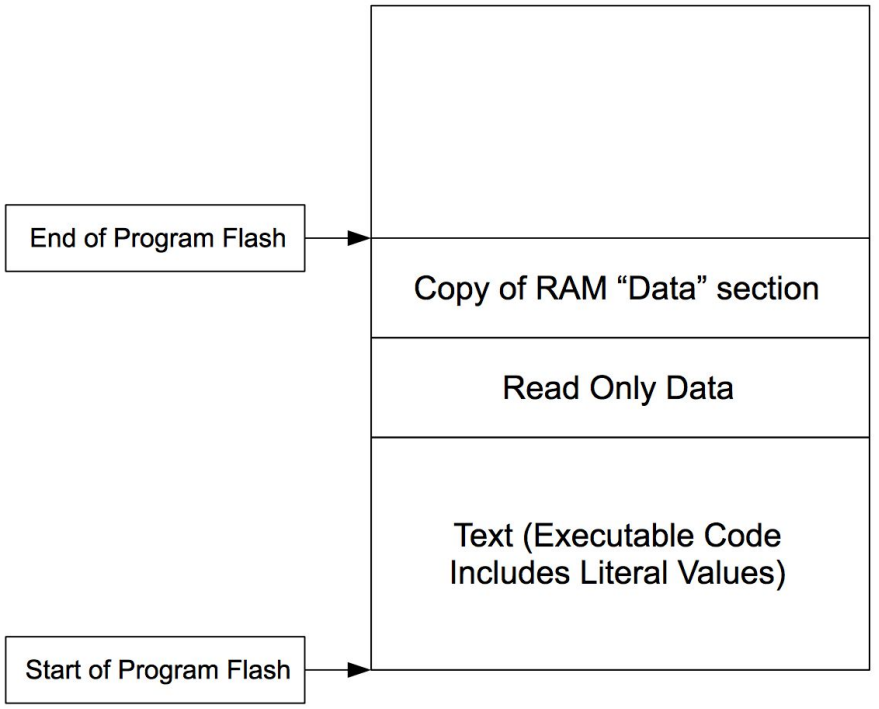
Program Image and Execution

- ✓ **RO (or TEXT), Read Only**
  - RO-CODE (or CODE)
  - RO-DATA (or CONST)
- ✓ **RW, Read/Write**
  - RW-CODE
  - RW-DATA
- ✓ **ZI (or BSS), Zero Initialised**



Program Flash

```
const int read_only_variable = 2000;
int data_variable = 500;

void my_function(void){
     int x;
     x = 200;
}
```

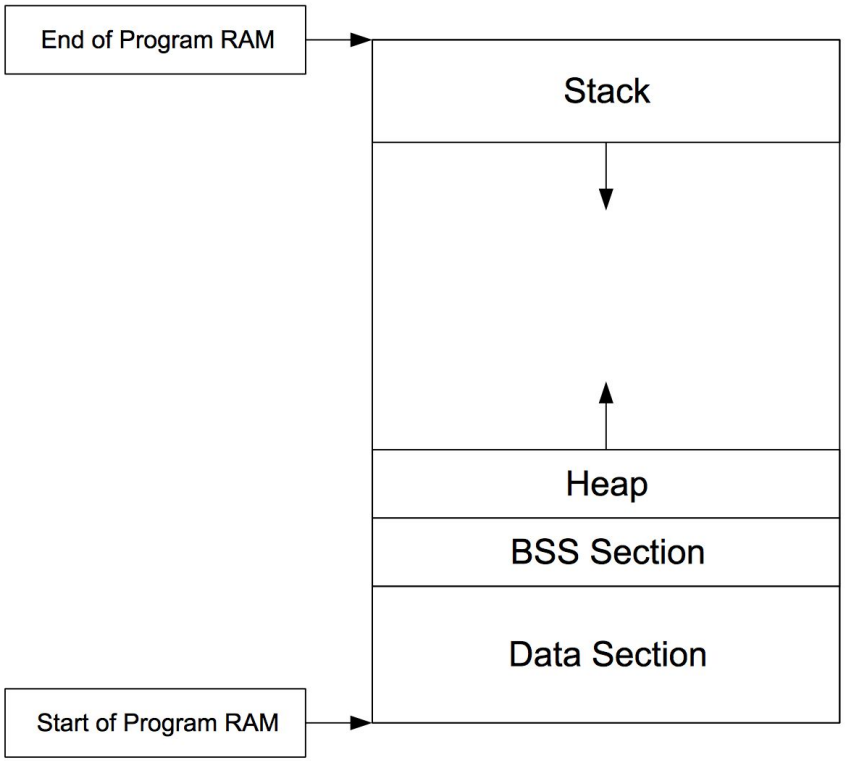


- **read_only_variable** is stored in the Read Only Data and set to 2000
- **data_variable** is stored in the "Copy of RAM Data" section (RW init) and set to 500. It will be copied to the RAM Data section (RW Data) when the program starts (double space)
- **x** value is stored in the literal pool. When my_function is called **x** is allocated in the stack and set to its value stored in the literal pool

Program RAM - Static Allocation

```
int data_var = 500;
int bss_var0;
int bss_var1 = 0;

void my_function(void){
     int uninitialized_var;
}
```

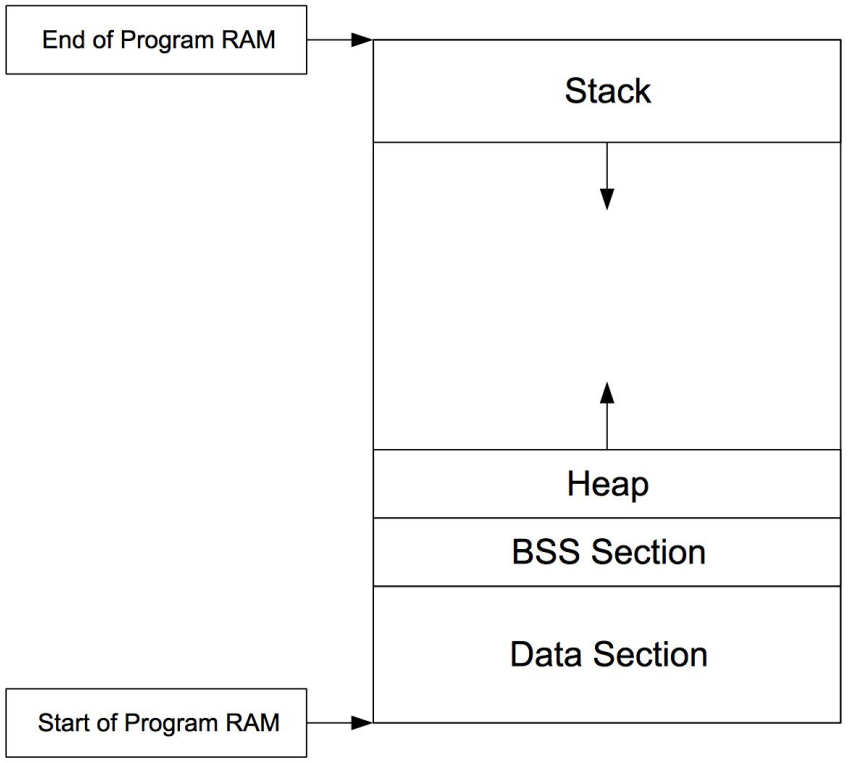
- **data_var** is stored in the Data Section and is set to 500
- **bss_var0** and **bss_var1** are stored in the BSS section (ZI Data) (no double space)
- When my_function is called **uninitialized_var** is allocated on the stack



Program RAM - Dynamic Allocation

```
void my_func(void){
     char * buffer;
     int  uninitialized_var = 3;

     buffer = malloc(512);
     if ( buffer == NULL )
          return;
     memset(buffer, 0, 512);
     free(buffer);
}
```

- **buffer** is allocated on the heap
- When my_function is called **uninitialized_var** is allocated on the stack and set to its value stored in the literal pool



Main Stack and Process Stack

- The processor uses a full descending stack (SP holds the address of the last stacked item)
- New stack items decrements the Stack Pointer
- The processor implements two stacks: **Main Stack** and **Process Stack**

Processor Mode	Privilege Level	Stack Used
Thread	**Privileged or Unprivileged**	**Main Stack or Process Stack**
Handle	**Privileged**	**Main Stack**

Scatter Files

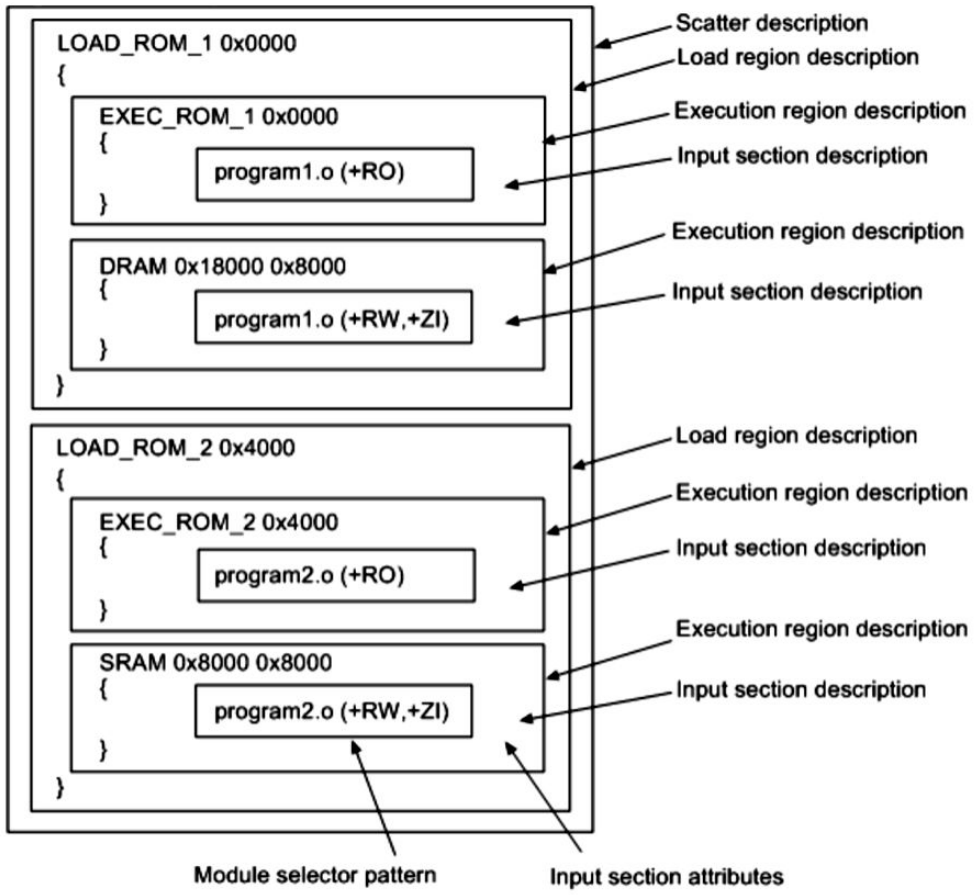
Scatter-loading is usually required for implementing embedded systems because these use ROM, RAM and memory-mapped peripherals

Situations where scatter-loading is either required or very useful

- **Complex memory maps**
  Code and data that must be placed into many distinct areas of memory require detailed instructions on where to place the sections in the memory space.
- **Different types of memory**
  Many systems contain a variety of physical memory devices such as flash, ROM, SDRAM, and fast SRAM. A scatter-loading description can match the code and data with the most appropriate type of memory. For example, interrupt code might be placed into fast SRAM to improve interrupt response time but infrequently-used configuration information might be placed into slower flash memory.
- **Memory-mapped peripherals**
  The scatter-loading description can place a data section at a precise address in the memory map so that memory mapped peripherals can be accessed.
- **Functions at a constant location**
  A function can be placed at the same location in memory even though the surrounding application has been modified and recompiled. This is useful for jump table implementation.

Scatter File Structure

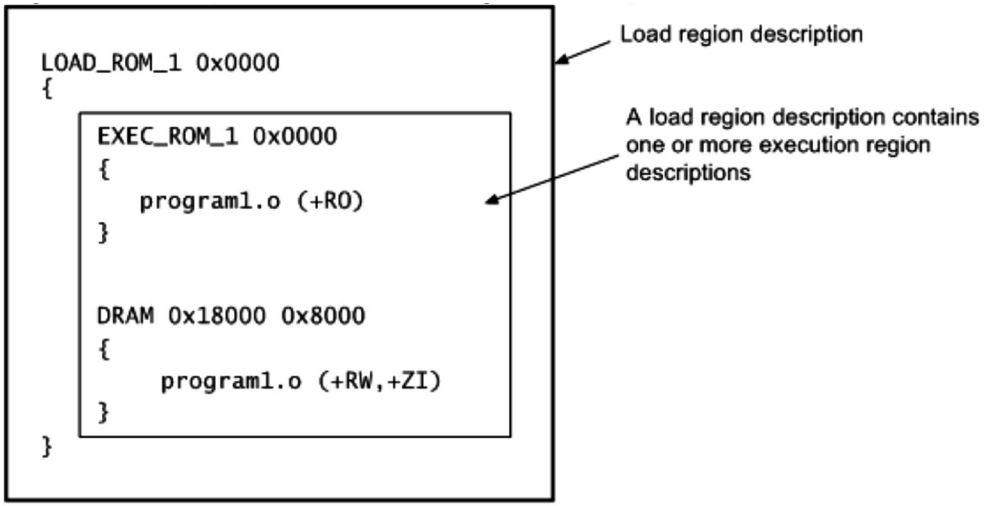
- A scatter file contains one or more **load regions**
- Each load region contains one or more **execution regions**



Load Region

A **load region** description has the following components:

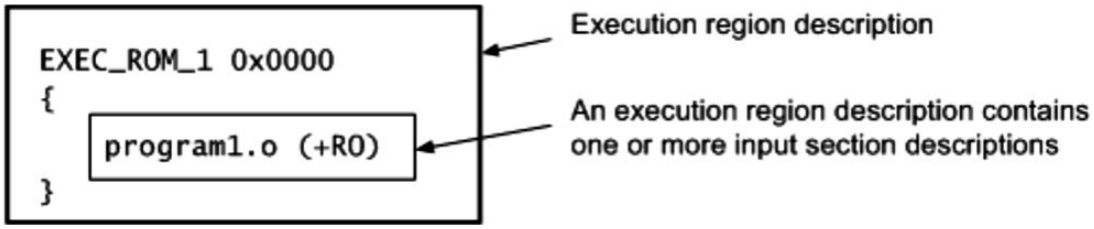
- A name (used by the linker to identify different load regions)
- A base address (the start address for the code and data in the load view)
- Attributes that specify the properties of the load region
- An optional maximum size specification
- One or more execute regions



Execution Region

An **execution region** description has the following components:

- A name (used by the linker to identify different execution regions)
- A base address (either absolute or relative)
- Attributes that specify the properties of the execution region
- An optional maximum size specification
- One or more input section descriptions (the modules placed into this execution region)

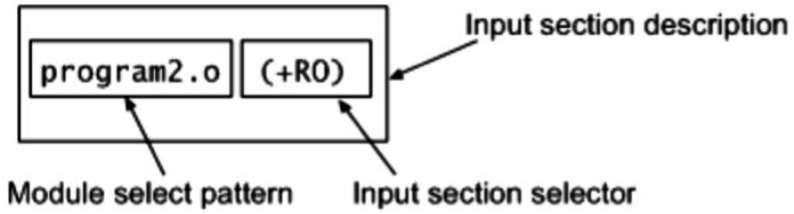


Input Section

An **input section** description identifies input sections by:

- Module name (object filename, library member name or library filename)
- Input section name or input section attributes such as READ-ONLY or CODE.
- Symbol name

Wildcard characters can be used



Static Calls through function pointers

- **Define a function pointer**
- **Set function pointer to a proper Code Entry Point**

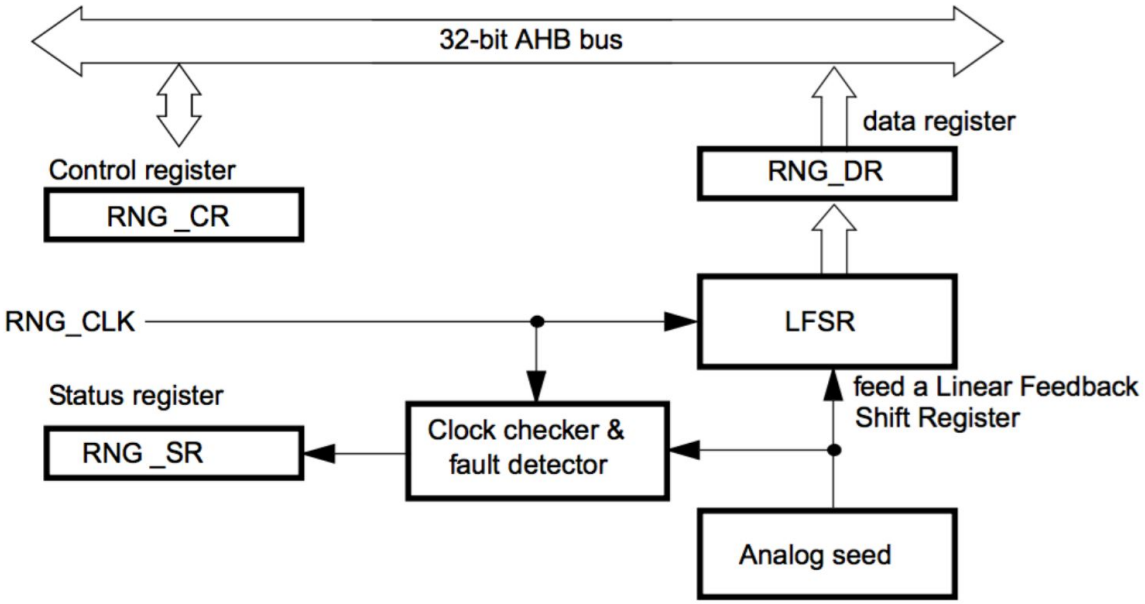
```
typedef int (* tMyAppFunction) ( unsigned char *inputData,
                                 unsigned int   inputDataLen,
                                 unsigned char *outputData,
                                 unsigned int   outputDataLen);

static tMyAppFunction           myAppFunction = (tMyAppFunction) (0x08004000 + 1);
```

```
FLASH_SECTOR_1_2_3 0x08004000 (0xC000)  {
    FLASH_SECTOR_1_2_3 0x08004000 (0xC000)
    {
        *.o (APP_Entry,+FIRST)
        .ANY
    }

    RAM_APPLICATION (0x20000000 + 0x8000) (0x8000)
    {
        .ANY (+RW +ZI)
    }

}
```

True Random Noise Generator

TRNG Block Diagram

- Based on an analog circuit which generates continuous analog noise that feed a Linear Feedback Shift Register (LFSR) in order to produce a 32-bit random number.
- The analog circuit is made of several ring oscillators whose outputs are XORed.
- The LFSR is clocked by a dedicated clock (PLL48CLK) at a constant frequency, so that the quality of the random number is independent of the HCLK frequency.
- The contents of LFSR is transferred into the data register (RNG_DR) when a significant number of seeds have been introduced into LFSR.



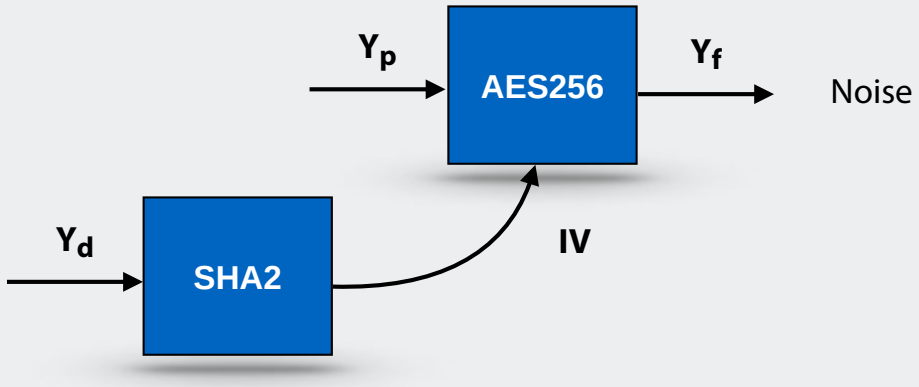
TRNG Customisation Path

- 32-bit random numbers, generated by the analog circuitry, seed the LFSR
- The Polynomial can be changed to customise the noise behaviour
- The Filtering block applies mathematical functions tu whiten the noise

Filtering Function

Most wanted properties

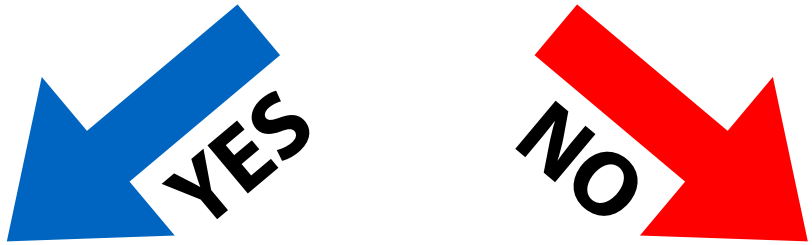
- Increase Entropy
- Generate white-like spectrum
- Fit statistical tests



Encryption algorithms usually fit the requirements above

Conclusion

Is you hardware target using Secure Programming Strategies?



Ready for DEFENCE

Ready for ATTACK

www.blu5group.com