Introduction to ARM Architecture

ARM ➔ It was named the Advanced RISC Machine, and before that, the Acorn RISC Machine
➔ Companies: Acorn - Apple - VLSI

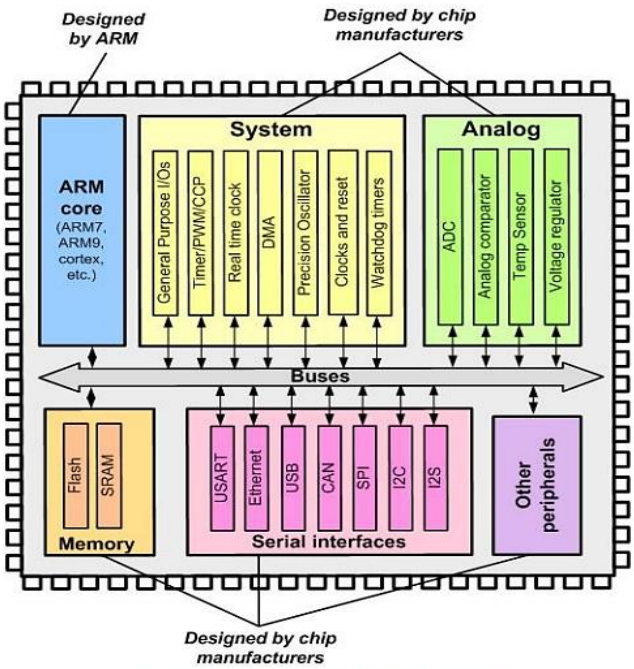


Figure 1- 4: ARM Simplified Block Diagram

→ ARM is a 32-bit reduced instruction set computer (RISC) instruction set architecture (ISA) developed by ARM Holdings

→ The ARM core processor is the central processing unit (CPU) of a microcontroller chip.

→ ARM holds the copyright for the architecture, registers, instruction set, memory map, and timing of the ARM CPU.

→ Design houses and semiconductor manufacturers obtain a license for the CPU's intellectual property (IP) from ARM. This license allows them to design and manufacture their own ARM-based chips.

→ While the CPU instructions and architecture are the same across ARM chips from different vendors, their peripherals may vary.

→ Due to differences in peripherals, a program written for a specific peripheral on an ARM chip from one vendor may not run on an ARM chip from another vendor.

→ Note that architecture version numbers are independent from processor names.
  → For example, the ARM7TDMI processor is based on the ARMv4T architecture

→ the ARM7TDMI supports two instruction sets:
  1. ARM instruction set with 32-bit instructions (Size of instruction is 4 Bytes).
  2. Thumb instruction set with 16-bit instructions (Size of instruction is 2 Bytes).

→ This enables ARM processors to be used in many portable devices, which require low power and small memory.

→ As a result, ARM processors are the first choice for mobile devices like mobile phones.

→ ARM Cortex is a wide set of 32/64-bit architectures and cores really popular in the embedded world

→ Cortex microcontrollers are divided into three main subfamilies:
  1. **Cortex-A**, which stands for Application
  2. **Cortex-M**, which stands for eMbedded or Microcontroller
  3. **Cortex-R**, which stand for Real-Time

a) **Cortex-A** (Application Processors):

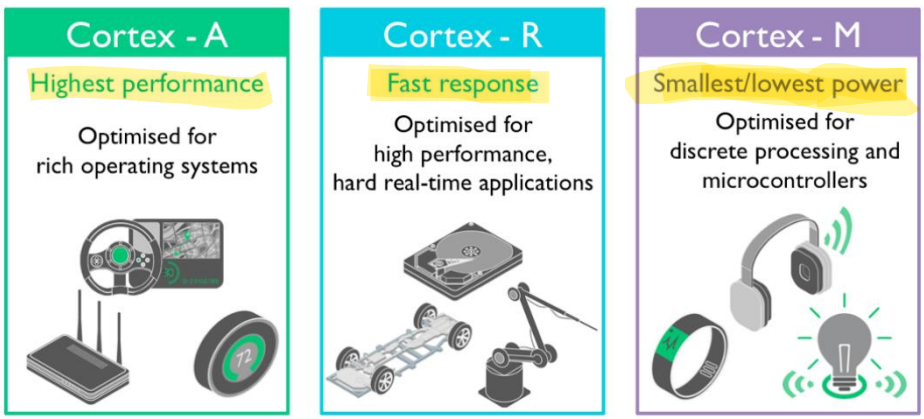
- These are Application processors, which are designed to provide high performance and include features to support advanced operation systems (e.g., Android, Linux, Windows, IOS).
- Cortex-A cores equip the processors found in most of mobile devices, like phones and tablets.
- These processors typically have longer processor pipeline and can run at relatively high clock frequency (e.g., over 1 GHz).
- While the Cortex-A processors have high performance, the processor is not designed to provide rapid response time to hardware events (i.e., real-time requirements).

b) **Cortex-R** (Real-Time Processors):

- These are Real-Time, high performance processors that are very good at data crunching, can run at fairly high clock speed (e.g., 500 MHz to 1 GHz range), and at the same time can be very responsive to hardware events.
- The Cortex-R processors target high-performance real-time applications such as hard disk controllers (or solid state drive controllers), networking equipment and printers in the enterprise segment, consumer devices such as Blu-ray players and media players
- The Cortex-R4, for example, is well suited for automotive applications. Such as airbags, braking systems and engine management.

c) **Cortex-M**, which stands for eMbedded or Microcontroller

- Unlike earlier ARM CPUs, the Cortex-M processor family has been designed specifically for use within a small microcontroller.
- A range of scalable, compatible, energy efficient and easy to use processors designed for the low-cost embedded market.
- The Cortex-M family is optimized for cost and power sensitive MCUs suitable for applications such as Internet of Things, motor control, smart metering, human interface devices, automotive and medical instruments.
- All STM32 microcontrollers have a Cortex-M core, plus some distinctive ST features (like the ARTTM accelerator)
- Internally, each microcontroller consists of the processor core, static RAM, flash memory, debugging interface, and various other peripherals.
- Some MCUs provide additional types of memory (EEPROM, CCM, etc.), and a whole line of devices targeting low-power applications is continuously growing



→ Advantages of the STM32 Portfolio

- ✓ They are Cortex-M based MCUs.
- ✓ Free ARM based tool-chain.
- ✓ Integrated Bootloader.
- ✓ STM32F is the right choice if you want to migrate from 8/16-bit MCUs.
- ✓ Most STM32 pins are 5V tolerant.

→ **STM32® THE LEADING CORTEX-M PORTFOLIO**

→ The diagram aggregates the subfamilies in three macro groups:

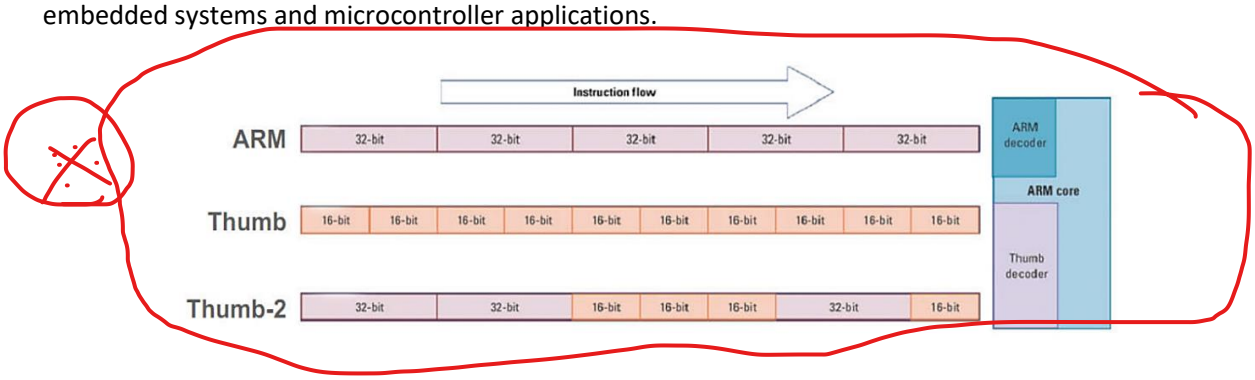
1. High-performance
2. Mainstream
3. Ultra Low-Power MCUs

ARM Processor Cortex-M3 / Cortex-M4

- The Cortex-M3 processor was the first of the Cortex generation of processors, released by ARM in 2005.
- The Cortex-M4 processor was released in 2010
- The Cortex-M3 and Cortex-M4 processors use a **32-bit** architecture. Internal registers in the register bank, the data path, and the bus interfaces are all 32 bits wide.

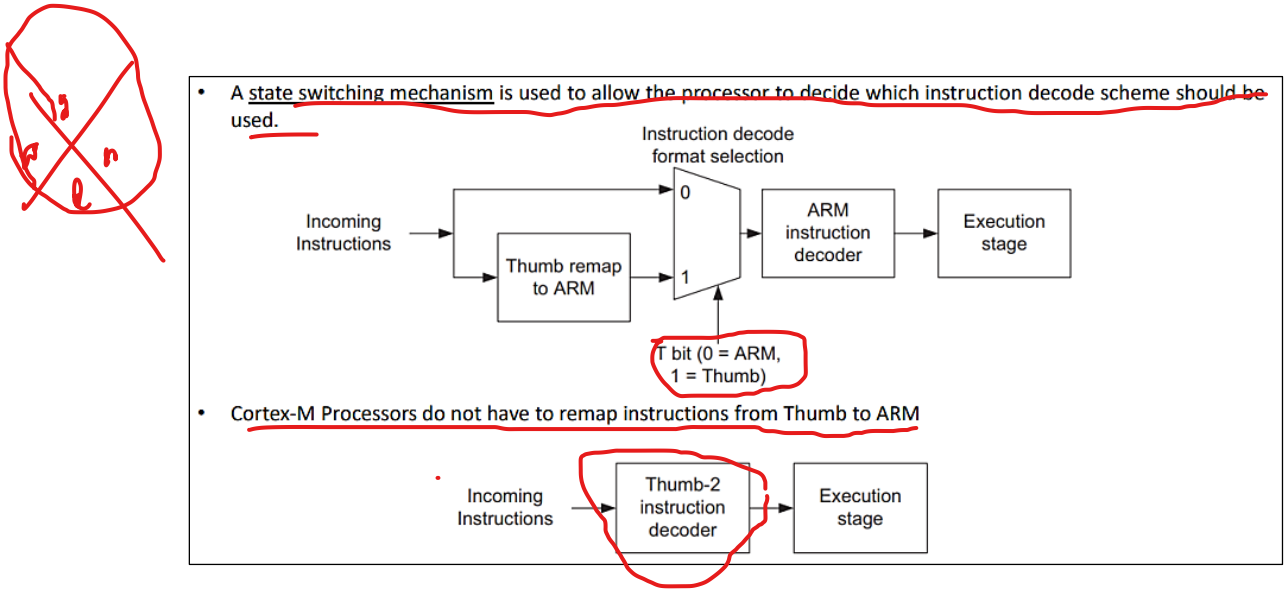
The Instruction Set Architecture (ISA):

- The Instruction Set Architecture (ISA) is a set of rules that define the interface between software and hardware, and How the CPU is controlled by the software.
- Specifying both what the processor is capable of doing as well as how it gets done.
- The ISA defines the supported data types, the registers, how the hardware manages main memory, which instructions a microprocessor can execute
- The Instruction Set Architecture (ISA) in the Cortex-M processors is called the Thumb ISA and is based on Thumb-2 technology which supports a mixture of 16-bit and 32-bit instructions.
- Using a 32-bit instruction set comes with a trade-off in terms of firmware memory usage. Programs written with a 32-bit ISA require a larger number of bytes in flash storage. This increased memory footprint impacts power consumption and overall costs of the microcontroller (MCU). but also guarantees the best performance during the execution of instructions involving arithmetic operations and memory transfers between core registers and SRAM
- To address such issues, ARM introduced the Thumb 16-bit instruction set, which is a subset of the most commonly used 32-bit one
- The earlier ARM CPUs, specifically ARM7 and ARM9, supported two instruction sets:
    1. The 32-bit ARM instruction set.
    2. The 16-bit Thumb instruction set.
- Developers had the option to compile their code as either 32-bit ARM or 16-bit Thumb code.
- The ARM instruction set provided maximum performance, while the Thumb code achieved greater code density.
- Programmers had to decide which functions or sections of code should be compiled using the ARM instruction set and which should use the Thumb instruction set.
- When using the Thumb instruction set, 16-bit instructions were automatically expanded to full 32-bit ARM instructions in real-time execution without any performance loss.
- The linker was responsible for interworking, seamlessly combining the two instruction sets in the final executable code.
- Unlike classic ARM processors like ARM7TDMI, there is no ARM state because the Cortex-M processors do not support the ARM instruction set.
- The focus on the Thumb instruction set allows Cortex-M processors to optimize performance and resources for embedded systems and microcontroller applications.



→ **How the processor deals with the ARM instructions and the Thumb instructions?**

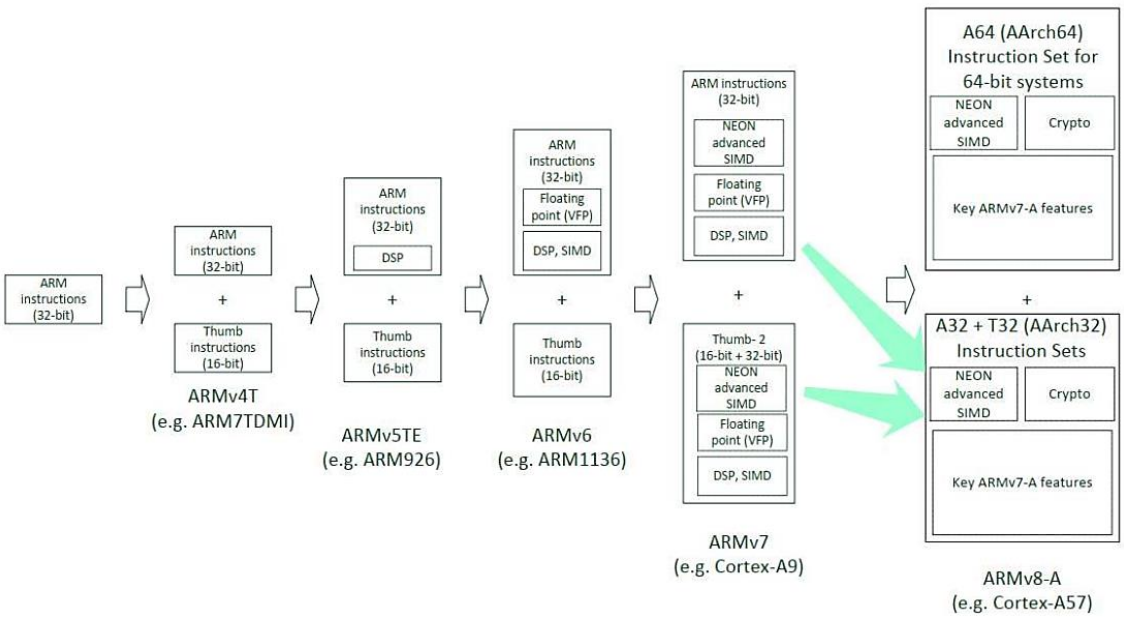
- The processor utilizes a state switching mechanism to handle both ARM and Thumb instructions. This mechanism allows the processor to determine which instruction decode scheme to use based on the current execution state.



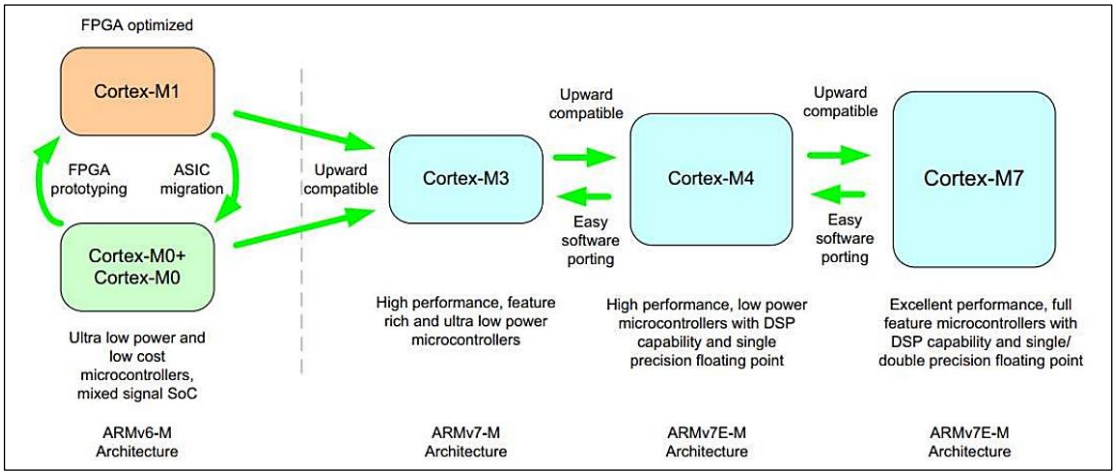
- The first circuit shows how the processor deals with ARM and Thumb instructions. The incoming instructions are first decoded to determine whether they are ARM or Thumb instructions. If they are ARM instructions, they are executed directly. If they are Thumb instructions, they are remapped to ARM instructions before being executed.
- The second circuit shows how the processor deals with Thumb-2 instructions. And Cortex-M Processors do not have to remap instructions from Thumb to ARM
- The T-bit is a bit in the instruction that indicates whether the instruction is a Thumb instruction or an ARM instruction.
  - ✓ If the T-bit is 0, the instruction is an ARM instruction.
  - ✓ If the T-bit is 1, the instruction is a Thumb instruction.

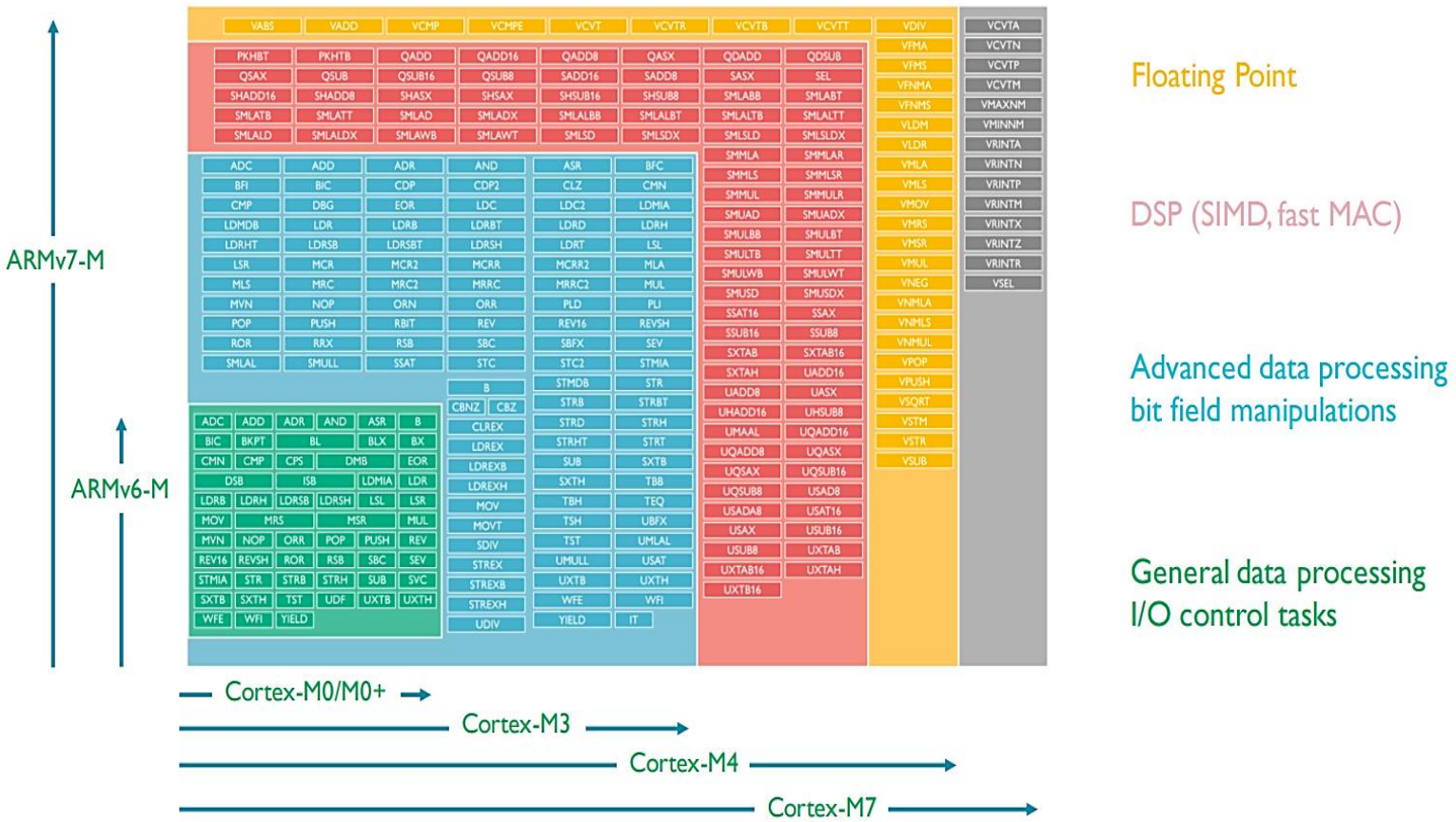
Software Portability between Cortex®-M Processors.

- One of the key characteristics of the ISA in the Cortex-M processors is the upward compatibility: Instruction supported in the Cortex-M3 processor is a superset of Cortex-M0/M0+/M1. So theoretically if the memory map is identical, a binary image for Cortex-M0/M0+/M1 can run directly on a Cortex-M3. The same applies to the relationship between Cortex-M4/M7 and other Cortex-M processors; instructions available on Cortex-M0/M0+/M1/M3 can run on a Cortex-M4/M7.
- The Cortex-M0, Cortex-M0þ, and Cortex-M1 Processors are based on the ARMv6-M Architecture, whereas the Cortex-M3, Cortex-M4, and the Cortex-M7 Processors are based on the ARMv7-M Architecture.



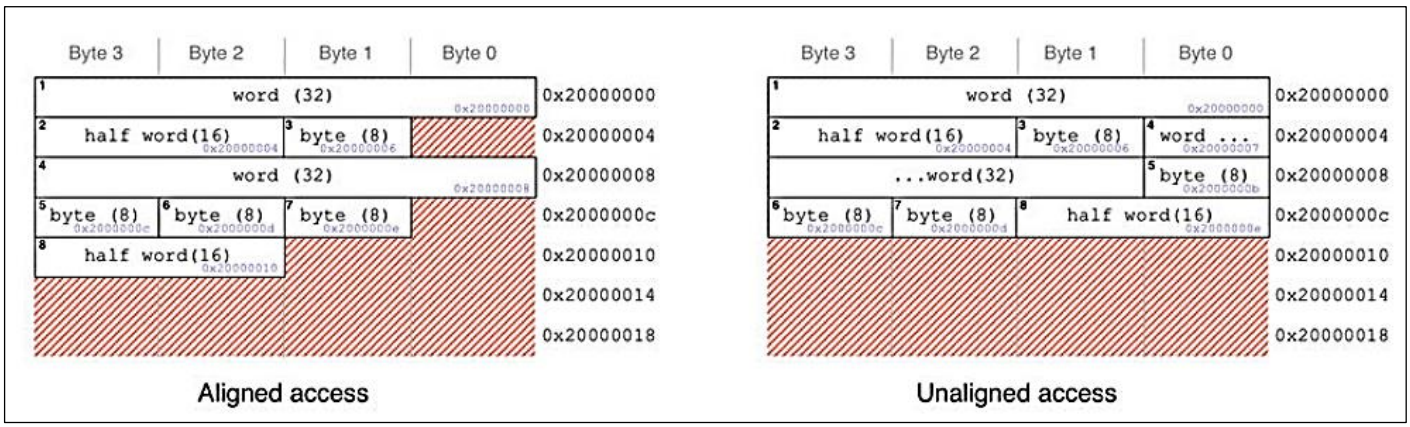
- Latest development of the instruction set in ARM processors supports 64-bit architecture
- In most cases software developed for the Cortex-M0 and Cortex-M0+ can run on the Cortex-M3 and Cortex-M4 Processors without changes, assuming the system has same memory maps and peripherals.
- The Cortex-M7 processor supports all instructions available in the Cortex-M4 processor





Data alignment and unaligned data access support

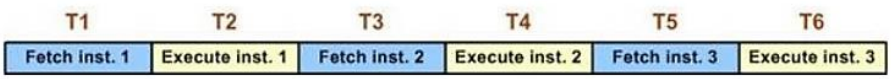
- Cortex-M3/4/7 cores have the ability to do unaligned access to memory
- ARM based CPUs are traditionally capable of accessing byte (8-bit), half word (16-bit) and word (32-bit) signed and unsigned variables, without increasing the number of assembly instructions as it happens on 8-bit MCU architectures.
- early ARM architectures were unable to perform unaligned memory access, causing a waste of memory locations



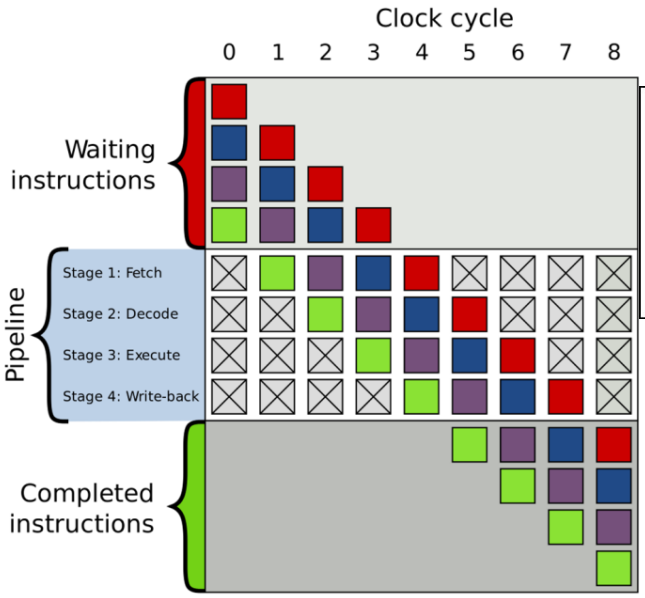
- To understand the problem, consider the left diagram in Figure. Here we have eight variables. With memory aligned access we mean that to access the word variables (1 and 4 in the diagram), we need to access addresses which are multiples of 32-bits (4 bytes). That is, a word variable can be stored only in 0x2000 0000, 0x2000 0004, 0x2000 0008 and so on
- Every attempt to access a location which is not a multiple of 4 causes a Usage Faults exception. So, the following ARM pseudo-instruction is not correct: ➔ **STR R2, 0x20000002**
- The same applies for half word access: it is possible to access to memory locations stored at multiple of 2 bytes: 0x2000 0000, 0x2000 0002, 0x2000 0004 and so on.
- This limitation causes fragmentation inside the RAM memory.
- To solve this issue, Cortex-M3/4/7 based MCUs are able to perform unaligned memory access, as shown in the right diagram in Figure
- As we can see, variable 4 is stored starting at address 0x2000 0007 (in early ARM architectures this was only possible with single byte variables).
- This allows us to store variable 5 in memory location 0x2000 000b, causing variable 8 to be stored in 0x2000 000e. Memory is now packed, and we have saved 4 bytes of SRAM

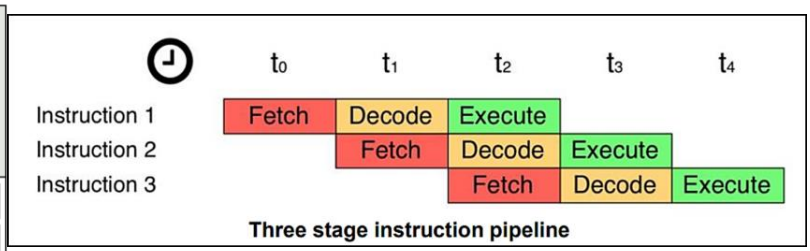
➔Pipeline

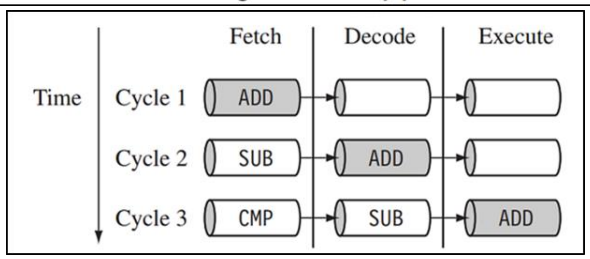
- In early microprocessors such as the 8085, the CPU could either fetch or execute at a given time. In other words, the CPU had to fetch an instruction from memory, decode, and then execute it, and then fetch the next instruction, decode and execute it, and so on as shown in Figure



- Modern CPUs introduce a way to parallelize these operations in order to increase their instructions throughput (the number of instructions which can be executed in a unit of time).
- A pipeline is a technique used in computer processor design to increase the speed of execution of instructions.
- In a pipelined architecture, the processor splits the instruction execution into multiple stages and processes each stage independently in parallel
- Each stage performs a specific part of the instruction execution process and passes the results to the next stage in the pipeline.
- The idea of pipelining in its simplest form is to allow the CPU to fetch and execute at the same time. That is an instruction is being fetched while the previous instruction is being decoded and executed.
- Rather than processing each instruction sequentially (one at a time, finishing one instruction before starting with the next one), each instruction is split into a sequence of stages so that different steps can be executed in parallel.
    - → Fetch loads an instruction from memory
    - → Decode identifies the instruction to be executed.
    - → Execute processes the instruction and writes the result back to a register
- This example shows a sequence of three instructions being fetched, decoded, and executed by the processor. Each instruction takes a single cycle to complete after the pipeline is filled. The three instructions are placed into the pipeline sequentially.
    - → In the first cycle the core fetches the ADD instruction from memory.
    - → In the second cycle the core fetches the SUB instruction and decodes the ADD instruction.
    - → In the third cycle, both the SUB and ADD instructions are moved along the pipeline
    - → After 3 cycles the ADD instruction is executed, the SUB instruction is decoded, and the CMP instruction is fetched

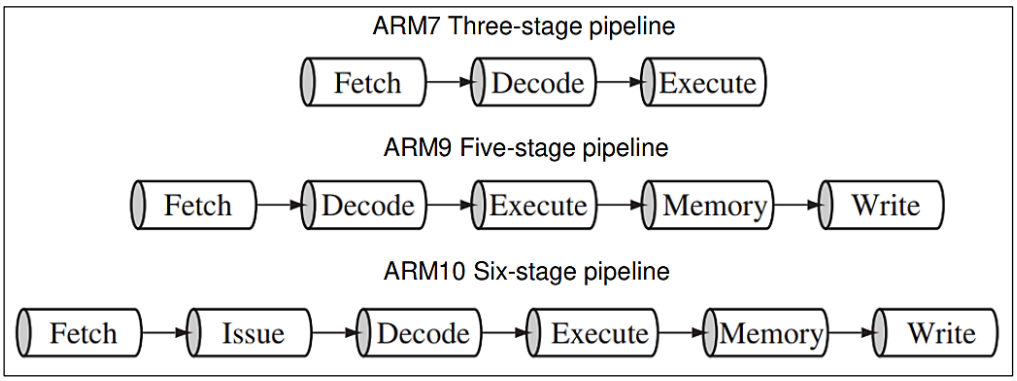




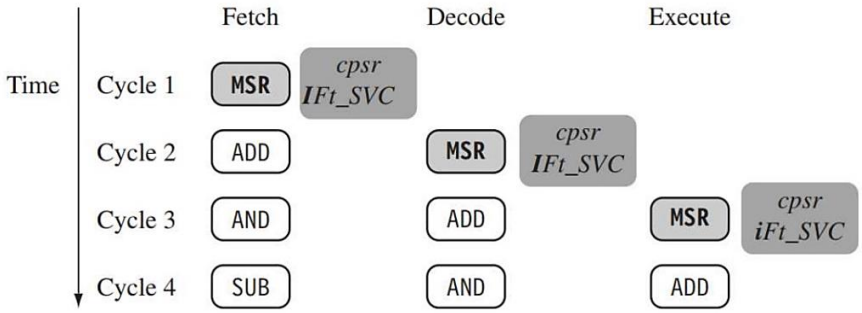


- All Cortex-M based microcontrollers introduce a form of pipelining. The most common one is the 3-stage pipeline.
- 3-stage pipeline is supported by Cortex-M0/3/4.
- Cortex-M0+ cores, which are dedicated to low-power MCUs, provide a 2-stage pipeline.
- Cortex-M7 cores provide a 6-stage pipeline
- The pipeline design for each ARM family differs. For example,
    - ✓ The ARM-9 core increases the pipeline length to five stages.

✓ The ARM-10 core increases the pipeline length to six stages.



- Instruction Fetch (IF): The instruction fetch stage fetches the instruction from memory and stores it in the instruction register. This stage determines the address of the next instruction to be executed.
- Instruction Decode (ID): The Instruction Decode stage decodes the instruction stored in the instruction register and determines the type of operation to be performed.
- Execute (EX): The execute stage performs the actual computation or data manipulation specified by the instruction. This stage may also include arithmetic and logical operations, data transfer operations, and control operations.
- Write Back (WB): The write back stage writes the results of the execution back to memory or to the destination register. This stage also prepares the processor for the next instruction.
- As the pipeline length of a processor increases, several effects can be observed:
  - ✓ Increasing the pipeline length of a processor reduces the amount of work done at each stage, enabling the processor to achieve a higher operating frequency.
  - ✓ The higher operating frequency leads to improved performance, allowing the processor to execute instructions at a faster rate.
  - ✓ However, the longer pipeline also increases system latency, as it takes more cycles to fill the pipeline before the core can execute an instruction.
  - ✓ The increased pipeline length can result in data dependencies between stages.
  - ✓ In short, longer pipelines:
    - Use less power per clock cycle
    - Do less work per clock cycle
    - Needs more clock cycles to perform the same amount of work (And increasing the clock speed pushes power usage up)
    - Create longer wait times when a prediction is missed
    - Allow more space for other instructions to be added
- The ARM pipeline has not processed an instruction until it passes completely through the execute stage. For example, an ARM7 pipeline (with three stages) has executed an instruction only when the fourth instruction is fetched.
- The MSR instruction is used to enable IRQ interrupts, which only occurs once the MSR instruction completes the execute stage of the pipeline
- Once the ADD instruction enters the execute stage of the pipeline, IRQ interrupts are enabled.



CISC vs. RISC

- **CISC** (Complex Instruction Set Computer) and **RISC** (Reduced Instruction Set Computer) are two different approaches to designing computer architecture.
- RISC is the way to make hardware simpler whereas CISC is the single instruction that handles multiple works.

Reduced Instruction Set Architecture (RISC):

- The main idea behind this is to simplify hardware by using an instruction set composed of a few basic steps for loading, evaluating, and storing operations just like a load command will load data, a store command will store the data.
- Characteristics of RISC ➔ **(complex SW / simpler HW)**

efficiency

- **Advantages of RISC**
  - ✓ **Simpler instructions:** RISC processors use a smaller set of simple instructions, which makes them easier to decode and execute quickly. This results in faster processing times.
  - ✓ **Lower power consumption:** RISC processors consume less power than CISC processors, making them ideal for portable devices.
- **Disadvantages of RISC**
  - ✓ **More instructions required:** RISC processors require more instructions to perform complex tasks than CISC processors.
  - ✓ **Increased memory usage:** RISC processors require more memory to store the additional instructions needed to perform complex tasks.
  - ✓ The RISC processor's performance may vary according to the code executed because subsequent instructions may depend on the previous instruction for their execution in a cycle.

Complex Instruction Set Architecture (CISC):

- The main idea is that a single instruction will do all loading, evaluating, and storing operations just like a multiplication command will do stuff like loading data, evaluating, and storing it, hence it's complex.
- Characteristics of CISC ➔ **(complex HW / simpler SW)**
  - ✓ Complex instruction, hence complex instruction decoding.
  - ✓ Instruction may take more than a single clock cycle to get executed.
  - ✓ Instructions can perform complex operations in a single step.
  - ✓ Less number of general-purpose registers as operations get performed in memory itself.
  - ✓ Complex Addressing Modes.
- **Advantages of CISC**
  - ✓ **Reduced code size:** CISC processors use complex instructions that can perform multiple operations, reducing the amount of code needed to perform a task.
  - ✓ **More memory efficient:** Because CISC instructions are more complex, they require fewer instructions to perform complex tasks, which can result in more memory-efficient code.

- ✓ **Widely used:** CISC processors have been in use for a longer time than RISC processors, so they have a larger user base and more available software.

- **Disadvantages of CISC**
  - ✓ **Slower execution:** CISC processors take longer to execute instructions because they have more complex instructions and need more time to decode them.
  - ✓ **More complex design:** CISC processors have more complex instruction sets, which makes them more difficult to design and manufacture.
  - ✓ **Higher power consumption:** CISC processors consume more power than RISC processors because of their more complex instruction sets.

RISC	CISC
RISC is Reduced Instruction Cycle	CISC is Complex Instruction Cycle.
An instruction executed in a single clock cycle	Instruction takes more than one clock cycle
The number of instructions are less as compared to CISC.	The number of instructions are more as compared to RISC.
It consumes the low power.	It consumes more/high power.
RISC required more RAM.	CISC required less RAM.
It emphasizes on software to optimize the instruction set.	It emphasizes on hardware to optimize the instruction set.
RISC has simple decoding of instruction.	CISC has complex decoding of instruction.
It uses LOAD and STORE that are independent instructions in the register-to-register a program's interaction.	It uses LOAD and STORE instruction in the memory-to-memory interaction of a program.
Small code sizes, high cycles per second	Low cycles per second, large code sizes

- **Ex: X = 4*3**

RISC ➔ X = 4+4+4 ➔ 3 steps → (ADD X, 4) (ADD X, 4) (ADD X, 4)

- In RISC the operation will be converted into more than one instruction; as it doesn't have a multiplication circuit
- Each instruction(ADD X, 4) will be executed in one clock cycle

CISC ➔ X = 4*3 ➔ 1 step → (MULL X, 4*3)

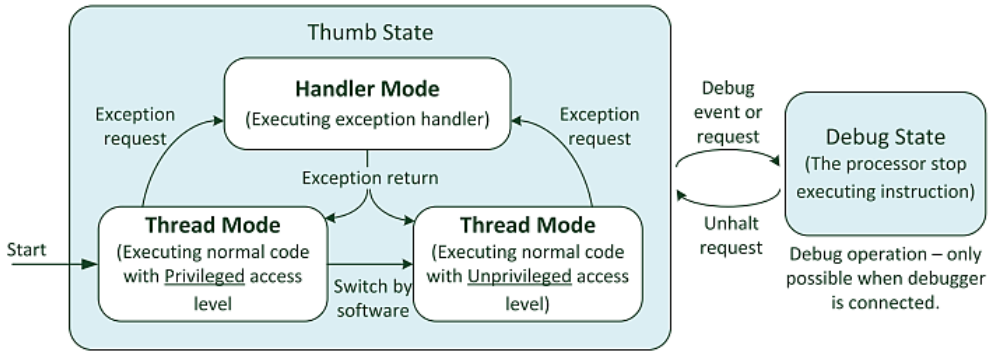
- In CISC the operation will be converted into one instruction; as it has many instruction and circuits more than RISC like multiplication. But the decoding phase will take more than one cycle as complex as the instruction
- So the performance of CISC and RISC almost the same

- The size of two architectures:
  - ✓ RISC ➔ fewer amount of circuits but the instruction decoder circuit is big.
  - ✓ CISC ➔ a large amount of circuits but the instruction decoder circuit is small (micro programmed).
  - ✓ So the overall size of two architectures is almost the same.
- The cost of two architectures:
  - ✓ RISC ➔ fewer amount of circuits have a little cost, but the tool chain for the RISC machines is extremely expensive
  - ✓ CISC ➔ a large amount of circuits have an expensive cost, but the tool chain for the CISC machines is inexpensive
  - ✓ So the overall cost of two architectures is almost the same.

- The power consumption of two architectures:
  - ✓ RISC ➔ it has a circuits which solve the operation at more than one step and more than clock cycles
  - ✓ CISC ➔ it has a circuits which solve the operation at one step but at more than one clock cycle
  - ✓ So the overall power consumption of two architectures is almost the same.

Criteria	RISC	CISC
Performance	The same performance	
Size	The same size	
Cost	The same cost	
Power consumption	The same	

Modes of processor operations and execution:



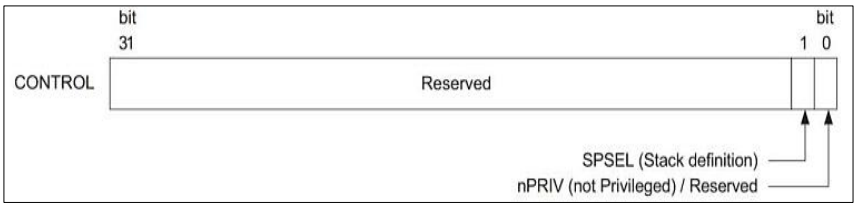
- The Cortex-M4 processors have two operation states and two operation modes:
- In addition, the processors can have privileged and unprivileged access levels.
  - ✓ The privileged access level can access all resources in the processor
  - ✓ While unprivileged access level means some memory regions are inaccessible, and a few operations cannot be used. In some documents, the unprivileged access level might also be referred as "User" state
- **Operation states:**

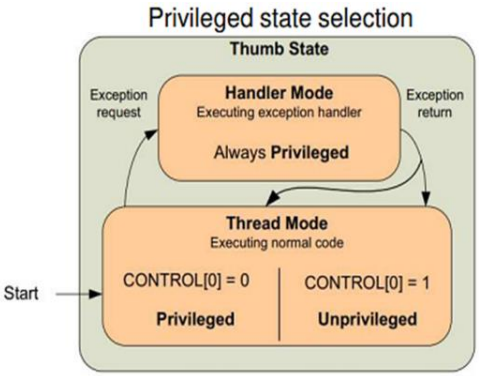
1. Debug state:
   - ✓ When the processor is halted (e.g., by the debugger, or after hitting a breakpoint), it enters debug state and stops executing instructions.
2. Thumb State:
   - ✓ If the processor is running program code (Thumb instructions), it is in the Thumb state.
   - ✓ Unlike classic ARM processors like ARM7TDMI, there is no ARM state because the Cortex-M processors do not support the ARM instruction set.
   - ✓ The focus on the Thumb instruction set allows Cortex-M processors to optimize performance and resources for embedded systems and microcontroller applications.

- **Operation modes:**

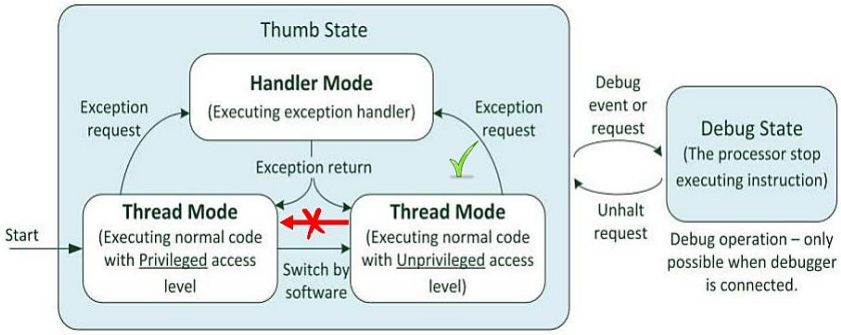
1. Handler mode:
   - ✓ When executing an exception handler such as an Interrupt Service Routine (ISR).
   - ✓ When in handler mode, the processor always has privileged access level.
2. Thread mode:
   - ✓ When executing normal application code, the processor can be either in privileged access level or unprivileged access level
   - ✓ This is controlled by a special register called "CONTROL."

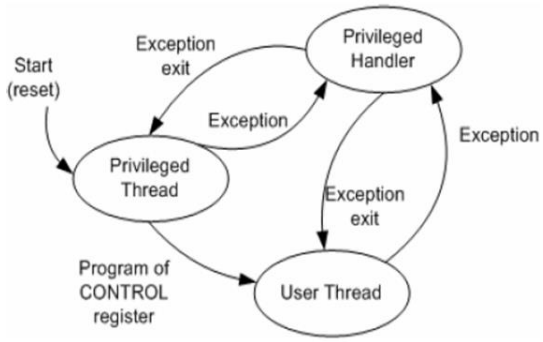
→ Software can switch the processor in privileged Thread mode to unprivileged Thread mode.and this is also controlled by CONTROL (bit0 : nPRIV).





→ However, it cannot switch itself back from unprivileged to privileged. If this is needed, the processor has to use the exception mechanism to handle the switch.

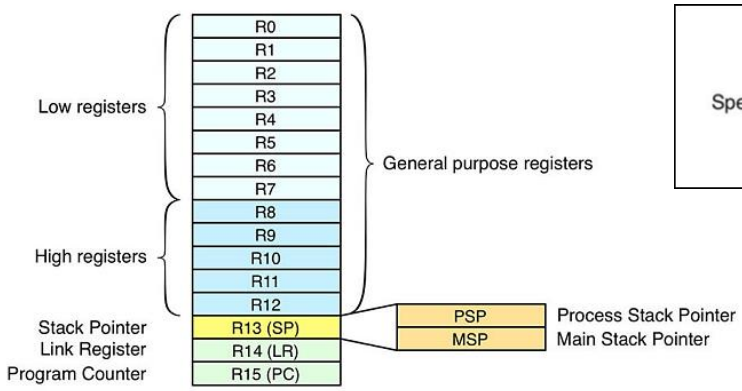


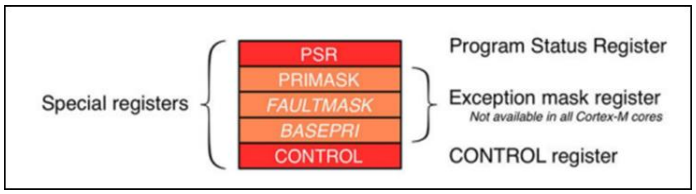


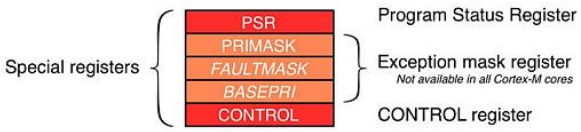
1. The privileged and unprivileged access levels ensure robustness and security in embedded systems.
2. By separating access levels, memory accesses to critical regions are safeguarded and a basic security model is established.
3. This separation prevents untrusted application tasks from gaining privileged access without going through the OS services.
4. The Memory Protection Unit (MPU) can be used to set up memory access permissions, preventing corruption of memory and peripherals used by the OS kernel and other tasks.
5. In case of an application task crash, the remaining tasks and the OS kernel can continue running.
6. It's important to note that most of the NVIC registers are accessible only in privileged access mode.
7. By default, the Cortex-M processors start in privileged Thread mode and in Thumb state
8. Unprivileged Thread model is not available in some processors like Cortex-M0 processor, but is optional in the CortexM0+ processor.

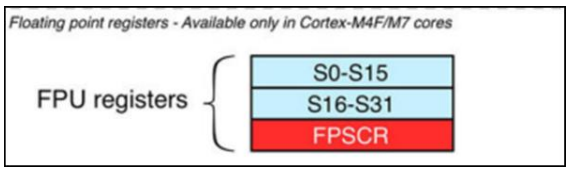
ARM Cortex M4 Architecture [Registers & Special Registers]

- In order to perform data processing and controls, a number of registers are required inside the processor core.
- Each data processing instruction specifies the operation required, the source register(s), and the destination register(s) if applicable
- ARM is a **load/store architecture**
  - ✓ Meaning that most of the instructions can only work on registers. To work with data we must first load it into a register. When we are done working on it, we store it back to memory.
- The register bank in the Cortex-M3 and Cortex-M4 processors has 16 registers. Thirteen of them are general purpose 32-bit registers, and the other three have special uses

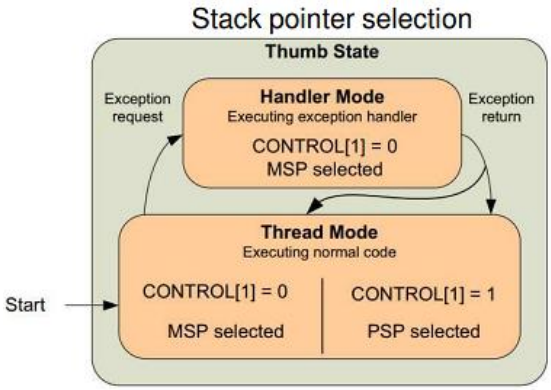




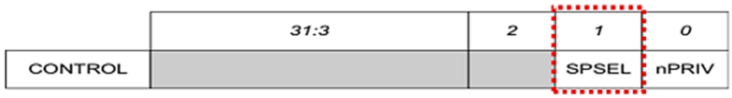




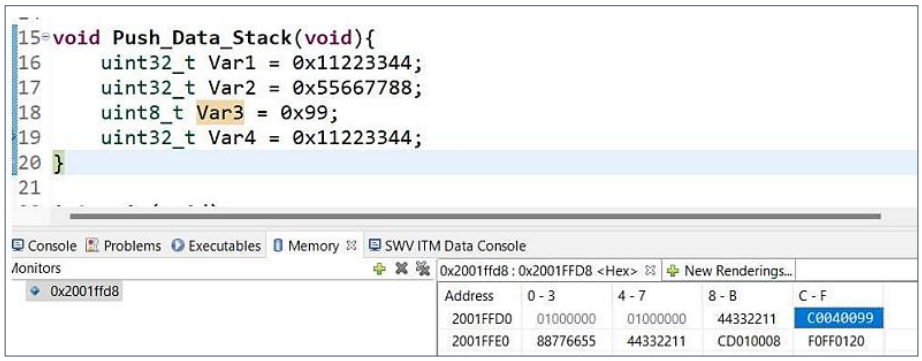
- **General Purpose Registers [R0 : R12]**
  - ✓ The initial values of R0 to R12 are undefined.
  - ✓ When using these registers with ARM development tools such as the ARM assembler, you can use either upper case (Ex. R0) or lower case (Ex. r0) to specify the register to be used.
  - ✓ The first eight **[R0 : R7]** are also called [**Low Registers**].
    - Many 16-bit instructions can only access the low registers.
  - ✓ The first eight **[R8 : R12]** are also called **[High Registers].**
    - Used with 32-bit instructions.
    - A few with 16-bit instructions, like **MOV** (move) instruction.
- **R13, stack pointer (SP) ➔ (PUSH, POP) OR (LOAD, STORE)**
  - ✓ R13 is the Stack Pointer. It is used for accessing the stack memory via PUSH and POP operations.
  - ✓ Physically there are two different Stack Pointers:
    1. The Main Stack Pointer (MSP, or SP_main in some ARM documentation)
       - It's is the default Stack Pointer
       - It is selected after reset, or when the processor is in Handler Mode.
    2. The Process Stack Pointer (PSP, or SP_process in some ARM documentation).
       - Only be used in Thread Mode (we can use the MSP in thread mode).
  - ✓ The selection of Stack Pointer is determined by a special register called "CONTROL":
    - When this bit is 0 (**Default**), Thread mode uses Main Stack Pointer (MSP).
    - When this bit is 1, Thread mode uses Process Stack Pointer (PSP).



Cortex-M3
Cortex-M4



- When using ARM development tools, you can access the stack pointer using either “R13” or “SP.”
  - ✓ Both upper case and lower case (e.g., “r13” or “sp”) can be used.
- In normal programs, only one of these Stack Pointers will be visible
- Both MSP and PSP are 32-bit, but the lowest two bits of the Stack Pointers (either MSP or PSP) are always zero.
  - ✓ Any writes to these two bits are ignored.
- In ARM Cortex-M processors, PUSH and POP are always 32-bit, and the addresses of the transfers in stack operations must be aligned to 32-bit word boundaries.



- For most cases, it is not necessary to use the PSP if the application doesn’t require **an embedded OS**.
- Many simple applications can rely on the MSP completely. The PSP is normally used when an **embedded OS is involved**, where the stack for the OS kernel and application tasks are separated.
- The initial value of PSP is undefined, and the initial value of MSP is taken from the first word of the memory during the reset sequence.
- The instructions "**load**" and "**store**" are generally used for transferring data between registers and memory in the ARM architecture. On the other hand, "**push**" and "**pop**" instructions specifically deal with stack operations.
  - ✓ Load and Store Instructions:
    - Load and store instructions can be used to transfer data of various sizes (such as byte, half-word, or word) between registers and memory.
    - These instructions require the memory addresses to be properly aligned based on the size of the data being transferred. For example, a word-sized load or store instruction would require a 32-bit word-aligned address.
  - ✓ Push and Pop Instructions:
    - Push and pop instructions are primarily used for stack operations, where data is pushed onto the stack or popped from the stack.
    - In ARM Cortex-M processors, push and pop instructions are always 32-bit, meaning they transfer data in word-sized chunks.
    - The addresses of the transfers in stack operations (push and pop) must be aligned to 32-bit word boundaries.
- **R14, link register (LR)**
  - ✓ This is used for holding the return address when calling a function or subroutine.
  - ✓ At the end of the function or subroutine, the program control can return to the calling program and resume by loading the value of LR into the Program Counter (PC).
  - ✓ When a function or subroutine call is made, the value of LR is updated automatically.
  - ✓ If a function needs to call another function or subroutine, it needs to save the value of LR in the stack first. Otherwise, the current value in LR will be lost when the function call is made.
  - ✓ During exception handling, the LR is also updated automatically to a special EXC_RETURN (Exception Return) value, which is then used for triggering the exception return at the end of the exception handler. This will be covered later.

- ✓ Although the return address values in the Cortex-M processors are always even (bit 0 is zero because the instructions must be aligned to half-word addresses), bit 0 of LR is **readable and writeable.**
- ✓ Some of the branch/call operations require that bit zero of LR (or any register being used) be set to 1 to indicate Thumb state.
- ✓ On reset, the processor sets the LR value to 0xFFFFFFFF.

- **R15, program counter (PC)**
  - ✓ It is readable and writeable:
    - A read returns the current instruction address plus 4
    - Writing to PC (e.g., using data transfer/processing instructions) causes a branch operation.
    - Since the instructions must be aligned to half-word or word addresses, the Least Significant Bit (LSB) of the PC is zero.
    - However, when using some of the branch/ memory read instructions to update the PC, you need to set the LSB of the new PC value to 1 to indicate the Thumb state.
      - Otherwise, a fault exception can be triggered, as it indicates an attempt to switch to use ARM instructions (32-bit ARM instructions as in ARM7TDMI), which is not supported.

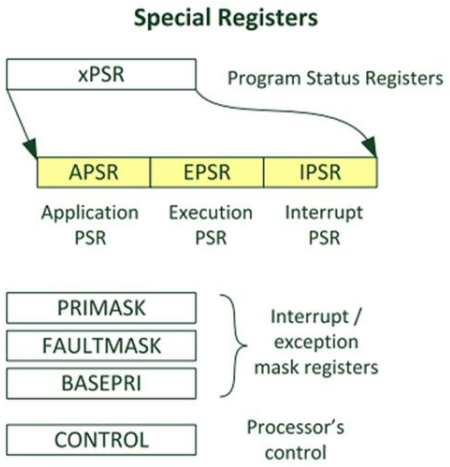
Table 4.1 Allowed Register Names as Assembly Code

Register	Possible Register Names	Notes
R0-R12	R0, R1 … R12, r0, r1 … r12	
R13	R13, r13, SP, sp	Register name MSP and PSP are used in special register access instructions (MRS, MSR)
R14	R14, r14, LR, lr	
R15	R15, r15, PC, pc	

➢ Special registers

- Besides the registers in the register bank, there are a number of special registers. These registers contain the processor status and define the operation states and interrupt/exception masking.
- In the development of simple applications with high level programming languages such as C, there are not many scenarios that require access to these registers. However, they are needed for development of an embedded OS, or when advanced interrupt masking features are needed.
- Special registers are not memory mapped, and can be accessed using special register access instructions such as MSR and MRS
- MRS <reg>, <special_reg>; Read special register into register
- MSR <special_reg>, <reg>; write to special register

1. **Program status registers(PSR):**
   - ✓ The Program Status Register is composed of three status registers
     - Application PSR (APSR)
     - Interrupt PSR (IPSR)
     - Execution PSR (EPSR)



	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
APSR	N	Z	C	V	Q				GE*							
IPSR												Exception Number				
EPSR						ICI/IT	T			ICI/IT						

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T		GE*	ICI/IT		Exception Number				

- These three registers can be accessed as one combined register, referred to as xPSR in some documentation.
- The combined Program Status Register (PSR) provides information about program execution and the ALU flags.
- In ARM assembler, when accessing xPSR, the symbol PSR is used:

```
MRS   r0, PSR   ; Read the combined program status word
MSR   PSR, r0   ; Write combined program state word
```

- You can also access an individual PSR:

```
MRS r0, APSR ; Read Flag state into R0
MRS r0, IPSR ; Read Exception/Interrupt state
MSR APSR, r0 ; Write Flag state
```

- **Notes:**
  - ✓ The EPSR cannot be accessed by software code directly using MRS (read as zero) or MSR.
  - ✓ The IPSR is read only and can be read from combined PSR (xPSR).
  - ✓ *GE is available in ARMv7E-M processors such as the Cortex-M4. It is not available in the Cortex-M3 processor

Table 4.2 Bit Fields in Program Status Registers

Bit	Description
N	Negative flag
Z	Zero flag
C	Carry (or NOT borrow) flag
V	Overflow flag
Q	Sticky saturation flag (not available in ARMv6-M)
GE[3:0]	Greater-Than or Equal flags for each byte lane (ARMv7E-M only; not available in ARMv6-M or Cortex®-M3).
ICI/IT	Interrupt-Continuable Instruction (ICI) bits, IF-THEN instruction status bit for conditional execution (not available in ARMv6-M).
T	Thumb state, always 1; trying to clear this bit will cause a fault exception.
Exception Number	Indicates which exception the processor is handling.

  - ✓ ARMv7-M processors have different PSRs with revised bit fields compared to traditional ARM processors like ARM7TDMI, accommodating features specific to Cortex-M. The APSR and EPSR differ, with some bit fields removed or rearranged.

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
ARM general (Cortex-A/R)	N	Z	C	V	Q	IT	J	Reserved	GE[3:0]	IT	E	A	I	F	T	M[4:0]
ARM7TDMI (ARMv4)	N	Z	C	V	Reserved								I	F	T	M[4:0]
ARMv7-M (Cortex-M3)	N	Z	C	V	Q	ICI/IT	T			ICI/IT		Exception Number				
ARMv7E-M (Cortex-M4)	N	Z	C	V	Q	ICI/IT	T		GE[3:0]	ICI/IT		Exception Number				
ARMv6-M (Cortex-M0)	N	Z	C	V			T									Exception Number

→Comparing PSR of various ARM architectures

- The APSR contains the ALU flags.
  - ✓ N (negative flag), Z (zero flag), C (carry or borrow flag), and V (overflow flag)
  - ✓ These bits are at the top 4 bits of the APSR.
  - ✓ The common use of these flags is to control conditional branches.
- The IPSR contains the current executing ISR (Interrupt Service Routine) number.
  - ✓ Each exception on the Cortex-M4 processor has a unique associated ISR number (exception type).
  - ✓ This is useful for identifying the current interrupt type during debugging and allows an exception handler that is shared by several exceptions to know which exception it is serving.
- The EPSR on the Cortex-M4 processor contains the T bit which indicates that the processor is in the Thumb state.
  - ✓ This bit is normally set to 1 because the Cortex-M processors only support Thumb state.
  - ✓ If this bit is cleared, a HardFault exception will be generated in the next instruction execution.
- **Integer Status Flags:**
  - ✓ The integer status flags are very similar to ALU status flags in many other processor architectures.
  - ✓ The N, Z, C, and V flags are updated by arithmetic operations.

	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T		GE*	ICI/IT		Exception Number				

  - ✓ **N (bit 31):**
    - → "1" the result has a negative value (when interpreted as a signed integer).
    - → "0" the result has a positive value or equal zero.
  - ✓ **Z (bit 30):**
    - → "1" if the result of the executed instruction is zero.
    - → It can also be set to "1" after a compare instruction is executed if the two values are the same.
  - ✓ **C (bit 29) Carry flag of the result:**
    - → For (unsigned addition) operations, "1" if an unsigned overflow occurred.
    - → For unsigned subtract operations, this bit is the inverse of the borrow output status.
    - → This bit is also updated by shift and rotate operations.
  - ✓ **V (bit 28) Overflow of the result:**
    - → For signed addition or subtraction, this bit is set to "1" if a signed overflow occurred.

- **Carry Flag vs. Overflow Flag:**
  - ✓ The carry flag can occur on its own or together with the overflow flag.
  - ✓ The CPU's Arithmetic Logic Unit (ALU) does not differentiate between signed and unsigned mathematics.
  - ✓ The ALU always sets both the carry and overflow flags appropriately during integer math operations.
  - ✓ It is the responsibility of the programmer to know which flag to check after the math operation is performed.
  - ✓ If the program treats the bits in a word as unsigned numbers, it must watch for the carry flag, indicating an incorrect result.
  - ✓ If the program treats the bits in a word as **two's complement signed values**, it must watch for the **overflow flag**, indicating an incorrect result.
  - ✓ The Overflow flag is irrelevant in unsigned math.
  - ✓ In signed arithmetic, the Overflow flag is used to detect errors.
  - ✓ You don't care about the carry flag when doing signed, two's complement math. (The carry flag is only relevant to unsigned numbers, not signed.)
  - ✓ In unsigned arithmetic, watch the carry flag to detect errors.
  - ✓ In unsigned arithmetic, the overflow flag tells you nothing interesting.
  - ✓ In signed arithmetic, watch the overflow flag to detect errors.
  - ✓ In signed arithmetic, the carry flag tells you nothing interesting.

1. Carry Flag

- ✓ The rules for turning on the carry flag in binary/integer math are two:
  1. The carry flag is set if the addition of two numbers causes a carry out of the most significant (leftmost) bits added ➔ 1111 + 0001 = 0000 (carry flag is turned on)
  2. The carry (borrow) flag is also set if the subtraction of two numbers requires a borrow into the most significant (leftmost) bits subtracted. ➔ 0000 - 0001 = 1111 (carry flag is turned on)
  - Otherwise, the carry flag is turned off (zero).
    - → 0111 + 0001 = 1000 (carry flag is turned off [zero])
    - → 1000 - 0001 = 0111 (carry flag is turned off [zero])
  - In unsigned arithmetic, watch the carry flag to detect errors.
  - In signed arithmetic, the carry flag tells you nothing interesting.

2. Overflow Flag

- ✓ The rules for turning on the overflow flag in binary/integer math are two:
  1. If the sum of two numbers with the sign bits off yields a result number with the sign bit on, the "overflow" flag is turned on. ➔ 0100 + 0100 = 1000 (overflow flag is turned on)
  2. If the sum of two numbers with the sign bits on yields a result number with the sign bit off, the "overflow" flag is turned on. ➔ 1000 + 1000 = 0000 (overflow flag is turned on)
  - Otherwise, the overflow flag is turned off.
    - → 0100 + 0001 = 0101 (overflow flag is turned off)
    - → 0110 + 1001 = 1111 (overflow flag is turned off)
    - → 1000 + 0001 = 1001 (overflow flag is turned off)
    - → 1100 + 1100 = 1000 (overflow flag is turned off)
  - Note that you only need to look at the sign bits (leftmost) of the three numbers to decide if the overflow flag is turned on or off.

→ **ALU Flags Example:**

Operation	Results, Flags
0x70000000 + 0x70000000	Result = 0xE0000000, N= 1, Z=0, C = 0, V = 1
0x90000000 + 0x90000000	Result = 0x30000000, N= 0, Z=0, C = 1, V = 1
0x80000000 + 0x80000000	Result = 0x00000000, N= 0, Z=1, C = 1, V = 1
0x00001234 – 0x00001000	Result = 0x00000234, N= 0, Z=0, C = 1, V = 0
0x00000004 – 0x00000005	Result = 0xFFFFFFFF, N= 1, Z=0, C = 0, V = 0
0xFFFFFFFF – 0xFFFFFFFC	Result = 0x00000003, N= 0, Z=0, C = 1, V = 0
0x80000005 – 0x80000004	Result = 0x00000001, N= 0, Z=0, C = 1, V = 0
0x70000000 – 0xF0000000	Result = 0x80000000, N= 1, Z=0, C = 0, V = 1
0xA0000000 – 0xA0000000	Result = 0x00000000, N= 0, Z=1, C = 1, V = 0

➢ **Exception mask registers: The PRIMASK, FAULTMASK, and BASEPRI**

- ✓ PRIMASK : Priority Mask Register
- ✓ FAULTMASK : Fault Mask Register
- ✓ BASEPRI : Base Priority Mask Register

- The PRIMASK, FAULTMASK, and BASEPRI registers are all used for exception or interrupt masking
  - ✓ The exception mask registers disable the handling of exceptions by the processor
  - ✓ Disable exceptions where they might impact on timing critical tasks
- Each exception (including interrupts) has a priority level where a smaller number is a higher priority and a larger number is a lower priority
- These special registers are used to mask exceptions based on priority levels.
- They can only be accessed in the privileged access level
  - ✓ in **unprivileged** state writes to these registers are **ignored** and reads return **zero**
  - ✓ To access the exception mask registers use the **MSR** and **MRS** instructions
  - ✓ Also the **CPS** instruction to change the value of **PRIMASK** or **FAULTMASK**
- By default, they are all **zero**, which means the masking (disabling of exception/interrupt) is not active.

➢ **Priority Mask Register(PRIMASK):**

→ The PRIMASK register is a 1-bit wide interrupt mask register.

→ When set, **it blocks all exceptions** (including interrupts) apart from the Non-Maskable Interrupt (NMI) and the HardFault exception.

- ✓ The PRIMASK register prevents the activation of all exceptions with configurable priority
- ✓ Effectively it raises the current exception priority level to 0, which is the highest level for a programmable exception/interrupt.
- ✓ The most common usage for PRIMASK is to disable all interrupts for a time critical process.

Bits	Description
Bits 31:1	Reserved
Bit 0	**PRIMASK**: 0: No effect 1: Prevents the activation of all exceptions with configurable priority.

- After the time critical process is completed, the PRIMASK needs to be cleared to re-enable interrupts.
- The **PRIMASK** register is used to disable all exceptions except **NMI** and **HardFault**. It effectively changes the current priority **level to 0** (highest programmable level).
- In C programming, you can use the functions provided in CMSIS-Core to set and clear PRIMASK:

```
void __enable_irq(); // Clear PRIMASK
void __disable_irq(); // Set PRIMASK
void __set_PRIMASK(uint32_t priMask); // Set PRIMASK to value
uint32_t __get_PRIMASK(void); // Read the PRIMASK value
```

- In assembly language programming, you can change the value of PRIMARK register using CPS (Change Processor State) instructions:

```
CPSIE I ; Clear PRIMASK (Enable interrupts)
CPSID I ; Set PRIMASK (Disable interrupts)
```

- The PRIMASK register can also be accessed using the MRS and MSR instructions. For example:

```
MOVS R0, #1
MSR PRIMASK, R0 ; Write 1 to PRIMASK to disable all interrupts


MOVS R0, #0
MSR PRIMASK, R0 ; Write 0 to PRIMASK to allow interrupts
```

Exception Number	IRQ Number	Exception Type	Priority	Function
1	-15	Reset	-3, the highest	Reset
2	-14	NMI	-2	Non-Maskable Interrupt
3	-13	Hard Fault	-1	All faults that hang the processor
4	-12	Memory Fault	Configurable	Memory issue
5	-11	Bus Fault	Configurable	Data Bus issue
6	-10	Usage Fault	Configurable	Instruction/State/Access issue
7 ~ 10		Reserved	—	Reserved
11	-5	SVCall	Configurable	System Service Call when call SVC instruction
12		Debug	Configurable	Debug monitor (via SWD)
13		Reserved	—	Reserved
14	-2	PendSV	Configurable	For context switching in an OS
15	-1	SysTick	Configurable	System Timer
16 ~ 255	0~239	Interrupt (IRQ)	Configurable	Interrupt Request by a peripheral, or software request

- The reset vector in the system includes the initial stack pointer, which is the starting point for the program execution after a reset event.
- Peripherals in the system are assigned positive Interrupt Request (IRQ) numbers, which are used to identify and handle interrupts generated by these peripherals.
- CPU exceptions are assigned negative IRQ numbers.
- IRQ numbers are utilized in function calls within the Cortex Microcontroller Software Interface Standard (CMSIS). This allows software developers to handle interrupts and perform related operations.
- The Cortex-M4 processor architecture supports a maximum of 240 IRQs
- The priorities assigned to the various IRQs in the system are programmable by the user. This allows for customized handling of interrupts based on their importance or urgency.
- The priorities for Non-Maskable Interrupts (NMI) and HardFault exceptions are fixed and cannot be modified by the user.

➢ **Fault Mask Register(FAULTMASK):**

- The FAULTMASK is very similar to PRIMASK except that it changes the effective current priority level to -1, so that even the HardFault handler is blocked.
- Only the **NMI exception handler** can be executed when FAULTMASK is set.
- In terms of usage, FAULTMASK is intended for configurable fault handlers (i.e., MemManage, Bus Fault, Usage Fault) to raise its priority to -1.
  - ✓ so that these handlers can have access to some special features for HardFault exceptions
- By raising the current priority level to -1, the FAULTMASK also allows configurable fault handlers to prevent other exceptions or interrupt handlers executing while an issue is being addressed.
- The FAULTMASK register can only be accessed in privileged state, but cannot be set within NMI and HardFault handlers.
  - ✓ **PRIMASK ➔ Disable all interrupts except the Non-Maskable-Interrupt (NMI) and hard fault**
  - ✓ **FAULTMASK ➔ Disable all interrupts except the NMI**

Bits	Function
Bits 31:1	Reserved
Bit 0	**FAULTMASK**: 0: No effect 1: Prevents the activation of all exceptions except for NMI.

- In C programming with CMSIS-compliant driver libraries, you can use the following CMSIS-Core functions to set and clear the FAULTMASK as follows:

```
void __enable_fault_irq(void); // Clear FAULTMASK
void __disable_fault_irq(void); // Set FAULTMASK to disable interrupts
void __set_FAULTMASK(uint32_t faultMask);
uint32_t __get_FAULTMASK(void);
```

- For assembly language users, you can change the current status of the FAULTMASK using CPS instructions as follows:

```
MOVS R0, #1
MSR FAULTMASK, R0 ; Write 1 to FAULTMASK to disable all interrupts

MOVS R0, #0
MSR FAULTMASK, R0 ; Write 0 to FAULTMASK to allow interrupts
```

- FAULTMASK is cleared automatically upon exiting the exception handler except return from NMI handler.
- This characteristic provides an interesting usage for FAULTMASK: if in a lower-priority exception handler we want to trigger a higher-priority handler (except NMI), but want this higher-priority handler to start after the lower-priority handler is completed, we can:
  1. Set the FAULTMASK to disable all interrupts and exceptions (apart from the NMI exception)
  2. Set the pending status of the higher-priority interrupt or exception
  3. Exit the handler ➔ Clear FAULTMASK (Enable all interrupts and exceptions)
- Because the pending higher-priority exception handler cannot start while the FAULTMASK is set, the higher-priority exception stays in the pending state until FAULTMASK is cleared, this happens when the lower-priority handler finishes.
- As a result, you can force the higher-priority handler to start after the lower-priority handler ends

➢ Base Priority Mask Register(BASEPRI):

- In some cases, you might only want to disable interrupts with priority lower than a certain level. In this case, you could use the BASEPRI register.
- To do this, simply write the required masking priority level to the BASEPRI register.
- For example, if you want to block all exceptions with priority level equal to or lower than 0x60, you can write the value to BASEPRI:

```
__set_BASEPRI(0x60); // Disable interrupts with priority
                     // 0x60-0xFF using CMSIS-Core function
```

- For users of assembly language, the same operation can be written as:

```
MOVS R0, #0x60
MSR BASEPRI, R0 ; //Disable interrupts with priority 0x60-0xFF
```

- You can also read back the value of BASEPRI:

```
x = __get_BASEPRI(void); // Read value of BASEPRI
```

- or in assembly language:

```
MRS R0, BASEPRI
```

- To cancel the masking, just write 0 to the BASEPRI register:

```
__set_BASEPRI(0x0); // Turn off BASEPRI masking
```

- Or in assembly language:

```
MOVS R0, #0x0
MSR BASEPRI, R0 ; //Turn off BASEPRI masking
```

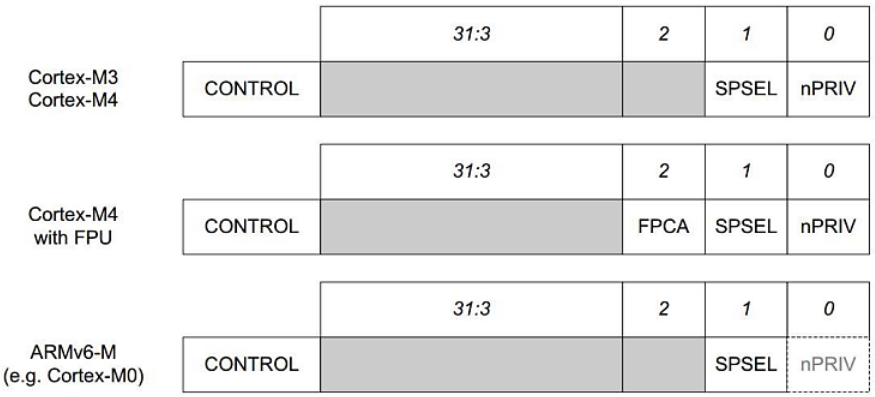
- Using the **BASEPRI register,** we can create interrupt-safe-critical section.
  - ✓ Example: Whenever the **FreeRTOS** is creating a critical section, it writes a (**configured value**) to the **BASEPRI** mask register, effectively blocking any interrupts with NVIC value (**configured value**) or greater.
- The most common usage for PRIMASK is to disable all interrupts for a time critical process:

```
/* A task that calls vDemoFunction() from within a critical section. */
void vTask1( void * pvParameters )
{
    for( ;; )
    {
        /* Perform some functionality here. */

        /* Call taskENTER_CRITICAL() to create a critical section. */
        taskENTER_CRITICAL();


        /* Execute the code that requires the critical section here. */

        /* Calls to taskENTER_CRITICAL() can be nested so it is safe to call a
        function that includes its own calls to taskENTER_CRITICAL() and
        taskEXIT_CRITICAL(). */
        vDemoFunction();

        /* The operation that required the critical section is complete so exit the
        critical section.  After this call to taskEXIT_CRITICAL(), the nesting depth
        will be zero, so interrupts will have been re-enabled. */
        taskEXIT_CRITICAL();
    }
}
```

➢ Control Register and It's usage

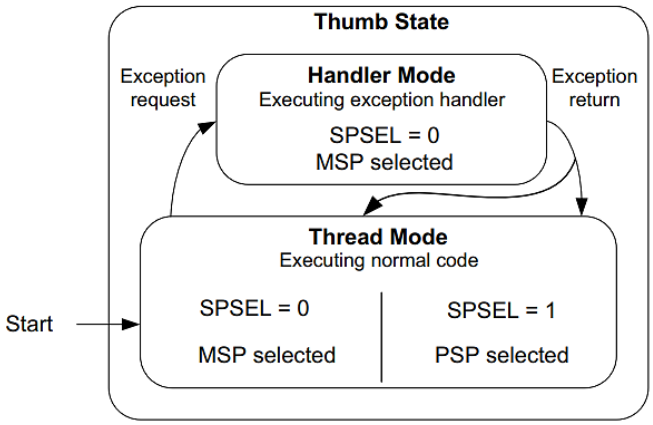
- The selection of stack pointer (Main Stack Point/Process Stack Pointer)
- Access level in Thread mode (Privileged/Unprivileged)
- In addition, for Cortex-M4 processor with a floating point unit, one bit of the CONTROL register indicates if the current context (currently executed code) uses the floating point unit or not
- The CONTROL register can only **be modified in the privileged access level** and can be read in both privileged and unprivileged access levels.

- After reset**, the CONTROL register is 0.**
  - ✓ This means **the Thread mode uses the Main Stack Pointer** as Stack Pointer and Thread mode has privileged accesses.
  - ✓ Programs **in privileged Thread mode** can switch the Stack Pointer selection or switch to unprivileged access level by writing to CONTROL.
  - ✓ Once **nPRIV (CONTROL bit 0) is set**, the program running in Thread can no longer access the CONTROL register.
- Control Register in different cortex M architecture
  - CONTROL register in Cortex-M3, Cortex-M4, and Cortex-M4 with FPU.
  - The bit **nPRIV** is not available in the Cortex-M0 and is optional in the Cortex-M0+ processor



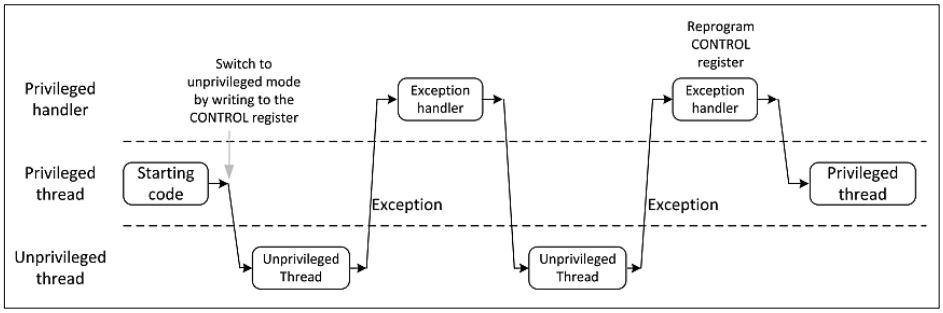
1. **nPRIV (bit 0)** Thread mode privilege level:
   - ✓ Defines the privileged level in Thread mode:
   - ✓ When this bit is **0 (default)**, it is **privileged** level when **in Thread mode**.
   - ✓ When this bit is **1**, it is **unprivileged** when in **Thread mode**.
   - ✓ In Handler mode, the processor is always in privileged access level.
2. **SPSEL (bit 1)** Defines the Stack Pointer selection:
   - ✓ When this bit is **0 (default), Thread mode** uses **Main Stack Pointer (MSP).**
   - ✓ When this bit is **1**, **Thread mode** uses **Process Stack Pointer (PSP).**
   - ✓ In Handler mode, **this bit is always 0 and write to this bit is ignored**
3. **FPCA (bit 2)** Floating Point Context Active:
   - ✓ This bit is only available in Cortex-M4 with floating point unit implemented.
   - ✓ The exception handling mechanism uses this bit to determine if registers in the floating point unit need to be saved when an exception has occurred.
   - ✓ When this bit is **0 (default),** the floating point unit has **not been used** in the current context and therefore there is no need to save floating point registers.
   - ✓ When this bit is **1**, the current context **has used floating point instructions** and therefore need to save floating point registers.
   - ✓ The FPCA bit is set automatically when a floating point instruction is executed.
   - ✓ **This bit is clear by hardware on exception entry.**

- After reset**, the CONTROL register is 0.**



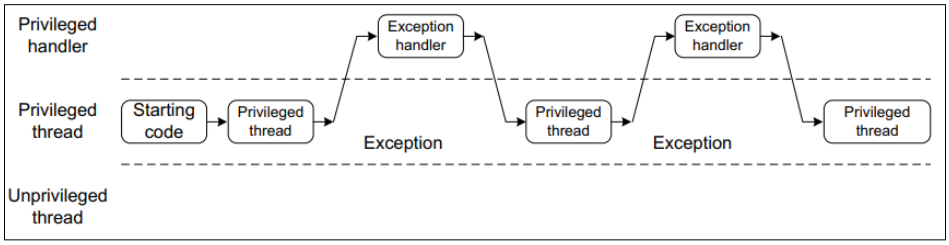
- A program in unprivileged access level cannot switch itself back to privileged access level.
  - ✓ This is essential in order to provide a basic security usage model.

- For example:
  - ✓ An embedded system might contain untrusted applications running in unprivileged access level and the access permission of these applications must be restricted to prevent security breaches or to prevent an unreliable application from crashing the whole system.
  - ✓ If it is necessary to switch the processor back to using privileged access level in Thread mode, then the exception mechanism is needed
  - ✓ During exception handling, the exception handler can clear the nPRIV bit.
  - ✓ When returning to thread mode, the processor will be in privileged access level.
  - ✓ When an embedded OS is used, the CONTROL register could be reprogrammed at each context switch to allow some application tasks to run with privileged access level and the others to run with unprivileged access level.



Switching between privileged thread mode 1

- Simple applications do not require unprivileged Thread mode



Simple applications do not require unprivileged Thread mode

nPRIV	SPSEL	Usage Scenario
0	0	Simple applications – the whole application is running in privileged access level. Only one stack is used by the main program and interrupt handlers. Only the Main Stack Pointer (MSP) is used.
0	1	Applications with an embedded OS, with current executing task running in privileged Thread mode. The Process Stack Pointer (PSP) is selected in current task, and the MSP is used by OS Kernel and exception handlers.
1	0	Thread mode tasks running with unprivileged access level and use MSP. This can be observed in Handler mode but is less likely to be used for user tasks because in most embedded OS, the stack for application tasks is separated from the stack used by OS kernel and exception handlers.
1	1	Applications with an embedded OS, with current executing task running in unprivileged Thread mode. The Process Stack Pointer (PSP) is selected in current task, and the MSP is used by OS Kernel and exception handlers.

- In most simple applications without an embedded OS, there is no need to change the value of the CONTROL register.
  - ✓ The whole application can run in privileged access level and use only the MSP.
- To access the CONTROL register in C, the following functions are available in CMSIS-compliant device-driver libraries:

```
/**
  \brief   Get Control Register
  \details Returns the content of the Control Register.
  \return               Control Register value
 */
__STATIC_INLINE uint32_t __get_CONTROL(void)
{
  register uint32_t __regControl         __ASM("control");
  return(__regControl);
}
```

```
/**
  \brief   Set Control Register
  \details Writes the given value to the Control Register.
  \param [in]    control  Control Register value to set
 */
__STATIC_INLINE void __set_CONTROL(uint32_t control)
{
  register uint32_t __regControl         __ASM("control");
  __regControl = control;
}
```

```
#define __get_CONTROL()             (__arm_rsr("CONTROL"))
#define __set_CONTROL(VALUE)        (__arm_wsr("CONTROL", (VALUE)))
```

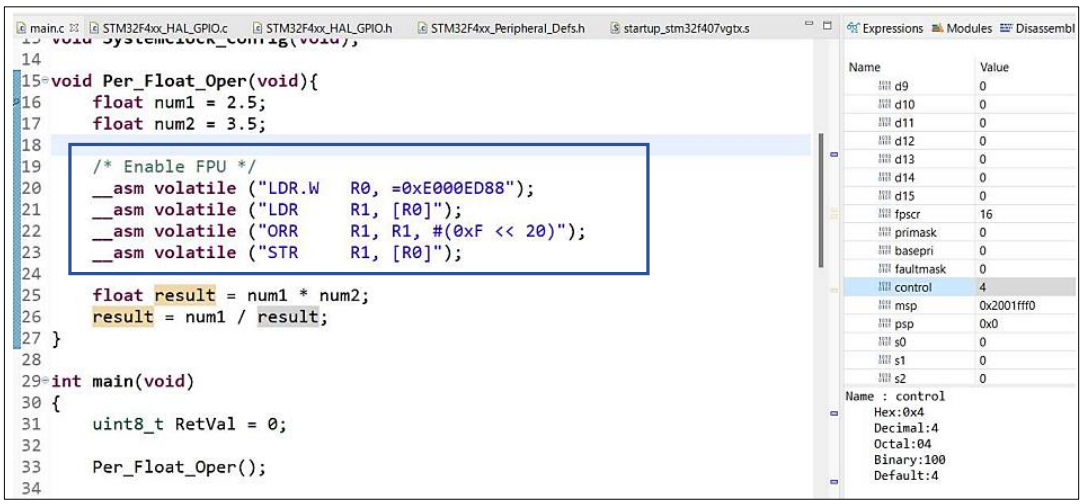
- To access the Control register in assembly, the MRS and MSR instructions are used:

```
MRS   r0, CONTROL ; Read CONTROL register into R0
MSR   CONTROL, r0 ; Write R0 into CONTROL register
```

- You can detect if the current execution level is privileged by checking the value of IPSR and CONTROL registers

```
int in_privileged(void)
{
    if (__get_IPSR() != 0)
    {
        return 1; // True
    }
    else
    {
        if ((__get_CONTROL() & 0x1) == 0)
        {
            return 1; // True
        }
        else
        {
            return 0; // False
        }
    }
}
```

- There are two points that you need to be aware of when changing the value of the CONTROL register:
  1. For the Cortex-M4 processor with floating point unit (FPU), or any variant of ARMv7-M processors with (FPU), the FPCA bit can be set automatically due to the presence of floating point instructions.
     - ✓ If the program contains floating point operations and the FPCA bit is cleared accidentally, and subsequently an interrupt occurs, the data in registers in the floating point unit will not be saved by the exception entry sequence and could be overwritten by the interrupt handler.
     - ✓ In this case, the program will not be able to continue correct processing when resuming the interrupted task
  2. After modifying the CONTROL register, architecturally an Instruction Synchronization Barrier (ISB) instruction (or __ISB() function in CMSIS compliant driver) should be used to ensure the effect of the change applies to subsequent code.
     - ✓ Due to the simple nature of the Cortex-M3, Cortex-M4, Cortex-M0þ, Cortex-M0, and Cortex-M1 pipeline, omission of the ISB instruction does not cause any problem.
     - ✓ Note:: The FPCA bit is clear by hardware on exception entry.



_____

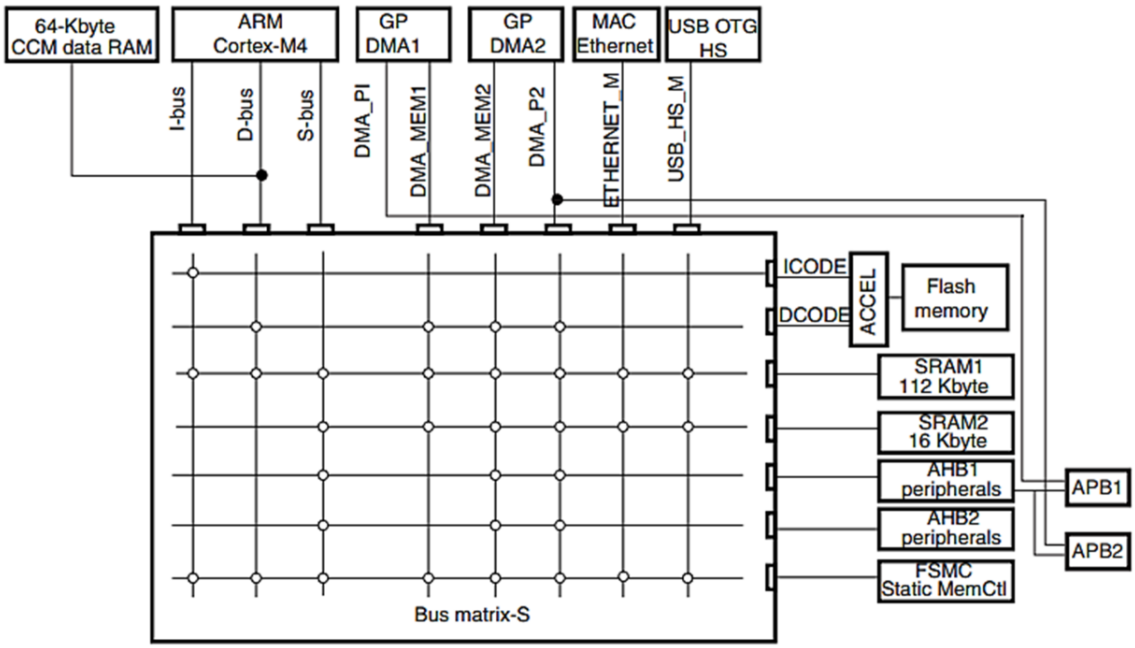
_____

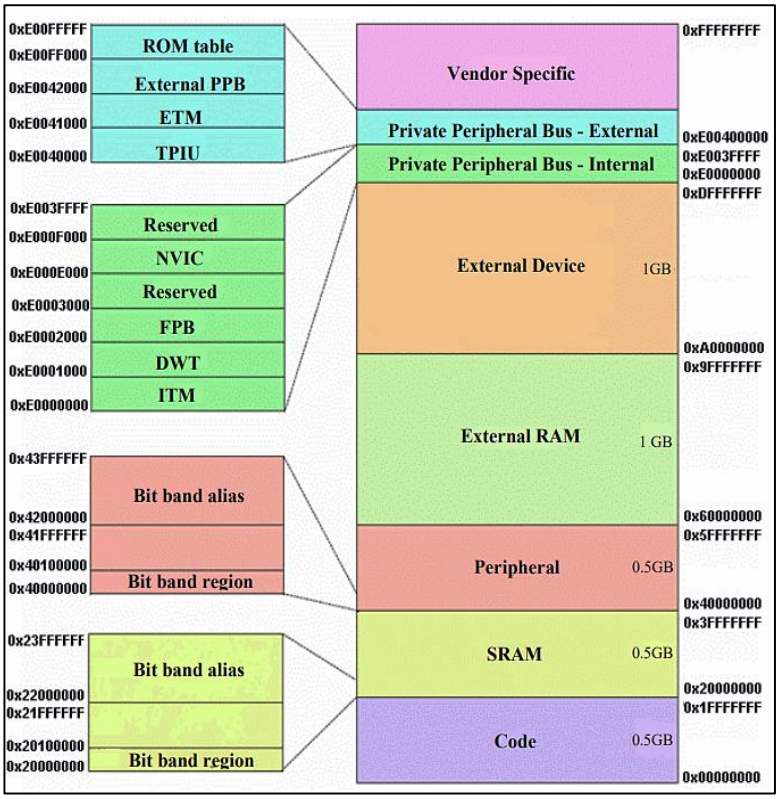
_____

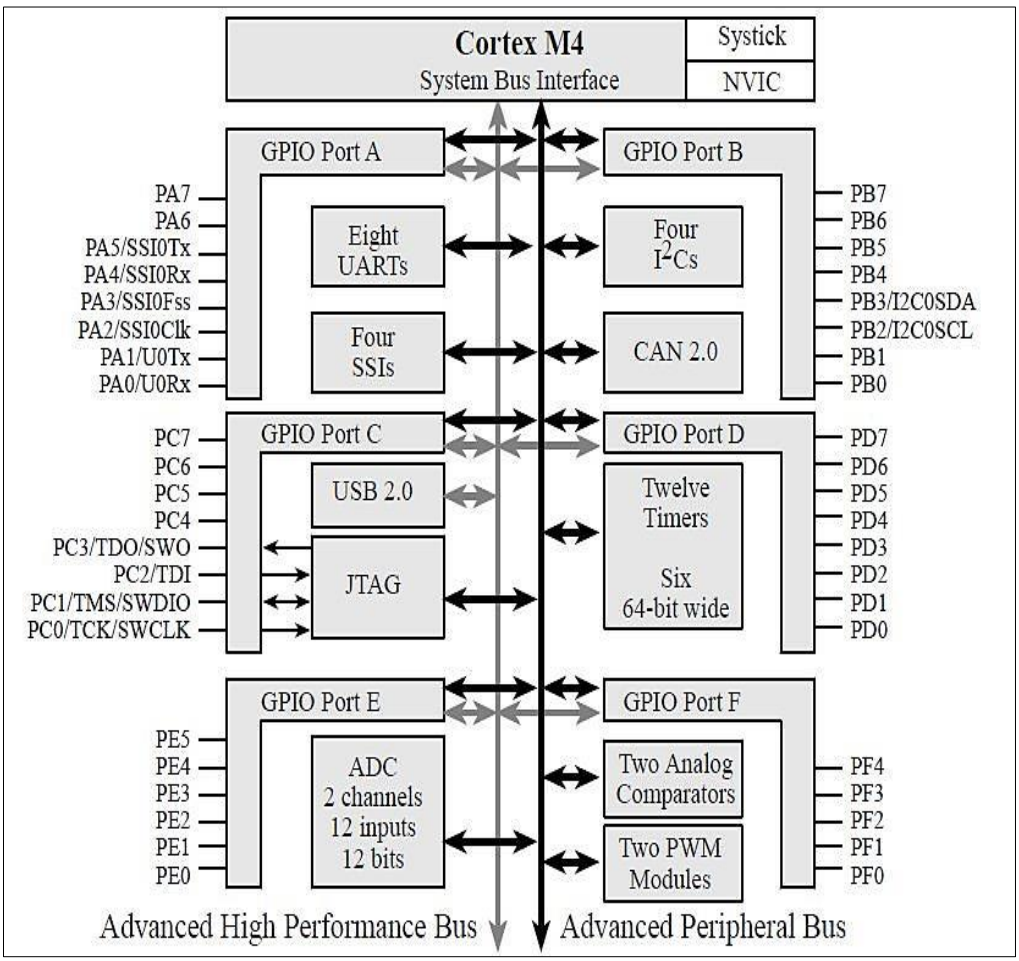
➢ Memory Map and Bus Interface:

- The Cortex-M processors have 32-bit memory addressing bus
  - ✓ The Cortex-M processors can access a memory size (2^ 32-Bit) = **4GB** memory space
- While each manufacturer's Cortex-M device has different peripherals and memory sizes, ARM has defined a basic **memory template** that all devices must adhere to.
- There was no specified memory layout for ARM 7, or ARM 9
- This provides a standard layout so all the vendor provided memory and peripherals are located in the same blocks of memory.
- The Cortex-M memory map has a standard template which splits the 4 GB address range into specific memory regions.
  - ✓ This memory template is common to all Cortex-M devices.
- The Cortex-M memory template defines **eight regions** which cover the 4 GB address space of the Cortex-M processor.
- The **first three regions** are each **0.5 GB** in size and are dedicated to the executable code space, internal SRAM, and internal peripherals.
- The next two regions are dedicated to **external memory (SDRAM) and memory mapped device**, both regions are 1 GB in size
- The final three regions make up the Cortex-M processor memory space and contain the configuration registers for the Cortex-M processor and any vendor-specific registers.
- **Notes:**
  - → The "Addressable memory space" depends on the size of address bus.
  - → The 4GB is called the "Addressable memory space"➔ addressable by the Cortex-M processor.
  - → MCU memory map shows where the (Code – Peripheral – SRAM – …….) are located.
  - → Memory map explains the mapping of different peripheral registers in the processor addressable memory location range.
  - → Multiple bus interfaces to allow concurrent instructions and data accesses (Harvard Arch).
- Any MCU at least has one processor, and in our case it is ARM Cortex-M4
- This processor connected with a lot of peripherals through Buses
- To control a peripherals you need to access a specific register
- A register is just a part of our memory
- We have many buses in the MCU, e.g. a data bus and address bus
- Buses differs on the speed and the used protocol

➢ System architecture for STM32F405xx/07xx and STM32F415xx/17xx devices

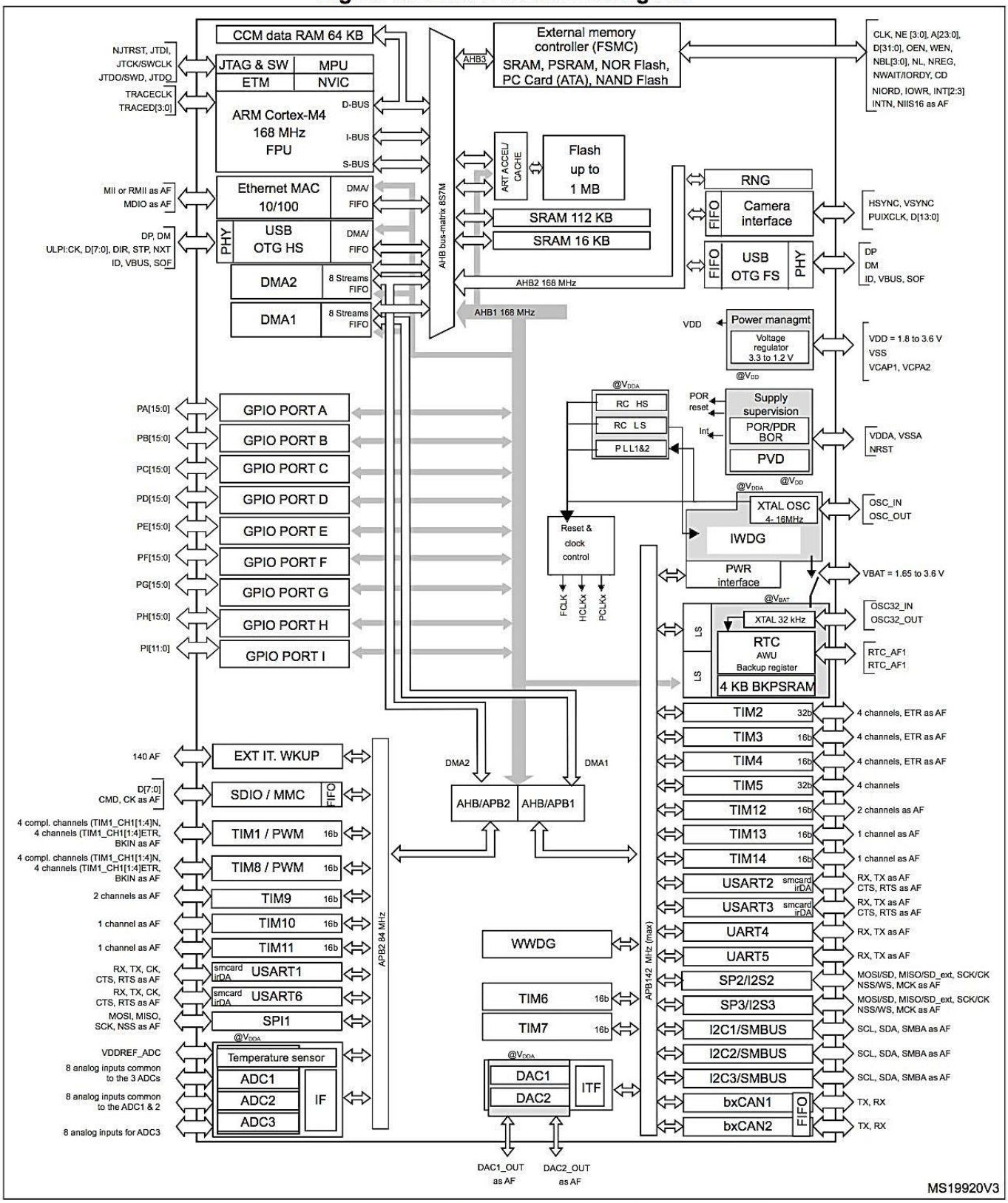



0xE00FFFFF
0xE00FF000
0xE0042000
0xE0041000
0xE0040000
ROM table
External PPB
ETM
TPIU
0xE003FFFF
0xE000F000
0xE000E000
0xE0003000
0xE0002000
0xE0001000
0xE0000000
Reserved
NVIC
Reserved
FPB
DWT
ITM
0x43FFFFFF
0x42000000
0x41FFFFFF
0x40100000
0x40000000
Bit band alias
Bit band region
0x23FFFFFF
0x22000000
0x21FFFFFF
0x20100000
0x20000000
Bit band alias
Bit band region
0xFFFFFFFF
Vendor Specific
Private Peripheral Bus - External
0xE00400000
0xE003FFFF
Private Peripheral Bus - Internal
0xE0000000
0xDFFFFFFF
External Device
1GB
0xA0000000
0x9FFFFFFF
External RAM
1 GB
0x60000000
0x5FFFFFFF
Peripheral
0.5GB
0x40000000
0x3FFFFFFF
SRAM
0.5GB
0x20000000
0x1FFFFFFF
Code
0.5GB
0x00000000


Cortex M4
System Bus Interface
Systick
NVIC
GPIO Port A
GPIO Port B
PA7
PA6
PA5/SSI0Tx
PA4/SSI0Rx
PA3/SSI0Fss
PA2/SSI0Clk
PA1/U0Tx
PA0/U0Rx
Eight UARTs
Four I2Cs
Four SSIs
CAN 2.0
PB7
PB6
PB5
PB4
PB3/I2C0SDA
PB2/I2C0SCL
PB1
PB0
GPIO Port C
GPIO Port D
PC7
PC6
PC5
PC4
PC3/TDO/SWO
PC2/TDI
PC1/TMS/SWDIO
PC0/TCK/SWCLK
USB 2.0
JTAG
Twelve Timers
Six 64-bit wide
PD7
PD6
PD5
PD4
PD3
PD2
PD1
PD0
GPIO Port E
GPIO Port F
PE5
PE4
PE3
PE2
PE1
PE0
ADC
2 channels
12 inputs
12 bits
Two Analog Comparators
Two PWM Modules
PF4
PF3
PF2
PF1
PF0
Advanced High Performance Bus
Advanced Peripheral Bus

2.2 Device overview

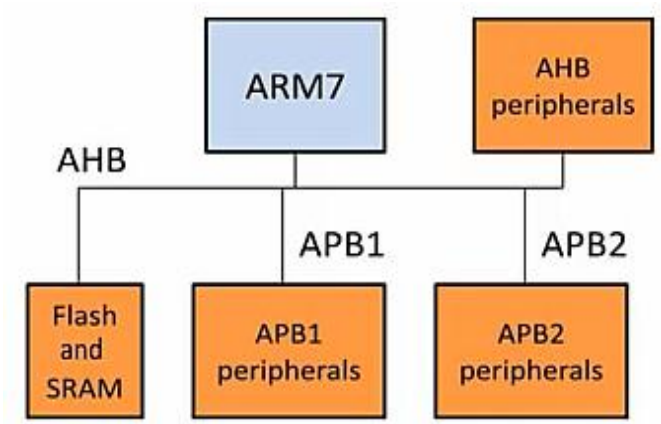
Figure 5. STM32F40x block diagram



1. The timers connected to APB2 are clocked from TIMxCLK up to 168 MHz, while the timers connected to APB1 are clocked from TIMxCLK either up to 84 MHz or 168 MHz, depending on TIMPRE bit configuration in the RCC_DCKCFGR register.
2. The camera interface and ethernet are available only on STM32F407xx devices.

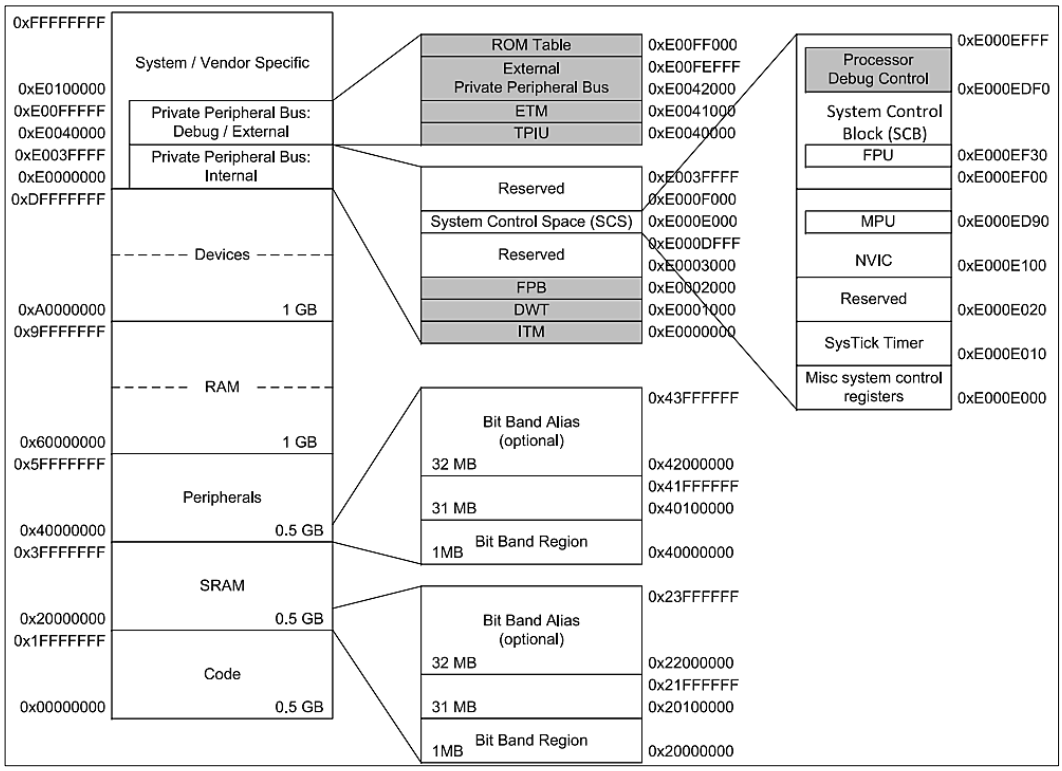
➢ **AHB Lite Bus Interface**

- In Cortex Mx processors the bus interfaces are based on "**Advanced Microcontroller Bus Architecture**" (**AMBA**).
  - ✓ AMBA is a specifications designed by ARM and acts as a **standard for on-chip communication**.
- In first generation of ARM-based microcontrollers the CPU was interfaced to the microcontroller through two types of busses.
  1. **Advanced high speed bus (AHB Lite).**
  2. **Advanced peripheral bus (APB).**
- The **high-speed bus** connected **the CPU to the Flash and SRAM memory** while the microcontroller peripherals were connected to one or more APB busses.
- As multiple bus masters (CPU, DMA) were introduced, a bus arbitration phase had to completed before a transfer could be made across the bus
- The **AHB (AMBA High-performance Bus) Lite protocol** is used for **the main bus interfaces**, and the **APB protocol** is used for the **Private Peripheral Bus (PPB)**, which is mainly used for **debug components**. Additional bus segments based on APB can be added onto the system bus by using additional **bus bridge components**.



➢ The Cortex-M4 processor contains **three external Advanced High-performance Bus(AHB)-Lite Bus**:

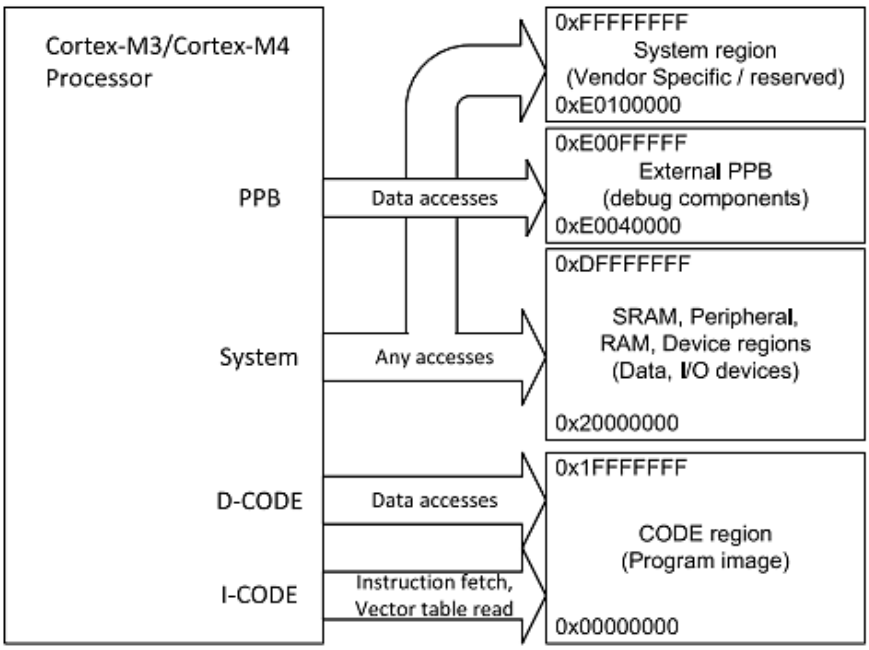
1. **I-Code memory interface(Instruction Bus):**
   - Instruction **fetches from Code memory space**, 0x00000000 to 0x1FFFFFFC, are performed over the 32-bit AHB-Lite bus.
   - The Debugger cannot access this interface.
   - All fetches are **word-wide 32-Bit.**
   - The number of instructions fetched per word depends on the code running and the alignment of the code in memory.
2. **D-Code memory interface(Data Bus):**
   - **Data and debug** accesses to **Code memory space**, 0x00000000 to 0x1FFFFFFF, are performed **over the 32-bit AHB- Lite bus.**
3. **System interface(System Bus):**
   - **Instruction fetches and data and debug accesses** to address ranges 0x20000000 to OxDFFFFFFF and OxE0100000 to OxFFFFFFFF are performed over the 32-bit AHB-Lite bus.



- **Code Region ➔ 0x00000000 to 0x1FFFFFFF**
  - ✓ A 512MB memory space primarily for **program code**, including the **vector table** that is a part of the program memory.
  - ✓ This is the region that the microcontroller vendors should connect their code memory.
  - ✓ This region also allow data accesses to read the (.rodata→ Our constants).
- **RAM Region ➔ 0x20000000 to 0x3FFFFFFF**
  - ✓ The SRAM region is located in the next 512MB of memory space.
  - ✓ It is primarily for connecting SRAM, mostly on-chip SRAM, but there is no limitation of exact memory type.
  - ✓ The first 1MB of the SRAM region is bit addressable if the optional bit-band feature is included.
  - ✓ You can also execute program code from this region.
- **Peripheral Region ➔ 0x40000000 to 0x5FFFFFFF**
  - ✓ The Peripheral memory region also has the size of 512MB, and is use mostly for vendor on-chip peripherals.
  - ✓ The processor peripherals (Ex. NVIC) are and their registers are not part of this region.
  - ✓ Similar to SRAM region, the first 1MB of the peripheral region is bit addressable if the optional bit-band feature is included.
  - ✓ This is execute never "XN" region, you can't execute code from this region, a fault will be triggered if you tried to inject a code to this area.
- **External RAM Region ➔ 0x60000000 to 0x9FFFFFFF**
  - ✓ The RAM region contains two slots of 512MB memory space (total 1GB) for other RAM such as off-chip memories.
  - ✓ The RAM region can be used for program code as well as data.
- **Devices Region ➔ 0xA0000000 to OxDFFFFFFF**
  - ✓ The Device region contains two slots of 512MB memory space (total 1GB) for other peripherals such as off-chip peripherals.

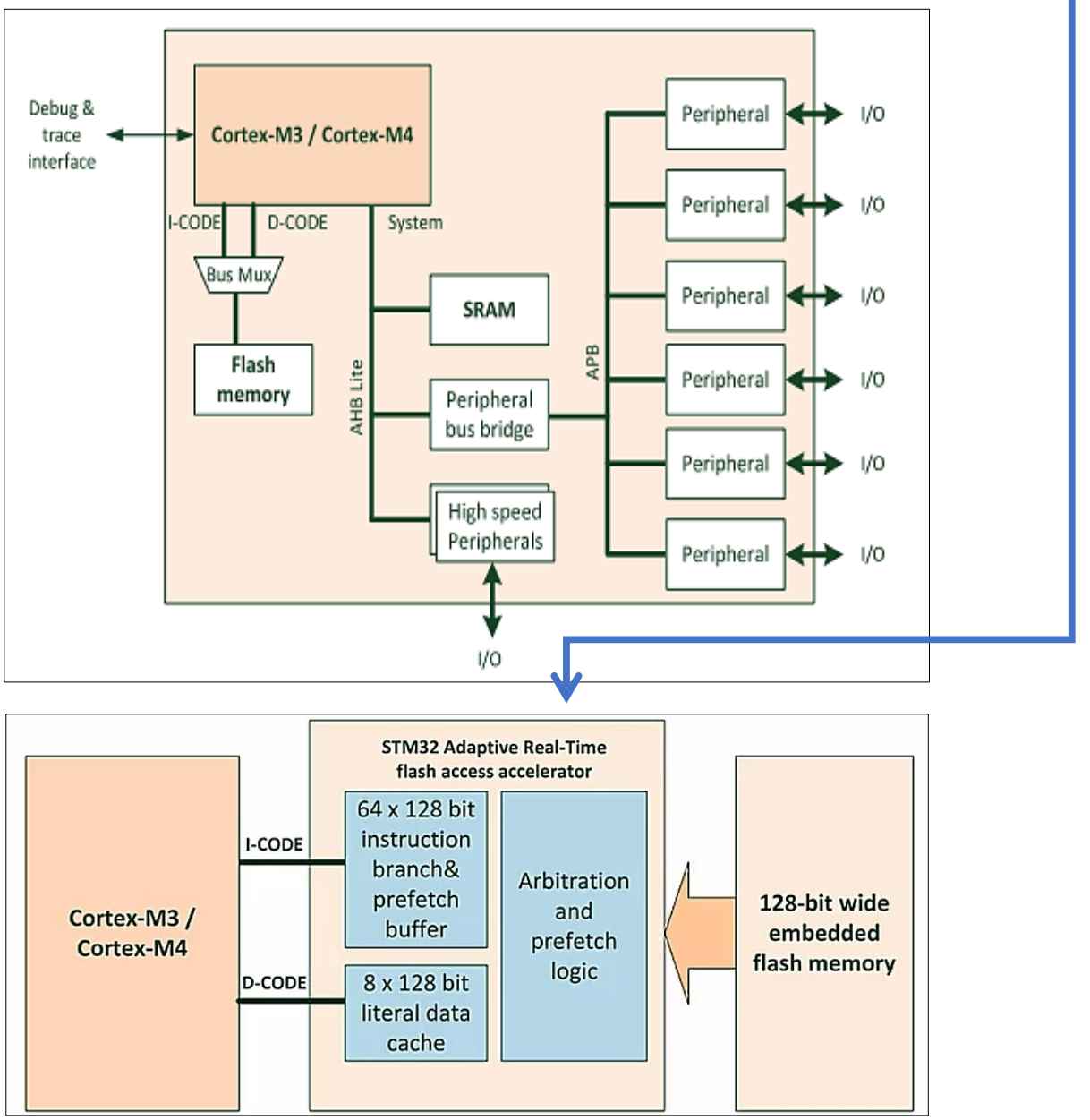
- **System Region ➔ 0xE0000000 to OxFFFFFFFF → Contains several parts**
  - ✓ **Internal Private Peripheral Bus (PPB), 0xE0040000 to OxE00FFFFF→ System Control Space (SCS) area:**
    - The internal Private Peripheral Bus (PPB) is used to access system components such as the NVIC, SysTick, MPU, as well as debug components inside the Cortex-M3/M4 processors.
    - In most cases this memory space can only be accessed by program code running in **privileged state.**
  - ✓ **External Private Peripheral Bus (PPB), OxE0040000 to OXE00FFFFF**
    - An addition PPB region is available for additional optional debug components and so allow silicon vendors to add their own debug or vendor specific components.
    - This memory space can only be accessed by program code running in privileged state.
    - Note that the base address of debug components on this bus can potentially be changed by silicon designers.
  - ✓ **Vendor-specific area, OxE0100000 to 0xFFFFFFFF**
    - The remaining memory space is reserved for vendor-specific components and in most cases this is not used.

- The **AHB (AMBA High-performance Bus) Lite protocol** is used for **the main bus interfaces**, and the **APB protocol** is used for the **Private Peripheral Bus (PPB)**, which is mainly used for **debug components**.
- Additional bus segments based on APB can be added onto the system bus by using additional **bus bridge components**.
- In order to provide better performance, the CODE memory region has **separated the bus interfaces** from the **system bus**, as shown in Figure.
- In this way, data accesses and instruction fetches can be carried out **in parallel**.



- The separate bus arrangement also helps to improve the interrupt responsiveness because during the interrupt handling sequence, stack accesses and reading the vector table in the program image can be carried out at the same time.
- In a simple microcontroller design, typically the program memory is connected to the I-CODE and D-CODE bus, and the SRAM and peripherals are connected to the system bus.

- **Connecting the processor to memory and peripherals:**
  - ✓ Two bus interfaces (I-CODE and D-CODE) are provided for the accesses to program memory.
  - ✓ In simple designs, the two buses can be merged together using a simple bus multiplexer component provided by ARM.
  - ✓ Microcontroller vendors can also utilize these two interfaces to develop customized flash access accelerators to allow the processor to run much faster than to access speed of the flash memory.

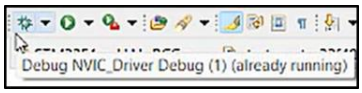


- We can find multiple peripheral bus segments in the design. This allows each bus to be operated at different speeds for best **power optimization**
- Peripheral interfaces are usually based on the **APB protocol**. However, for high performance peripherals, AHB Lite could be used instead for higher bandwidth and operation speed.

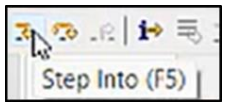
➢ ARM Cortex M4 Debugging Feature

- Debugging is the process of eliminating the bugs/errors in software. Assuming that the hardware is perfect all that remains to check is the software
- The most primitive method of debugging is using LEDs. This is similar to using a printf or a cout statement in c/c++ programs to test if the control enters the loop or not.
- Similarly, an LED blind or a pattern of LED blinks can be used to check if the control enters a particular piece of code.
- Debugging Tools : Software Tools and Hardware Tools.
- Invasive Debug vs. Noninvasive Debug
  - → **Invasive Debug**
    - Invasive debug is defined as a debug process where you can control and observe the processor
    - Mostly used ➔ Halt the processor and modify its state.
  - → **Noninvasive Debug**
    - Noninvasive debug is defined as a debug process where you can observe the processor but not control it.
    - The Embedded Trace Macrocell (ETM) interface and the performance monitor registers are features of noninvasive debug.

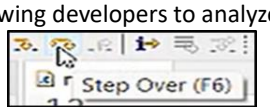
➢ **The Cortex-M3 and Cortex-M4 processors support a wide range of debug features.**



- → **LEDs** : as indicators for specific conditions or events during program execution.
- → **printf** : The processors enable the use of the printf function for debugging purposes. This allows developers to output text messages or variable values to a debug console
  - this way is also called **Semihosting, which** enables code that is running on a target to communicate with and use the Input/Output (I/O) facilities on a host computer. A good example is a program running on a target that uses the printf () function to output a message and the message appears in a debugger console instead of in a target output device or console.

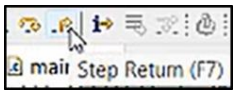


- → **Stepping**
  - **Step into :**
    - ✓ Executes the next instruction and enters any called function, allowing developers to analyze the code line by line, including function calls.



  - **Step over**
    - ✓ Executes the next instruction but does not enter any called function, enabling developers to skip over function calls and focus on the current context.

  - **Single step**
    - ✓ We should first activate the istruction stepping mode
    - ✓ The we can use step into to excute a single assembly instruction
    - ✓ Executes the next assembly instruction and halts, providing an opportunity to examine the effects of that specific instruction.



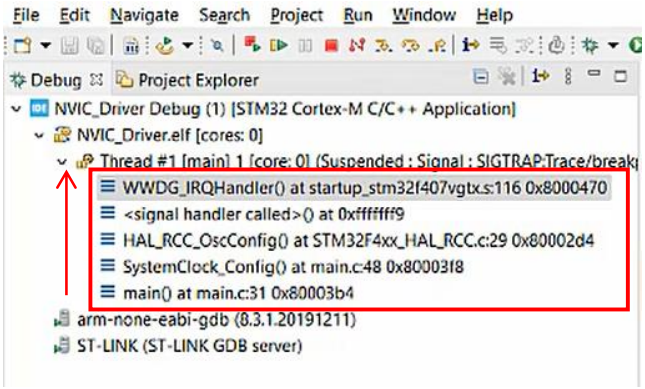
  - **Step return or step out**
    - ✓ Executes instructions until the current function returns and goes out from the function.
- → **Breakpoints**
  - Breakpoints allow developers to pause program execution at specific points in the code(assembly or C code), typically at lines of interest or suspected problem areas. This enables detailed inspection and analysis of the program's state at those breakpoints.

→ **Disassembly**

- The processors provide the ability to view the disassembled instructions, which represent the program's binary code. This feature allows developers to examine the low-level instruction sequence and gain insights into the program's execution flow.

→ **Call stack**

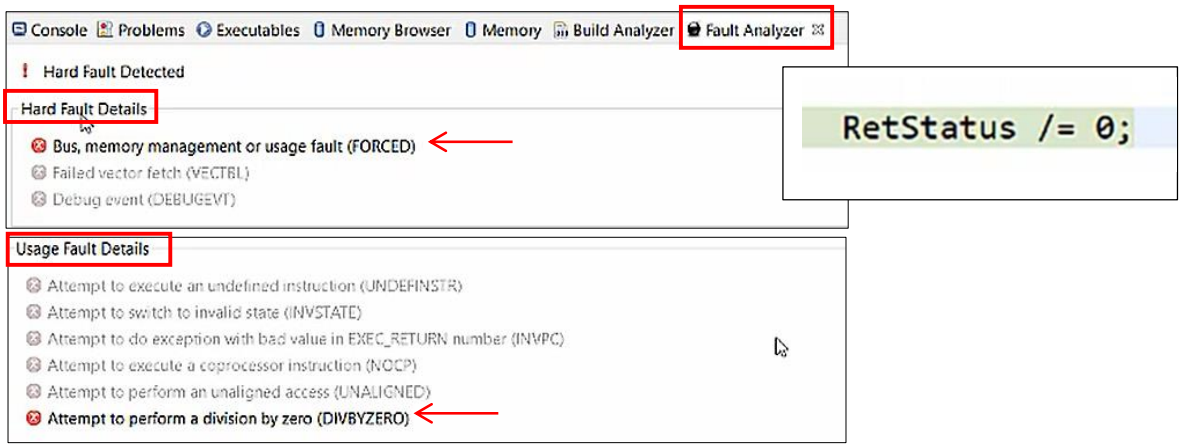
- The processors offer a call stack feature that shows the currently active functions and their order of execution. This helps developers track the program's execution path and identify the sequence of function calls.



- The **main function** ➔ call **SystemClock_Config()** function ➔ call **HAL_RCC_OscConfig()** function and so on.

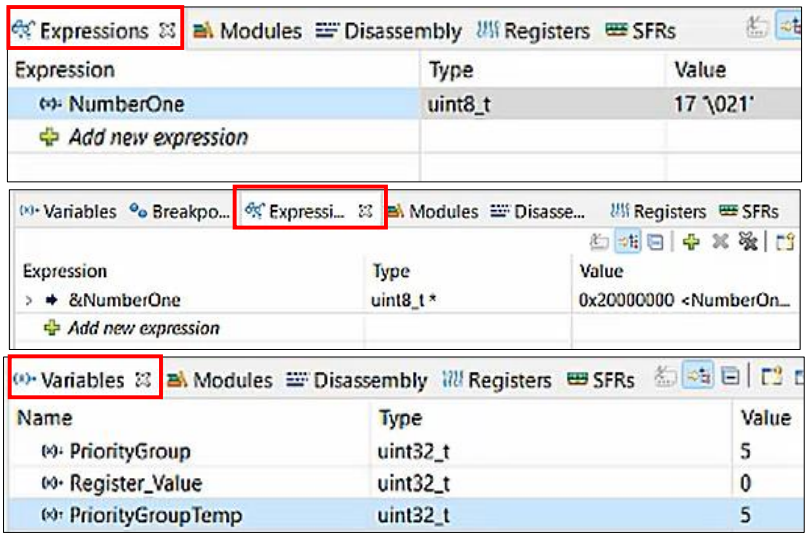
→ **fault analyzer**

- The fault analyzer provides additional information about exceptions or faults that occur during program execution, and takes a snpshot for registers contents during fault exception.



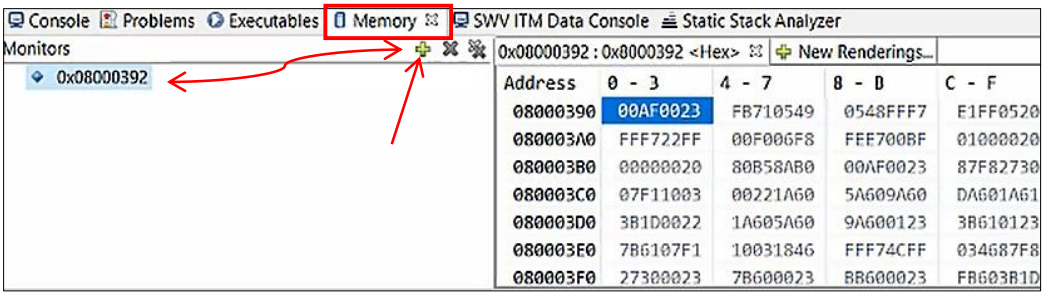
→ **Expressions and variables window**

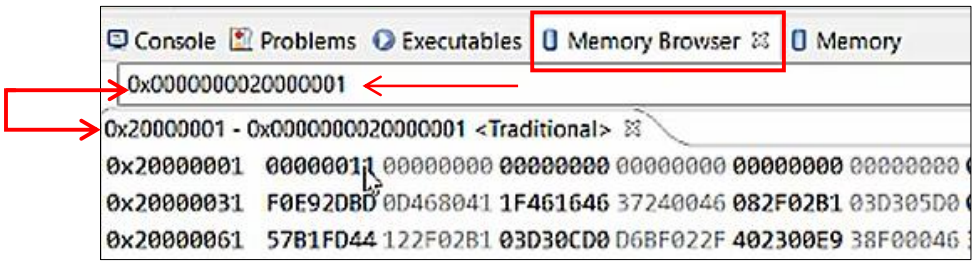
- The processors support an expressions and variables window, which allows developers to monitor and evaluate the values of variables and expressions in real-time during program execution. This helps in tracking the changes in variables and verifying their correctness.
- The variables window ➔ only Local variables anf function parameters



→ **Memory window or memory browser**

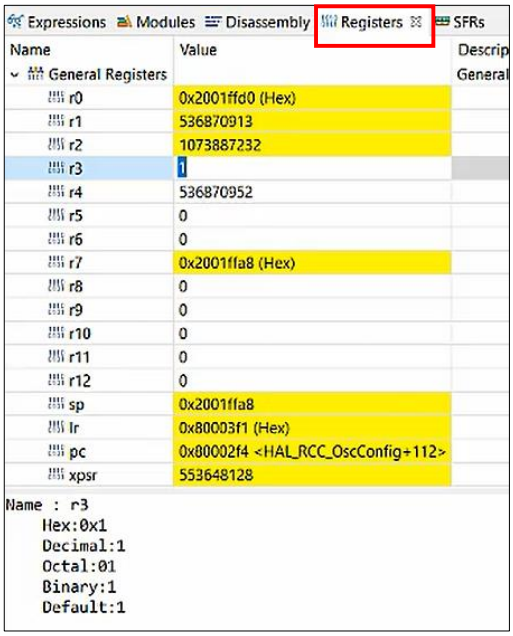
- allows developers to view the contents of specific memory locations in real-time. It provides a convenient interface to observe the values stored in memory.

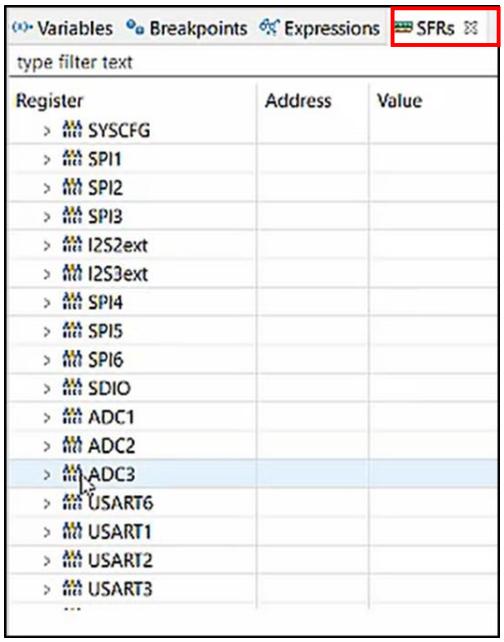


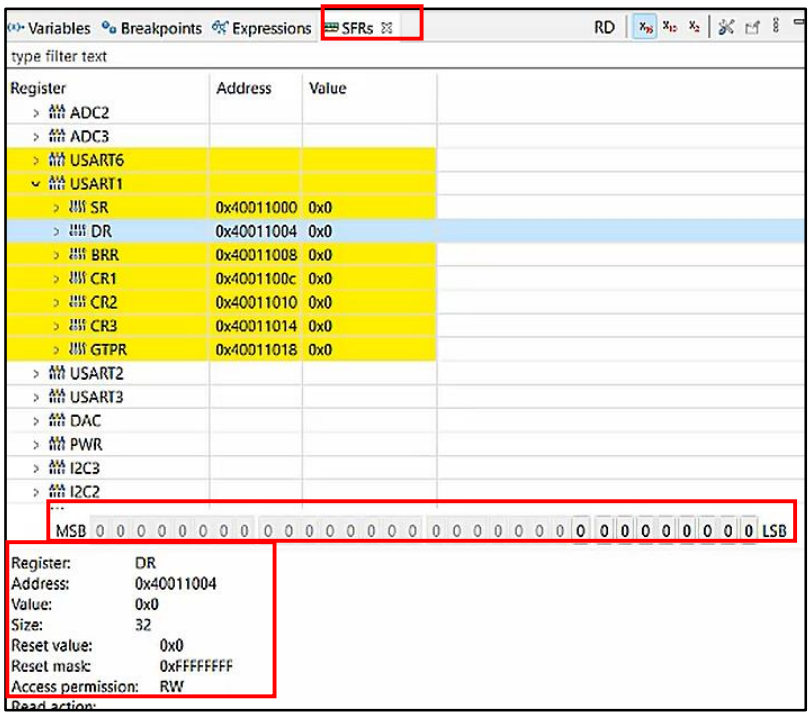


→ **Registers and SFR window**

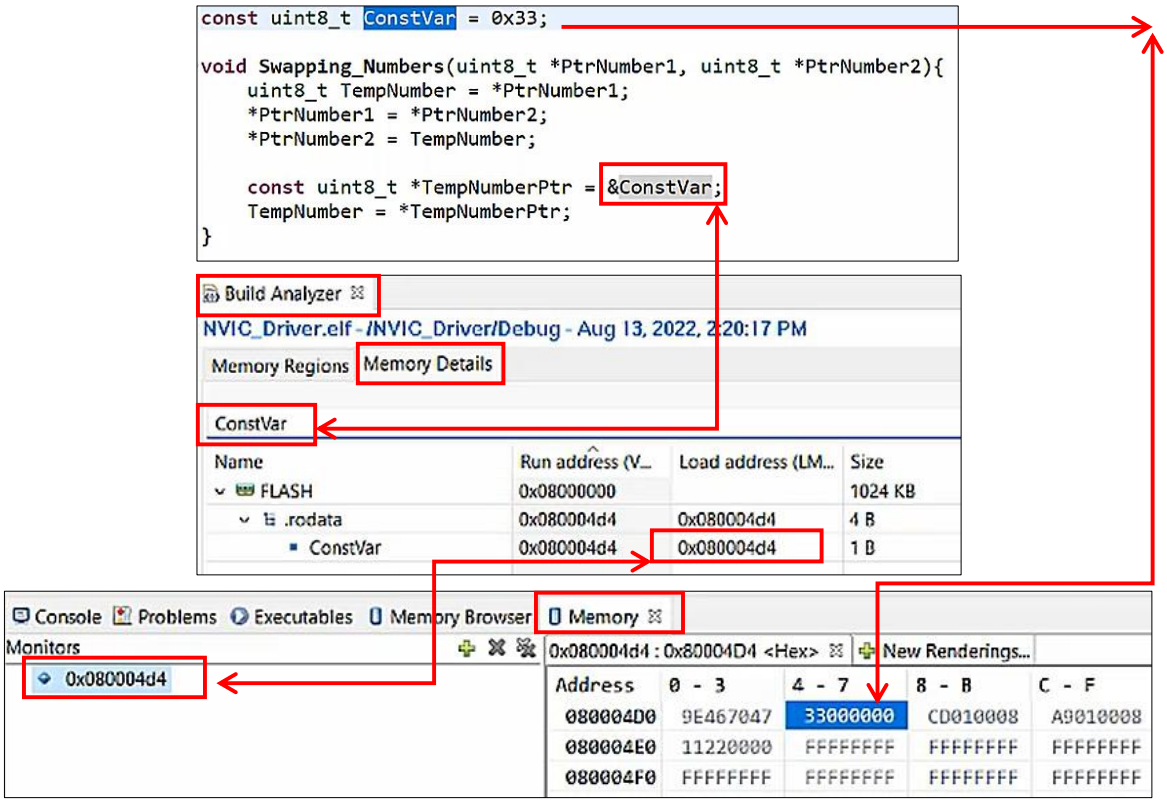
- The processors provide a window **<u>to view and modify</u>** the contents of registers and special function registers. This allows developers to inspect the processor's internal state and interact with specific registers to control the program's behavior.





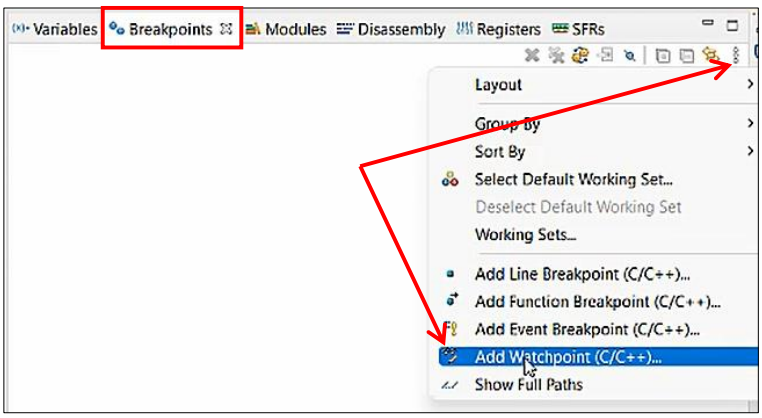


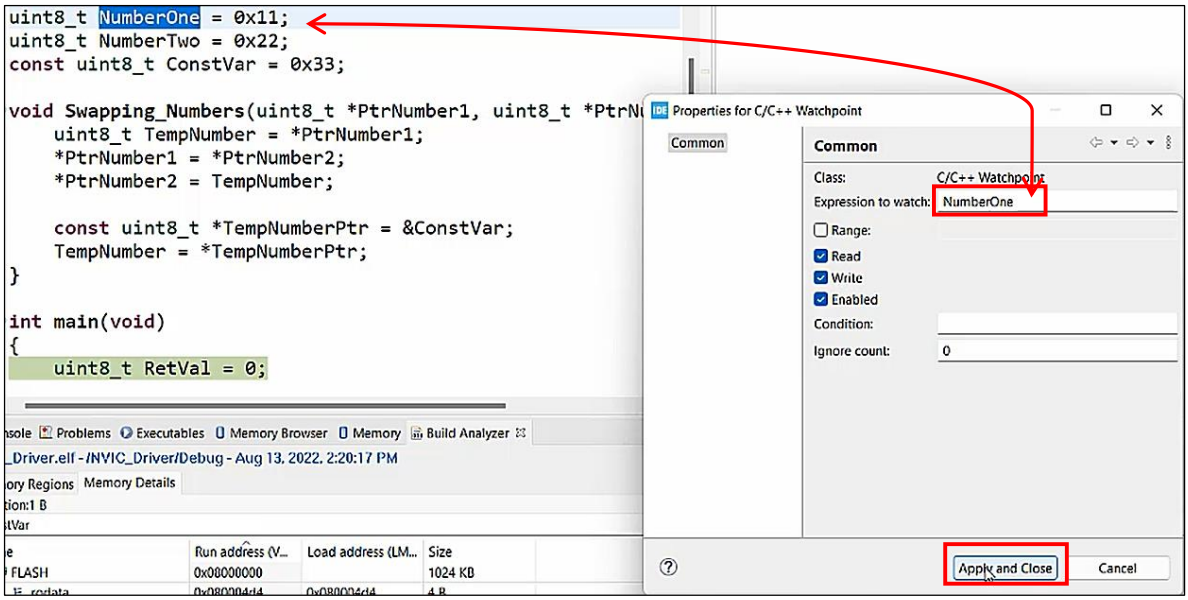
→ **Build Analyzer:**



→ **Data watch point**

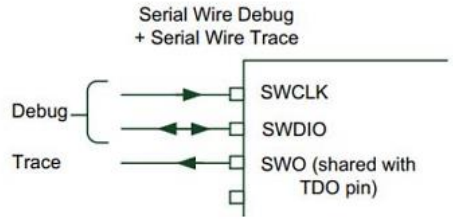
- is a debugging feature that allows developers to monitor the access and modification of a specific variable or memory location during program execution. It enables you to set a watchpoint on a variable or memory address and receive notifications when that variable is read from or written to.
- After setting up the data watchpoint, you can restart the program, and the debugger will provide detailed information about the read or write operations on the variable





- **trace features:**
  - ✓ Trace features on the Cortex-M3 and Cortex-M4 processors use a separate interface to export information during program execution
  - ✓ These features enable real-time (with a small timing delay) data collection without halting the processor, allowing for continuous monitoring and analysis.
  - ✓ There are two main types of trace interface operation modes supported by these processors:
    - **Serial Wire Viewer (SWV):** SWV utilizes a single-pin model and relies on a signal called Serial Wire Output (**SWO**). It provides a relatively simple and cost-effective method for exporting trace information. With SWV, real-time data can be streamed out through the SWO pin, allowing for analysis and debugging without interrupting program execution.
    - **Trace Port Interface:**
      - The Trace Port interface, on the other hand, employs a multi-pin configuration, typically consisting of four data pins and one clock pin. This interface offers a significantly higher bandwidth for transmitting trace data. However, capturing the trace information using the Trace Port interface typically requires a more complex and often more expensive trace adaptor.
      - The 4-pin trace output can be used to provide trace data coming from the Embedded Trace Macrocell (ETM). It is helpful for system development purposes, such as examining timing issues, and for application profiling or performance analysis.
  - ✓ Although the debug and trace interfaces operate independently, they are commonly connected using a single connector. This integration simplifies the setup and allows developers to access both debugging and trace features through a unified interface.
- **SWD** and **JTAG** are popular debugging **interfaces** for those MCU basing on Cortex-M.

- **Debug Interfaces:**
  1. **JTAG:**
     - JTAG is the industry-standard interface used to download and debug programs on a target processor, as well as many other functions. It offers a convenient and easy way to connect to devices and is available on all ARM processor-based devices. The JTAG interface can be used with Cortex-M based devices to access the CoreSight debug capabilities.
  2. **Serial Wire Debug:**
     - The Serial Wire Debug (SWD) mode is an alternative to the standard JTAG interface. It uses only two pins to provide the same debug functionality as JTAG with no performance penalty and introduces data trace capabilities with the Serial Wire Viewer (SWV).
     - The SWD interface pins can be overlaid with JTAG signals, allowing the standard target connectors to be used:
       - TCLK - SWCLK (Serial Wire Clock)
       - TMS - SWDIO (Serial Wire Data Input/Output)
       - TDO - SWO (Serial Wire Output - required for SWV)
         → The Serial Wire Output pin can be used to provide trace data. For connectors that share JTAG and SWD signals, this pin is shared with the TDO signal.

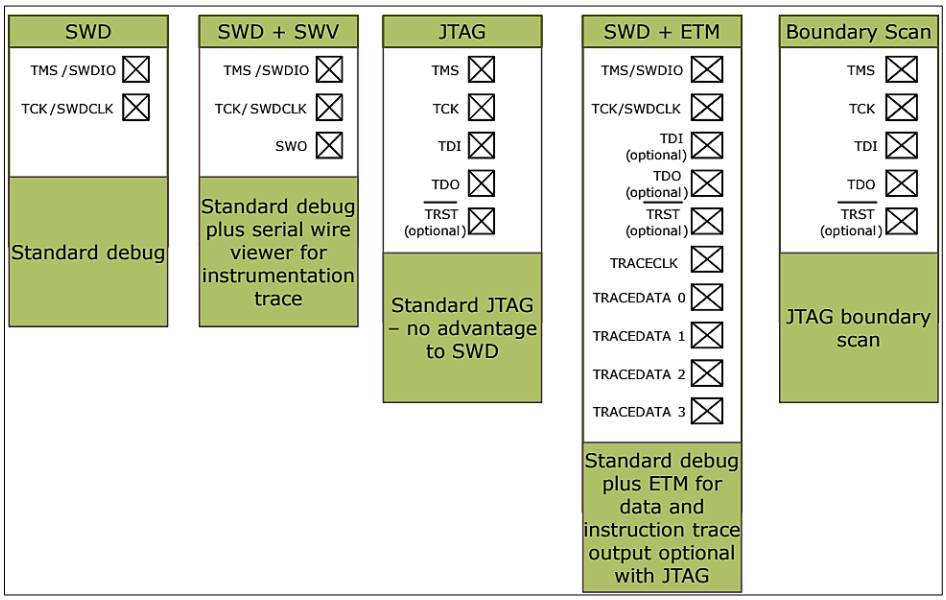


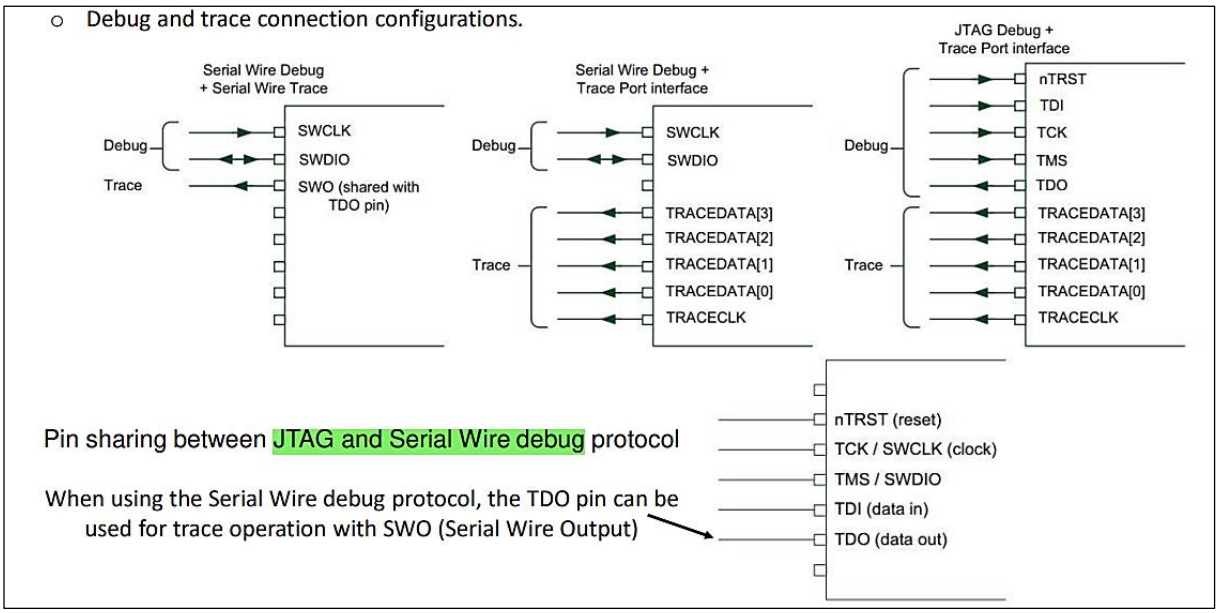
- **Trace Components or interfaces:**
  1. **Embedded Trace Macrocell (ETM):**
     - The Embedded Trace Macrocell (ETM) provides high bandwidth **instruction trace** via four dedicated trace pins accessible on the 20-pin Cortex Debug + ETM connector. This enhanced trace capability records a program's execution instruction-by-instruction
  2. **Instruction Trace Macrocell (ITM):**
     - Trace data generation. This includes:
       → Printf style debugging using the stimulus port registers which generate instrumentation packets.
       → Global and local timestamp packet generation.
       → Synchronization packet generation.
     - Arbitration between trace packets (prioritizing multiple sources and selecting a single source at a time):
       1. External Data Watchpoint and Trace (DWT) packets and internally generated packets.
  3. **Data Watchpoint and Trace (DWT):**
     - The Data Watchpoint and Trace (DWT) unit receives data transactions and instruction execution information from the processor core.
     - Exception information and core profiling information is also delivered to the DWT from the processor core. Any profiling counter and exception information are passed to the packet generator so it can generate, buffer, and arbitrate packets to be sent to the ITM.
     - **The DWT features:**
       - Watchpoints
       - Program Counter (PC) tracing
       - Exception tracing
       - Performance profiling counters
       - Data tracing
       - Cycle count matching
       - Match event tracing

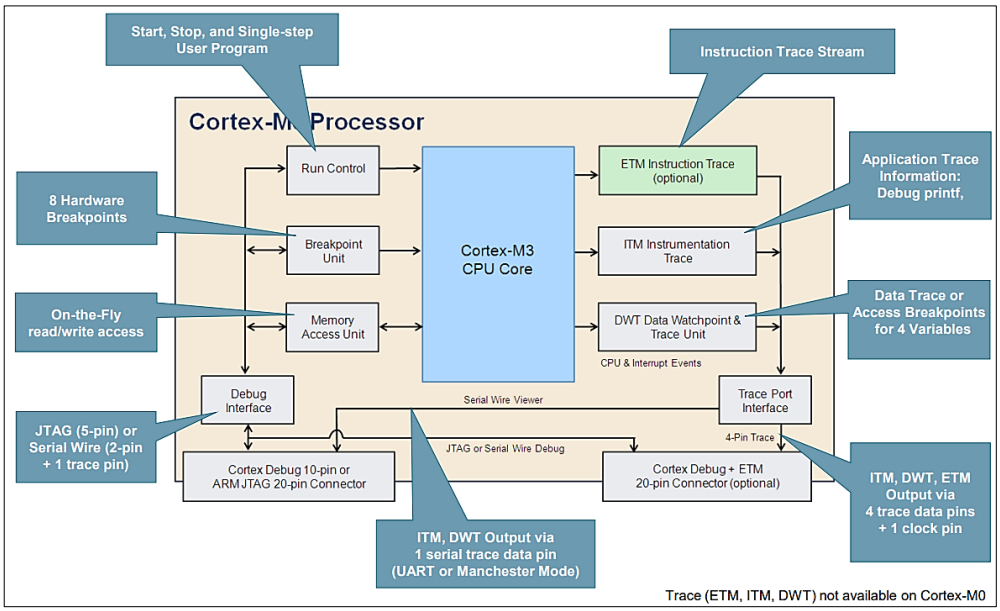
4. **Serial Wire Viewer (SWV):**
    - SWV provides real-time data trace information from various sources within an Armv7-M/Armv8-M device.
    - It is transmitted via the SWO pin while your system processor continues to run at full speed. Information is available from the ITM and DWT units, providing:
    - **Note**: Data trace via SWV is not available using the JTAG interface. SWV is only available with SWD.

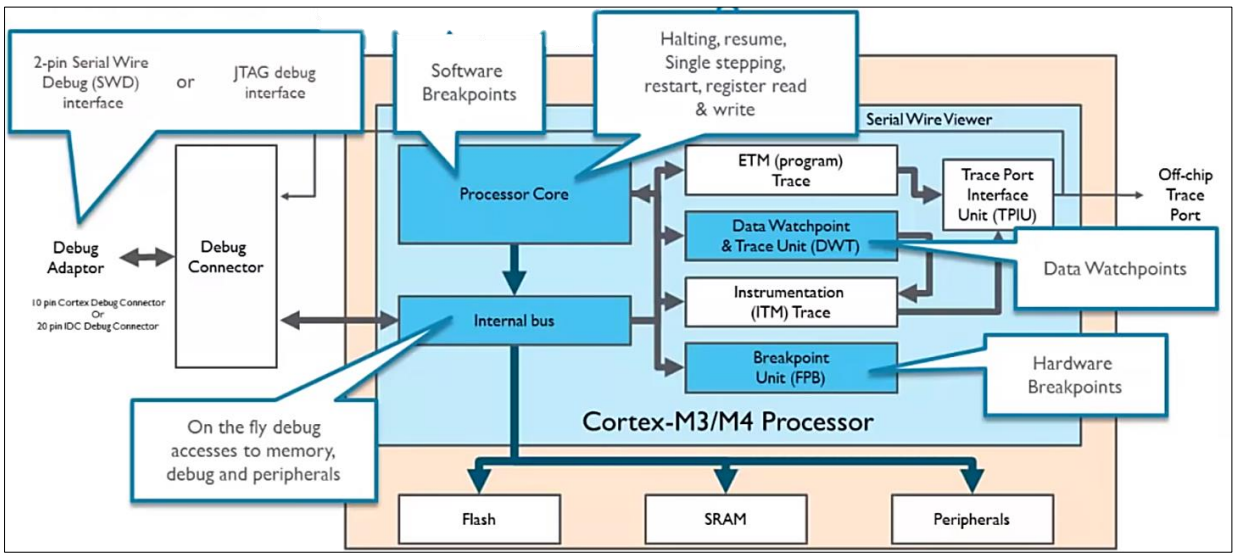
- PC (Program Counter) sampling
- Exception and Interrupt execution with timing statistics
- ITM trace information used for simple printf-style debugging
- Event counters that show CPU cycle statistics
- Trace data - data reads and writes used for timing analysis

- **Debug and trace connection configurations:**



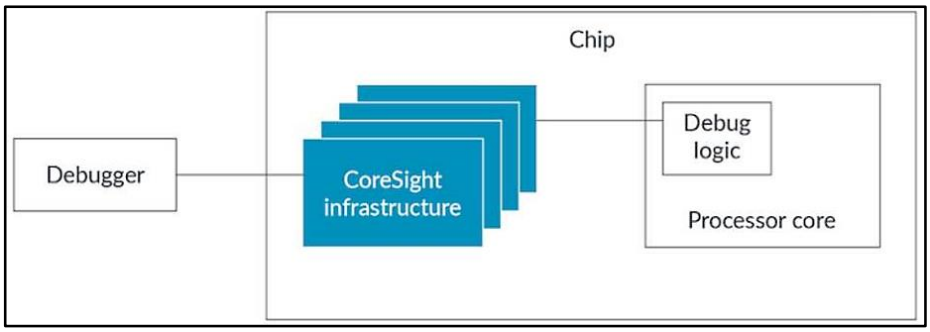




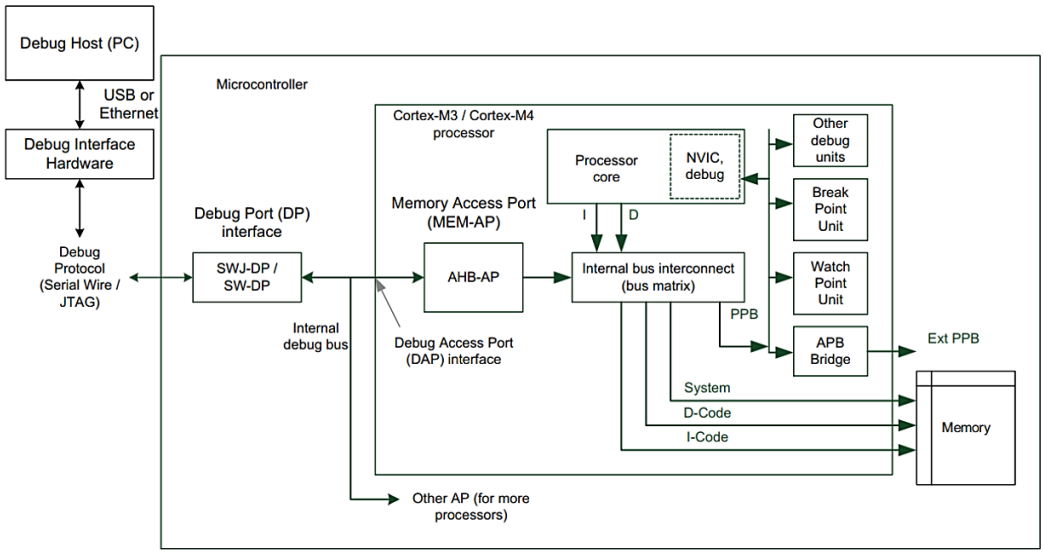


- Every ARM processor core has some debug functionality that is built in and this debug functionality is used in two general modes:
  - → **Self-hosted Debug,** in which code that is running on the processor core uses the debug capability to identify problems in the software
  - → **External Debug,** in which an external debugging component, the debugger, can access the debug features and use them to identify problems with the software.
    - For the external component to access the built-in debug capability of the processor, a debug infrastructure must be built into the chip.
    - This infrastructure provides the connectivity between the debugger and the processor.
    - The mechanism that ARM uses to provide this debug infrastructure is based on the CoreSight architecture.
    - The CoreSight architecture defines a set of capabilities that can be designed into a processor or system level components.
    - The system level capabilities allow a debugging component to access and use the processor debug and trace capabilities.
    - ARM has developed a set of components that are based on this architecture.

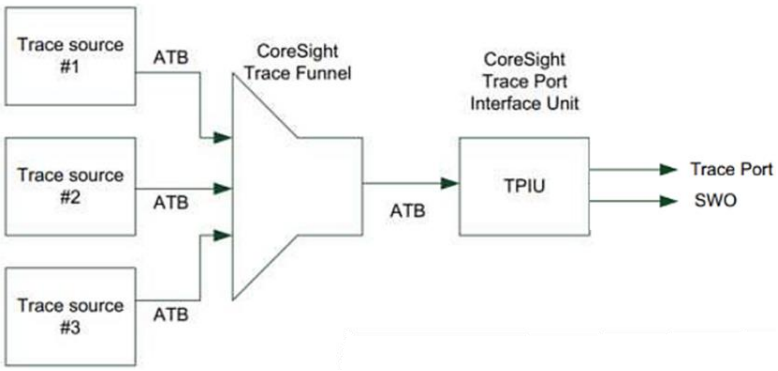
- These components are used to create a customized debug infrastructure for a device, and are delivered in the CoreSight SOC products.



- The debug and trace features of the Cortex-M processors are designed based on the CoreSight Debug Architecture.
- This architecture covers a wide area, including debug interface protocols, on-chip bus for debug accesses, control of debug components, security features, trace data interface, and more.
- For normal software development, it is not necessary to have an in-depth understanding of CoreSight technology.
- One important aspect of the debug support in the Cortex-M processors is the support for multiple processor designs.
  - **The CoreSight** architecture allows the sharing of debug connections and trace connections.
  - By default, the Cortex-M3 and Cortex-M4 processors have a pre-configured processor system for single core environments, and the system can be modified to support multi-processor designs by adding additional CoreSight debug and trace components from ARM
- **Connection from debug host to the Cortex-M processor:**



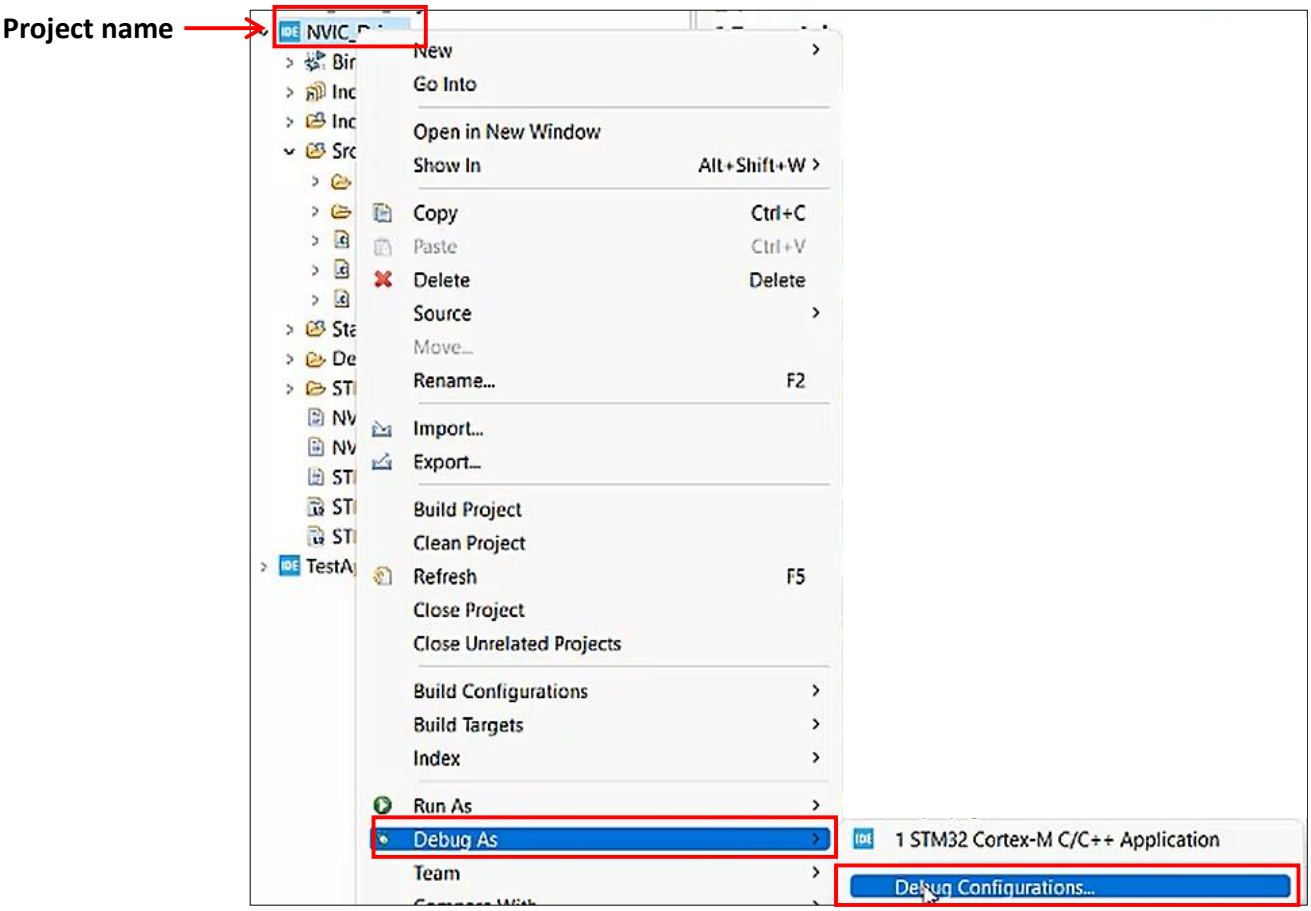
- During tracing, the trace data in the form of packets are output from the trace sources using a set of on-chip trace data bus called Advanced Trace Bus (ATB).
- Typical trace stream merging in CoreSight systems.

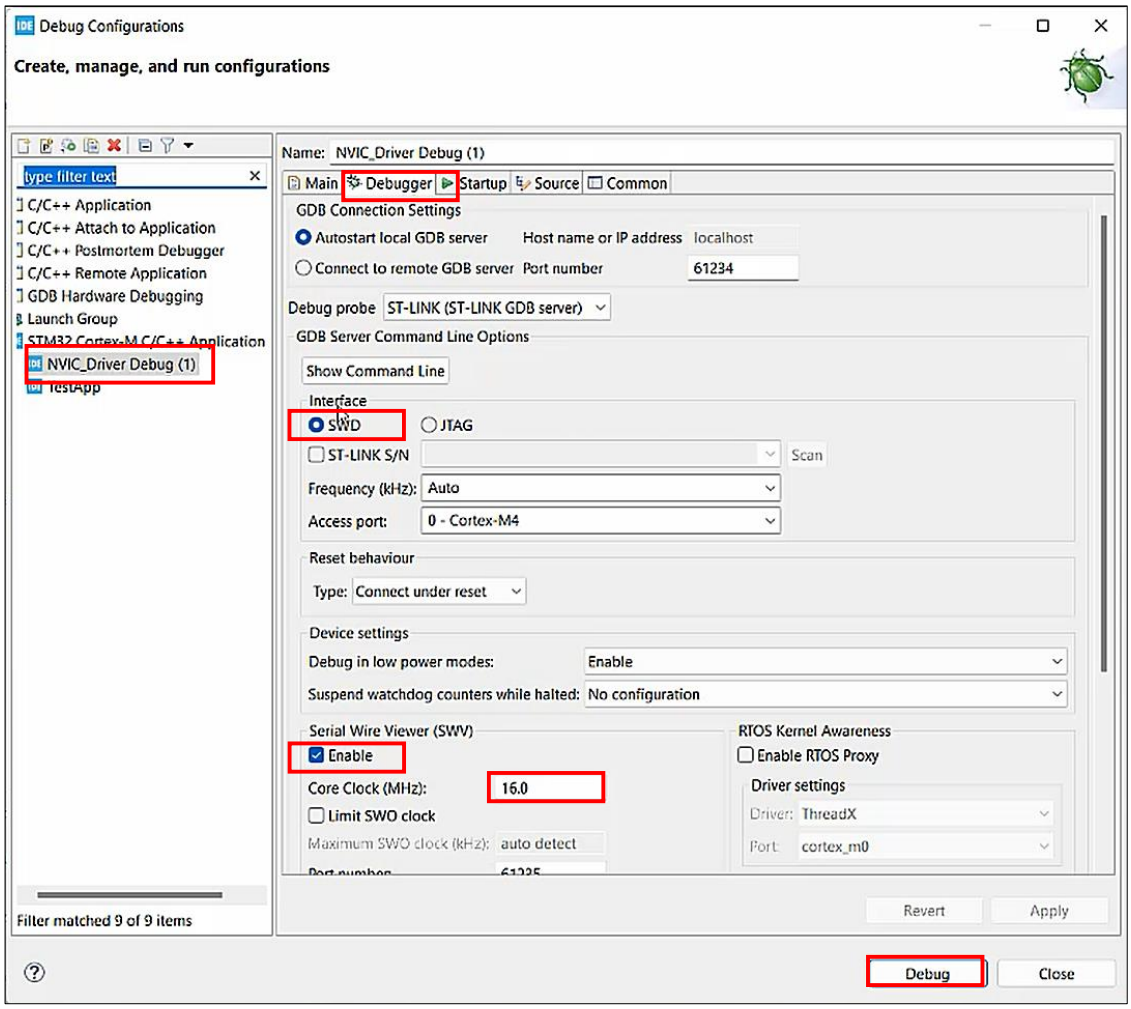


- **Getting Started with Instrumentation Trace Macrocell in STM32CubeIDE:**
- the core clock configuration of the MCU will be at its default configuration, and here is the list of default core clock frequency for different STM32 series of MCUs,

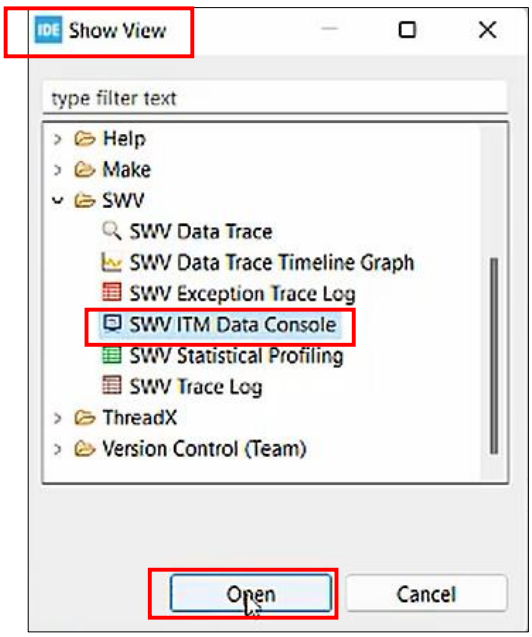
Core	Model	Default Core Clock Frequency (MHz)
Cortex M3	STM32F1x	8
Cortex M4	STM32F3x	8
Cortex M4	STM32F4x	16
Cortex M4	STM32L4x	4

- Once we know the core clock frequency, now it's time for setting up debug configuration. Navigate to the menu as shown below, and enable the SWV mode, also enter the proper core clock frequency.

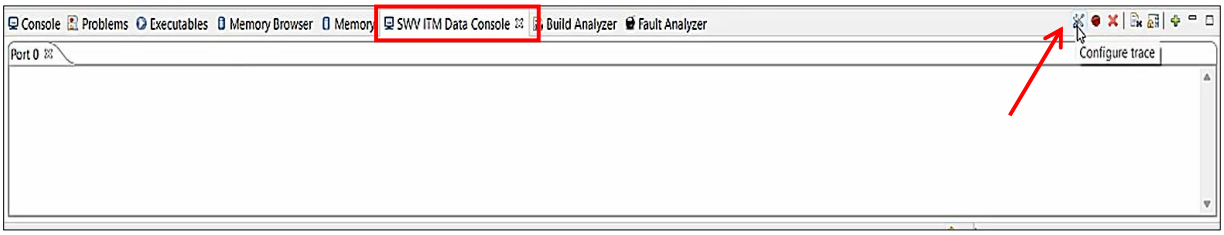




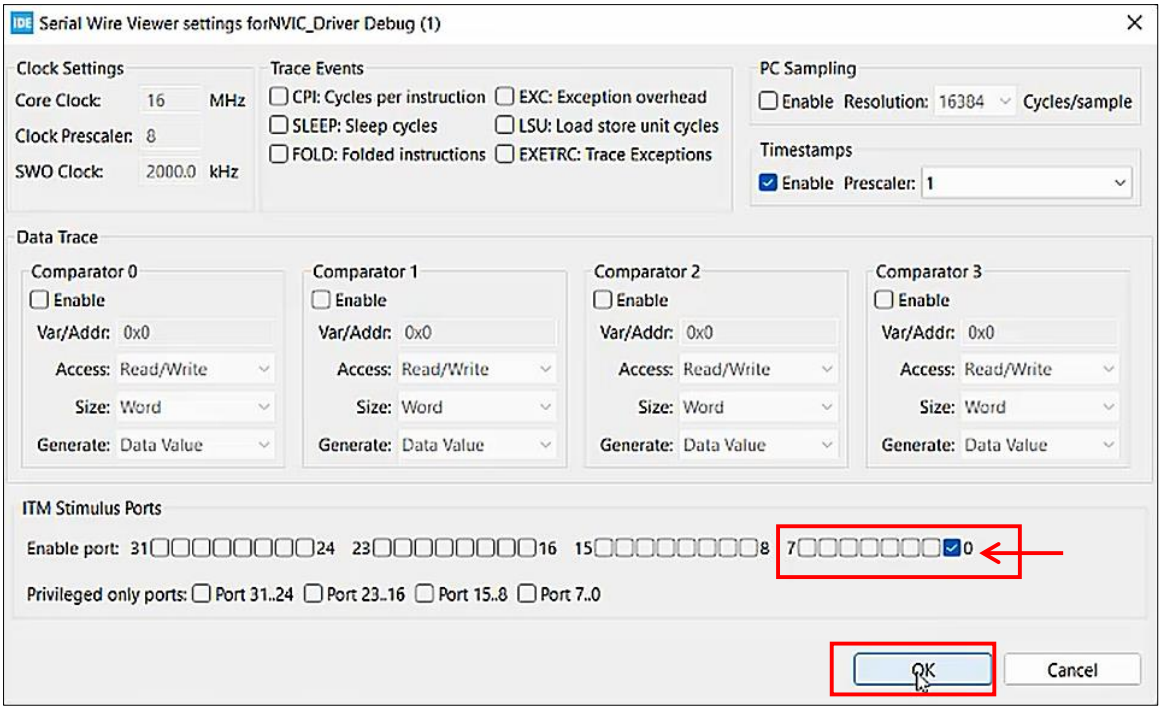
- Once you started debugging, you will navigate to the SWV ITM Data Console menu as shown below.



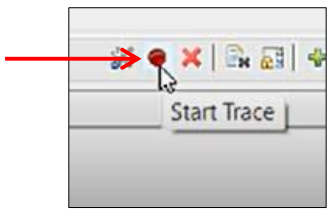
- Once you click it, a tab “SWV ITM Data Console” will get displayed at the bottom of workspace, now click configure trace.



- A new panel will open, in that, check the port 0 checkbox and click OK.

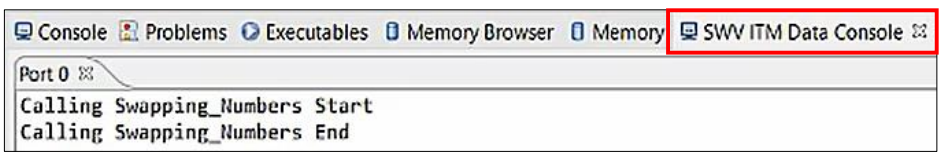


- At last, start the trace by click the red button.

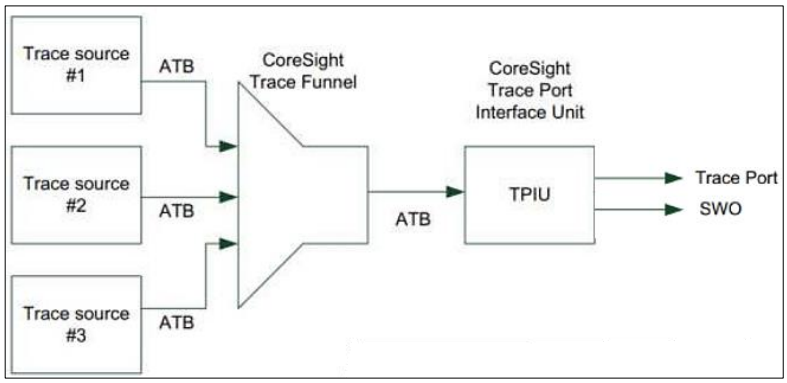


- We are done with the entire configuration. Now you can start debugging.

```
void Swapping_Numbers(uint8_t *PtrNumber1, uint8_t *PtrNumber2){
    printf("Calling Swapping_Numbers Start \n");
    uint8_t TempNumber = *PtrNumber1;
    *PtrNumber1 = *PtrNumber2;
    *PtrNumber2 = TempNumber;
    printf("Calling Swapping_Numbers End \n");
}
```

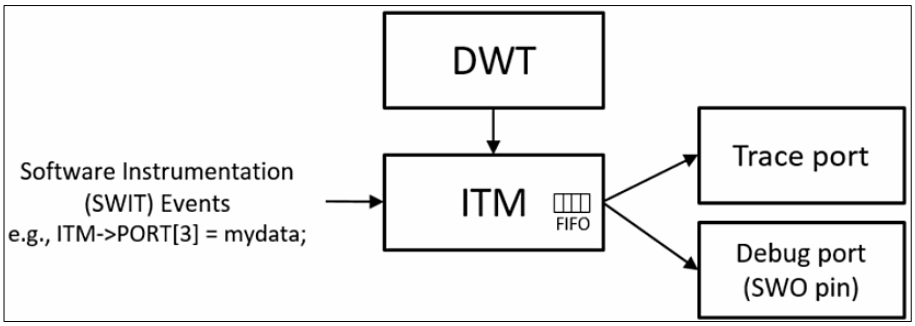


- **Trace Interface:**
- Another part of the CoreSight architecture concerns trace data handling.
- In the Cortex-M4 processor, there can be three types of trace source:
  - → **Embedded Trace Macrocell (ETM)**
    - This optional component generates instruction Trace.
  - → **Data Watch point and Trace (DWT)**
    - This unit can be used to generate data trace, event trace, and profiling trace information.
  - → **Instrumentation Trace Macrocell (ITM)**
    - This allows software-generated debug messages such as using printf, and is also used to generate timestamp information.
- During tracing, the trace data in the form of packets are output from the trace sources using a set of on-chip trace data bus called Advanced Trace Bus (ATB).
- Typical trace stream merging in CoreSight systems



- Based on the CoreSight architecture, trace data from multiple trace sources (e.g., multiple processors) can be merged using an ATB merger hardware called the CoreSight Trace Funnel.
- The merged data can then be converted and exported to the trace interface of the chip using another module called the Trace Port Interface Unit (TPIU).
- Once the converted data is exported, it can be captured using trace capturing devices and analyzed by the debug host (e.g., a personal computer).
- The data stream can then be converted back into multiple data streams.
- In the Cortex-M4 processor designs, in order to reduce the overall silicon size, the arrangement of the trace system is a bit different.
- The Cortex-M4 TPIU module is designed with two ATB ports so that there is no need to use a separate trace funnel module.
- Also it supports both Trace Port mode and SWV mode (use SWO output signal), whereas in CoreSight systems the SWV operation requires a separate module.
- When developing embedded software, understanding the real-time behavior is crucial.
- Diagnostic logging or tracing is necessary, but using traditional logging methods like "printf" calls over a serial port can introduce significant overhead.
- The Instrumentation Trace Macrocell (ITM) module, available in ARM Cortex-M3, Cortex-M4, and Cortex-M7 cores, provides a solution for efficient diagnostic logging and tracing.

- The ITM module can transfer two main types of data: debug events generated by the DWT unit and software instrumentation (SWIT) events.
- Debug events include exception events and data watchpoint events, which can be captured and transmitted by the ITM module for analysis and debugging.
- SWIT events are custom data logged by your code, and the ITM module allows you to transmit this data by writing it to a memory-mapped register on the ARM-based MCU.
- With a fast debug probe, the ITM module can transmit data quickly, typically in just a few clock cycles.
- The ITM module supports automatic time-stamping of the logged data using hardware capabilities.
- Many integrated development environments (IDEs) offer features to view ITM data in a debug window or write it to a file for later analysis.
- Leveraging (Benefit) the ITM module and a fast debug probe enables efficient diagnostic logging and tracing without significant disruption to the real-time behavior of the embedded software.



- The ITM unit provides 32 logical channels for SWIT events, each with a corresponding stimulus register where it accepts input.
    - → These channels allow for separating the diagnostic data into different categories.
    - → For instance, ARM recommends channel **0 for text data** (e.g., from printf) and channel **31 for RTOS events**, while the other channels can be used for whatever purpose you like.
    - → The ITM channels share a **common FIFO buffer** in the ITM unit that in turn is connected to one or two output ports.
    - → The ITM data is included with the instruction trace (ETM) if using the full trace port together with an advanced trace debugger, but it is also available via the commonly available Serial Wire Out (SWO) interface in the debug port.
    - → The SWO interface is supported by most debug probes targeting ARM MCUs
    - → The ITM FIFO buffer is pretty small, only 10 bytes, so if using a slow debug probe some data might be lost if writing too frequently to the ITM ports.
    - → This can be prevented by checking if the ITM FIFO has room for additional data before writing it and delay the write in case there is no room.
    - → This way may however cause a significant impact on the timing of your system, if your debug probe is too slow in receiving the data

---

Memory System [Stack Memory]

→ Stack is a kind of memory usage mechanism that allows a portion of memory to be used as Last-In-First-Out data storage buffer.

→ Stack memory is part of the main memory (Internal RAM - External RAM).

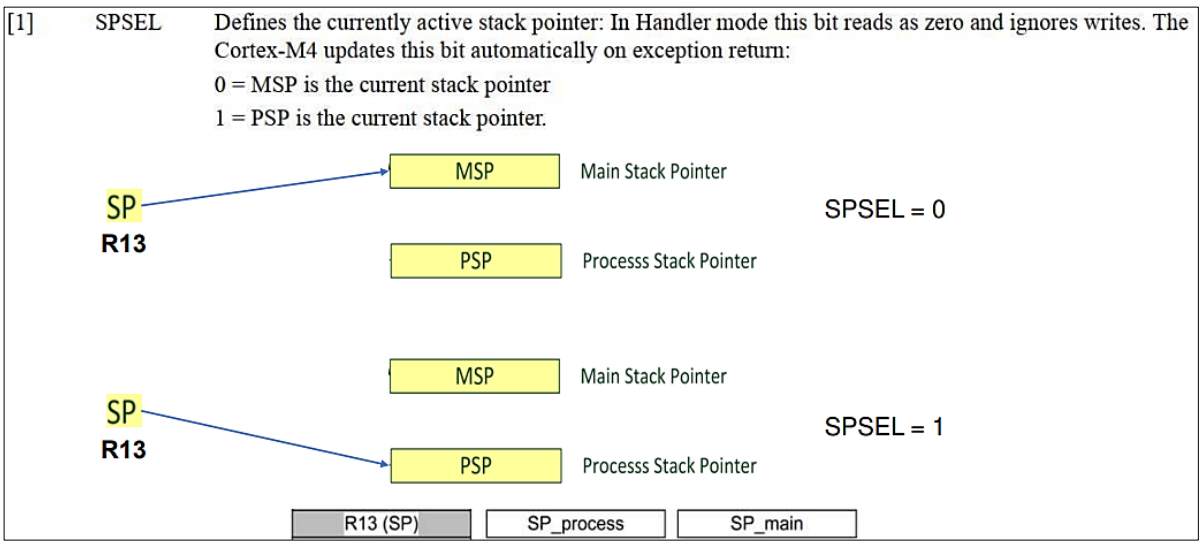
→ ARM processors use the main system memory [Usually Internal SRAM] for stack memory operations.

→ Stack memory mainly used during function calls, interrupt handling - exceptions handling.

→ Stack memory can be accessed using PUSH and POP instructions or using any memory manipulation instructions (LD, STR).

→ One of the essential elements of stack memory operation is a register called the Stack Pointer.

- The stack pointer indicates where the current stack memory location is, and is adjusted automatically each time a stack operation is carried out.
- In the Cortex-M processors, the Stack Pointer is register R13 in the register bank.
- Physically there are two stack pointers in the Cortex-M processors, but only one of them is used at a time, depending on the current value of the CONTROL register and the state of the processor.

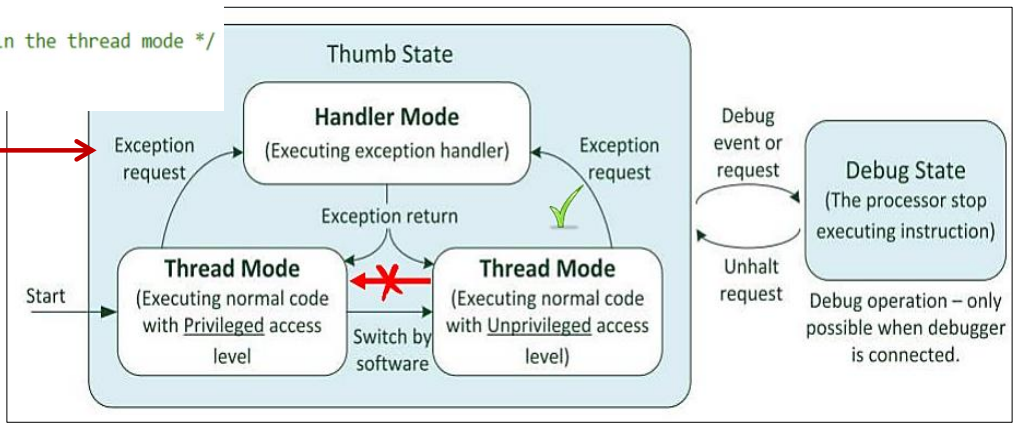


→ After reset the SP (R13) will use the MSP (**Default stack pointer**).

→ In "Thread Mode" we can change the SP (R13) will use the MSP or PSP using the (SPSEL) bit.

→ In "Handler Mode" SP (R13) will always use the MSP and we can't change it. Any change will be ignored.

```
void GenerateException(void){
    /* SVC Instruction can be called in the thread mode */
    __asm volatile ("SVC #0x2");
}
```



→ As mentioned previously, physically there are two stack pointers in the Cortex-M processors.

1. Main Stack Pointer (MSP), this is the default stack pointer used after reset, and is used for all exception handlers.
2. Process Stack Pointer (PSP), this is an alternate stack pointer that can only be used in thread mode.
    → It is usually used for application tasks in embedded systems running an embedded OS.

→ The selection between MSP and PSP can be controlled by the value of SPSEL in bit 1 of the CONTROL register.

		31:3	2	1	0
Cortex-M3 Cortex-M4	CONTROL			SPSEL	nPRIV

[1] SPSEL Defines the currently active stack pointer: In Handler mode this bit reads as zero and ignores writes. The Cortex-M4 updates this bit automatically on exception return:
0 = MSP is the current stack pointer
1 = PSP is the current stack pointer.

→ In addition, during exception return from Handler mode to Thread mode:

- The selection can be controlled by the value of **EXC_RETURN** (Exception Return) value.
- In that case the value of SPSEL will be updated by the processor hardware accordingly.

→ The stack is traced using a stack pointer (SP) register.

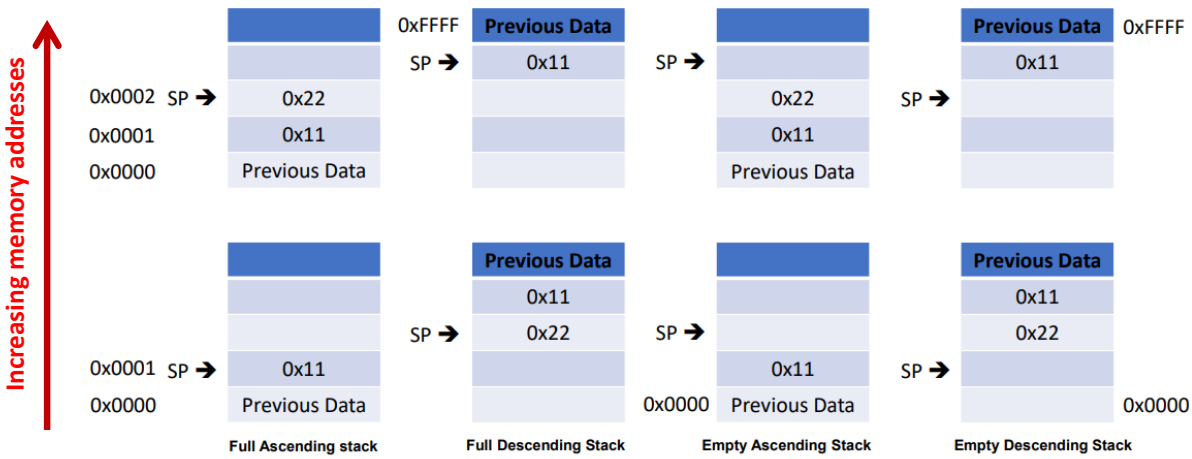
- The PUSH and POP instructions affect (decrement or increment) the SP ➔ R13.

→ **Main stack memory usage:**

- Temporary storage of original data (**transient data**) when a function being executed needs to use registers (in the register bank) for data processing.
    → Values can be restored at the end of the function so the program that called the function will not lose its data.
    → The temporary storage of processor register values
    → Any temporary storage of "local variables of the function" when a function being executed.
    → Passing of information to functions or subroutines (Function parameters).
    → During system exceptions or interrupts to save the context (status register & register values).

→ **Stack operation Models:**

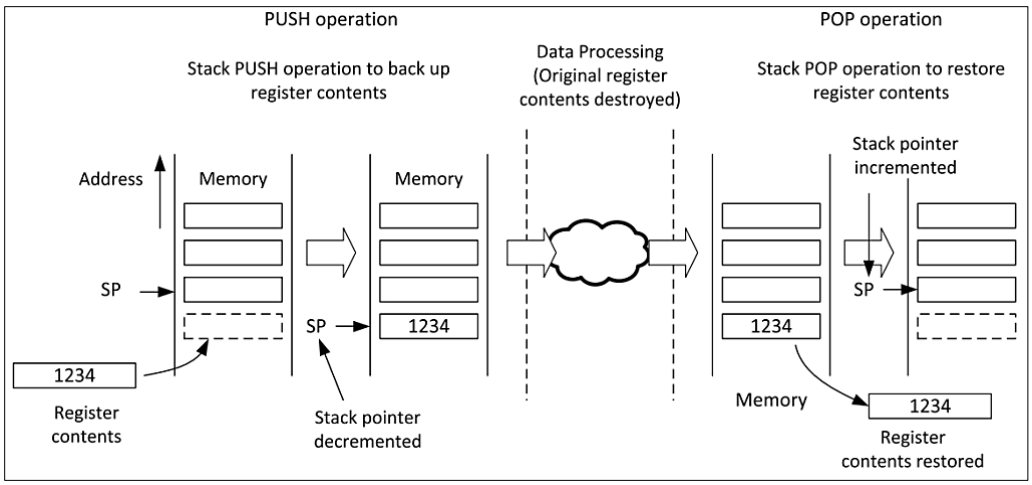
- Full Ascending stack (SP will point to the current used location "Stack Top")
- Full Descending stack (SP will point to the current used location "Stack Top")
- Empty Ascending stack (SP will point to the current empty location)
- Empty Descending stack (SP will point to the current empty location)



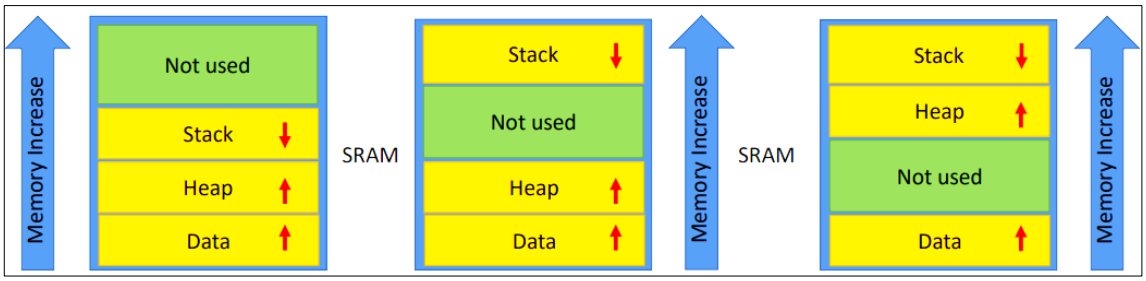
➢ **Cortex-M processors use the "Full Descending Stack" model.**

→ Each PUSH operation, the processor first decrements the SP, then stores the value in the memory location pointed by SP and during operations, the SP points to the memory location where the last data was pushed to the stack

→ Each POP operation, the value of the memory location pointed by SP is read, and then the value of SP is incremented automatically



➢ **Stack Placement:**



→ Stack placement is an important consideration when allocating memory in a program.

→ There are different ways to allocate the stack, such as starting it at the end of RAM.

→ The linker script determines the boundaries for stack placement and other memory segments.

→ The stack is typically placed at the end of RAM, descending towards lower memory addresses.

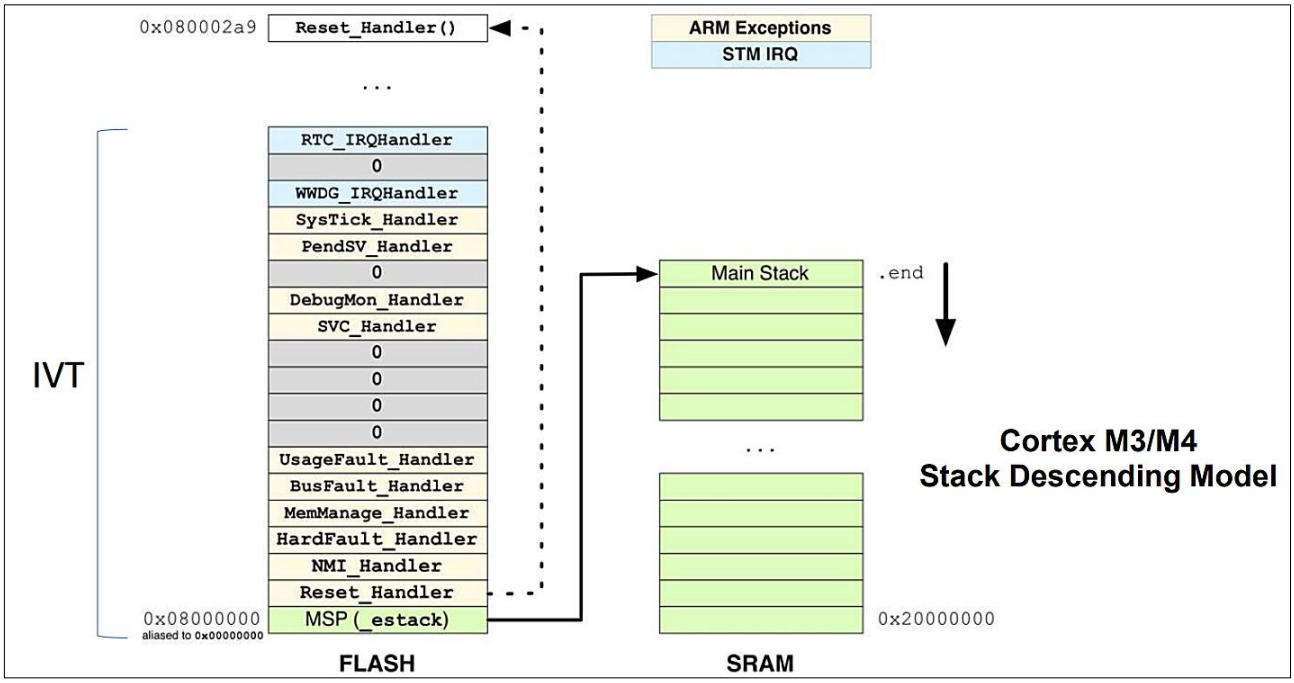
→ It is crucial to set a stack limit to prevent stack overflow.

→ Stack overflow can be detected by comparing stack pointer increments with a predefined limit.

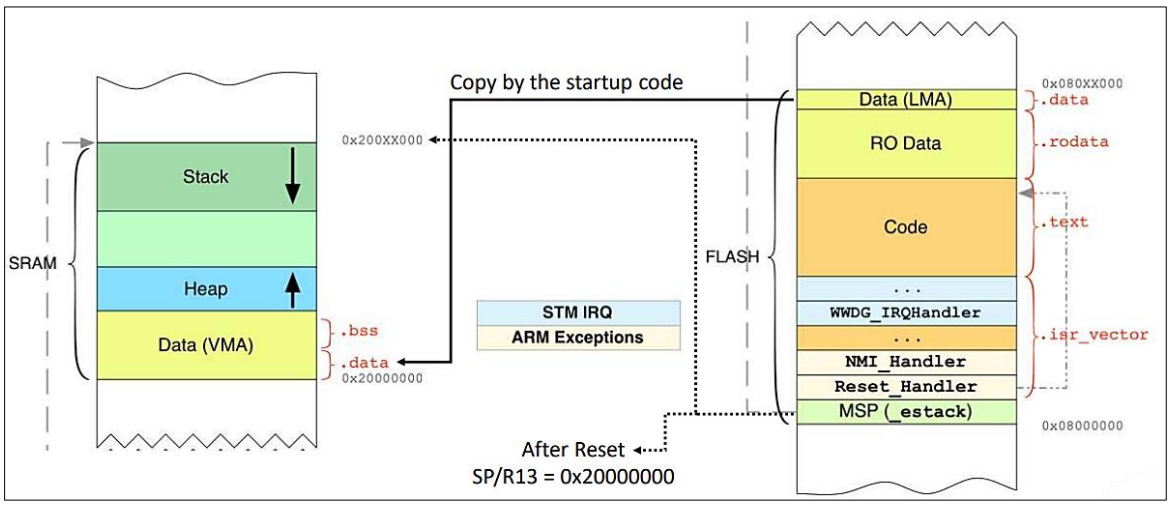
→ The placement of the stack is separate from the heap and data memory segments.

→ The Cortex-M processor uses stack pointers and stack banks for efficient stack management.

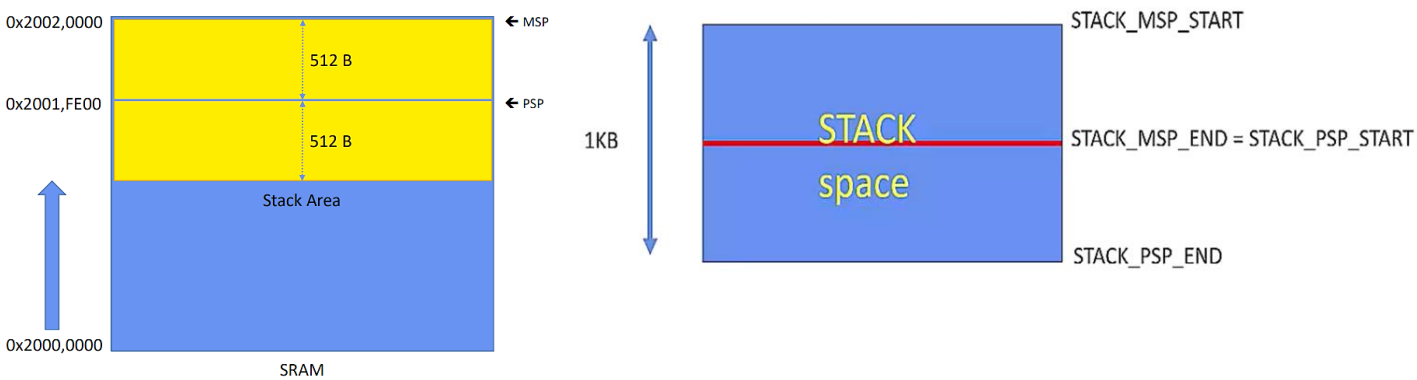
→ **After reset MSP will automatically initialized by the processor by reading the content of 0x0000,0000 from the "Interrupt Vector Table" which is the end of the memory space reserved for stack memory.**



➢ **Full view if typical layout of STM32F4 flash and SRAM memories:**



➢ **Practical example to change the SP(R13) from MSP to PSP and return back to MSP:**



```
__attribute__((naked)) void Change_SP_To_PSP(void){
    /* Crate a symbolic name to the PSP start address */
    __asm volatile (".equ MSP_STACK_START, (0x20000000 + (128 * 1024))");
    __asm volatile (".equ MSP_STACK_END, (MSP_STACK_START - 512)");
    __asm volatile (".equ PSP_STACK_START, MSP_STACK_END");

    /* Set the PSP with the process stack area address */
    __asm volatile ("LDR R0, =PSP_STACK_START");
    __asm volatile ("MSR PSP, R0");

    /* Set the SPSEL in the CONTROL register */
    __asm volatile ("MOV R0, #0x02");
    __asm volatile ("MSR CONTROL, R0");

    /* Return to the caller -> main */
    __asm volatile ("BX LR");
}
```

- **AAPCS (Procedure Call Standard for the Arm Architecture):**

→ It summarizes the uses of the core registers in this standard.

Regis-ter	Syn-onym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8	FP	Frame Pointer or Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable-register 4.
r6	v3		Variable-register 3.
r5	v2		Variable-register 2.
r4	v1		Variable-register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

→ The specification defines how separately compiled and assembled subroutines can work together.

→ The stack & core registers are a vital part of this procedure call standard.

→ The use of stack or core registers for passing values between subroutines is dependent on the fundamental data types.

→ **Routine / Subroutine:** A fragment of program to which control can be transferred.

→ **Procedure**: A routine that returns no result value.

→ **Function**: A routine that returns a result value.

→ **Variadic Routine**: A routine is variadic if the number of arguments it takes, and their type, is determined by the caller instead of the callee.

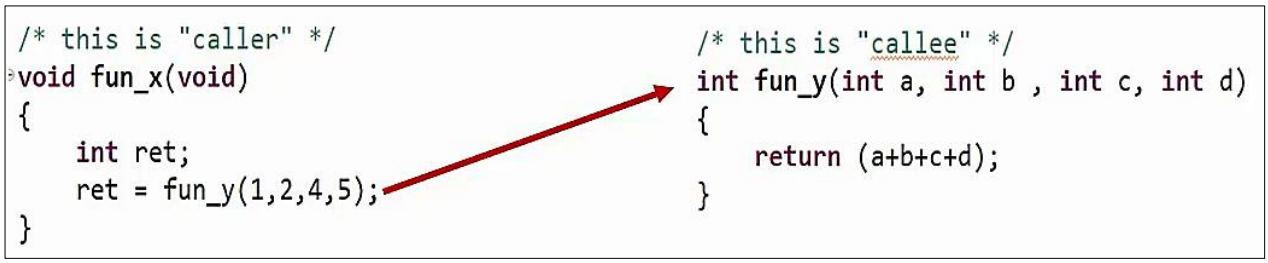
→ **Activation Stack / Call-frame Stack**: The stack of routine activation records (call frames).

→ **Activation Record / Call Frame**: The memory used by a routine for saving registers and holding local variables (usually allocated on a stack, once per activation of the routine).

→ **Argument / Parameter**: They may denote a formal parameter of a routine given the value of the actual parameter when the routine is called, or an actual parameter, according to context.

→ **Scratch Register / Temporary Register**: A register used to hold an intermediate value during a calculation (usually, such values are not named in the program source and have a limited lifetime).

→ **Variable Register / V-Register**: A register used to hold the value of a variable, usually one local to a routine, and often named in the source code.



- The AAPCS defines how subroutines can be separately written, separately compiled, and separately assembled to work together.
- It describes a contract between a calling routine and a called routine that defines:
    1. Obligations on the caller to create a program state in which the called routine may start to execute
    2. Obligations on the called routine to preserve the program state of the caller across the call
    3. The rights of the called routine to alert the program state of its caller
- When the "C Compiler" compiles your c program code for ARM architecture based processor, it should follow the AAPCS standard to generate the code.
- According to the AAPCS standard the "C Function" can modify these registers: R0, R1, R2, R3, R12, R14 (LR), and PSR.
- It is not the responsibility of the function to save these registers before any modification.
- It is the responsivity of the caller function to save these registers "**Caller Saved Registers (R0, R1, R2, R3, R12, R14)"** to the stack before calling the callee function if those values will be needed after the function call. Then it can modify them, lastly retrieved their values back before exiting the callee function.
- According to AAPCS standard, the first four registers R0-R3 are used to pass argument values to the callee function.
- If a function wants to make use of R4 to R11 registers, then it's the responsibility of the function to save its previous contents before modifying those registers and retrieve it back before exiting the function.
- R4 to R11 are called "callee saves registers."
    - The function or subroutine being called needs to make sure that,
        → contents of these registers will be unaltered before exiting the function
- The callee function uses registers R0 (32-Bit Value) or R0+R1 (64-Bit Value) to send/return its result value to the caller function.
- It is the responsibility of the caller function to update the R0 ~ R3 with the callee function parameters.
    - After updating the parameter it calls the callee function.

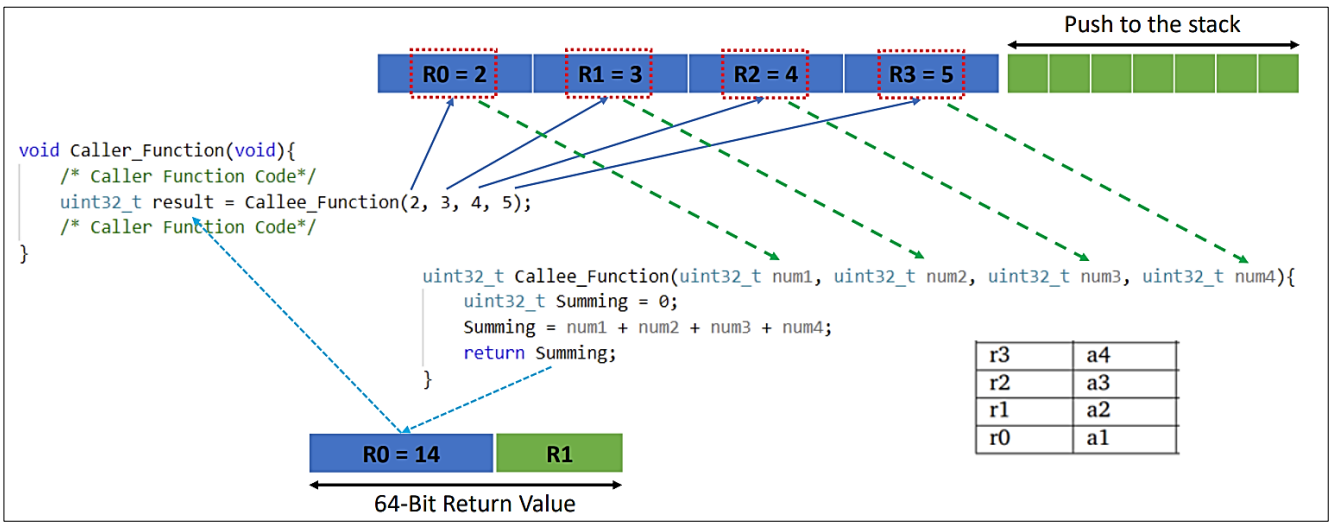
Main Program

```
    ...
; R4 = X, R5 = Y, R6 = Z
BL    function1
```

Subroutine

```
function1
    PUSH    {R4} ; store R4 to stack & adjust SP
    PUSH    {R5} ; store R5 to stack & adjust SP
    PUSH    {R6} ; store R6 to stack & adjust SP
    ... ; Executing task (R4, R5 and R6
      ; could be changed)
    POP     {R4} ; restore R4 and SP re-adjusted
    POP     {R5} ; restore R5 and SP re-adjusted
    POP     {R6} ; restore R6 and SP re-adjusted
    BX      LR   ; Return
; Back to main program
; R4 = X, R5 = Y, R6 = Z
... ; next instructions
```

Register	Synonym
r15	
r14	
r13	
r12	
r11	v8
r10	v7
r9	
r8	v5
r7	v4
r6	v3
r5	v2
r4	v1
r3	a4
r2	a3
r1	a2
r0	a1



THE END