Features of the Cortex M0/M3/M4 processor

* Operational mode of the processor
* the difference access levels of the processor
* the register set of the processor
* the banked stack design of the processor
* exceptions and exception handling
* interrupt handling
* bus interface and bus matrix
* memory architecture of the processor, bit banding, memory
* endianness
* aligned and unaligned data transfer
* bootloader and IAP

Operational mode of the processor

* Processor gives 2 operational modes
  + Thread mode
  + Handler mode
* All your application code will execute under "Thread mode" of the processor. This is

also called as "User mode"

* All the exception handlers or interrupt handlers will run under the "handler mode" of the

processor

* Processor always starts with "Thread mode"
* Whenever the core meets the system exception or any external interrupts then the core will

change its mode to handler mode to service the ISR associated with that system exception or the interrupt

Access level of the processor

* Processor offers 2 access levels
  + Privileged Access Level (PAL)
  + Non-privileged Access Level (NPAL)
* If your code is running with PAL, then your code has full access to all the processor specific resources and restricted registers
* If your code is running with NPAL, then your code may not have access to some of the restricted registers of the processor
* By default, your code will run in PAL
* When the processor is in "Thread mode", it’s possible to move processor into NPAL. Once you move out of the PAL to NPAL being in thread mode, then it’s not possible to come back to PAL unless you change the processor operational mode to "Handler Mode"
* "Handler Mode" code execution is always with PAL
* Use the CONTROL register of the processor if you want to switch between the access level

Cortex M Processor Core Registers

* R0 - R12 (total 13) registers are for general purpose
* All the core registers are 32 bits wide
* The register R13 is called as SP
* The register R14 is called LR and used to hold the return information during function call and exception handling
* The register R15 is called PC which holds the address of the next instruction to be executed
* Note:
  + bx: branch indirect
  + bl: branch with link

Memory mapped and non-memory mapped register of the MCU:

* Non memory mapped register:
  + The register do not have unique address to access them. Hence they are not part of the processor memory map
  + You cannot access these registers in a ‘C’ program using address dereferencing
  + To access these registers, you have to use assembly instructions
  + For instance, processor core registers
* Memory mapped register:
  + Register of the processor specific peripherals (NVICM, MPU, SCB, DEBUG, etc)
  + Registers of the microcontroller specific peripherals (RTC, I2C, TIMER, CAN, USB, etc)
  + Every register has its address in the processor memory map
  + You can access these registers in a ‘C’ program using address dereferencing

ARM GCC inline assembly code usage

* Inline assembly code is used to write pure assembly code inside a ‘C’ program
* GCC inline assembly code syntax as shown below
  + \_\_asm volatile(“MOV RO, R1”);
* \_\_asm volatile(code:output operand list:input operand list: clobber list);
  + Volatile attribute to the asm statement to instruct the compiler not to optimize the assembler code
  + A list of output/input operands separated by commas
  + Clobber list is mainly used to tell the compiler about modifications done by the assembler code

Input/output operands and Constraint string

* Each input and output operand is described by a constraint string followed by a c expression in parentheses
  + Format: “<Constraint string>” (<’C’ expression>)
  + Constraint string = constraint modifier + constraint character

Reset sequence of the Cortex M Processor

* When you reset the processor, the PC is loaded with the value 0x0000\_0000
* The processor reads the value at memory location 0x0000\_0000 into MSP
  + MSP = value at 0x0000\_0000
  + MSP is a Main Stack Pointer register, that means, processor first initialize the Stack Pointer
* After that processor reads the value at memory location 0x0000\_0004 into PC
  + That value is the address of the reset handler
* PC jumps to the reset handler
* A reset handler is just a C or assembly function written by you to carry out any initializations required
* From reset handler you call your main() function of the application

Discussion about T bit of the EPSR

* Various ARM processors support ARM-Thumb interworking, that means the ability to switch between ARM and Thumb state
* The processor must be in ARM state to execute instructions which are from ARM ISA (instruction set architecture) and the processor must be in Thumb state to execute instructions of Thumb ISA.
* If ‘T’ bit of the EPSR is set (1), processor thinks that the next instruction which it is about to execute is from Thumb ISA
* If ‘T’ bit of the EPSR is set (0), processor thinks that the next instruction which it is about to execute is from ARM ISA
* The cortex Mx processor does not support the “ARM” state. Hence, the value of ‘T’ bit must always be 1. Failing to maintain this is illegal and this will result in the “Usage fault” exception
* The lsb (bit 0) of the program counter (PC) is linked to this ‘T’ bit. When you load a value or an address into PC the bit [0] of the value is loaded into the T-bit
* Hence, any address you place in the PC must have its 0th bit as 1. This is usually taken care by the compiler and programmers need not to worry most of the time.
* This is the reason why you see all vector addresses are incremented by 1 in the vector table

Stack memory

* Stack memory is part of the main memory (internal RAM or external RAM) reserved for the temporary storage of data (transient data)
* Mainly used during function, interrupt/exception handling
* Stack memory is used in last in first out fashion (LIFO)
* The stack can be accessed using PUSH and POP instructions or using any memory manipulation instructions (LD, STR)
* The stack is traced using a stack pointer register (SP), PUSH and POP (increment or decrement) instructions affect stack pointer register (SP, R13)

Stack memory uses

* The temporary storage of processor register values
* The temporary storage of local variables of the function
* During system exception or interrupt, stack memory will be used to save the context (some general-purpose register, processor status register, return address) of the currently executing code

Stack operation model

* In ARM Cortex Mx processor stack consumption model is Full Descending (FD)

Different stack operation model

* Full Ascending Stack (FA)
* Full Descending Stack (FD)
* Empty Ascending Stack (EA)
* Empty Descending Stack (ED)

Banked stack pointer

* Cortex Mx Processor physically has 3 stack pointers, SP(R13), MSP and PSP. MSP: main stack pointer, PSP: process stack pointer. SP is called as current stack pointer
* After processor reset, by default, MSP will be selected as current stack pointer. That means, SP copies the value of MSP
* Thread mode can change the current stack pointer to PSP by configuring the CONTROL register’s SPSEL bit
* Handler mode execution will always use MSP as the current stack pointer. That also means that, changing the value of SPSEL bit being in the handler mode does not make sense. The write will be ignored
* MSP will be initialized automatically by the processor after reset by reading the content of the address 0x00000000
* If you want to use the PSP then make sure that you initialize the PSP to valid stack address in your code