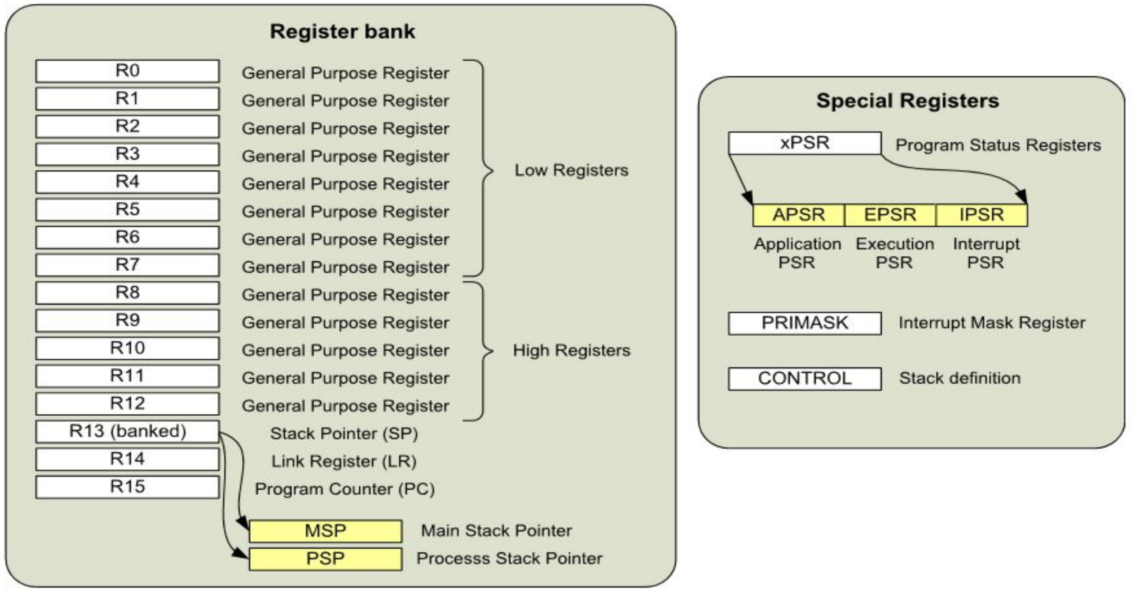
CORTEX M0
Registers and Special Registers

Register Banks and Special Registers



R14-Link Register

- R14, Link Register (LR) R14 is the Link Register.
- The Link Register is used for storing the return address of a subroutine or function call.
- At the end of the subroutine or function, the return address stored in LR is loaded into the program counter so that the execution of the calling program can be resumed.
- In the case where an exception occurs, the LR also provides a special code value, which is used by the exception return mechanism.
- When using ARM development tools, you can access to the Link Register using either “R14” or “LR.”

R14-Link Register

- Both upper and lowercase (e.g., “r14” or “lr”) can be used.
- Although the return address in the Cortex-M0 processor is always an even address (bit[0] is zero because the smallest instructions are 16-bit and must be half-word aligned), bit zero of LR is readable and writeable.
- In the ARMv6-M architecture, some instructions require bit zero of a function address set to 1 to indicate Thumb state.

Stack Pointer

```
FUNCTION
      SUB SP, SP, #0x8 ; Reserve 2 words of stack;(8 bytes) for local
variables
;Data processing in function
MOVS r0, #0x12 ; set a dummy value
STR r0, [sp, #0] ; Store 0x12 in 1st local variable
               STR r0, [sp, #4] ; Store 0x12 in 2nd local variable
               LDR r1, [sp, #0] ; Read from 1st local variable
               LDR r2, [sp, #4] ; Read from 2nd local variable
               ADD SP, SP, #0x8; Restore SP to original position
               BX LR;(Branch to address stored in the Link;Register. This
instruction is often used for;function return.)
         __main
        BL FUNCTION;branch and link to the address of FUNCTION
      ADDS   R2, R2, #10; Add R0 a


        END
```

R15-Program Counter

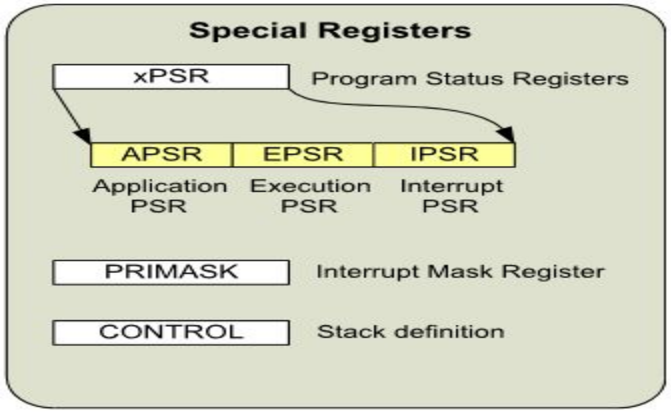
- R15, Program Counter (PC) R15 is the Program Counter. It is readable and writeable. A read returns the current instruction address plus four (this is caused by the pipeline nature of the design).
- Writing to R15 will cause a branch to take place (but unlike a function call, the Link Register does not get updated).

R15-Program Counter

- In the ARM assembler, you can access the Program Counter, using either “R15” or “PC,” in either upper or lower case (e.g., “r15” or “pc”). Instruction addresses in theCortex-M0 processor must be aligned to half-word address,which means the actual bit zero of the PC should be zero all the time.
- However, when attempting to carry out a branch using the branch instructions (BX or BLX), the LSB of the PC should be set to 1.
- This is to indicate that the branch target is a Thumb program region. Otherwise, it can imply trying to switch the processor to ARM state (depending on the instruction used), which is not supported and will cause a fault exception.

xPSR

- xPSR, combined Program Status Register The combined Program Status Register provides information about program execution and the ALU flags.
- It is consists of the following three Program Status Registers (PSRs)
  - Application PSR (APSR)
  - Interrupt PSR (IPSR)
  - Execution PSR (EPSR)



APSR,IPSR and EPSR

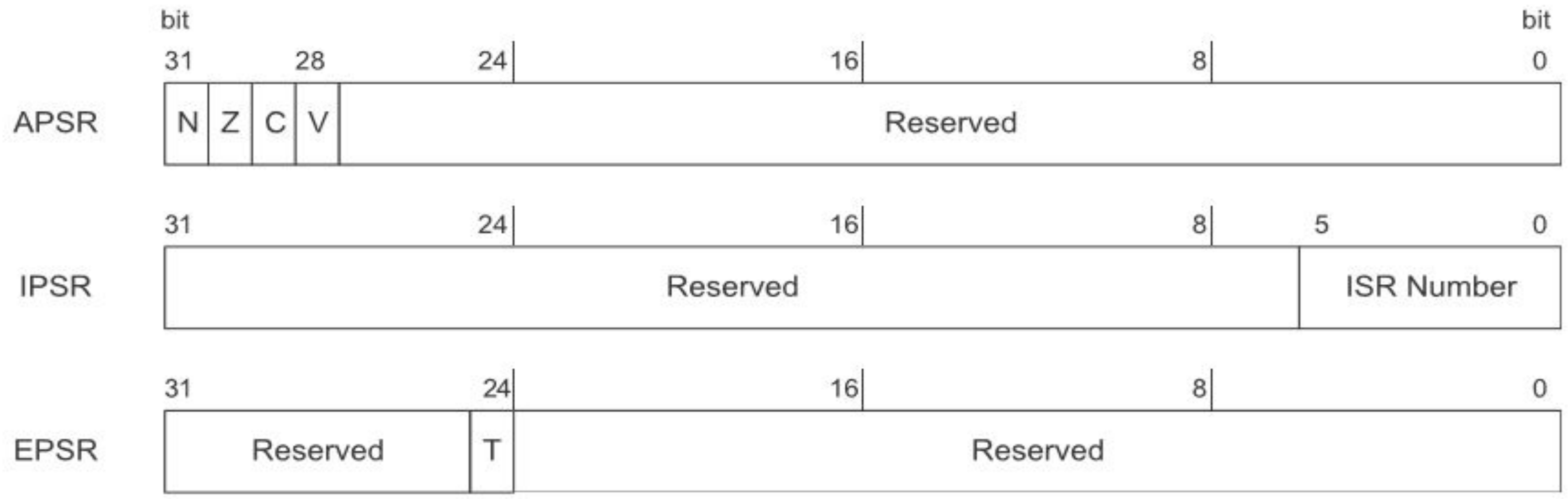


Figure 3.3:
APSR, IPSR, and EPSR.

APSR,IPSR and EPSR

- 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 interrupt service routine (ISR) number. Each exception on the Cortex-M0 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 what exception it is serving.
- The EPSR on the Cortex-M0 processor contains the T-bit, which indicates that the processor is in the Thumb state.
- On the Cortex-M0 processor, this bit is normally set to 1 because the Cortex-M0 only supports the Thumb state.

APSR,IPSR and EPSR

- If this bit is cleared, a hard fault exception will be generated in the next instruction execution. These three registers can be accessed as one register called xPSR (Figure 3.4).
- For example, when an interrupt takes place, the xPSR is one of the registers that is stored onto the stack memory automatically and is restored automatically after returning from an exception.
- During the stack store and restore, the xPSR is treated as one register.

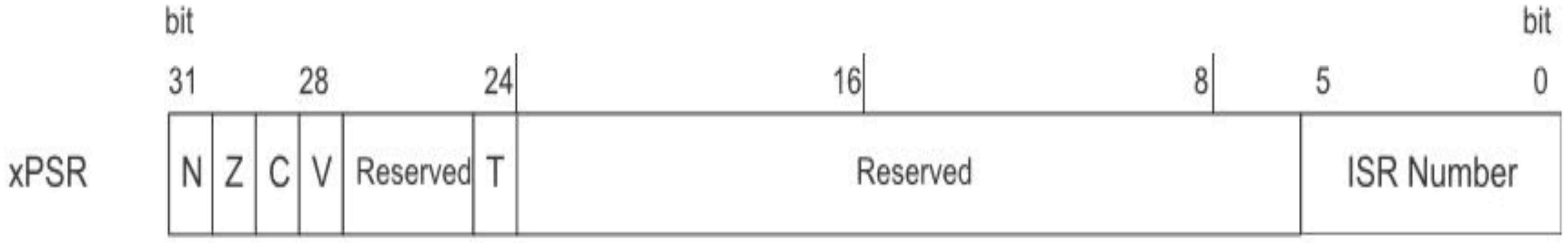
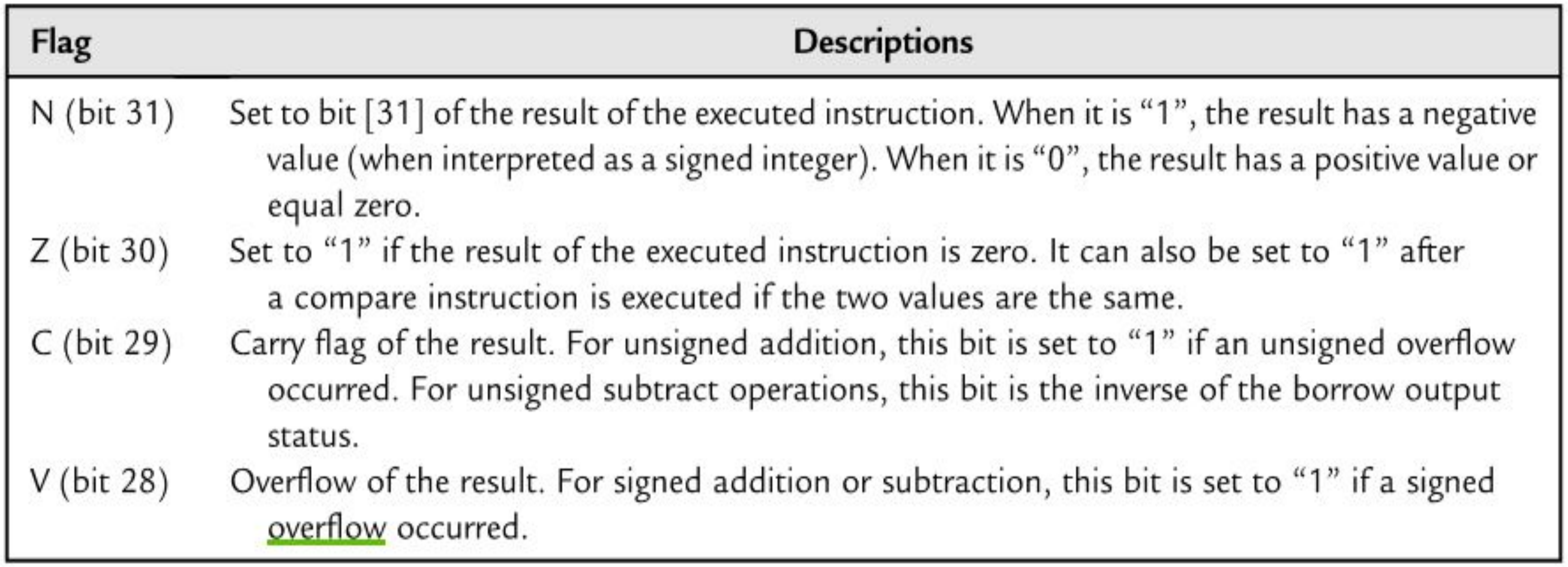


Figure 3.4:
xPSR.

Behaviors of the Application Program Status Register (APSR)

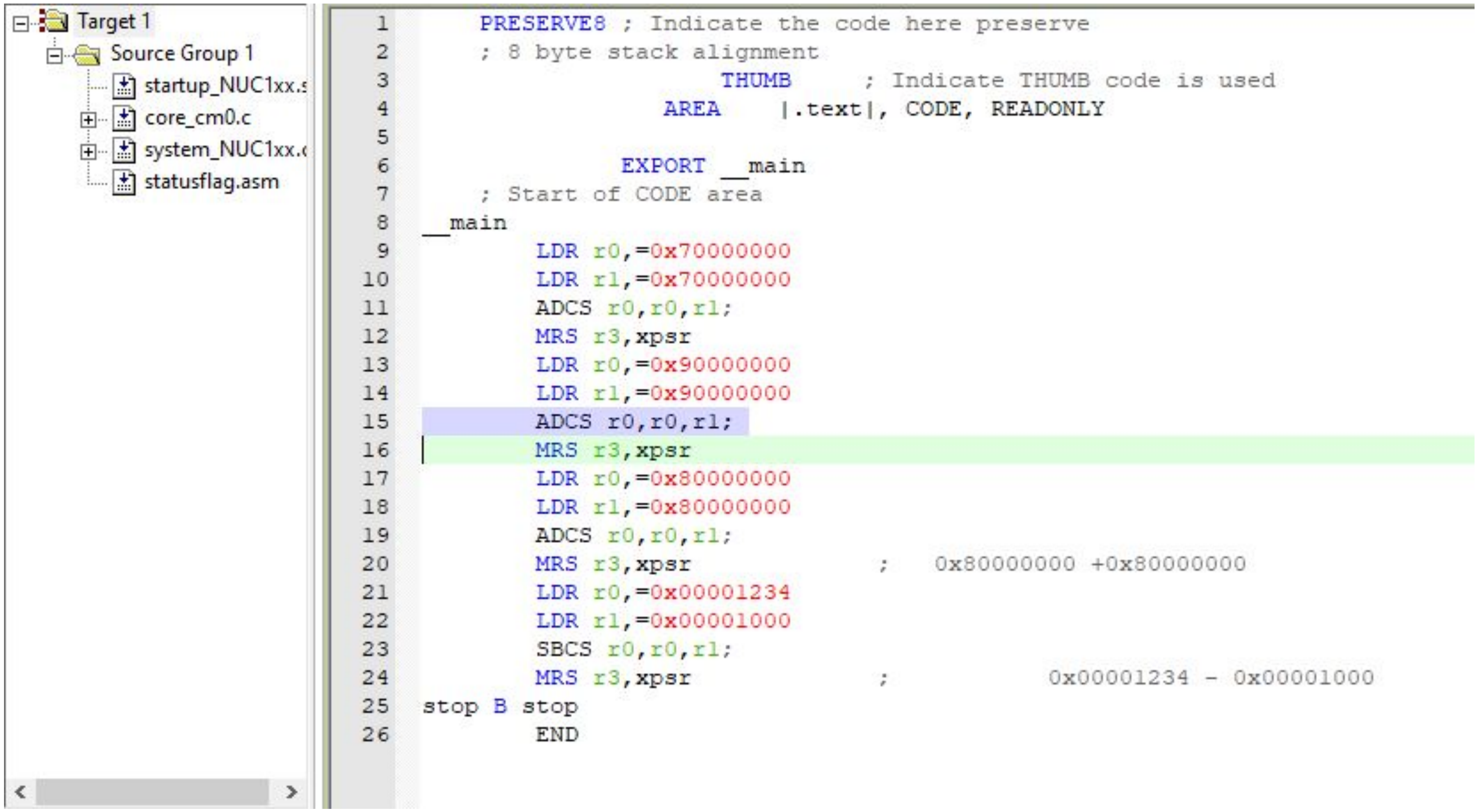
Table 3.1: ALU Flags on the Cortex-M0 Processor



Flag	Descriptions
N (bit 31)	Set to bit [31] of the result of the executed instruction. When it is "1", the result has a negative value (when interpreted as a signed integer). When it is "0", the result has a positive value or equal zero.
Z (bit 30)	Set to "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, this bit is set to "1" if an unsigned overflow occurred. For unsigned subtract operations, this bit is the inverse of the borrow output status.
V (bit 28)	Overflow of the result. For signed addition or subtraction, this bit is set to "1" if a signed overflow occurred.

xPSR

The overflow flag `OF` gets set when the addition step changes the sign of the value in a way inconsistent with the operands, such as adding two positive values giving a negative result, or adding two negative values and getting a positive result.

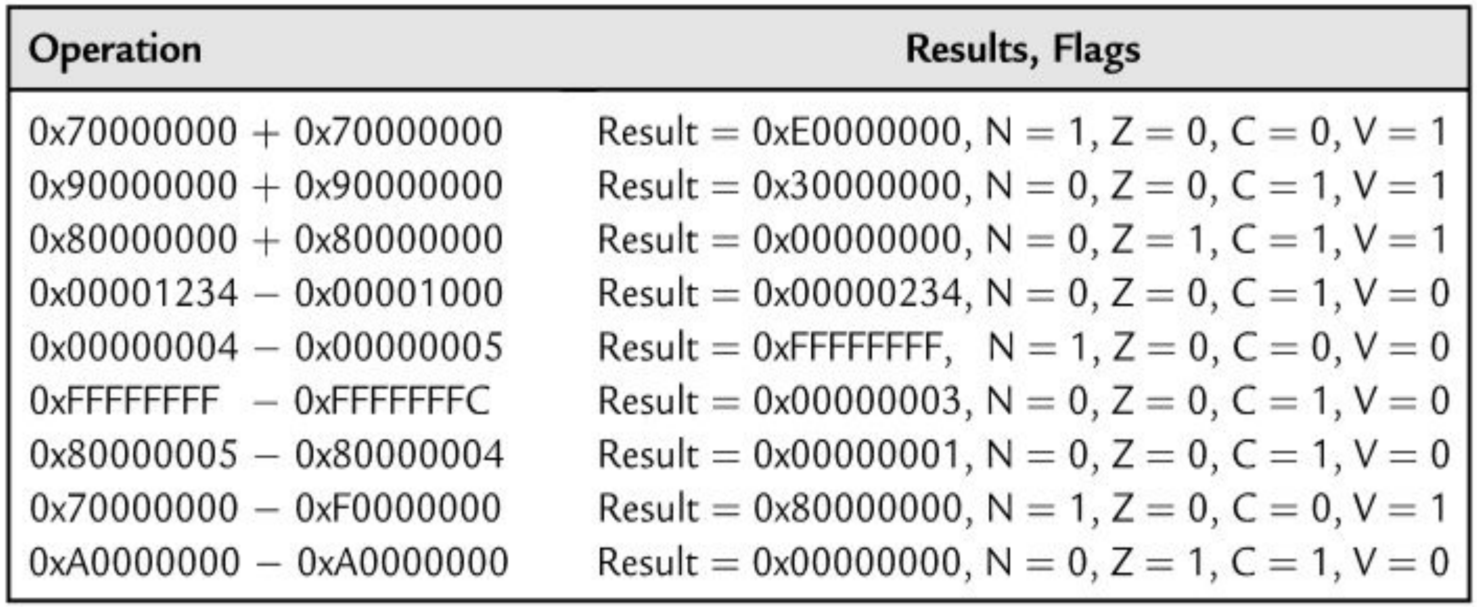


```
    PRESERVE8 ; Indicate the code here preserve
    ; 8 byte stack alignment
                    THUMB     ; Indicate THUMB code is used
                AREA    |.text|, CODE, READONLY

             EXPORT __main
    ; Start of CODE area
__main
        LDR r0,=0x70000000
        LDR r1,=0x70000000
        ADCS r0,r0,r1;
        MRS r3,xpsr
        LDR r0,=0x90000000
        LDR r1,=0x90000000
        ADCS r0,r0,r1;
        MRS r3,xpsr
        LDR r0,=0x80000000
        LDR r1,=0x80000000
        ADCS r0,r0,r1;
        MRS r3,xpsr             ;   0x80000000 +0x80000000
        LDR r0,=0x00001234
        LDR r1,=0x00001000
        SBCS r0,r0,r1;
        MRS r3,xpsr             ;           0x00001234 - 0x00001000
stop B stop
        END
```

Example

Table 3.2: 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

The other common usage of APSR flag is to control branching.

Example

0x70000000 +0x70000000

Result = 0xE0000000, N = 1, Z =0, C =0, V =1

1st Data:0x0111|0000|0000|0000|0000|0000|0000|0000

2ndData:0x0111|0000|0000|0000|0000|0000|0000|0000|

Result:01110|0000|0000|0000|0000|0000|0000|0000

0xE0000000

Check the MSB if 1 N=1

if Result =0 then Z=1 ,in this case Z=0,

if MSB results in a carry ,then C=1, in this case C=0,

if MSBs of the 2 data are either 0/1 and the MSB of the result is 1/0 Then V=1 ,in this case V=1

Example

0x90000000 + 0x90000000 ;check there is an error in the text book

Result =0x20000000, N = 0, Z =0, C = 1, V =1

```
 1001|0000|0000|0000|0000|0000|0000|0000
 1001|0000|0000|0000|0000|0000|0000|0000
10010|0000|0000|0000|0000|0000|0000|0000
```

0x80000000 +0x80000000 Result = 0x00000000, N =0, Z =1, C= 1, V =1

0x00001234 - 0x00001000 Result = 0x00000234, N =0, Z = 0, C = 1, V = 0

Example

0x80000000 +0x80000000 Result = 0x00000000,

1000|0000|0000|0000|0000|0000|0000|0000

1000|0000|0000|0000|0000|0000|0000|0000

10000|0000|0000|0000|0000|0000|0000|0000

N =0, Z =1, C= 1, V =1

A+`B+1

0x00001234 - 0x00001000 Result = 0x00000234,

00000|0000|0000|0000|0001|0000|0000|0000|2nd No

0000|0000|0000|0000|0001|0010|0011|0100|1st no

1111|1111|1111|1111|1110|1111|1111|1111|2s complement of 2nd no

0000|0000|0000|0000|0000|0000|0000|0001|

10000|0000|0000|0000|0010|0011|0100

N =0, Z = 0, C = 1, V = 0

PRIMASK

- PRIMASK: Interrupt Mask Special Register The PRIMASK register is a 1-bit-wide interrupt mask register (Figure 3.5).
- When set, it blocks all interrupts apart from the nonmaskable interrupt (NMI) and the hard fault exception.
- Effectively it raises the current interrupt priority level to 0, which is the highest value for a programmable exception.
- ThePRIMASK register can be accessed using special register access instructions(MSR,MRS) as well as using an instruction called the Change Processor State (CPS).
- This is commonly used for handling time-critical routines.

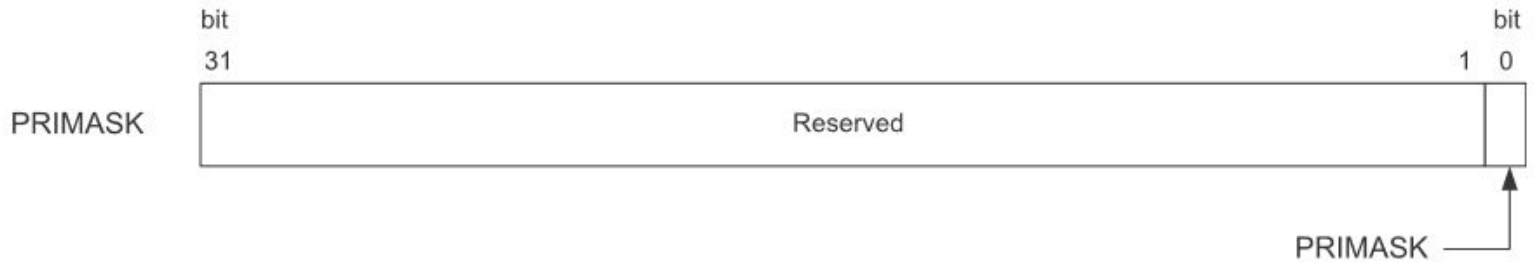


Figure 3.5:
PRIMASK.

Control Register

- CONTROL: Special Register As mentioned earlier, there are two stack pointers in the Cortex-M0 processor.
- The stack pointer selection is determined by the processor mode as well as the configuration of the CONTROL register (Figure 3.6).
- After reset, the main stack pointer (MSP) is used, but can be switched to the process stack pointer (PSP) in Thread mode (when not running an exception handler) by setting bit [1] in the CONTROL register (Figure 3.7).

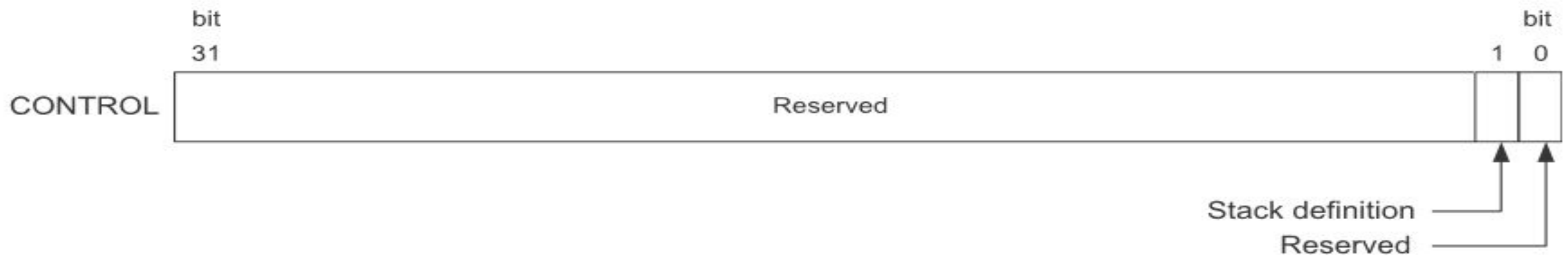


Figure 3.6:
CONTROL

Setting of CONTROL

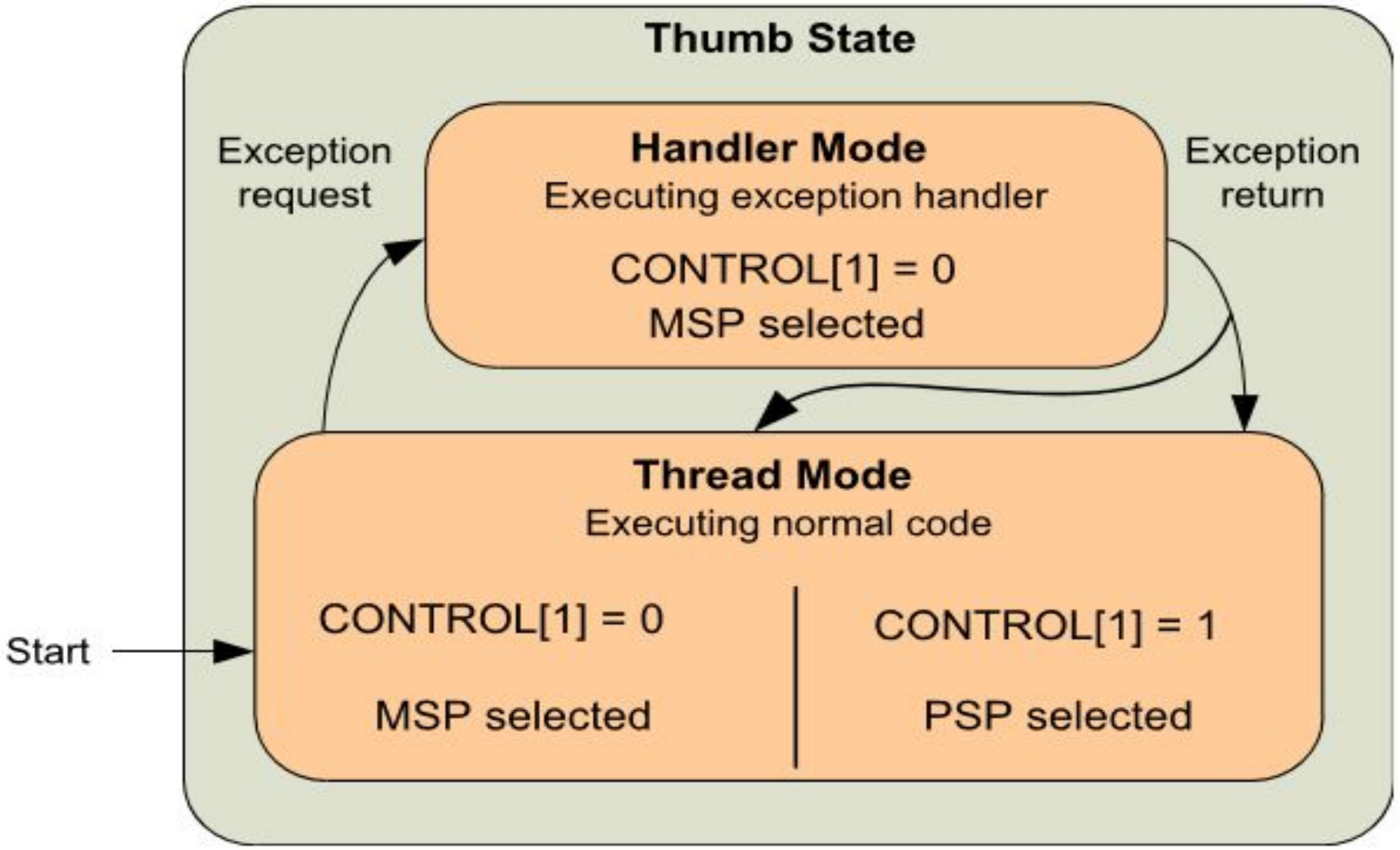


Figure 3.7:
Stack pointer selection.

CONTROL REGISTERS

- During running of an exception handler (when the processor is in Handler mode), only the MSP is used, and the CONTROL register reads as zero.
- The CONTROL register can only be changed in Thread mode or via the exception entrance and return mechanism. Bit 0 of the CONTROL register is reserved to maintain compatibility with the Cortex-M3 processor.
- In the Cortex-M3 processor, bit 0 can be used to switch the processor to User mode (non-privileged mode). This feature is not available in the Cortex-M0 processor.

Memory System Overview

- The Cortex-M0 processor has 4 GB of memory address space (Figure 3.8).
- The memory space is architecturally defined as a number of regions, with each region having a recommended usage to help software porting between different devices.
- The Cortex-M0 processor contains a number of built-in components like the NVIC and a number of debug components.
- These are in fixed memory locations within the system region of the memory map. As a result, all the devices based on the Cortex-M0 have the same programming model for interrupt control and debug.
- This makes it convenient for software porting and helps debug tool vendors to develop debug solutions for the Cortex-M0 based microcontroller or system-on-chip (SoC) products.

Memory System Overview

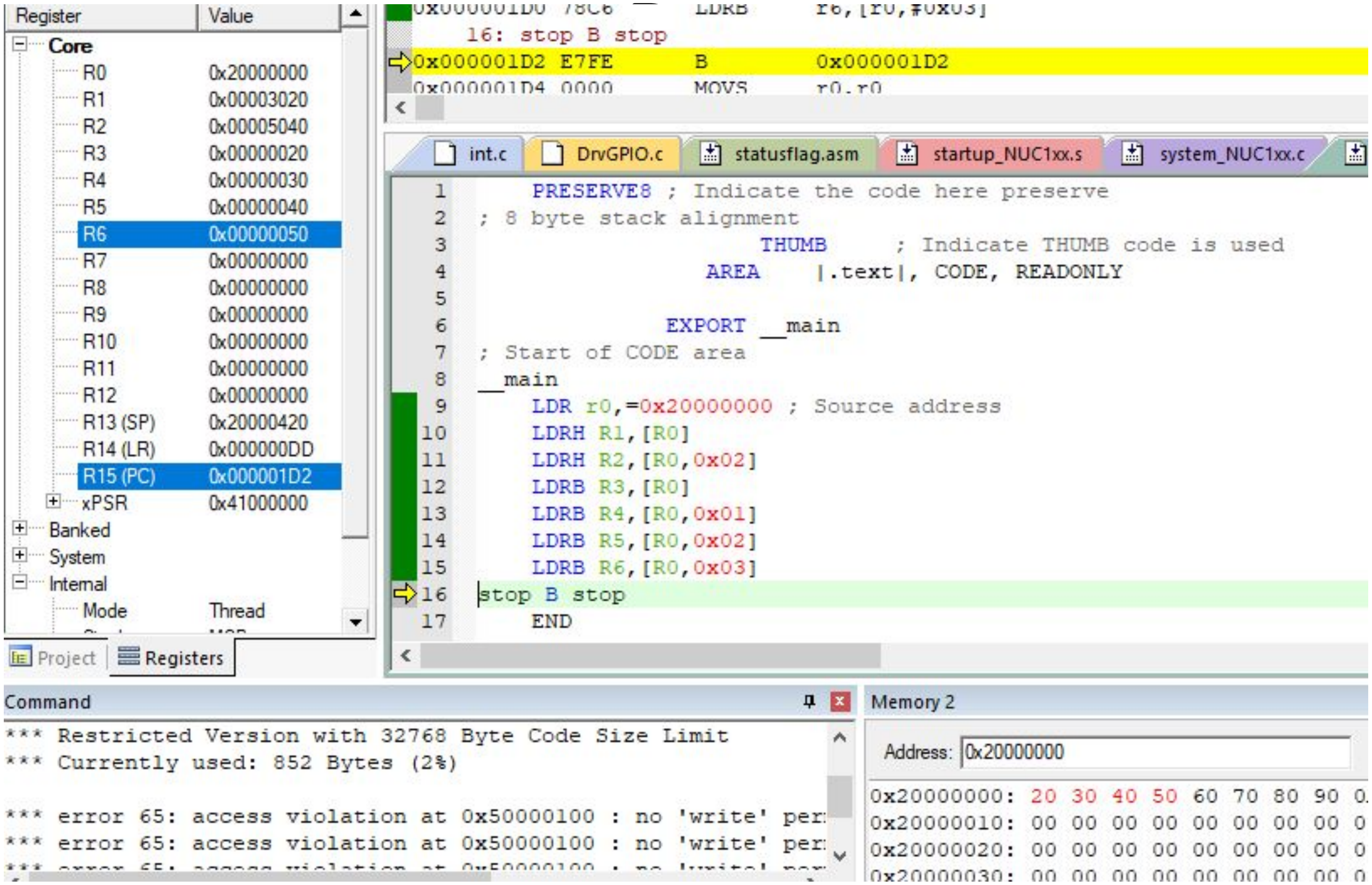
- In most cases, the memories connected to the Cortex-M0 are 32-bits, but it is also possible to connect memory of different data widths to the Cortex-M0 processor with suitable memory interface hardware.
- The Cortex-M0 memory system supports memory transfers of different sizes such as byte (8-bit), half word (16-bit), and word (32-bit). The Cortex-M0

Use of Loading Half Word and Byte

```
PRESERVE8 ; Indicate the code here preserve
; 8 byte stack alignment
                THUMB     ; Indicate THUMB code is used
             AREA    |.text|, CODE, READONLY

           EXPORT __main
; Start of CODE area
__main
       LDR r0,=0x20000000 ; Source address
       LDRH R1,[R0]
       LDRH R2,[R0,0x02]
       LDRB R3,[R0]
       LDRB R4,[R0,0x01]
       LDRB R5,[R0,0x02]
       LDRB R6,[R0,0x03]
stop B stop
       END
```

Use of Loading Half Word and



Register	Value
Core	
R0	0x20000000
R1	0x00003020
R2	0x00005040
R3	0x00000020
R4	0x00000030
R5	0x00000040
R6	0x00000050
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x20000420
R14 (LR)	0x000000DD
R15 (PC)	0x000001D2
xPSR	0x41000000
Banked	
System	
Internal	
Mode	Thread

```
0x000001D0 78C6      LDRB     r6,[r0,#0x03]
    16: stop B stop
0x000001D2 E7FE      B        0x000001D2
0x000001D4 0000      MOVS     r0,r0
```

int.c | DrvGPIO.c | statusflag.asm | startup_NUC1xx.s | system_NUC1xx.c

```
    PRESERVE8 ; Indicate the code here preserve
; 8 byte stack alignment
                THUMB      ; Indicate THUMB code is used
            AREA     |.text|, CODE, READONLY

         EXPORT __main
; Start of CODE area
__main
    LDR r0,=0x20000000 ; Source address
    LDRH R1,[R0]
    LDRH R2,[R0,0x02]
    LDRB R3,[R0]
    LDRB R4,[R0,0x01]
    LDRB R5,[R0,0x02]
    LDRB R6,[R0,0x03]
stop B stop
    END
```

Command

```
*** Restricted Version with 32768 Byte Code Size Limit
*** Currently used: 852 Bytes (2%)

*** error 65: access violation at 0x50000100 : no 'write' per
*** error 65: access violation at 0x50000100 : no 'write' per
```

Memory 2

Address: 0x20000000

```
0x20000000: 20 30 40 50 60 70 80 90 0
0x20000010: 00 00 00 00 00 00 00 00 0
0x20000020: 00 00 00 00 00 00 00 00 0
0x20000030: 00 00 00 00 00 00 00 00 0
```

Memory Overview

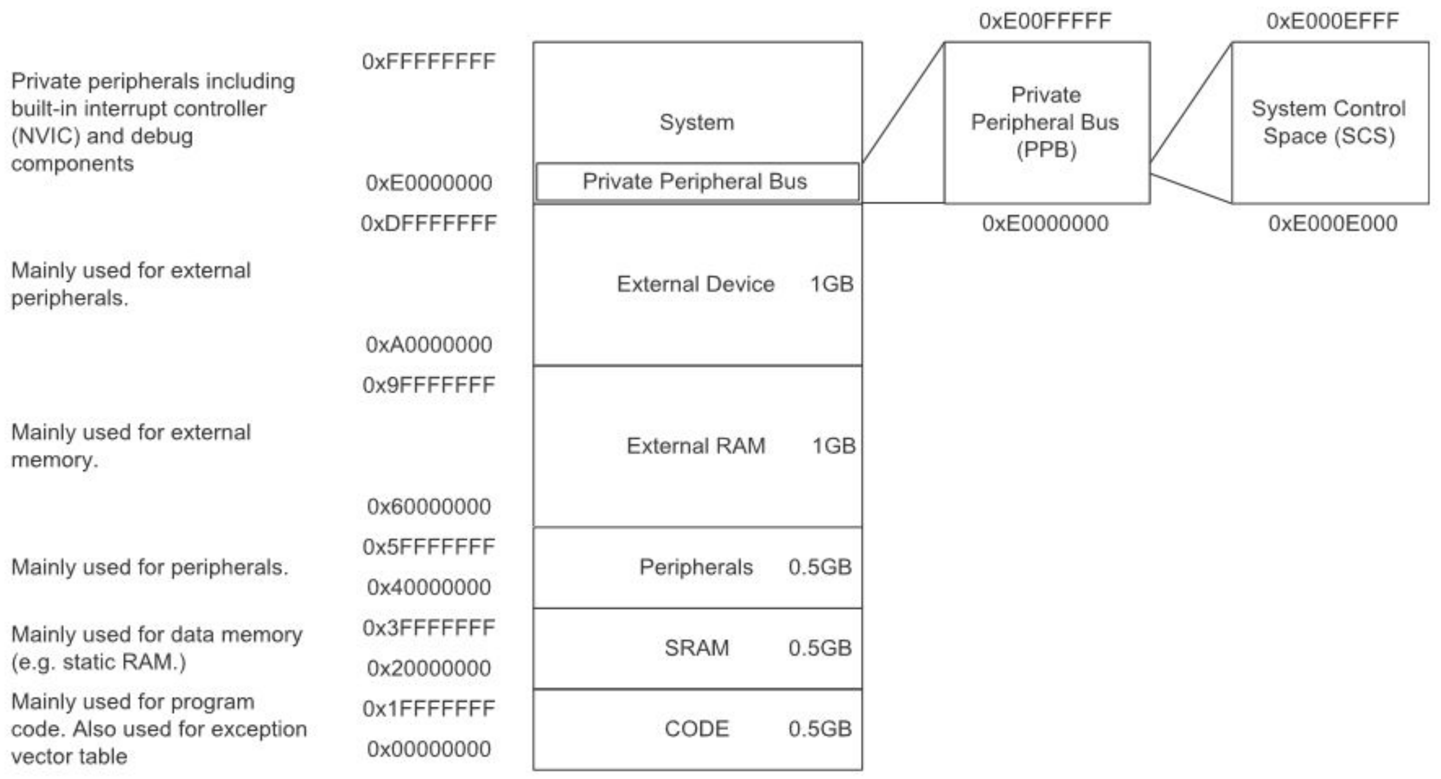

Private peripherals including built-in interrupt controller (NVIC) and debug components
Mainly used for external peripherals.
Mainly used for external memory.
Mainly used for peripherals.
Mainly used for data memory (e.g. static RAM.)
Mainly used for program code. Also used for exception vector table
0xFFFFFFFF
0xE0000000
0xDFFFFFFF
0xA0000000
0x9FFFFFFF
0x60000000
0x5FFFFFFF
0x40000000
0x3FFFFFFF
0x20000000
0x1FFFFFFF
0x00000000
System
Private Peripheral Bus
External Device 1GB
External RAM 1GB
Peripherals 0.5GB
SRAM 0.5GB
CODE 0.5GB
0xE00FFFFF
Private Peripheral Bus (PPB)
0xE0000000
0xE000EFFF
System Control Space (SCS)
0xE000E000

Figure 3.8:
Memory map.

Stack Memory Operations

- Stack memory is a memory usage mechanism that allows the system memory to be used as temporary data storage that behaves as a first-in, last-out buffer.
- One of the essential elements of stack memory operation is a register called the stack pointer. The stack pointer is adjusted automatically each time a stack operation is carried out.
- In the Cortex-M0 processor, the stack pointer is register R13 in the register bank. Physically there are two stack pointers in the Cortex-M0 processor, but only one of them is used at one time, depending on the current value of the CONTROL register and the state of the processor (see Figure 3.7).
- In common terms, storing data to the stack is called pushing (using the PUSH instruction) and restoring data from the stack is called popping (using the POP instruction).

Stack Memory Operations

- Depending on processor architecture, some processors perform storing of new data to stack memory using incremental address indexing and some use decrement address indexing.
- In the Cortex-M0 processor, the stack operation is based on a “full-descending” stack model. This means stack pointer always points to the last filled data in the stack memory, and the stack pointer predecrements for each new data store (PUSH)

PUSH POP OPERATION

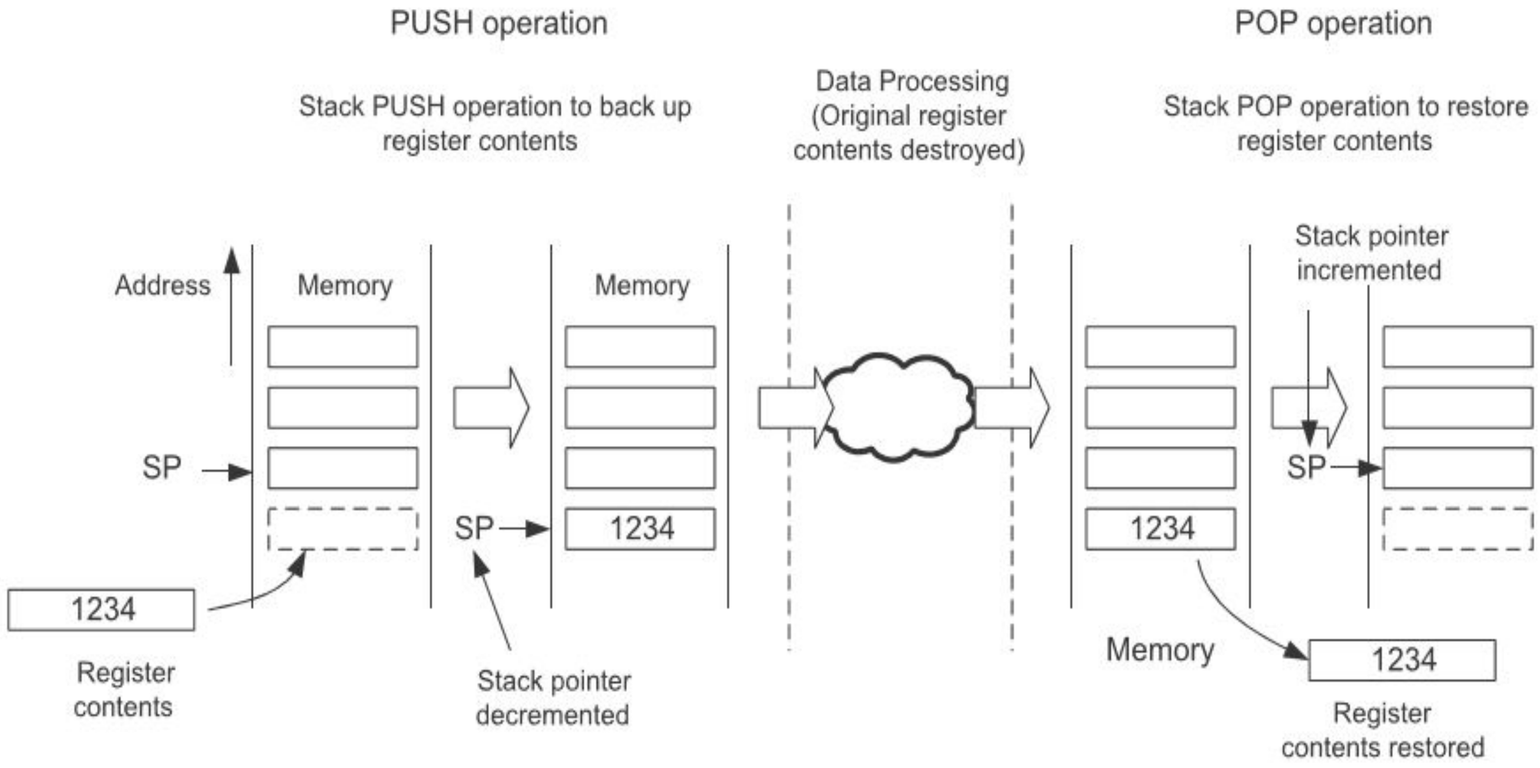

PUSH operation
Stack PUSH operation to back up register contents
Data Processing (Original register contents destroyed)
POP operation
Stack POP operation to restore register contents
Address
Memory
Memory
SP
SP
1234
1234
1234
1234
Register contents
Stack pointer decremented
Stack pointer incremented
SP
Memory
Register contents restored

Figure 3.9:
Stack PUSH and POP in the Cortex-M0 processor.

Stack Operation

- stack pointer always points to the last filled data in the stack memory, and the stack pointer predecrements for each new data store (PUSH) (Figure 3.9).
- PUSH and POP are commonly used at the beginning and end of a function or subroutine. At the beginning of a function, the current contents of the registers used by the calling program are stored onto the stack memory using a PUSH operation, and at the end of the function, the data on the stack memory is restored to the registers using a POP operation.
- Typically, each register PUSH operation should have a corresponding register POP operation, otherwise the stack pointer will not be able to restore registers to their original values.
- This can result in unpredictable behavior, for example, stack overflow. The minimum data size to be transferred for each push and pop operations is one word (32-bit), and multiple registers can be pushed or popped in one instruction.

Continueed..

- The stack memory accesses in the Cortex-M0 processor are designed to be always word aligned (address values must be a multiple of 4, for example, 0x0, 0x4, 0x8, etc.), as this gives the best efficiency for minimum design complexity.
- For this reason, bits [1:0] of both stack pointers in the Cortex-M0 processor are hardwired to zeros and read as zeros. The stack pointer can be accessed as either R13 or SP. Depending on the processor state and the CONTROL register value, the stack pointer accessed can either be the main stack pointer (MSP) or the process stack pointer (PSP).
- In many simple applications, only one stack pointer is needed and by default the main stack pointer (MSP) is used. The process stack pointer (PSP) is usually only required when an operating system (OS) is used in the embedded application (Table 3.3).

SP with PUSH and POPInstructions

- __main
-                LDR r3,=0x20000100
-                 LDR r0,=0x20000050
-                 LDMIA r3!,{r1,r2}
-
-                 mov SP,r0
-                 PUSH {r1,r2}
-                 POP {r4,r5}
- stop   B   stop
-                END   ; End of file

Exceptions and Interrupts

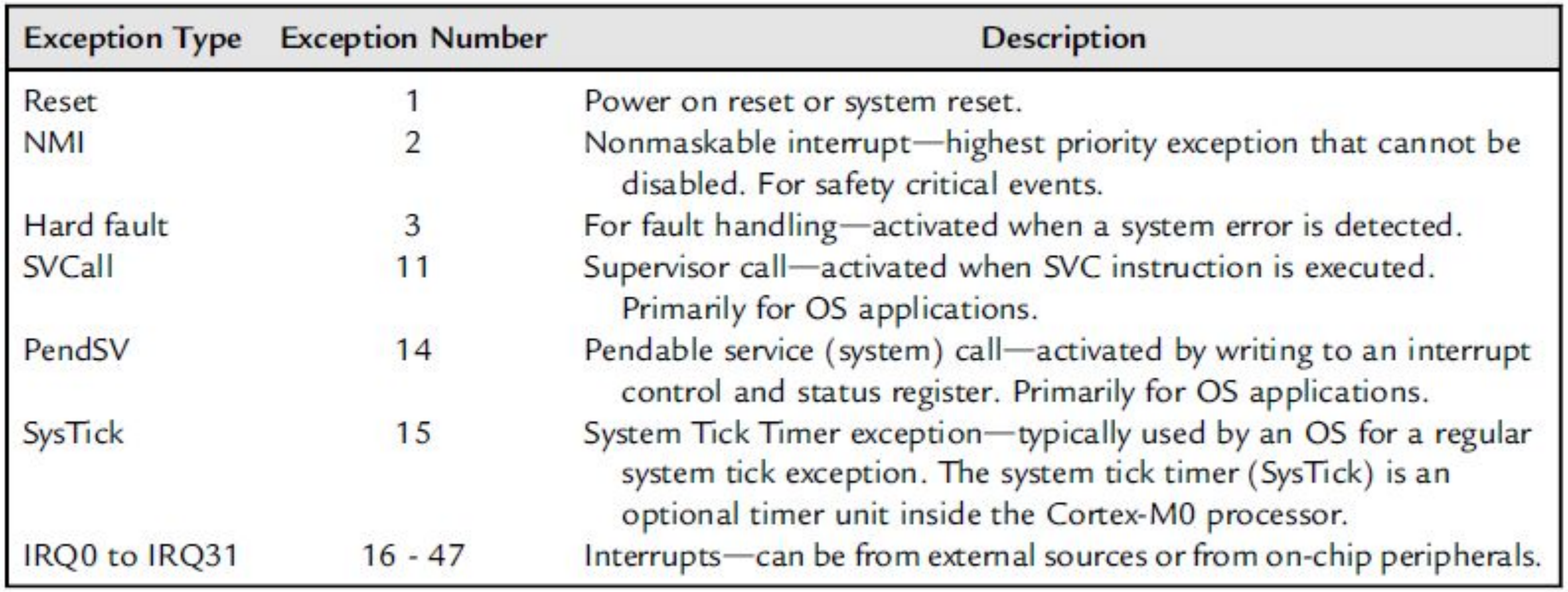
- Exceptions are events that cause change to program control: instead of continuing program execution, the processor suspends the current executing task and executes a part of the programcode called the exception handler.
- After the exception handler is completed, it will then resume the normal program execution.
- There are various types of exceptions, and interrupts are a subset of exceptions.
- The Cortex-M0 processor supports up to 32 external interrupts (commonly referred as IRQs) and an additional special interrupt called the nonmaskable interrupt (NMI).

- The exception handlers for interrupt events are commonly known as interrupt service routines (ISRs).
- Interrupts are usually generated by on-chip peripherals, or by external input through I/O ports.
- The number of available interrupts on the Cortex-M0 processor depends on the microcontroller product you use.
- In systems with more peripherals, it is possible for multiple interrupt sources to share one interrupt connection.

INTERRUPTS AND EXCEPTIONS

- Besides the NMI and IRQ, there are a number of system exceptions in the Cortex-M0 processor, primarily for OS use and fault handling (Table 3.4).

Table 3.4: Exception Types



Exception Type	Exception Number	Description
Reset	1	Power on reset or system reset.
NMI	2	Nonmaskable interrupt—highest priority exception that cannot be disabled. For safety critical events.
Hard fault	3	For fault handling—activated when a system error is detected.
SVCall	11	Supervisor call—activated when SVC instruction is executed. Primarily for OS applications.
PendSV	14	Pendable service (system) call—activated by writing to an interrupt control and status register. Primarily for OS applications.
SysTick	15	System Tick Timer exception—typically used by an OS for a regular system tick exception. The system tick timer (SysTick) is an optional timer unit inside the Cortex-M0 processor.
IRQ0 to IRQ31	16 - 47	Interrupts—can be from external sources or from on-chip peripherals.

INTERRUPTS AND EXCEPTIONS

- Each exception has an exception number. This number is reflected in various registers
- including the IPSR and is used to define the exception vector addresses.
- Note that exception numbers are separated from interrupt numbers used in device driver libraries.

Nested Vectored Interrupt Controller (NVIC)

- To prioritize the interrupt requests and handle other exceptions, the Cortex-M0 processor hasa built-in interrupt controller called the Nested Vectored Interrupt Controller (NVIC).
- The interrupt management function is controlled by a number of programmable registers in the NVIC.

- These registers are memory mapped, with the addresses located within the System Control Space (SCS) as illustrated in Figure 3.8.

The NVIC supports a number of features

- Flexible interrupt management
- Nested interrupt support
- Vectored exception entry
- Interrupt masking

Flexible Interrupt Management

- In the Cortex-M0 processor, each external interrupt can be enabled or disabled and can have its pending status set or clear by software.
- It can also accept exception requests at a signal level (interrupt request from a peripheral remain asserted until the interrupt service routine clears the interrupt request), as well as an exception request pulse (minimum1 clock cycle).

- This allows the interrupt controller to be used with any interrupt source.

- ***Nested Interrupt Support***
- In the Cortex-M0 processor, each exception has a priority level. The priority level can
- be fixed or programmable. When an exception occurs, such as an external interrupt, the

- NVIC will compare the priority of this exception to the current level.
- If the new exception has a higher priority, the current running task will be suspended.
- Some of the registers will be stored on to the stack memory, and the processor will start executing the exception handler of the new exception.

- This process is called “preemption.” When the higher priority exception handler is complete, it is terminated with an exception return operation and the processor automatically restores the registers from the stack and resumes the task that was running previously.
- This mechanism allows nesting of exception services without any software overhead.

Vectored Exception Entry

- When an exception occurs, the processor will need to locate the starting point of thecorresponding exception handler.
- Traditionally, in ARM processors such as the ARM7TDMI, software usually handles this step.
- The Cortex-M0 automatically locates the starting point of the exception handler from a vector table in the memory.

- As a result, the delay from the start of the exception to the execution of the exception handlers is reduced.

Interrupt Masking

- The NVIC in the Cortex-M0 processor provides an interrupt masking feature via the PRIMASK special register.
- This can disable all exceptions except hard fault and NMI.

Program Image and Startup Sequence

- To understand the startup sequence of the Cortex-M0 processor, we need to have a quick overview on the program image first.
- Normally, the program image for the Cortex-M0processor is located from address 0x00000000.

- The beginning of the program image contains the vector table (Figure 3.11).
- It contains the starting addresses (vectors) of exceptions.
- Each vector is located in address of "Exception_Number 4." For example, external IRQ #0 is exception type #16, therefore the address of the vector for IRQ#0 is in 16 * 4 = 040.

- These vectors have LSB set to 1 to indicate that the exceptions handlers are to be executed with Thumb instructions.
- The size of the vector table depends on how many interrupts are implemented.

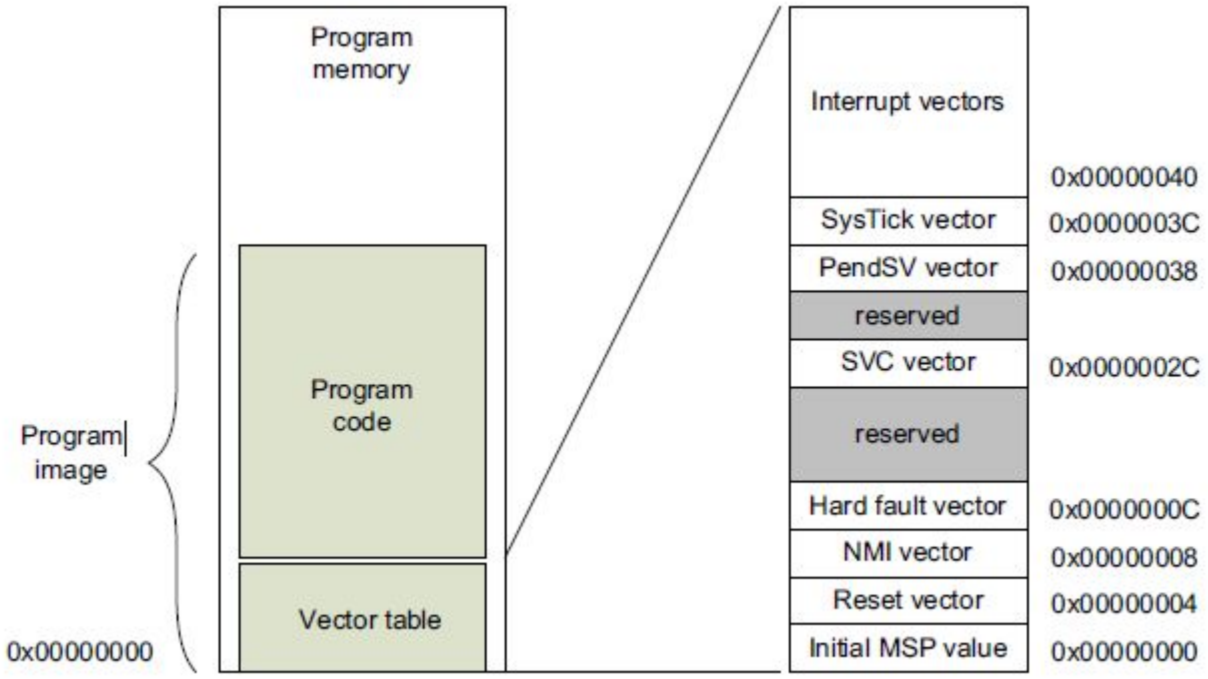


Figure 3.11:
Vector table in a program image.

- The vector table also defines the initial value of the main stack pointer (MSP). This is stored in the first word of the vector table.
- When the processor exits from reset, it will first read the first two-word addresses in the vector table.
- The first word is the initial MSP value, and the second word is the reset vector(Figure 3.12), which determines the starting of the program execution address (reset handler). which determines the starting of the program execution address

RESET SEQUENCE

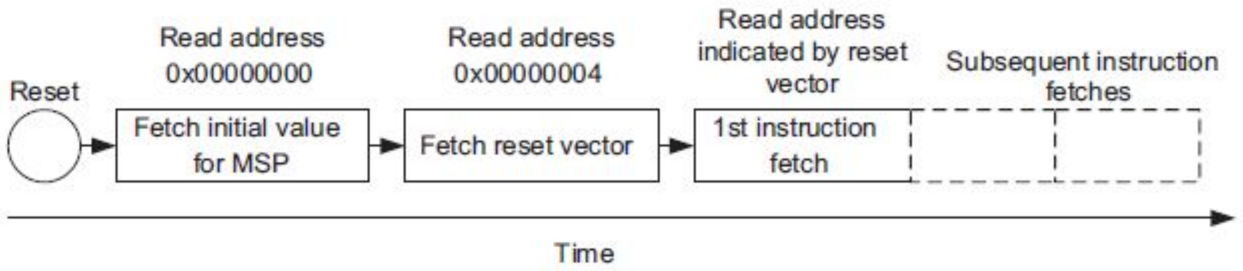
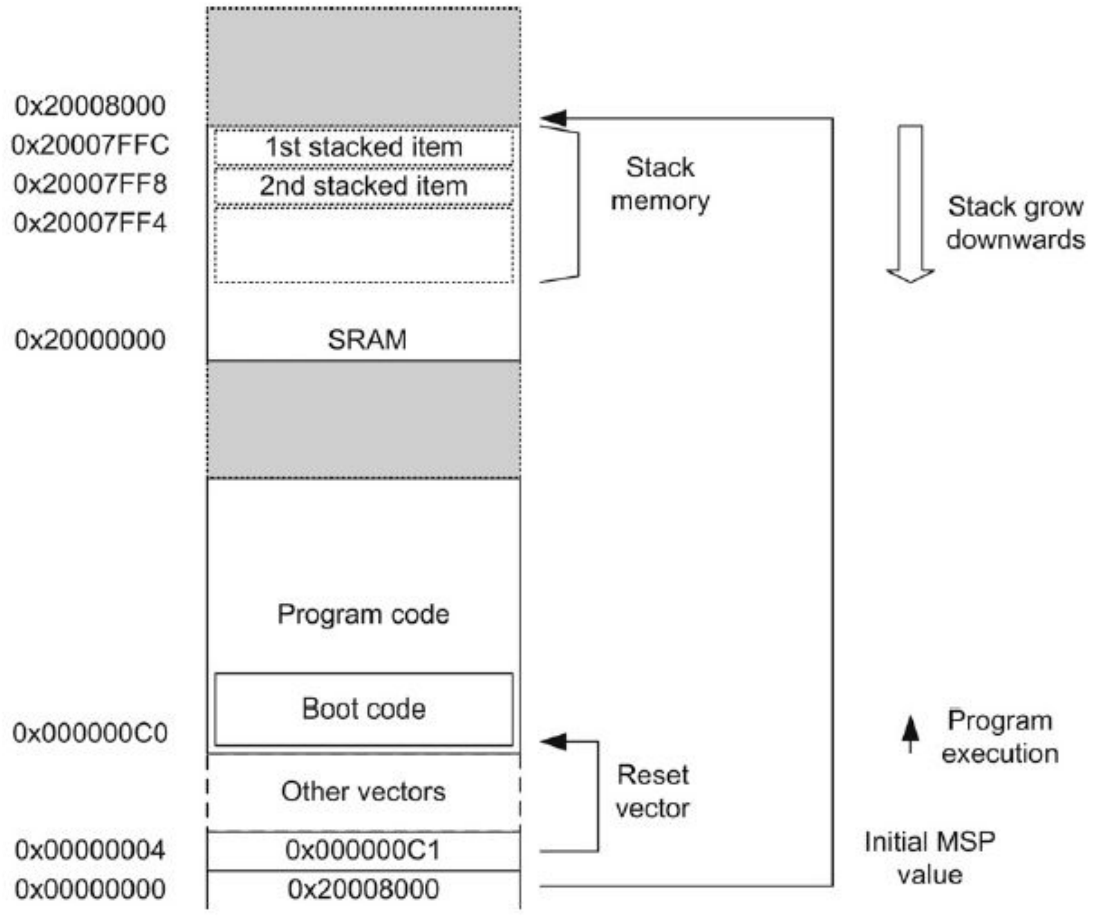

Reset
Read address 0x00000000
Fetch initial value for MSP
Read address 0x00000004
Fetch reset vector
Read address indicated by reset vector
1st instruction fetch
Subsequent instruction fetches
Time

Figure 3.12:
Reset sequence.

HOW RESETTING and INTIATION OF BOOT CODE STARTS



- For example, if we have boot code starting from address 0x000000C0, we need to put this address value in the reset vector location with the LSB set to 1 to indicate that it is Thumb code.
- Therefore, the value in address 0x00000004 is set to 0x000000C1 (Figure 3.13).
- After the processor fetches the reset vector, it will start executing program code from the address foundthere..

- This behavior is different from traditional ARM processors (e.g., ARM7TDMI), where
- the processor executes the program starting from address 0x00000000, and the vectors in the vector table are instructions as oppose to address values in the Cortex-M processors

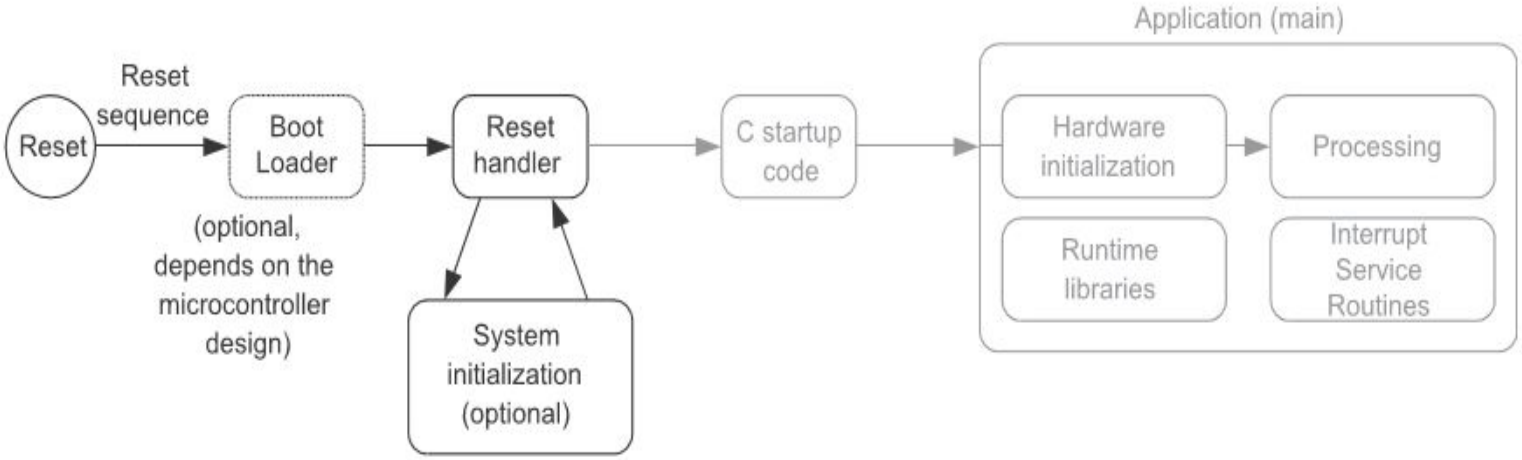

Reset
Reset sequence
Boot Loader
(optional, depends on the microcontroller design)
Reset handler
System initialization (optional)
C startup code
Application (main)
Hardware initialization
Processing
Runtime libraries
Interrupt Service Routines

Figure 4.1:

- Most modern microcontrollers have on-chip flash memory to hold the compiled program.
- The flash memory holds the program in binary machine code format, and therefore programs written in C must be compiled before programmed to the flash memory.
- Some of these microcontrollers might also have a separate boot ROM, which contains a small boot loader program that is executed when the microcontroller starts, before executing the user program in the flash memory.
- In most cases, only the program code in the flash memory can be changed and the boot loader is fixed. After the flash memory (or other types of program memory) is programmed, the program is then accessible by the processor.
- After the processor is reset, it carries out the reset sequence, as outlined at the end of the previous chapter (Figure 4.1).

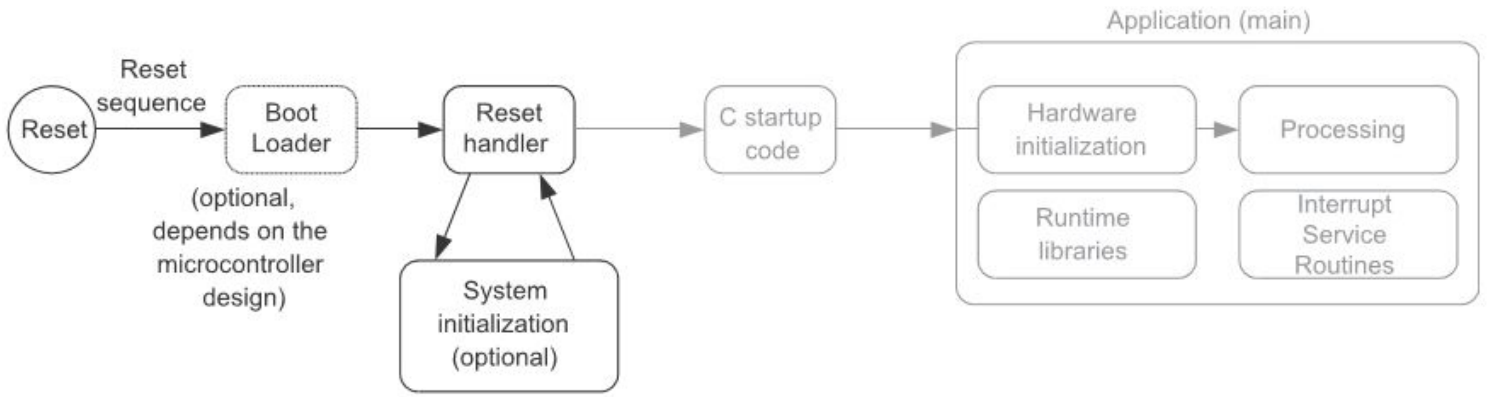

Reset
Reset sequence
Boot Loader
(optional, depends on the microcontroller design)
Reset handler
System initialization (optional)
C startup code
Application (main)
Hardware initialization
Processing
Runtime libraries
Interrupt Service Routines

Figure 4.1:

- In the reset sequence, the processor obtains the initial MSP value and reset vector, and then it executes the reset handler.
- All of this required information is usually stored in a program file called startup code.
- The reset handler in the startup code might also perform system initialization (e.g., clock control circuitry and Phase Locked Loop [PLL]), although in some cases system initialization is carried out later when the C program “main()” starts. Example startup code can usually be found in the installation of the development suite or from software packages available from the microcontroller vendors.

Continueed..

- After the C startup code is executed, the application starts. The application program often contains the following elements: •
- Initialization of hardware (e.g., clock, PLL, peripherals)
- The processing part of the application
- Interrupt service routines

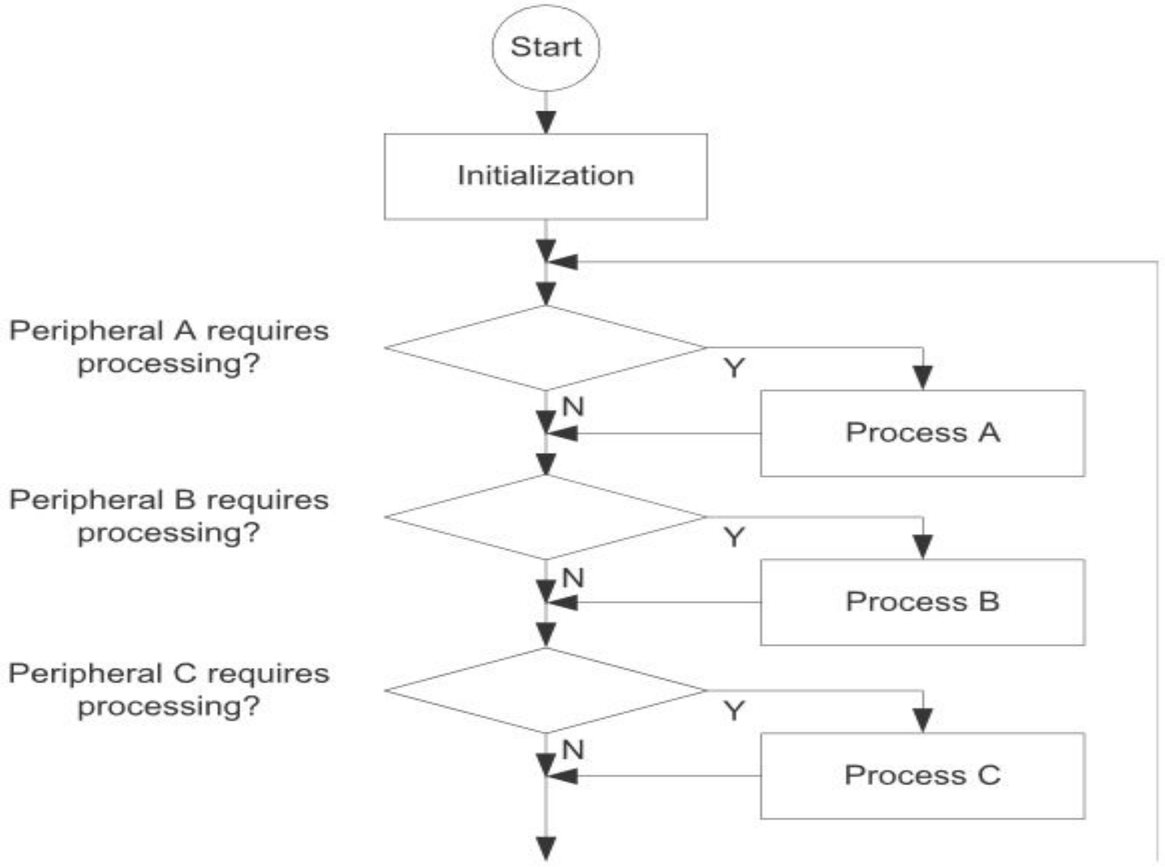
GPIO A PORT

```
•  PRESERVE8 ; Indicate the code here preserve
• ; 8 byte stack alignment
•                 THUMB     ; Indicate THUMB code is used
•               AREA    |.text|, CODE, READONLY
•
•             EXPORT main
• ; Start of CODE area
• main
•     ldr r1, =0x50004080
•    ldr r2, =0x08
•    ldr r3, =0x01
•    lsls r4, r3, #30
•
•    str r4, [r1]
•    ldr r4, =0x0f
•    adds r1, r1,r2
•    str r4, [r1]
• stop  b   stop
•       end
```

Designing Embedded Programs

- There are many ways to structure the flow of the application processing. Here we will cover a few fundamental concepts.
- Polling
- Interrupt Driven
- Combination of Polling and Interrupt Driven.
- Handling Concurrent Processes.

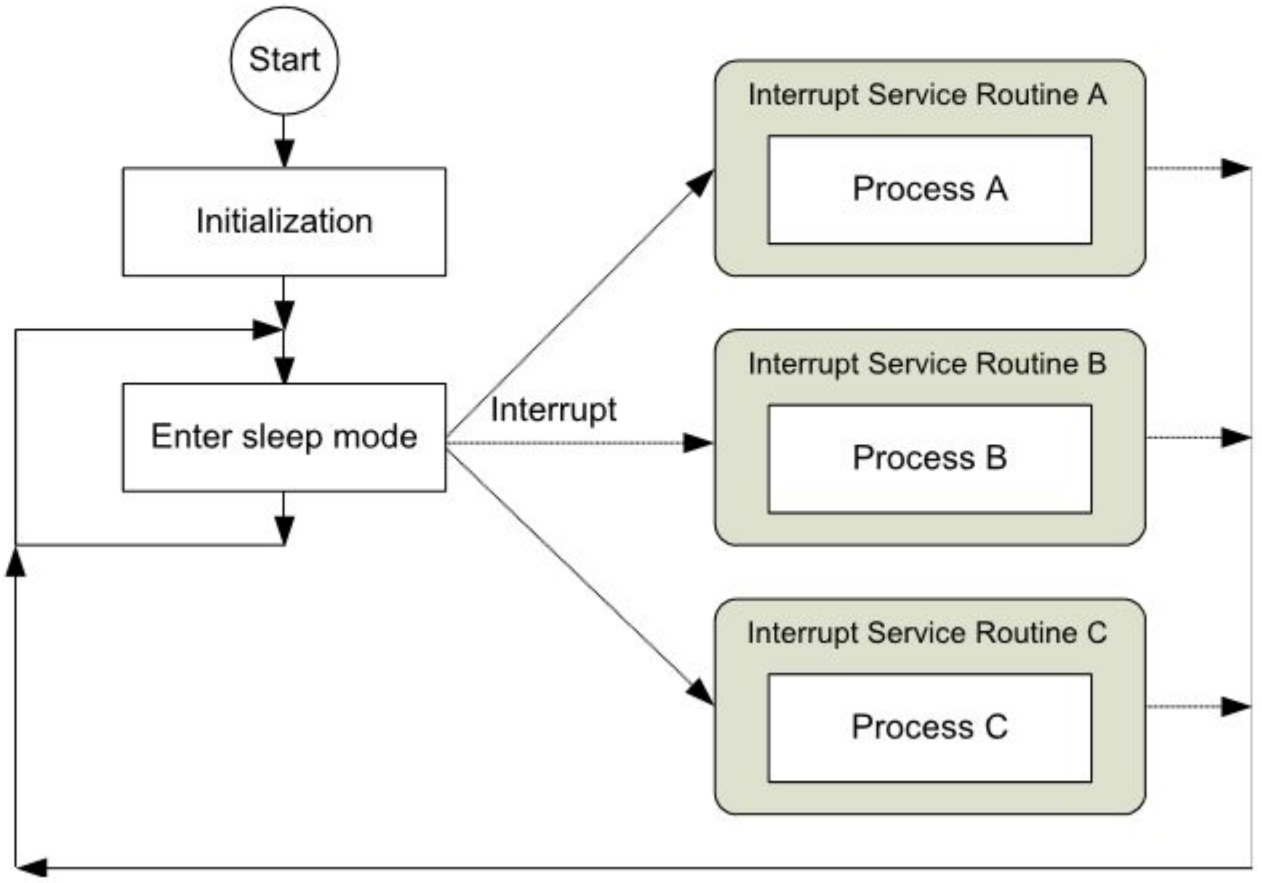
Polling



Start
Initialization
Peripheral A requires processing?
Y
N
Process A
Peripheral B requires processing?
Y
N
Process B
Peripheral C requires processing?
Y
N
Process C

Figure 4.4:

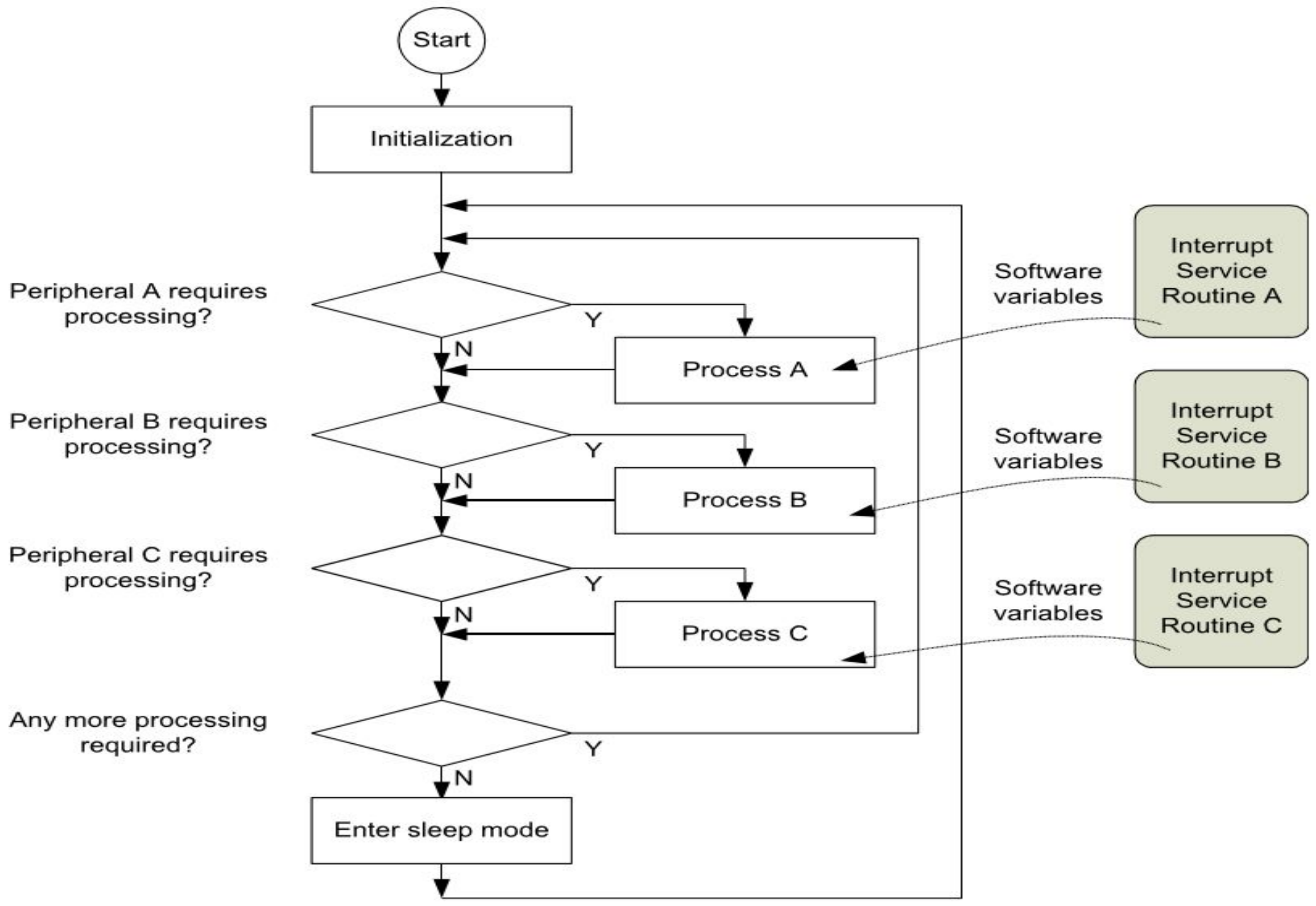
Interrupt Driven


Start
Initialization
Enter sleep mode
Interrupt
Interrupt Service Routine A
Process A
Interrupt Service Routine B
Process B
Interrupt Service Routine C
Process C

Interrupt Driven

- In applications that require lower power, processing can be carried out in interrupt service routines so that the processor can enter sleep mode when no processing is required. Interrupts are usually generated by external sources or on chip peripherals to wake up the processor. In interrupt-driven applications (Figure 4.5),
- the interrupts from different devices can be set at different priorities. In this way a high-priority interrupt request can obtain service even when a lower-priority interrupt service is running, which will be temporarily stopped. As a result, the latency for the higher-priority interrupt is reduced.

Combination of Interrupt Driven and Polling


Start
Initialization
Peripheral A requires processing?
Y
N
Process A
Peripheral B requires processing?
Y
N
Process B
Peripheral C requires processing?
Y
N
Process C
Any more processing required?
Y
N
Enter sleep mode
Software variables
Interrupt Service Routine A
Software variables
Interrupt Service Routine B
Software variables
Interrupt Service Routine C

Handling Concurrent Processes

- In some cases, an application process could take a significant amount of time to complete and therefore it is undesirable to handle it in a big loop as shown in Figure 4.6.
- If process A takes too long to complete, processes B and C will not able to respond to peripheral requests fast enough, resulting in system failure.
- Common solutions are as follows:
- 1. Breaking down a long processing task to a sequence of states. Each time the process is accessed, only one state is executed.
- 2. Using a real-time operating system (RTOS) to manage multiple tasks.

Handling Concurrent Process.

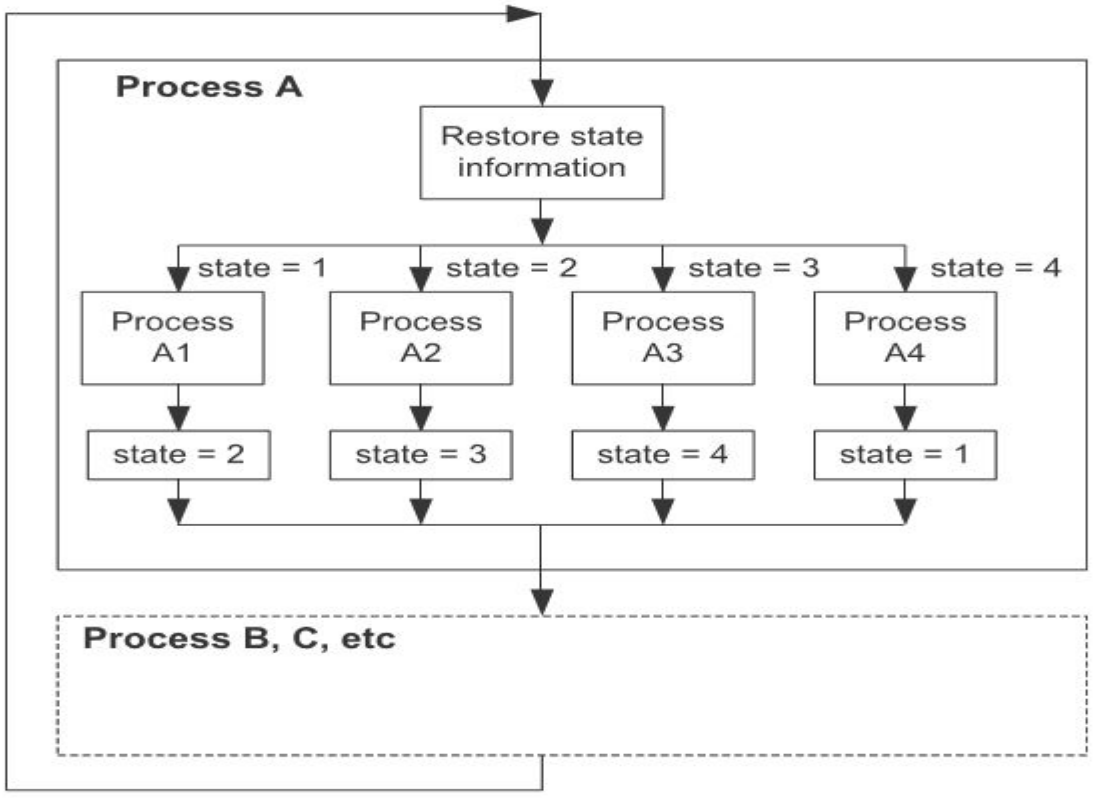


Figure 4.7:

Cortex Microcontroller Software Interface Standard (CMSIS)

- Introduction of CMSIS As the complexity of embedded systems increase, the compatibility and reusability of software code becomes more important.
- Having reusable software often reduces development time for subsequent projects and hence speeds up time to market, and software compatibility helps the use of third-party software components.
- For example, an embedded system project might involve the following software components:
  - Software from in-house software developers
  - Software reused from other projects
  - Device driver libraries from microcontroller vendors
  - Embedded OS
  - Other third-party software products like a communication protocol stack and codec (compressor/decompressor)
- The use of the third-party software components is becoming more and more common. With all these software components being used in one project, compatibility is becoming critical for many large-scale software projects.

- To allow a high level of compatibility between these software products and improve software portability, ARM worked with various microcontroller vendors and software solution providers to develop the CMSIS, a common software framework covering most Cortex-M processors and Cortex-M microcontroller products (Figure 4.16).
- The CMSIS is implemented as part of device driver library from microcontroller vendors. It provides a standardized software interface to the processor features like NVIC control and system control functions. Many of these processors feature access functions are available in CMSIS for the Cortex-M0, Cortex-M3 and Cortex-M4, allowing easy software porting between these processors.

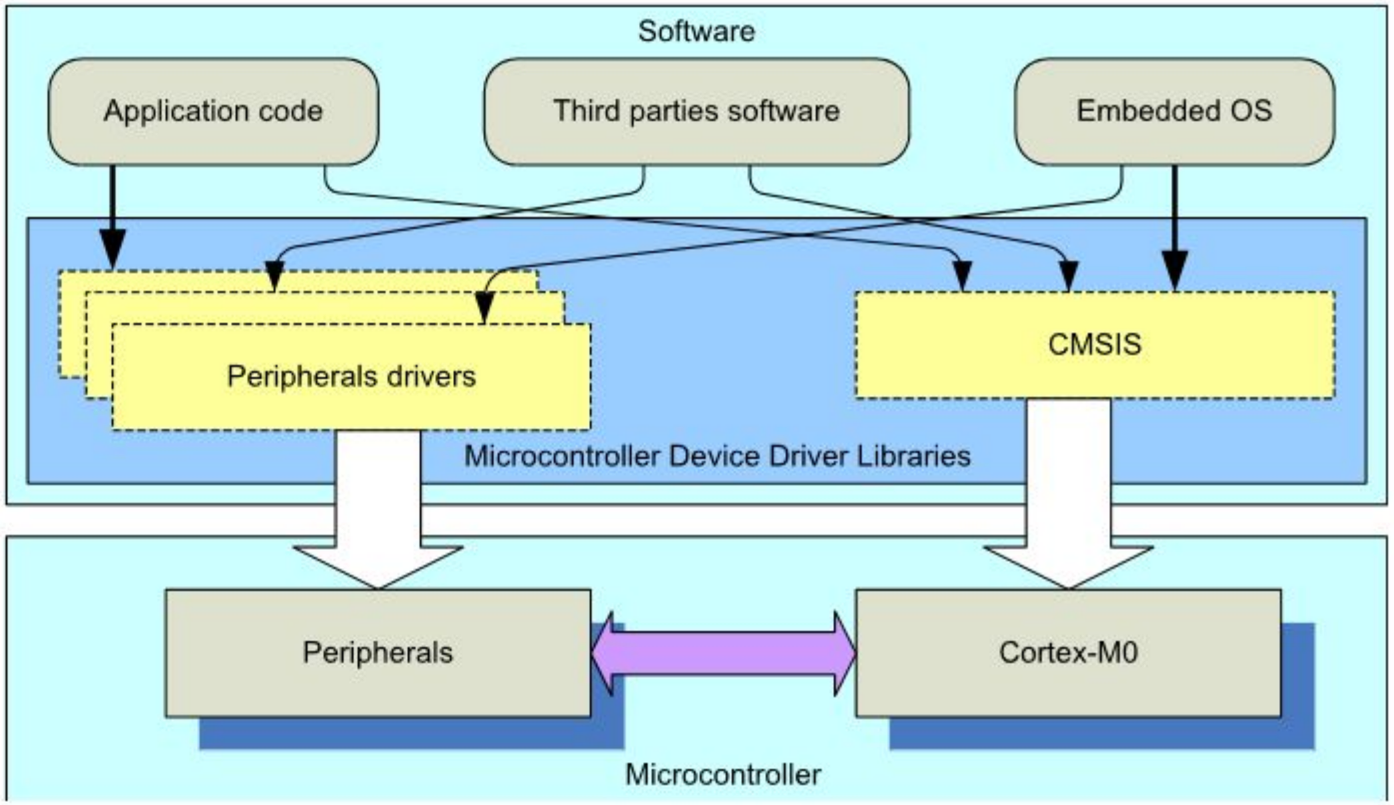

Software
Application code
Third parties software
Embedded OS
Peripherals drivers
CMSIS
Microcontroller Device Driver Libraries
Peripherals
Cortex-M0
Microcontroller

Figure 4.16:

What Is Standardized in CMSIS

- The CMSIS standardized the following areas for embedded software:
- Standardized access functions for accessing NVIC,
- System Control Block (SCB), and System Tick timer (SysTick) such as interrupt control and SysTick initialization.
- Standardized register definitions for NVIC, SCB, and SysTick registers.
- For best software portability, we should use the standardized access functions. However, in some cases we need to directly access the registers in NVIC, SCB, or the SysTick.
- In such cases, the standardized register definitions help the software to be more portable.
- Standardized functions for accessing special instructions in Cortex-M microcontrollers. Some instructions on the Cortex-M microcontroller cannot be generated by normal C code. If they are needed, they can be generated by these functions provided in CMSIS. Otherwise, users will have to use intrinsic functions provided by the C compiler or embedded/inline assembly language, which are tool chain specific and less portable.

Standardization

- Standardized names for system exceptions handlers. An embedded OS often requires system exceptions. By having standardized system exception handler names, supporting different device driver libraries in an embedded OS is much easier.
- Standardized name for the system initialization function. The common system initializa- tion function "void SystemInit(void)" makes it easier for software developers to set up their system with minimum effort.
- Standardize variable for clock speed information. A standardized software variable called "SystemFreq" (CMSIS v1.00 to v1.20) or "SystemCoreClock" (CMSIS v1.30 or newer). This is used to determine the processor clock frequency.

CMSIS

- The CMSIS also provides the following:
- A common platform for device driver libraries each device driver library has the same look and feel, making it easier for beginners to learn and making it easier for software porting.
- In future release of CMSIS, it could also provide a set of common communication access functions so that middleware that has been developed can be reused on different devices without porting.

Organization of the CMSIS

- The CMSIS is divided into multiple layers:
- Core Peripheral Access Layer

  -Name definitions, address definitions, and helper functions to access core registers and core peripherals like the NVIC, SCB, and SysTick.
- Middleware Access Layer (work in progress)

  -Common method to access peripherals for typical embedded systems Targeted at communication interfaces including UART, Ethernet, and SPI

  -Allows embedded software to be used on any Cortex microcontrollers that support the required communication interface
- Device Peripheral Access Layer (MCU specific)

  - Register name definitions, address definitions, and device driver code to access peripherals
- Access Functions for Peripherals (MCU specific) - Optional helper functions for peripherals

CMSIS Structure

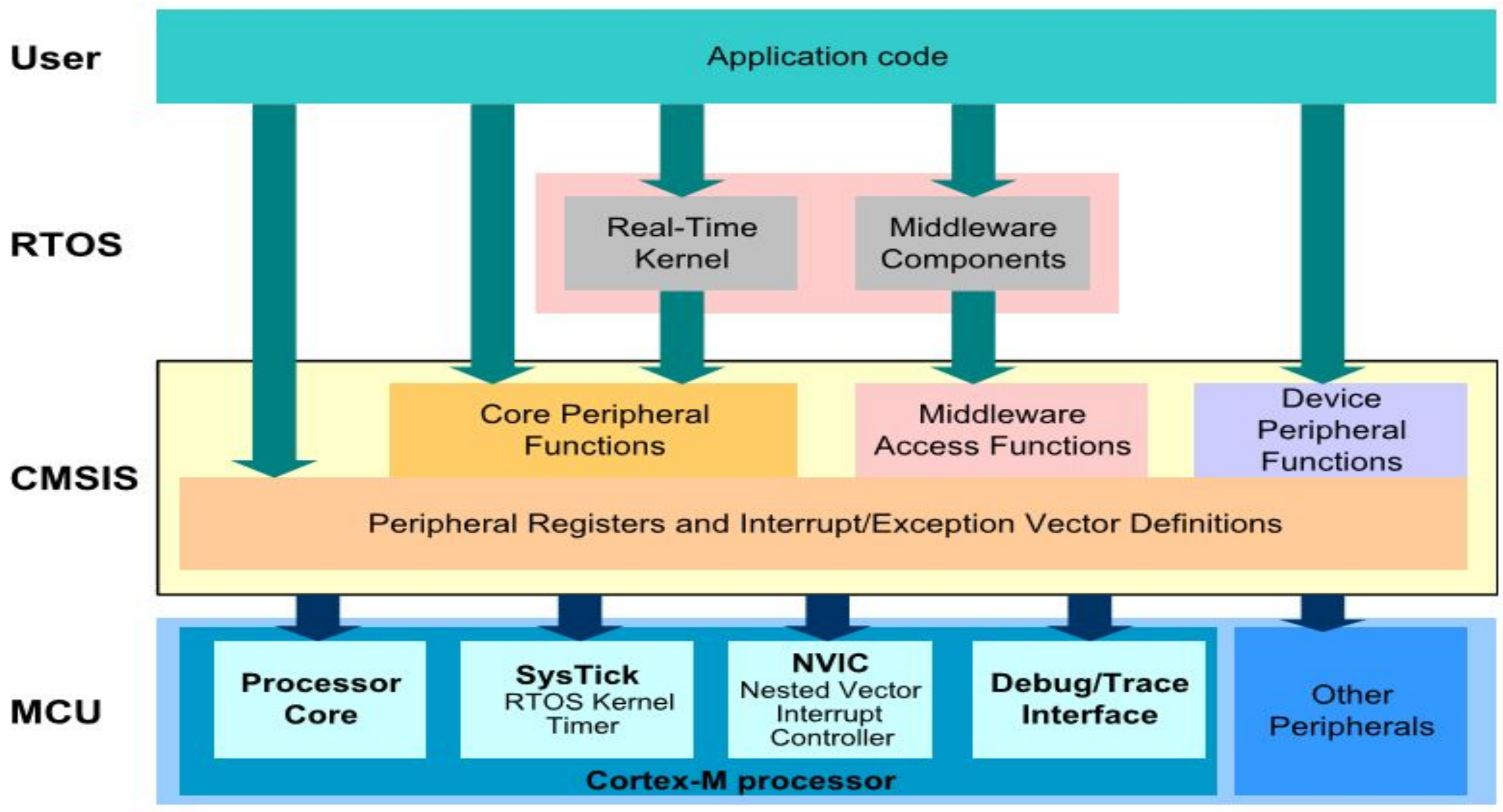
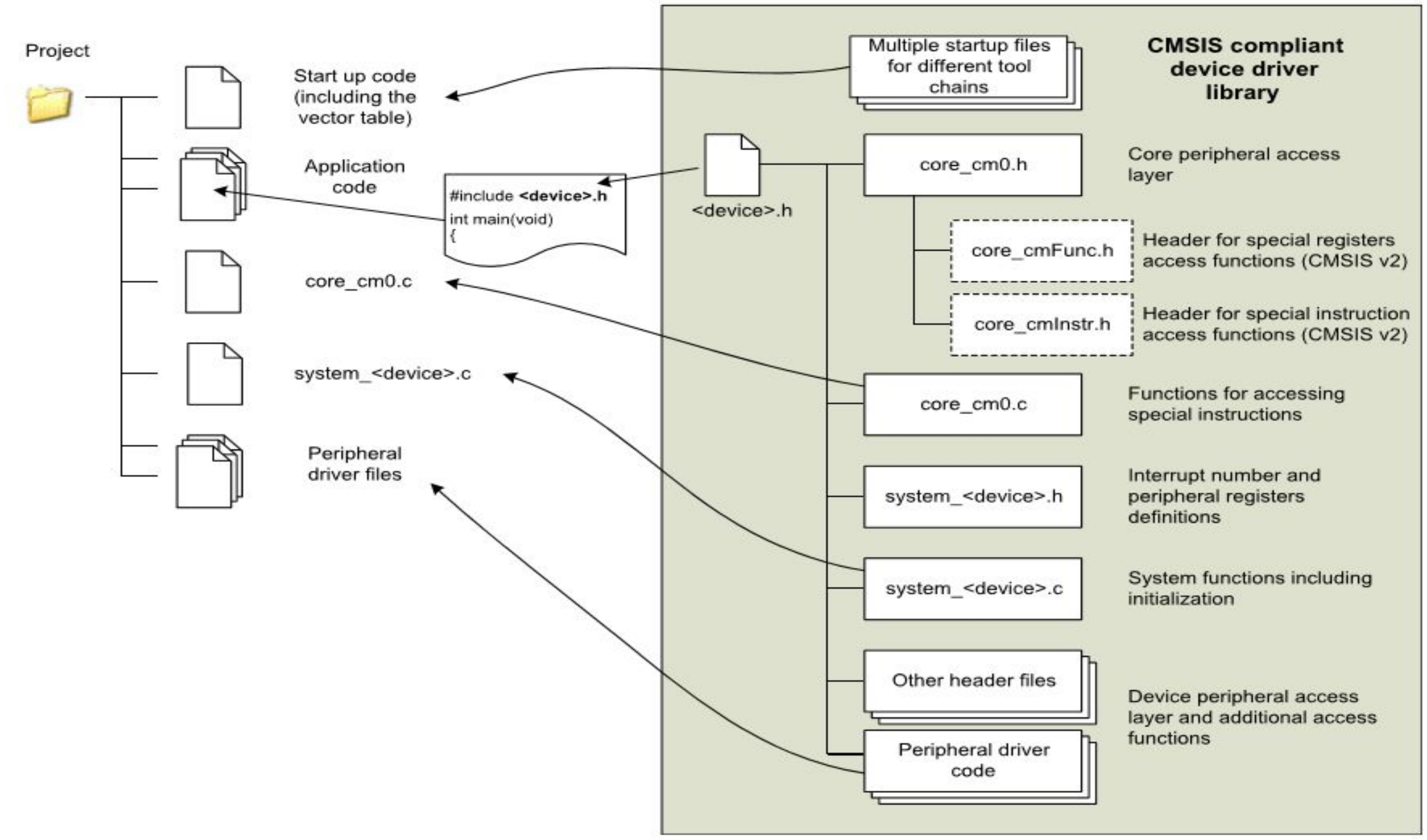

User
Application code
RTOS
Real-Time Kernel
Middleware Components
CMSIS
Core Peripheral Functions
Middleware Access Functions
Device Peripheral Functions
Peripheral Registers and Interrupt/Exception Vector Definitions
MCU
Processor Core
SysTick RTOS Kernel Timer
NVIC Nested Vector Interrupt Controller
Debug/Trace Interface
Other Peripherals
Cortex-M processor

Figure 4.17:
CMSIS structure.

- Using CMSIS The CMSIS is an integrated part of the device driver package provided by the microcontroller vendors.
- If you are using the device driver libraries for software development, you are already using theCMSIS.
- If you are not using device driver libraries from microcontroller vendors, you can still useCMSIS by downloading theCMSIS package from OnARM web site (www.onarm. com), unpacking the files, and adding the required files for your project.
- For C program code, normally you only need to include one header file provided in the device driver library from your microcontroller vendor. This header file then pulls in the all the required header files for CMSIS features as well as peripheral drivers.

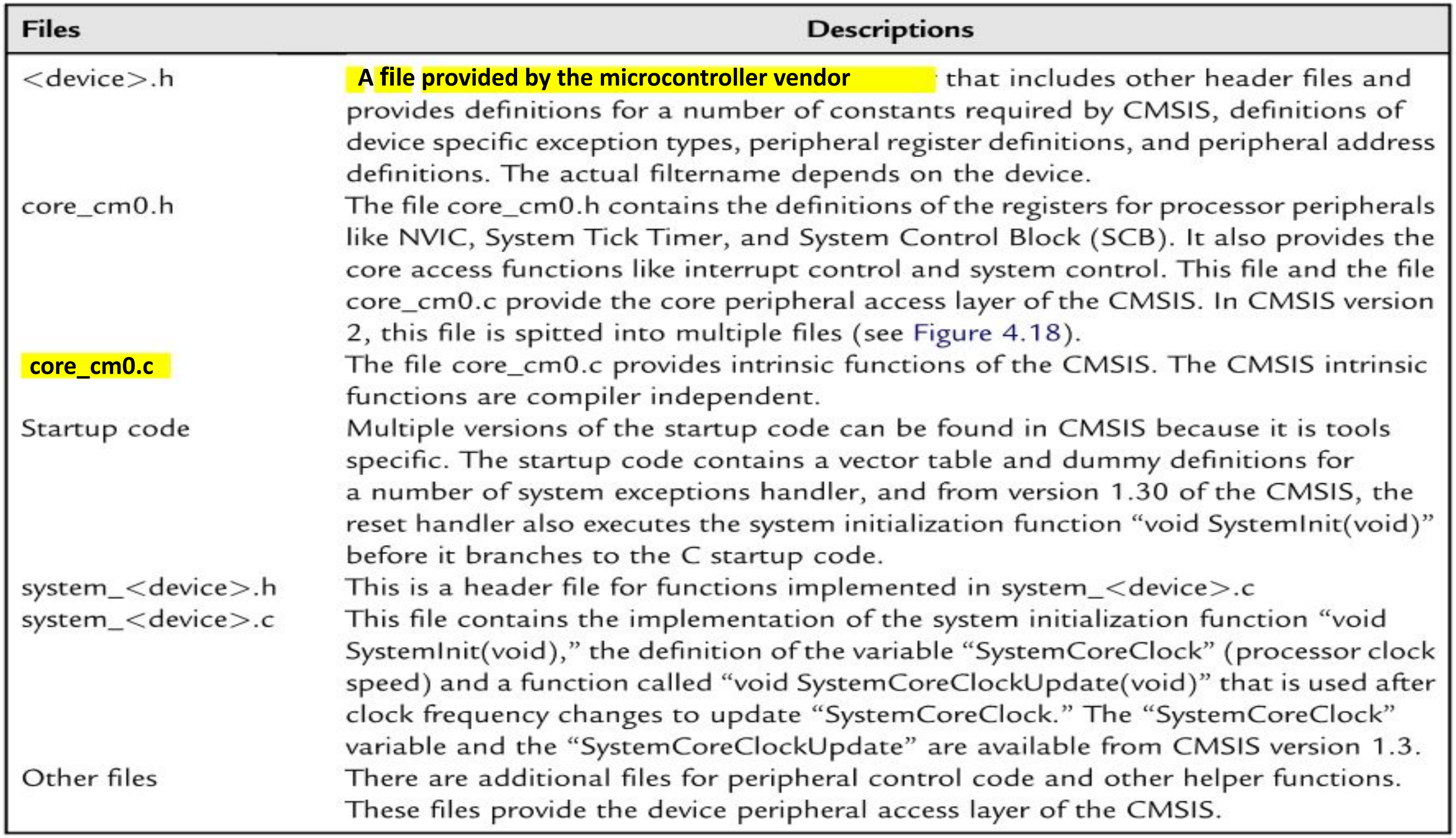
- You also need to include the CMSIS-compliant startup code, which can be either in C or assembly code. CMSIS provides various versions of startup code customized for different tool chains.
- Figure 4.18 shows a simple project setup using theCMSIS package. The name of some the files depends on the actual microcontroller device name (indicated as <device> in Figure 4.18). When you use the header file provided in the device driver library, it automatically includes the other required header files for you (Table 4.4).

Project


Project
Start up code (including the vector table)
Application code
core_cm0.c
system_<device>.c
Peripheral driver files
#include <device>.h
int main(void)
{
CMSIS compliant device driver library
Multiple startup files for different tool chains
<device>.h
core_cm0.h
Core peripheral access layer
core_cmFunc.h
Header for special registers access functions (CMSIS v2)
core_cmInstr.h
Header for special instruction access functions (CMSIS v2)
core_cm0.c
Functions for accessing special instructions
system_<device>.h
Interrupt number and peripheral registers definitions
system_<device>.c
System functions including initialization
Other header files
Peripheral driver code
Device peripheral access layer and additional access functions

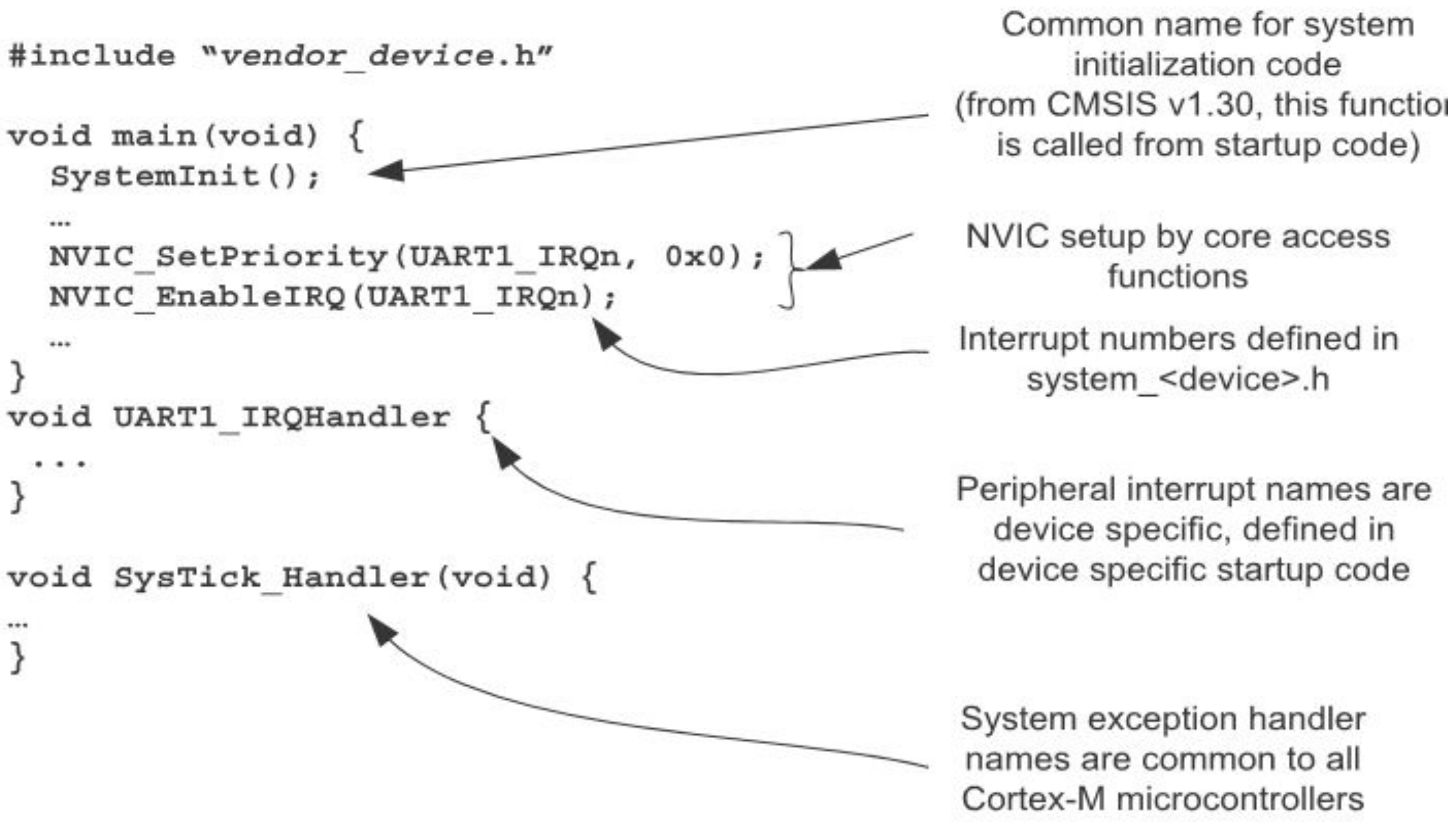
Files in CMSIS

Table 4.4: Files in CMSIS



Files	Descriptions
<device>.h	A file provided by the microcontroller vendor that includes other header files and provides definitions for a number of constants required by CMSIS, definitions of device specific exception types, peripheral register definitions, and peripheral address definitions. The actual filtername depends on the device.
core_cm0.h	The file core_cm0.h contains the definitions of the registers for processor peripherals like NVIC, System Tick Timer, and System Control Block (SCB). It also provides the core access functions like interrupt control and system control. This file and the file core_cm0.c provide the core peripheral access layer of the CMSIS. In CMSIS version 2, this file is spitted into multiple files (see Figure 4.18).
core_cm0.c	The file core_cm0.c provides intrinsic functions of the CMSIS. The CMSIS intrinsic functions are compiler independent.
Startup code	Multiple versions of the startup code can be found in CMSIS because it is tools specific. The startup code contains a vector table and dummy definitions for a number of system exceptions handler, and from version 1.30 of the CMSIS, the reset handler also executes the system initialization function "void SystemInit(void)" before it branches to the C startup code.
system_<device>.h	This is a header file for functions implemented in system_<device>.c
system_<device>.c	This file contains the implementation of the system initialization function "void SystemInit(void)," the definition of the variable "SystemCoreClock" (processor clock speed) and a function called "void SystemCoreClockUpdate(void)" that is used after clock frequency changes to update "SystemCoreClock." The "SystemCoreClock" variable and the "SystemCoreClockUpdate" are available from CMSIS version 1.3.
Other files	There are additional files for peripheral control code and other helper functions. These files provide the device peripheral access layer of the CMSIS.

Vendor_device.h



```
#include "vendor_device.h"

void main(void) {
  SystemInit();
  …
  NVIC_SetPriority(UART1_IRQn, 0x0);
  NVIC_EnableIRQ(UART1_IRQn);
  …
}
void UART1_IRQHandler {
 ...
}

void SysTick_Handler(void) {
…
}
```

Common name for system initialization code (from CMSIS v1.30, this functio is called from startup code)

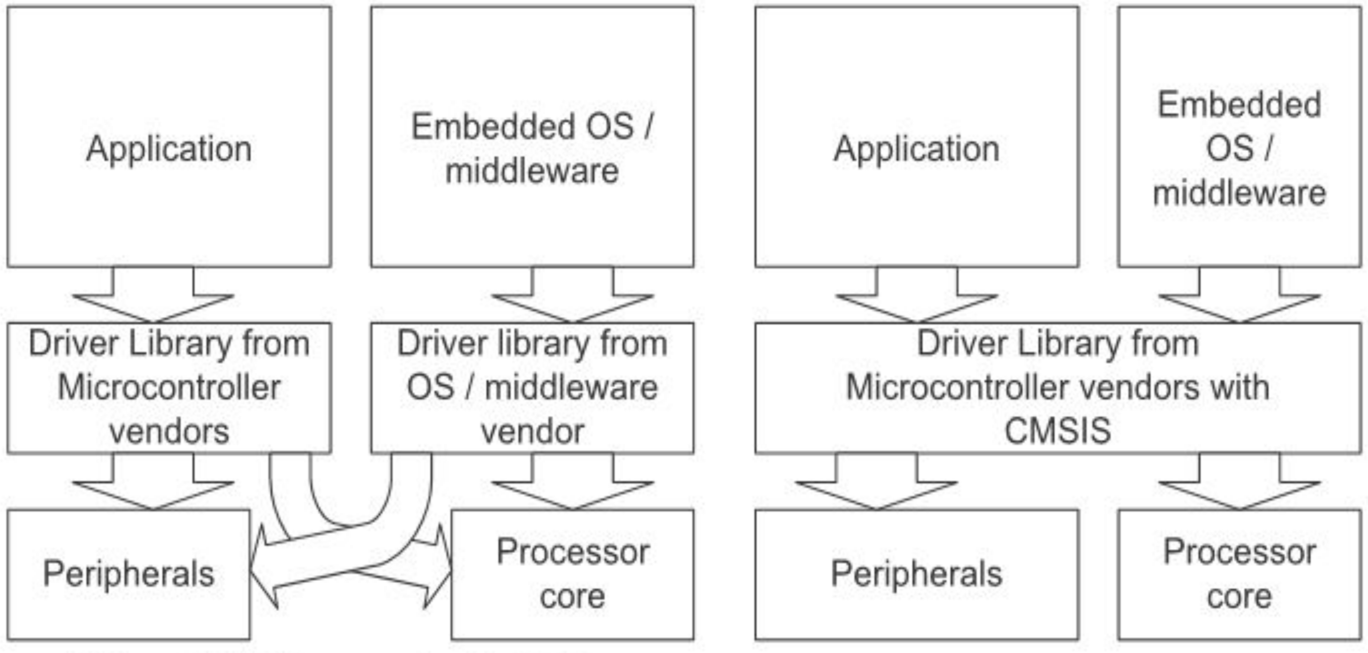
NVIC setup by core access functions

Interrupt numbers defined in system_<device>.h

Peripheral interrupt names are device specific, defined in device specific startup code

System exception handler names are common to all Cortex-M microcontrollers

CMSIS avoids overlapping of driver code.


Application
Embedded OS / middleware
Driver Library from Microcontroller vendors
Driver library from OS / middleware vendor
Peripherals
Processor core
Application
Embedded OS / middleware
Driver Library from Microcontroller vendors with CMSIS
Peripherals
Processor core

Without CMSIS, an embedded OS or middleware needs to include processor core access functions, and might need to include a few peripheral drivers.

With CMSIS, an embedded OS or middleware can use standardized core access functions from a driver library