CORTEX MO TECHNICAL OVERVIEW

The Cortex-M0 processor is a 32-bit Reduced Instruction Set Computing (RISC) processor with a von Neumann architecture (single bus interface).

It uses an instruction set called Thumb, which was first supported in the ARM7TDMI processor; however, several newer instructions from the ARMv6 architecture and a few instructions from the Thumb-2 technology are also included.

Thumb-2 technology extended the previous Thumb instruction set to allow all operations to be carried out in one CPU state.

The instruction set in Thumb-2 included both 16-bit and 32-bit instructions;

most instructions generated by the C compiler use the 16-bit instructions, and the 32-bit instructions are used when the 16-bit version cannot carry out the required operations.

This results in high code density and avoids the overhead of switching between two instruction sets.

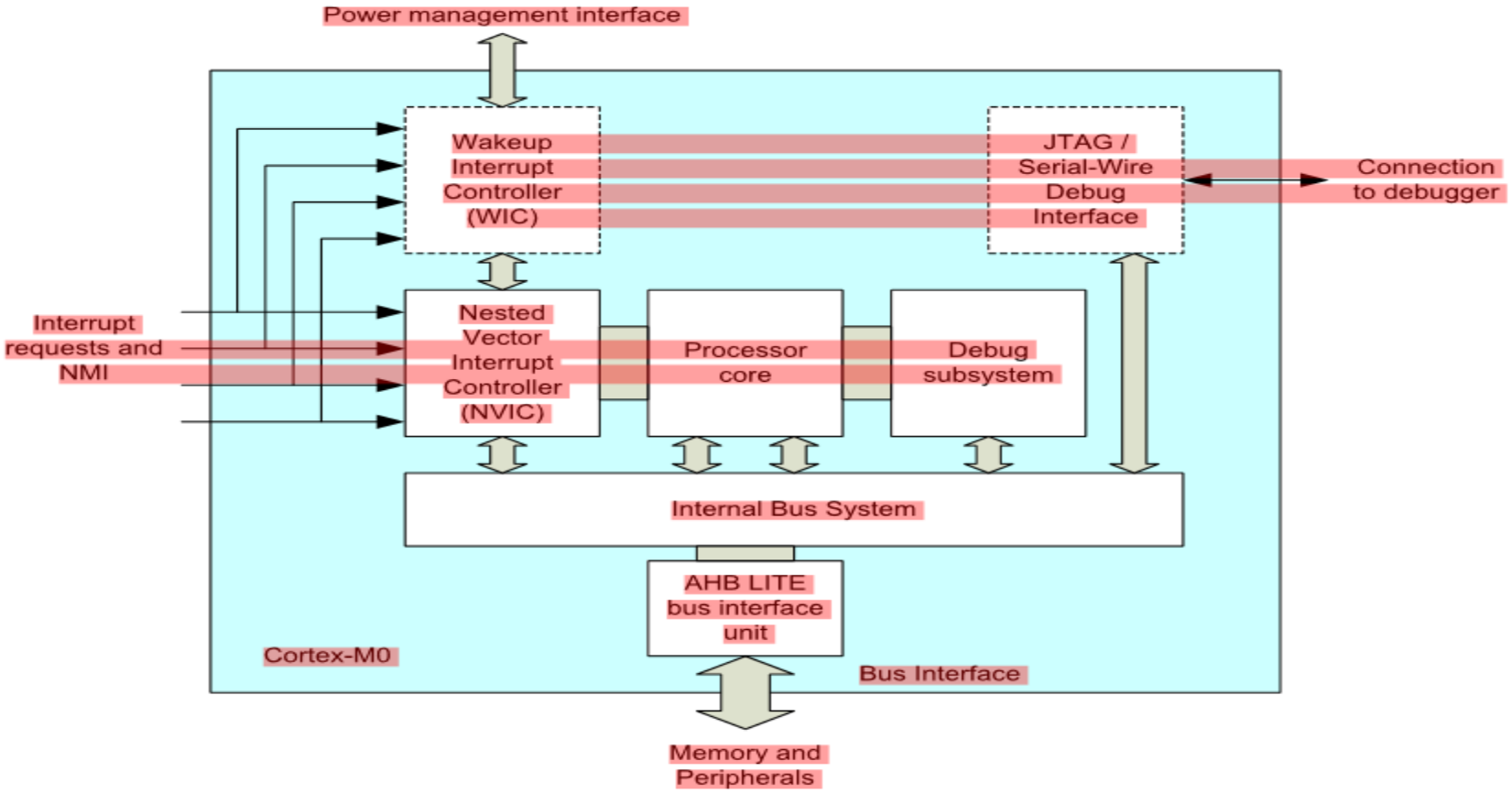
In total, the Cortex-M0 processor supports only 56 base instructions, although some instructions can have more than one form. Although the instruction set is small, the Cortex- M0 processor is highly capable because the Thumb instruction set is highly optimized.

Academically, the Cortex-M0 processor is classified as load-store architecture

Introduction to Cortex-M0 Programming Introduction to Embedded System

- 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

Simplified Block Diagram of Cortex M0



Continued….

- The processor core contains the register banks, ALU, data path, and control logic.
- It is a three- stage pipeline design with fetch stage, decode stage, and execution stage.
- The register bank has sixteen 32-bit registers. A few registers have special usages.
- The Nested Vectored Interrupt Controller (NVIC) accepts up to 32 interrupt request signals and a nonmaskable interrupt (NMI) input.
- It contains the functionality required for comparing priority between interrupt requests and the current priority level so that nested interrupts can be handled automatically.
- If an interrupt is accepted, it communicates with the processor so that the processor can execute the correct interrupt handler.
-

Continue…

- When an interrupt request is detected,the WIC informs the power management to power up the system so that the NVIC and the processor core can then handle the rest of the interrupt processing.
- The debug subsystem contains Various functional blocks to handle debug control, program breakpoints, and data watchpoints.
- When a debug event occurs, it can put the processor core in a halted state so that embedded developers can examine the status of the processor at that point.
- The JTAG or serial wire interface units provide access to the bus system and debugging functionalities.
- The JTAG protocol is a popular five-pin communication protocol commonly used for testing.
- The serial wire protocol is a newer communication protocol that only requires two wires, but it can handle the same debug functionalities as JTAG.
- The internal bus system, the data path in the processor core, and the AHB LITE bus interface are all 32 bits wide.

- The Wakeup Interrupt Controller (WIC) is an optional unit. In low-power applications, the microcontroller can enter standby state with most of the processor powered down. In this situation, the WIC can perform the function of interrupt masking while the NVIC and the processor core are inactive.
- AHB-Lite is an on-chip bus protocol used in many ARM processors.
- This bus protocol is part of the Advanced Microcontroller Bus Architecture (AMBA) specification, a bus architecture developed by ARM that is widely used in the IC design industry.

- processor core are inactive. When an interrupt request is detected, the WIC informs the power
- management to power up the system so that the NVIC and the processor core can then handle
- the rest of the interrupt processing.
- The debug subsystem contains various functional blocks to handle debug control, program
- breakpoints, and data watchpoints. When a debug event occurs, it can put the processor core in
- a halted state so that embedded developers can examine the status of the processor at that point.

- The JTAG or serial wire interface units provide access to the bus system and debugging functionalities.
- The JTAG protocol is a popular five-pin communication protocol commonly used for testing.
- The serial wire protocol is a newer communication protocol that only requires two wires, but it can handle the same debug functionalities as JTAG.
- The internal bus system, the data path in the processor core, and the AHB LITE bus interface are all 32 bits wide. AHB-Lite(advanced High Performance Bus) is an on-chip bus protocol used in many ARM processors.
- This bus protocol is part of the Advanced Microcontroller Bus Architecture (AMBA) specification, a bus architecture developed by ARM that is wide

- interrupt handler is executed automatically without the need to determine the exception
- vector in software.
- • Interrupts can have four different programmable priority levels. The NVIC automatically
- handles nested interrupts.
- • Deterministic exception response timing. The design can be set up to respond
- to exceptions (e.g., interrupts) with a fixed number of cycles (constant interrupt
- latency arrangement) or to respond to the exception as soon as possible (minimum
- 16 clock cycles).

- Nonmaskable interrupt (NMI) input for safety critical systems.
- Architectural predefined memory map. The memory space of the Cortex-M0 processor

System Features

- Thumb instruction set. Highly efficient, high code density and able to execute all Thumb Instructions from the ARM7TDMI processor.
- High performance. Up to 0.9 DMIPS/MHz (Dhrystone 2.1million instructions per second/MHZ) with fast multiplier or 0.85 DMIPS/MHz with smaller multiplier.
- Built-in Nested Vectored Interrupt Controller (NVIC). This makes interrupt configuration and coding of exception handlers easy.
- When an interrupt request is taken, the corresponding is architecturally predefined to make software porting easier and to allow easier optimization of chip design.
- However, the arrangement is very flexible. The memory space is linear and there is no memory paging required like in a number of other processor architectures.
-

- Easy to use and C friendly. There are only two modes (Thread mode and Handler mode).
- The whole application, including exception handlers, can be written in C without any assembler.
- Built-in optional System Tick timer for OS support. A 24-bit timer with a dedicated exception type is included in the architecture, which the OS can use as a tick timer or as a general timer in other applications without an OS.
- SuperVisor Call (SVC) instruction with a dedicated SVC exception and PendSV (Pendable Supervisor service) to support various operations in an embedded OS.
- Architecturally defined sleep modes and instructions to enter sleep. The sleep features allow power consumption to be reduced dramatically.

Continued…

- Defining sleep modes as an architectural feature makes porting of software easier because sleep is entered by a specific instruction rather than implementation defined control registers.
- Fault handling exception to catch various sources of errors in the system.

Implementation Features

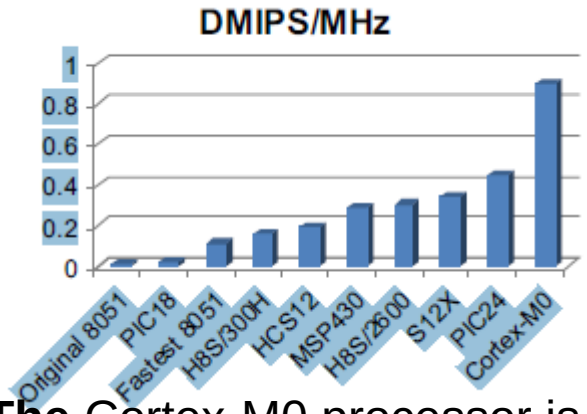
- Configurable number of interrupts (1 to 32)
- Fast multiplier (single cycle) or small multiplier (for a smaller chip area and lower power,32 cycles)
- Little endian or big endian memory support
- Optional Wakeup Interrupt Controller (WIC) to allow the processor to be powered down during sleep, while still allowing interrupt sources to wake up the system
- Very low gate count, which allows the design to be implemented in mixed signal
- semiconductor processes

Debug Features

- • Halt mode debug. Allows the processor activity to stop completely so that register values can be accessed and modified. No overhead in code size and stack memory size.
- CoreSight technology. Allows memories and peripherals to be accessed from the debugger without halting the processor. It also allows a system-on-chip design with multiple processors to share a single debug connection.
- Supports JTAG connection and serial wire debug connections. The serial wire debug protocol can handle the same debug features as the JTAG, but it only requires two wires and is already supported by a number of debug solutions from various tools vendors.

- Configurable number of hardware breakpoints (from 0 to maximum of 4) and watchpoints (from 0 to maximum of 2). The chip manufacturer defines this during implementation.
- Breakpoint instruction support for an unlimited number of software breakpoints.
- All debug features can be omitted by chip vendors to allow minimum size implementations.

Advantages of the Cortex-M0 Processor



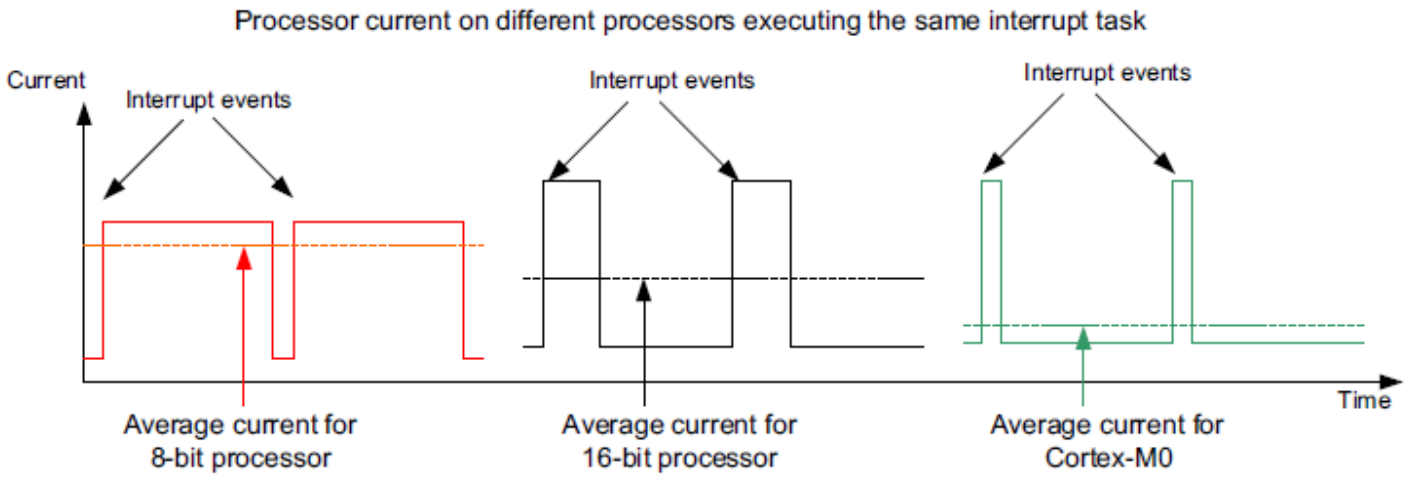
Energy Efficiency:The Cortex-M0 processor is about the same size as a typical 16-bit processor and possibly several times bigger than some of the 8-bit processors.
However, it has much better performance than 16-bit and 8-bit architectures

Performance of Cortex M0 in Terms of DMIPS/MHz

Table 2.1: Dhrystone Performance Data Based on Information Available on the Internet

Architecture	Estimated DMIPS/MHz with Dhrystone 2.1
Original 80C51	0.0094
PIC18	0.01966
Fastest 8051	0.113
H8S/300H	0.16
HCS12	0.19
MSP430	0.288
H8S/2600	0.303
S12X	0.34
PIC24	0.445
Cortex-M0	0.896 (if a small multiplier is used, the performance is 0.85)

Average Current for different processors



Microcontroller Current on different Architectures executing the same interrupt task

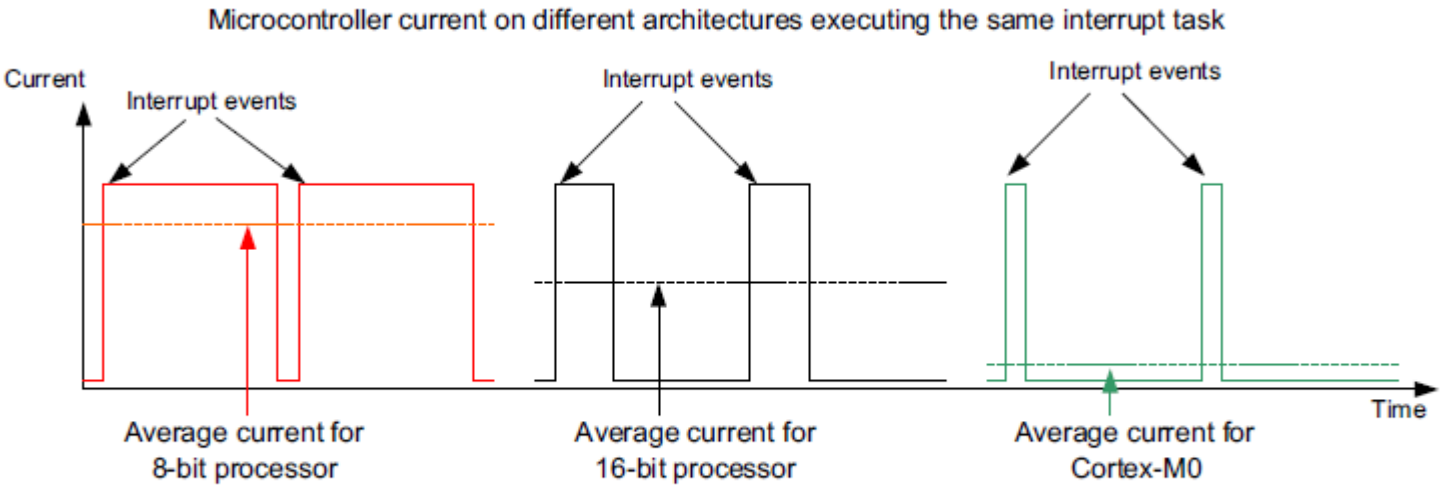


Figure 2.4:
At the chip level, the duty cycle of processor activity becomes more significant.

Limitations in 8-Bit and 16-Bit Architectures

- Another important reason to use the 32-bit Cortex-M0 processor rather than the traditional 16-bit or 8-bit architectures is that it does not have m
- The first obvious limitation of 8-bit and 16-bit architectures is memory size.
- Many 8-bit and 16-bit microcontrollers allow access to a larger memory range by dividing memory space into memory pages whereas ARM microcontrollers use 32-bit linear addresses.
- Another limitation of 8-bit microcontroller architectures can be the limitations of their instruction sets. For example, 8051 heavily relies on the accumulator register to handle data processing and memory transfers.
- This increases the code size because you need to keep transferring data into the accumulator and taking it out before and after operations

- Addressing modes account for another factor that limits performance in many 8-bit and 16-bit microcontrollers.
- A number of addressing modes are available in the Cortex-M0,

  allowingbetter code density and making it easier to use.
- The instruction set limitations on 8-bit and 16-bit architectures not only reduce the performance of the embedded system, but they also increase code size and hence increase power consumption, as a larger program memory is required.

Performance and Power Consumption V/s frequency Plot

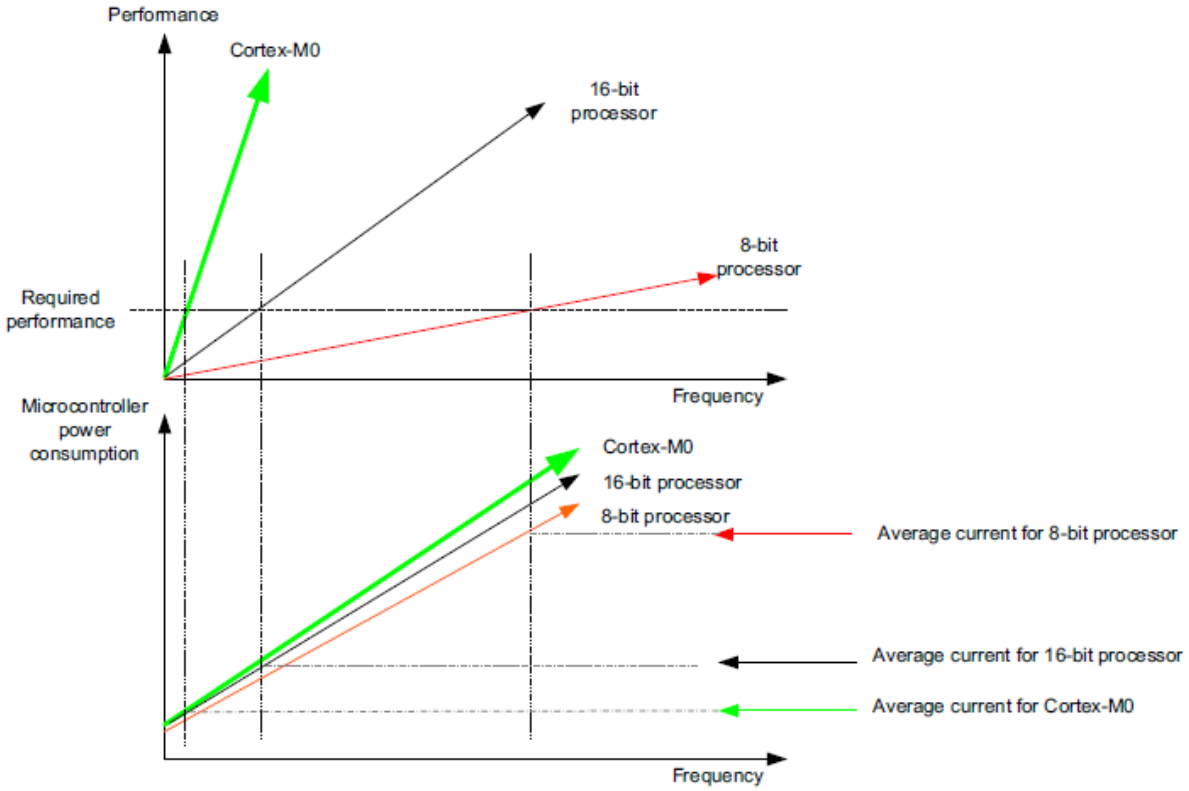


Figure 2.5:
The Cortex-M0 can provide lower power consumption by running at lower clock frequencies.

Easy to Use, Software Portability

- All the software code for the ARM Cortex microcontrollers can be written in C, allowing shorter software development time as well as improving software portability

Low-Power Applications

- Small gate count
- High efficiency
- Low-power features (sleep modes)
- Logic cell enhancement

Small gate count

- The Cortex-M0 processor’s small gate count characteristic directly reduces the active current and leakage current of the processor.
- During the development of the Cortex-M0 processor, various design techniques and optimizations were used to make the circuit size as small as possible.
- Each part of the design was carefully developed and reviewed to ensure that the circuit size is small (it is a bit likewriting an application program in assembly to achieve the best optimization).
- This allows the gate count to be 12k gates at minimum configuration.
- Typically, the gate count could be 17k to 25k gates when including more features.
- This is about the same size or smaller than typical 16-bit microprocessors, with more than double the system performance.

High efficiency

- By having a highly efficient architecture, embedded system designers can develop their product so that it has a lower clock frequency while still being able to provide the required performance, reducing the active current of the product.
- With a performance of 0.9DMIPS/ MHz, despite not being very high compared with some modern 32-bit processors, the Dhrystone benchmark result of Cortex-M0 is still higher than the older generation of 32-bit desktop processors like the 80486DX2 (0.81DMIPS/MHz), and it is a lot smaller.
- The high efficiency of the Cortex-M0 processor is mostly due to the efficiency of the Thumb instruction set, as well as highly optimized hardware implementation.

Low-Power Features

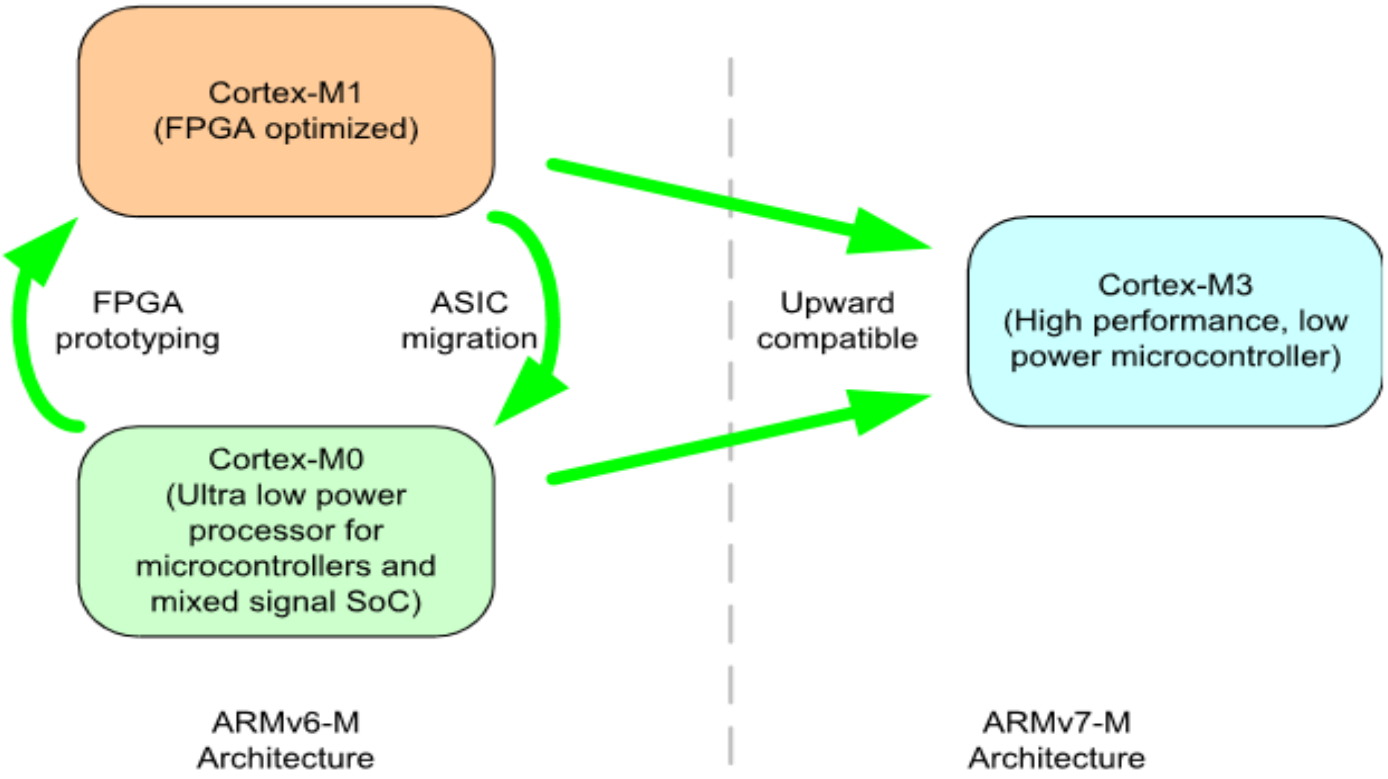
- The Cortex-M0 processors have a number of low-power features that allow embedded product developers to reduce the product's power consumption.
- First, the processor provides two sleep modes and they can be entered easily with "Wait-for-Interrupt" (WFI) or "Wait- for-Event" (WFE) instructions.
- The power management unit on the chip can use the sleep status of the processor to reduce the power consumption of the system.
- The Cortex-M0 processor also provides a "Sleep-on-Exit" feature, which causes the processor to run only when an interrupt service is required.
- In addition, the Cortex-M0 processor has been carefully developed so that some parts of the processor, like the debug system, can be switched off when not required.
-

- Apart from these normal sleep features, the Cortex-M0 processor also supports a unique feature called the Wakeup Interrupt Controller(WIC).
- This allows the processor to be powered down while still allowing interrupt events to power up the system and resume operation almost instantaneously when required. This greatly reduces the leakage current (static power consumption) of the system during sleep.

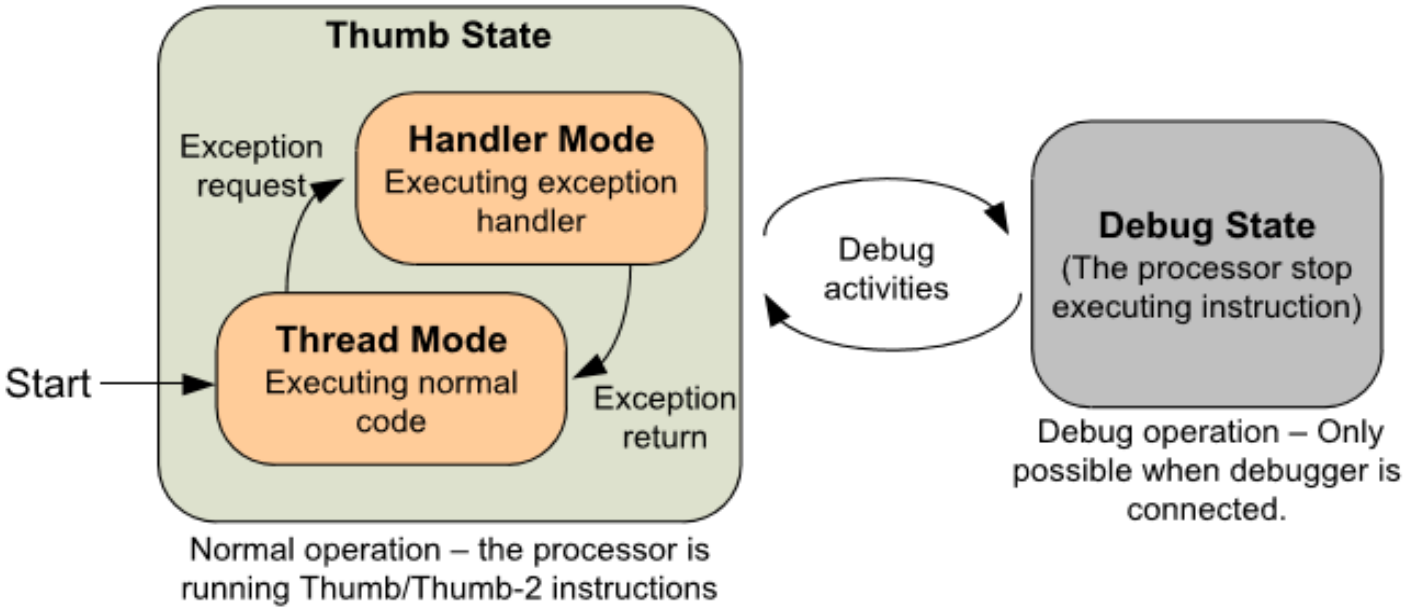
Logic Cell Enhancement

- In recent years, there have been enhancements in logic cell designs. Apart from pushing logic gate designs to smaller transistor sizes, the Physical IP (intellectual property) division in ARM has also been working hard to find innovative ways to reduce power consumption in embedded systems.
- One of the major developments is the introduction of the Ultra Low Leakage (ULL) logic cell library.
- The first ULL cell library has been developed with a 0.18um process. Apart from reducing the leakage current, the new cell library also supports special state retention cells that can hold state information while the rest of the system is powered down.
- ARM also works with leading EDA tools vendors to allow chip vendors to make use of these new technologies in their chip designs.

Cortex-M0 Software Portability



PROGRAMMERS MODEL



- The Cortex-M0 processor has two operation modes and two states.
- When the processor is running a program, it is in the Thumb state. In this state, it can be either in the Thread mode or the Handler mode.

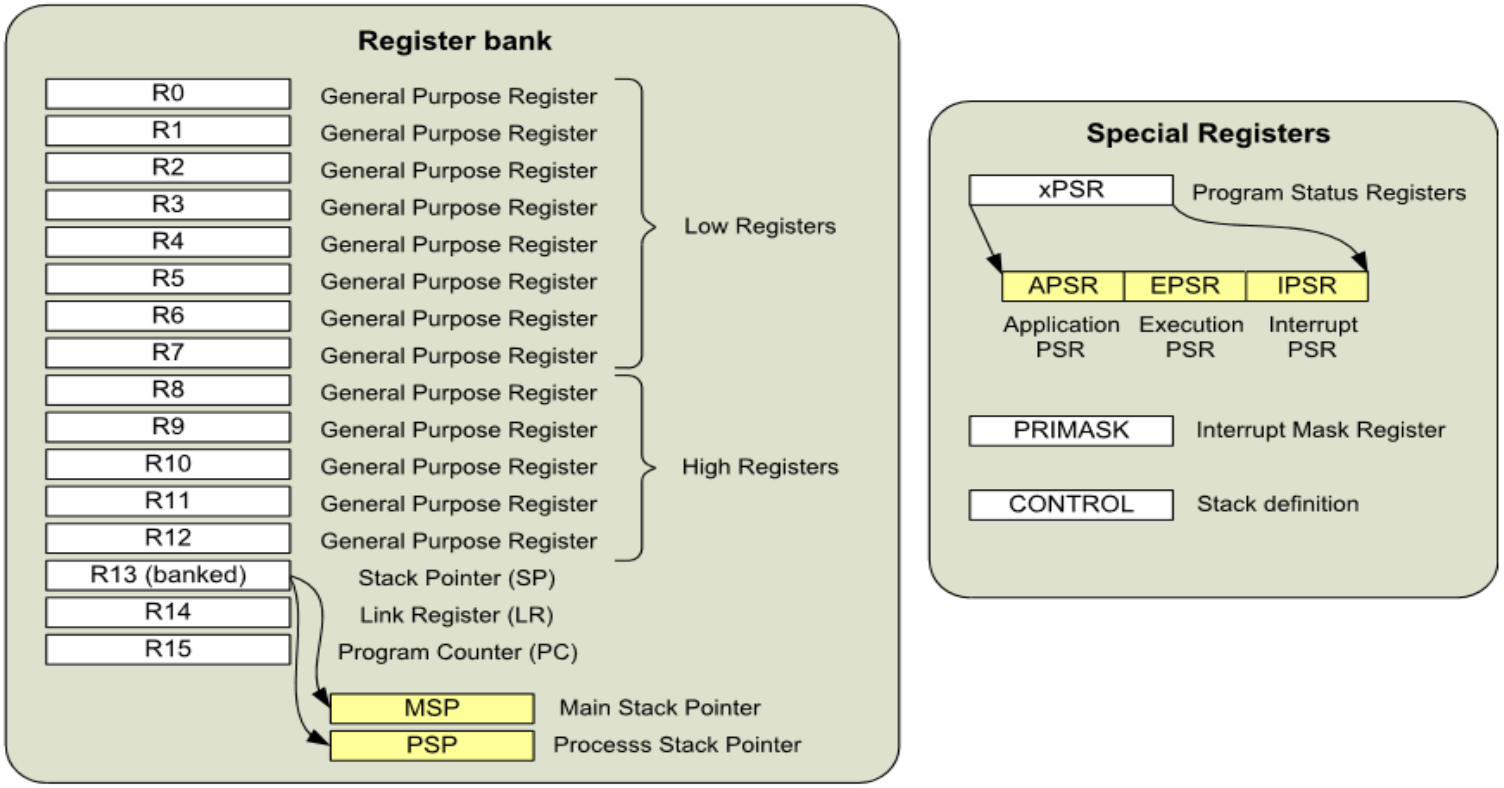
Registers and Special Registers

- To perform data processing and controls, a number of registers are required inside the processor core.
- If data from memory are to be processed, they have to be loaded from the memory to a register in the register bank, processed inside the processor, and then written back to the memory if needed.
- This is commonly called a “load-store architecture.” By having a sufficient number of registers in the register bank, this mechanism is easy to use and is C friendly.
- It is easy for C compilers to compile a C program into machine code with good performance. By using internal registers for short-term data storage, the amount of memory accesses can be reduced.
- The Cortex-M0 processor provides a register bank of 13 general-purpose 32-bit registers and a number of special registers

Continueed..

- R0-R12 Registers R0 to R12 are for general uses.
- Because of the limited space in the 16-bit Thumb instructions, many of the Thumb instructions can only access R0 to R7, which are also called the low registers, whereas some instructions, like MOV (move), can be used on all registers.
- When using these registers with ARM development tools such as the ARM assembler, you can use either uppercase (e.g., R0) or lowercase (e.g., r0) to specify the register to be used.
- The initial values of R0 to R12 at reset are undefined.
- R13, Stack Pointer (SP) R13 is the stack pointer. It is used for accessing the stack memory via PUSH and POP operations.
- There are physically two different stack pointers in Cortex-M0.
- The main stack. pointer (MSP, or SP_main in ARM documentation) is the default stack pointer after reset, and it is used when running exception handlers.
- The process stack pointer (PSP, or SP_process in ARM documentation) can only be used in Thread mode (when not handling exceptions).
- The stack pointer selection is determined by the CONTROL register, one of the special registers that will be introduced later.

Register Banks and Special Registers



Stack Pointer

- R13, Stack Pointer (SP) R13 is the stack pointer. It is used for accessing the stack memory via PUSH and POP operations. There are physically two different stack pointers in Cortex-M0.
- The main stack pointer (MSP, or SP_main in ARM documentation) is the default stack pointer after reset, and it is used when running exception handlers.
- The process stack pointer (PSP, or SP_process in ARM documentation) can only be used in Thread mode (when not handling exceptions).
- The stack pointer selection is determined by the CONTROL register, one of the special registers that will be introduced later. When using ARM development tools, you can access the stack pointer using either “R13” or “SP.” Both uppercase and lowercase (e.g., “r13” or “sp”) can be used.

- Only one of the stack pointers is visible at a given time. However, you can access to the MSP or PSP directly when using the special register access instructions MRS and MSR.
- In such cases, the register names “MSP” or “PSP” should be used.
- The lowest two bits of the stack pointers are always zero, and writes to these two bits are ignored.
- In ARM processors, PUSH and POP are always 32-bit accesses because the registers are 32-bit, and the transfers in stack operations must be aligned to a 32-bit word boundary.
- The initial value of MSP is loaded from the first 32-bit word of the vector table from the program memory during the startup sequence.
- The initial value of PSP is undefined. It is not necessary to use the PSP. In many applications, the system can completely rely on the MSP.
- The PSP is normally used in designs with an OS, where the stack memory for OS Kernel and the thread level application code must be separated.

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.”

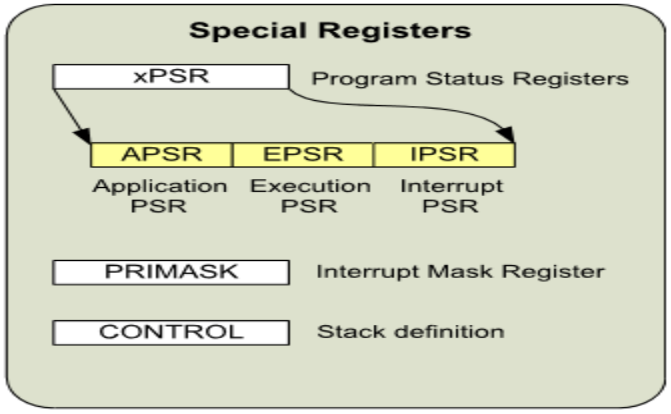
- Both upper and lowercase (e.g., “r14” or “lr”) can be used.
- Although the return address in the Cortex-M0 processor is always an even address (bit[0] is zero because the smallest instructions are 16-bit and must be half-word aligned), bit zero of LR is readable and writeable.
- In the ARMv6-M architecture,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

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).
- 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 2 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

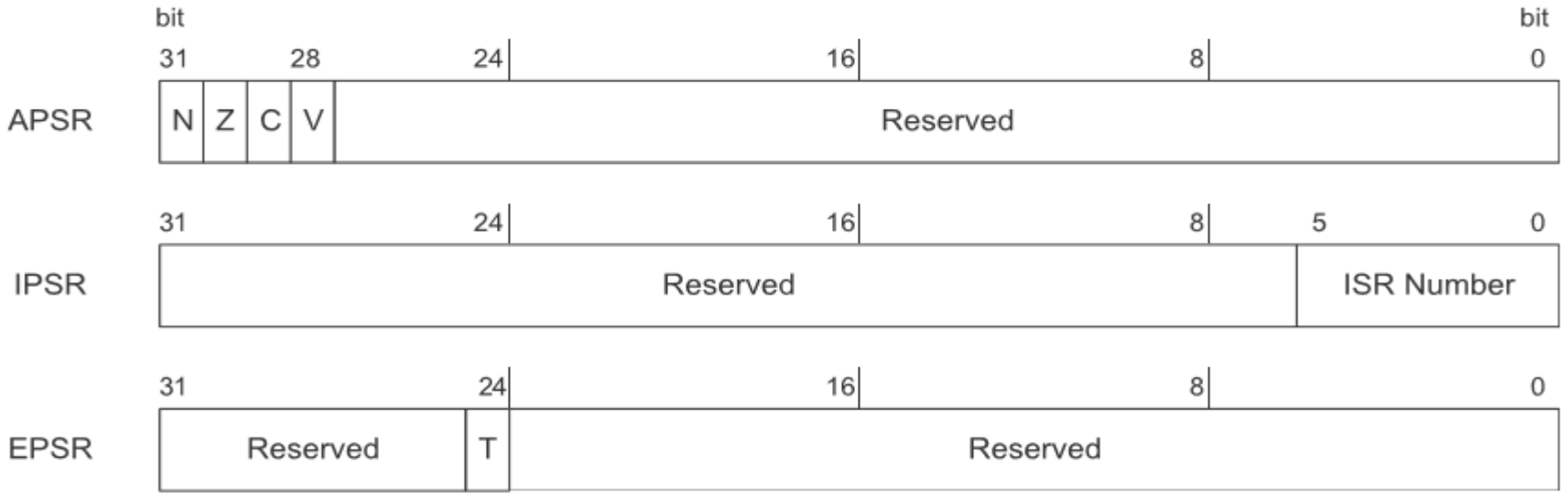


Figure 3.3:
APSR, IPSR, and EPSR.

- 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.
-

- 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.

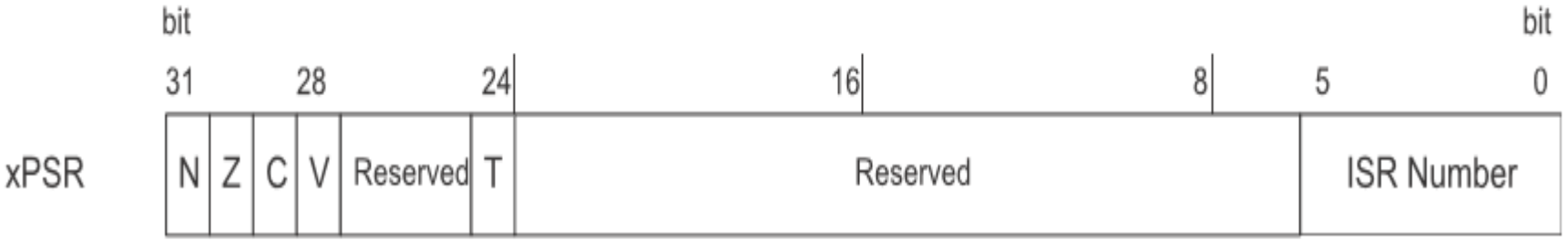


Figure 3.4:
xPSR.

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.

Example

- s

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.

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

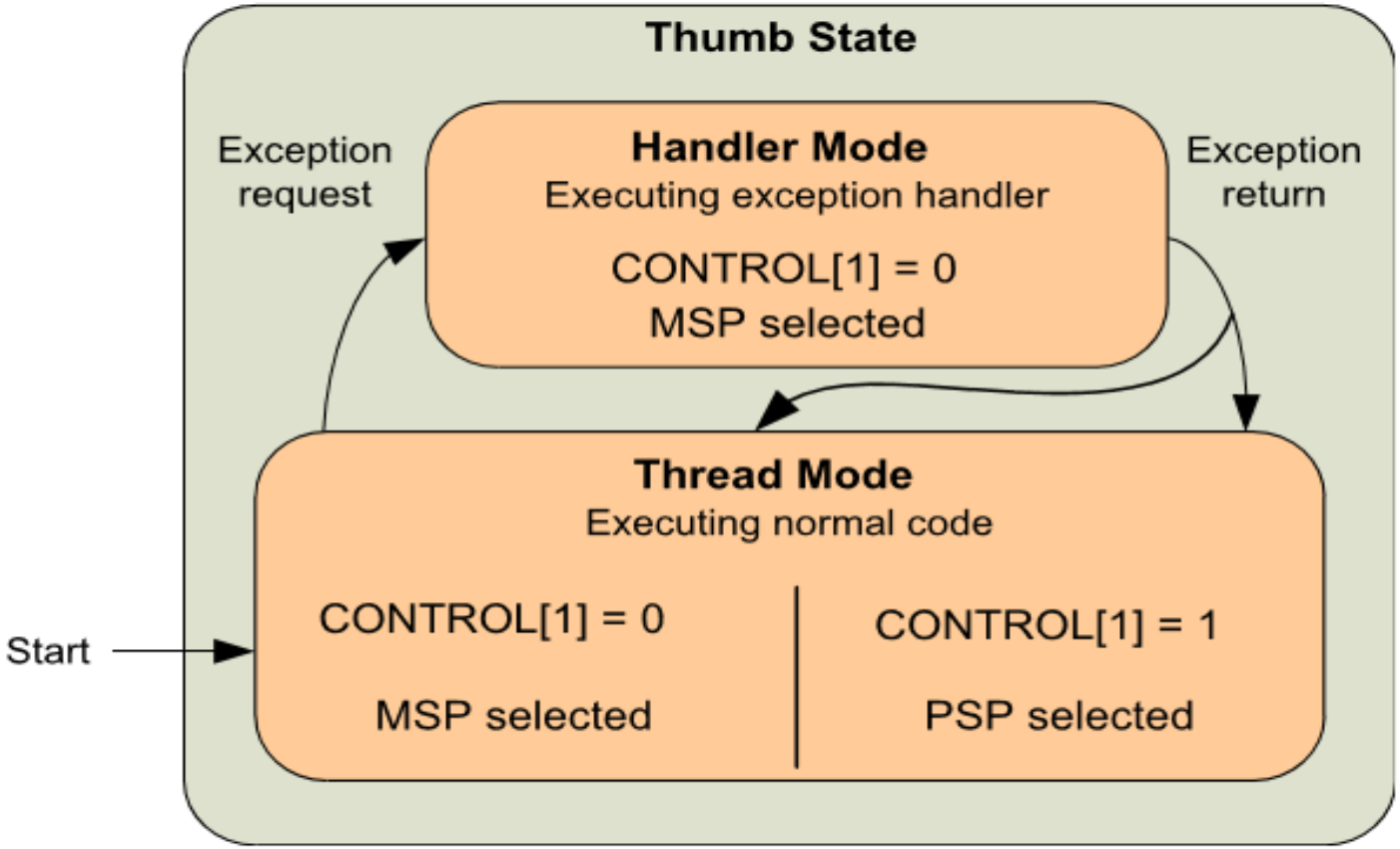


Figure 3.7:
Stack pointer selection.

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.
-

- 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

Memory Overview

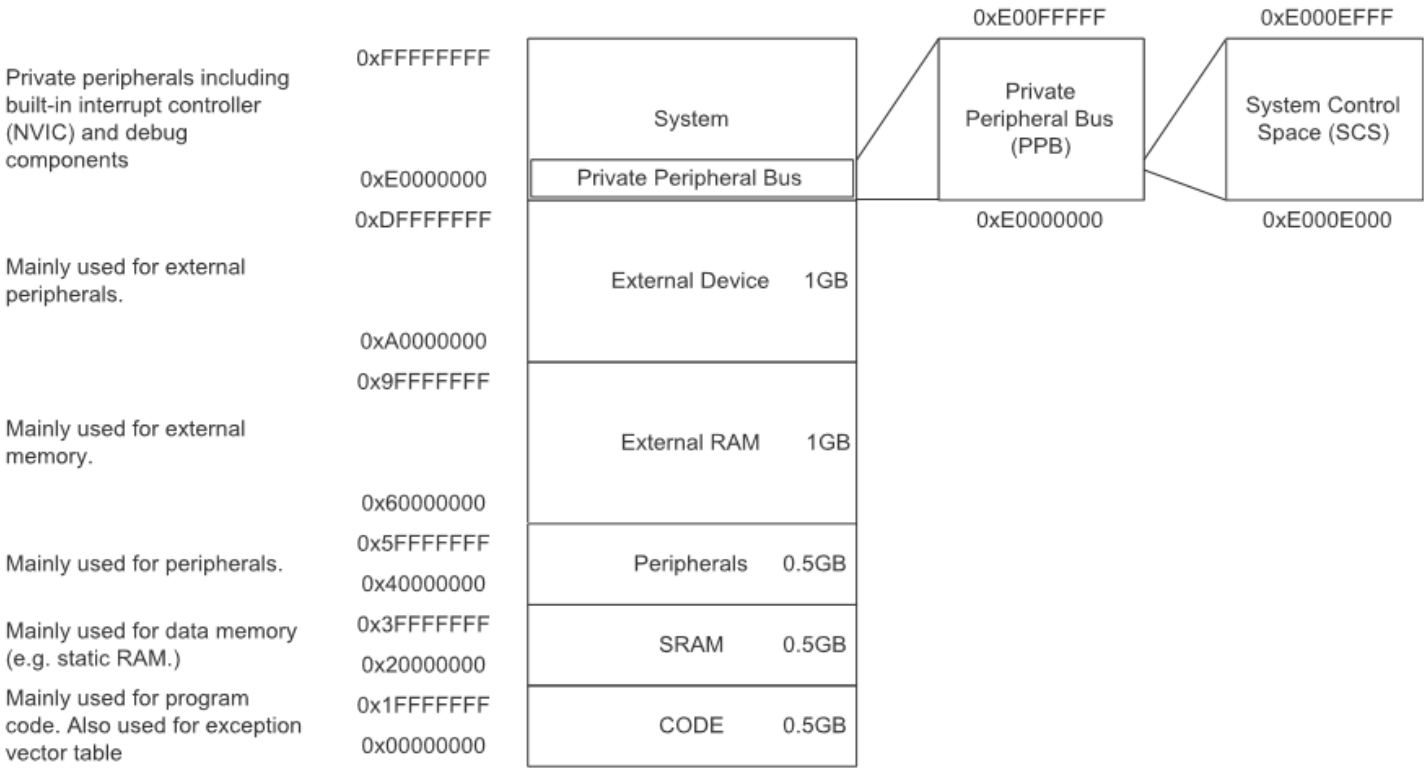


Figure 3.8:
Memory map.

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).
- the

- 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

PUSH POP OPERATION

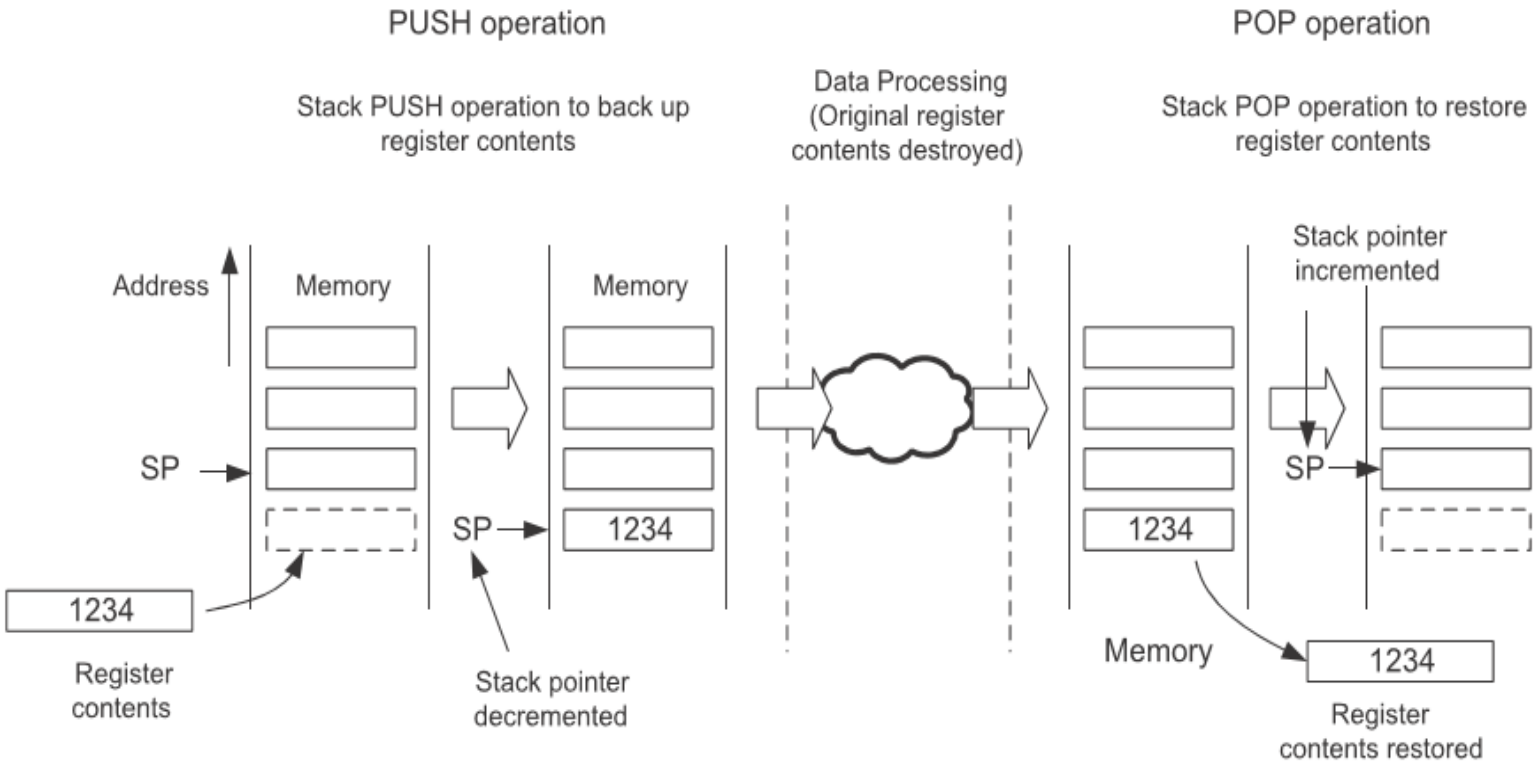


Figure 3.9:
Stack PUSH and POP in the Cortex-M0 processor.

Continueed…

- 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 a signal level (interrupt request froma 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 ¼ 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.

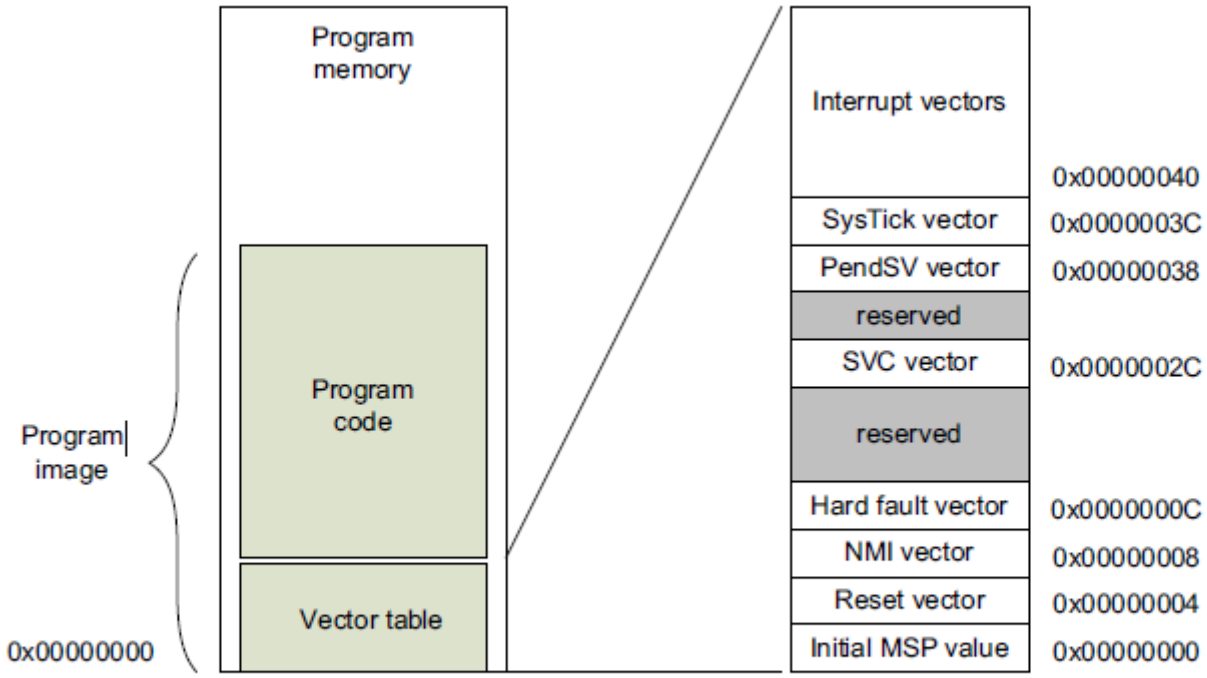


Figure 3.11:
Vector table in a program image.

- 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

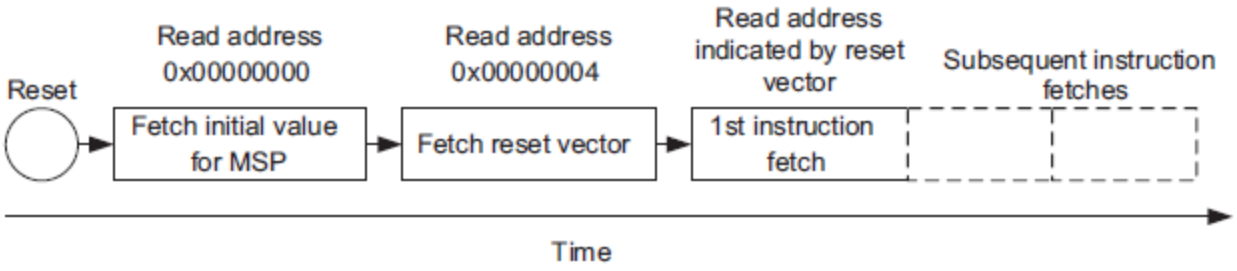
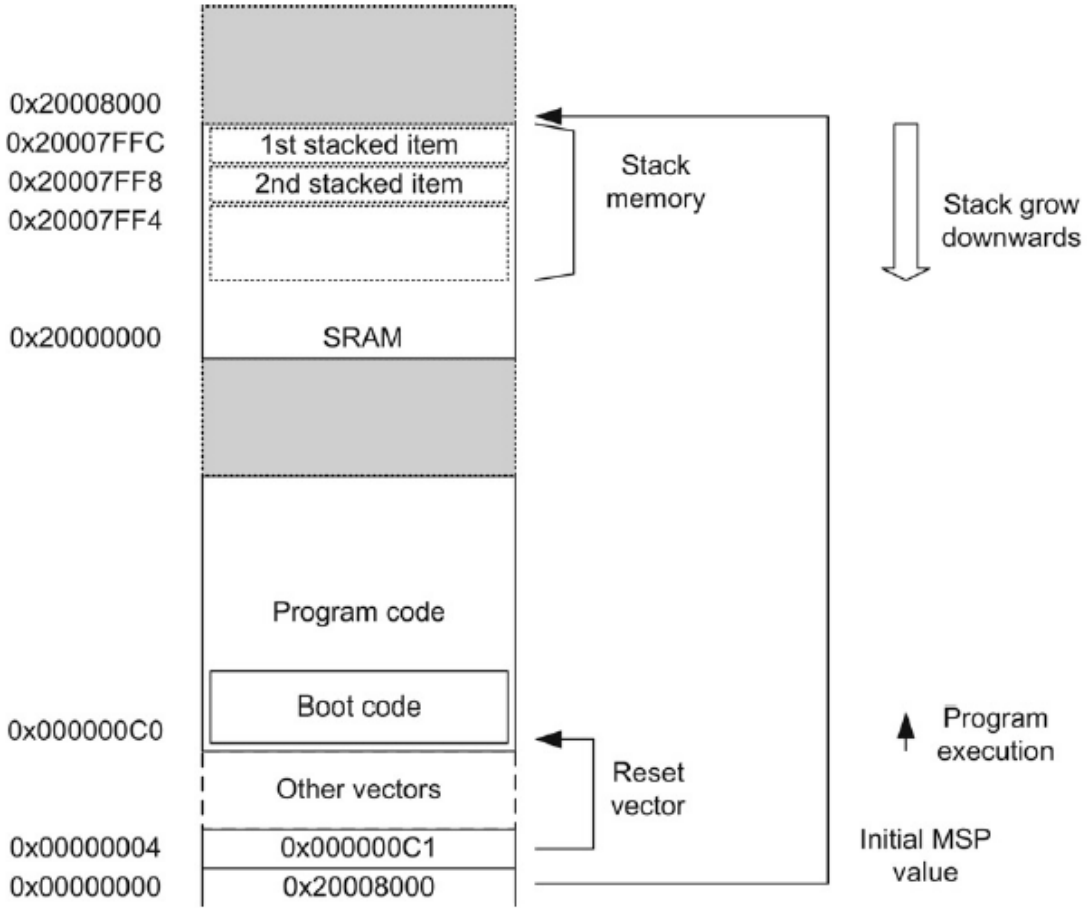


Figure 3.12:
Reset sequence.

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

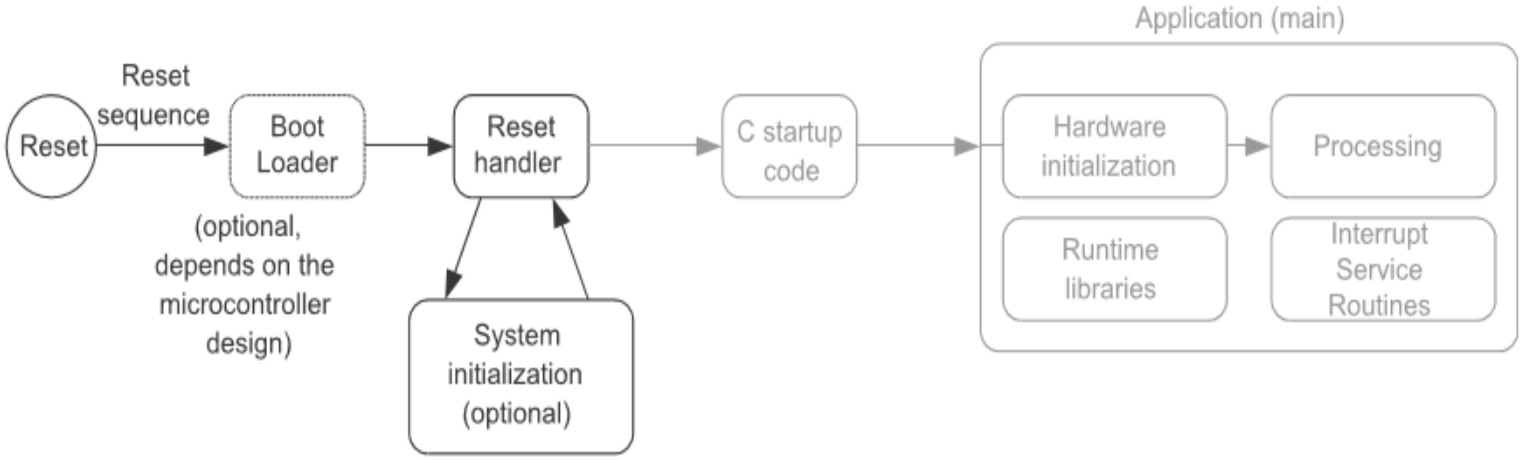

Application (main)
Reset
Reset sequence
Boot Loader
(optional, depends on the microcontroller design)
Reset handler
System initialization (optional)
C startup code
Hardware initialization
Processing
Runtime libraries
Interrupt Service Routines

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).

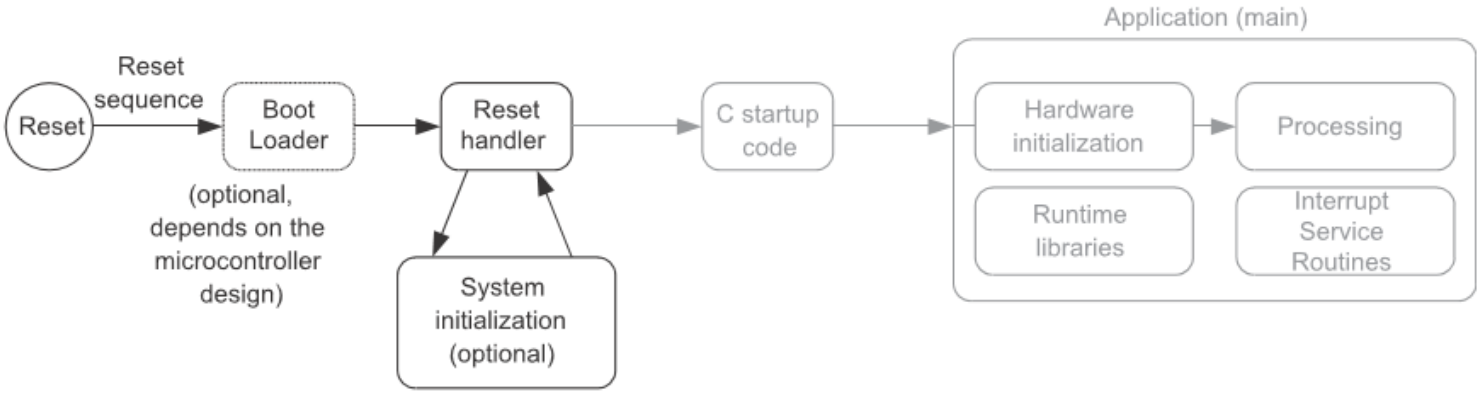

Application (main)
Reset
Reset sequence
Boot Loader
(optional, depends on the microcontroller design)
Reset handler
System initialization (optional)
C startup code
Hardware initialization
Processing
Runtime libraries
Interrupt Service Routines

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

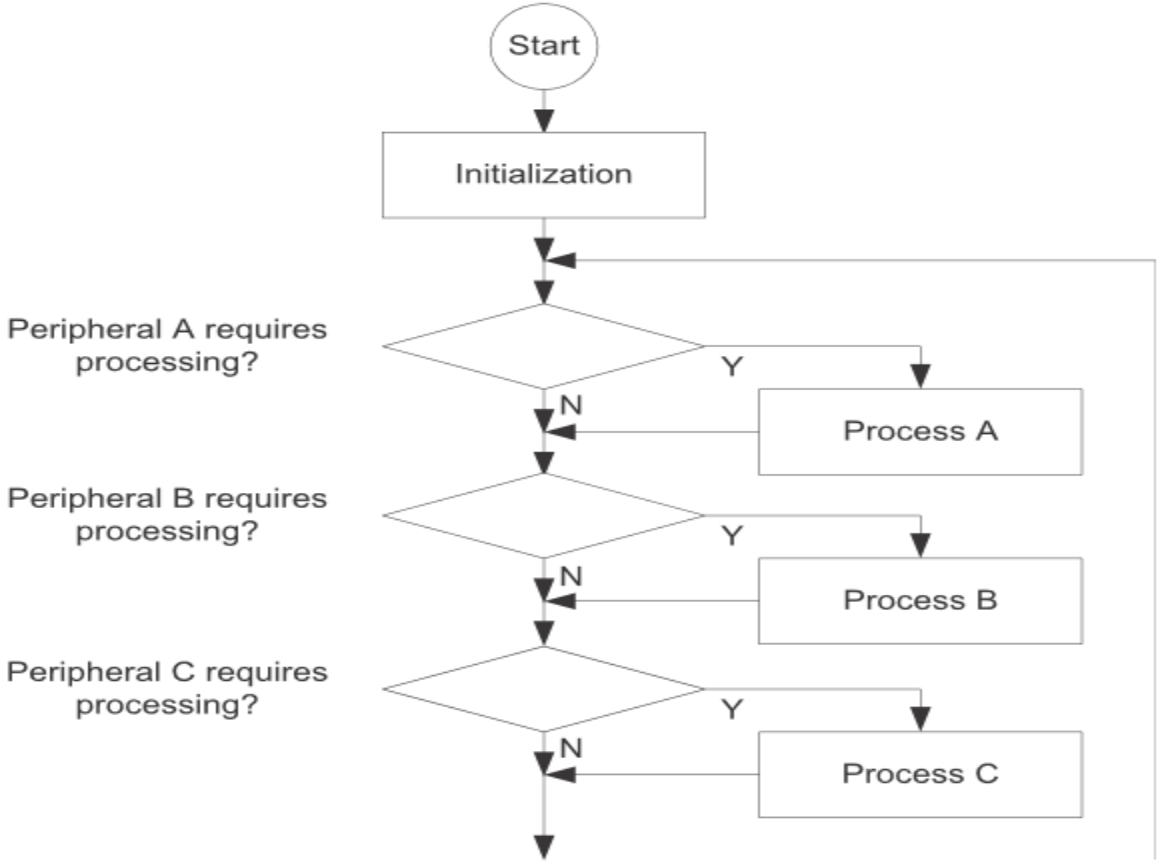
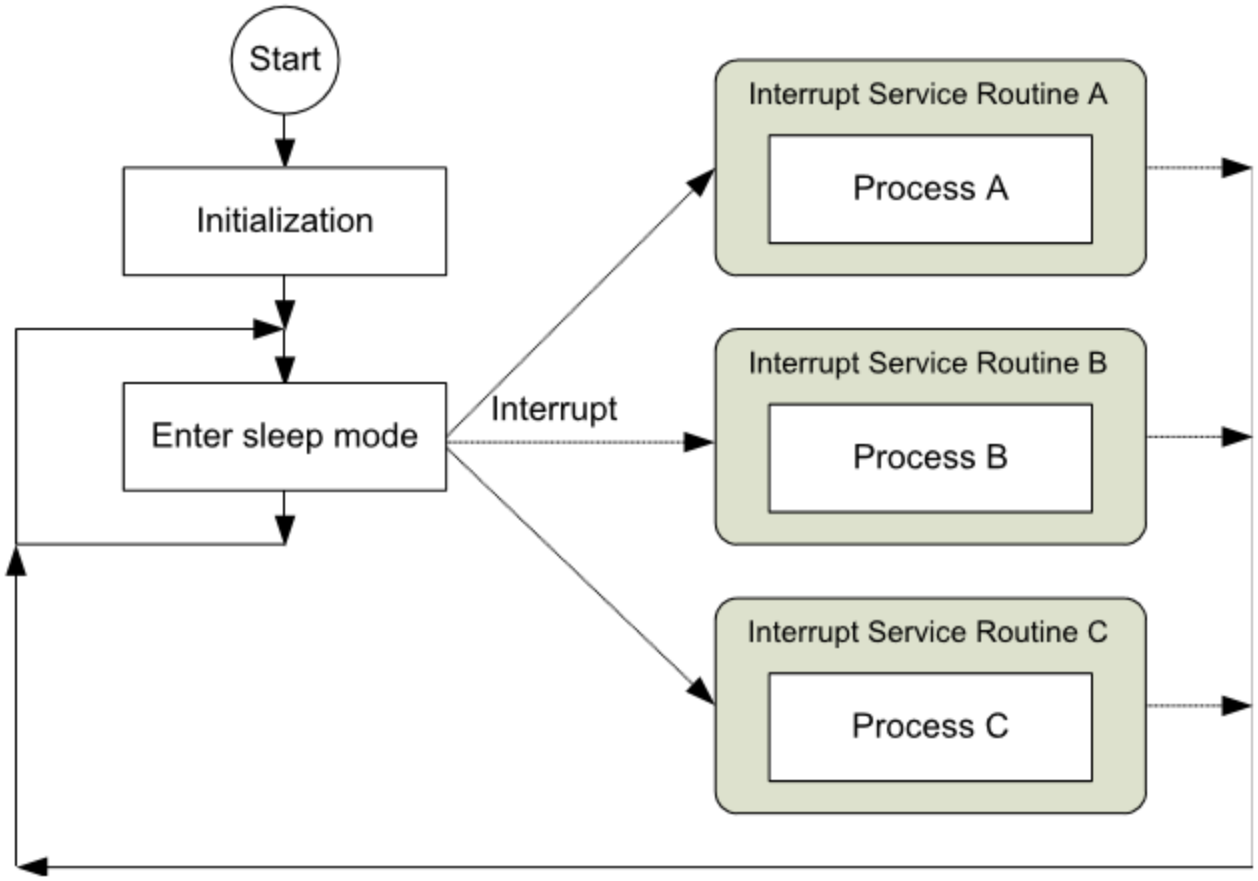


Figure 4.4:

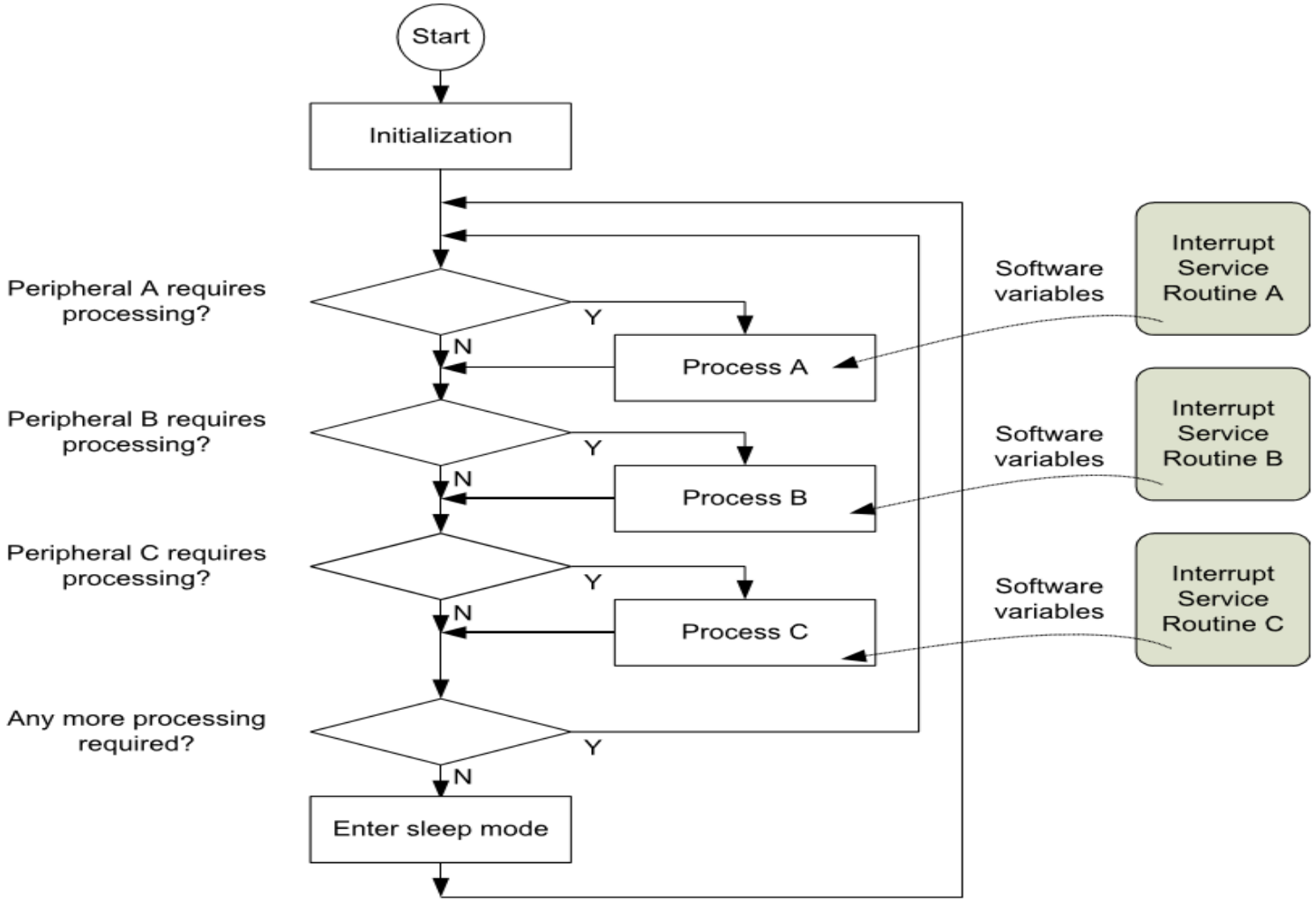
Interrupt Driven



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



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.

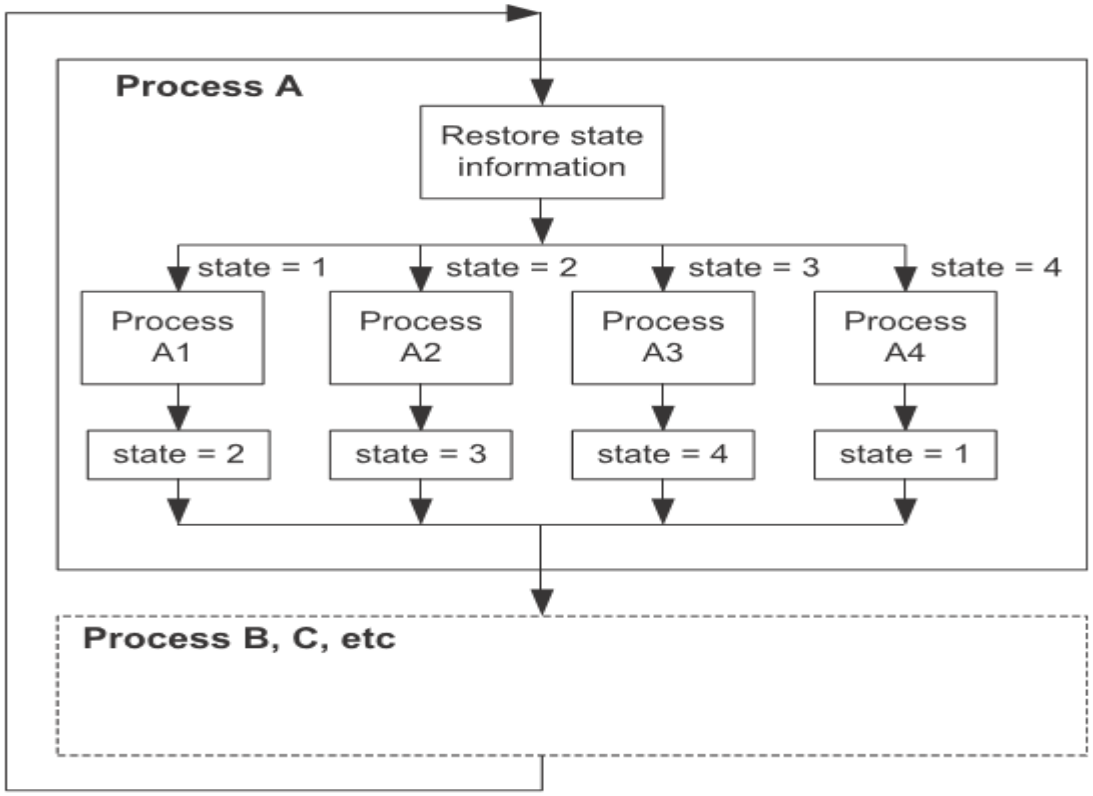


Figure 4.7:

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.

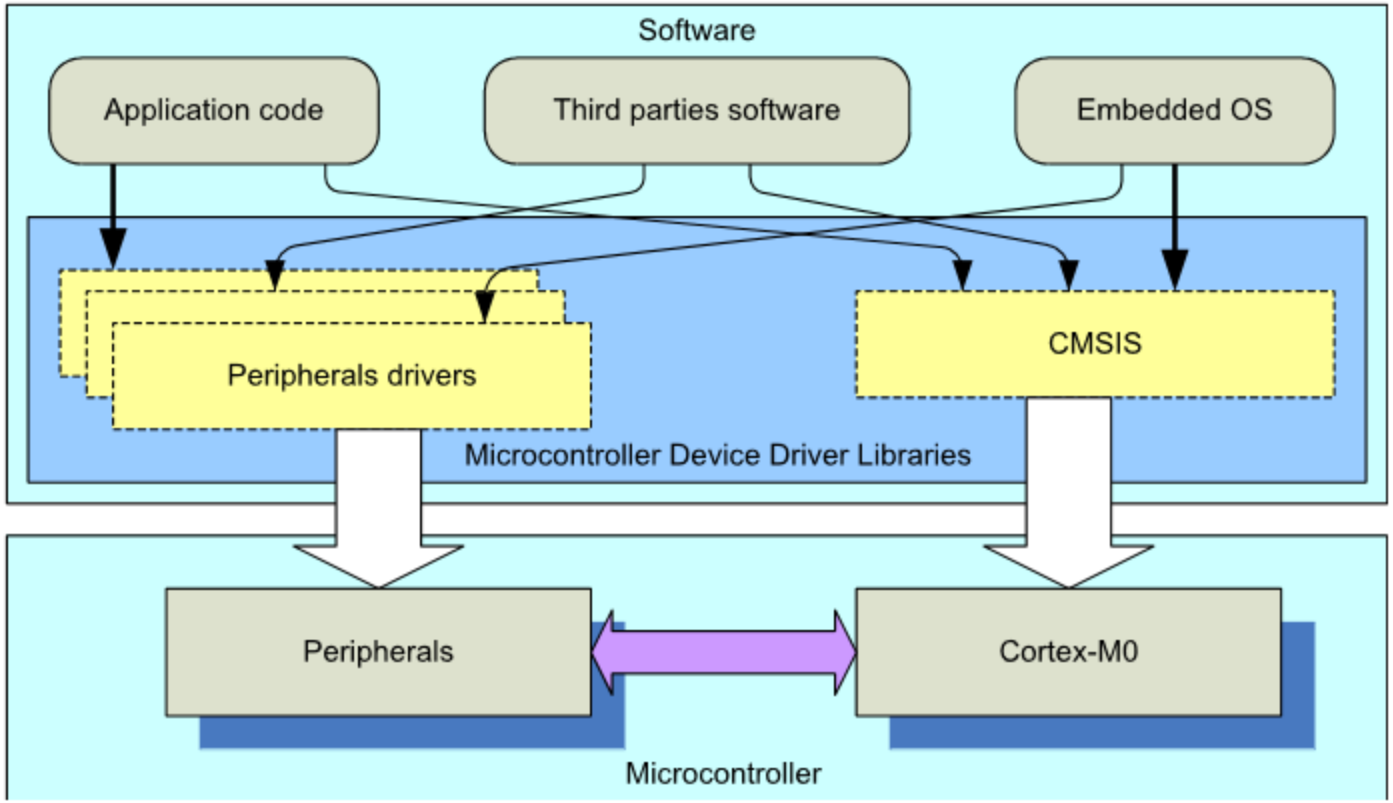

Software
Application code
Third parties software
Embedded OS
Peripherals drivers
CMSIS
Microcontroller Device Driver Libraries
Peripherals
Cortex-M0
Microcontroller

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

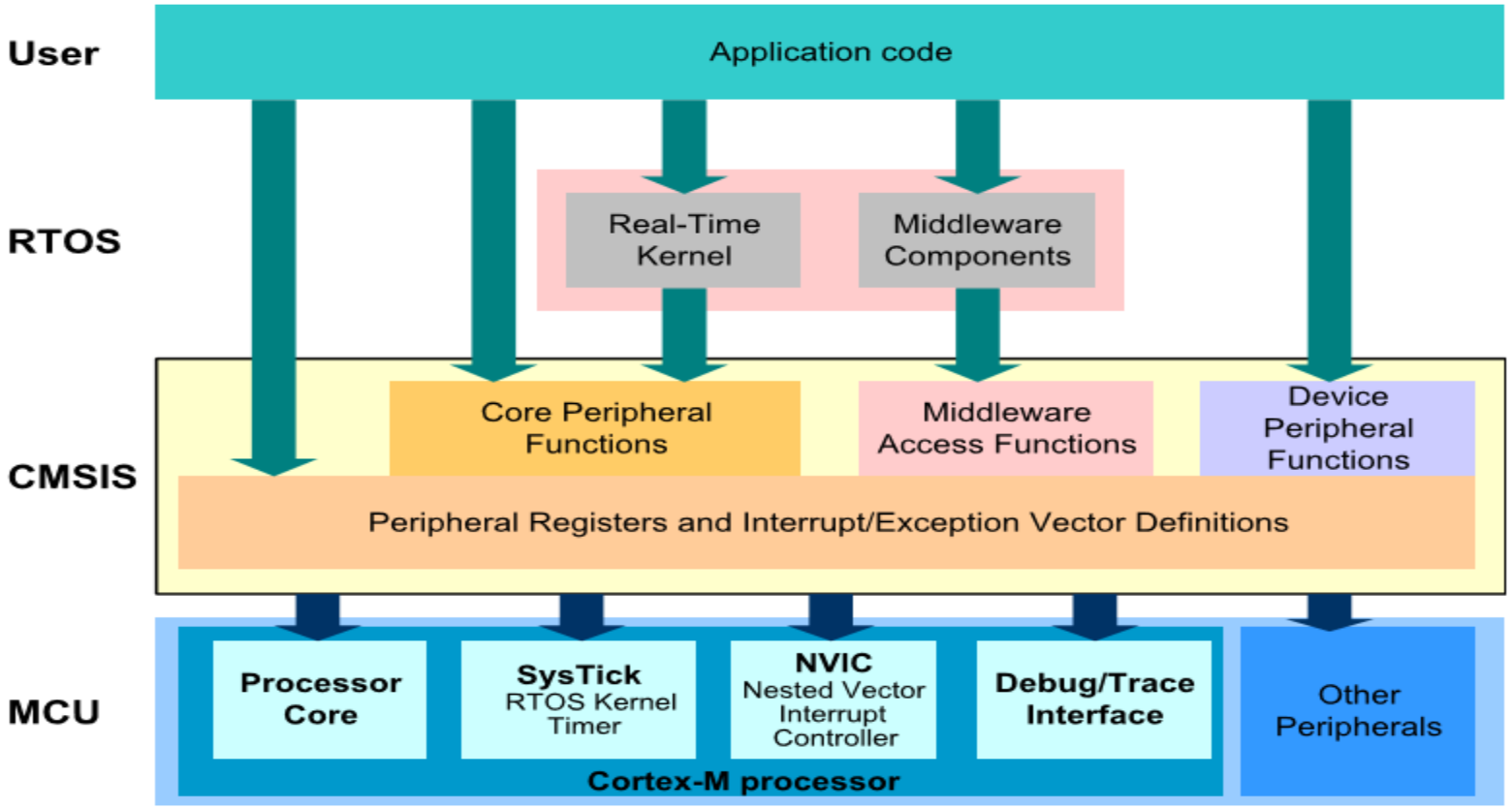
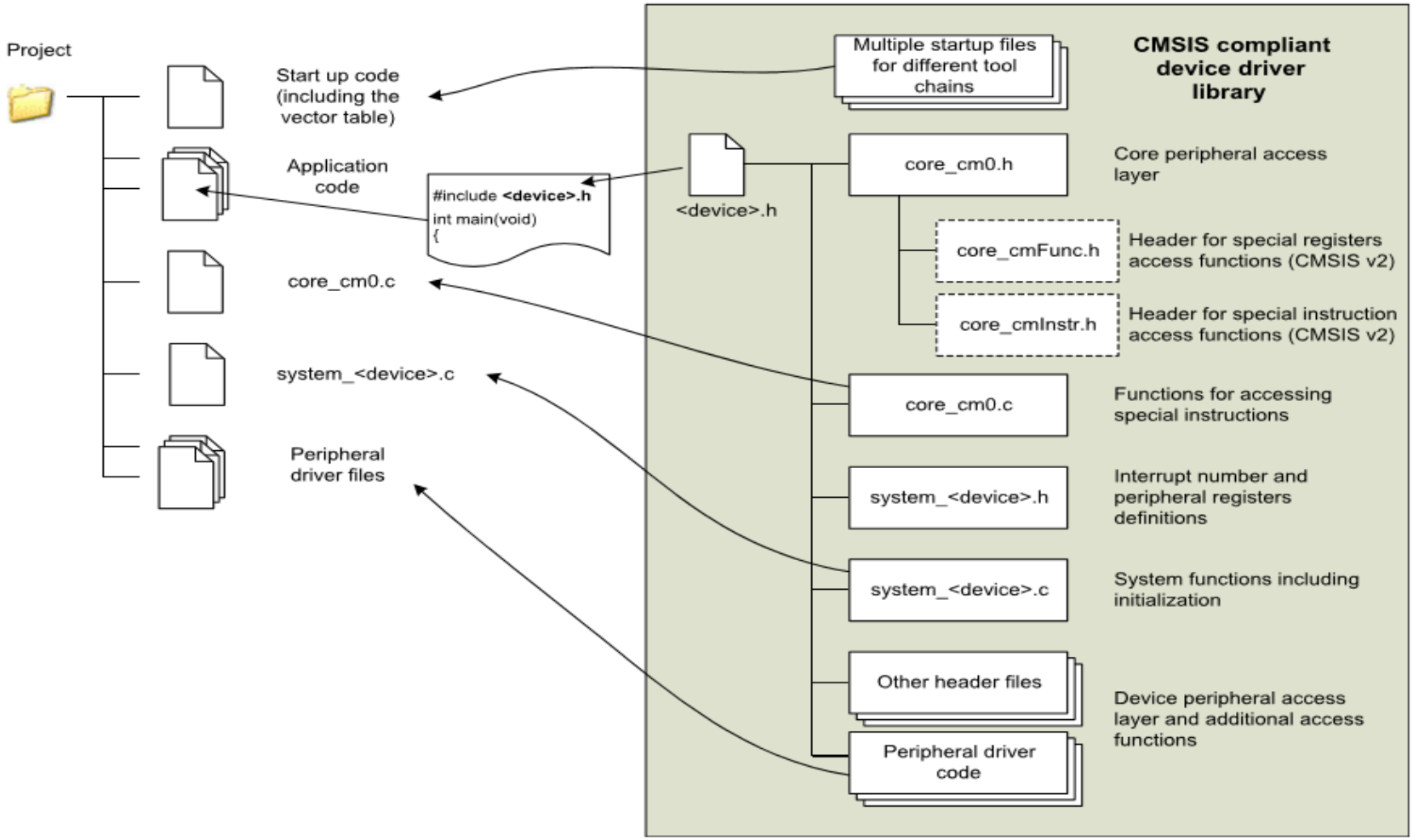


Figure 4.17:
CMSIS structure.

- 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

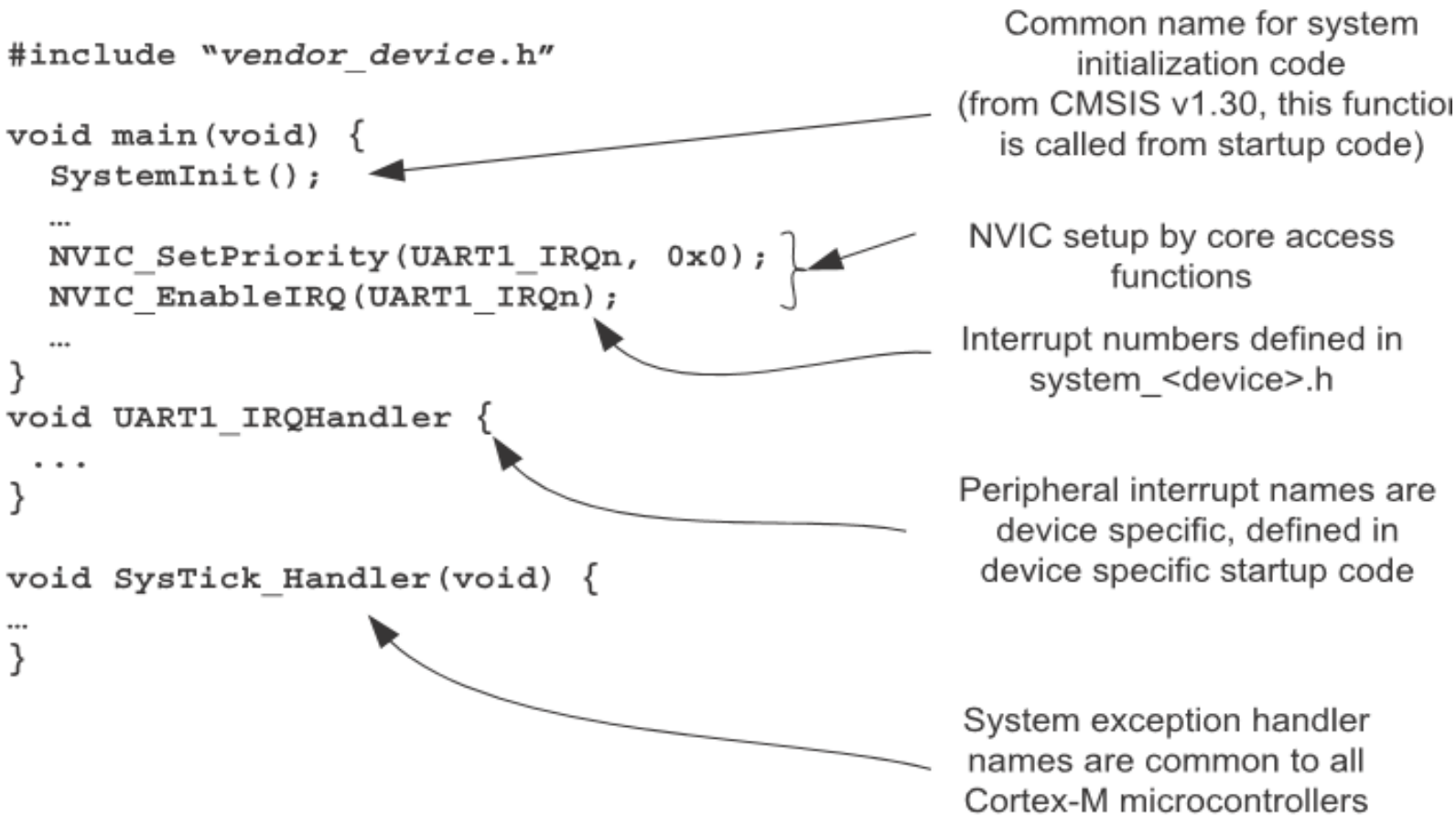

Project
Start up code (including the vector table)
Application code
#include <device>.h
int main(void)
{
core_cm0.c
system_<device>.c
Peripheral driver files
Multiple startup files for different tool chains
CMSIS compliant device driver library
<device>.h
core_cm0.h
Core peripheral access layer
core_cmFunc.h
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

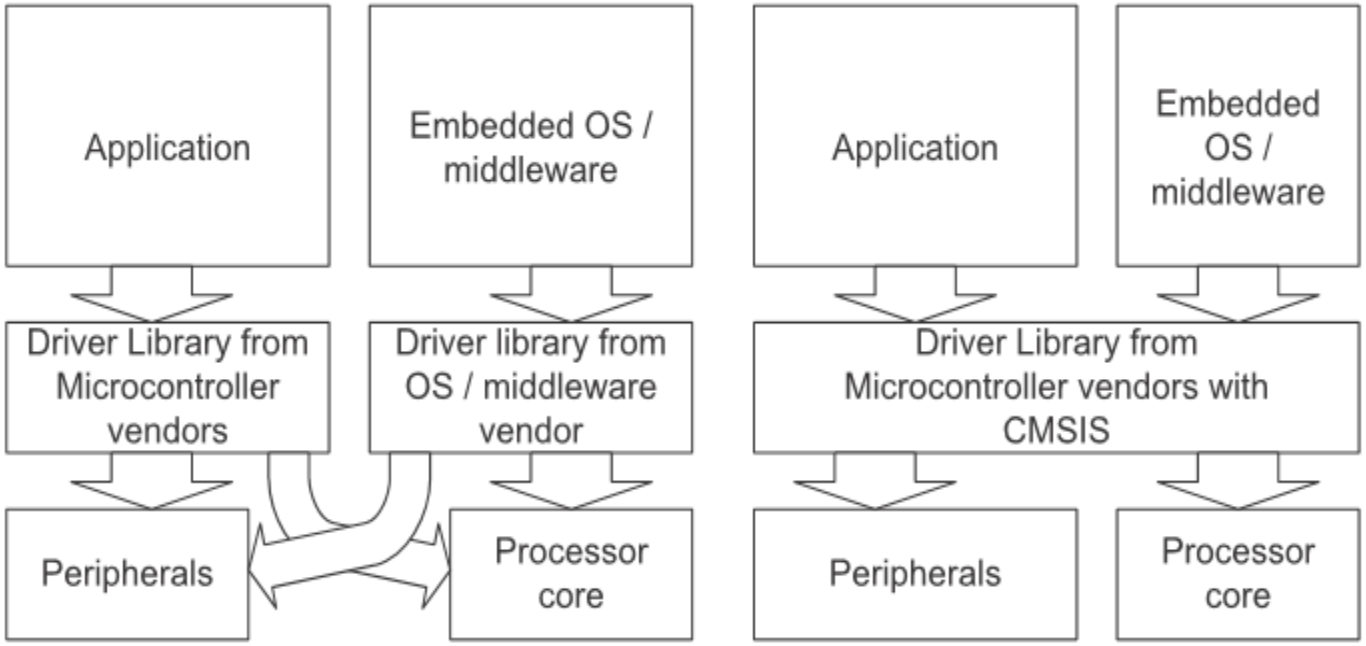
Table 4.4: Files in CMSIS

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



CMSIS avoids overlapping of driver code.



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

Instruction Set

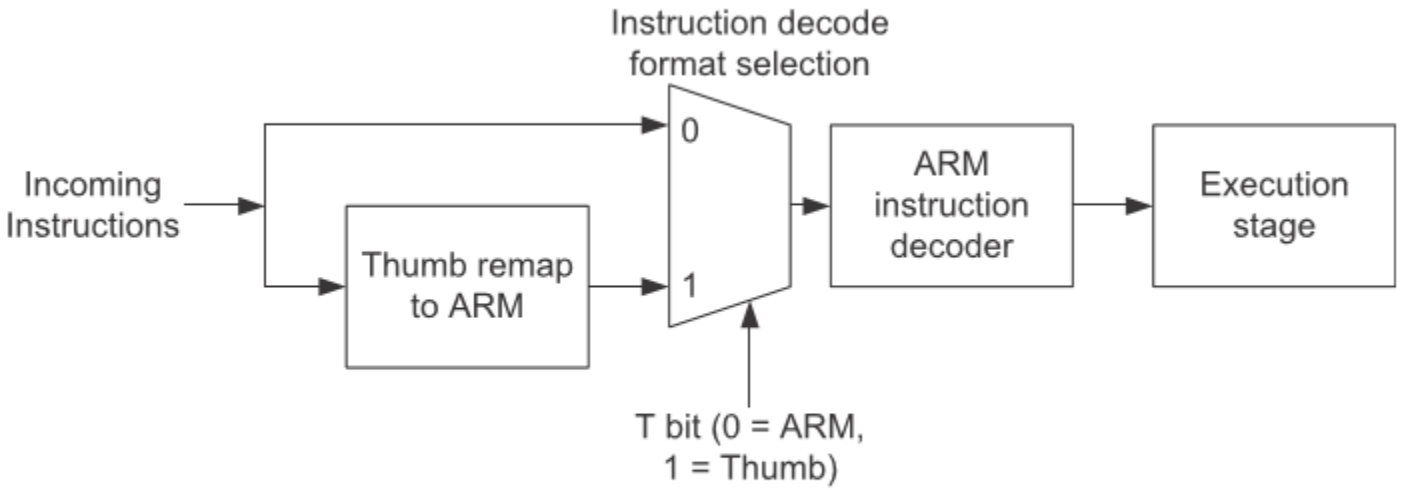


Figure 5.1:
ARM7TDMI design supports both ARM and the Thumb instruction set.

16 Bit Thumb Instructions

Table 5.1: 16-Bit Thumb Instructions Supported on the Cortex-M0 Processor

16-Bit Thumb Instructions Supported on Cortex-M0									
ADC	ADD	ADR	AND	ASR	B	BIC	BLX	BKPT	BX
CMN	CMP	CPS	EOR	LDM	LDR	LDRH	LDRSH	LDRB	LDRSB
LSL	LSR	MOV	MVN	MUL	NOP	ORR	POP	PUSH	REV
REV16	REVSH	ROR	RSB	SBC	SEV	STM	STR	STRH	STRB
SUB	SVC	SXTB	SXTH	TST	UXTB	UXTH	WFE	WFI	YIELD

Cortex –M0 processor supports 32-bit Thumb Instructions.

The Cortex-M0 processor also supports a number of 32-bit Thumb instructions from Thumb-2 technology (Table 5.2):

- MRS and MSR special register access instructions
- ISB, DSB, and DMB memory synchronization instructions
- BL instruction (BL was supported in traditional Thumb instruction set, but the bit field definition was extended in Thumb-2)

Table 5.2: 32-Bit Thumb Instructions Supported on the Cortex-M0 Processor

32-Bit Thumb Instructions Supported on Cortex-M0					
BL	DSB	DMB	ISB	MRS	MSR

```
MOVS     R0, #0x12   ; Set R0 = 0x12 (hexadecimal)
MOVS     R1, #'A'    ; Set R1 = ASCII character A
```

One of the commonly required features in assembly code is constant definitions. By using constant definitions, the program code can be more readable and can make code maintenance easier. In ARM assembly, an example of defining a constant is

```
NVIC_IRQ_SETEN      EQU   0xE000E100
NVIC_IRQ0_ENABLE    EQU   0x1
```

```
...
LDR     R0,=NVIC_IRQ_SETEN    ; Put 0xE000E100 into R0
        ; LDR here is a pseudo instruction that will be converted
        ; to a PC relative literal data load by the assembler
MOVS    R1, #NVIC_IRQ0_ENABLE ; Put immediate data (0x1) into
                              ; register R1
STR     R1,[ R0] ; Store 0x1 to 0xE000E100, this enable external
                 ; interrupt IRQ#0
```

Similarly, the same code can be written with GNU tool chain assembler syntax:

```
.equ    NVIC_IRQ_SETEN,      0xE000E100
.equ    NVIC_IRQ0_ENABLE,    0x1
...
LDR     R0,=NVIC_IRQ_SETEN /* Put 0xE000E100 into R0
         LDR here is a pseudo instruction that will be
         converted to a PC relative load by the assembler */
MOVS    R1, #NVIC_IRQ0_ENABLE /* Put immediate data (0x1) into
                                 register R1 */
STR     R1,[ R0] /* Store 0x1 to 0xE000E100, this enable
                    external interrupt IRQ#0 */
```