EMBEDDED SYSTEMS (18EC62)

MODULE – 1

ARM – 32-Bit Microcontroller

What is ARM?

- ARM is a 32-bit reduced instruction set computer (RISC) instruction set architecture (ISA) developed by ARM Holdings.
- It was named the Advanced RISC Machine, and before that, the Acorn RISC Machine.

What is the ARM Cortex-M3 Processor?

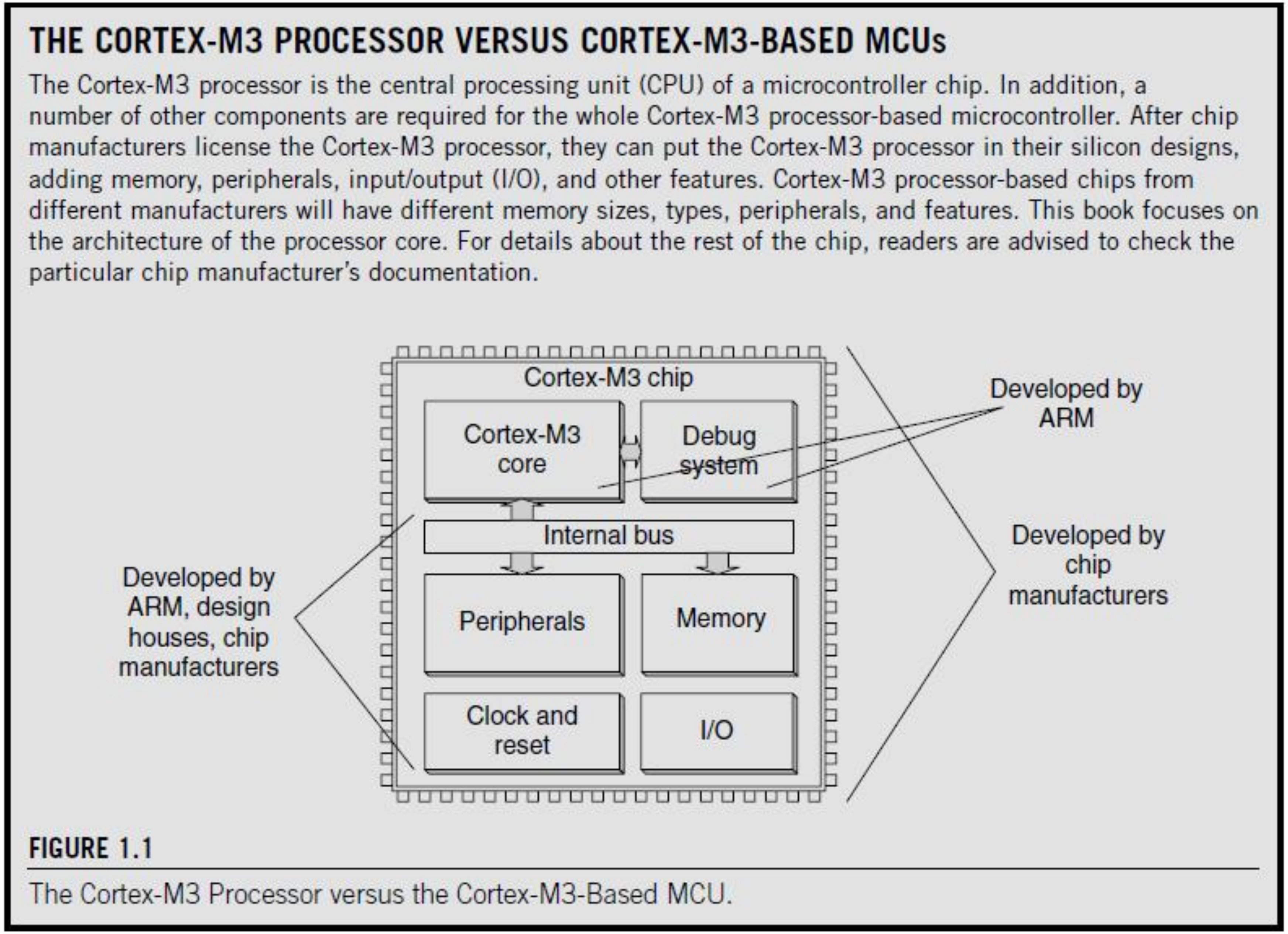
- The ARM Cortex-M3 processor, the first of the Cortex generation of processors released by ARM in 2006, was primarily designed to target the 32-bit microcontroller market.
- The Cortex-M3 processor provides excellent performance at low gate count and comes with many new features previously available only in high-end processors.

Background of ARM

- ARM was formed in 1990 as Advanced RISC Machines Ltd., a joint venture of Apple Computer, Acorn Computer Group, and VLSI Technology.
- In 1991, ARM introduced the ARM6 processor family.
    - VLSI became the initial licensee.
    - Subsequently, additional companies, including Texas Instruments, NEC, Sharp, and ST Microelectronics, licensed the ARM processor designs, extending the applications of ARM processors into mobile phones, computer hard disks, personal digital assistants (PDAs), home entertainment systems, and many other consumer products.

Background of ARM (continued)

- ARM does not manufacture processors or sell the chips directly.
- Instead, ARM licenses the processor designs to business partners, including a majority of the world’s leading semiconductor companies.
- Based on the ARM low-cost and power-efficient processor designs, these partners create their processors, microcontrollers, and system-on-chip solutions.
- This business model is commonly called intellectual property (IP) licensing.
- In addition to processor designs, ARM also licenses systems-level IP and various software IPs.
  - To support these products, ARM has developed a strong base of development tools, hardware, and software products to enable partners to develop their own products.



THE CORTEX-M3 PROCESSOR VERSUS CORTEX-M3-BASED MCUs

The Cortex-M3 processor is the central processing unit (CPU) of a microcontroller chip. In addition, a number of other components are required for the whole Cortex-M3 processor-based microcontroller. After chip manufacturers license the Cortex-M3 processor, they can put the Cortex-M3 processor in their silicon designs, adding memory, peripherals, input/output (I/O), and other features. Cortex-M3 processor-based chips from different manufacturers will have different memory sizes, types, peripherals, and features. This book focuses on the architecture of the processor core. For details about the rest of the chip, readers are advised to check the particular chip manufacturer's documentation.

FIGURE 1.1

The Cortex-M3 Processor versus the Cortex-M3-Based MCU.

Architecture Versions

- Classic Processors
- ARM Cortex-A Processors
- ARM Cortex-R Processors
- ARM Cortex-M Processors

ARM7

- The Arm7TDMI-S is an excellent workhorse processor capable of a wide array of applications.
- Traditionally used in mobile handsets, the processor is now broadly in many non-mobile applications.

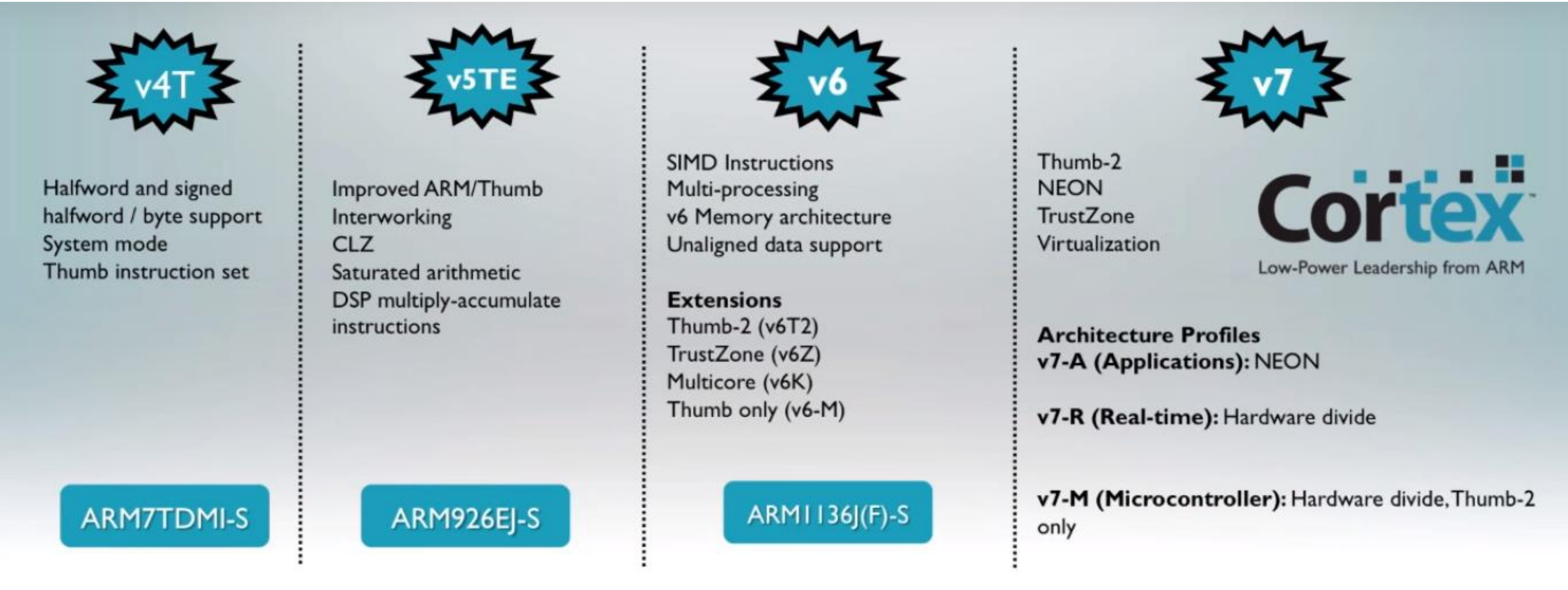
ARM9

- The Arm9 family includes three processors:
- Arm968E-S is the smallest and lowest-power Arm9 processor, built with interfaces for Tightly Coupled Memory and aimed at real-time applications.
- Arm946E-S is a real-time orientated processor with optional cache interfaces, a full Memory Protection Unit, and Tightly Coupled Memory.
- Arm926EJ-S is the entry point processor capable of supporting full Operating Systems including Linux, WindowsCE, and Symbian.

ARM11

- The Arm11 family includes four processors:
- Arm11MPCore introduced multicore technology and is still used in a wide range of applications.
- Arm1176JZ(F)-S is the highest-performance single-core processor in the Classic Arm family. It also introduced TrustZone technology to enable secure execution outside of the reach of malicious code.
- Arm1156T2(F)-S is the highest-performance processor in the real-time Classic Arm family.
- Arm1136J(F)-S is very similar to Arm926EJ-S, but includes an extended pipeline, basic SIMD (Single Instruction Multiple Data) instructions, and improved frequency and performance.

Development of the ARM Architecture



v4T

Halfword and signed halfword / byte support
System mode
Thumb instruction set

ARM7TDMI-S

v5TE

Improved ARM/Thumb Interworking
CLZ
Saturated arithmetic
DSP multiply-accumulate instructions

ARM926EJ-S

v6

SIMD Instructions
Multi-processing
v6 Memory architecture
Unaligned data support

Extensions
Thumb-2 (v6T2)
TrustZone (v6Z)
Multicore (v6K)
Thumb only (v6-M)

ARM1136J(F)-S

v7

Thumb-2
NEON
TrustZone
Virtualization

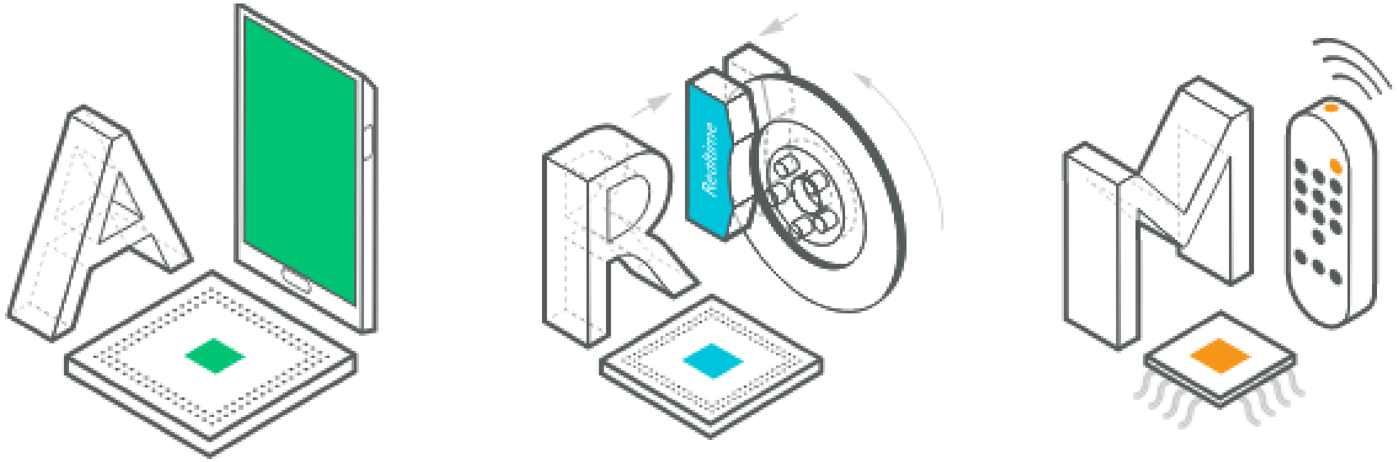
Cortex
Low-Power Leadership from ARM

Architecture Profiles
v7-A (Applications): NEON

v7-R (Real-time): Hardware divide

v7-M (Microcontroller): Hardware divide, Thumb-2 only

ARM Cortex Family



ARM Cortex Processors

- Cortex-A Series
  - Designed for high-performance open application platforms
- Cortex-R Series
  - Designed for high-end embedded systems in which real-time performance is needed
- Cortex-M Series
  - Designed for deeply embedded microcontroller-type systems

A Profile (ARMv7-A)

- Application processors which are designed to handle complex applications such as high-end embedded operating systems (OSs) (e.g., Symbian, Linux, and Windows Embedded).
- These processors requiring the highest processing power, virtual memory system support with memory management units (MMUs), and, optionally, enhanced Java support and a secure program execution environment.
- Example products include high-end mobile phones and electronic wallets for financial transactions.

R Profile (ARMv7-R)

- Real-time, high-performance processors targeted primarily at the higher end of the real-time market
- Suitable for those applications, such as high-end breaking systems and hard drive controllers, in which high processing power and high reliability are essential and for which low latency is important.

M Profile (ARMv7-M)

- Processors targeting low-cost applications in which processing efficiency is important and cost, power consumption, low interrupt latency, and ease of use are critical, as well as industrial control applications, including real-time control systems.
- The Cortex processor families are the first products developed on architecture v7, and the Cortex-M3 processor is based on one profile of the v7 architecture, called ARM v7-M, an architecture specification for microcontroller products.

Evolution of ARM Processor Architecture

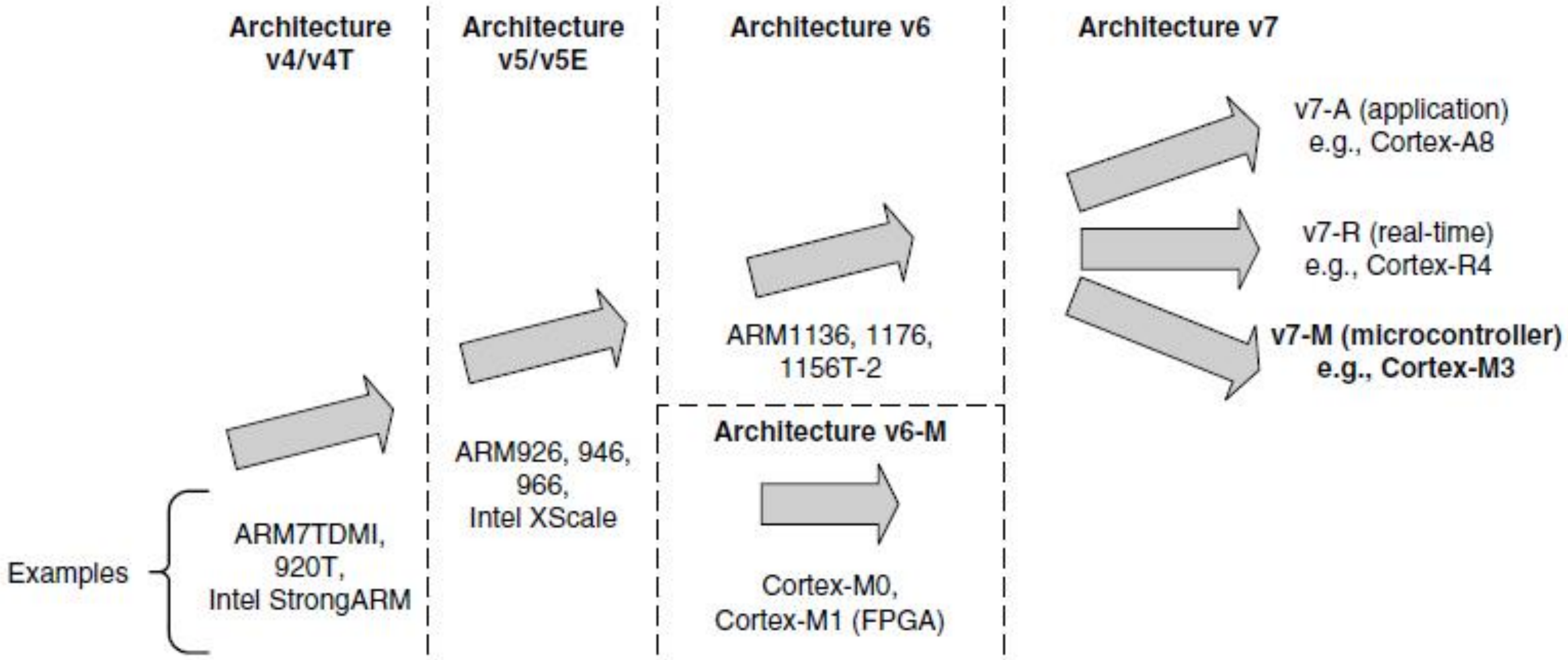
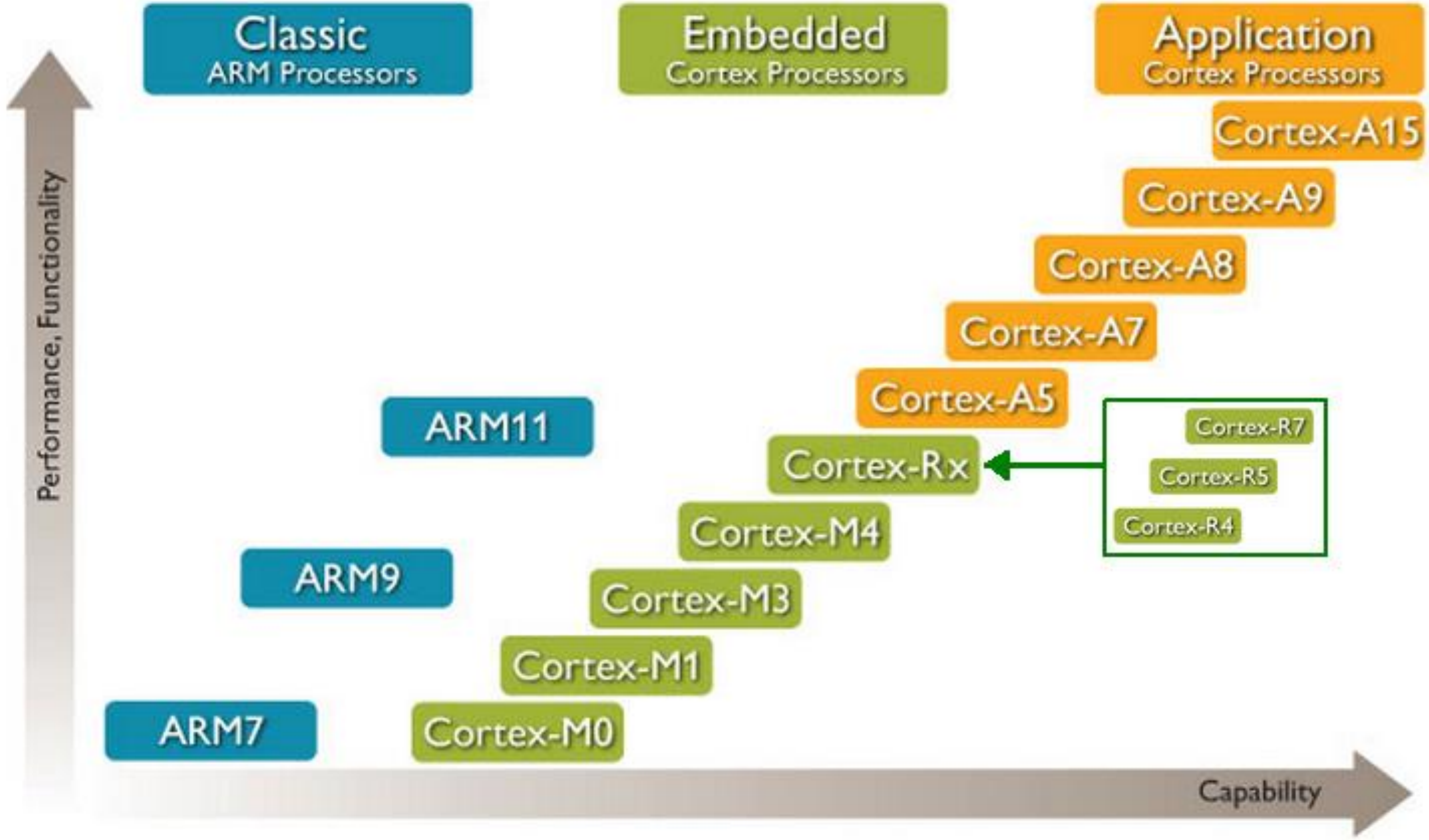


FIGURE 1.2

The Evolution of ARM Processor Architecture.

Evolution of ARM Processor Architecture (continued)



Evolution of ARM Processor Architecture (continued)

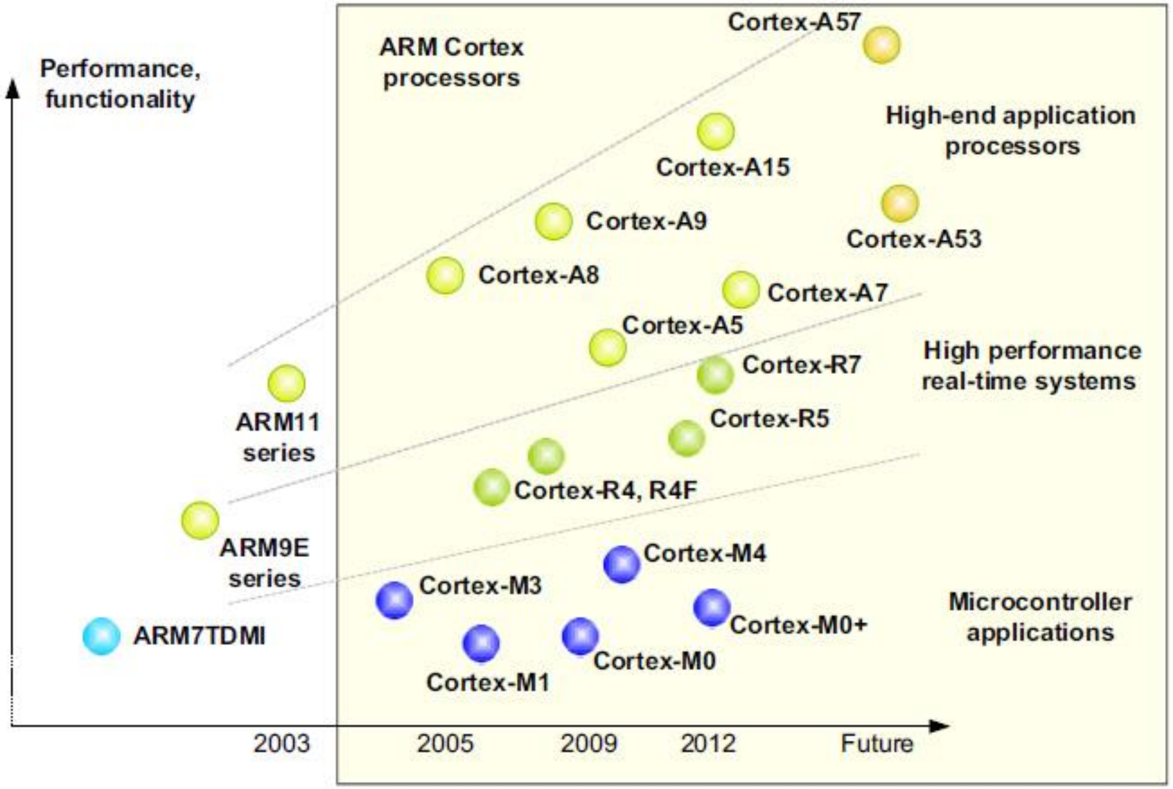


FIGURE 1.5

Diversity of processor products for three areas in the Cortex processor family

Instruction Set Development

- Historically (since ARM7TDMI), two different instruction sets are supported on the ARM processor:
  - The ARM instructions that are 32 bits and Thumb instructions that are 16 bits.
- During program execution, the processor can be dynamically switched between the ARM state and the Thumb state to use either one of the instruction sets.
- The Thumb instruction set provides only a subset of the ARM instructions, but it can provide higher code density.
  - It is useful for products with tight memory requirements.

Instruction Set Development (continued)

- As the architecture version has been updated, extra instructions have been added to both ARM instructions and Thumb instructions.
- In 2003, ARM announced the Thumb-2 instruction set, which is a new superset of Thumb instructions that contains both 16-bit and 32-bit instructions.

The Thumb-2 Technology and Instruction Set Architecture

- The Thumb-2 technology extended the Thumb Instruction Set Architecture (ISA) into a highly efficient and powerful instruction set that delivers significant benefits in terms of ease of use, code size, and performance.
- The extended instruction set in Thumb-2 is a superset of the previous 16-bit Thumb instruction set, with additional 16-bit instructions alongside 32-bit instructions.
- It allows more complex operations to be carried out in the Thumb state, thus allowing higher efficiency by reducing the number of states switching between ARM state and Thumb state.

The Thumb-2 Technology and Instruction Set Architecture (continued)

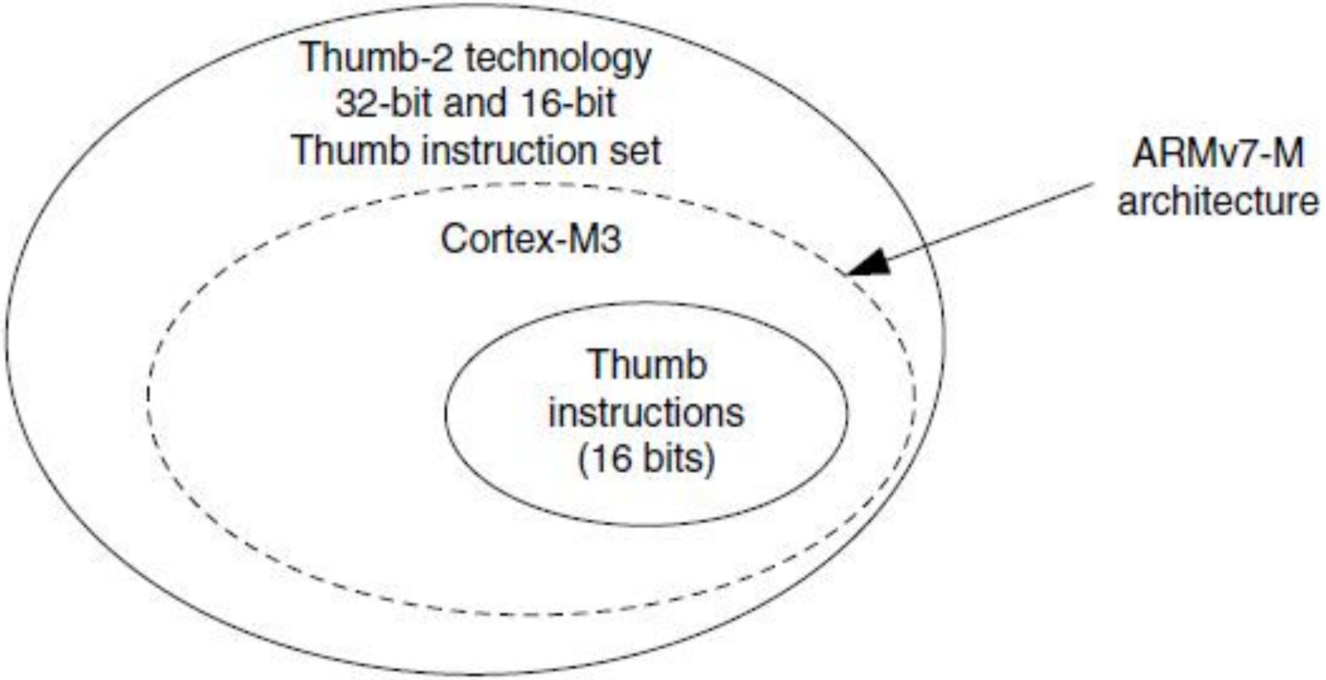


FIGURE 1.4

The Relationship between the Thumb Instruction Set in Thumb-2 Technology and the Traditional Thumb.

The Thumb-2 Technology and Instruction Set Architecture (continued)

- The Cortex-M3 supports only the Thumb-2 (and traditional Thumb) instruction set.
- Instead of using ARM instructions for some operations, as in traditional ARM processors, it uses the Thumb-2 instruction set for all operations.
- As a result, the Cortex-M3 processor is not backward compatible with traditional ARM processors.

The Thumb-2 Technology and Instruction Set Architecture (continued)

- With support for both 16-bit and 32-bit instructions in the Thumb-2 instruction set, there is no need to switch the processor between Thumb state (16-bit instructions) and ARM state (32-bit instructions).
- In the Cortex-M3 processor, 32-bit instructions can be mixed with 16-bit instructions without switching state, getting high code density and high performance with no extra complexity.
- The Thumb-2 instruction set is a very important feature of the ARMv7 architecture.

Cortex-M3 Processor Applications

- Low-cost microcontrollers:
  - The Cortex-M3 processor is ideally suited for low-cost microcontrollers, which are commonly used in consumer products, from toys to electrical appliances.
  - It is a highly competitive market due to the many well-known 8-bit and 16-bit microcontroller products on the market.
  - Its lower power, high performance, and ease-of-use advantages enable embedded developers to migrate to 32-bit systems and develop products with the ARM architecture.
- Automotive:
  - The Cortex-M3 processor has very high-performance efficiency and low interrupt latency, allowing it to be used in real-time systems.
  - The Cortex-M3 processor supports up to 240 external vectored interrupts, with a built-in interrupt controller with nested interrupt supports and an optional MPU, making it ideal for highly integrated and cost-sensitive automotive applications.

Cortex-M3 Processor Applications (continued)

- Data communications:
  - The processor’s low power and high efficiency, coupled with instructions in Thumb-2 for bit-field manipulation, make the Cortex-M3 ideal for many communications applications, such as Bluetooth and ZigBee.
- Industrial control:
  - In industrial control applications, simplicity, fast response, and reliability are key factors.
  - Again, the Cortex-M3 processor’s interrupt feature, low interrupt latency, and enhanced fault-handling features make it a strong candidate in this area.

Cortex-M3 Processor Applications (continued)

- Consumer products:
  - In many consumer products, a high-performance microprocessor (or several of them) is used.
  - The Cortex-M3 processor, being a small processor, is highly efficient and low in power and supports an MPU enabling complex software to execute while providing robust memory protection

Advantages of Cortex-M3 Processor

- Greater performance efficiency: Allows more work to be done without increasing the frequency or power requirements
- Low power consumption: Enables longer battery life, especially critical in portable products
- Enhanced determinism: Guarantees that critical tasks and interrupts are serviced as quickly as possible and in a known number of cycles
- Improved code density: Ensures that code fits in even the smallest memory footprints
- Ease of use: Provides easier programmability and debugging for the growing number of 8-bit and 16-bit users migrating to 32 bits
- Lower cost solutions: Reduces 32-bit-based system costs close to those of legacy 8-bit and 16-bit devices and enabling low-end, 32-bit microcontrollers to be priced at less than US$1 for the first time
- Wide choice of development tools: From low-cost or free compilers to full-featured development suites from many development tool vendors

Architecture of ARM Cortex-M3

Architecture of ARM Cortex-M3

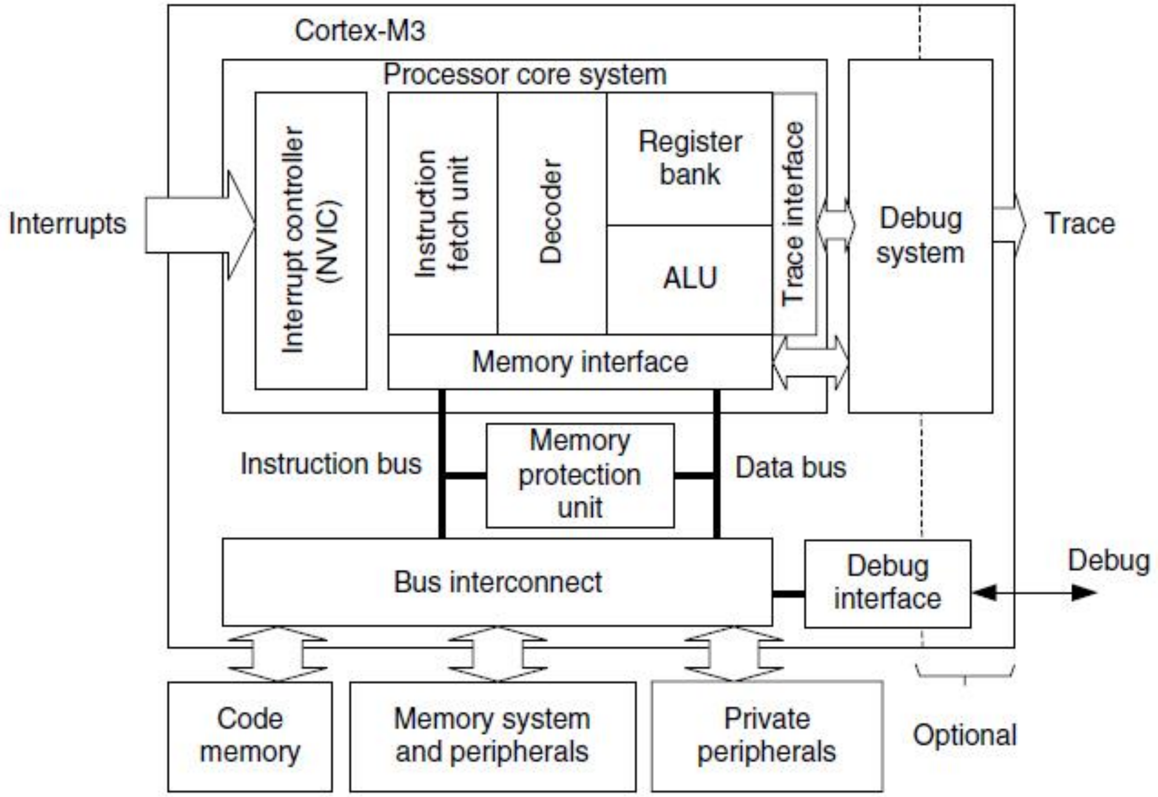


FIGURE 2.1

A Simplified View of the Cortex-M3.

Architecture of ARM Cortex-M3 (continued)

- The Cortex-M3 is a 32-bit microprocessor.
  - It has a 32-bit data path, a 32-bit register bank, and 32-bit memory interfaces.
- The processor has a Harvard architecture, which means that it has a separate instruction bus and data bus.
  - This allows instructions and data accesses to take place at the same time.
  - The performance of the processor increases because data accesses do not affect the instruction pipeline.
  - This feature results in multiple bus interfaces on Cortex-M3, each with optimized usage and the ability to be used simultaneously.
- However, the instruction and data buses share the same memory space (a unified memory system).
  - In other words, you cannot get 8 GB of memory space just because you have separate bus interfaces.

Architecture of ARM Cortex-M3 (continued)

- For complex applications that require more memory system features, the Cortex-M3 processor has an optional Memory Protection Unit (MPU), and it is possible to use an external cache if it’s required.
- Both little endian and big endian memory systems are supported.
- The Cortex-M3 processor includes a number of fixed internal debugging components.
  - These components provide debugging operation supports and features, such as breakpoints and watchpoints.
- In addition, optional components provide debugging features, such as instruction trace, and various types of debugging interfaces.

Features of ARM Cortex-M3

- Three-stage pipeline design
- Harvard bus architecture with unified memory space: instructions and data use the same address space
- 32-bit addressing, supporting 4GB of memory space
- On-chip bus interfaces based on ARM AMBA (Advanced Microcontroller Bus Architecture) Technology, which allow pipelined bus operations for higher throughput
- An interrupt controller called NVIC (Nested Vectored Interrupt Controller) supporting up to 240 interrupt requests and from 8 to 256 interrupt priority levels (dependent on the actual device implementation)

Features of ARM Cortex-M3 (continued)

- Support for various features for OS (Operating System) implementation such as a system tick timer, shadowed stack pointer
- Sleep mode support and various low power features
- Support for an optional MPU (Memory Protection Unit) to provide memory protection features like programmable memory, or access permission control
- Support for bit-data accesses in two specific memory regions using a feature called Bit Band
- The option of being used in single processor or multi-processor designs

Registers

- The Cortex-M3 processor has registers R0 through R15 and a number of special registers.
  - R0-R12: General-Purpose Registers
  - R13: Stack Pointer
  - R14: Link Register
  - R15: Program Counter

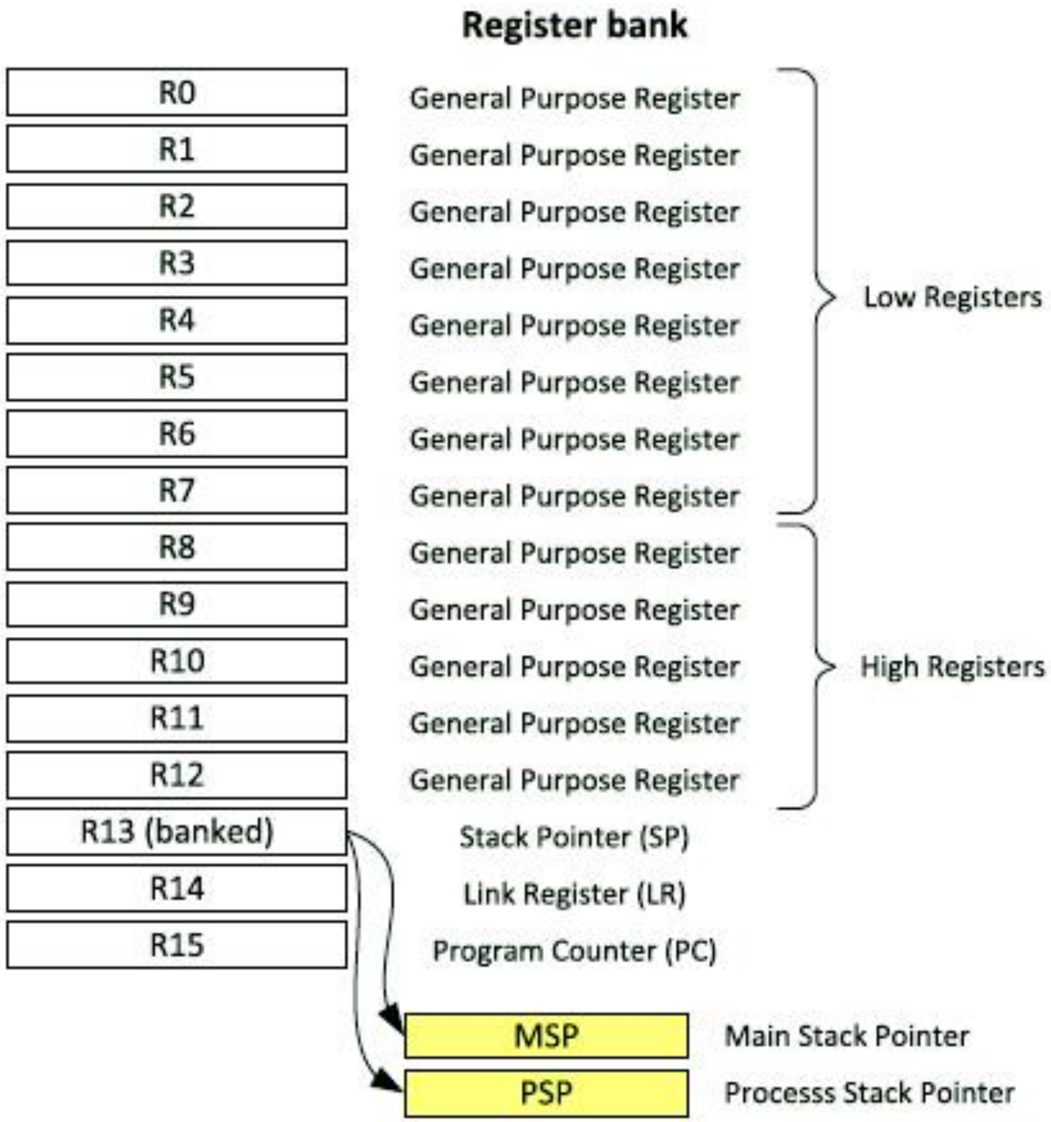


FIGURE 2.2

Registers in the Cortex-M3.

Registers (continued)

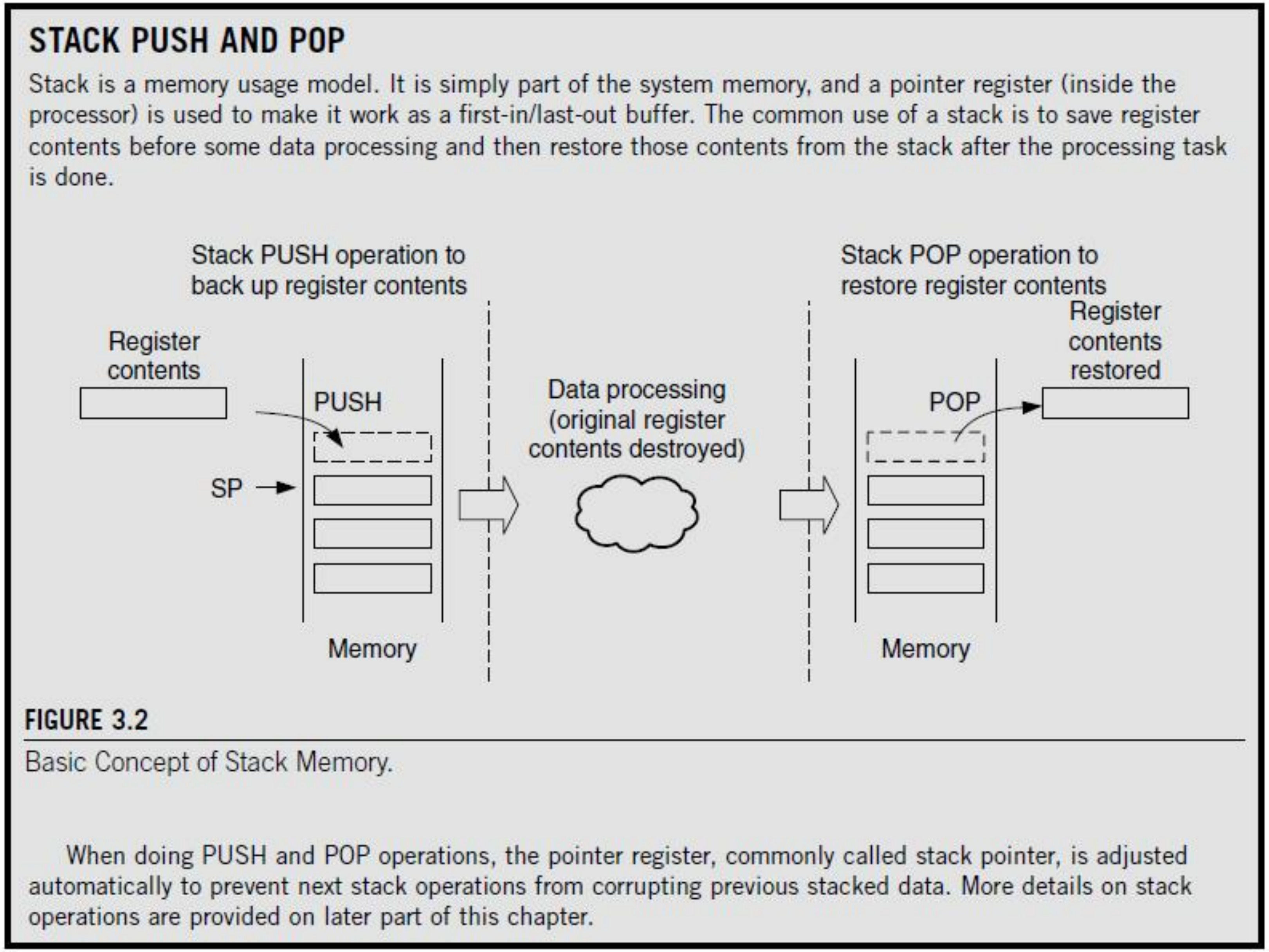
- R0-R12: General-Purpose Registers
  - R0-R12 are 32-bit general-purpose registers for data operations.
  - R0-R7
    - The R0 through R7 general purpose registers are also called low registers.
    - They can be accessed by all 16-bit Thumb instructions and all 32-bit Thumb-2 instructions.
    - These registers are all 32 bits.
    - The reset value is unpredictable
  - R8-R12
    - The R8 through R12 registers are also called high registers.
    - They are accessible by all Thumb-2 instructions but not by all 16-bit Thumb instructions.
    - These registers are all 32 bits
    - The reset value is unpredictable.

Registers (continued)

- R13: Stack Pointer
  - The Cortex-M3 contains two stack pointers (SPs).
  - They are banked so that only one is visible at a time.
    - This duality allows two separate stack memories to be set up.
  - The lowest 2 bits of the stack pointers are always 0, which means they are always word aligned.

Registers (continued)

- R13: Stack Pointer (continued)
  - The two SPs are as follows:
  - Main Stack Pointer (MSP) or *SP_main*:
    - This is the default SP.
    - It is used by the operating system (OS) kernel, exception handlers, and all application codes that require privileged access.
  - Process Stack Pointer (PSP) or *SP_process*:
    - This is used by the user (base-level) application code (when not running an exception handler).
  - When using the register name R13, we can only access the current SP; the other one is inaccessible unless we use special instructions MSR and MRS.



STACK PUSH AND POP

Stack is a memory usage model. It is simply part of the system memory, and a pointer register (inside the processor) is used to make it work as a first-in/last-out buffer. The common use of a stack is to save register contents before some data processing and then restore those contents from the stack after the processing task is done.

Stack PUSH operation to back up register contents
Register contents
PUSH
SP
Memory
Data processing (original register contents destroyed)
Stack POP operation to restore register contents
Register contents restored
POP
Memory

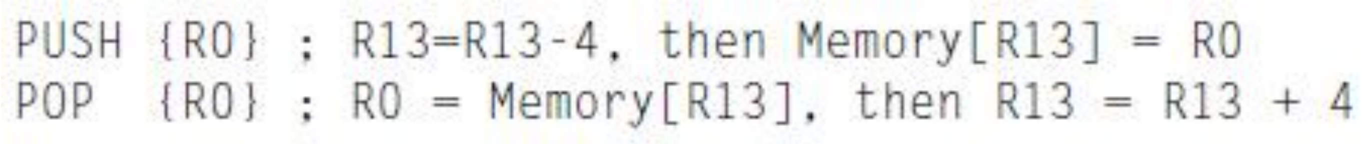
FIGURE 3.2

Basic Concept of Stack Memory.

When doing PUSH and POP operations, the pointer register, commonly called stack pointer, is adjusted automatically to prevent next stack operations from corrupting previous stacked data. More details on stack operations are provided on later part of this chapter.

Registers (continued)

- R13: Stack Pointer (continued)
  - It is not necessary to use both SPs.
  - Simple applications can rely purely on the MSP.
  - In the Cortex-M3, the instructions for accessing stack memory are PUSH and POP.
  - The assembly language syntax is as follows

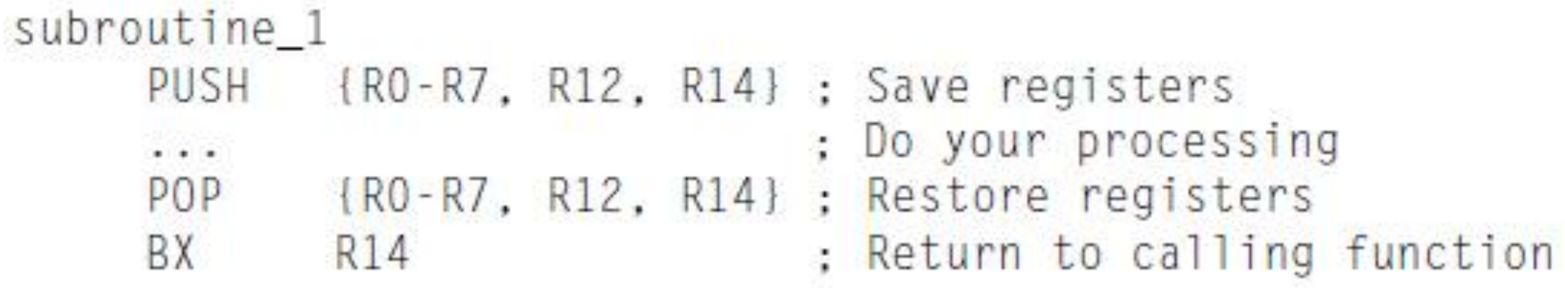


```
PUSH {R0} ; R13=R13-4, then Memory[R13] = R0
POP  {R0} ; R0 = Memory[R13], then R13 = R13 + 4
```

  - PUSH and POP are usually used to save register contents to stack memory at the start of a subroutine and then restore the registers from stack at the end of the subroutine.

Registers (continued)

- R13: Stack Pointer (continued)
  - We can PUSH or POP multiple registers in one instruction:

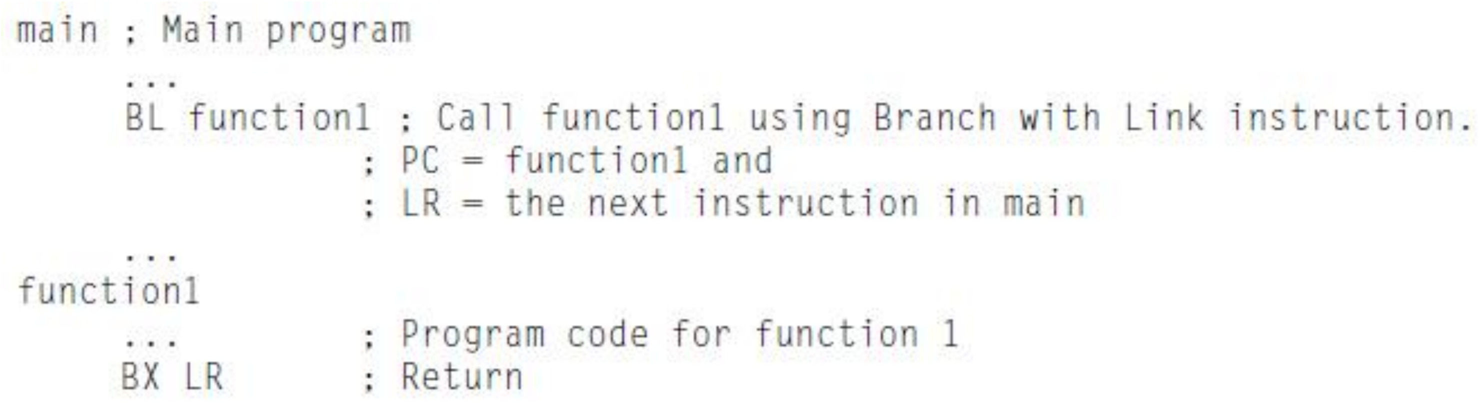


```
subroutine_1
     PUSH   {R0-R7, R12, R14} ; Save registers
     ...                      ; Do your processing
     POP    {R0-R7, R12, R14} ; Restore registers
     BX     R14               ; Return to calling function
```

  - Instead of using R13, you can use SP in program codes.
  - Because register PUSH and POP operations are always word aligned (their addresses must be 0x0, 0x4, 0x8, ...), the SP/R13 bit 0 and bit 1 are hardwired to 0 and always read as zero (RAZ).

Registers (continued)

- R14: Link Register
  - Inside an assembly program, we can write it as either R14 or LR.
  - LR is used to store the return program counter (PC) when a subroutine or function is called.
  - E.g.: when we’re using the branch and link (BL) instruction:



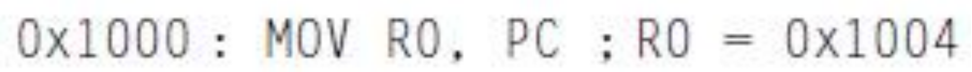
```
main ; Main program
     ...
     BL function1 ; Call function1 using Branch with Link instruction.
                 ; PC = function1 and
                 ; LR = the next instruction in main
     ...
function1
     ...         ; Program code for function 1
     BX LR       ; Return
```

Registers (continued)

- R14: Link Register (continued)
  - Despite the fact that bit 0 of the PC is always 0 (because instructions are word aligned or half word aligned), the LR bit 0 is readable and writable.
  - This is because in the Thumb instruction set, bit 0 is often used to indicate ARM/Thumb states.
  - To allow the Thumb-2 program for the Cortex-M3 to work with other ARM processors that support the Thumb-2 technology, this least significant bit (LSB) is writable and readable.

Registers (continued)

- R15: Program Counter
  - The program counter is the current program address.
  - This register can be written to control the program flow.
  - It can be accessed in assembler code by either R15 or PC.
  - Because of the pipelined nature of the Cortex-M3 processor, when you read this register, you will find that the value is different than the location of the executing instruction, normally by 4.
    - For example:



```
0x1000 : MOV R0, PC ; R0 = 0x1004
```

Registers (continued)

- R15: Program Counter (continued)
  - Because an instruction address must be half word aligned, the LSB (bit 0) of the PC read value is always 0.
  - However, in branching, either by writing to PC or using branch instructions, the LSB of the target address should be set to 1 because it is used to indicate the Thumb state operations.
  - If it is 0, it can imply trying to switch to the ARM state and will result in a fault exception in the Cortex-M3.

Special Registers

- The Cortex-M3 processor also has a number of special registers.
  - They are as follows:
    - Program Status Registers (PSRs)
    - Interrupt Mask Registers (PRIMASK, FAULTMASK, and BASEPRI)
    - Control Register (CONTROL)
- These registers have special functions and can be accessed only by special instructions.
- They cannot be used for normal data processing.

Special Registers (continued)

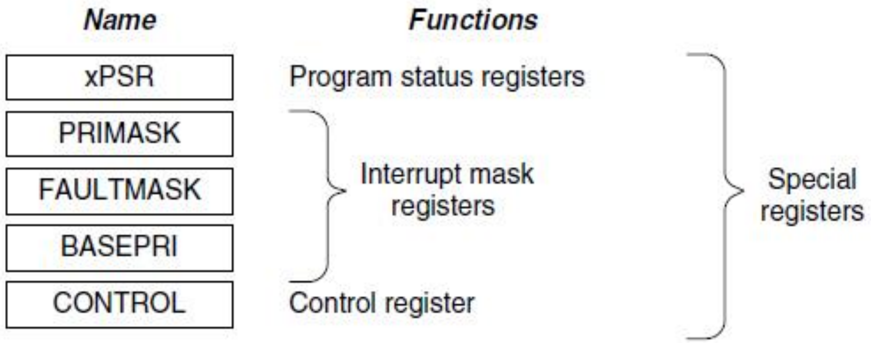


FIGURE 2.3

Special Registers in the Cortex-M3.

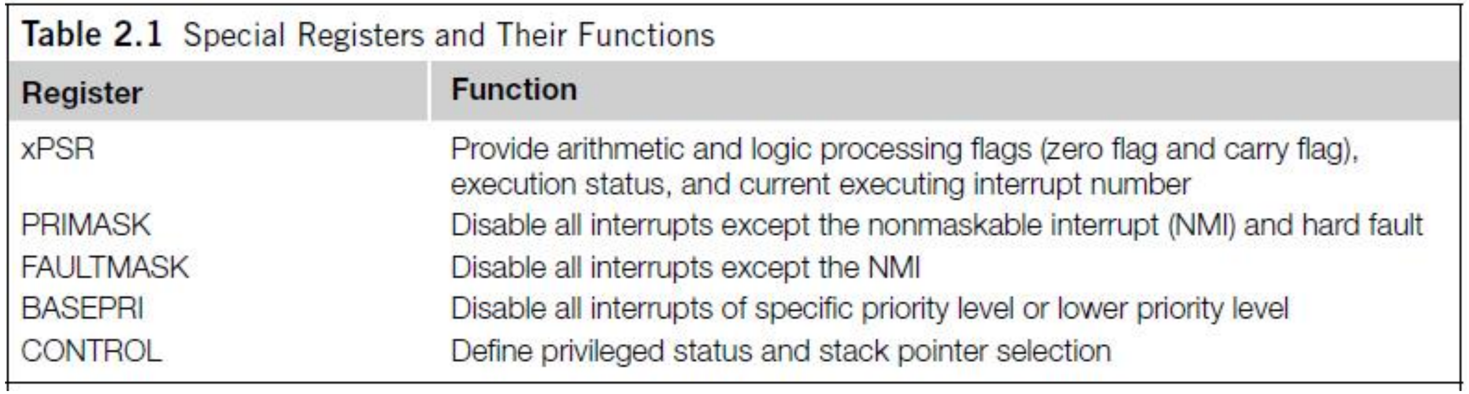
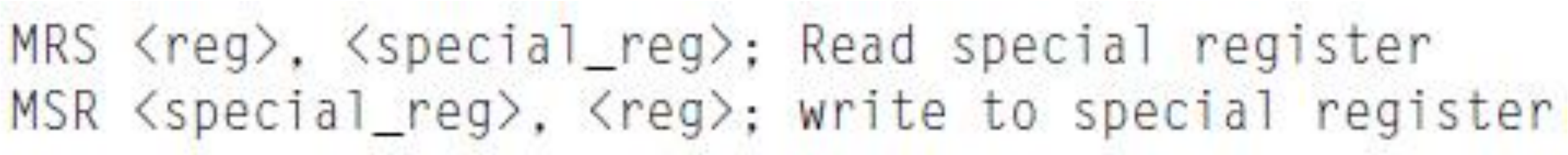


Table 2.1 Special Registers and Their Functions

Register	Function
xPSR	Provide arithmetic and logic processing flags (zero flag and carry flag), execution status, and current executing interrupt number
PRIMASK	Disable all interrupts except the nonmaskable interrupt (NMI) and hard fault
FAULTMASK	Disable all interrupts except the NMI
BASEPRI	Disable all interrupts of specific priority level or lower priority level
CONTROL	Define privileged status and stack pointer selection

Special Registers (continued)

- Special registers can only be accessed via MSR and MRS instructions; they do not have memory addresses.



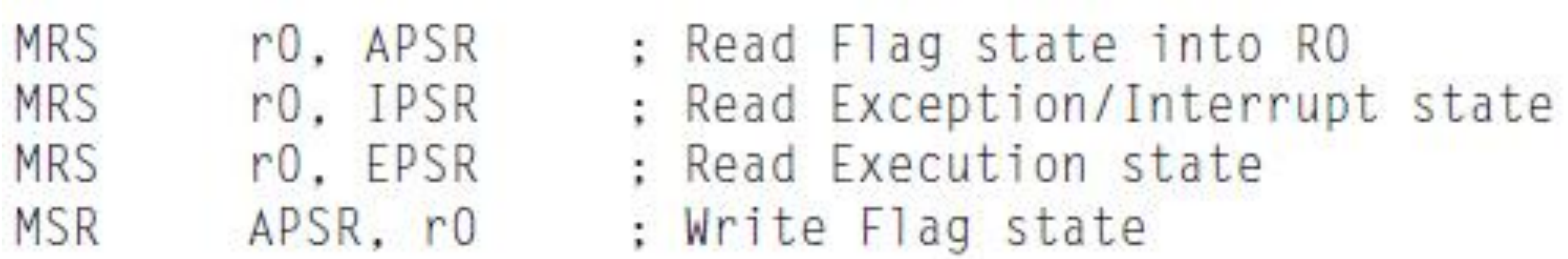
```
MRS <reg>, <special_reg>; Read special register
MSR <special_reg>, <reg>; write to special register
```

Program Status Registers

- The PSRs are subdivided into three status registers:
  - Application Program Status register (APSR)
  - Interrupt Program Status register (IPSR)
  - Execution Program Status register (EPSR)
- The three PSRs can be accessed together or separately using the special register access instructions MSR and MRS.
- When they are accessed as a collective item, the name *xPSR* is used.

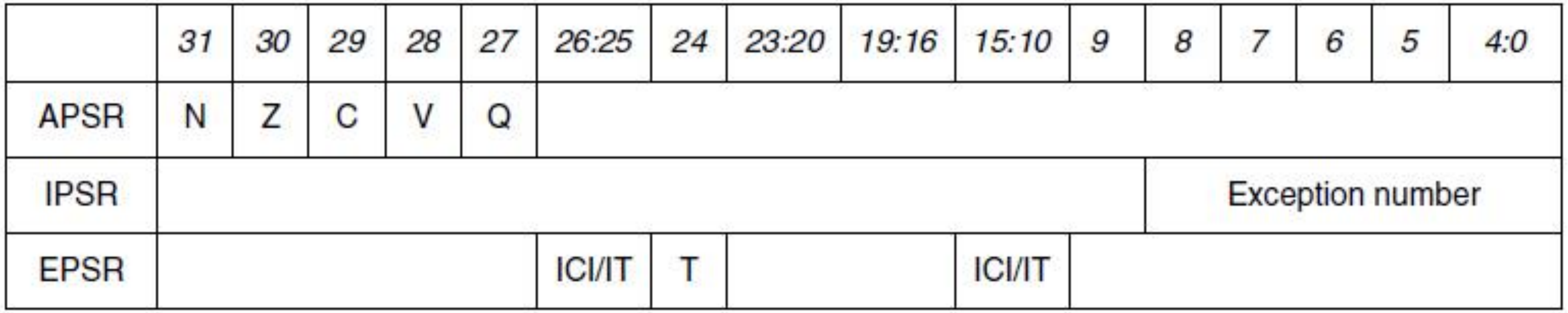
Program Status Registers (continued)

- You can read the PSRs using the MRS instruction.
- You can also change the APSR using the MSR instruction, but EPSR and IPSR are read-only.
- For example:



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

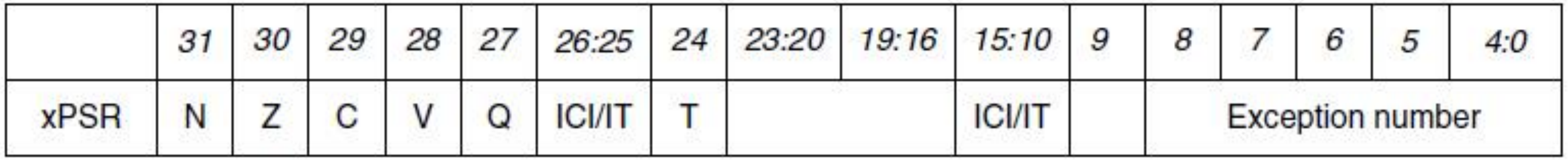
Program Status Registers (continued)



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

FIGURE 3.3

Program Status Registers (PSRs) in the Cortex-M3.



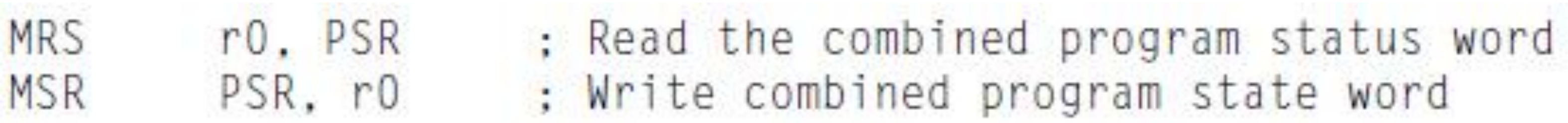
	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
xPSR	N	Z	C	V	Q	ICI/IT	T			ICI/IT		Exception number				

FIGURE 3.4

Combined Program Status Registers (xPSR) in the Cortex-M3.

Program Status Registers (continued)

- In ARM assembler, when accessing *xPSR* (all three PSRs as one), the symbol PSR is used:



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

- The descriptions for the bit fields in PSR are shown in Table 3.1.

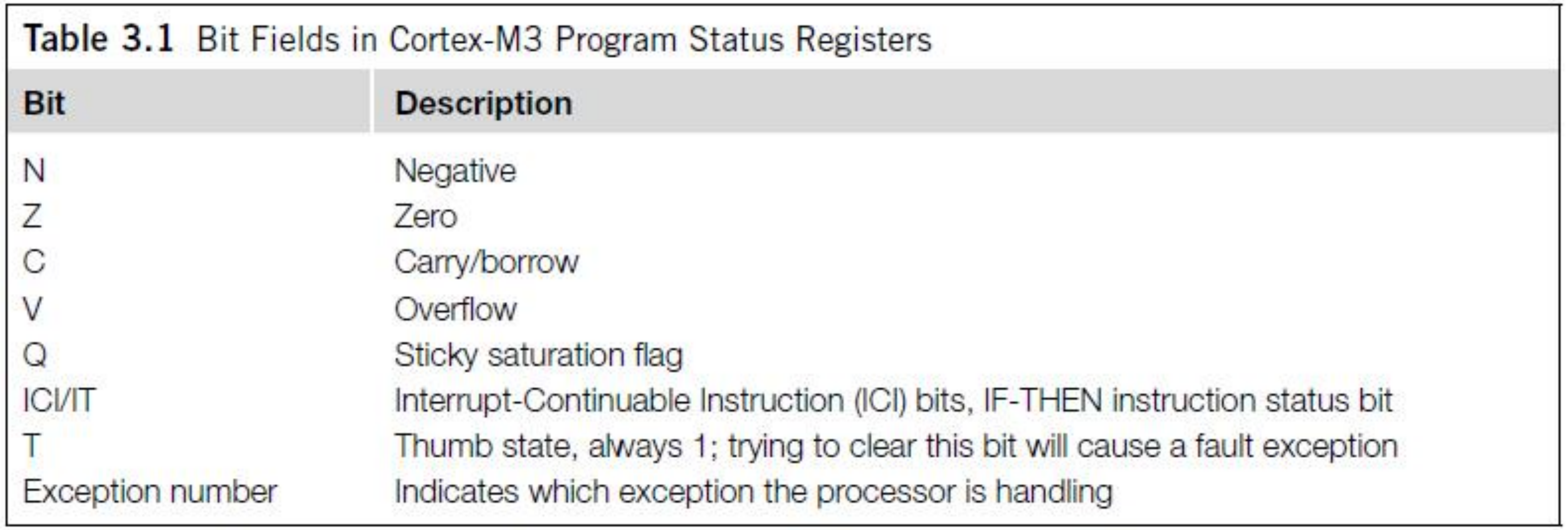
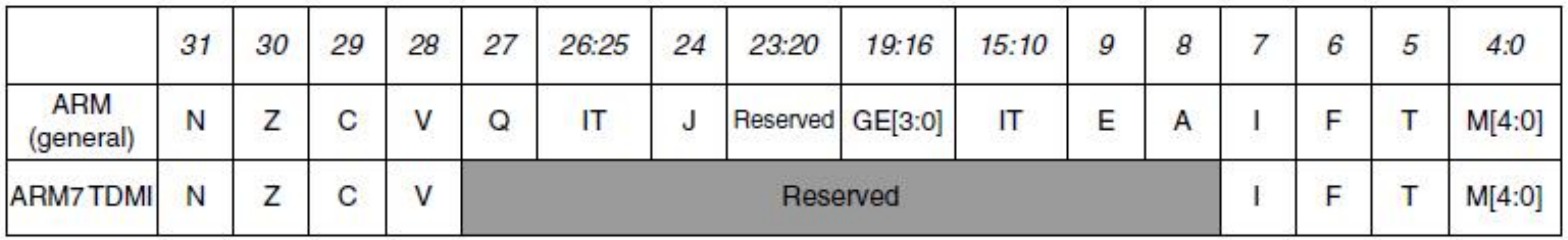


Table 3.1 Bit Fields in Cortex-M3 Program Status Registers

Bit	Description
N	Negative
Z	Zero
C	Carry/borrow
V	Overflow
Q	Sticky saturation flag
ICI/IT	Interrupt-Continuable Instruction (ICI) bits, IF-THEN instruction status bit
T	Thumb state, always 1; trying to clear this bit will cause a fault exception
Exception number	Indicates which exception the processor is handling

Program Status Registers (continued)

- If you compare this with the Current Program Status register (CPSR) in ARM7, you might find that some bit fields that were used in ARM7 are gone.
  - The Mode (M) bit field is gone because the Cortex-M3 does not have the operation mode as defined in ARM7.
  - Thumb-bit (T) is moved to bit 24.
  - Interrupt status (I and F) bits are replaced by the new interrupt mask registers (PRIMASKs), which are separated from PSR.
- For comparison, the CPSR in traditional ARM processors is shown in Figure 3.5.



	31	30	29	28	27	26:25	24	23:20	19:16	15:10	9	8	7	6	5	4:0
ARM (general)	N	Z	C	V	Q	IT	J	Reserved	GE[3:0]	IT	E	A	I	F	T	M[4:0]
ARM7 TDMI	N	Z	C	V	Reserved								I	F	T	M[4:0]

FIGURE 3.5

Current Program Status Registers in Traditional ARM Processors.

Interrupt Mask Registers

- The PRIMASK, FAULTMASK, and BASEPRI registers are used to disable exceptions.
- The PRIMASK and BASEPRI registers are useful for temporarily disabling interrupts in timing-critical tasks.
- An OS could use FAULTMASK to temporarily disable fault handling when a task has crashed.
    - In this scenario, a number of different faults might be taking place when a task crashes.
    - Once the core starts cleaning up, it might not want to be interrupted by other faults caused by the crashed process.
    - Therefore, the FAULTMASK gives the OS kernel time to deal with fault conditions.

Interrupt Mask Registers (continued)

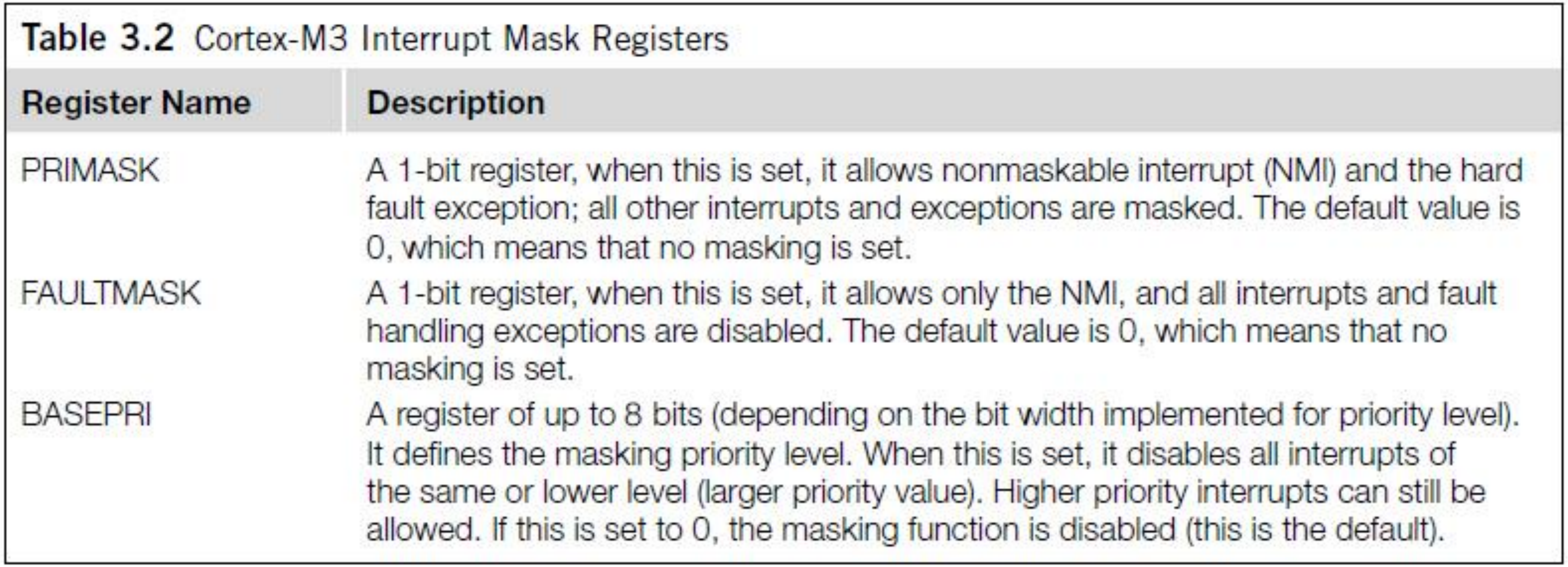
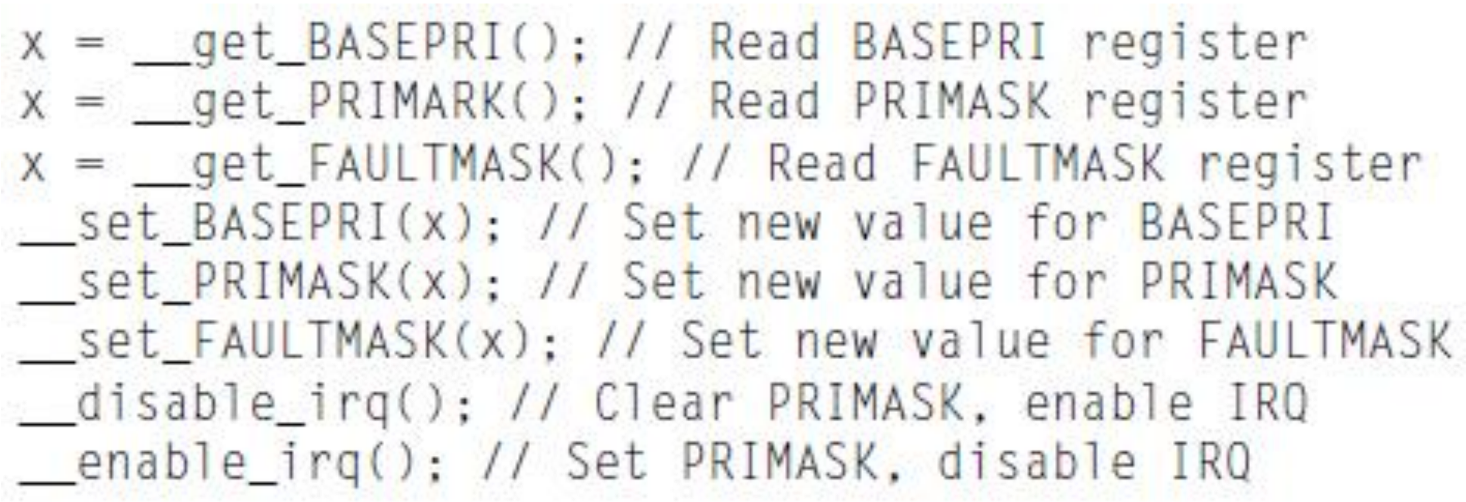


Table 3.2 Cortex-M3 Interrupt Mask Registers

Register Name	Description
PRIMASK	A 1-bit register, when this is set, it allows nonmaskable interrupt (NMI) and the hard fault exception; all other interrupts and exceptions are masked. The default value is 0, which means that no masking is set.
FAULTMASK	A 1-bit register, when this is set, it allows only the NMI, and all interrupts and fault handling exceptions are disabled. The default value is 0, which means that no masking is set.
BASEPRI	A register of up to 8 bits (depending on the bit width implemented for priority level). It defines the masking priority level. When this is set, it disables all interrupts of the same or lower level (larger priority value). Higher priority interrupts can still be allowed. If this is set to 0, the masking function is disabled (this is the default).

Interrupt Mask Registers (continued)

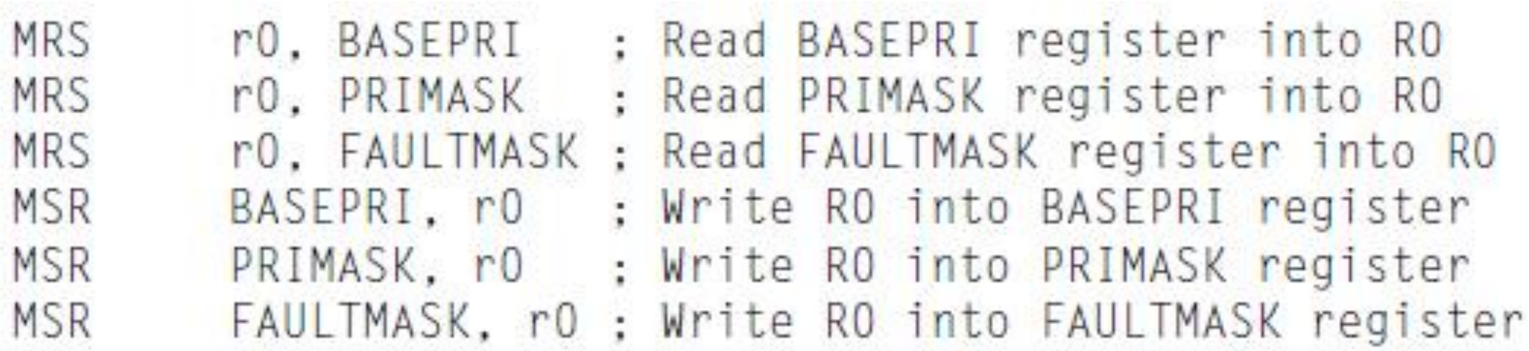
- To access the PRIMASK, FAULTMASK, and BASEPRI registers, a number of functions are available in the device driver libraries provided by the microcontroller vendors.
- For example, the following:



```
x = __get_BASEPRI(); // Read BASEPRI register
x = __get_PRIMARK(); // Read PRIMASK register
x = __get_FAULTMASK(); // Read FAULTMASK register
__set_BASEPRI(x); // Set new value for BASEPRI
__set_PRIMASK(x); // Set new value for PRIMASK
__set_FAULTMASK(x); // Set new value for FAULTMASK
__disable_irq(); // Clear PRIMASK, enable IRQ
__enable_irq(); // Set PRIMASK, disable IRQ
```

Interrupt Mask Registers (continued)

- In assembly language, the MRS and MSR instructions are used.
- For example:



```
MRS     r0, BASEPRI   ; Read BASEPRI register into R0
MRS     r0, PRIMASK   ; Read PRIMASK register into R0
MRS     r0, FAULTMASK ; Read FAULTMASK register into R0
MSR     BASEPRI, r0   ; Write R0 into BASEPRI register
MSR     PRIMASK, r0   ; Write R0 into PRIMASK register
MSR     FAULTMASK, r0 ; Write R0 into FAULTMASK register
```

- The PRIMASK, FAULTMASK, and BASEPRI registers cannot be set in the user access level.

The Control Register

- The control register is used to define the privilege level and the SP selection.
- This register has 2 bits, as shown in Table 3.3.

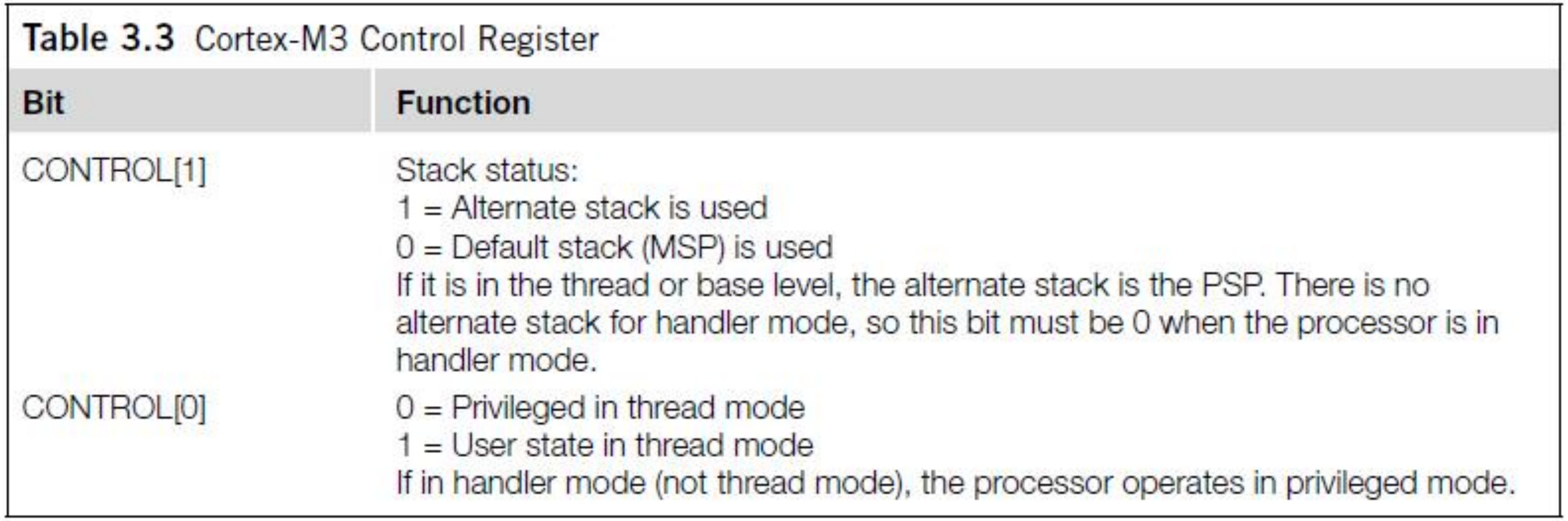


Table 3.3 Cortex-M3 Control Register

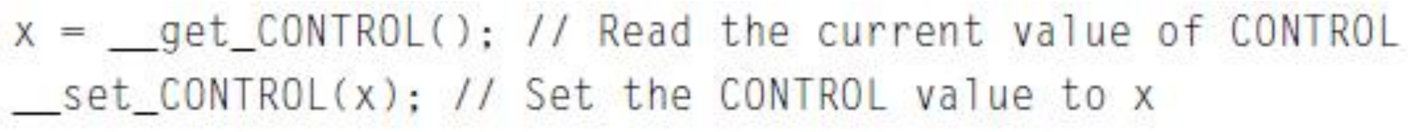
Bit	Function
CONTROL[1]	Stack status: 1 = Alternate stack is used 0 = Default stack (MSP) is used If it is in the thread or base level, the alternate stack is the PSP. There is no alternate stack for handler mode, so this bit must be 0 when the processor is in handler mode.
CONTROL[0]	0 = Privileged in thread mode 1 = User state in thread mode If in handler mode (not thread mode), the processor operates in privileged mode.

The Control Register (continued)

- CONTROL[1]
  - In the Cortex-M3, the CONTROL[1] bit is always 0 in handler mode.
  - However, in the thread or base level, it can be either 0 or 1.
  - This bit is writable only when the core is in thread mode and privileged.
  - In the user state or handler mode, writing to this bit is not allowed.
  - Aside from writing to this register, another way to change this bit is to change bit 2 of the LR when in exception return.

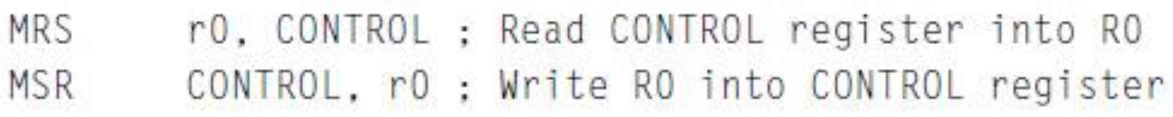
The Control Register (continued)

- CONTROL[0]
  - The CONTROL[0] bit is writable only in a privileged state.
  - Once it enters the user state, the only way to switch back to privileged is to trigger an interrupt and change this in the exception handler.
  - To access the control register in C, the following Cortex Microcontroller Software Interface Standard (CMSIS) functions are available in CMSIS compliant device driver libraries:



```
x = __get_CONTROL(); // Read the current value of CONTROL
__set_CONTROL(x); // Set the CONTROL value to x
```

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

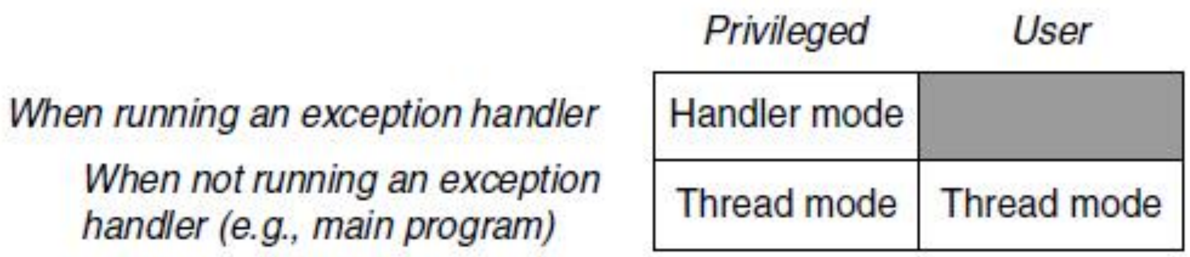


```
MRS     r0, CONTROL ; Read CONTROL register into R0
MSR     CONTROL, r0 ; Write R0 into CONTROL register
```

Operation Modes

Operation Modes

- The Cortex-M3 processor has two operation modes – thread mode and handler mode, and two privilege levels – privileged level and user level.
- The operation modes determine whether the processor is running a normal program or running an exception handler like an interrupt handler or system exception handler.
- The privilege levels provide a mechanism for safeguarding memory accesses to critical regions as well as providing a basic security model.



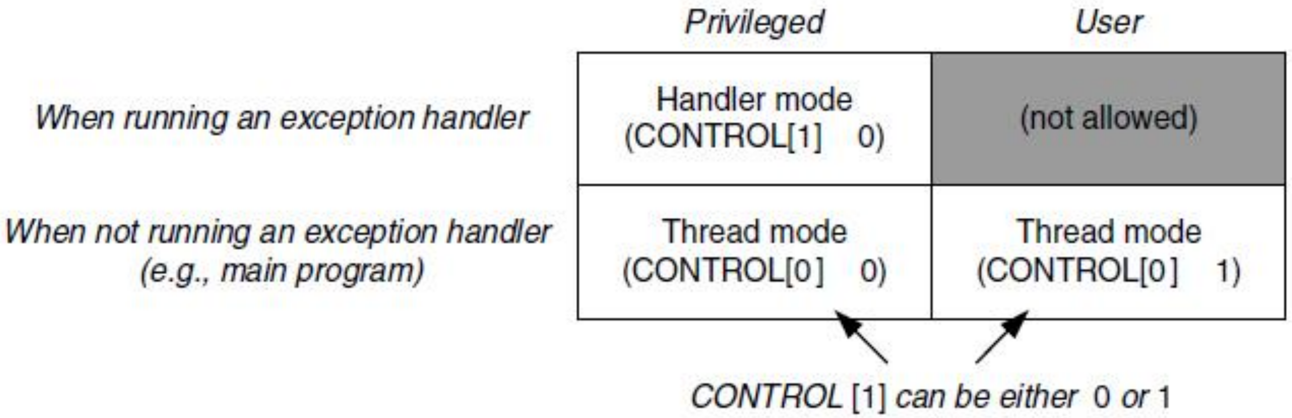
	Privileged	User
When running an exception handler	Handler mode	▨
When not running an exception handler (e.g., main program)	Thread mode	Thread mode

FIGURE 2.4

Operation Modes and Privilege Levels in Cortex-M3.

Operation Modes (continued)

- When the processor is running a main program (thread mode), it can be in either the privileged or user level, but exception handlers can only be in the privileged level.



	Privileged	*User*
When running an exception handler	Handler mode (CONTROL[1] 0)	(not allowed)
When not running an exception handler *(e.g., main program)*	Thread mode (CONTROL[0] 0)	Thread mode (CONTROL[0] 1)

CONTROL [1] *can be either* 0 *or* 1

FIGURE 3.6

Operation Modes and Privilege Levels in Cortex-M3.

Operation Modes (continued)

- When the processor exits reset, it is in thread mode, with privileged access rights.
- In the privileged state, a program has access to all memory ranges (except when prohibited by MPU settings) and can use all supported instructions.

Operation Modes (continued)

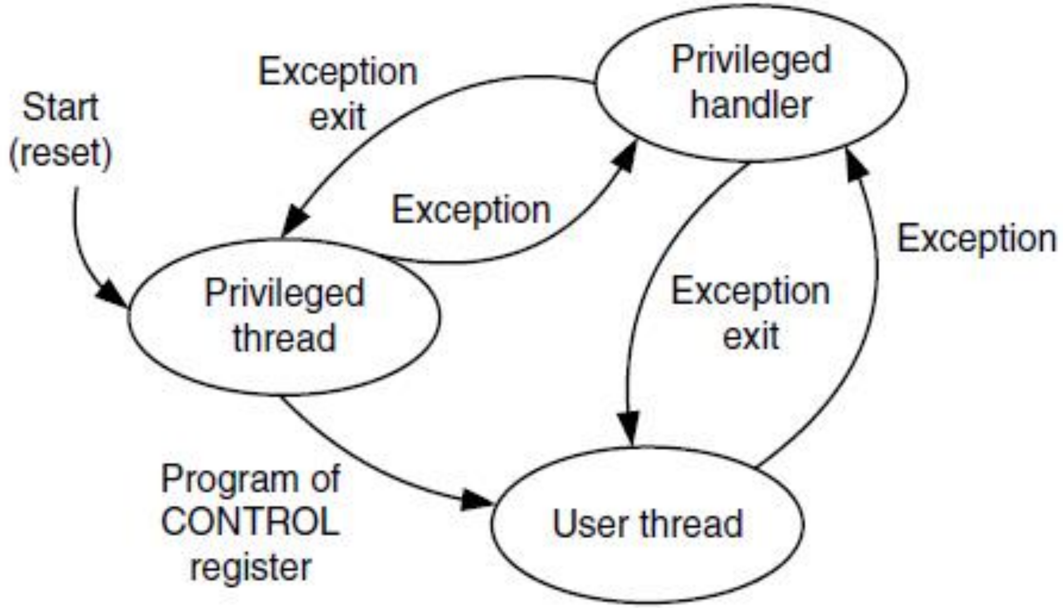


FIGURE 2.5

Allowed Operation Mode Transitions.

Operation Modes (continued)

- Software in the privileged access level can switch the program into the user access level using the control register.
- When an exception takes place, the processor will always switch back to the privileged state and return to the previous state when exiting the exception handler.
- A user program cannot change back to the privileged state by writing to the control register.
  - It has to go through an exception handler that programs the control register to switch the processor back into the privileged access level when returning to thread mode.

Operation Modes (continued)

- The separation of privilege and user levels improves system reliability by preventing system configuration registers from being accessed or changed by some untrusted programs.
- If an MPU is available, it can be used in conjunction with privilege levels to protect critical memory locations, such as programs and data for OSs.
  - For example, with privileged accesses, usually used by the OS kernel, all memory locations can be accessed (unless prohibited by MPU setup).
  - When the OS launches a user application, it is likely to be executed in the user access level to protect the system from failing due to a crash of untrusted user programs.

Operation Modes (continued)

- In the user access level (thread mode), access to the system control space (SCS)—a part of the memory region for configuration registers and debugging components—is blocked.
- Furthermore, instructions that access special registers (such as MSR, except when accessing APSR) cannot be used.
- If a program running at the user access level tries to access SCS or special registers, a fault exception will occur.

Operation Modes (continued)

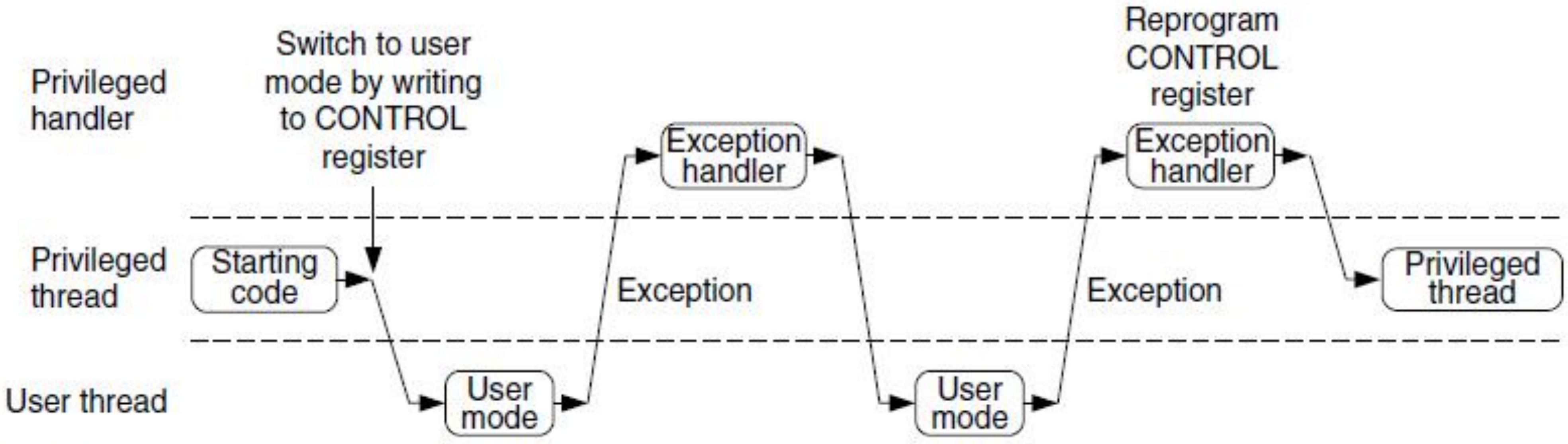


FIGURE 3.7

Switching of Operation Mode by Programming the Control Register or by Exceptions.

Operation Modes (continued)

- In simple applications, there is no need to separate the privileged and user access levels.
  - In these cases, there is no need to use user access level and no need to program the control register.

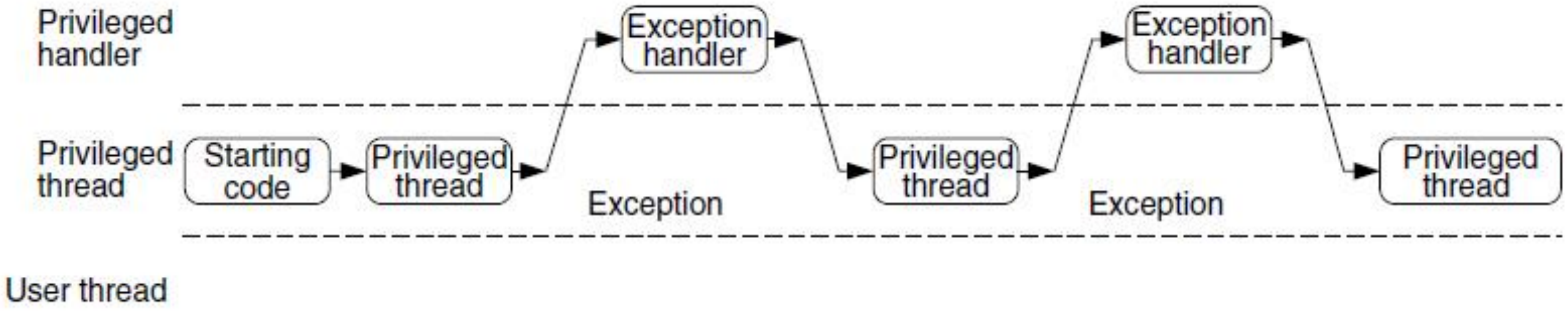


FIGURE 3.8

Simple Applications Do Not Require User Access Level in Thread Mode.

Operation Modes (continued)

- The mode and access level of the processor are defined by the control register.
- When the control register bit 0 is 0, the processor mode changes when an exception takes place.
- When control register bit 0 is 1 (thread running user application), both processor mode and access level change when an exception takes place.
- Control register bit 0 is programmable only in the privileged level.
  - For a user-level program to switch to privileged state, it has to raise an interrupt (for example, supervisor call [SVC]) and write to CONTROL[0] within the handler.

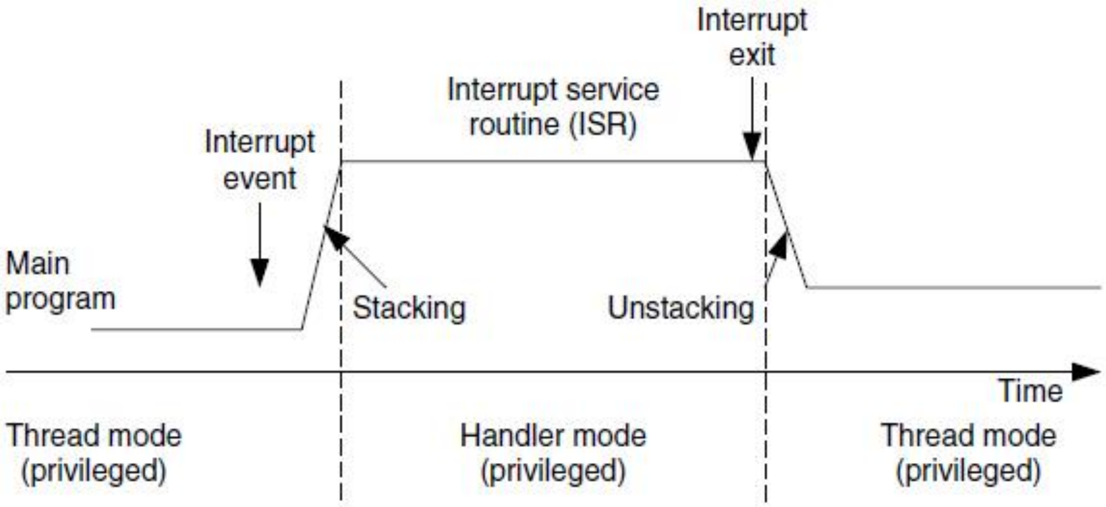


FIGURE 3.9

Switching Processor Mode at Interrupt.

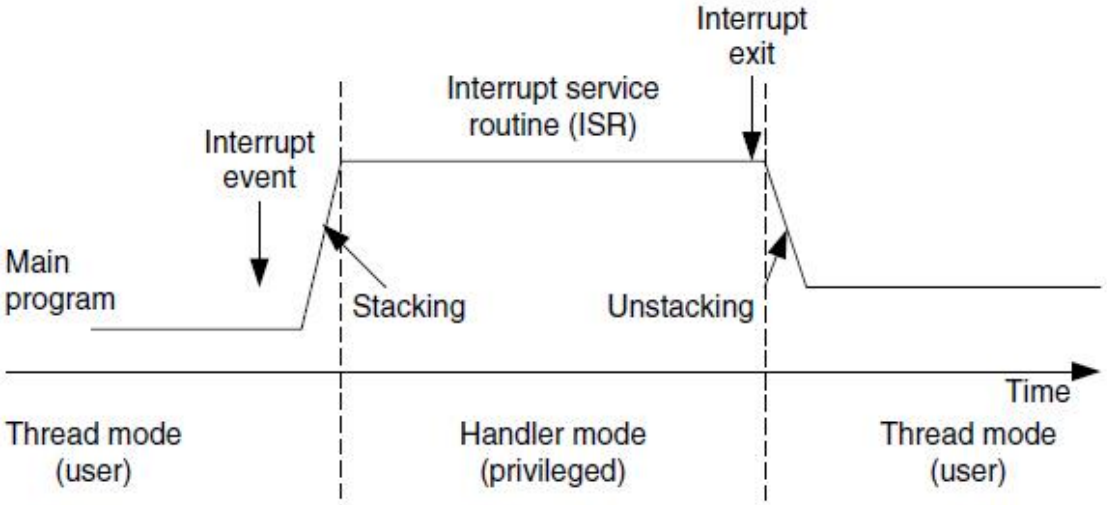


FIGURE 3.10

Switching Processor Mode and Privilege Level at Interrupt.

The Memory Map

- The Cortex-M3 has a predefined memory map.
- This allows the built-in peripherals, such as the interrupt controller and the debug components, to be accessed by simple memory access instructions.
- Thus, most system features are accessible in C program code.
- The predefined memory map also allows the Cortex-M3 processor to be highly optimized for speed and ease of integration in system-on-a-chip (SoC) designs.
- Overall, the 4 GB memory space can be divided into ranges as shown in Figure 2.6.

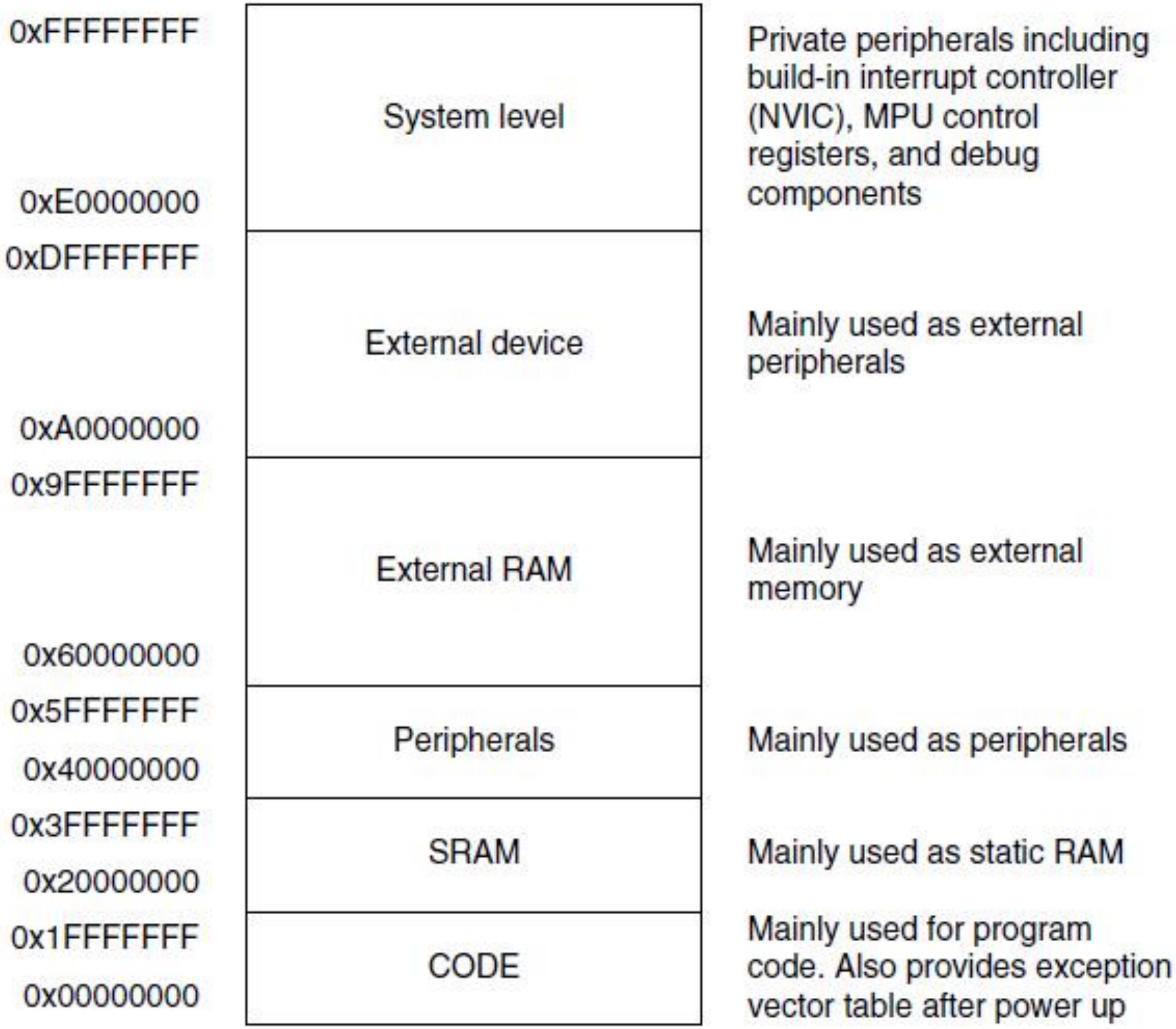

0xFFFFFFFF
0xE0000000
System level
Private peripherals including build-in interrupt controller (NVIC), MPU control registers, and debug components
0xDFFFFFFF
0xA0000000
External device
Mainly used as external peripherals
0x9FFFFFFF
0x60000000
External RAM
Mainly used as external memory
0x5FFFFFFF
0x40000000
Peripherals
Mainly used as peripherals
0x3FFFFFFF
0x20000000
SRAM
Mainly used as static RAM
0x1FFFFFFF
0x00000000
CODE
Mainly used for program code. Also provides exception vector table after power up

FIGURE 2.6

The Cortex-M3 Memory Map.

The Memory Map (continued)

- The Cortex-M3 design has an internal bus infrastructure optimized for this memory usage.
- In addition, the design allows these regions to be used differently.
- For example, data memory can still be put into the CODE region, and program code can be executed from an external Random Access Memory (RAM) region.
- The system-level memory region contains the interrupt controller and the debug components.
- By having fixed addresses for these peripherals, you can port applications between different Cortex-M3 products much more easily.

The Bus Interface

- There are several bus interfaces on the Cortex-M3 processor.
- They allow the Cortex-M3 to carry instruction fetches and data accesses at the same time.
- The main bus interfaces are as follows:
  - Code memory buses
    - The code memory region access is carried out on the code memory buses, which physically consist of two buses, one called I-Code and other called D-Code.
    - These are optimized for instruction fetches for best instruction execution speed.
  - System bus
    - The system bus is used to access memory and peripherals.
    - This provides access to the Static Random Access Memory (SRAM), peripherals, external RAM, external devices, and part of the system-level memory regions.
  - Private peripheral bus
    - The private peripheral bus provides access to a part of the system-level memory dedicated to private peripherals, such as debugging components.

The Memory Protection Unit (MPU)

- The Cortex-M3 has an optional MPU.
- This unit allows access rules to be set up for privileged access and user program access.
- When an access rule is violated, a fault exception is generated, and the fault exception handler will be able to analyse the problem and correct it, if possible.
- The MPU can be used in various ways.
  - In common scenarios, the OS can set up the MPU to protect data use by the OS kernel and other privileged processes to be protected from untrusted user programs.
  - The MPU can also be used to make memory regions read-only, to prevent accidental erasing of data or to isolate memory regions between different tasks in a multitasking system.
- Overall, it can help make embedded systems more robust and reliable.
- The MPU feature is optional and is determined during the implementation stage of the microcontroller or SoC design.

Nested Vectored Interrupt Controller (NVIC)

- The Cortex-M3 processor includes an interrupt controller called the Nested Vectored Interrupt Controller (NVIC).
- It is closely coupled to the processor core and provides a number of features as follows:
  - Nested interrupt support
  - Vectored interrupt support
  - Dynamic priority changes support
  - Reduction of interrupt latency
  - Interrupt masking

Nested Vectored Interrupt Controller (NVIC) (continued)

- Nested Interrupt Support
  - All the external interrupts and most of the system exceptions can be programmed to different priority levels.
  - When an interrupt occurs, the NVIC compares the priority of this interrupt to the current running priority level.
  - If the priority of the new interrupt is higher than the current level, the interrupt handler of the new interrupt will override the current running task.
- Vectored Interrupt Support
  - When an interrupt is accepted, the starting address of the interrupt service routine (ISR) is located from a vector table in memory.
  - There is no need to use software to determine and branch to the starting address of the ISR.
  - Thus, it takes less time to process the interrupt request.

Nested Vectored Interrupt Controller (NVIC) (continued)

- Dynamic Priority Changes Support
  - Priority levels of interrupts can be changed by software during run time.
  - Interrupts that are being serviced are blocked from further activation until the ISR is completed, so their priority can be changed without risk of accidental re-entry.
- Reduction of Interrupt Latency
  - The Cortex-M3 processor also includes a number of advanced features to lower the interrupt latency.
  - These include automatic saving and restoring some register contents, reducing delay in switching from one ISR to another, and handling of late arrival interrupts.

Nested Vectored Interrupt Controller (NVIC) (continued)

- Interrupt Masking
  - Interrupts and system exceptions can be masked based on their priority level or masked completely using the interrupt masking registers BASEPRI, PRIMASK, and FAULTMASK.
  - They can be used to ensure that time-critical tasks can be finished on time without being interrupted.

Interrupts and Exceptions

- The Cortex-M3 processor implements a new exception model, introduced in the ARMv7-M architecture.
  - Enables very efficient exception handling.
- The Cortex-M3 supports a number of exceptions, including a fixed number of system exceptions and a number of external Interrupt Request (IRQs) (external interrupt inputs).
- Interrupt priority handling and nested interrupt support are now included in the interrupt architecture.
- The interrupt features in the Cortex-M3 are implemented in the NVIC.
- Aside from supporting external interrupts, the Cortex-M3 also supports a number of internal exception sources, such as system fault handling.
- As a result, the Cortex-M3 has a number of predefined exception types, as shown in Table 2.2.

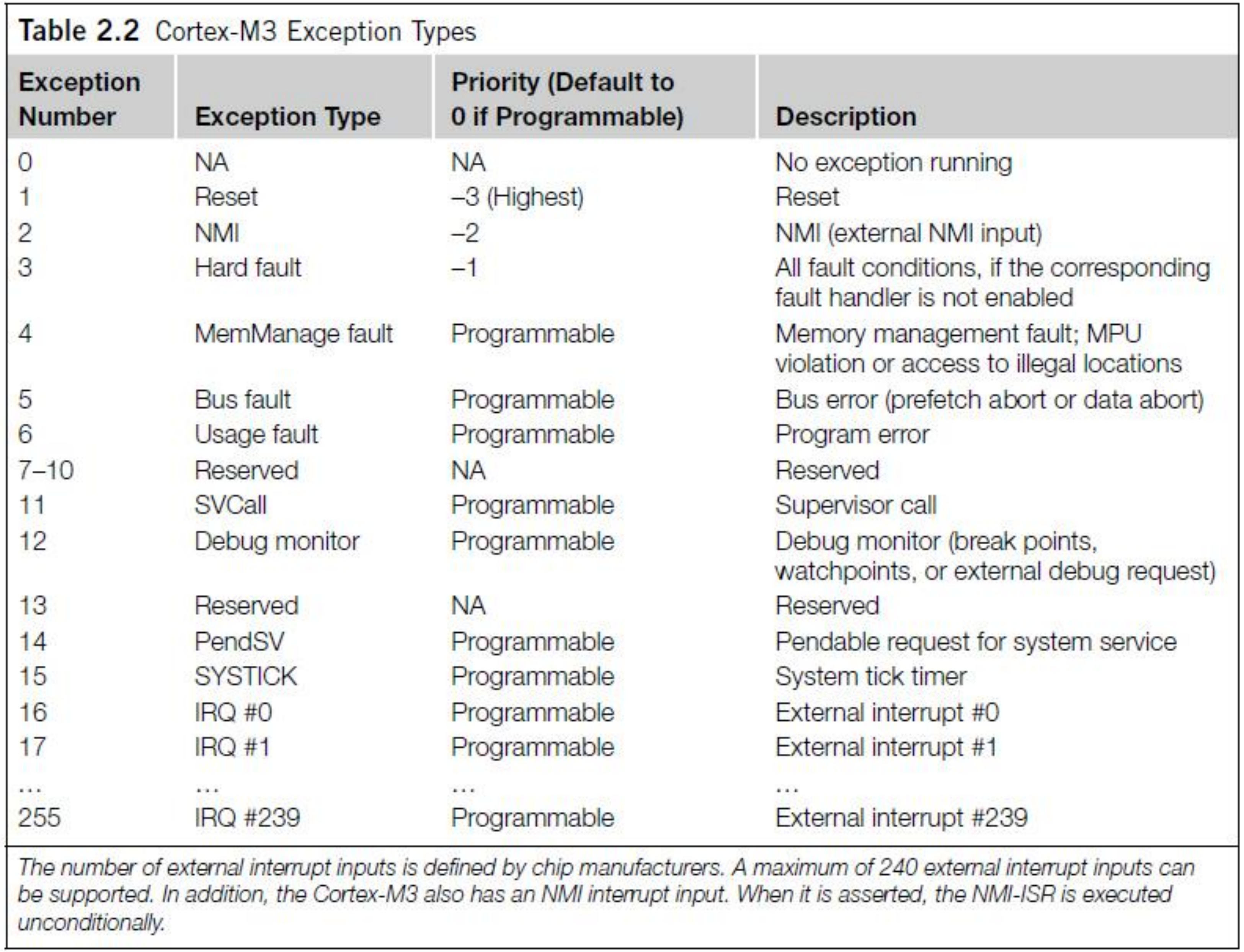


Table 2.2 Cortex-M3 Exception Types

Exception Number	Exception Type	Priority (Default to 0 if Programmable)	Description
0	NA	NA	No exception running
1	Reset	–3 (Highest)	Reset
2	NMI	–2	NMI (external NMI input)
3	Hard fault	–1	All fault conditions, if the corresponding fault handler is not enabled
4	MemManage fault	Programmable	Memory management fault; MPU violation or access to illegal locations
5	Bus fault	Programmable	Bus error (prefetch abort or data abort)
6	Usage fault	Programmable	Program error
7–10	Reserved	NA	Reserved
11	SVCall	Programmable	Supervisor call
12	Debug monitor	Programmable	Debug monitor (break points, watchpoints, or external debug request)
13	Reserved	NA	Reserved
14	PendSV	Programmable	Pendable request for system service
15	SYSTICK	Programmable	System tick timer
16	IRQ #0	Programmable	External interrupt #0
17	IRQ #1	Programmable	External interrupt #1
...	...	...	...
255	IRQ #239	Programmable	External interrupt #239

The number of external interrupt inputs is defined by chip manufacturers. A maximum of 240 external interrupt inputs can be supported. In addition, the Cortex-M3 also has an NMI interrupt input. When it is asserted, the NMI-ISR is executed unconditionally.

Interrupts and Exceptions (continued)

- The number of interrupt inputs on a Cortex-M3 microcontroller depends on the individual design.
  - The typical number of interrupt inputs is 16 or 32.
- Besides the interrupt inputs, there is also a nonmaskable interrupt (NMI) input signal.
  - In most cases, the NMI could be connected to a watchdog timer or a voltage-monitoring block that warns the processor when the voltage drops below a certain level.
  - The NMI exception can be activated any time, even right after the core exits reset.

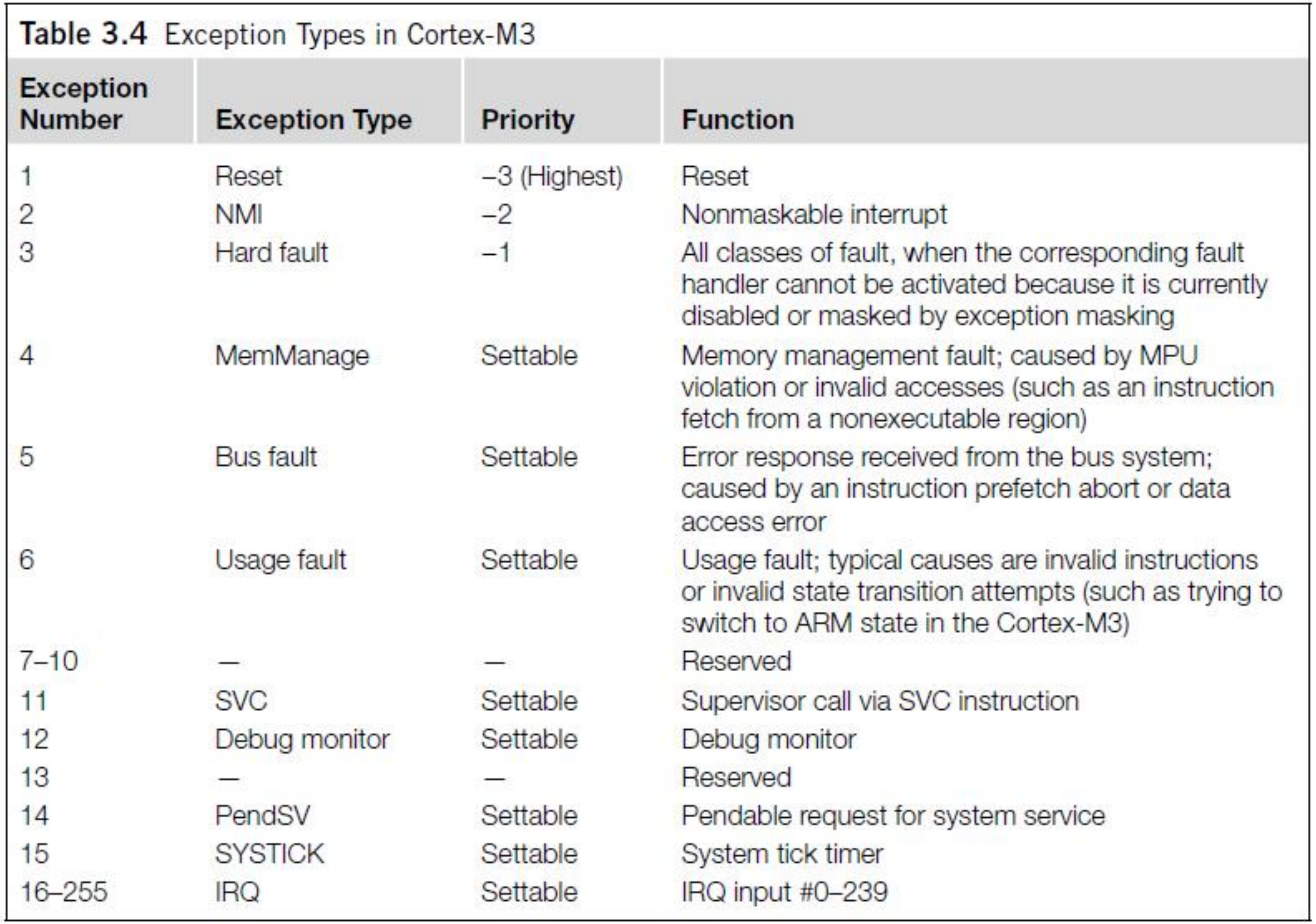


Table 3.4 Exception Types in Cortex-M3

Exception Number	Exception Type	Priority	Function
1	Reset	–3 (Highest)	Reset
2	NMI	–2	Nonmaskable interrupt
3	Hard fault	–1	All classes of fault, when the corresponding fault handler cannot be activated because it is currently disabled or masked by exception masking
4	MemManage	Settable	Memory management fault; caused by MPU violation or invalid accesses (such as an instruction fetch from a nonexecutable region)
5	Bus fault	Settable	Error response received from the bus system; caused by an instruction prefetch abort or data access error
6	Usage fault	Settable	Usage fault; typical causes are invalid instructions or invalid state transition attempts (such as trying to switch to ARM state in the Cortex-M3)
7–10	—	—	Reserved
11	SVC	Settable	Supervisor call via SVC instruction
12	Debug monitor	Settable	Debug monitor
13	—	—	Reserved
14	PendSV	Settable	Pendable request for system service
15	SYSTICK	Settable	System tick timer
16–255	IRQ	Settable	IRQ input #0–239

Vector Tables

- When an exception event takes place on the Cortex-M3 and is accepted by the processor core, the corresponding exception handler is executed.
- To determine the starting address of the exception handler, a vector table mechanism is used.
- The vector table is an array of word data inside the system memory, each representing the starting address of one exception type.
- The vector table is relocatable, and the relocation is controlled by a relocation register in the NVIC (see Table 3.5).
- After reset, this relocation control register is reset to 0; therefore, the vector table is located in address 0x0 after reset.

Vector Tables (continued)

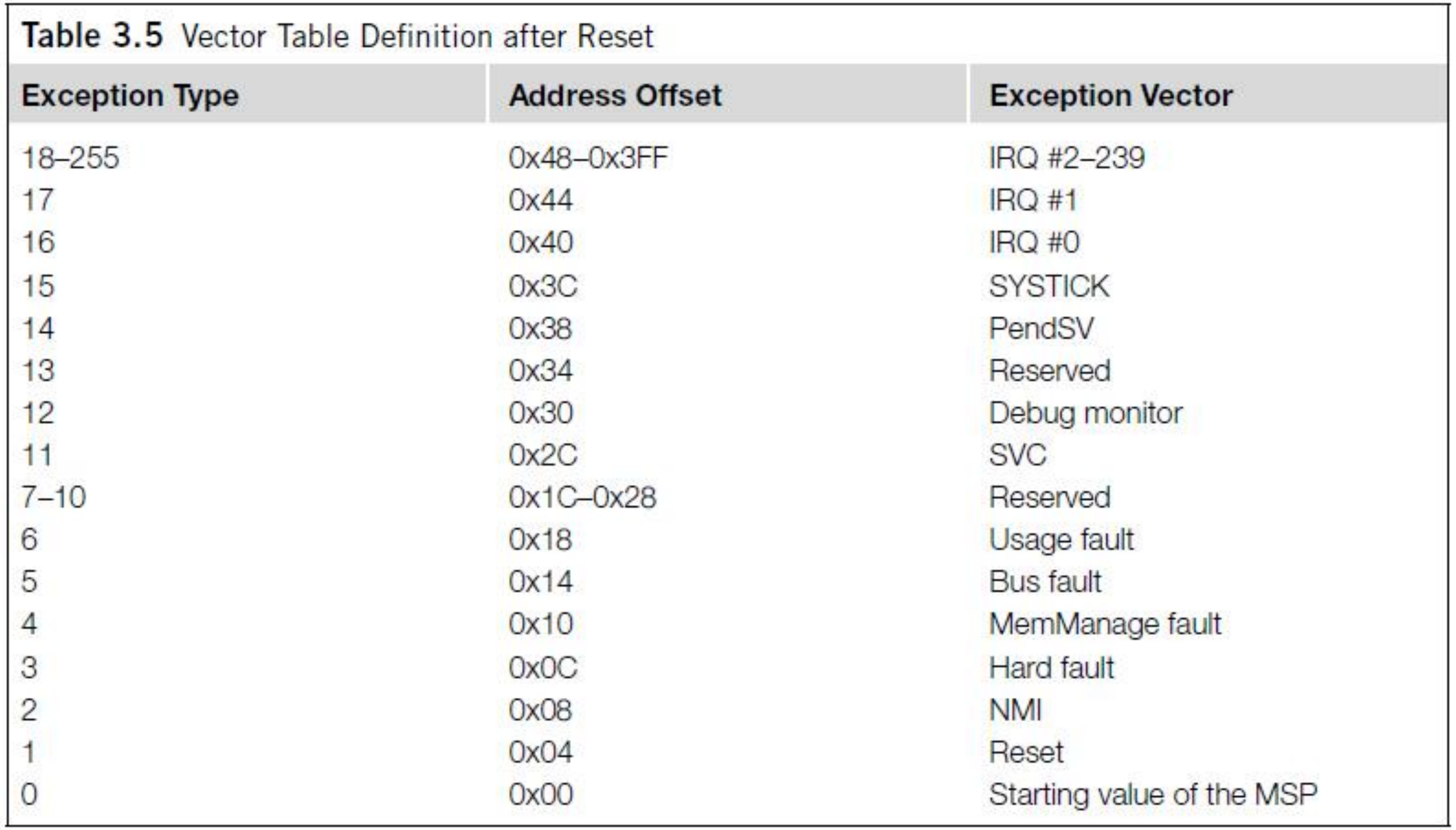


Table 3.5 Vector Table Definition after Reset

Exception Type	Address Offset	Exception Vector
18–255	0x48–0x3FF	IRQ #2–239
17	0x44	IRQ #1
16	0x40	IRQ #0
15	0x3C	SYSTICK
14	0x38	PendSV
13	0x34	Reserved
12	0x30	Debug monitor
11	0x2C	SVC
7–10	0x1C–0x28	Reserved
6	0x18	Usage fault
5	0x14	Bus fault
4	0x10	MemManage fault
3	0x0C	Hard fault
2	0x08	NMI
1	0x04	Reset
0	0x00	Starting value of the MSP

Vector Tables (continued)

- For example, if the reset is exception type 1, the address of the reset vector is 1 times 4 (each word is 4 bytes), which equals 0x00000004, and NMI vector (type 2) is located in 2 × 4 = 0x00000008.
- The address 0x00000000 is used to store the starting value for the MSP.
- The LSB of each exception vector indicates whether the exception is to be executed in the Thumb state.
- Because the Cortex-M3 can support only Thumb instructions, the LSB of all the exception vectors should be set to 1.

The Instruction Set

- The Cortex-M3 supports the Thumb-2 instruction set.
  - It allows 32-bit instructions and 16-bit instructions to be used together for high code density and high efficiency.
  - It is flexible and powerful yet easy to use.
- In previous ARM processors, the central processing unit (CPU) had two operation states – a 32-bit ARM state and a 16-bit Thumb state.
  - In the ARM state, the instructions are 32 bits and can execute all supported instructions with very high performance.
  - In the Thumb state, the instructions are 16 bits, so there is a much higher instruction code density
    - The Thumb state does not have all the functionality of ARM instructions and may require more instructions to complete certain types of operations.

The Instruction Set (continued)

- Many applications have mixed ARM and Thumb codes.
  - However, there is overhead (in terms of both execution time and instruction space) to switch between the states, and ARM and Thumb codes might need to be compiled separately in different files.
  - This increases the complexity of software development and reduces maximum efficiency of the CPU core.
- With the introduction of the Thumb-2 instruction set, it is now possible to handle all processing requirements in one operation state.
  - There is no need to switch between the two.
  - In fact, the Cortex-M3 does not support the ARM code.
  - Even interrupts are now handled with the Thumb state.

The Instruction Set (continued)

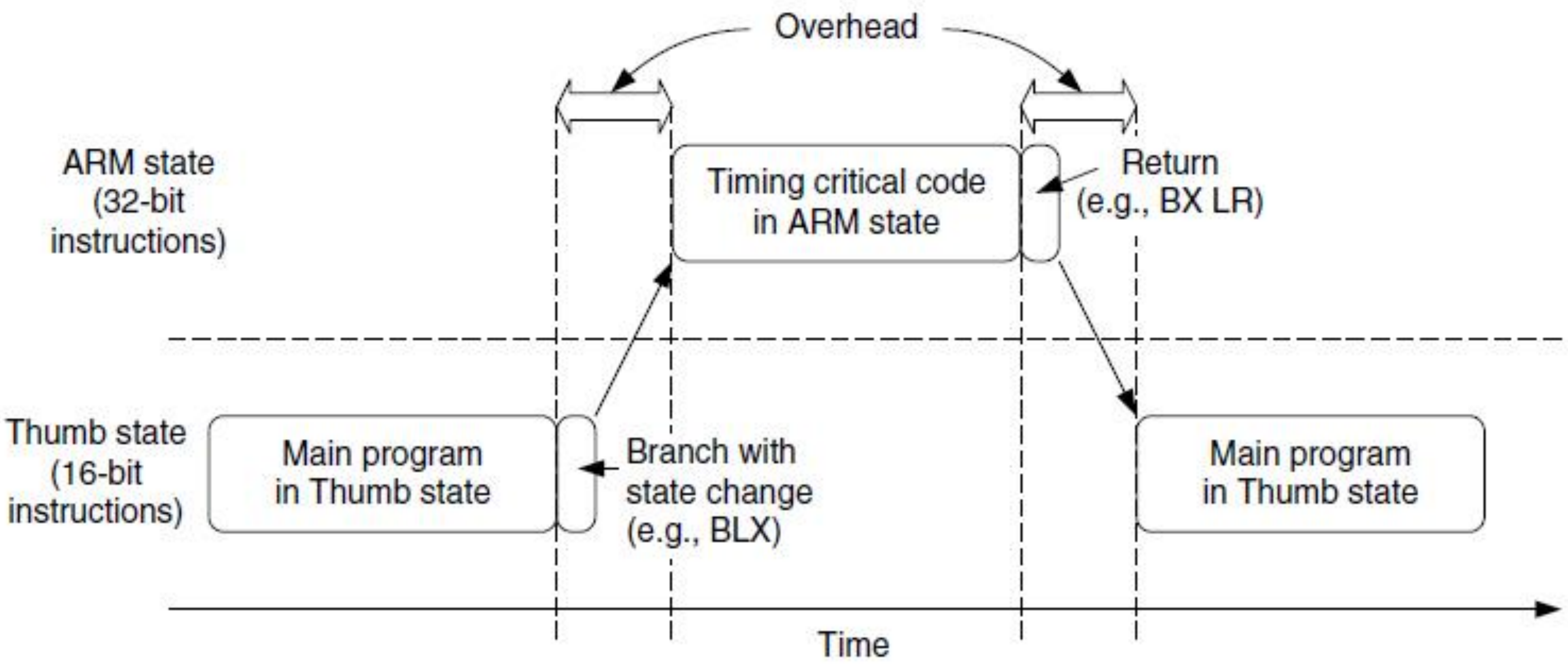


FIGURE 2.7

Switching between ARM Code and Thumb Code in Traditional ARM Processors Such as the ARM7.

The Instruction Set (continued)

- Since there is no need to switch between states, the Cortex-M3 processor has a number of advantages over traditional ARM processors, such as:
  - No state switching overhead, saving both execution time and instruction space
  - No need to separate ARM code and Thumb code source files, making software development and maintenance easier
  - It's easier to get the best efficiency and performance, in turn making it easier to write software, because there is no need to worry about switching code between ARM and Thumb to try to get the best density/performance

The Instruction Set (continued)

- The Cortex-M3 processor has a number of interesting and powerful instructions. Here are a few examples:
  - UFBX, BFI, and BFC: Bit field extract, insert, and clear instructions
  - UDIV and SDIV: Unsigned and signed divide instructions
  - WFE, WFI, and SEV: Wait-For-Event, Wait-For-Interrupts, and Send-Event; these allow the processor to enter sleep mode and to handle task synchronization on multiprocessor systems
  - MSR and MRS: Move to special register from general-purpose register and move special register to general-purpose register; for access to the special registers

Debugging Support

- The Cortex-M3 processor includes a number of debugging features, such as program execution controls, including halting and stepping, instruction breakpoints, data watchpoints, registers and memory accesses, profiling, and traces.
- The debugging hardware of the Cortex-M3 processor is based on the CoreSight architecture.
  - Unlike traditional ARM processors, the CPU core itself does not have a Joint Test Action Group (JTAG) interface.
  - Instead, a debug interface module is decoupled from the core, and a bus interface called the Debug Access Port (DAP) is provided at the core level.
  - Through this bus interface, external debuggers can access control registers to debug hardware as well as system memory, even when the processor is running.

Debugging Support (continued)

- The control of DAP bus interface is carried out by a Debug Port (DP) device.
- The DPs currently available are the Serial-Wire JTAG Debug Port (SWJ-DP) (supports the traditional JTAG protocol as well as the Serial-Wire protocol) or the SW-DP (supports the Serial-Wire protocol only).
- A JTAG-DP module from the ARM CoreSight product family can also be used.
- Chip manufacturers can choose to attach one of these DP modules to provide the debug interface.
- Chip manufacturers can also include an Embedded Trace Macrocell (ETM) to allow instruction trace.
  - Trace information is output via the Trace Port Interface Unit (TPIU), and the debug host (usually a Personal Computer [PC]) can then collect the executed instruction information via external trace-capturing hardware.

Debugging Support (continued)

- Within the Cortex-M3 processor, a number of events can be used to trigger debug actions.
  - Debug events can be breakpoints, watchpoints, fault conditions, or external debugging request input signals.
- When a debug event takes place, the Cortex-M3 processor can either enter halt mode or execute the debug monitor exception handler.
- The data watchpoint function is provided by a Data Watchpoint and Trace (DWT) unit in the Cortex-M3 processor.
  - This can be used to stop the processor (or trigger the debug monitor exception routine) or to generate data trace information.
  - When data trace is used, the traced data can be output via the TPIU.

Debugging Support (continued)

- In addition to these basic debugging features, the Cortex-M3 processor also provides a Flash Patch and Breakpoint (FPB) unit that can provide a simple breakpoint function or remap an instruction access from Flash to a different location in SRAM.
- An Instrumentation Trace Macrocell (ITM) provides a new way for developers to output data to a debugger.
  - By writing data to register memory in the ITM, a debugger can collect the data via a trace interface and display or process them.
  - This method is easy to use and faster than JTAG output.
- All these debugging components are controlled via the DAP interface bus on the Cortex-M3 or by a program running on the processor core, and all trace information is accessible from the TPIU.

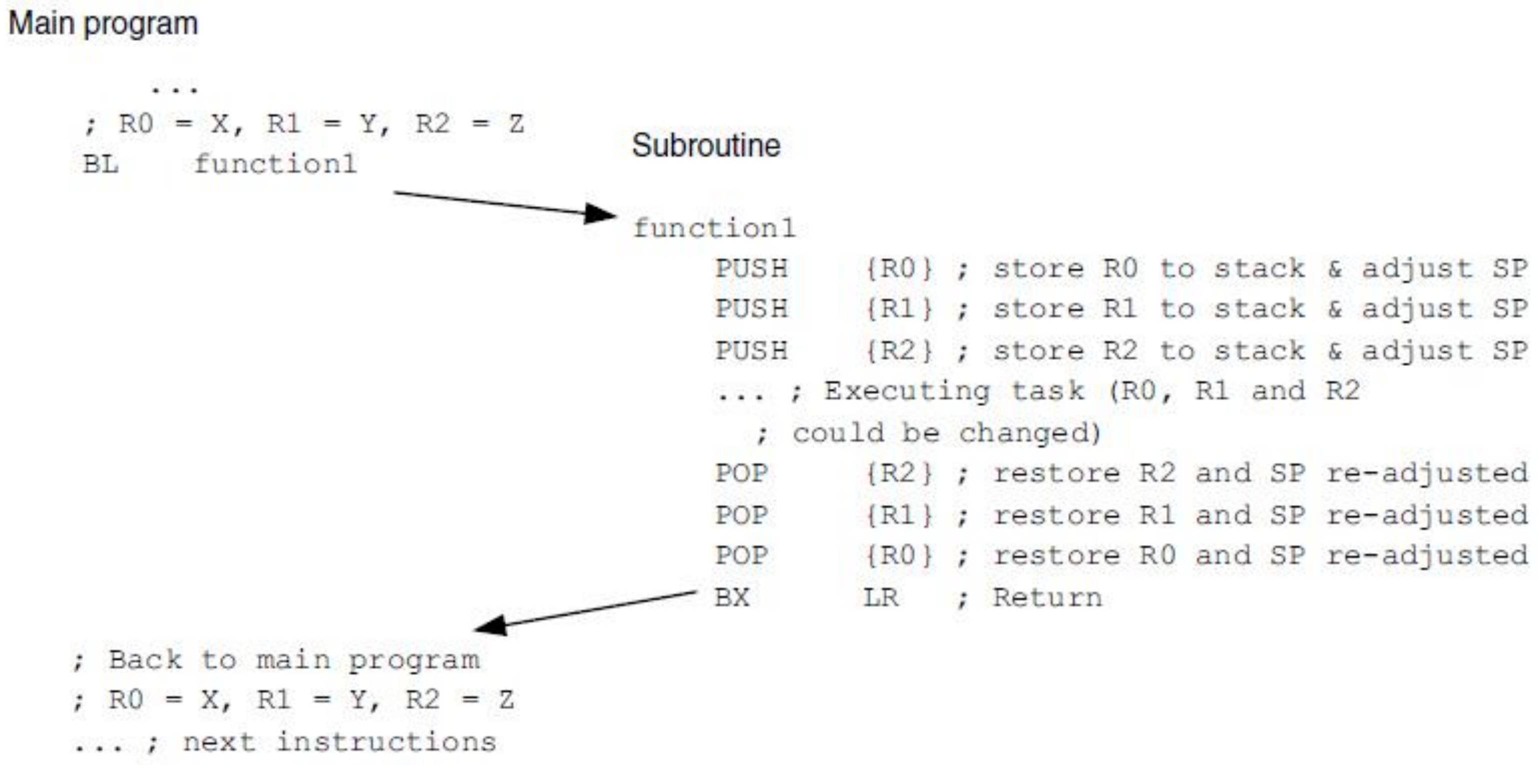
Stack Memory Operations

- In the Cortex-M3, besides normal software-controlled stack PUSH and POP, the stack PUSH and POP operations are also carried out automatically when entering or exiting an exception/interrupt handler.

Basic Operations of the Stack

- In general, stack operations are memory write or read operations, with the address specified by an SP.
- Data in registers is saved into stack memory by a PUSH operation and can be restored to registers later by a POP operation.
- The SP is adjusted automatically in PUSH and POP so that multiple data PUSH will not cause old stacked data to be erased.
- The function of the stack is to store register contents in memory so that they can be restored later, after a processing task is completed.
- For normal uses, for each store (PUSH), there must be a corresponding read (POP), and the address of the POP operation should match that of the PUSH operation.
- When PUSH/POP instructions are used, the SP is incremented/decremented automatically.

Basic Operations of the Stack (continued)



Main program

```
    ...
; R0 = X, R1 = Y, R2 = Z
BL    function1
```

Subroutine

```
function1
    PUSH    {R0} ; store R0 to stack & adjust SP
    PUSH    {R1} ; store R1 to stack & adjust SP
    PUSH    {R2} ; store R2 to stack & adjust SP
    ... ; Executing task (R0, R1 and R2
      ; could be changed)
    POP     {R2} ; restore R2 and SP re-adjusted
    POP     {R1} ; restore R1 and SP re-adjusted
    POP     {R0} ; restore R0 and SP re-adjusted
    BX      LR   ; Return
```

```
; Back to main program
; R0 = X, R1 = Y, R2 = Z
... ; next instructions
```

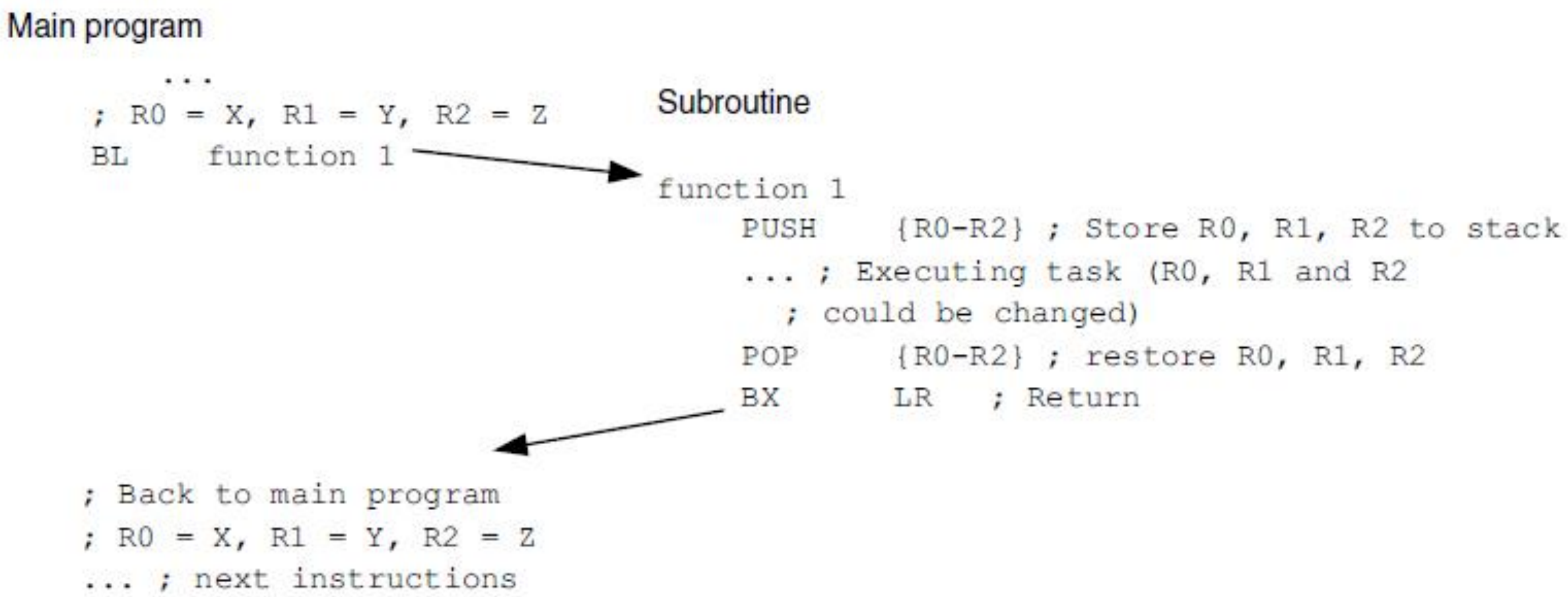
FIGURE 3.11

Stack Operation Basics: One Register in Each Stack Operation.

Basic Operations of the Stack (continued)

- When program control returns to the main program, the R0 – R2 contents are the same as before.
- Notice the order of PUSH and POP: The POP order must be the reverse of PUSH.
- These operations can be simplified, thanks to PUSH and POP instructions allowing multiple load and store.
- In this case, the ordering of a register POP is automatically reversed by the processor.

Basic Operations of the Stack (continued)



Main program

```
    ...
; R0 = X, R1 = Y, R2 = Z
BL    function 1
```

Subroutine

```
function 1
     PUSH    {R0-R2} ; Store R0, R1, R2 to stack
     ... ; Executing task (R0, R1 and R2
       ; could be changed)
     POP     {R0-R2} ; restore R0, R1, R2
     BX      LR    ; Return
```

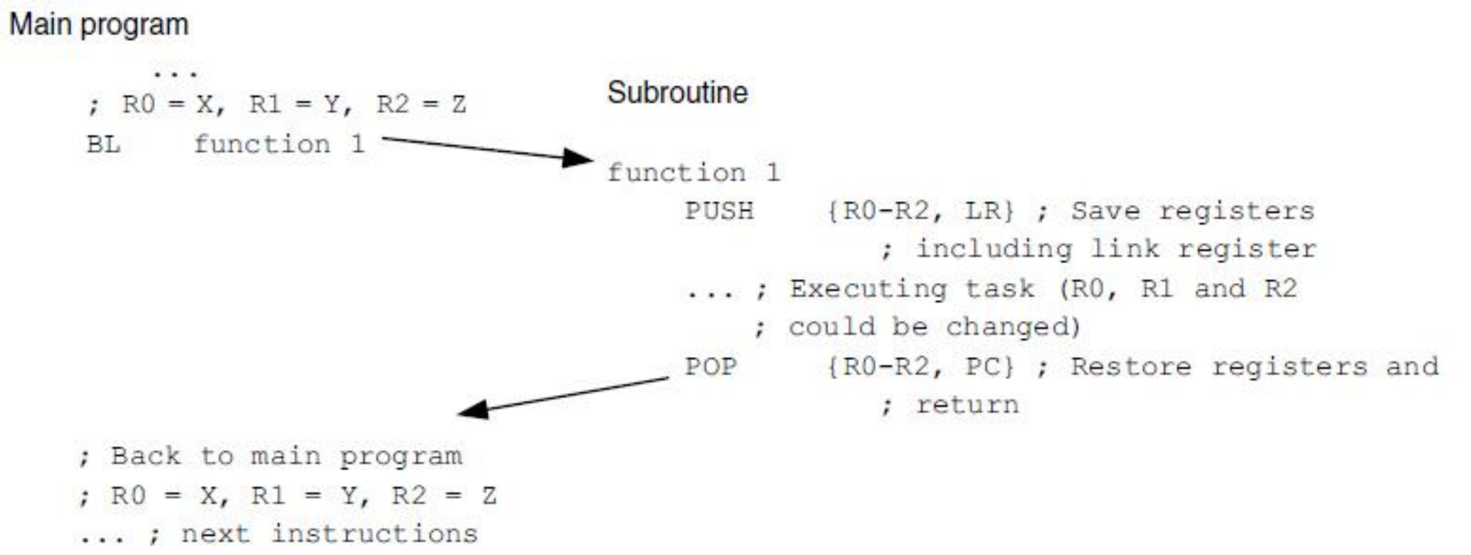
```
; Back to main program
; R0 = X, R1 = Y, R2 = Z
... ; next instructions
```

FIGURE 3.12

Stack Operation Basics: Multiple Register Stack Operation.

Basic Operations of the Stack (continued)

- We can also combine RETURN with a POP operation.
  - This is done by pushing the LR to the stack and popping it back to PC at the end of the subroutine.



Main program

```
       ...
    ; R0 = X, R1 = Y, R2 = Z
    BL     function 1
```

Subroutine

```
function 1
    PUSH    {R0-R2, LR} ; Save registers
                        ; including link register
    ... ; Executing task (R0, R1 and R2
        ; could be changed)
    POP     {R0-R2, PC} ; Restore registers and
                        ; return
```

```
; Back to main program
; R0 = X, R1 = Y, R2 = Z
... ; next instructions
```

FIGURE 3.13

Stack Operation Basics: Combining Stack POP and RETURN.

Cortex-M3 Stack Implementation

- The Cortex-M3 uses a full-descending stack operation model.
- The SP points to the last data pushed to the stack memory, and the SP decrements before a new PUSH operation.
- Figure 3.14 shows execution of the instruction PUSH {R0}.

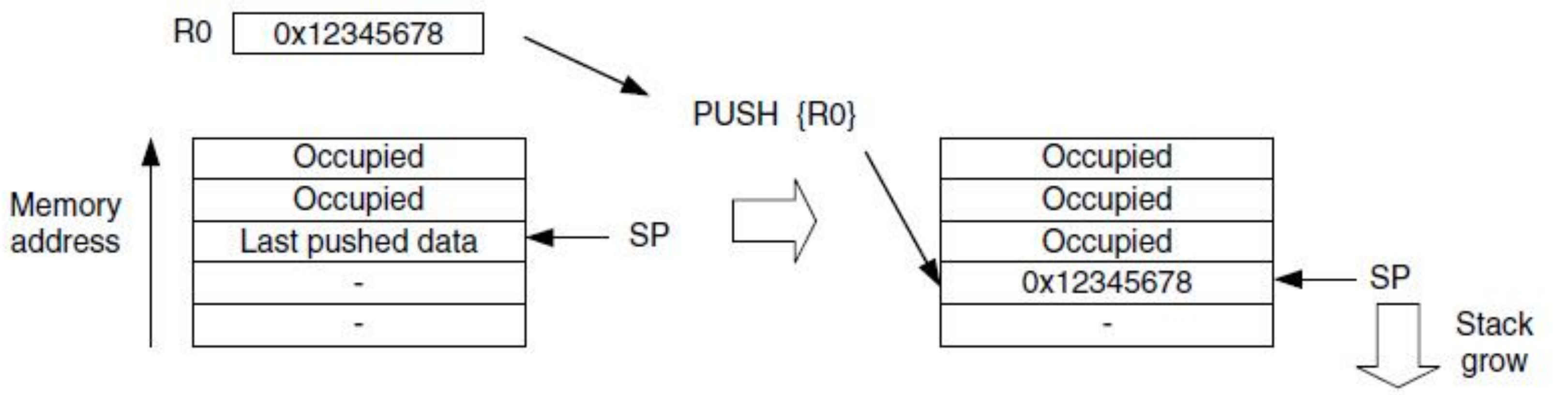


FIGURE 3.14

Cortex-M3 Stack PUSH Implementation.

Cortex-M3 Stack Implementation (continued)

- For POP operations, the data is read from the memory location pointed by SP, and then, the SP is incremented.
- The contents in the memory location are unchanged but will be overwritten when the next PUSH operation takes place.

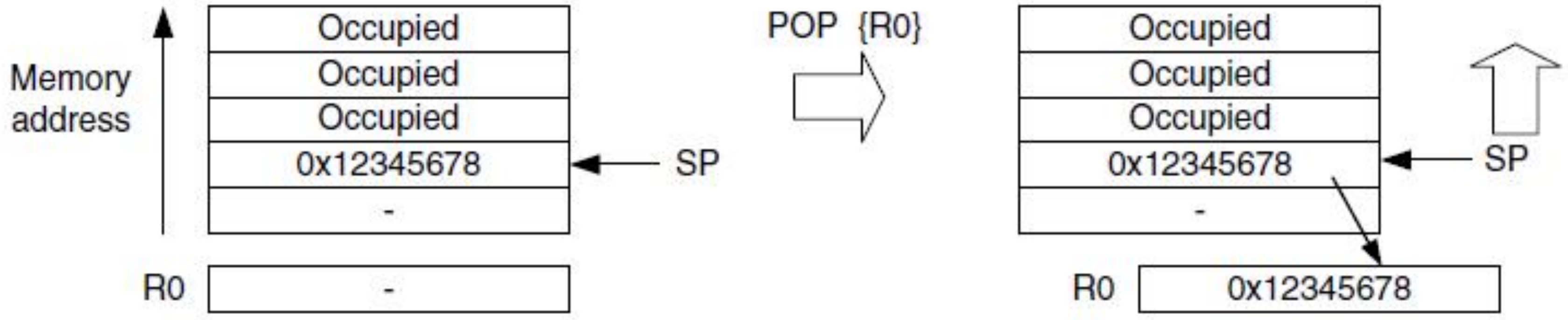


FIGURE 3.15

Cortex-M3 Stack POP Implementation.

Cortex-M3 Stack Implementation (continued)

- Because each PUSH/POP operation transfers 4 bytes of data (each register contains 1 word, or 4 bytes), the SP decrements/increments by 4 at a time or a multiple of 4 if more than 1 register is pushed or popped.
- In the Cortex-M3, R13 is defined as the SP.
  - When an interrupt takes place, a number of registers will be pushed automatically, and R13 will be used as the SP for this stacking process.
- Similarly, the pushed registers will be restored/popped automatically when exiting an interrupt handler, and the SP will also be adjusted.

The Two-Stack Model in the Cortex-M3

- The Cortex-M3 has two SPs: the MSP and the PSP.
- The SP register to be used is controlled by the control register bit 1 (CONTROL[1]).
- When CONTROL[1] is 0, the MSP is used for both thread mode and handler mode.
    - In this arrangement, the main program and the exception handlers share the same stack memory region.
    - This is the default setting after power-up.

The Two-Stack Model in the Cortex-M3 (continued)

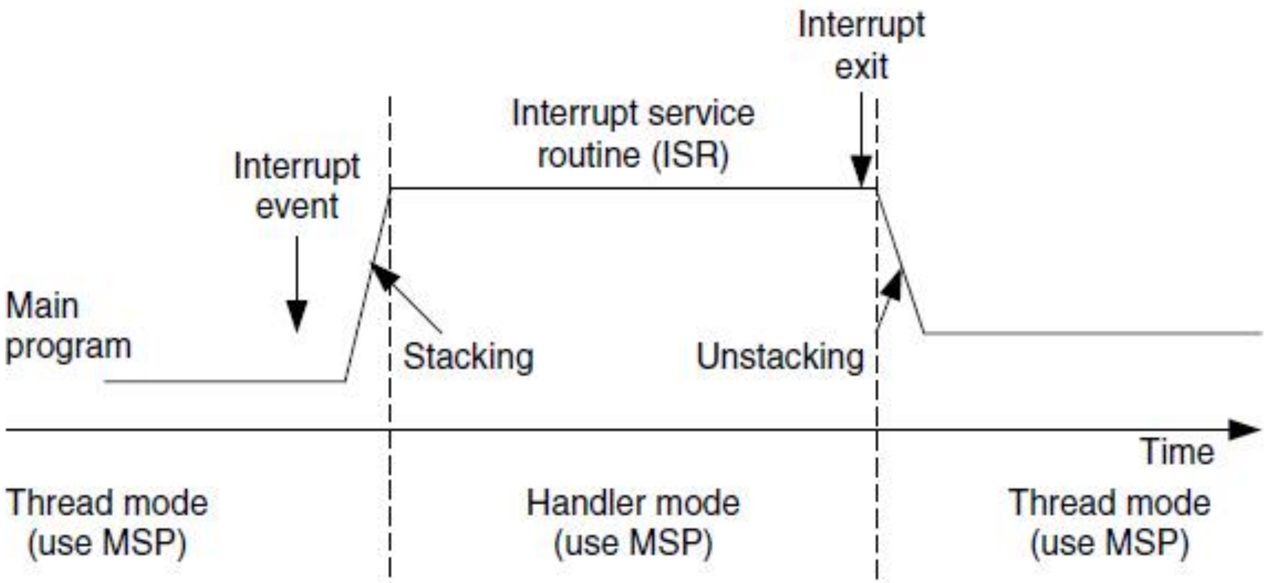


FIGURE 3.16

CONTROL[1]□0: Both Thread Level and Handler Use Main Stack.

The Two-Stack Model in the Cortex-M3 (continued)

- When the CONTROL[1] is 1, the PSP is used in thread mode.
  - In this arrangement, the main program and the exception handler can have separate stack memory regions.
  - This can prevent a stack error in a user application from damaging the stack used by the OS.
  - The automatic stacking and unstacking mechanism will use PSP, whereas stack operations inside the handler will use MSP.

The Two-Stack Model in the Cortex-M3 (continued)

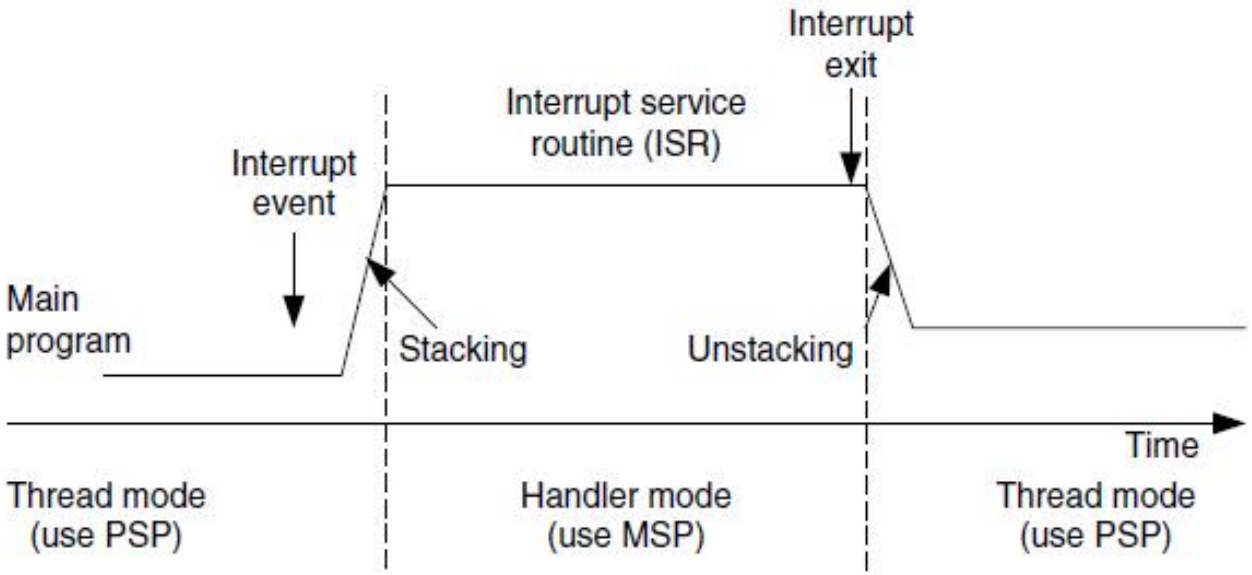
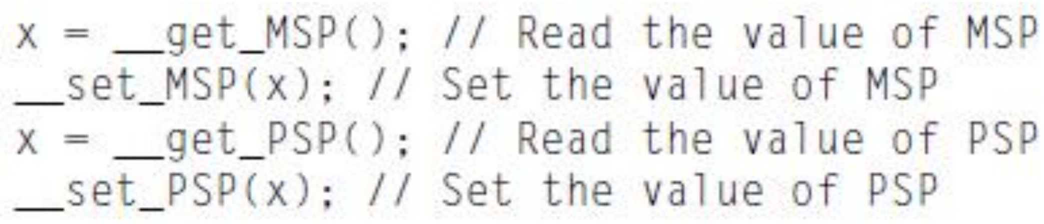


FIGURE 3.17

CONTROL[1]=1: Thread Level Uses Process Stack and Handler Uses Main Stack.

The Two-Stack Model in the Cortex-M3 (continued)

- It is possible to perform read/write operations directly to the MSP and PSP, without any confusion of which R13 you are referring to.
- Provided that you are in privileged level, you can access MSP and PSP values:

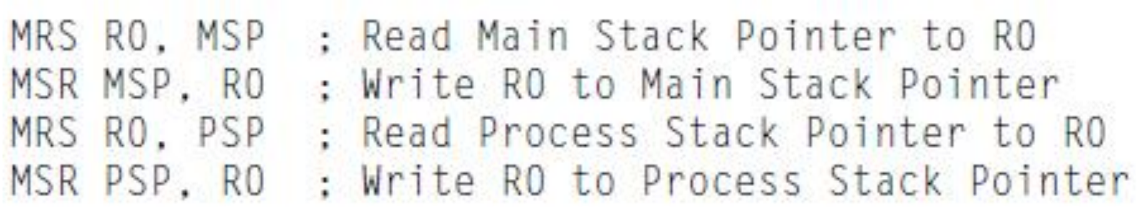


```
x = __get_MSP(); // Read the value of MSP
__set_MSP(x); // Set the value of MSP
x = __get_PSP(); // Read the value of PSP
__set_PSP(x); // Set the value of PSP
```

- In general, it is not recommended to change current selected SP values in a C function, as the stack memory could be used for storing local variables.

The Two-Stack Model in the Cortex-M3 (continued)

- To access the SPs in assembly, you can use the MRS and MSR instructions:



```
MRS R0, MSP   ; Read Main Stack Pointer to R0
MSR MSP, R0   ; Write R0 to Main Stack Pointer
MRS R0, PSP   ; Read Process Stack Pointer to R0
MSR PSP, R0   ; Write R0 to Process Stack Pointer
```

- By reading the PSP value using an MRS instruction, the OS can read data stacked by the user application (such as register contents before SVC).
- In addition, the OS can change the PSP pointer value—for example, during context switching in multitasking systems.

Reset Sequence

- After the processor exits reset, it will read two words from memory
  - Address 0x00000000: Starting value of R13 (the SP)
  - Address 0x00000004: Reset vector (the starting address of program execution; LSB should be set to 1 to indicate Thumb state)

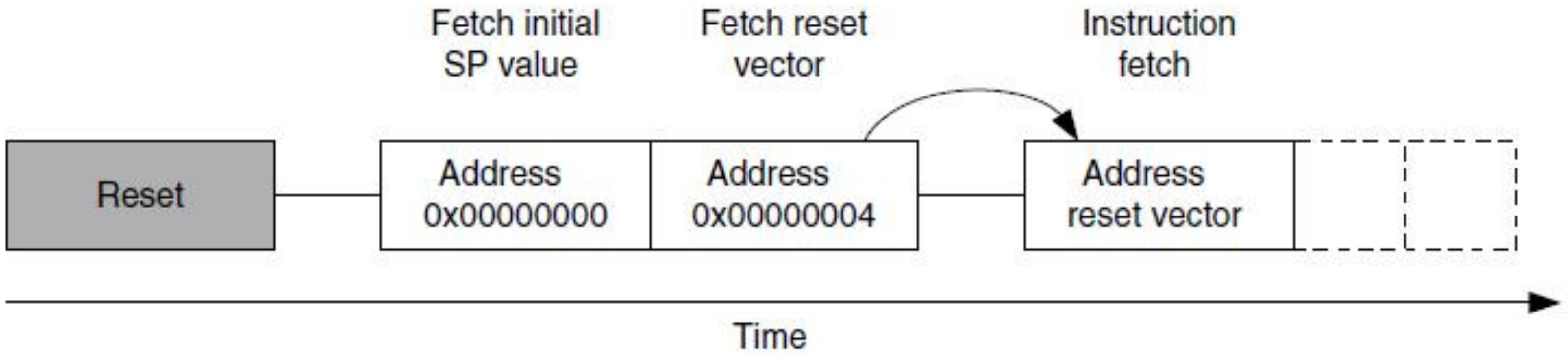


FIGURE 3.18

Reset Sequence.

Reset Sequence (continued)

- Because the stack operation in the Cortex-M3 is a full descending stack (SP decrement before store), the initial SP value should be set to the first memory after the top of the stack region.
- For example, if you have a stack memory range from 0x20007C00 to 0x20007FFF (1 KB), the initial stack value should be set to 0x20008000.

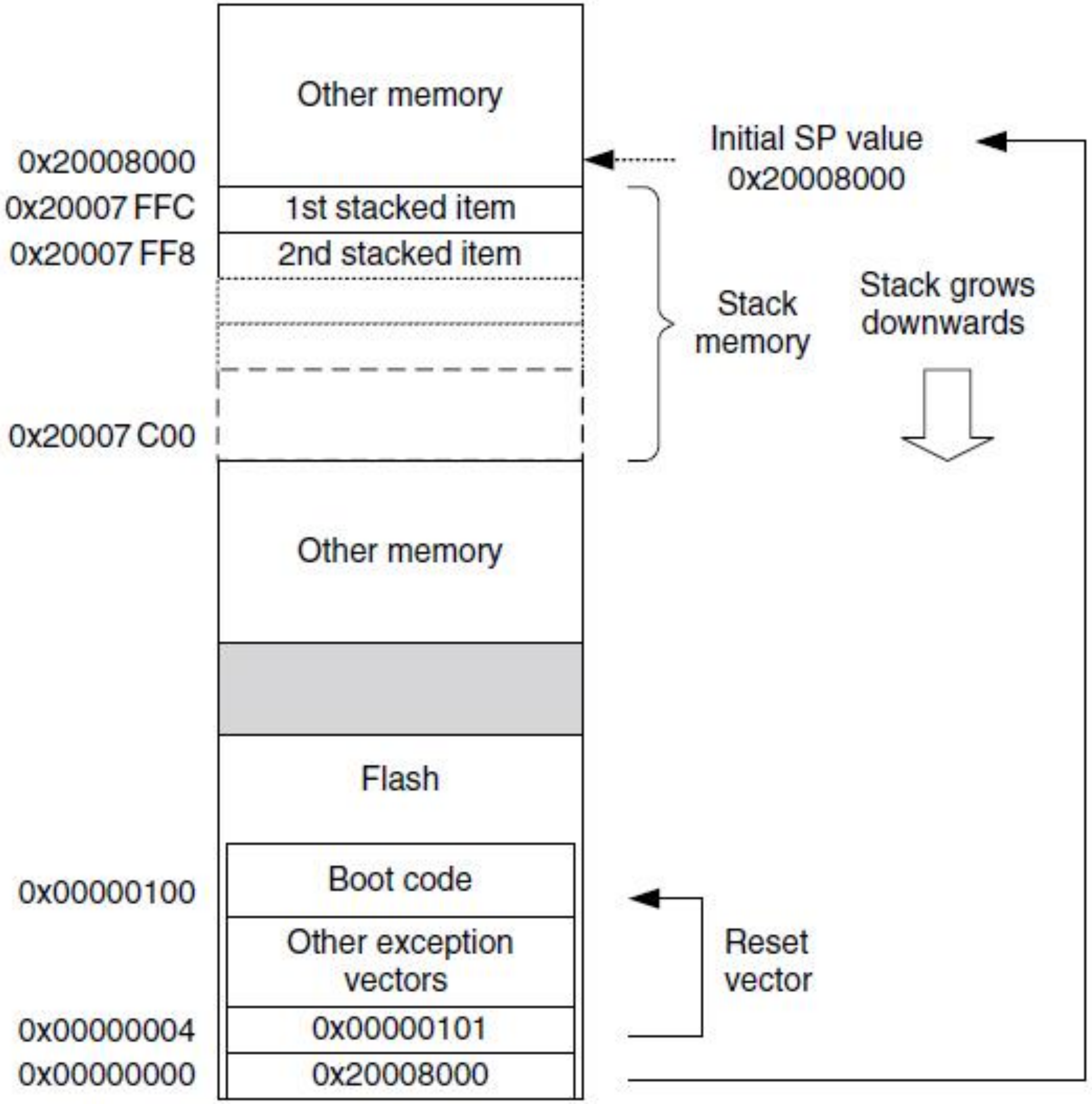


FIGURE 3.19

Initial Stack Pointer Value and Initial Program Counter Value Example.

Reset Sequence (continued)

- The vector table starts after the initial SP value.
- The first vector is the reset vector.
- In the Cortex-M3, vector addresses in the vector table should have their LSB set to 1 to indicate that they are Thumb code.
- For that reason, the previous example has 0x101 in the reset vector, whereas the boot code starts at address 0x100 (see Figure 3.19).
- After the reset vector is fetched, the Cortex-M3 can then start to execute the program from the reset vector address and begin normal operations.
- It is necessary to have the SP initialized, because some of the exceptions (such as NMI) can happen right after reset, and the stack memory could be required for the handler of those exceptions.

Low Power and High Energy Efficiency

- The Cortex-M3 processor is designed with various features to allow designers to develop low power and high energy efficient products.
  - It has sleep mode and deep sleep mode supports, which can work with various system-design methodologies to reduce power consumption during idle period.
  - Its low gate count and design techniques reduce circuit activities in the processor to allow active power to be reduced.
  - It has high code density and hence it has lowered the program size requirement.
  - It allows processing tasks to be completed in a short time, so that the processor can return to sleep modes as soon as possible to cut down energy use.
- Starting from Cortex-M3 revision 2, a new feature called Wakeup Interrupt Controller (WIC) is available.
  - This feature allows the whole processor core to be powered down, while processor states are retained and the processor can be returned to active state almost immediately when an interrupt takes place.
  - This makes the Cortex-M3 even more suitable for many ultra-low power applications

Characteristics Summary

- High Performance
  - The Cortex-M3 processor delivers high performance in microcontroller products:
    - Many instructions are single cycle
    - Separate data and instruction buses
    - No state switching overhead
    - The Thumb-2 instruction set provides extra flexibility in programming
    - Instruction fetches are 32 bits
    - Operate at high clock frequency (over 100 MHz)

Characteristics Summary (continued)

- Advanced Interrupt-Handling Features
  - The interrupt features on the Cortex-M3 processor are easy to use, very flexible, and provide high interrupt processing throughput:
    - The built-in NVIC supports up to 240 external interrupt inputs
    - It reduces the interrupt handling latency
    - Interrupt arrangement is extremely flexible
    - A minimum of eight levels of priority are supported, and the priority can be changed dynamically.
    - Some of the multicycle operations are now interruptible
    - Immediate execution of the NMI handler is guaranteed on receipt of NMI request

Characteristics Summary (continued)

- Low Power Consumption
  - The Cortex-M3 processor is suitable for various low-power applications:
    - Suitable for low-power designs because of the low gate count.
    - It has power-saving mode support (SLEEPING and SLEEPDEEP).
      - The processor can enter sleep mode using WFI or WFE instructions.
      - The design has separated clocks for essential blocks, so clocking circuits for most parts of the processor can be stopped during sleep.
    - The fully static, synchronous, synthesizable design makes the processor easy to be manufactured using any low power or standard semiconductor process technology.

Characteristics Summary (continued)

- System Features
  - The Cortex-M3 processor provides various system features making it suitable for a large number of applications:
    - The system provides bit-band operation, byte-invariant big endian mode, and unaligned data access support.
    - Advanced fault-handling features include various exception types and fault status registers, making it easier to locate problems.
    - With the shadowed stack pointer, stack memory of kernel and user processes can be isolated.
    - With the optional MPU, the processor is more than sufficient to develop robust software and reliable products.

Characteristics Summary (continued)

- Debug Supports
  - The Cortex-M3 processor includes comprehensive debug features to help software developers design their products:
    - Supports JTAG or Serial-Wire debug interfaces
    - Based on the CoreSight debugging solution, processor status or memory contents can be accessed even when the core is running
    - Built-in support for six breakpoints and four watchpoints
    - Optional ETM for instruction trace and data trace using DWT
    - New debugging features, including fault status registers, new fault exceptions, and Flash Patch operations, make debugging much easier
    - ITM provides an easy-to-use method to output debug information from test code
    - PC sampler and counters inside the DWT provide code-profiling information

References

1. Joseph Yiu, ***“The Definitive Guide to the ARM Cortex-M3”***, 2nd Edition, Newnes (Elsevier), 2010.
2. https://www.arm.com