ENEL 351 W2022

Week 9
“Chapter 6 Memory System”

6.1 Overview of memory system features

- The processor and the architecture allow **very flexible memory configurations**
- You can find Cortex-M3 and Cortex-M4 based microcontrollers with **many different memory sizes and memory maps**
- You can also find **device-specific memory system features** in some Cortex-M microcontroller products, such as memory address remapping/alias.

6.1 Overview of memory system features

- The Cortex-M processors have **32-bit memory addressing and therefore have 4GB memory space**
- The memory space is **unified**, which means instructions and data share the same address space
- The 4GB memory space is **architecturally divided** into a number of regions, specified by the memory map
- Multiple bus interfaces to allow **concurrent instructions and data accesses** (Harvard bus architecture)

6.1 Overview of memory system features

- Bus interface designs based on **AMBA (Advanced Microcontroller Bus Architecture)**, a de facto on-chip bus standard: **AHB (AMBA High-performance Bus) Lite** protocol for pipelined operations in memory and system bus, and **APB (Advanced Peripheral Bus)** protocol for communication to peripheral and debug components
- Support for **unaligned data transfers**
- Support **exclusive accesses** (for semaphore operations in systems with an embedded OS or RTOS)

6.1 Overview of memory system features

- **Bit addressable memory spaces** (bit-band)
- **Memory attributes and access permissions** for different memory regions
- An **optional Memory Protection Unit (MPU)**. Memory attributes and access permission configurations can be programmed at runtime if the MPU is available.

6.2 Memory map

- In the 4GB addressable memory space, **some parts are allocated to internal peripherals**
- The memory locations of internal components are fixed
- Architecture allows:
  - The processor design to **support different types of memories and devices out of the box**
  - Optimized arrangement for **higher performance**
  - **High flexibility** to allow silicon designers to design their products with different memories and peripherals **for better product differentiation**

6.2 Memory map

- Possible to **store and execute** program code **in SRAM and RAM regions**, however the processor design is **not optimized** for such operation and requires one extra clock cycle per instruction for each instruction fetch
- Performance is **slightly slower** when executing program code **though the system bus**.
- **Program execution from Peripherals, Devices, and System memory regions is not allowed**

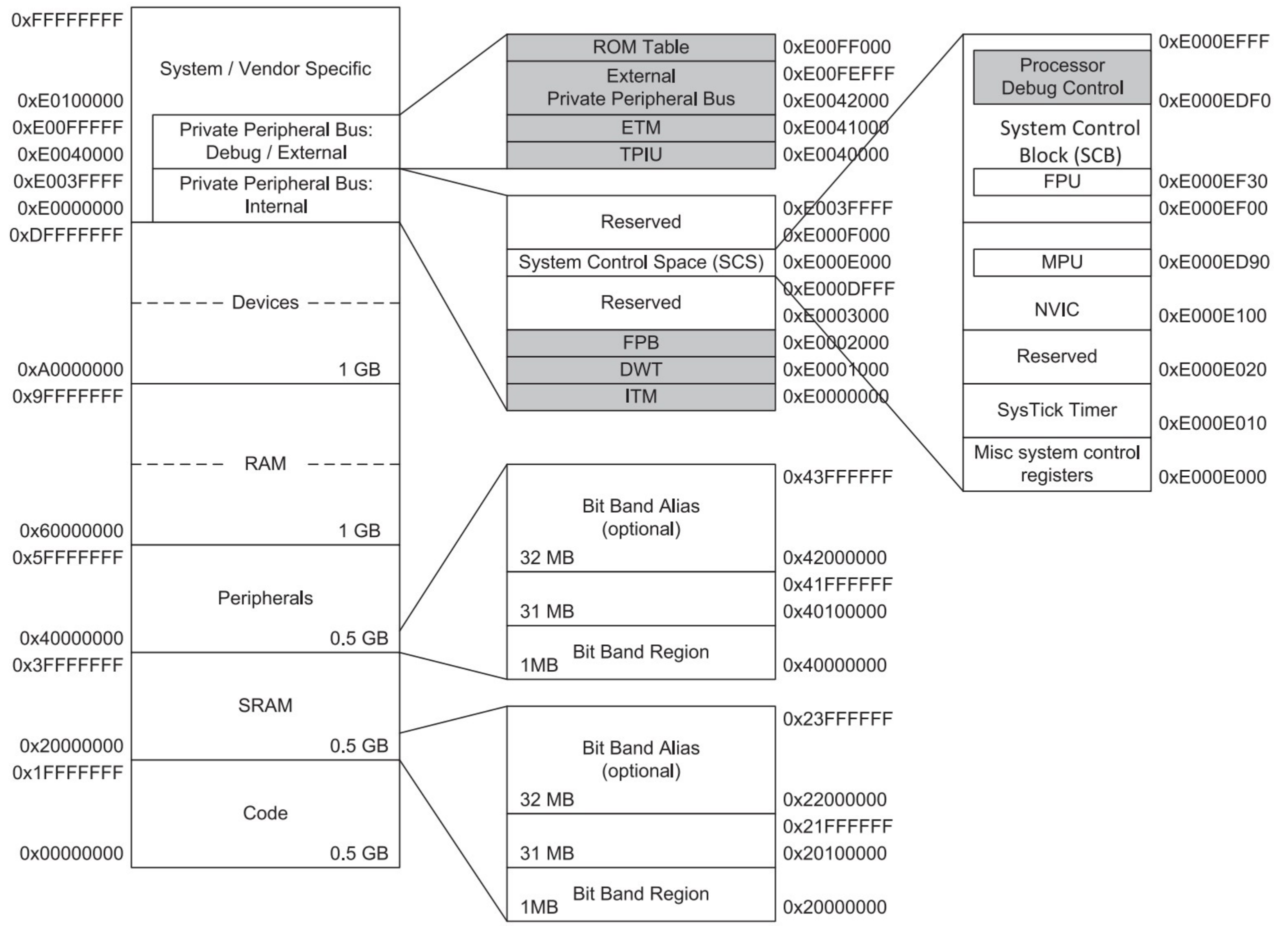


FIGURE 6.1

Pre-defined memory map of the Cortex®-M3 and Cortex-M4 processors (shaded areas are components for debug purpose)

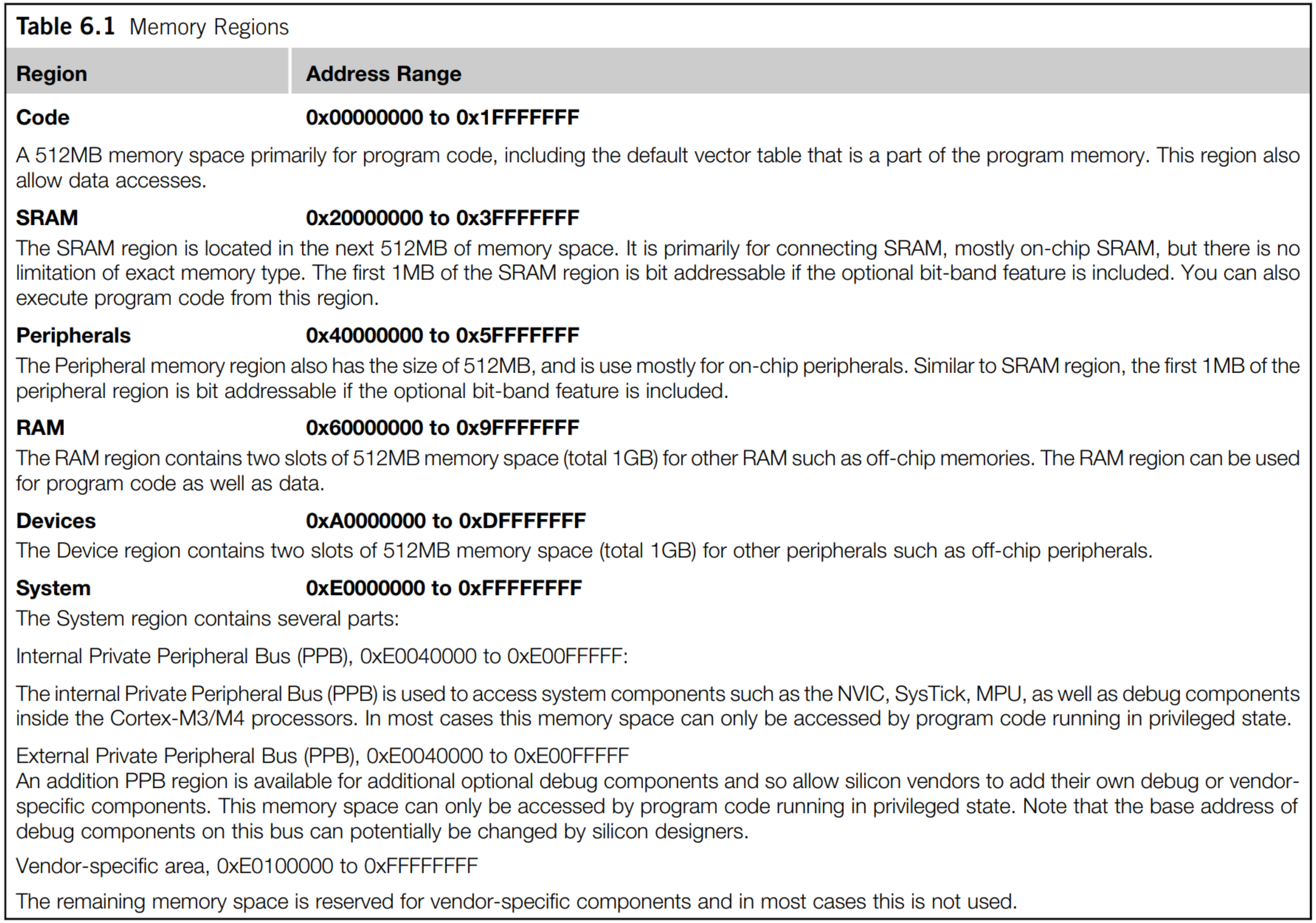


Table 6.1 Memory Regions

Region	Address Range
Code	**0x00000000 to 0x1FFFFFFF**
A 512MB memory space primarily for program code, including the default vector table that is a part of the program memory. This region also allow data accesses.	
SRAM	**0x20000000 to 0x3FFFFFFF**
The SRAM region is located in the next 512MB of memory space. It is primarily for connecting SRAM, mostly on-chip SRAM, but there is no limitation of exact memory type. The first 1MB of the SRAM region is bit addressable if the optional bit-band feature is included. You can also execute program code from this region.	
Peripherals	**0x40000000 to 0x5FFFFFFF**
The Peripheral memory region also has the size of 512MB, and is use mostly for on-chip peripherals. Similar to SRAM region, the first 1MB of the peripheral region is bit addressable if the optional bit-band feature is included.	
RAM	**0x60000000 to 0x9FFFFFFF**
The RAM region contains two slots of 512MB memory space (total 1GB) for other RAM such as off-chip memories. The RAM region can be used for program code as well as data.	
Devices	**0xA0000000 to 0xDFFFFFFF**
The Device region contains two slots of 512MB memory space (total 1GB) for other peripherals such as off-chip peripherals.	
System	**0xE0000000 to 0xFFFFFFFF**
The System region contains several parts:	
Internal Private Peripheral Bus (PPB), 0xE0040000 to 0xE00FFFFF:	
The internal Private Peripheral Bus (PPB) is used to access system components such as the NVIC, SysTick, MPU, as well as debug components inside the Cortex-M3/M4 processors. In most cases this memory space can only be accessed by program code running in privileged state.	
External Private Peripheral Bus (PPB), 0xE0040000 to 0xE00FFFFF An addition PPB region is available for additional optional debug components and so allow silicon vendors to add their own debug or vendor-specific components. This memory space can only be accessed by program code running in privileged state. Note that the base address of debug components on this bus can potentially be changed by silicon designers.	
Vendor-specific area, 0xE0100000 to 0xFFFFFFFF	
The remaining memory space is reserved for vendor-specific components and in most cases this is not used.	

6.3 Connecting the processor to memory and peripherals

- Generic bus interfaces based on **AMBA (Advanced Microcontroller Bus Architecture)**
- The AMBA specification **supports several bus protocols** in the Cortex-M3 and Cortex-M4 processors
  - The **AHB (AMBA High-performance Bus) Lite protocol is used for the main bus interfaces**
  - The **APB (Advanced Peripheral Bus) Lite** protocol is used for connecting peripherals and the **Private Peripheral Bus (PPB)**, which is mainly used for **debug components**, to the internal bus system
- **Additional bus segments** based on APB can be added onto the system bus by using **additional bus bridge components**

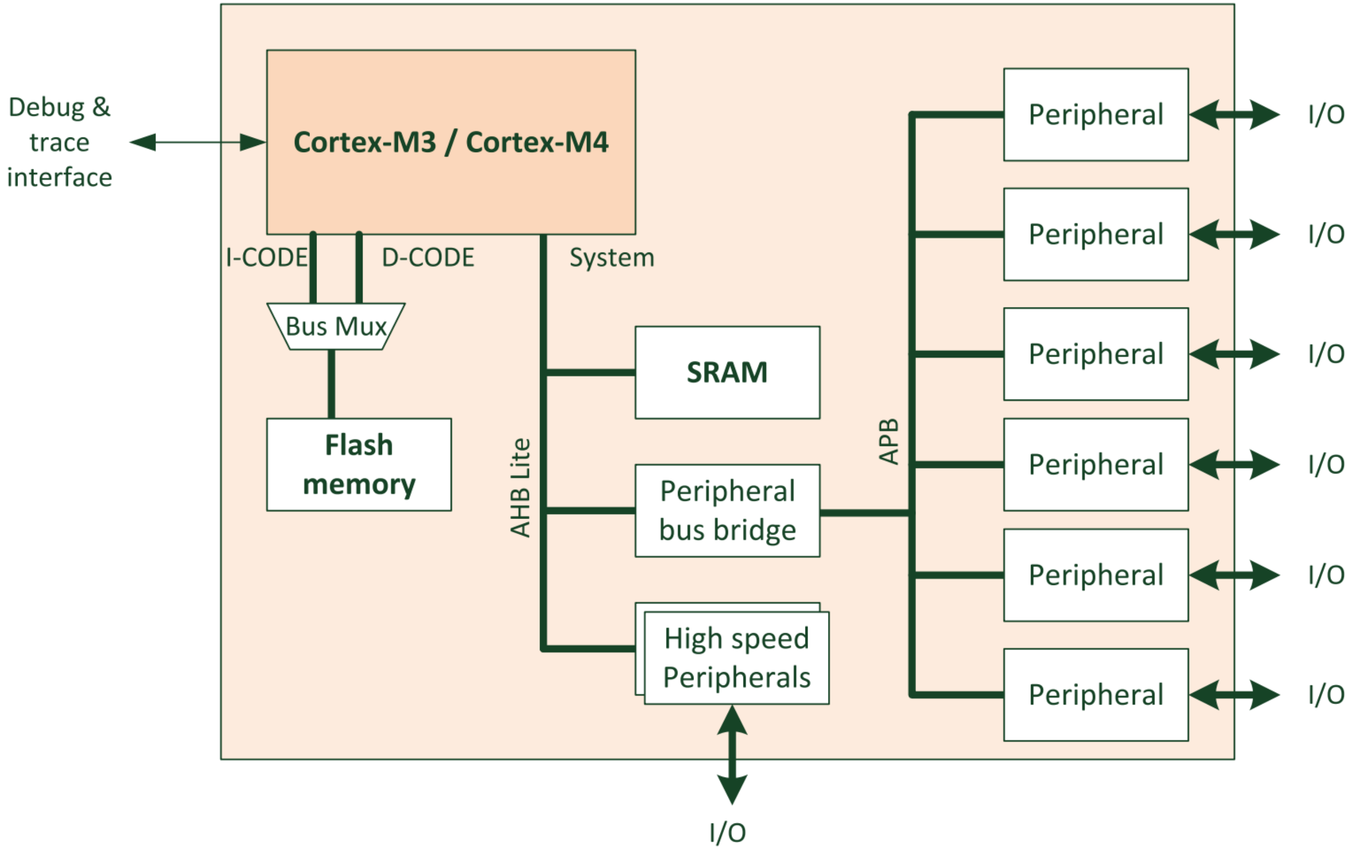


FIGURE 6.3

A simple Cortex-M3 or Cortex-M4 processor based system

6.3 Connecting the processor to memory and peripherals

- In order to provide **better performance**, the CODE memory region has **separated the bus interfaces from the system bus**
- In this way, **data accesses and instruction fetches** can be carried out **in parallel**
- The separate bus arrangement also helps to **improve the interrupt responsiveness** because during the interrupt handling sequence, stack accesses and reading the vector table in the program image can be carried out at the same time

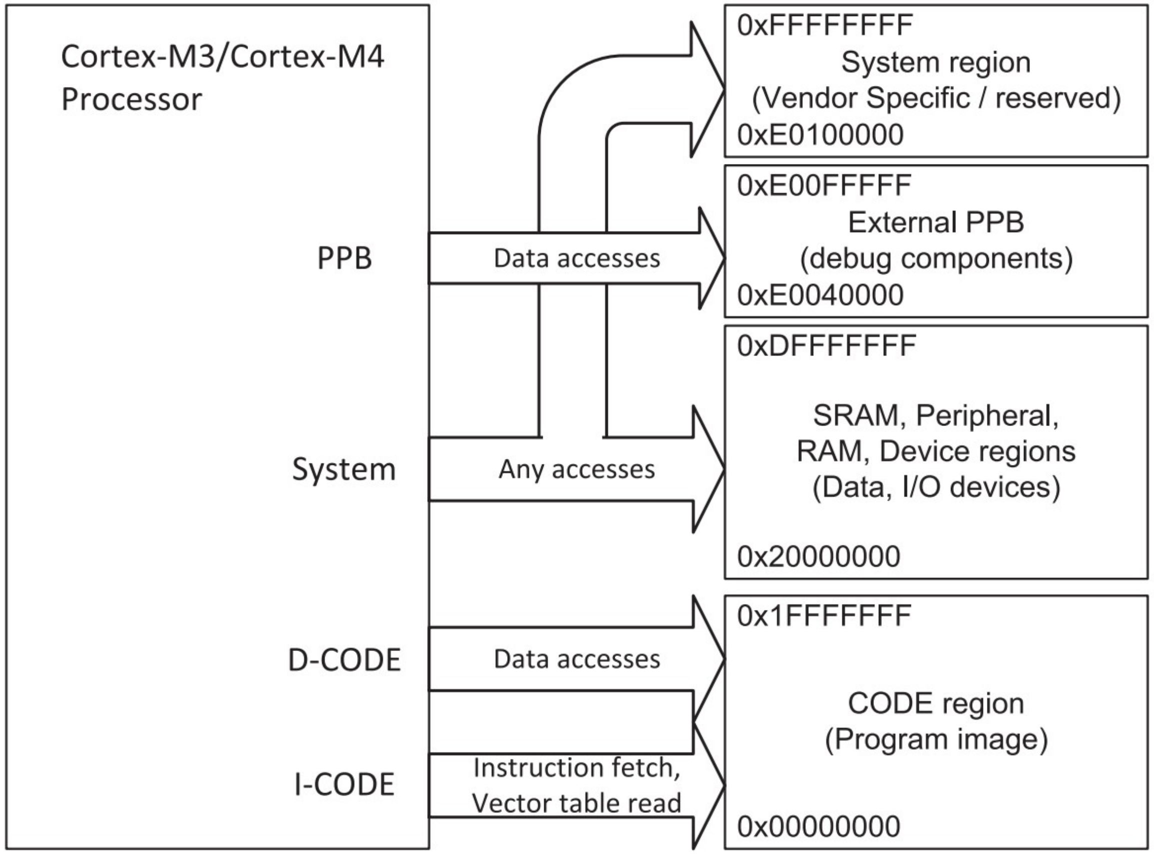

Cortex-M3/Cortex-M4 Processor
PPB
System
D-CODE
I-CODE
Data accesses
Any accesses
Data accesses
Instruction fetch, Vector table read
0xFFFFFFFF
System region (Vendor Specific / reserved)
0xE0100000
0xE00FFFFF
External PPB (debug components)
0xE0040000
0xDFFFFFFF
SRAM, Peripheral, RAM, Device regions (Data, I/O devices)
0x20000000
0x1FFFFFFF
CODE region (Program image)
0x00000000

FIGURE 6.2

Multiple bus interface for different memory regions

6.3 Connecting the processor to memory and peripherals

- Each bus may be operated at **different speeds** for best power optimization
- In some cases, this **allows higher system bandwidth** (e.g., when the peripheral system needs to support DMA accesses for Ethernet or high speed USB)
- Peripheral interfaces are usually based on the **APB protocol**
- However, for high-performance peripherals, **AHB Lite could be used instead for higher bandwidth** and operation speed
- Note the **Private Peripheral Bus (PPB)** is **not used for normal peripherals**

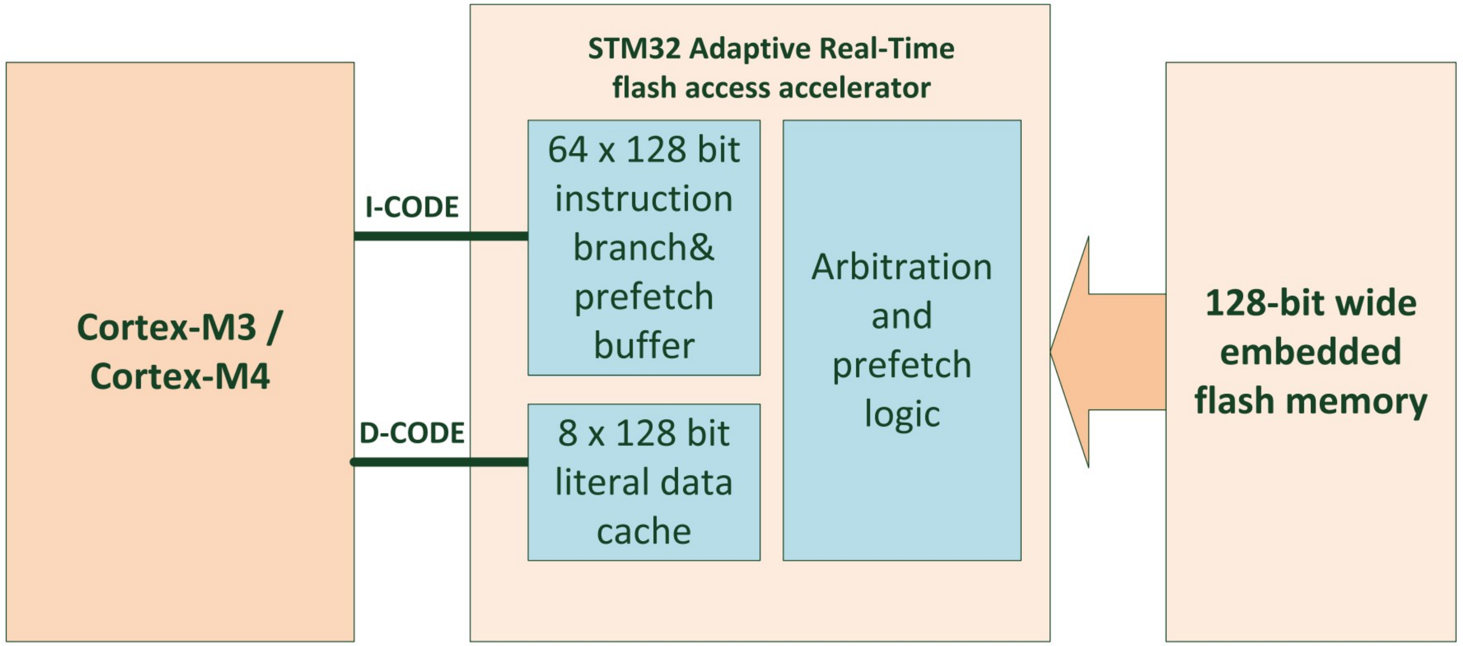

STM32 Adaptive Real-Time flash access accelerator
Cortex-M3 / Cortex-M4
I-CODE
D-CODE
64 x 128 bit instruction branch& prefetch buffer
8 x 128 bit literal data cache
Arbitration and prefetch logic
128-bit wide embedded flash memory

FIGURE 6.4

Concept of flash access accelerator in STM32F2 and STM32F4 from ST Microelectronics

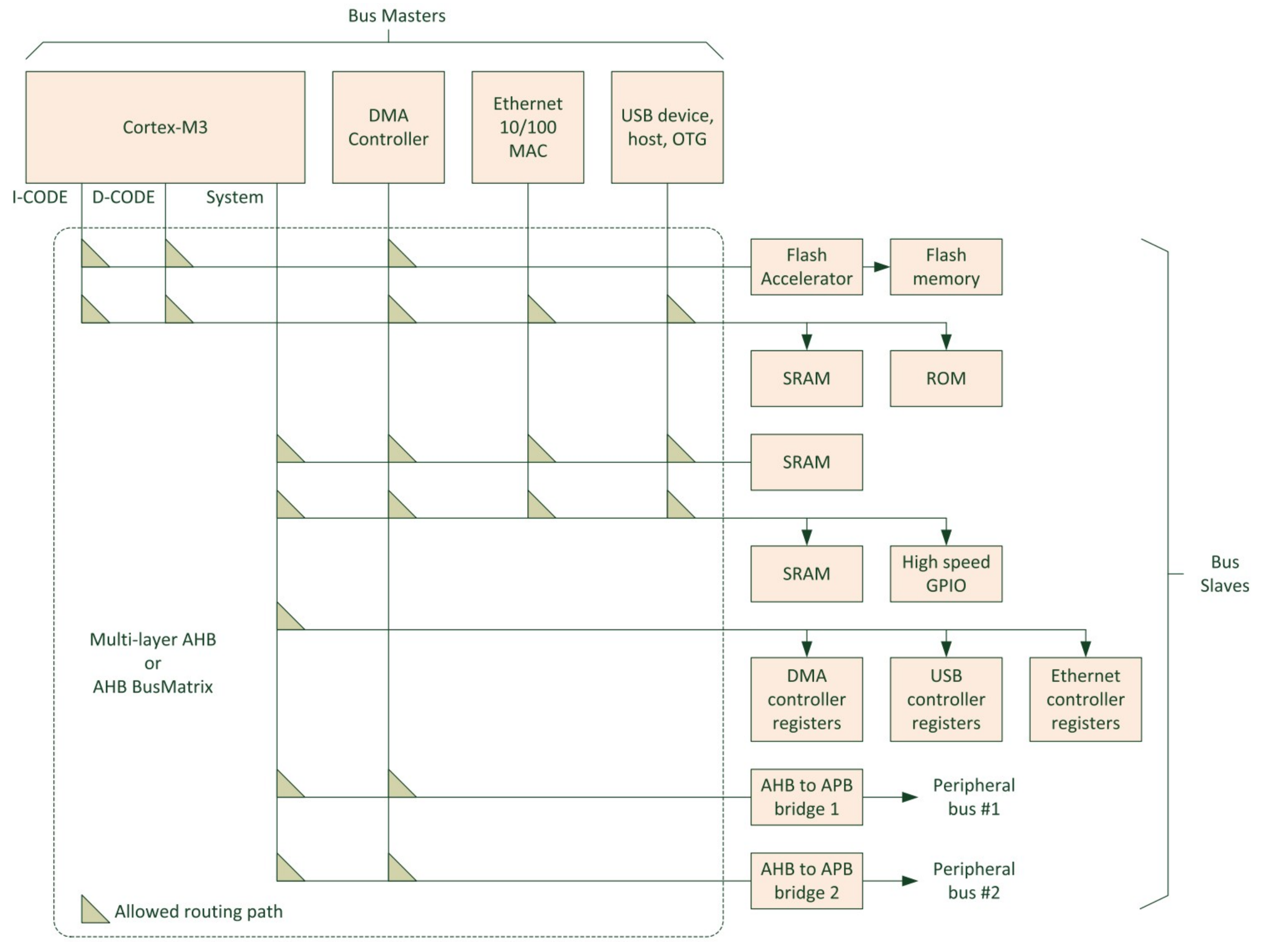


FIGURE 6.5

Multi-layer AHB example (NXP LPC1700)

6.4 Memory requirements

- **Different types of memories** can be connected to the AHB Lite bus interfaces with suitable memory interface logic
- Although the bus size is 32-bit, it is possible to connect memories with **other width sizes** (e.g., 8 bits, 16 bits, 64 bits, 128 bits, etc.) if appropriate conversion hardware is used
- Note that although the architecture uses names like SRAM and RAM for memory region names, there is **no real limitation on what types of memories** can be connected to the processors
  - For example, some memories connected could be PSRAM, SDRAM, DDR DRAM, etc.

6.4 Memory requirements

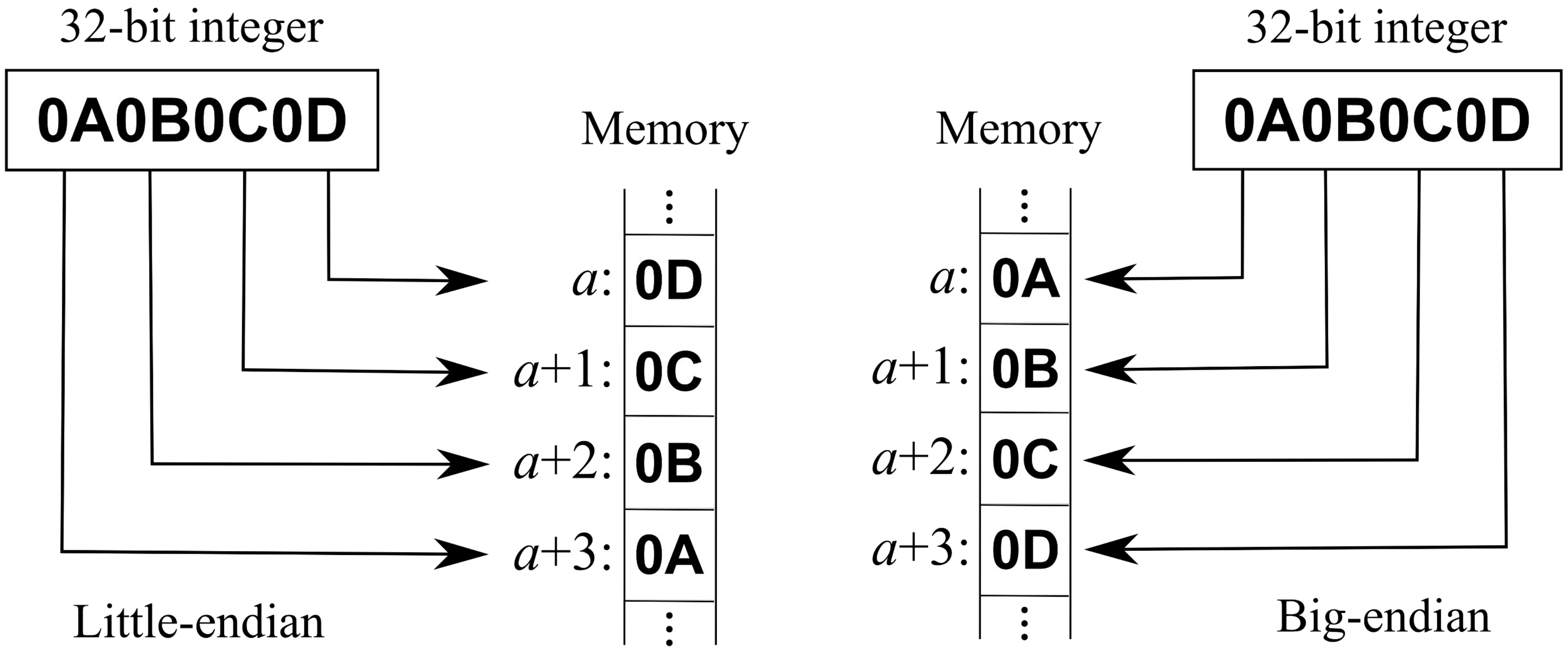
- Also, there is **no real limitation on what type of program memories** have to be used
  - For example, the program code could be in flash memory, EPROM, OTP ROM, etc.
- The only real requirement for data memory (e.g., SRAM) is that the memory **must be byte addressable and the memory interface needs to support byte, halfword, and word transfers**.

6.5 Memory endianness

- The Cortex-M3 and Cortex-M4 processors support both **little endian and big endian** memory systems
- In **normal cases** the memory systems are designed to be either little endian only, or big endian only
- Endianness of the memory system is **determined at a system reset**
- In a few cases some **peripheral registers can contain data of a different endianness**
  - In that case, the applications accessing such peripheral registers need to convert the data to the correct endianness inside the program code

6.5 Memory endianness

- With little endian systems, the bus lane usages of Cortex-M3, Cortex-M4, and classic ARM processors are the same
- In the **Cortex-M** processors:
  - **Instruction fetches are always in little endian**
  - Access to 0xE0000000 to 0xE00FFFFF including System Control Space (**SCS**), **debug components**, and Private Peripheral Bus (**PPB**) are always **little endian**
- If your software application needs to process big endian data and the microcontroller you are using is little endian, you **can convert** the data between little endian and big endian using instructions such as **REV, REVSH, and REV16**


32-bit integer
0A0B0C0D
Memory
a: 0D
a+1: 0C
a+2: 0B
a+3: 0A
Little-endian
Memory
32-bit integer
0A0B0C0D
a: 0A
a+1: 0B
a+2: 0C
a+3: 0D
Big-endian

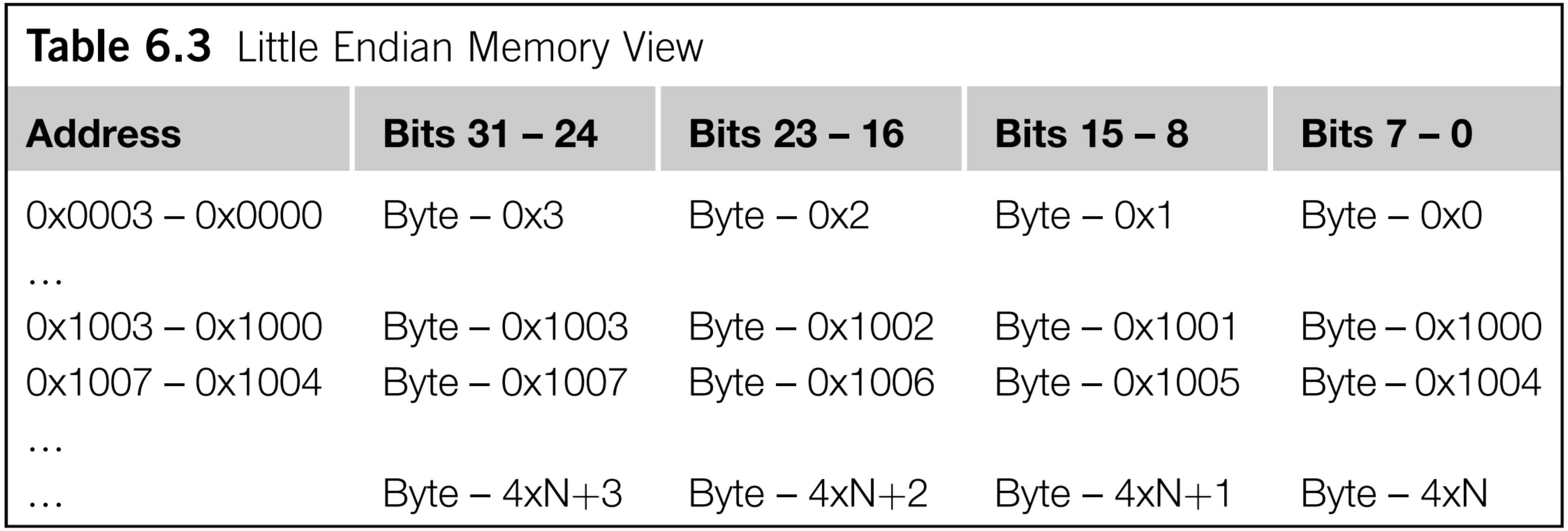


Table 6.3 Little Endian Memory View

Address	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x0003 – 0x0000	Byte – 0x3	Byte – 0x2	Byte – 0x1	Byte – 0x0
…				
0x1003 – 0x1000	Byte – 0x1003	Byte – 0x1002	Byte – 0x1001	Byte – 0x1000
0x1007 – 0x1004	Byte – 0x1007	Byte – 0x1006	Byte – 0x1005	Byte – 0x1004
…				
…	Byte – 4xN+3	Byte – 4xN+2	Byte – 4xN+1	Byte – 4xN

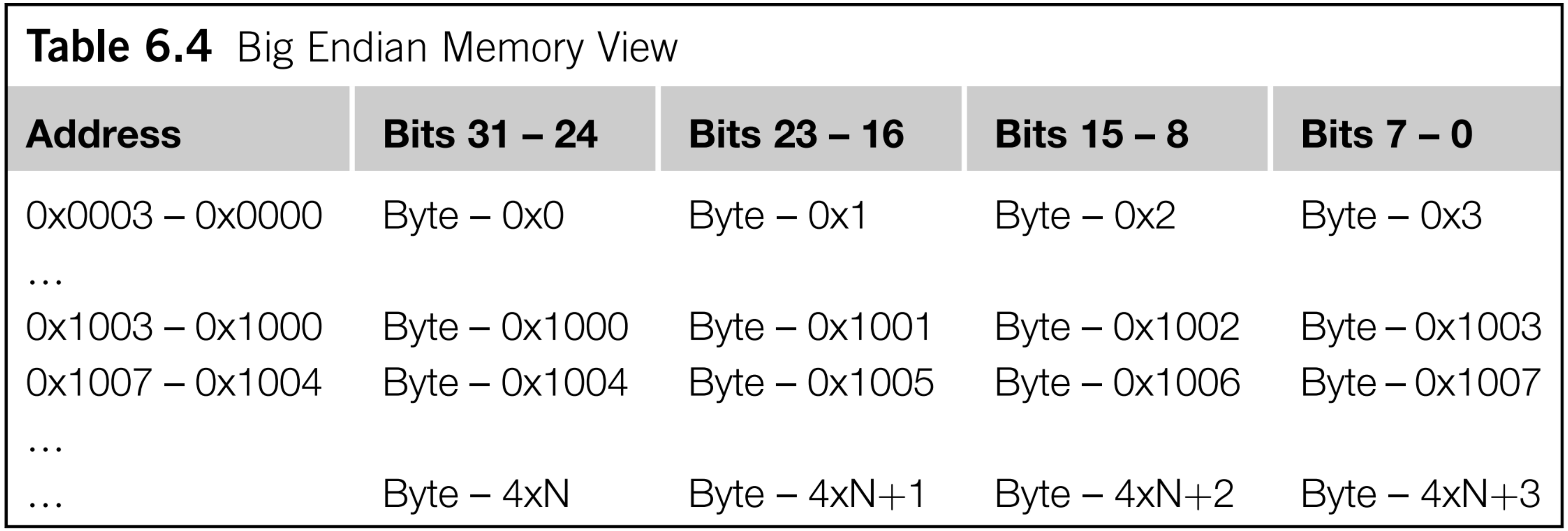


Table 6.4 Big Endian Memory View

Address	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x0003 – 0x0000	Byte – 0x0	Byte – 0x1	Byte – 0x2	Byte – 0x3
…				
0x1003 – 0x1000	Byte – 0x1000	Byte – 0x1001	Byte – 0x1002	Byte – 0x1003
0x1007 – 0x1004	Byte – 0x1004	Byte – 0x1005	Byte – 0x1006	Byte – 0x1007
…				
…	Byte – 4xN	Byte – 4xN+1	Byte – 4xN+2	Byte – 4xN+3

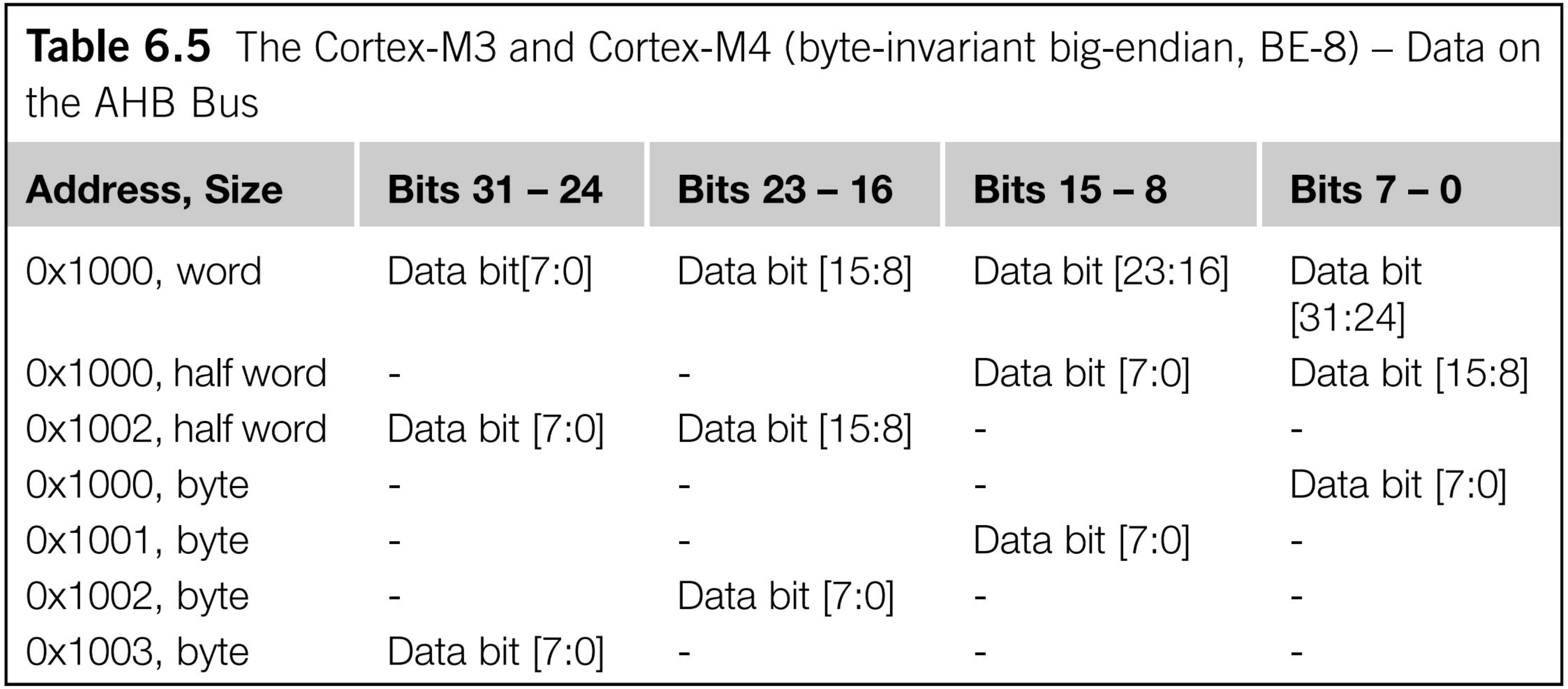


Table 6.5 The Cortex-M3 and Cortex-M4 (byte-invariant big-endian, BE-8) – Data on the AHB Bus

Address, Size	**Bits 31 – 24**	**Bits 23 – 16**	**Bits 15 – 8**	**Bits 7 – 0**
0x1000, word	Data bit[7:0]	Data bit [15:8]	Data bit [23:16]	Data bit [31:24]
0x1000, half word	-	-	Data bit [7:0]	Data bit [15:8]
0x1002, half word	Data bit [7:0]	Data bit [15:8]	-	-
0x1000, byte	-	-	-	Data bit [7:0]
0x1001, byte	-	-	Data bit [7:0]	-
0x1002, byte	-	Data bit [7:0]	-	-
0x1003, byte	Data bit [7:0]	-	-	-

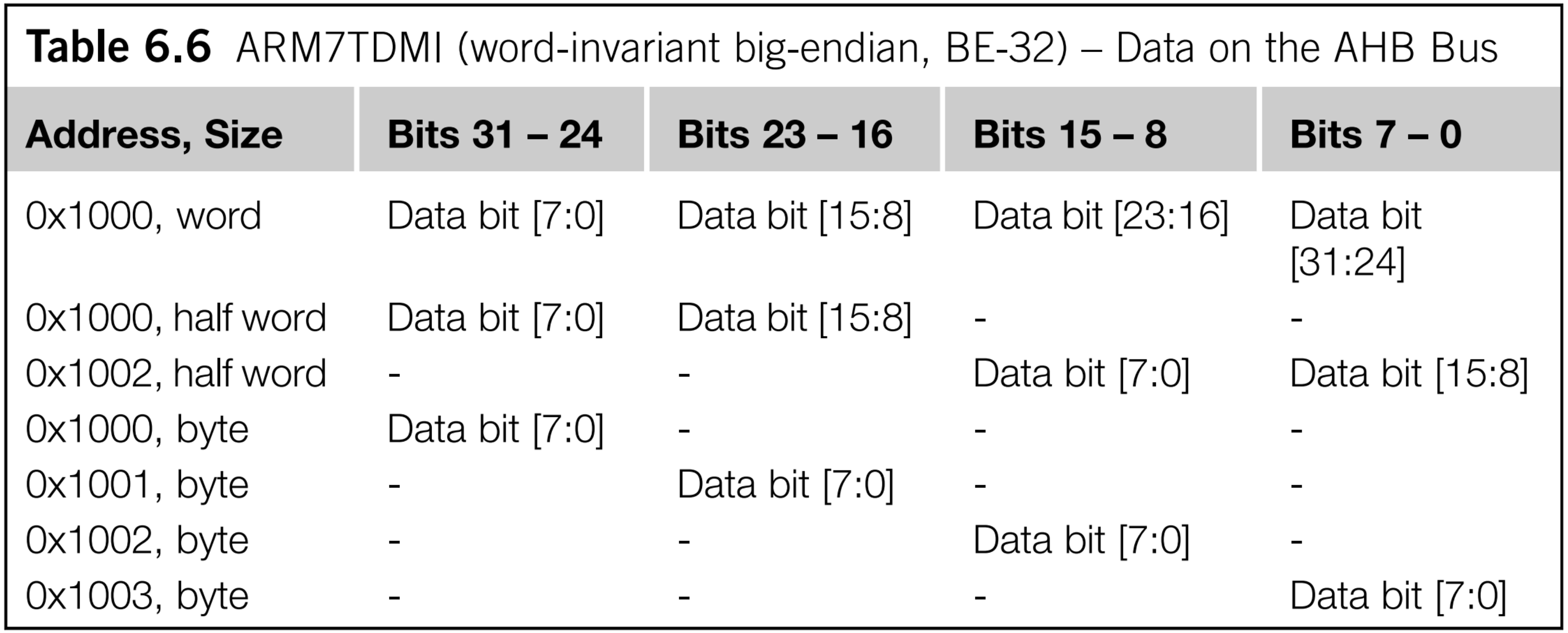


Table 6.6 ARM7TDMI (word-invariant big-endian, BE-32) – Data on the AHB Bus

Address, Size	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x1000, word	Data bit [7:0]	Data bit [15:8]	Data bit [23:16]	Data bit [31:24]
0x1000, half word	Data bit [7:0]	Data bit [15:8]	-	-
0x1002, half word	-	-	Data bit [7:0]	Data bit [15:8]
0x1000, byte	Data bit [7:0]	-	-	-
0x1001, byte	-	Data bit [7:0]	-	-
0x1002, byte	-	-	Data bit [7:0]	-
0x1003, byte	-	-	-	Data bit [7:0]

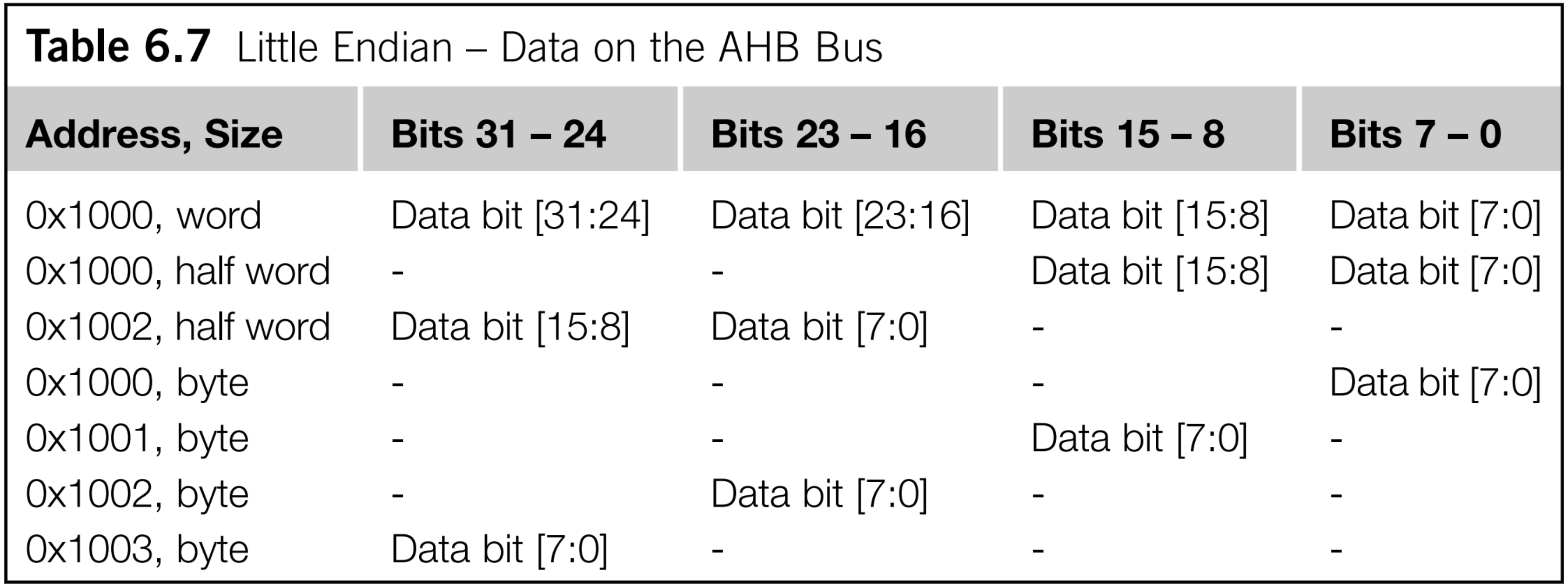


Table 6.7 Little Endian – Data on the AHB Bus

Address, Size	Bits 31 – 24	Bits 23 – 16	Bits 15 – 8	Bits 7 – 0
0x1000, word	Data bit [31:24]	Data bit [23:16]	Data bit [15:8]	Data bit [7:0]
0x1000, half word	-	-	Data bit [15:8]	Data bit [7:0]
0x1002, half word	Data bit [15:8]	Data bit [7:0]	-	-
0x1000, byte	-	-	-	Data bit [7:0]
0x1001, byte	-	-	Data bit [7:0]	-
0x1002, byte	-	Data bit [7:0]	-	-
0x1003, byte	Data bit [7:0]	-	-	-

6.6 Data alignment and unaligned data access support

- Since the memory system is 32-bit (at least from a programmer's model point of view), a data access that is 32-bit (4 bytes, or word) or 16-bit (2 bytes, or halfword) in size **can either be aligned or unaligned**
- **Aligned transfers means the address value is a multiple of the size (in bytes)**
  - **For example, a word-size aligned transfer can be carried out to address 0x00000000, 0x00000004 . 0X00001000, 0x00001004, ., and so on**
  - **Similarly, a half-word size aligned transfer can be carried out to 0x00000000, 0x00000002 . 0x00001000, 0x00001002, ., and so on**

6.6 Data alignment and unaligned data access support

- **Traditionally**, most classic ARM processors (such as the ARM7/ARM9/ARM10) **allow only aligned transfers**.
  - That means that in accessing memory, a word transfer must have address bit[1] and bit[0] equal to 0, and a half-word transfer must have an address bit[0] equal to 0
    - For example, word data can be located at 0x1000 or 0x1004, but it cannot be located at 0x1001, 0x1002, or 0x1003.
    - For half-word data, the address can be 0x1000 or 0x1002, but it cannot be 0x1001.
  - All byte size transfers are aligned
- **The Cortex-M3 and Cortex-M4 processors support unaligned data transfers in *normal memory* accesses (e.g., LDR, LDRH, STR, STRH instructions)**

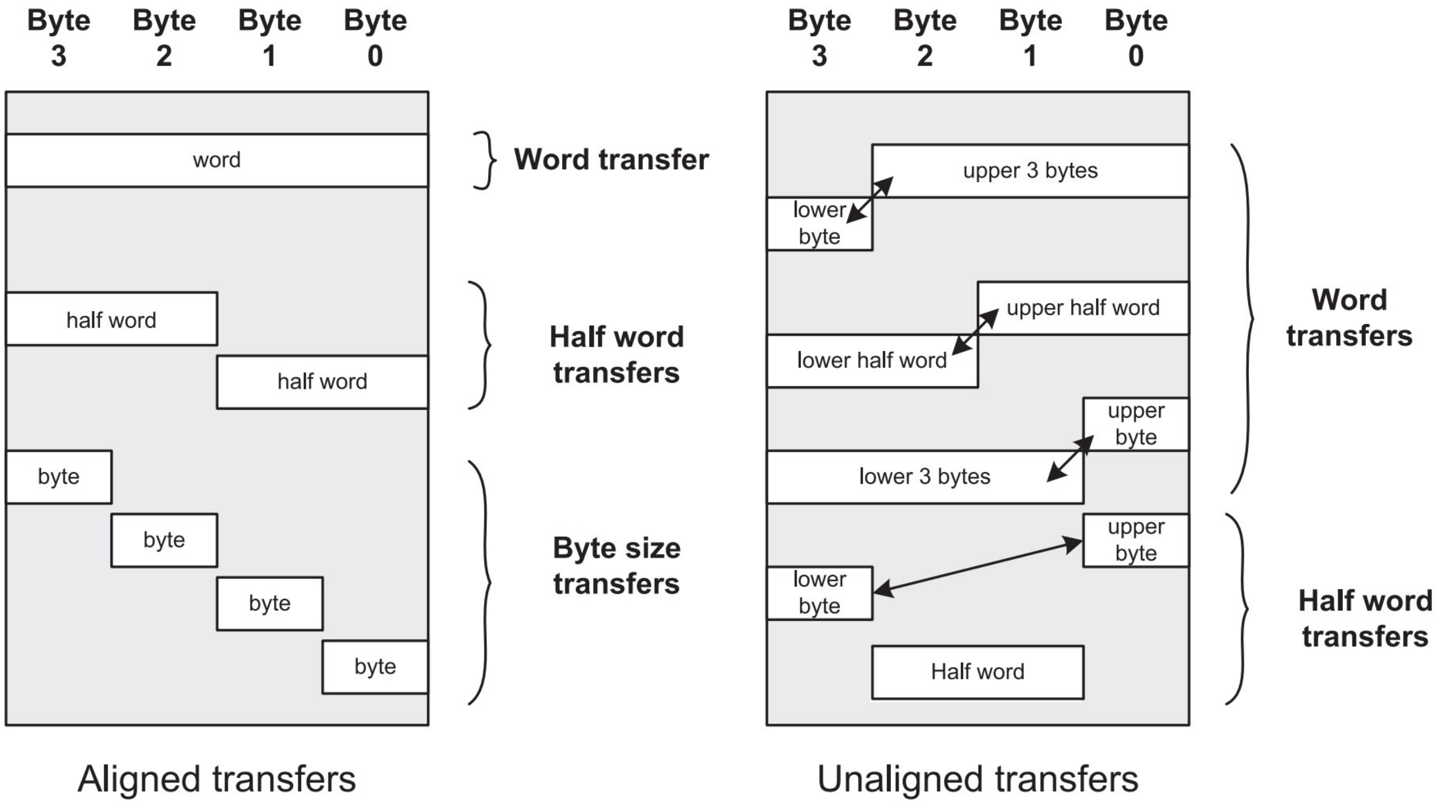


FIGURE 6.6

Example of aligned and unaligned data transfers in little endian memory system

6.6 Data alignment and unaligned data access support

- There are a number of limitations:
  - Unaligned transfers are not supported in **Load/Store multiple** instructions
  - **Stack operations** (PUSH/POP) must be aligned
  - **Exclusive accesses** (such as LDREX or STREX) must be aligned; otherwise, a fault exception (usage fault) will be triggered
  - Unaligned transfers are not supported in **bit-band operations.** Results will be unpredictable if you attempt to do so

6.6 Data alignment and unaligned data access support

- When unaligned transfers are issued by the processor, they are **actually converted into multiple aligned transfers** by the processor’s bus interface unit
- This **conversion is transparent**, so application programmers do not have to worry about it
- However, when an unaligned transfer takes place, it is broken into several separate aligned transfers and as a result **it takes more clock cycles** for a single data access and might not be good in situations in which high performance is required

6.6 Data alignment and unaligned data access support

- To get the **best performance**, it’s worth making sure that data are **aligned properly**
- In most cases, C compilers do not generate unaligned data accesses. It can only happen in:
  - Direct manipulation of pointers
  - Accessing data structures with “__packed” attributes that contain unaligned data
  - Inline/Embedded Assembly code

6.7 Bit-band operations

- Bit-band operation support allows a **single load/store operation to access (read/write) to a single data bit**
- In the Cortex-M3 and Cortex-M4 processors there are two **bit-band regions**
- One in the **first 1MB of the SRAM region**, and the other in the **first 1MB of the peripheral region**
- These two memory regions can be accessed like normal memory, but they can also be accessed via a separate memory region called the **bit-band alias**
- When the bit-band alias address is used, each individual bit can be accessed separately in the **least significant bit (LSB) of each word-aligned address**

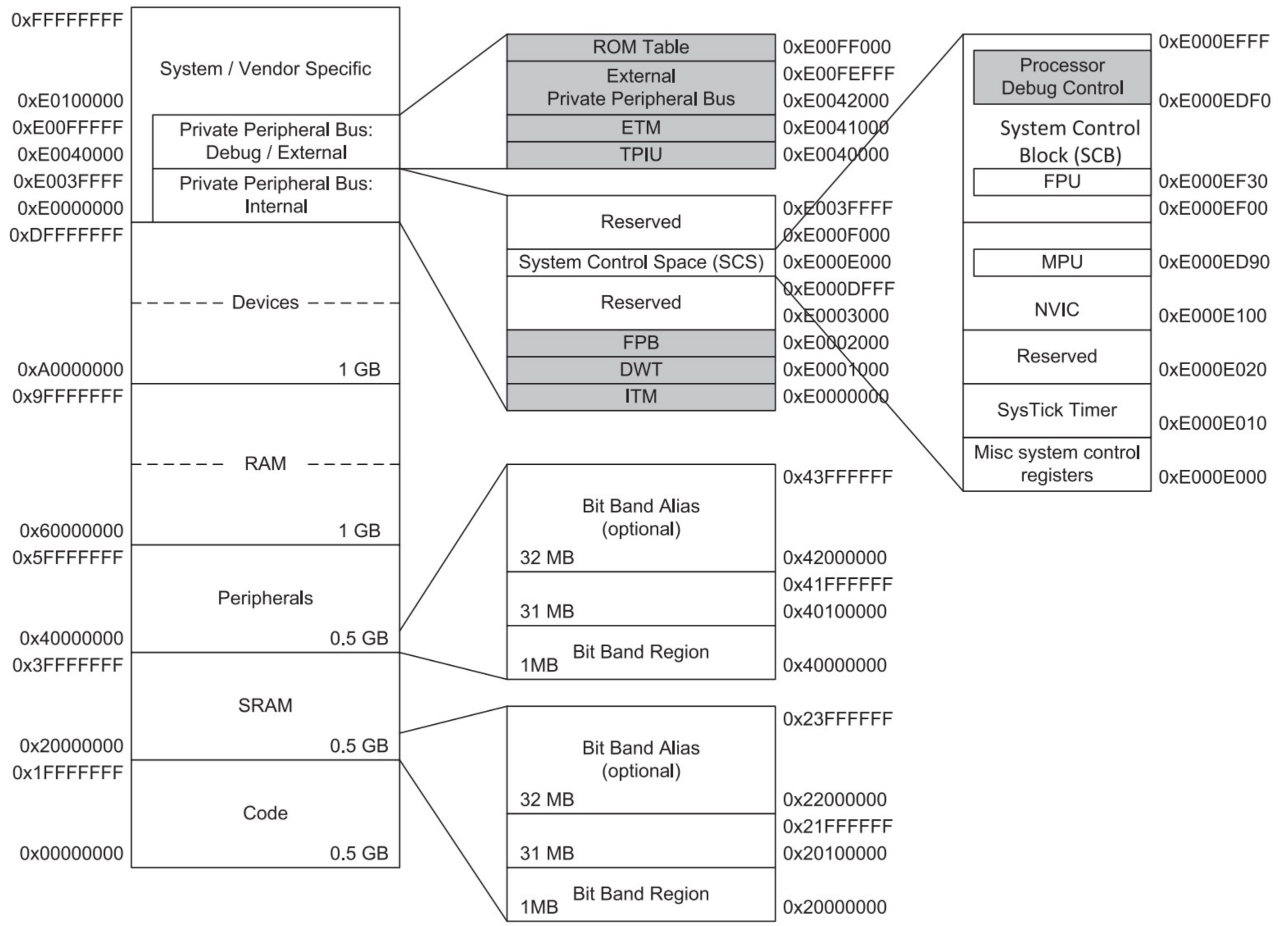


FIGURE 6.1

Pre-defined memory map of the Cortex®-M3 and Cortex-M4 processors (shaded areas are components for debug purpose)

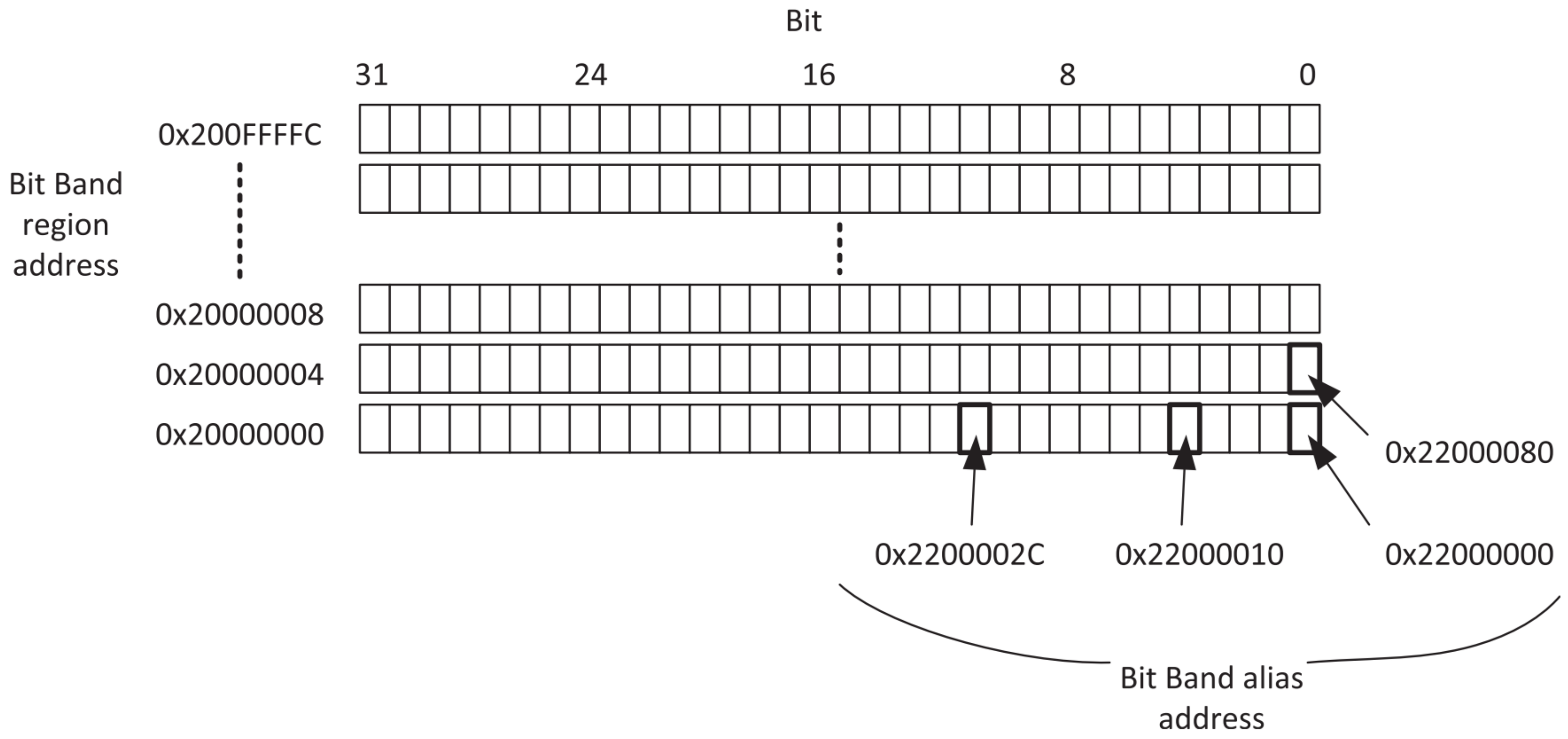


FIGURE 6.7

Bit accesses to bit-band region via the bit-band alias (SRAM region)

Write

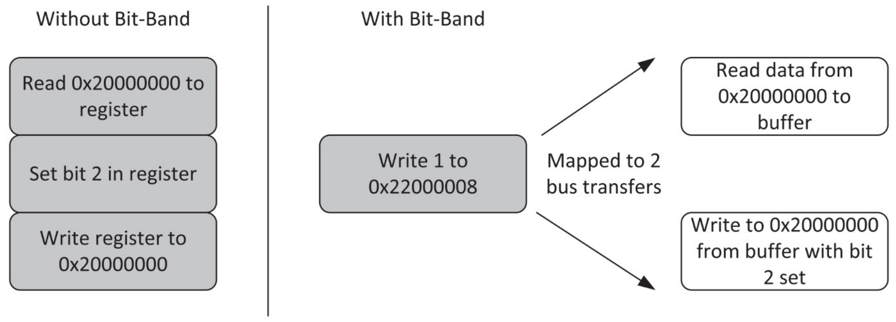

Without Bit-Band
Read 0x20000000 to register
Set bit 2 in register
Write register to 0x20000000
With Bit-Band
Write 1 to 0x22000008
Mapped to 2 bus transfers
Read data from 0x20000000 to buffer
Write to 0x20000000 from buffer with bit 2 set

FIGURE 6.8

Write to bit-band alias

Without Bit-Band

```
LDR   R0,=0x20000000 ; Setup address
LDR   R1, [R0]       ; Read
ORR.W R1, #0x4       ; Modify bit
STR   R1, [R0] ; Write back result
```

With Bit-Band

```
LDR   R0,=0x22000008 ; Setup address
MOV   R1, #1         ; Setup data
STR   R1, [R0]       ; Write
```

FIGURE 6.9

Example assembler sequence to write a bit with and without bit-band

Read

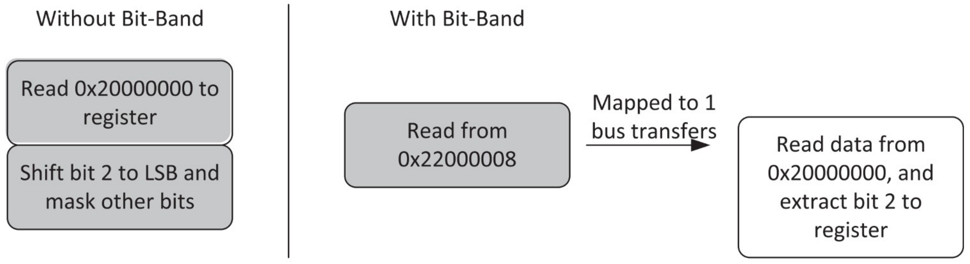

Without Bit-Band
Read 0x20000000 to register
Shift bit 2 to LSB and mask other bits
With Bit-Band
Read from 0x22000008
Mapped to 1 bus transfers
Read data from 0x20000000, and extract bit 2 to register

FIGURE 6.10

Read from the bit-band alias

Without Bit-Band

```
LDR    R0,=0x20000000 ; Setup address
LDR    R1, [R0]       ; Read
UBFX.W R1, R1, #2, #1 ; Extract bit[2]
```

With Bit-Band

```
LDR   R0,=0x22000008 ; Setup address
LDR   R1, [R0]       ; Read
```

FIGURE 6.11

Read from the bit-band alias

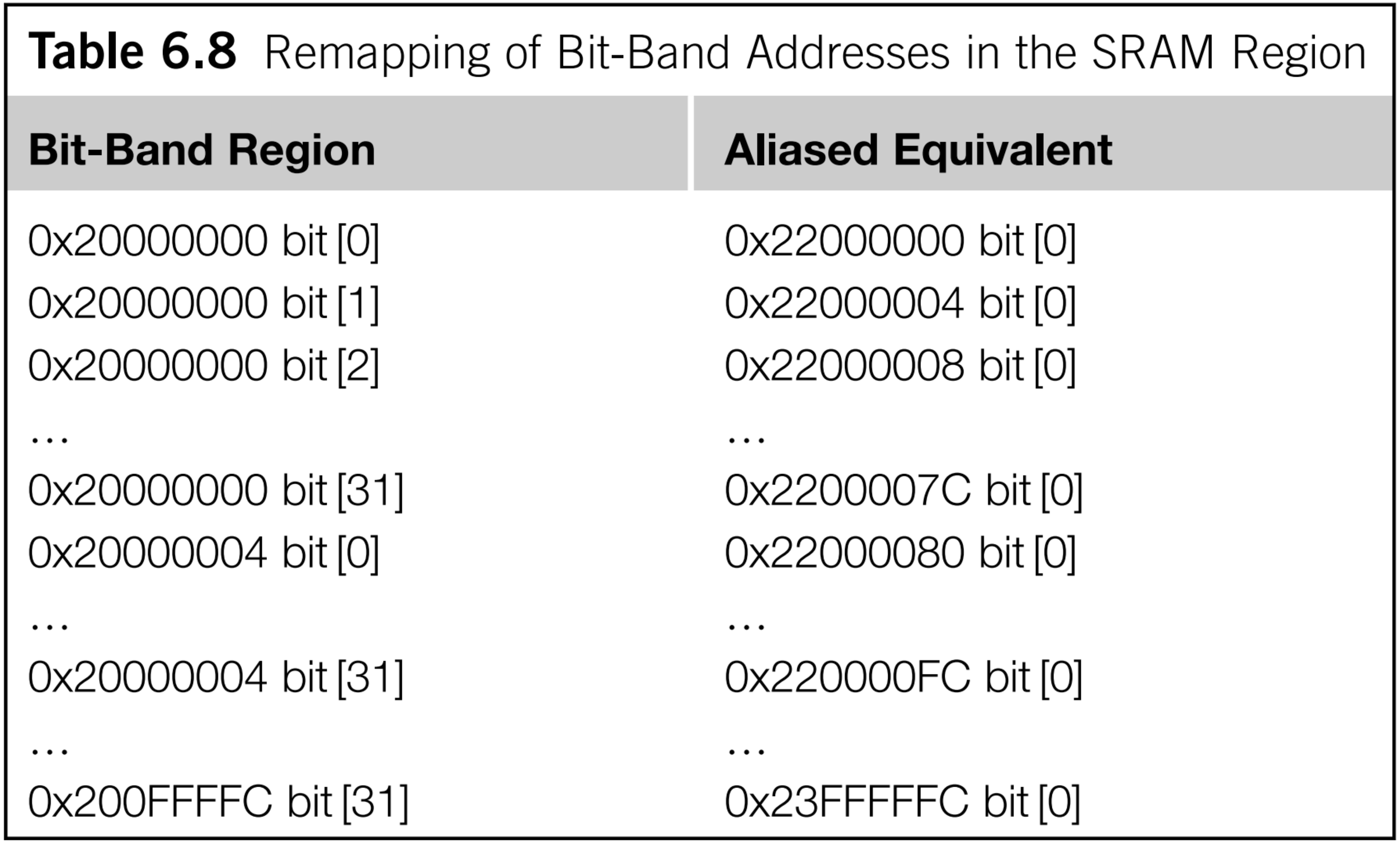


Table 6.8 Remapping of Bit-Band Addresses in the SRAM Region

Bit-Band Region	Aliased Equivalent
0x20000000 bit [0]	0x22000000 bit [0]
0x20000000 bit [1]	0x22000004 bit [0]
0x20000000 bit [2]	0x22000008 bit [0]
…	…
0x20000000 bit [31]	0x2200007C bit [0]
0x20000004 bit [0]	0x22000080 bit [0]
…	…
0x20000004 bit [31]	0x220000FC bit [0]
…	…
0x200FFFFC bit [31]	0x23FFFFFC bit [0]

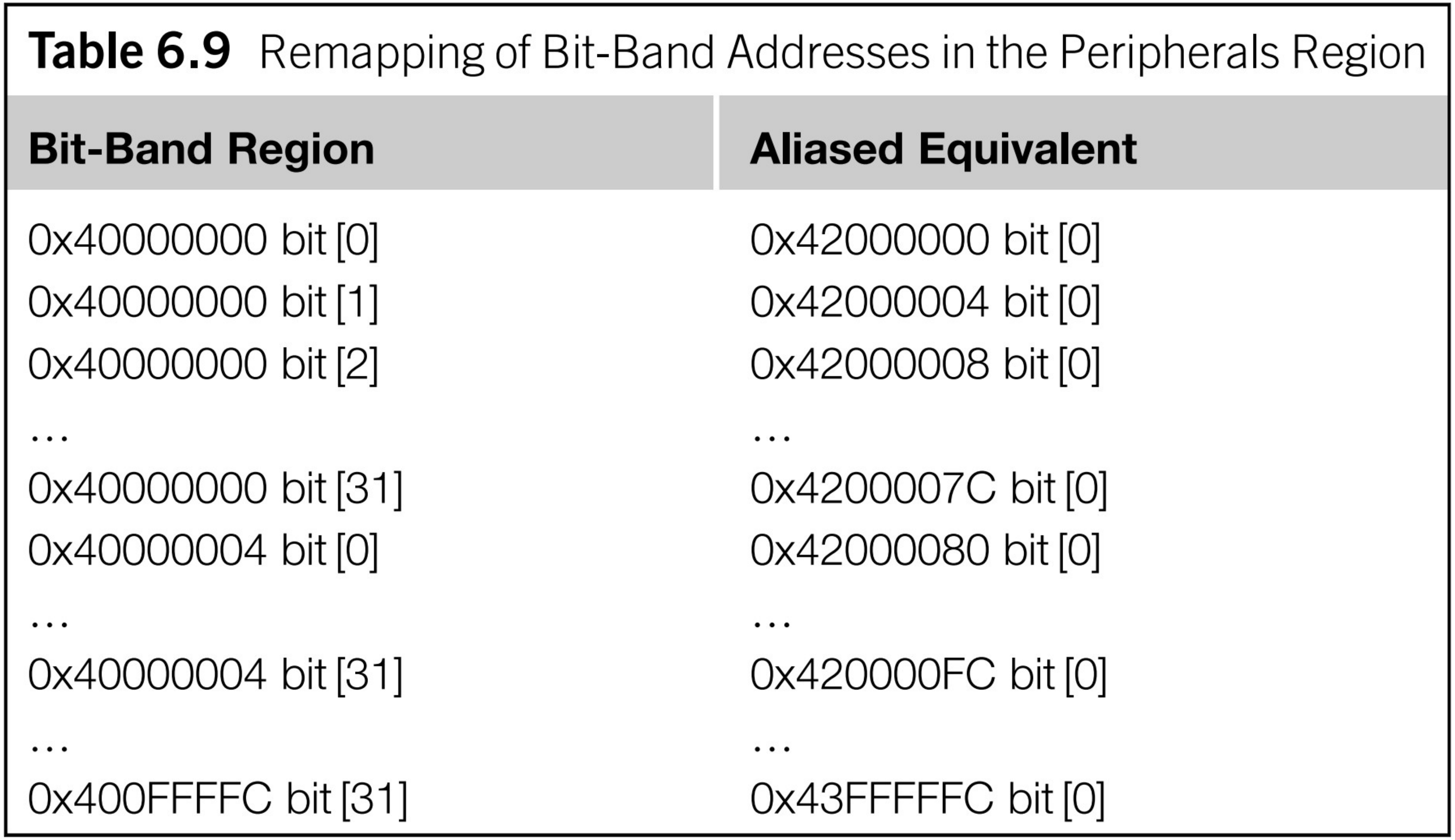
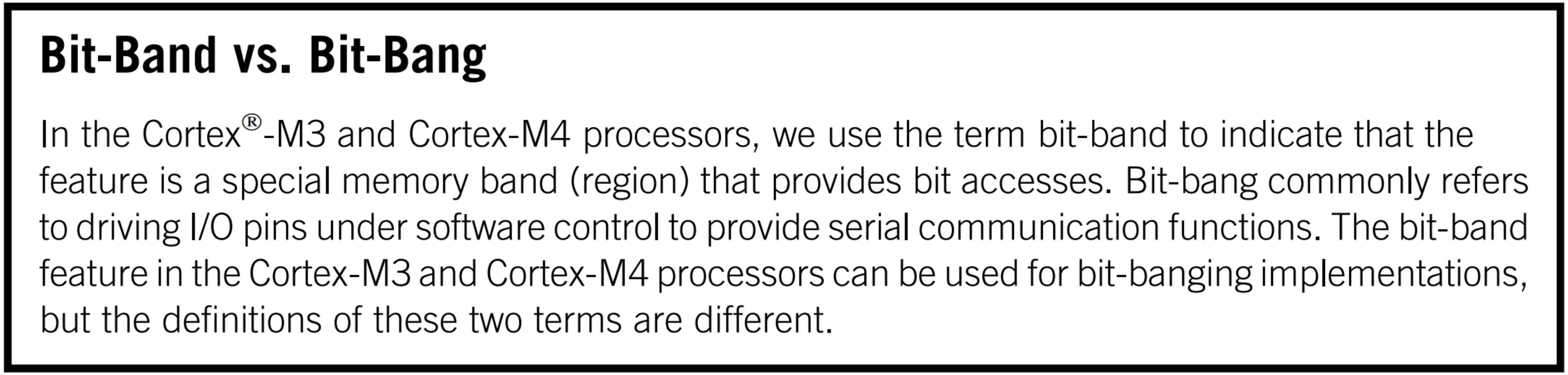


Table 6.9 Remapping of Bit-Band Addresses in the Peripherals Region

Bit-Band Region	Aliased Equivalent
0x40000000 bit [0]	0x42000000 bit [0]
0x40000000 bit [1]	0x42000004 bit [0]
0x40000000 bit [2]	0x42000008 bit [0]
…	…
0x40000000 bit [31]	0x4200007C bit [0]
0x40000004 bit [0]	0x42000080 bit [0]
…	…
0x40000004 bit [31]	0x420000FC bit [0]
…	…
0x400FFFFC bit [31]	0x43FFFFFC bit [0]

6.7.2 Advantages of bit-band operations

- We can more easily implement serial data transfers in general-purpose input/output (GPIO) ports to serial devices



Bit-Band vs. Bit-Bang

In the Cortex®-M3 and Cortex-M4 processors, we use the term bit-band to indicate that the feature is a special memory band (region) that provides bit accesses. Bit-bang commonly refers to driving I/O pins under software control to provide serial communication functions. The bit-band feature in the Cortex-M3 and Cortex-M4 processors can be used for bit-banging implementations, but the definitions of these two terms are different.

6.7.2 Advantages of bit-band operations

- Can be used to **simplify branch decisions**
- Essential for situations in which **resources (e.g., various pins in I/O port) are being shared** by more than one process
- One of the most important advantages or properties of bit-band operation is that it is **atomic**

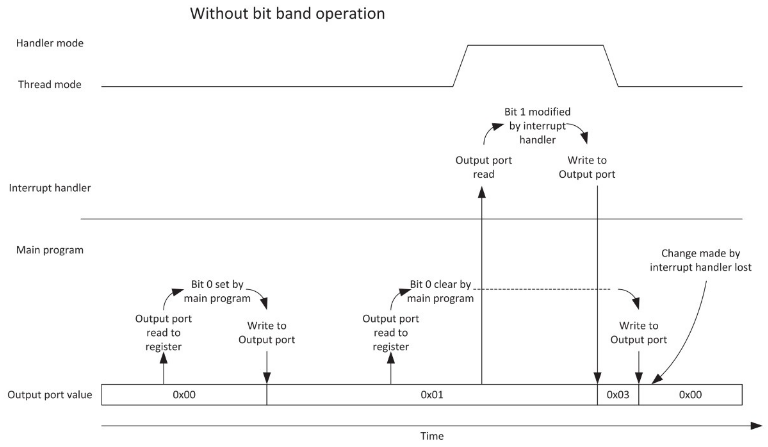

Without bit band operation
Handler mode
Thread mode
Bit 1 modified by interrupt handler
Output port read
Write to Output port
Interrupt handler
Main program
Change made by interrupt handler lost
Bit 0 set by main program
Bit 0 clear by main program
Output port read to register
Write to Output port
Output port read to register
Write to Output port
Output port value
0x00
0x01
0x03
0x00
Time

FIGURE 6.12

Data are lost when an exception handler modifies a shared memory location

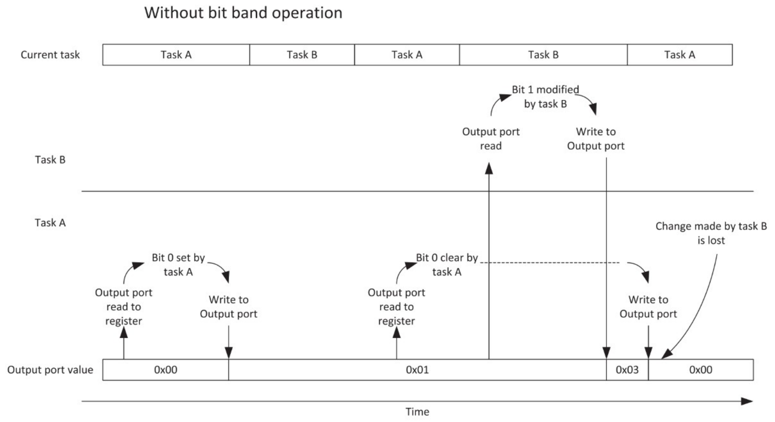


FIGURE 6.14

Data are lost when a different task modifies a shared memory location

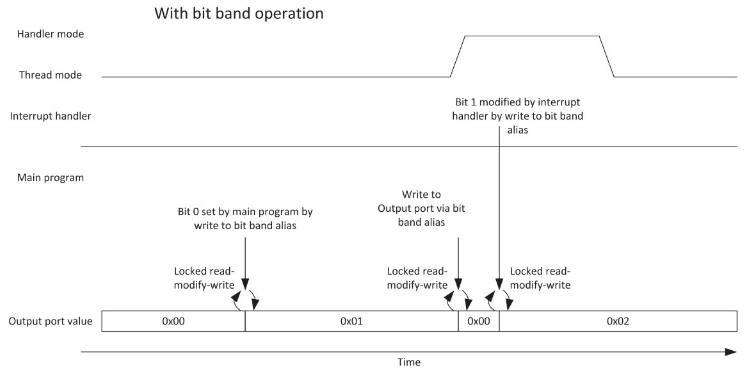


FIGURE 6.13

Data loss prevention with locked transfers using the bit-band feature

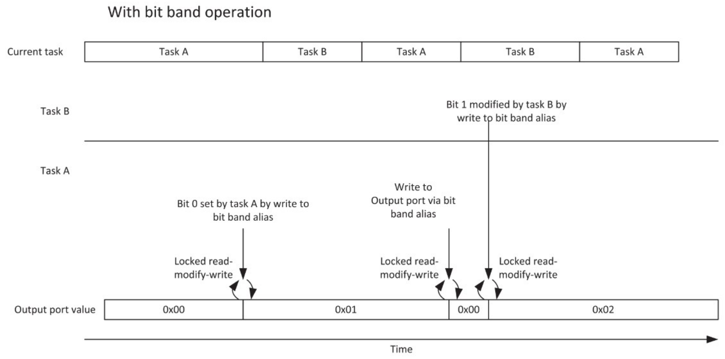


FIGURE 6.15

Data loss prevention with locked transfers using the bit-band feature

6.7.4 Bit-band operations in C programs

- There is **no native support of bit-band operations in C/C++** languages
  - For example, C compilers do not understand that the same memory can be accessed using two different addresses
  - and they do not know that accesses to the bit-band alias will only access the LSB of the memory location
- To use the bit-band feature in C, the simplest solution is to **separately declare the address and the bit-band alias of a memory location**

6.7.4 Bit-band operations in C programs

```
#define DEVICE_REG0       *((volatile unsigned long *) (0x40000000))
#define DEVICE_REG0_BIT0  *((volatile unsigned long *) (0x42000000))
#define DEVICE_REG0_BIT1  *((volatile unsigned long *) (0x42000004))
...
DEVICE_REG0 = 0xAB; // Accessing the hardware register by normal
// address
...
DEVICE_REG0 = DEVICE_REG0 | 0x2; // Setting bit 1 without using
// bit-band feature
...
DEVICE_REG0_BIT1 = 0x1; // Setting bit 1 using bit-band feature
// via the bit-band alias address
```

6.9 Memory access attributes

- **Bufferable:** Write to memory can be carried out by a write buffer while the processor continues on to next instruction execution
- **Cacheable:** Data obtained from memory read can be copied to a memory cache so that next time it is accessed the value can be obtained from the cache to speed up program execution

6.9 Memory access attributes

- **Executable:** The processor can fetch and execute program code from this memory region
- **Sharable:** Data in this memory region could be shared by multiple bus masters. The memory system needs to ensure coherency of data between different bus masters in the shareable memory region

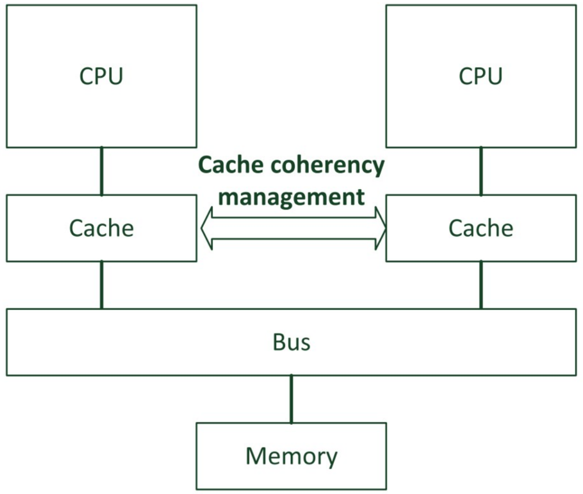


FIGURE 6.16

Cache coherency in multi-processor system needs Sharable attribute

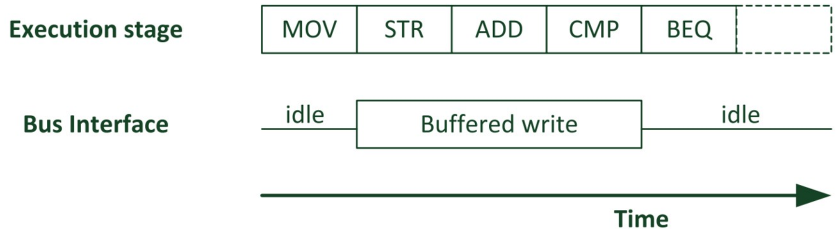


FIGURE 6.17

Buffered write operation

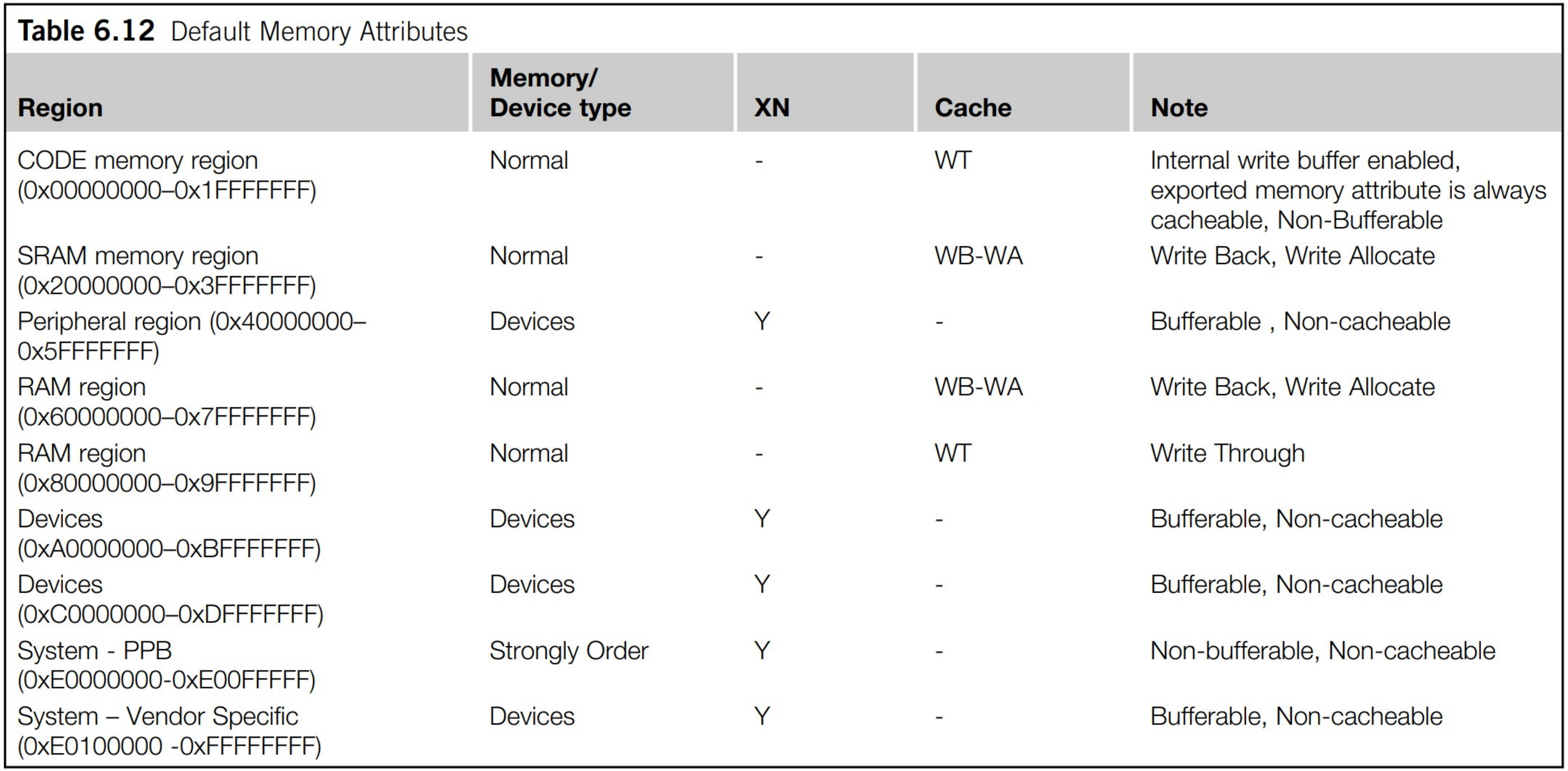


Table 6.12 Default Memory Attributes

Region	Memory/ Device type	XN	Cache	Note
CODE memory region (0x00000000–0x1FFFFFFF)	Normal	-	WT	Internal write buffer enabled, exported memory attribute is always cacheable, Non-Bufferable
SRAM memory region (0x20000000–0x3FFFFFFF)	Normal	-	WB-WA	Write Back, Write Allocate
Peripheral region (0x40000000–0x5FFFFFFF)	Devices	Y	-	Bufferable , Non-cacheable
RAM region (0x60000000–0x7FFFFFFF)	Normal	-	WB-WA	Write Back, Write Allocate
RAM region (0x80000000–0x9FFFFFFF)	Normal	-	WT	Write Through
Devices (0xA0000000–0xBFFFFFFF)	Devices	Y	-	Bufferable, Non-cacheable
Devices (0xC0000000–0xDFFFFFFF)	Devices	Y	-	Bufferable, Non-cacheable
System - PPB (0xE0000000-0xE00FFFFF)	Strongly Order	Y	-	Non-bufferable, Non-cacheable
System – Vendor Specific (0xE0100000 -0xFFFFFFFF)	Devices	Y	-	Bufferable, Non-cacheable

6.10 Exclusive accesses

- **Semaphores** are commonly used for allocating shared resources to applications.
- When a shared resource can only service one client or application processor, we also call it **Mutual Exclusion (MUTEX)**
- In such cases, when a resource is being used by one process, it is **locked** to that process and cannot serve another process until the **lock is released**
- To set up a **MUTEX semaphore**, a memory location is defined as the lock flag to indicate whether a shared resource is locked by a process
- When a process or application wants **to use the resource**, it needs to **check** whether the resource has **been locked** first. If it is not being used, it can set the **lock flag** to indicate that the resource is now locked

6.10 Exclusive accesses

- The concept of **exclusive access** operation is quite simple and allows the possibility that the memory location for a semaphore can be accessed by **another bus master or another process** running on the same processor
- To allow exclusive access to work properly in a multiple processor environment, an additional hardware unit called the "**exclusive access monitor**" is required
  - This monitor **checks the transfers** toward shared address locations and **replies to the processor** if an exclusive access is successful

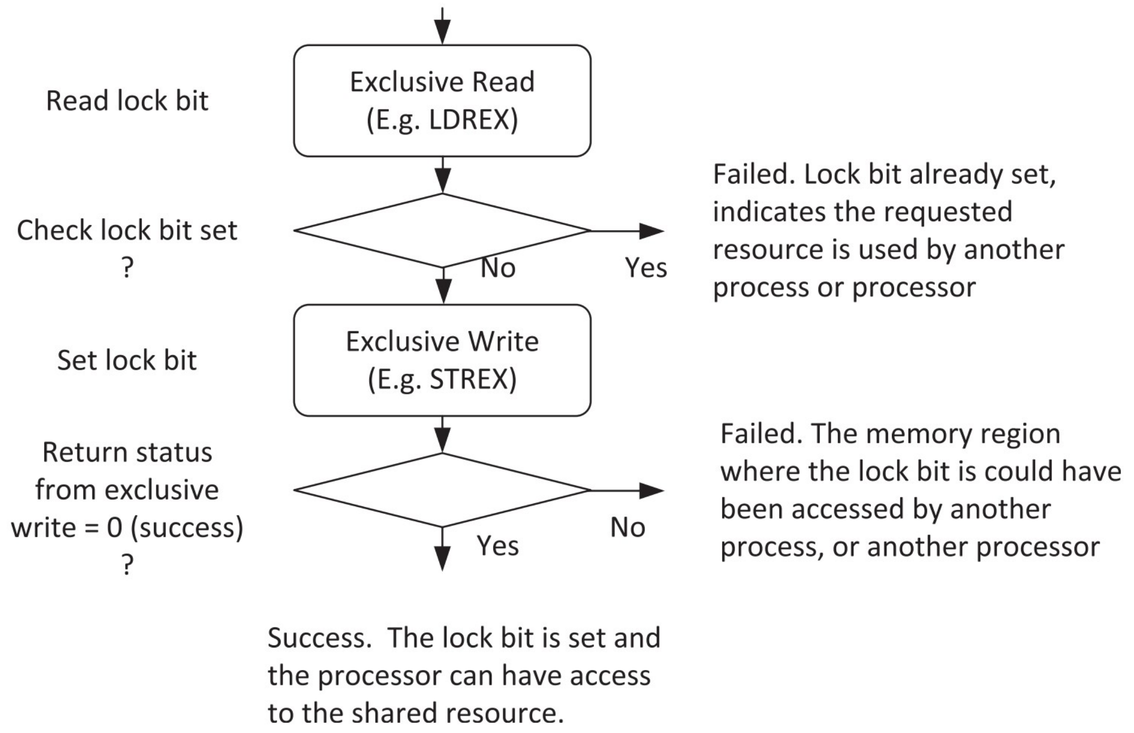

Read lock bit
Exclusive Read (E.g. LDREX)
Check lock bit set ?
No
Yes
Failed. Lock bit already set, indicates the requested resource is used by another process or processor
Set lock bit
Exclusive Write (E.g. STREX)
Return status from exclusive write = 0 (success) ?
Yes
No
Failed. The memory region where the lock bit is could have been accessed by another process, or another processor
Success. The lock bit is set and the processor can have access to the shared resource.

FIGURE 6.18

Using exclusive access in MUTEX semaphore

6.10 Exclusive accesses

- Exclusive access instructions in the Cortex-M3 and Cortex-M4 processors include `LDREX` (word), `LDREXB` (byte), `LDREXH` (half word), `STREX` (word), `STREXB` (byte), and `STREXH` (half word)
- You can access exclusive access instructions in C using intrinsic functions provided in **CMSIS-compliant device driver libraries** from microcontroller vendors: `__LDREX, __LEDEXH, __LDREXB, __STREX, __STREXH, __STREXB`

6.11 Memory barriers

- In most applications running in Cortex-M3 and Cortex-M4 microcontrollers, omission of **memory barrier instructions** (`ISB`, `DSB`, `DMB`) does not cause any issues because:
  - The Cortex-M3 and Cortex-M4 processors do not re-order any memory transfers or instruction execution
  - The simple nature of the AHB Lite and APB protocol does not allow a transfer to start before the previous one finishes.
- Likely would only use with a memory-remapping procedure such as our vector table remapping example

6.12 Memory system in a microcontroller

- In many microcontroller devices, the designs integrate additional memory system features such as:
  - Boot loader
  - Memory remapping
  - Memory alias
- These features are **not a part of the processor**, and are implemented differently in microcontrollers from different vendors
- There features can be used together to provide a **more flexible** memory-map arrangement