

ARM Cortex™-M Programming Guide to Memory Barrier Instructions

Application Note 321

Released on: 17 September 2012

ARM®

ARM Cortex-M Programming Guide to Memory Barrier Instructions

Application Note 321

Copyright © 2012 ARM Limited. All rights reserved.

Release Information

Table 1 Change history

Date	Issue	Confidentiality	Change
17 September 2012	A	Non-Confidential	First release

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM® Limited in the EU and other countries, except as otherwise stated below in this proprietary notice. Other brands and names mentioned herein may be the trademarks of their respective owners.

Neither the whole nor any part of the information contained in, or the product described in, this document may be adapted or reproduced in any material form except with the prior written permission of the copyright holder.

The product described in this document is subject to continuous developments and improvements. All particulars of the product and its use contained in this document are given by ARM in good faith. However, all warranties implied or expressed, including but not limited to implied warranties of merchantability, or fitness for purpose, are excluded.

This document is intended only to assist the reader in the use of the product. ARM Limited shall not be liable for any loss or damage arising from the use of any information in this document, or any error or omission in such information, or any incorrect use of the product.

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

The information in this document is final, that is for a developed product.

Web Address

<http://www.arm.com>

1 Introduction

This section introduces the ARM Cortex-M processors and memory barriers. It contains the following sub-sections:

- [ARM Cortex-M Processors](#)
- [Memory barriers](#)
- [The need for memory barriers on page 4](#)

1.1 ARM Cortex-M Processors

The ARM Cortex-M processors are high performance, low cost, low power, 32-bit RISC processors, designed for microcontroller applications. The range includes the Cortex-M3, Cortex-M4, Cortex-M0, Cortex-M0+, and Cortex-M1 processors. The Cortex-M1 processor is targeted at implementation in FPGA devices.

Cortex-M processors differ from other ARM processors, including the Cortex-A/R processors, because they only execute Thumb instructions. They are based on the ARMv7-M and ARMv6-M architectures and have an efficient instruction pipeline, with low-latency *Interrupt Service Routine* (ISR) entry and exit. The Cortex-M3 and Cortex-M4 processors also include hardware divide and *Multiply Accumulate* (MAC) operations.

In addition to the CPU core, the Cortex-M processors include a number of components that have a consistent architectural memory map. These include:

- *Nested Vectored Interrupt Controller* (NVIC)
- optional *Memory Protection Unit* (MPU)
- System Tick Timer
- debug and trace.

1.2 Memory barriers

Memory barriers are defined in the ARM architecture and are implemented by all processors in the ARM Cortex family. [Table 2](#) shows the Cortex-M processors covered by this document:

Table 2 Cortex-M processors covered by this document

Processor	Revision
Cortex-M0	r0p0
Cortex-M0+	r0p0
Cortex-M3	r0p0 to r2p1
Cortex-M4	r0p0 to r0p1
Cortex-M1	r0p0 to r1p0

Memory barriers are essential in high-performance processor implementations where instructions can be executed out of programmed order. Memory barriers might also be implemented in low-end processors.

Memory barriers can be triggered by hardware operations within the processor or by memory barrier instructions. This document describes the memory barrier instructions available in the Cortex-M processors, how they are related to the memory architecture, and when these instructions are required. [Case-by-case details on page 20](#) explains the differences between the architectural requirements and the specific Cortex-M processor implementations that can be more relaxed.

1.3 The need for memory barriers

The ordering of data transfers within a memory system can be different from the programmed transfer sequence. This does not mean that there is a system error or that, for example, a C compiler has generated incorrect code. In most cases, the C compiler might have cached some data in registers, or reordered some operations to permit the program to run faster.

For peripheral accesses, the peripheral registers must be declared with the `volatile` keyword to prevent the C compiler from caching the data. In this way, each time a volatile register is accessed, the processor will generate a corresponding transfer on the bus interface. However, problems might still occur if you define the peripheral address space as cacheable.

For most simple microcontrollers, if all peripheral devices are declared as volatile, and their memory spaces are defined as non-cacheable, then no further action is required to maintain memory access ordering. However, as microcontroller products become more sophisticated, these simple arrangements are not sufficient. For example:

- processors can re-order memory transfers to improve performance if it does not affect the behavior of the instruction sequence
- processors can have multiple bus interfaces that have different wait-states
- processors can have write buffers, and some write buffers can merge multiple transfers into a single transfer.

You might also have to consider that:

- memory, or devices in the memory map, can be presented on different branches of the internal bus
- some memory accesses can be speculative
- write buffers can be present on the system bus
- some instructions can be restarted, for example, multiple load/store instructions including LDM, STM.

You can use memory barrier instructions to aid ordering between memory accesses, or between memory accesses and other operations, when the operation sequences require such ordering to be preserved. Without memory barrier instructions, it is possible to have race conditions between steps of an operation that can cause errors in applications.

A memory barrier instruction:

- is required between two memory accesses if an external observer, for example, another processor, uses the related data
- can be required between a memory operation and an interrupt
- can be required between a memory operation and an event
- can be required between a memory operation and a system control operation.

For example, in the ARM Cortex-M processors, you can also use the *Data Memory Barrier* (DMB) instruction to ensure that the affect of a memory access takes effect before the next operation.

Figure 1 shows the DMB instruction being used to ensure memory ordering by stalling the pipeline.

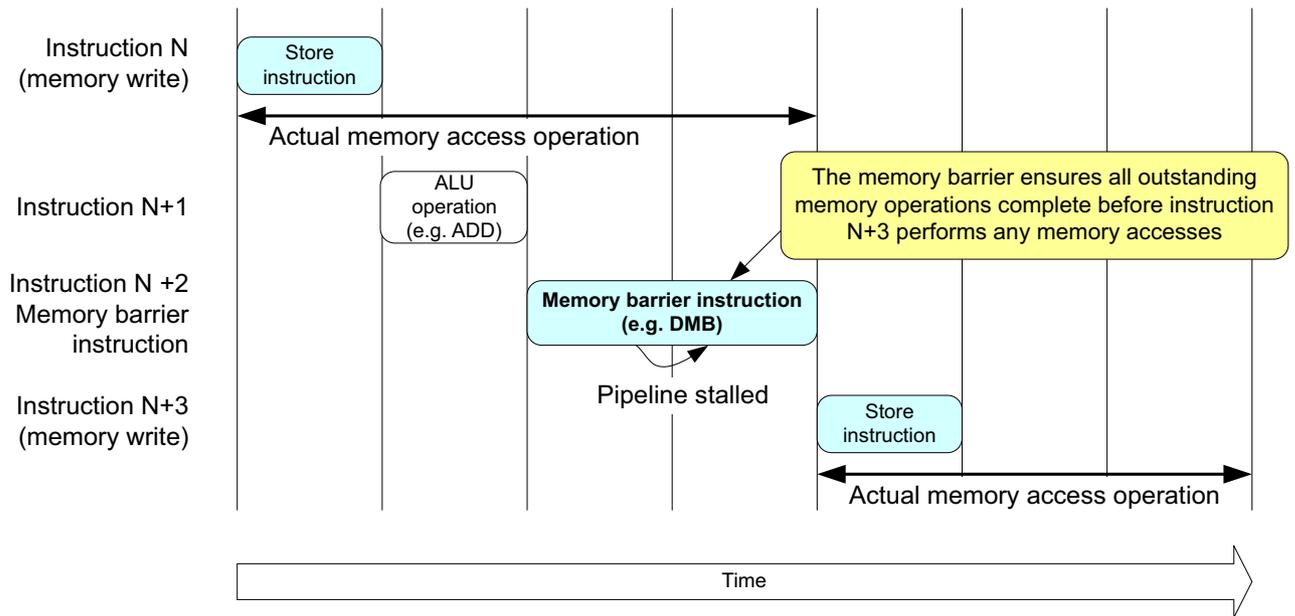


Figure 1 Data Memory Barrier instruction stalling the pipeline

2 Memory Type and Memory Ordering

This section gives an overview of memory types and the memory ordering model supported by the ARMv7-M and ARMv6-M architectures. This section contains the following sub-sections:

- [Overview of memory types](#)
- [Memory ordering on page 7](#)
- [Memory ordering restrictions on page 8](#)

2.1 Overview of memory types

In the ARMv7-M architecture, as implemented in the Cortex-M3 and Cortex-M4 processors, and in the ARMv6-M architecture, as implemented in the Cortex-M0 and Cortex-M0+ processors, there are three mutually exclusive memory types specified. These are:

- Normal
- Device
- Strongly-ordered.

Typically, memory used for program code and for data storage is Normal memory. System peripherals (I/O locations) generally conform to different access rules to Normal memory. Examples of I/O accesses are:

- interrupt controller registers where an access can be used as an interrupt acknowledge, changing the state of the controller
- memory controller configuration registers that are used to set up the timing and correctness of areas of Normal memory
- memory mapped peripherals, where accessing a memory location can cause system side-effects.

In ARMv7 including ARMv7-M, and in ARMv6-M regions of the memory map, accesses to system peripherals are defined as Device or Strongly-ordered memory accesses, and are more restrictive than accesses to Normal memory. That is:

- both read and write can cause system side-effects
- accesses must not be repeated, for example, on return from an exception
- the number, order, and sizes of the accesses must be maintained.

Within each memory type, the regions can be further divided by allocating different memory attributes, such as, access permission and cacheability.

Strongly-ordered memory

The Strongly-ordered memory type defines memory locations where an access to the location can cause system side-effects, or where the value returned for a load can vary depending on the number of loads performed.

Examples of memory regions normally marked as being Strongly-ordered are memory-mapped peripherals and I/O locations. In Cortex-M processors, the *System Control Space* (SCS) is Strongly-ordered. This memory space covers the NVIC, MPU, SysTick timer and debug components.

For explicit accesses from the processor to memory marked as Strongly-ordered:

- an access occurs in the programmed size
- the number of accesses is the number specified by the program.

An implementation must not perform more accesses to a Strongly-ordered memory location than are specified by a simple sequential execution of the program, except as a result of an exception occurring during load multiple and store multiple operations. For more information see *Exceptions in Load Multiple and Store Multiple operations* in the *ARMv7-M Architecture Reference Manual* (ARM DDI 0403).

Address locations in Strongly-ordered memory are not held in a cache, and are always treated as Shareable memory locations, that is, the memory system will have to maintain coherency of the data to permit the data to be shared by multiple processors.

All explicit accesses to Strongly-ordered memory must correspond to the ordering requirements described in *Memory ordering*.

2.2 Memory ordering

The ARMv7-M and ARMv6-M architectures support a wide range of implementations, from low-end microcontrollers through to high-end, superscalar, *System on Chip* (SoC) designs. To do so, the architectures permit, and only mandate, a weakly-ordered memory model. This model defines the three memory types, each with different properties and ordering requirements.

To support high-end implementations, the architecture does not require the ordering of transactions between Normal and Strongly-ordered memory, and it does not mandate the ordering of load/store instructions with respect to either instruction prefetch or instruction execution.

The order of instructions in the program flow does not always guarantee the order of the corresponding memory transactions. This is because:

- the processor can reorder some memory accesses to improve efficiency, providing this does not affect the behavior of the instruction sequence
- the processor can have multiple bus interfaces
- memory or devices in the memory map can be on different branches of an interconnect
- some memory accesses are buffered or speculative.

When considering the memory ordering model at the application level, the key factor is that for accesses to Normal memory, barriers are required, in some situations, where the order of accesses must be controlled.

ARMv7-M and ARMv6-M define restrictions in the permitted ordering of memory accesses. These restrictions depend on the memory attributes of the accesses.

The terms *address dependency* and *control dependency* are used in describing the memory access ordering requirements.

Address dependency

An address dependency exists when the value returned by a read access is used to compute the address of a subsequent read or write access. The address dependency exists even if the value read by the first read access does not change the address of the second read or write access. This might be due to the implementation of the processor's interface, for example, if the value returned is masked off before it is used, or if it has no affect on the predicted address value for the second access.

Control dependency

A control dependency exists when the data value returned by a read access is used to determine the condition code flags, and the values of the flags are evaluated to determine the address of a subsequent read access. This address determination might be through conditional execution, or through the evaluation of a branch.

2.3 Memory ordering restrictions

Table 3 shows the memory ordering between two explicit accesses A1 and A2, where A1 occurs before A2 in program order. An access refers to a read or a write to the specified memory type. For example, Device access, Non-shareable refers to a read or write access to Non-shareable Device memory.

The meaning of the symbols used in Table 3 are:

- < Accesses must be globally observed in program order, that is, A1 must be globally observed strictly before A2.
- Accesses can be globally observed in any order provided that the requirements of uniprocessor semantics are maintained, for example, respecting dependencies between instructions in a single processor.

Note

Table 3 presents information from ARMv6-M Architecture Reference Manual, and Revision D of the ARMv7-M Architecture Reference Manual. This differs from previous revisions of the ARMv7-M architecture, but aligns with the current ARMv7-A/R architecture definition that removes the ordering requirement between Strongly-ordered and Normal memory transactions. This arises from the ability for Normal transactions to be executed speculatively, and provides the capability for higher performance implementations to re-order Normal memory accesses around a Strongly-ordered one unless explicitly prevented from doing so via barriers.

Table 3 Memory ordering restrictions

		A2			
		Normal access	Device access, Non-shareable	Device access, Shareable	Strongly-ordered access
A1	Normal access	-	-	-	-
	Device access, Non-shareable	-	<	-	<
	Device access, Shareable	-	-	<	<
	Strongly-ordered access	-	<	<	<

The memory accesses in [Table 3 on page 8](#) that are designated by the - symbol have the following additional restrictions:

- If there is an address dependency then the two memory accesses are observed in program order.
- This ordering restriction does not apply if there is only a control dependency between the two read accesses.
- If there is an address dependency and a control dependency between two read accesses the ordering requirements of the address dependency apply.
- If the value returned by a read access is used for a subsequent write access, then the two memory accesses are observed in program order.
- An observer cannot view a write access to a memory location if that location would not be written to in a sequential execution of a program. An example of an observer would be a peripheral or a second processor.
- An observer cannot view a value written to a memory location if that value would not be written in a sequential execution of a program.

The Cortex-M processors never perform memory accesses out of order compared to instruction flow, however, the architecture does not prohibit this in future implementations. ARMv7-M code written to be portable to ARMv7-AR processors, like Cortex-A9, must already take account of this ordering model.

In Cortex-M processors, the memory types are linked to the memory map. A number of architectural regions are defined and each has a predefined memory type. You can also change some of the memory type definitions by programming the MPU when one is implemented.

It is possible to use the MPU on the Cortex-M processors to define an SRAM as a Device or Strongly-ordered region, however, this reduces the performance and does not prevent the C compiler from re-ordering the data transfers in the generated code. For best efficiency, ARM recommends that SRAM is defined as Normal memory and you use memory barriers in situations where memory ordering is important.

3 Memory barrier instructions

This section describes the available memory barrier instructions and their typical usage. The section contains the following sub-sections:

- [Overview](#)
- [Typical usages on page 11](#)
- [Architectural Requirements on page 11](#)
- [Architectural and implementation differences on page 14](#)
- [Cortex-M processor implementation specific requirements on page 14](#)
- [System implementation requirements on page 17](#)

3.1 Overview

Memory barrier is the general term applied to an instruction, or sequence of instructions that forces synchronization events by a processor with respect to retiring load/store instructions. A memory barrier is used to guarantee:

- completion of preceding load and store accesses by the programmers' model
- flushing of any prefetched instructions before the memory barrier event.

Both ARMv7-M and ARMv6-M architectures provide three explicit memory barriers to support the memory order model. The memory barrier instructions are:

Data Memory Barrier (DMB)

Ensures that all explicit data memory transfers before the DMB are completed before any subsequent data memory transfers after the DMB starts.

Data Synchronization Barrier (DSB)

Ensures that all explicit data memory transfer before the DSB are complete before any instruction after the DSB is executed.

Instruction Synchronization Barrier (ISB)

Ensures that the effects of all context altering operations prior to the ISB are recognized by subsequent instructions. This results in a flushing of the instruction pipeline, with the instruction following the ISB being re-fetched.

In addition to the ISB instruction, the architecture defines exception entry and exception return to also have ISB semantics, causing a fresh fetch of instructions and a re-evaluation of interrupts.

In C programming, barrier instructions can be generated using functions as defined in the *Cortex Microcontroller Software Interface Standard (CMSIS)* or using intrinsic functions provided by the various C compilers. The corresponding intrinsic functions are shown in [Table 4](#).

Table 4 Memory barrier intrinsic

Memory Barrier	CMSIS functions	C compiler intrinsic ^a Keil MDK-ARM, DS-5 and RVDS
DMB	<code>_DMB();</code>	<code>_dmb(0xF);</code>
DSB	<code>_DSB();</code>	<code>_dsb(0xF);</code>
ISB	<code>_ISB();</code>	<code>_isb(0xF);</code>

a. Input parameter 0xF is used to specify a full system barrier operation.

Note

Full system barrier operation ensures barrier operations for both read and write, and applies to Inner and Outer Cacheable memory systems, and to Shareable and Non-shareable memories.

3.2 Typical usages

Typical usages of the three memory barrier instructions are described in the following sections.

Data Memory Barrier (DMB)

The DMB instruction ensures that all explicit data memory transfers before the DMB are completed before any subsequent explicit data memory transactions after the DMB starts. This ensures correct ordering between two memory accesses.

The use of DMB is rarely needed in Cortex-M processors because they do not reorder memory transactions. However, it is needed if the software is to be reused on other ARM processors, especially multi-master systems. For example:

- DMA controller configuration. A barrier is required between a CPU memory access and a DMA operation.
- Semaphores in multi-core systems.
- Mailbox in multi-core systems.

In an application using CMSIS, you can use the `_DMB()` function to insert the DMB instruction.

Data Synchronization Barrier (DSB)

The DSB instruction ensures all explicit data transfers before the DSB are complete before any instruction after the DSB is executed. In Cortex-M processors, it can be used to:

- ensure effects of an access to SCS take place before the next operation
- ensure memory is updated before the next operation, for example, SVC, WFI, WFE.

In an application using CMSIS, you can use `_DSB()` function to insert the DSB instruction.

Instruction Synchronization Barrier (ISB)

The ISB instruction flushes the pipeline in Cortex-M processors and ensures effects of all context altering operations prior to the ISB are recognized by subsequent operations. It should be used after the CONTROL register is updated.

In an applications using CMSIS, you can use the `_ISB()` function to insert the ISB instruction.

3.3 Architectural Requirements

To meet the architectural requirements, the memory barrier instructions might be required in a number of situations.

System Control Space (SCS) Programming

The architecture defines the SCS as Strongly-ordered memory. In addition to the rules for the behavior of Strongly-ordered memory, the architecture requires that the side-effects of any access to the SCS that performs a context-altering operation take effect when the access completes. Software can issue a DSB instruction to guarantee completion of a previous SCS access.

The architecture guarantees the visibility of the effects of a context-altering operation only for instructions fetched after the completion of the SCS access that performed the context-altering operation. Executing an ISB instruction, or performing an exception entry or exception return, guarantees the re-fetching of any instructions that have been fetched but not yet executed.

To guarantee that the side effects of a previous SCS access are visible, software can execute a DSB instruction followed by an ISB instruction.

When programming the MPU:

- Use a DSB instruction to ensure the effect of the update takes place immediately at the end of context switching.
- Use an ISB instruction to ensure the new MPU setting takes effect immediately after programming the MPU if the MPU configuration code was accessed using a branch or call. If the effects of the MPU configuration are not required until after an exception return, then the exception return can be used in place of an explicit ISB.

The programming of interrupt priorities and enables can be done while all of the interrupts are disabled. This can be achieved by setting PRIMASK before the set up takes place and then clearing PRIMASK after the set up is complete. Follow the CPSIE/MSR with an ISB to ensure the recognition of the set up for subsequent instructions.

When dynamically changing an exception's priority include a DSB after the store to ensure that the result is visible.

A DSB should be used after programming control registers, such as the *System Control Register* (SCR), when the side-effect is needed immediately. For example, a DSB is needed between the write to the SCS and the next operation if:

- the SLEEPDEEP setting in the SCR is changed, followed by either:
 - enter sleep with WFI/WFE
 - an exception exit, with Sleep-on-exit enabled.
- the SLEEPONEXIT setting in the SCR is changed just before an exception exit.

CPS and MSR instructions

In addition to barriers, the CPSID instruction is also self synchronizing within the instruction stream, and ensures that the effect of setting PRIMASK and/or FAULTMASK is visible on the following instruction, thus ensuring that the appropriate interrupt/exceptions are disabled.

CPSIE that performs interrupt enabling is not guaranteed to be self synchronizing, with the architecture guaranteeing the recognition of the change only following an ISB. The same is true for MSR to CONTROL for stack-pointer selection and privilege level.

CONTROL updates

When writing to the CONTROL register, an ISB instruction should be used to ensure the new configuration is used in subsequent instructions' operations. For example, this will ensure that the subsequent instructions are re-fetched with the correct privilege information if MSR is used to switch between privilege and user modes.

Sleep

It is not a requirement for the processor hardware to drain any pending memory activity before suspending execution to enter a sleep mode. Therefore, software has to handle this by adding barrier instructions if the sleep mode used could affect data transfer. A DSB should be used to ensure that there are no outstanding memory transactions prior to executing the WFI or WFE instruction.

Vector table changes

If the program changes an entry in the vector table, and then enables the corresponding exception, a DSB instruction should be used between these two operations. This ensures that if the exception is taken after being enabled the processor uses the new exception vector. If the updated vector table is required immediately, for example if an SVC immediately follows an update to the SVC table entry via a store, then a DSB is also required.

Self modifying code

If a program contains self-modifying code, use a DSB instruction followed by an ISB instruction immediately after the code modification in the program. This ensures that the update completes and that the subsequent instruction execution uses the updated program. If the code is only required after an exception return then the exception return itself is sufficient.

Memory Map modifications

If the system contains a memory map switching mechanism then use a DSB instruction after switching the memory map in the program. This ensures subsequent instruction execution uses the updated memory map, if the memory system makes the updated memory map visible to all subsequent memory accesses.

———— Note —————

An ISB or an exception entry/return is required to ensure that the subsequent instructions are fetched using the new memory map.

Semaphores in a multi-master system

Two accesses to differing addresses in normal memory, or to different devices in Device memory, or to two different memory types, with the exception of those that [Table 3 on page 8](#) shows, are not guaranteed to be ordered. If the order of any of these transactions is required for the purpose of semaphore communication in a multi-master system then a DMB instruction must be inserted, for example, between a payload and spin-lock.

External factors

It cannot be guaranteed that the memory barrier instruction knows the status of an access outside of the processor. If the access on the bus completes, but the effects are not immediate, then a DMB/DSB can complete before the effects take place. This could happen, for example, if there are write buffers in the system. The architecture does not specify the time for a peripheral to make the side-effects of a read or write visible to the system. If required, it is the responsibility of the system designer to provide a means of ensuring the update has taken place, for example, via a read.

3.4 Architectural and implementation differences

Although the architecture permits memory re-ordering to happen in many cases, in practice, the majority of simple processors do not re-order memory transfers. As a result, there is a difference in the requirements of the architecture and the processor's implementation requirements. For example, most application programs can run correctly on existing Cortex-M processors without using any memory barrier instructions, although the architecture requires the use of memory barriers in some situations. Also, in most scenarios the applications are not sensitive to the affect of missing memory barriers.

However, if the application is to be ported to high-end processors, the omission of memory barrier instructions might result in glitches in the applications. The use of memory barriers can also be important if the software will be ported to a system with multiple processors. For example, when handling semaphores in a multiple processor system, memory barrier instructions should be used to ensure the other processors in the system can observe the data changes in the correct order.

Note

ARM recommends software developers to develop software based on architectural requirements rather than processor specific behaviors. This ensures portability and reusability of the software code. Processor specific behaviors might also vary between different released versions.

3.5 Cortex-M processor implementation specific requirements

The Cortex-M processor type can affect memory ordering.

Cortex-M3 and Cortex-M4 implementations

The Cortex-M3 and Cortex-M4 processors implement the ARMv7-M architecture. Both processors have three stage pipelines, an optional MPU, and have multiple AHB Lite interfaces. Code that follows the ARMv7-M barrier and ordering rules is guaranteed to function on Cortex-M3, Cortex-M4 and all other ARMv7-M compliant implementations. The Cortex-M3 and Cortex-M4 ordering model is at the simpler end of a permitted ARMv7-M implementation and so, although ARM recommends that barriers are always implemented, it is possible to implement software without barriers in a number of cases because:

- all loads and stores always complete in program order, even if the first is buffered
- all Strongly-ordered load/stores are automatically synchronized to the instruction stream
- at most, one instruction plus a folded IT instruction executes at one time
- interrupt evaluation is performed on every instruction, except on the folded IT
- interrupt evaluation is performed between a CPSIE instruction and a following CPSID instruction
- NVIC side-effects complete less than two instructions after an NVIC load or store
- the maximum prefetch size is six instructions, plus a fetch of two instructions in progress on the bus
- Device and Normal load/stores may be pipelined
- MSR side-effects are visible after one further instruction is executed.

This simple implementation results in a reduced need for barriers in a number of situations when code is only intended to run on Cortex-M3 or Cortex-M4. All use of DMB is redundant due to the inherent ordering of all loads and stores on Cortex-M3 and Cortex-M4.

Both the ARM Cortex-M3 and Cortex-M4 implement optional bit-banding. Accesses to the bit-band alias, and associated bit-band region on these two processors, do not require an intervening DMB instruction.

Due to the instruction buffers in the Cortex-M3 and Cortex-M4, the pipeline operations are different from traditional RISC processors:

- up to two instructions can be fetched in one cycle because most instructions are 16-bit
- an instruction can be fetched several cycles ahead of the decode and execution stages.

Figure 2 shows an example Cortex-M3, Cortex-M4 pipeline scenario.

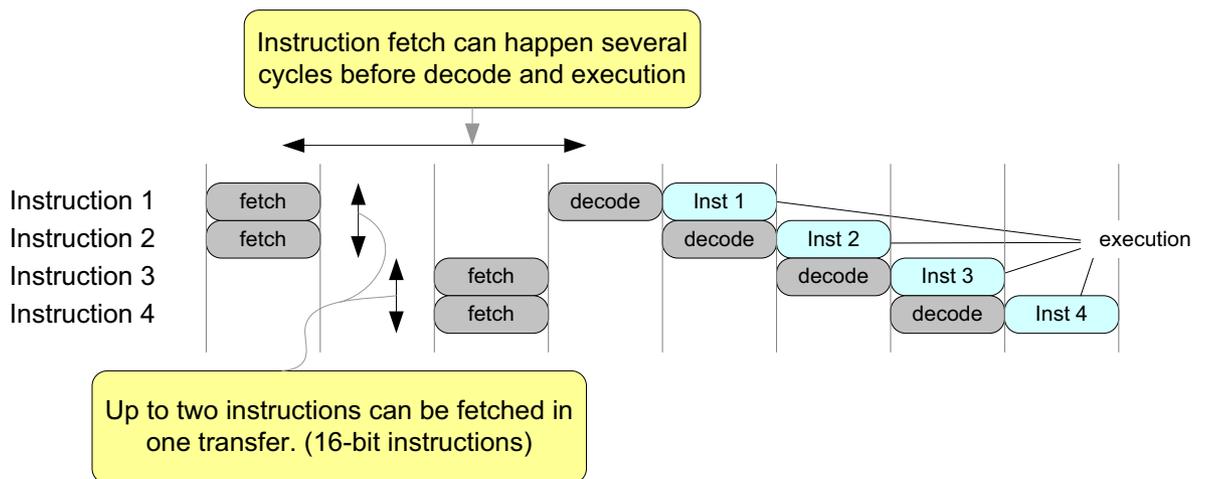


Figure 2 Pipeline stages in the Cortex-M3 and Cortex-M4 processors

More details about the Cortex-M3 and Cortex-M4 specific behaviors are covered in [Case-by-case details on page 20](#).

Cortex-M0 implementation

The ARM Cortex-M0 implements the ARMv6-M architecture. It has a three stage pipeline, and a single AHB Lite interface. Code that follows the ARMv6-M barrier and ordering rules is guaranteed to function on the Cortex-M0, Cortex-M0+, Cortex-M3, Cortex-M4, and all other ARMv6-M/ARMv7-M compliant implementations. The Cortex-M0 ordering model is at the simpler end of a permitted ARMv7-M implementation and so, although ARM recommends that barriers are always implemented, it is possible to implement software without barriers in a number of cases because:

- all loads and stores always complete in program order
- all Strongly-ordered load/stores are automatically synchronized to the instruction stream
- interrupt evaluation is performed on every instruction
- interrupt evaluation is performed between a CPSIE instruction and a following CPSID instruction
- NVIC side-effects complete after less than two instructions following an NVIC load or store

- the maximum prefetch size is two instructions, plus a fetch of two instructions in progress on the bus
- Device and Normal load/stores may be pipelined
- MSR side-effects are visible after one further instruction is executed.

This simple implementation results in a reduced need for barriers in a number of situations when code is only intended to ever run on the Cortex-M0. All use of DMB is redundant due to the inherent ordering of all loads and stores on the Cortex-M0.

Different from the Cortex-M3 and Cortex-M4, the Cortex-M0 has a smaller instruction buffer. However, it can also fetch up to two instructions in a cycle.

Figure 3 shows an example Cortex-M0 pipeline scenario.

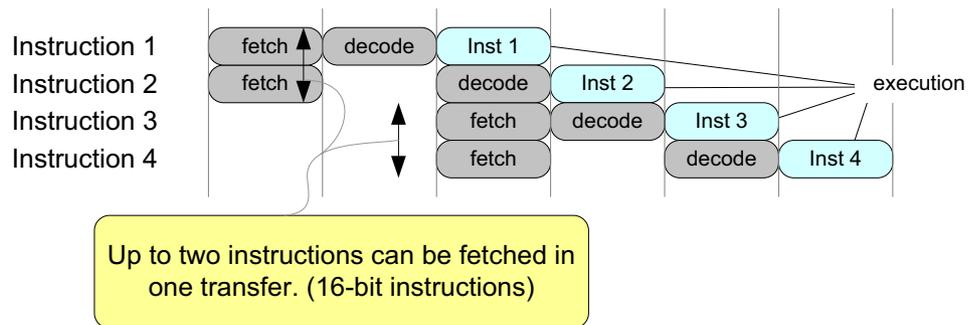


Figure 3 Pipeline stages in the Cortex-M0 processor

More details about the Cortex-M0 specific behaviors are covered in [Case-by-case details on page 20](#).

Cortex-M0+ implementation

The ARM Cortex-M0+ implements the ARMv6-M architecture. It has a two stage pipeline, and an AHB Lite interface and an optional IOPORT interface. Code that follows the ARMv6-M barrier and ordering rules is guaranteed to function on the Cortex-M0, Cortex-M0+, Cortex-M3, Cortex-M4 and all other ARMv6-M/ARMv7-M compliant implementations. The Cortex-M0+ ordering model is at the simpler end of a permitted ARMv7-M implementation and so, although ARM recommends that barriers are always implemented, it is possible to implement software without barriers in a number of cases because:

- all loads and stores always complete in program order
- all Strongly-ordered load/stores are automatically synchronized to the instruction stream
- interrupt evaluation is performed on every instruction
- interrupt evaluation is performed between a CPSIE instruction and a following CPSID instruction
- NVIC side-effects complete after less than two instructions following an NVIC load or store
- the maximum prefetch size including instructions fetching on the bus is two
- Device and Normal load/stores may be pipelined
- MSR side-effects are visible after one further instruction is executed.

This simple implementation results in a reduced need for barriers in a number of situations when code is only intended to ever run on the Cortex-M0+. All use of DMB is redundant due to the inherent ordering of all loads and stores on the Cortex-M0+.

The Cortex-M0+ has a smaller instruction buffer than the Cortex-M0, Cortex-M3, and Cortex-M4 processors.

Figure 4 shows an example Cortex-M0+ pipeline scenario.

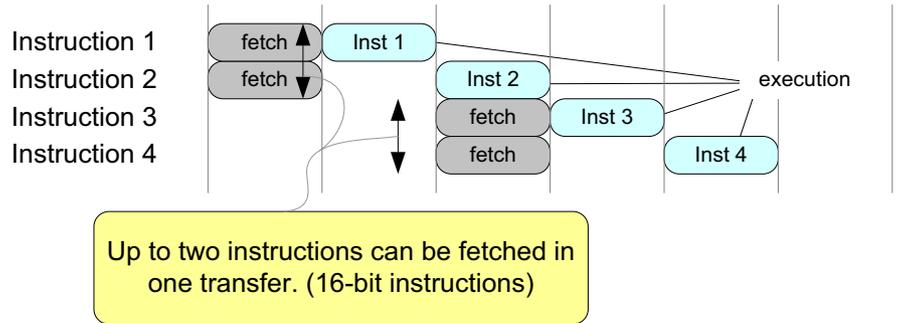


Figure 4 Pipeline stages in the Cortex-M0+ processor

3.6 System implementation requirements

As well as memory ordering at the processor level, system level design can also affect memory ordering and might require extra steps to ensure correctness of the program operations.

Write buffers

Write buffers might be present within the processor, or can be present in bus components, such as AHB to APB bridges, or memory interfaces. The advantage of a write buffer is that it can enhance system performance. By using a write buffer, during a memory write, subsequent instructions can be executed without waiting for the write operation to be completed.

Figure 5 shows locations where write buffers might be used and their effect.

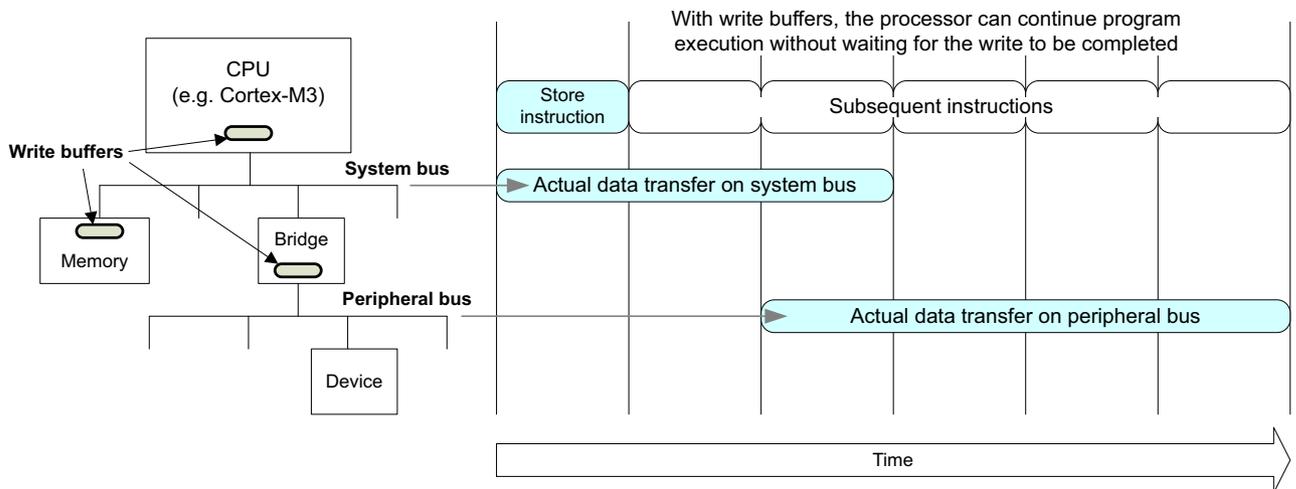


Figure 5 Locations where write buffers might be used and their effect

When a write buffer is present, it is possible to start another data transfer while the first one is still taking place. This might result in an error if the subsequent operations on the processor rely on the completion of the buffered write. In typical memory or peripheral accesses on the Cortex-M processors this is not a problem, because data can only be accessed with one bus path and no bus transaction reordering can take place within the bus system. But if the interaction between the two memory operations can be dependent in other ways, additional measures might be needed to ensure correct ordering of the operations.

The memory barrier instructions, DMB and DSB, can be used to ensure that the write buffer on the processor has completed its operation before subsequent operations can be started. However, it does not check the status of the bus level write buffers. In such cases, if the system is based on AHB or AHB Lite, you might need to perform a dummy read through the bus bridge to ensure that the bus bridge has completed its operation. Alternatively, a SoC design might have a device specific status register to indicate whether the bus system write buffer is idle.

———— **Note** ————

The Cortex-M0 processor (r0p0) and the Cortex-M0+ processor (r0p0) do not include a write buffer in their processor bus interface.

System level cache

The Cortex-M0, Cortex-M0+, Cortex-M1, Cortex-M3, and Cortex-M4 processors do not have any internal cache memory. However, it is possible for a SoC design to integrate a system level cache.

———— **Note** ————

A small caching component is present in the Cortex-M3 and Cortex-M4 processors to accelerate flash memory accesses during instruction fetches.

In a single processor system, the existence of a data cache is unlikely to cause a memory ordering issue, however, it might have implications for power saving modes. If there are multiple bus masters in the system, and if a data set is shared, then cache memory control and memory barriers will be required to ensure correct memory ordering is maintained. This is because a cache unit can re-order the update sequence of memory.

Figure 6 shows the reordering of a memory update sequence due to a cache component.

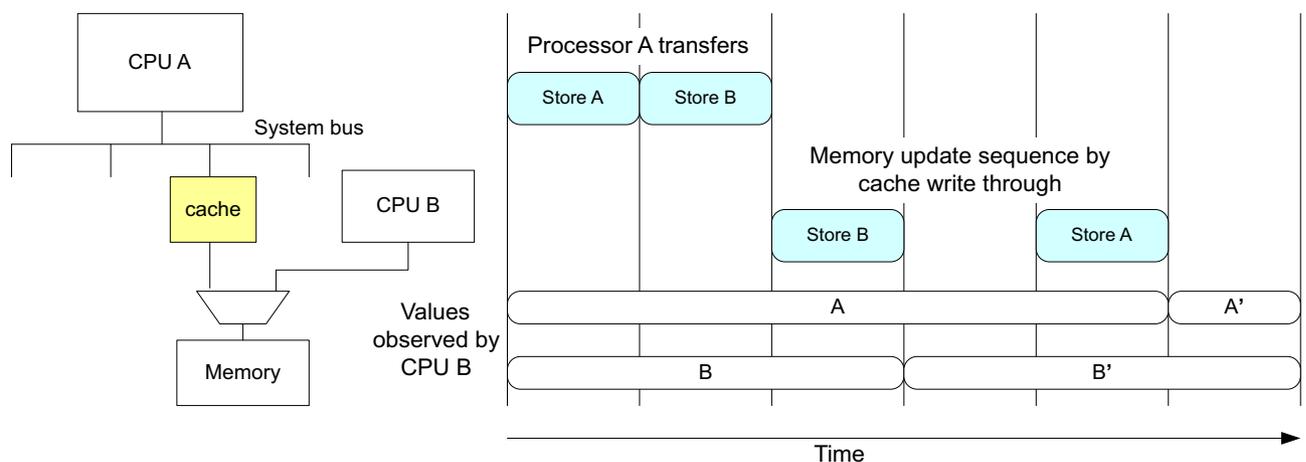


Figure 6 Reordering of a memory update sequence due to a cache component

Note

If multiple cache units are present, you will also require cache coherency support. Alternatively, data that is shared can be located in non-cacheable memory regions.

Hardware latency

Software errors or race conditions can also be caused by latency in hardware designs. Examples of hardware latency are the delay:

- between triggering interrupt generation at a peripheral by a memory access, to the time the processor actually received the interrupt
- between enabling the clock to a peripheral, to the time the peripheral can be safely accessed
- after switching memory map by writing to a system memory control register, to the time that the new memory map arrangement becomes valid.

Note

These cannot be resolved using memory barrier instructions and should be addressed during device specification.

4 Case-by-case details

This section describes the use of memory barriers in the Cortex-M processors on a case-by-case basis, it includes the following sub-sections:

- [Normal data access in memories](#)
- [Device \(Peripheral\) access](#)
- [Bit band access on page 21](#)
- [SCS peripheral accesses on page 21](#)
- [Enabling Interrupts using NVIC on page 24](#)
- [Disabling Interrupts using NVIC on page 27](#)
- [Enabling interrupts using CPS instructions and MSR instructions on page 28](#)
- [Disabling interrupts using CPS and MSR instructions on page 31](#)
- [Disabling interrupts at peripherals on page 32](#)
- [Changing priority level of an interrupt on page 33](#)
- [Vector table configuration - Vector Table Offset Register \(VTOR\) on page 36](#)
- [Vector table configuration on page 37](#)
- [Memory map change on page 39](#)
- [Entering sleep modes on page 41](#)
- [Self-reset on page 42](#)
- [CONTROL register on page 45](#)
- [MPU programming on page 46](#)
- [Multi-master systems on page 47](#)
- [Semaphores and Mutual Exclusives \(Mutex\) - uncore and multicore on page 49](#)
- [Self modifying code on page 50.](#)

4.1 Normal data access in memories

There is no need to use a memory barrier instruction between each access:

- Architecturally: A processor is allowed to re-order the data transfers as long as it does not affect the operation of the program.
- Implementation: In the Cortex-M processors data transfers are carried out in the programmed order.

4.2 Device (Peripheral) access

There is no need to use a memory barrier instruction between each step during peripheral programming or peripheral access:

- Architecturally: Access to the same device must be in the programmed order.
- Implementation: Cortex-M processors do not reorder data transfers.

If the programming sequence involves a number of different devices:

- Architecturally: A memory barrier is required when a different device is accessed and the programming order between the two devices can affect the result. This is because the bus structure might have different bus paths to each device and the bus paths might have different delays.
- Implementation: Cortex-M processors do not re-order data transfers and so do not require a memory barrier when a different device is accessed.

4.3 Bit band access

The bit band on the Cortex-M3 and Cortex-M4 processors is an implementation specific feature. It enables two parts of the memory map to be bit addressable by providing two bit band alias regions that enable bit data accesses:

- Architecturally: The bit band feature is not part of the ARMv7 or ARMv6 architectures, so there are no architecturally defined requirements on using memory barriers with bit band accesses.
- Implementation: The Cortex-M3 and Cortex-M4 processors handle bit band accesses, to bit band regions as well as bit band alias regions, in programmed order. There is no need to use a memory barrier.

The ARM Cortex-M0 and Cortex-M0+ processors do not have the bit band feature. It is possible to use a bus wrapper to add the bit band feature to the Cortex-M0 and Cortex-M0+ processors. In this case, the bus wrapper must retain correct memory ordering.

4.4 SCS peripheral accesses

SCS peripheral accesses, for example, NVIC and debug accesses, generally do not require the use of memory barrier instructions: There is no need to insert memory barrier instructions between each SCS access, or between an SCS access and a Device memory access.

- Architecturally: SCS is Strongly-ordered, and already exhibits the DMB behavior.
- Implementation: There is no need to insert memory barrier instructions between each SCS access, or between an SCS access and a Device memory access.

Figure 7 shows the architectural SCS access behavior.

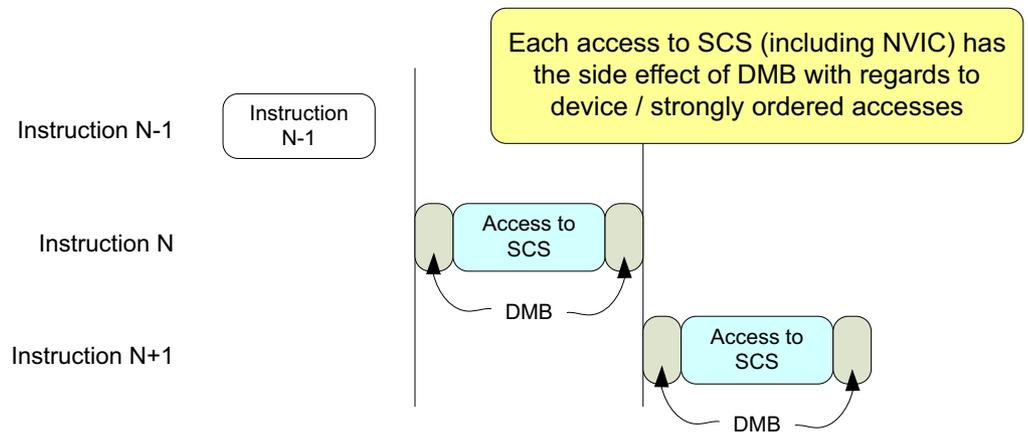


Figure 7 Architectural SCS access behavior

Architectural requirements

ARM recommends that the architectural requirements are adopted.

- if the effect of the SCS register-write needs to be visible immediately, a DSB is required
- there is no need to add memory barrier instructions between steps when programming a SCS device
- if the software requires that the next operation must not be carried out until the effect of the access has taken place, a DSB instruction must be used as shown in [Example 1](#).

Example 1 Ensuring the effect of an SCS register-write is visible immediately

```
SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk; /* Enable deepsleep */
__DSB(); /* Ensure effect of last store takes effect */
__WFI(); /* Enter sleep mode */
```

Or

```
void Device_IRQHandler(void) {
software_flag = 1; /* Update software variable used in thread */
SCB->SCR &= ~SCB_SCR_SLEEPONEXIT_Msk; /* Disable sleepexit */
__DSB(); /* Ensure effect of last store takes effect */
return; /* Exception return */}
```

———— Note ————

The DMB side-effect of SCS access does not ensure memory ordering when the program access is to Normal memory. If the program operation relies on the ordering between an access to a SCS location, and another access to a Normal memory location, then a memory barrier instruction like DMB or DSB is required. [Example 2](#) shows representative code for ordering accesses to SCS and Normal memory.

Example 2 Ensuring an access to SCS and Normal memory locations is ordered

```
STR R0, [R1] ; Access to a Normal Memory location
DMB ; Add DMB ensures ordering for ALL memory types
STR R3, [R2] ; Access to a SCS location
DMB ; Add DMB ensures ordering for ALL memory types
STR R0, [R1] ; Access to a Normal Memory location
```

For the same code, if [R1] is pointing to a Device or Strongly-ordered memory region, then the DMB is not required.

Implementation requirements

In existing Cortex-M processors, omission of the DMB, or a DSB instruction in [Example 2 on page 22](#) does not cause an error because the SCS in these processors contains DSB behavior already:

- In Cortex-M0, M0+ processors, the effect takes place immediately after the access completes. DSB is not strictly required after SCS accesses.
- In Cortex-M3 and M4 processors, the effect takes place immediately after the access completes. DSB is not strictly required after SCS accesses, accept for the special case of a SLEEPONEXIT update. See [Special case for the Cortex-M3 and Cortex-M4 processors](#).

[Figure 8](#) shows the implemented SCS access behavior.

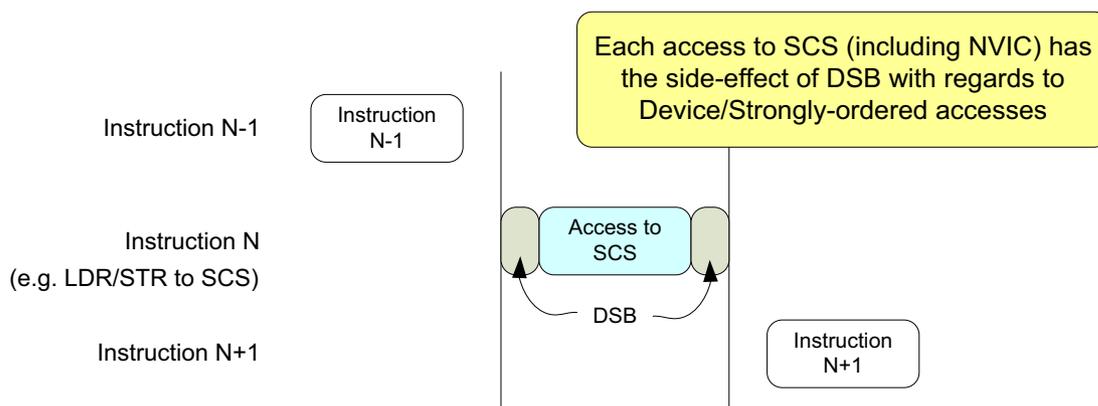


Figure 8 Implemented SCS access behavior

Therefore [Example 1 on page 22](#) can be simplified by omitting the DSB instruction without any resultant error, that is:

```
SCB->SCR |= SCB_SCR_SLEEPDEEP_Msk; /* Enable deepsleep */
__WFI(); /* Enter sleep mode */
```

Note

- The existing Cortex-M processors do not reorder any data transfers. As a result there is no need to use a DMB instruction.
- For the Cortex-M3 and Cortex-M4 processor, if the instruction after the SCS load/store is a NOP instruction, or a conditional instruction with condition failed (cc'failed), the NOP instruction or cc'failed instruction could execute in parallel to the SCS load/store instruction.

Special case for the Cortex-M3 and Cortex-M4 processors

If an exception handler disables the SLEEPONEXIT feature just before an exception return, a DSB instruction is needed after the SCR write, before the exception return.

4.5 Enabling Interrupts using NVIC

Normally NVIC operations do not require the use of memory barrier instructions.

[Example 3](#) shows example NVIC operations code.

Example 3 NVIC operations code

```
device_config(); // Setup peripheral
NVIC_ClearingPending(device_IRQn); // clear pending status
NVIC_SetPriority(device_IRQn, priority); // set priority level
NVIC_EnableIRQ(device_IRQn); // Enable interrupt
```

———— Note ————

Interrupts can enter pending state before they are enabled.

Architecturally, each SCS access has DMB behavior with regard to other Device or Strongly-ordered accesses.

[Figure 9](#) shows the architectural NVIC access behavior.

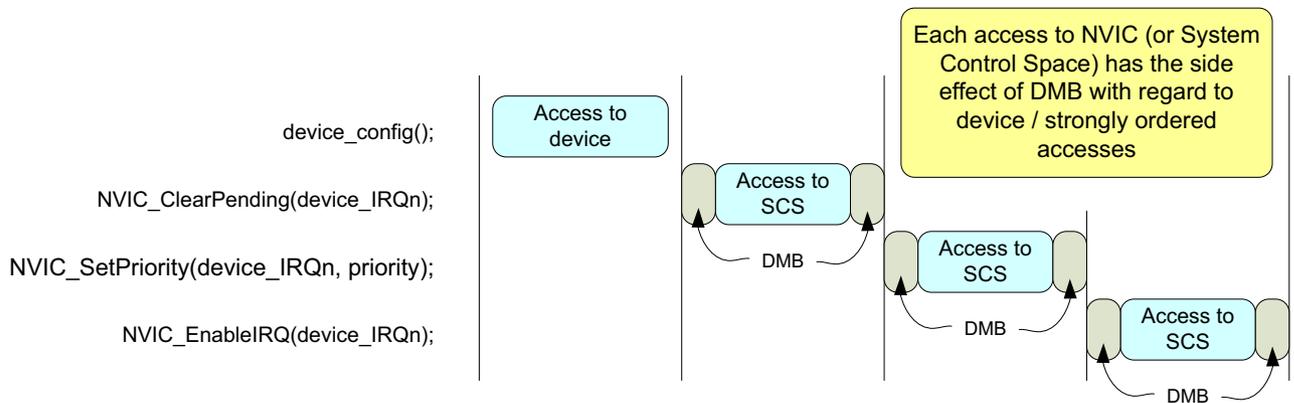


Figure 9 Architectural NVIC access behavior

In existing Cortex-M processor implementations, each SCS access has DSB instruction behavior with regard to other Device or Strongly-ordered accesses.

Figure 10 shows the implemented NVIC access behavior.

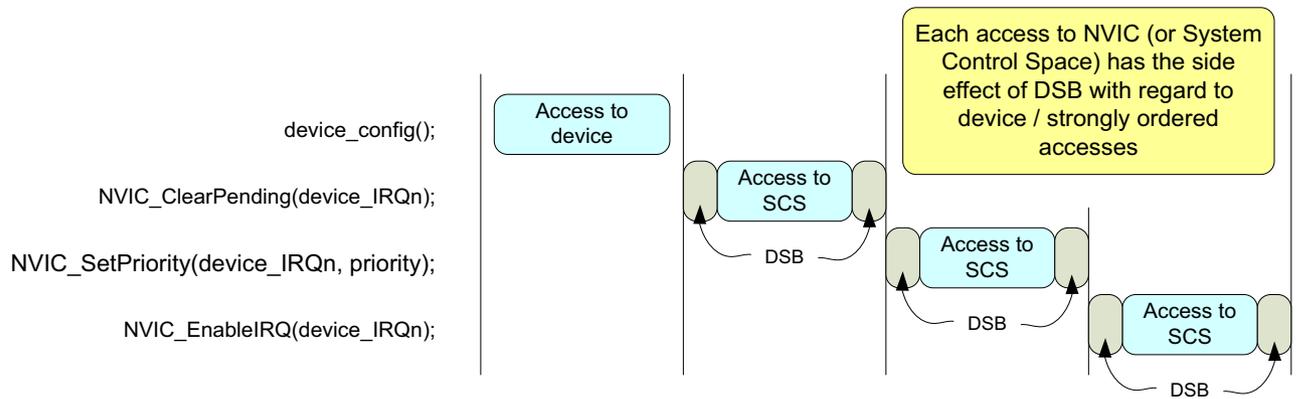


Figure 10 Implemented NVIC access behavior

For existing Cortex-M processors, due to the pipelined nature of the processor, if an interrupt was already in the pending state, the processor can execute up to two extra instructions after enabling the interrupt in the NVIC before executing the interrupt service routine.

Figure 11 shows that after a pended interrupt is enabled, the Cortex-M processors might execute additional instructions before entering the exception handler.

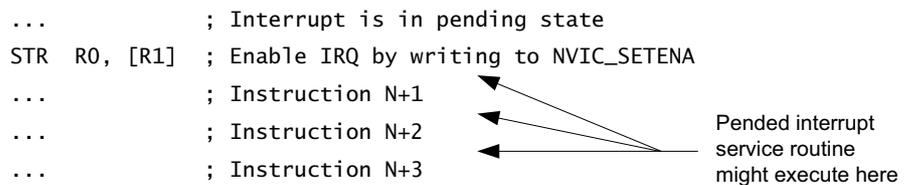


Figure 11 Implemented interrupt enabling delay in Cortex-M processors

Architectural requirements

ARM recommends that the architectural requirements are adopted.

Depending on the requirements in your application:

- memory barriers are not required for normal NVIC programming
- memory barriers are not required between NVIC programming and peripheral programming
- if a pended interrupt request needs to be recognized immediately after being enabled in the NVIC, add a DSB instruction and then an ISB instruction.

If the instruction after the interrupt depends on the result of the pended interrupt, then you should add memory barrier instructions.

Example 4 shows code for handling an interrupt.

Example 4 Interrupt handling code

```
LDR R0, =0xE00E100 ; NVIC_SETENA address
MOVS R1, #0x1
STR R1, [R0] ; Enable IRQ #0
DSB ; Ensure write is completed
; (architecturally required, but not strictly
; required for existing Cortex-M processors)
ISB ; Ensure IRQ #0 is executed
CMP R8, #1 ; Value of R8 dependent on the execution
; result of IRQ #0 handler
```

If the memory barrier instructions are omitted, then the CMP instruction is executed before the interrupt has taken place.

Figure 12 shows how a program error might be introduced if the program expects the interrupt to take place immediately.

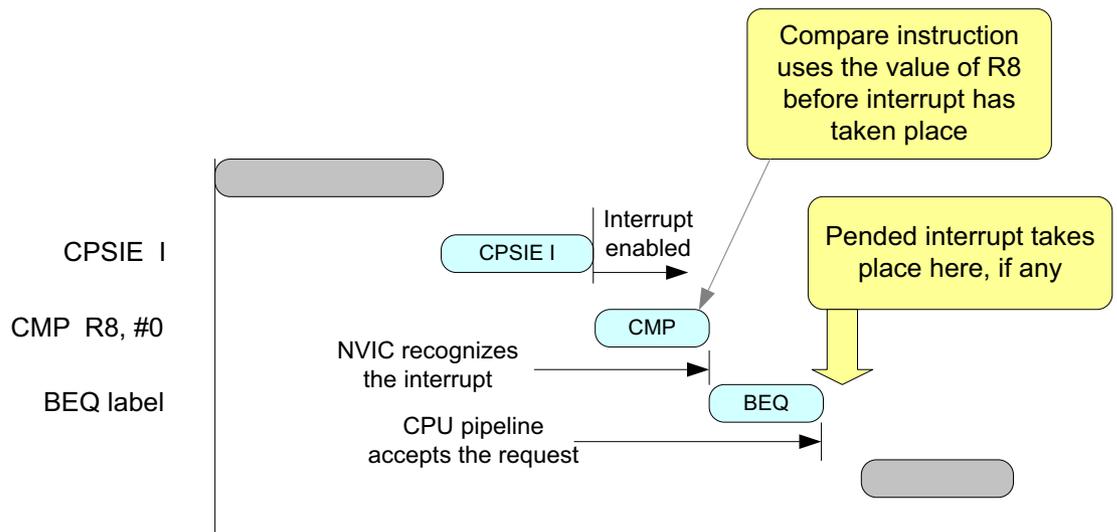


Figure 12 Program error due to the interrupt not taking place immediately

To prevent a program error, application code that relies on immediate recognition of the interrupt must have the memory barrier instructions added.

Implementation requirements

Depending on the requirements in your application:

- memory barriers are not required for normal NVIC programming
- if a pended interrupt request must be recognized immediately after being enabled in the NVIC, add an ISB instruction.

———— Note ————

Since a NVIC access already has DSB instruction behavior, omitting the DSB instruction still permits an enabled and pended interrupt to be recognized immediately.

4.6 Disabling Interrupts using NVIC

In common application scenarios there is no need to use memory barriers. For example:

```
NVIC_EnableIRQ(device_IRQn); // Disable interrupt
```

Due to the processor pipeline, the Cortex-M processors can be entering the interrupt sequence at the same time as writing to the NVIC to disable the interrupt. Therefore, it is possible that an interrupt handler might be executed immediately after it is disabled at the NVIC.

Figure 13 shows the implemented interrupt disabling delay in Cortex-M processors.

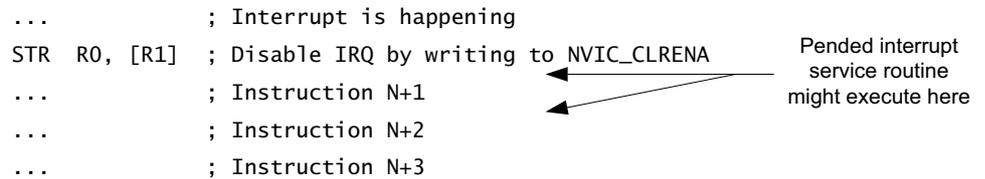


Figure 13 Implemented interrupt disabling delay in Cortex-M processors

Architectural requirement

ARM recommends that the architectural requirement is adopted.

Depending on the requirements in your application:

- memory barriers are not required for normal NVIC programming when disabling an IRQ
- memory barriers are not required between NVIC programming and peripheral programming
- if it is necessary to ensure an interrupt will not be triggered after disabling it in the NVIC, add a DSB instruction and then an ISB instruction.

Example 5 shows example code for changing an interrupt vector.

Example 5 Changing an interrupt vector

```

#define MEMORY_PTR(addr) (*((volatile unsigned long *)(addr)))
NVIC_DisableIRQ(device_IRQn); // Disable interrupt
__DSB();
__ISB();
MEMORY_PTR(SCB->VTOR+0x40+(device_IRQn<<2))=
(void) device_Handler; // Change vector to a different one
  
```

Implementation requirements

Depending on the requirements in your application:

- memory barriers are not required for normal NVIC programming when disabling an IRQ
- memory barriers are not required between NVIC programming and peripheral programming
- if it is necessary to ensure an interrupt will not be triggered after disabling it in the NVIC, add an ISB instruction.

4.7 Enabling interrupts using CPS instructions and MSR instructions

In normal applications there is no need to add any barrier instruction after using a CPS instruction to enable an interrupt:

```
_enable_irq(); /* Compiles to "CPSIE I" - Clear PRIMASK */
```

If an interrupt was already in the pending state, the processor accepts the interrupt after the "CPSIE I" is executed. However, additional instructions can be executed before the processor enters the exception handler:

- for Cortex-M3 or Cortex-M4, the processor can execute up to TWO additional instructions before entering the interrupt service routine
- for Cortex-M0, the processor can execute up to ONE additional instruction before entering the interrupt service routine.

Figure 14 shows the implemented interrupt enabling delay in the Cortex-M3 and Cortex-M4 processors.

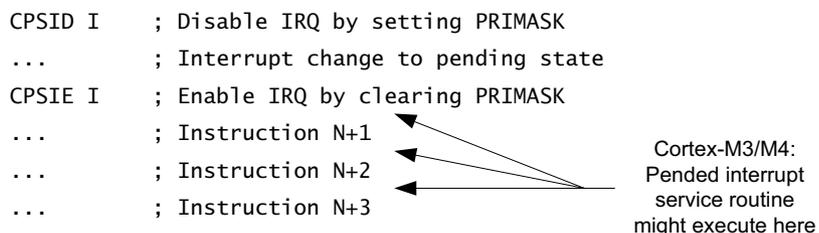


Figure 14 Implemented interrupt enabling delay in the Cortex-M3 and Cortex-M4 processors

Figure 15 shows the implemented interrupt enabling delay in the Cortex-M0 and Cortex-M0+ processors.

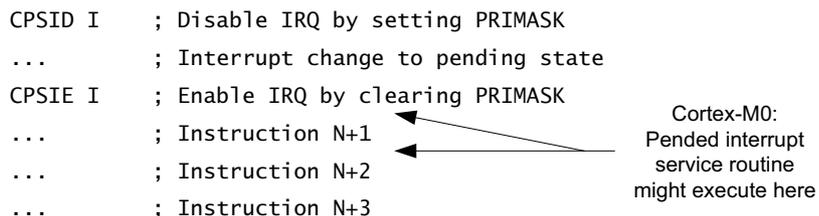


Figure 15 Implemented interrupt enabling delay in the Cortex-M0 and Cortex-M0+ processors

Architectural requirements

ARM recommends that the architectural requirements are adopted.

- If it is necessary to ensure a pended interrupt is recognized before subsequent operations, the ISB instruction should be used after CPSIE I. Figure 16 shows the use of the ISB instruction after enabling interrupts to permit immediate recognition of the pending interrupt.
- If it is not necessary to ensure that a pended interrupt will be recognized immediately before subsequent operations, it is not necessary to insert a memory barrier instruction.
- Between two time critical tasks, if you want to permit a pended interrupt to take place, you can use an ISB instruction as follows:

```
__enable_irq(); // CPSIE I : Enable interrupt
__ISB(); // Allow pended interrupts to be recognized
__disable_irq(); // CPSID I : Disable interrupt
```

Figure 16 shows the resulting behavior.

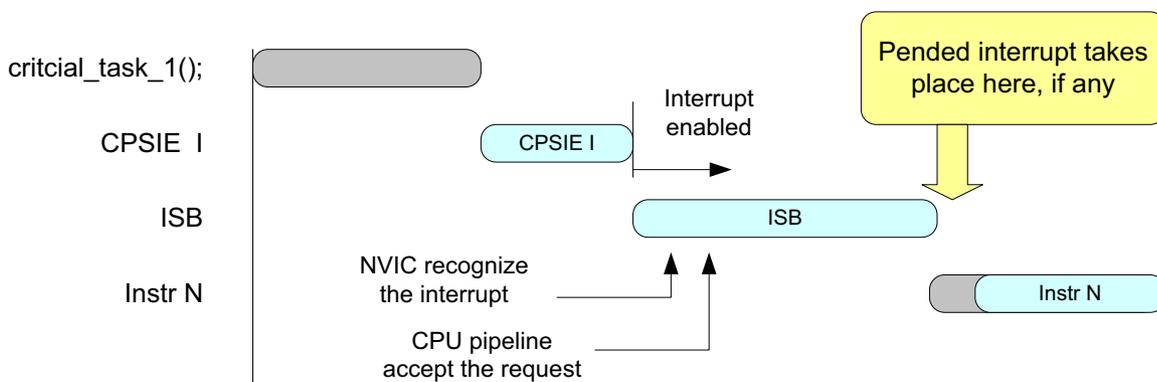


Figure 16 Use ISB after enabling interrupts to permit immediate recognition of a pending interrupt

A suitable architectural coding is:

```
__disable_irq(); /* CPSID I */
critical_task1(); /* Time critical task 1 */
__enable_irq(); /* CPSIE I */
__ISB();
__disable_irq(); /* CPSID I */
critical_task2(); /* Time critical task 2 */
__enable_irq(); /* CPSIE I */
```

Interrupt can take place here

Figure 17 shows the resulting behavior.

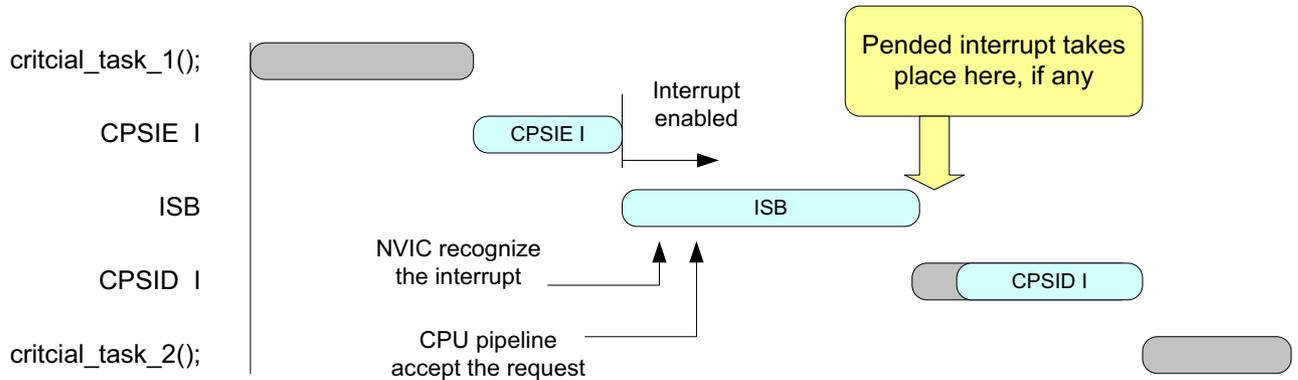


Figure 17 Architectural interrupt behavior between CSPIE and CPSID

———— **Note** ————

The same requirement applies when using the MSR instruction to enable interrupts.

Implementation requirements

In Cortex-M processors:

- If it is necessary to ensure a pended interrupt is recognized before subsequent operations, the ISB instruction should be used after CPSIE I. This is the same as the architectural requirement, see [Figure 16 on page 29](#).
- If it is not necessary to ensure that a pended interrupt is recognized immediately before subsequent operations, it is not necessary to insert a memory barrier instruction.
- An exception to this rule is the sequence CPSIE followed by CPSID. In Cortex-M processors, there is no need to insert an ISB between CPSIE and CPSID.

A suitable implementation coding is:

```

__disable_irq(); /* CPSID I */
critical_task1(); /* Time critical task 1 */
__enable_irq(); /* CPSIE I */
__disable_irq(); /* CPSID I */
critical_task2(); /* Time critical task 2 */
__enable_irq(); /* CPSIE I */
    
```

Interrupt can take place here

Figure 18 shows the implemented behavior.

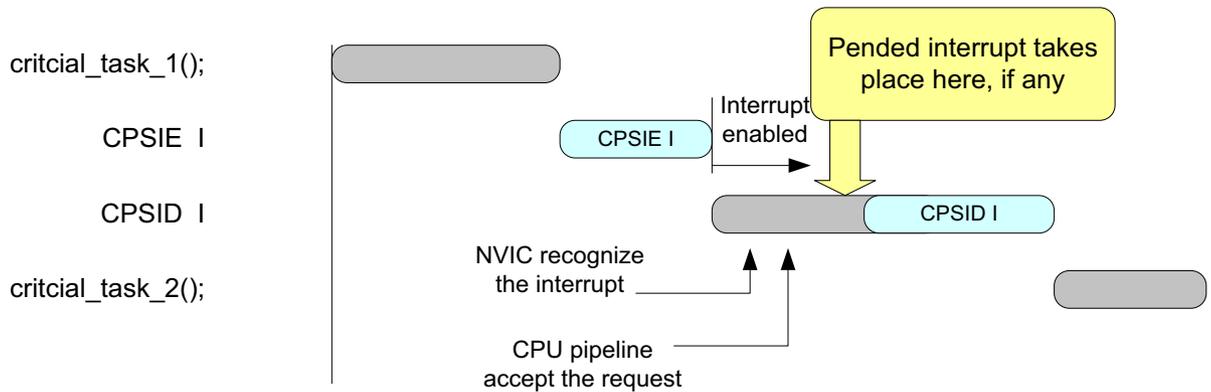


Figure 18 Implemented behavior requires no ISB between CPSIE and CPSID

Note

As the implementation requirements show, there is no need to add a memory barrier instruction between `__enable_irq()` and `__disable_irq()`. However, in the architecture, if the interrupt needs to be recognized between the `CPSIE` and `CPSID` instructions, then an `ISB` instruction is needed. The same applies when using the `MSR` instruction to enable interrupts.

4.8 Disabling interrupts using `CPS` and `MSR` instructions

The `CPSID` instruction is self-synchronized to the instruction stream and there is no requirement to insert memory barrier instructions after `CPSID`.

Architectural requirements

ARM recommends that these architectural requirements are adopted.

Memory barrier instruction is not required.

Implementation requirements

Memory barrier instruction is not required.

Note

The same applies when using a `MSR` instruction to disable interrupts.

4.9 Disabling interrupts at peripherals

Disabling of an interrupt at a peripheral might take additional time because of the various possible sources of delay in the system.

Figure 19 shows various sources of delay.

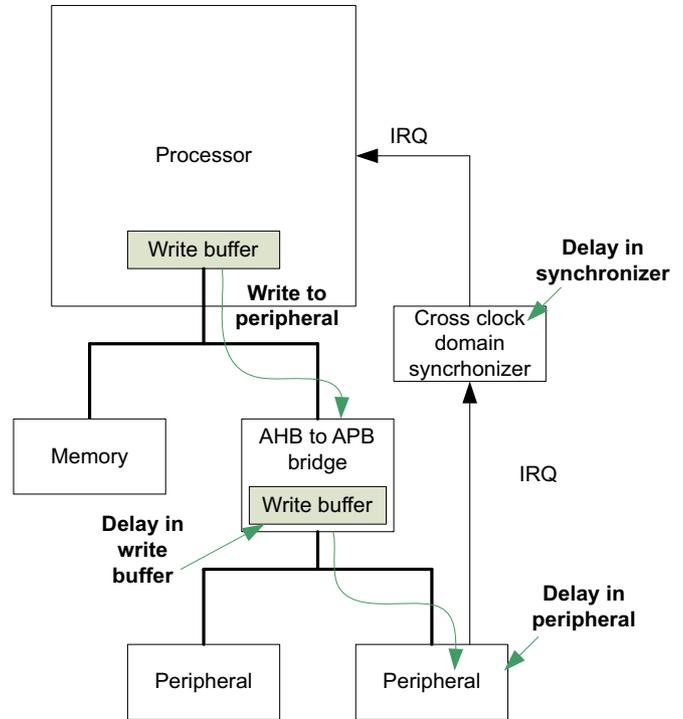


Figure 19 Various sources of delay during interrupt disabling

You might receive an interrupt request from a disabled peripheral, for a short period of time, immediately after the peripheral is disabled.

Architectural requirements

IMPLEMENTATION DEFINED, there are no architectural requirements.

Implementation requirements

The delay is device dependent. For most cases, if the delay in the IRQ synchronizer is small, you can use the sequence:

1. Disable the peripheral interrupt by writing to its control register.
2. Read back the control register of the peripheral to ensure it has been updated.
3. Disable the IRQ in the NVIC.
4. Clear the IRQ pending status in the NVIC.
5. Read back the IRQ pending status. If IRQ pending is set, clear the IRQ status at the peripheral, and clear the IRQ pending status in the NVIC again. This step must be repeated until the NVIC IRQ pending status remains cleared.

This sequence will successfully disable interrupts for most of the simple microcontroller devices. However, due to the various delay factors that might occur within the system, ARM recommends you contact your chip supplier or manufacturer to get definitive recommendations.

4.10 Changing priority level of an interrupt

The priority level settings are determined by Priority Level registers in the NVIC that is part of the SCS.

For Cortex-M3 or Cortex-M4 processors, the priority level can be changed dynamically.

For ARMv6-M processors, for example, the Cortex-M0 or Cortex-M0+, the dynamic changing of the priority level of enabled interrupts or exceptions is not supported. You should setup the priority levels before enabling the interrupts.

Architectural requirements

ARM recommends that the architectural requirements are adopted.

Architecturally, the SCS is Strongly-ordered. Therefore, normal NVIC configurations do not require a memory barrier. However, DSB and ISB instructions should be inserted after changing the interrupt priority if the interrupt is already enabled and it is necessary that the interrupt, if pending, executes at the new priority level.

———— **Note** ————

On ARMv6-M processors, you should only change the priority level of an interrupt when the interrupt is disabled, otherwise the result is UNPREDICTABLE.

Figure 20 shows the architectural recommendation if changing a priority level and then immediately triggering it.

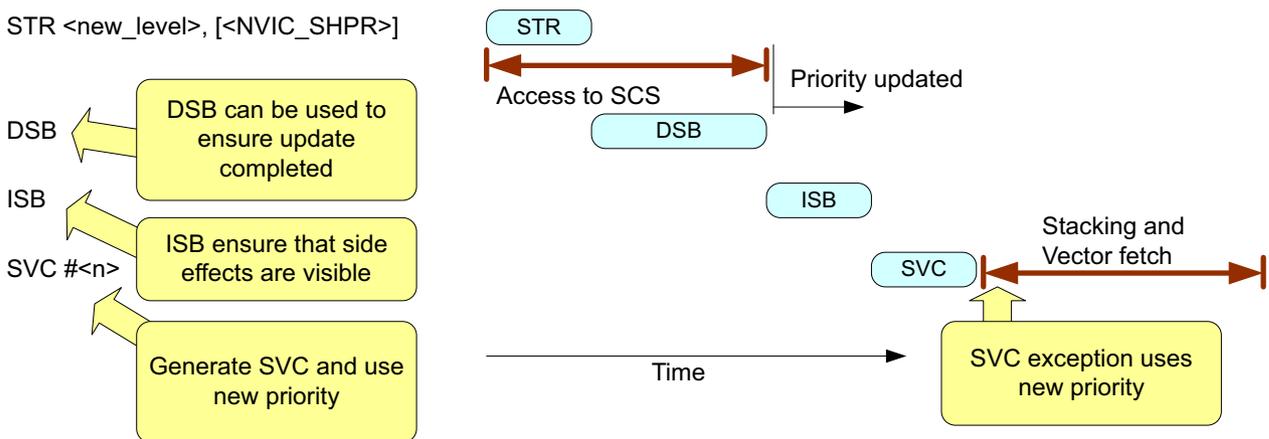


Figure 20 Architectural recommendation if changing a priority level and then immediately triggering it

If the next operation is a CPSIE instruction, or an MSR instruction that enables or disables interrupts, architecturally you should insert a DSB and then a ISB instruction.

Figure 21 shows how the intentional blocking of a pended IRQ, and the use of memory barrier instructions, ensures that the blocking is in place between configuration changes.

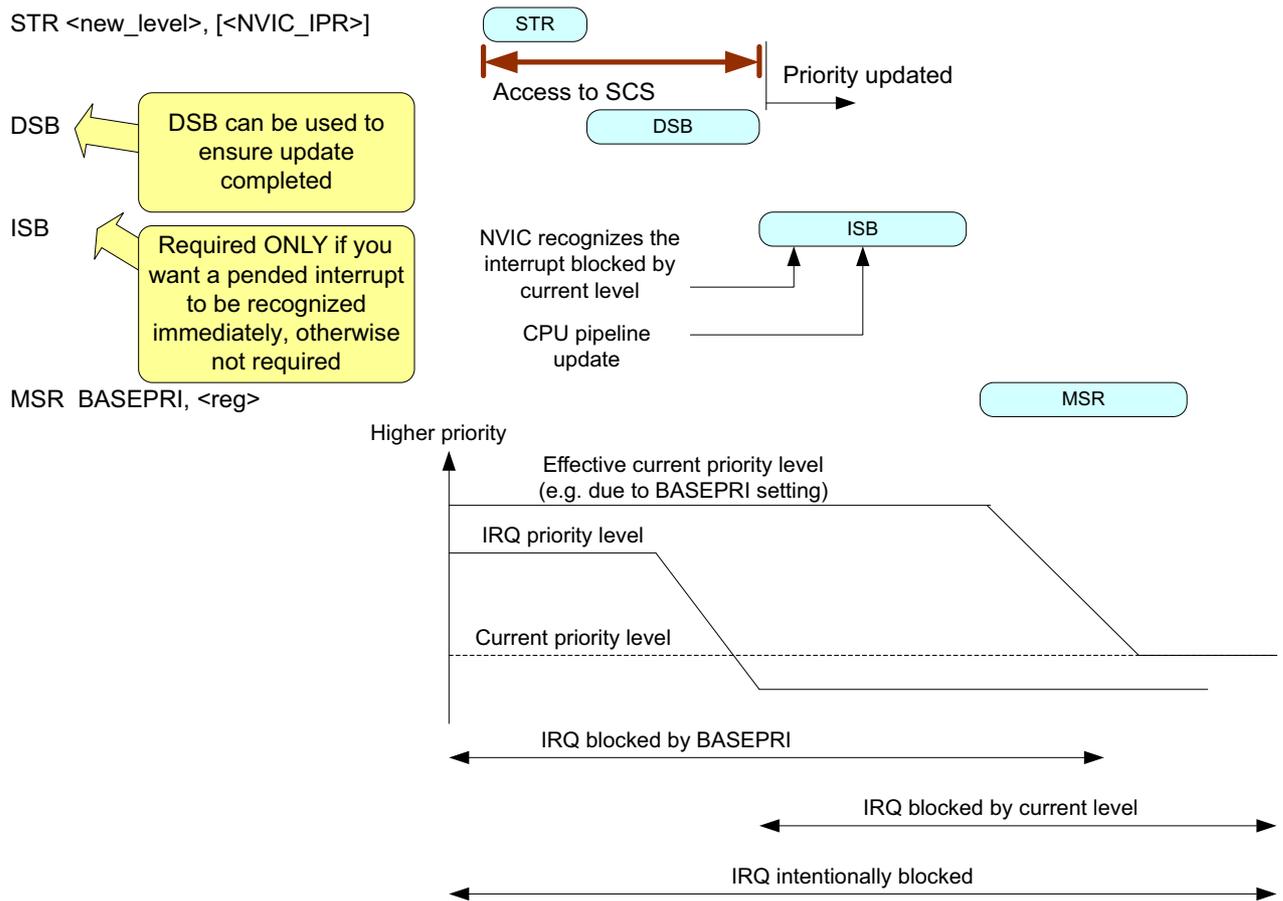


Figure 21 Barrier required after priority level change

Implementation requirements

In Cortex-M processors, an access to the interrupt priority registers already exhibits DSB behavior as these registers are implemented in the SCS.

On Cortex-M3 or Cortex-M4 processors:

- if it is necessary to have the side-effect of the priority change recognized immediately, an ISB instruction is required
- if it is not necessary to have the side-effect of the priority change recognized immediately, before subsequent operations, it is not necessary to insert a memory barrier instruction.

Note

On Cortex-M0 and Cortex-M0+ processors, dynamic changing of priority on enabled interrupts or exceptions is not supported.

In Cortex-M processors, there is no need to insert the memory barrier instructions if the next operation is an SVC, see Figure 22.

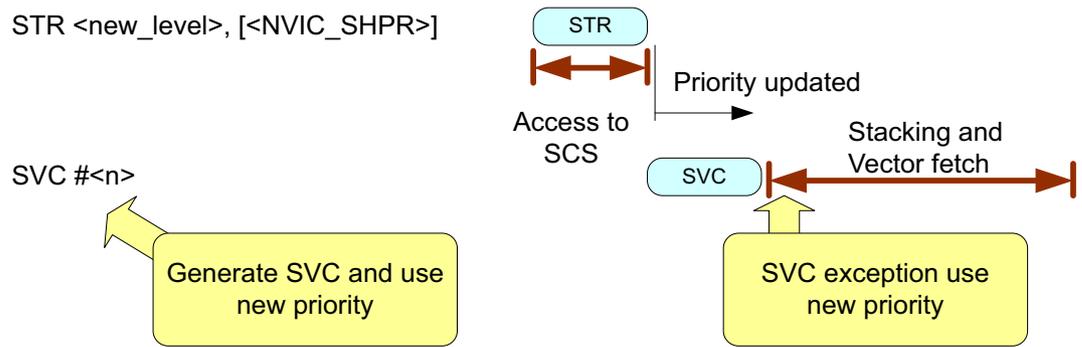


Figure 22 Cortex-M processors do not require a memory barrier after a priority level change for SVC

For Cortex-M3 or Cortex-M4 processor, you might want to insert an ISB instruction if the priority level change can result in the interrupt being accepted, and you want this interrupt to be executed immediately. For example, after raising the priority level of a pending interrupt from below current level to above current level, note that up to two more instructions can be executed if a memory barrier is not used.

Figure 23 shows that a barrier might be required after a priority level change in Cortex-M3 and Cortex-M4 processors.

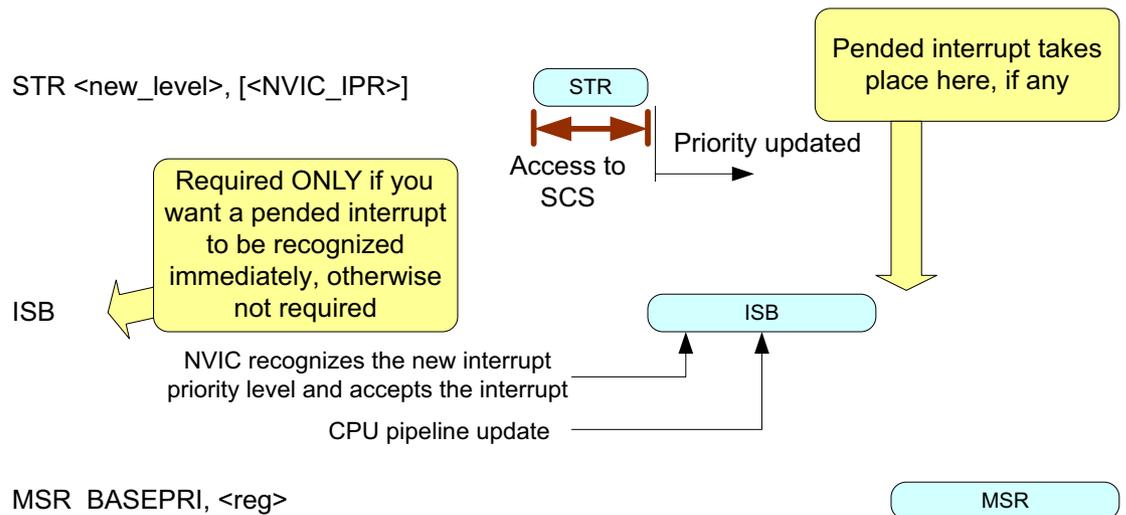


Figure 23 Barrier might be required after priority level change in Cortex-M3 and Cortex-M4 processors

Note

For the Cortex-M0 and the Cortex-M0+ processors, the interrupt should be disabled at the NVIC when changing priority level. The BASEPRI register is not available in these two processors.

4.11 Vector table configuration - Vector Table Offset Register (VTOR)

On the Cortex-M3 and Cortex-M4 processors, the location of the vector table is determined by the setting of VTOR (SCB->VTOR). This register is within the SCS and access to this register is Strongly-ordered.

Architectural requirements

ARM recommends that the architectural requirements are adopted.

Architecturally, a DSB instruction should be used after changing the VTOR if an exception is to be generated immediately and should use the latest vector table setting.

Figure 24 shows that architecturally, a barrier might be required after a VTOR update.

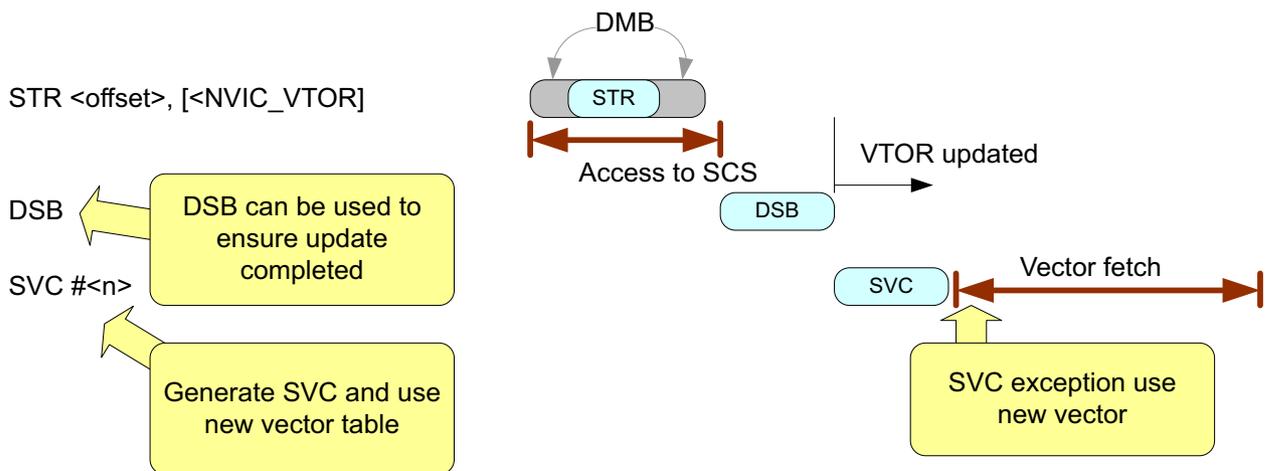


Figure 24 Barrier might be required after a VTOR update

Implementation requirements

In Cortex-M3, Cortex-M4 and Cortex-M0+ processors, accesses to the SCS have the DSB behavior, so there is no need to insert the DSB instruction.

Figure 25 on page 37 shows that in Cortex-M3, Cortex-M4 and Cortex-M0+ processors, a memory barrier is not required after a VTOR update.

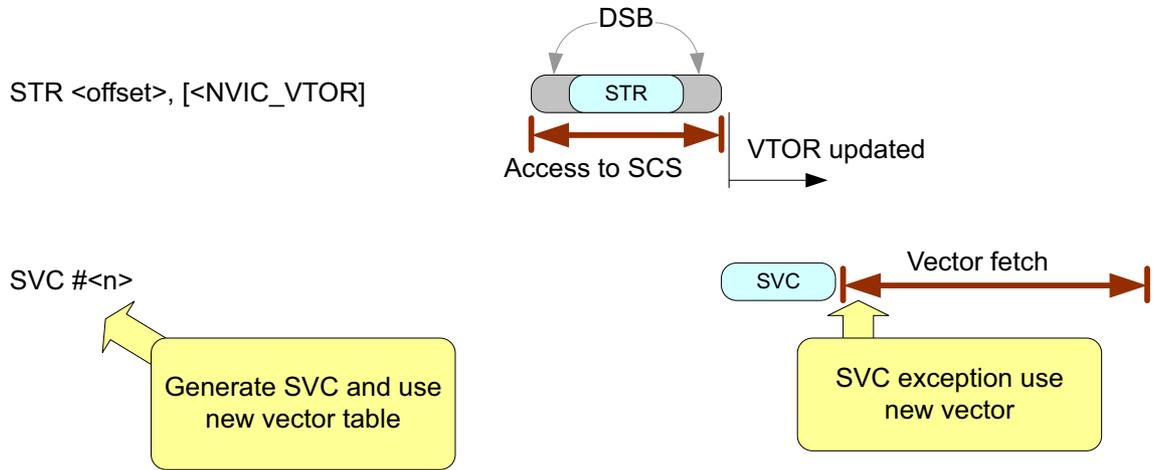


Figure 25 In Cortex-M3, Cortex-M4, and Cortex-M0+ processors, a memory barrier is not required after a VTOR update

———— **Note** ————

The VTOR register is not available on the Cortex-M0 processor.

4.12 Vector table configuration

Updating a vector table entry.

Architectural requirements

ARM recommends that the architectural requirements are adopted.

If the vector table is in writable memory such as SRAM, either relocated by VTOR or a device dependent memory remapping mechanism, then architecturally a memory barrier instruction is required after the vector table entry is updated, and if the exception is to be activated immediately.

Figure 26 shows that a barrier is required when updating a vector table entry.

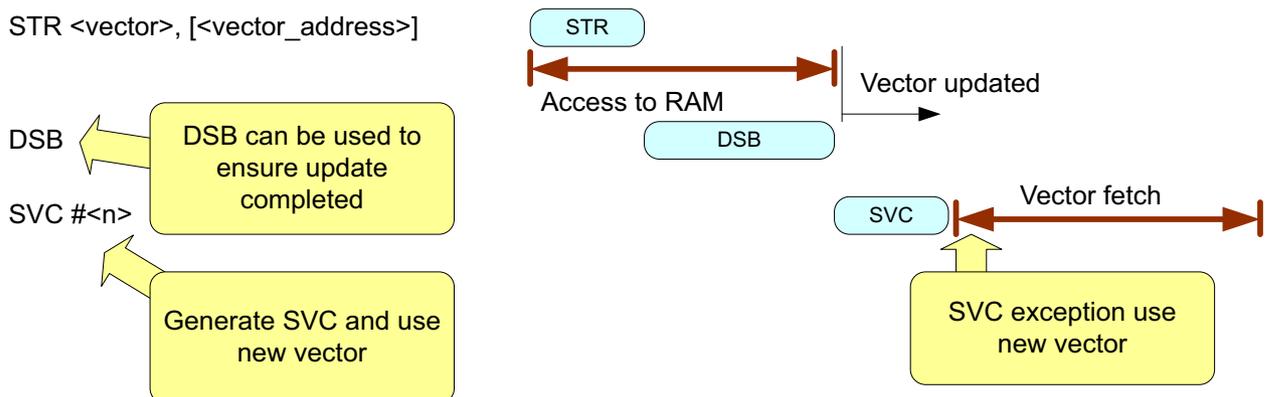


Figure 26 Barrier is required after a vector table update

If the subsequent operation is a memory access, you can use a DMB instruction.

Figure 27 shows the use of the DMB instruction.

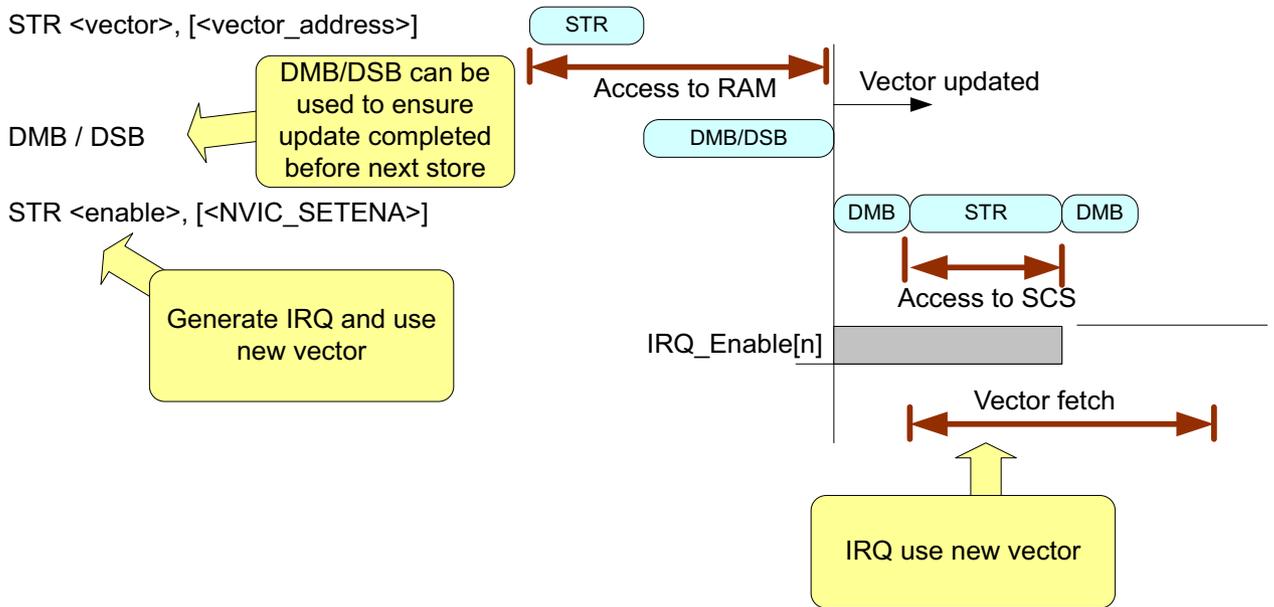


Figure 27 DMB instruction can be used during a vector table update if the subsequent operation is a memory access

As the example in Figure 27 shows, architecturally a DMB or DSB instruction should be inserted because Strongly-ordered accesses do not guarantee memory order related to the vector table update that is an access to Normal memory.

Implementation requirements

In the Cortex-M processors, omission of the DSB or DMB instructions does not cause any issue in vector entry modifications, because the exception entry sequence does not start until the last memory access is completed.

Figure 28 shows that in Cortex-M processors a SVC exception always gets the latest exception vector.

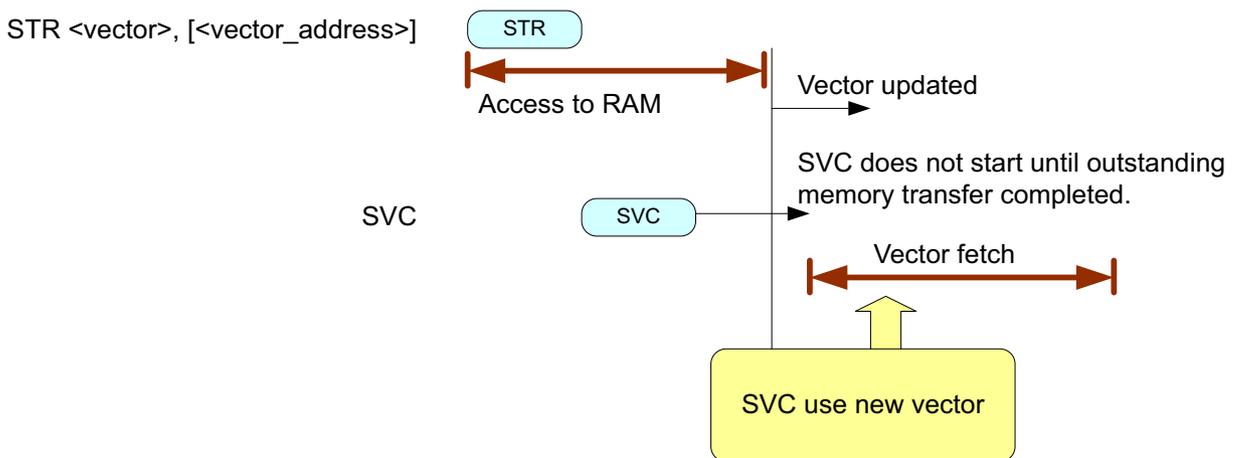


Figure 28 In Cortex-M processors a SVC exception always gets the latest exception vector

4.13 Memory map change

Many microcontrollers contain a device specific memory remapping feature to permit the memory map to be changed during runtime by programming a configuration register. The configuration register should be placed in Device memory space.

The need for memory barrier instructions during the changing of a memory map configuration depends on:

- whether the affected memory space covers program code, that is, it includes instructions
- the Device specific data path between the processor and the memory configuration register, for example, a write buffer.

Architectural requirements

ARM recommends that the architectural requirements are adopted.

In this case study, the following conditions apply:

- there is no device specific write-buffer affecting the memory-remap control register, apart from any internal write buffer in the processor
- there is no additional hardware delay in the memory map switching.

Architecturally, you should insert memory barrier instructions *before* and *after* the memory map changes:

```

... ; application code before switching
DSB ; Ensure all memory accesses are completed
STR <remap>, [<remap_reg>] ; Write to memory; map control register
DSB ; Ensure the write is completed
ISB ; Flush instruction buffer (optional, only required if
; the memory map change affects program memory)
... ; application code after switching
    
```

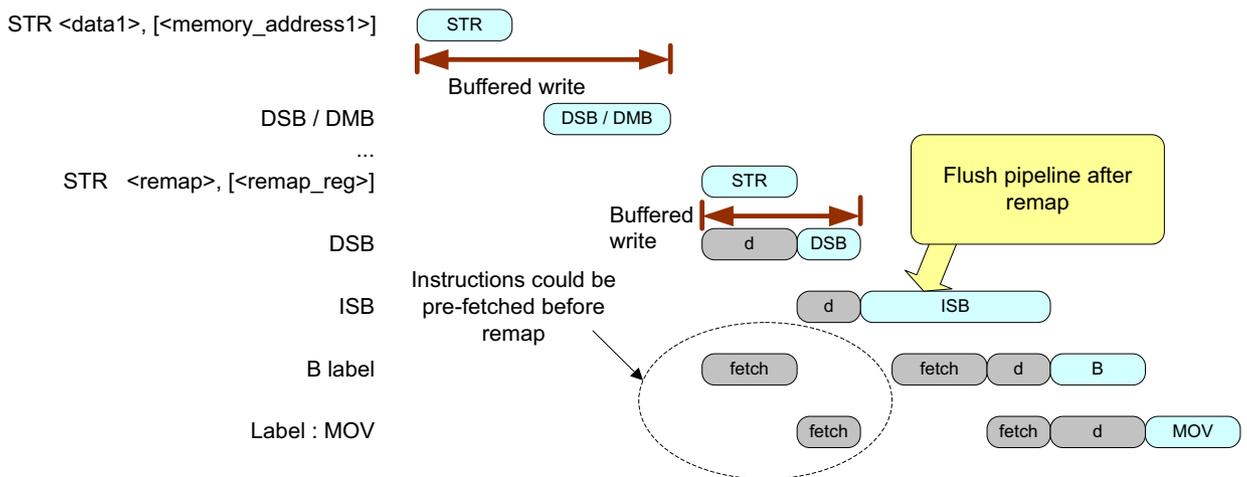


Figure 29 DSB and ISB are required after a memory map change

If the memory affected does not contain any program code, a DSB instruction is required after a memory map change, but the ISB instruction can be omitted.

Figure 30 shows the DSB requirement after a memory map change.

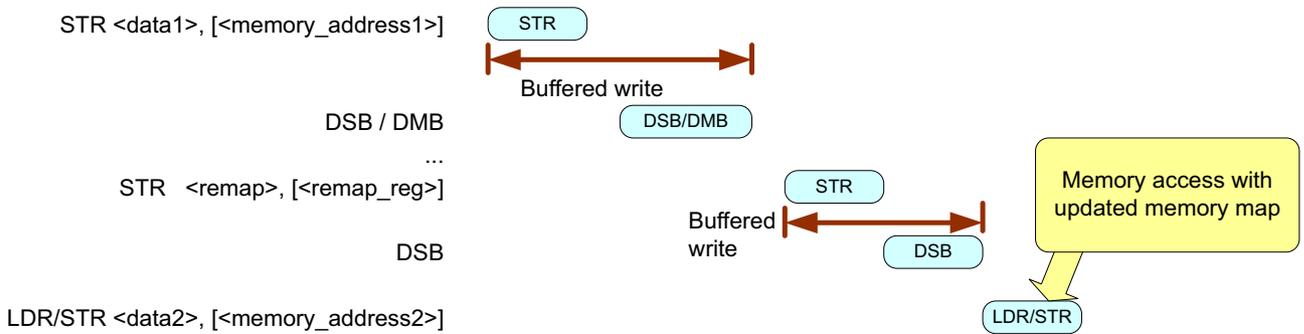


Figure 30 A DSB is required after a memory map change

Implementation requirements

In this case study, the following conditions apply:

- there is no device specific write-buffer affecting the memory-remap control register, apart from any internal write buffer in the processor
- there is no additional hardware delay in the memory map switching.

On Cortex-M processors:

- the DSB or DMB instruction before the memory map change is not required, because these processors do not permit overlapping of two write operations
- the DSB-and-then-ISB sequence after the remapping is required to ensure the program code is fetched using the latest memory map.

Note

Two assumptions are made in this case study. If these assumptions are invalid, for example, if the data path between the processor and the memory control register contains an additional system level write buffer, then the memory barrier instructions cannot guarantee that the transfer is completed. In this case:

- You can carry out a read operation from the previously accessed region to ensure the write buffer is drained. If multiple write transfers have been issued to various parts of the system, multiple read operations might be required to ensure all of the write buffers are drained.
- Alternatively, the microcontroller or SoC might have a status register that indicates if there is any active on-going transfer, and when the memory remapping has been completed. This permits the program code to account for an additional hardware delay on the memory remapping logic if required.

ARM recommends that you contact chip suppliers and manufacturers for detailed device specific recommendations.

4.14 Entering sleep modes

The Cortex-M processors provide WFI and WFE instructions for entering sleep.

Architectural requirements

ARM recommends that the architectural requirements are adopted.

Architecturally, you should use the DSB instruction before executing the WFI or WFE instructions.

Figure 31 shows the architectural requirement.

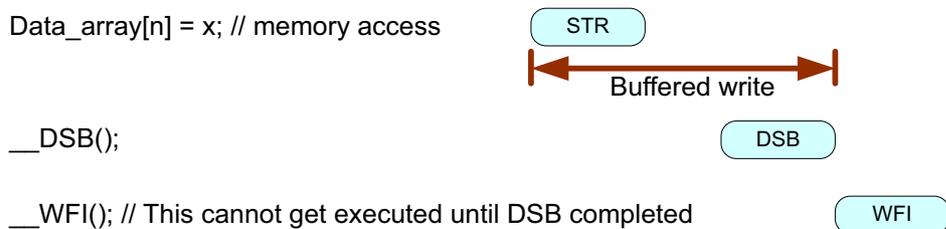


Figure 31 DSB instruction is required before entering sleep mode

Implementation requirements

In simple designs that do not include a system level write buffer, the memory barrier instruction is not required before entering sleep mode in the Cortex-M3 (r2p0 or later), Cortex-M4, Cortex-M0+, and Cortex-M0 processors. This is handled by the processor.

If the internal bus contains a write buffer at the system level, outside of the processor, then the situation is more complex. In this case, including a DSB instruction might not be sufficient because the system control logic might stop the clocks before the buffered write is completed.

Figure 32 shows the possible effect of a system level write buffer when entering sleep mode.

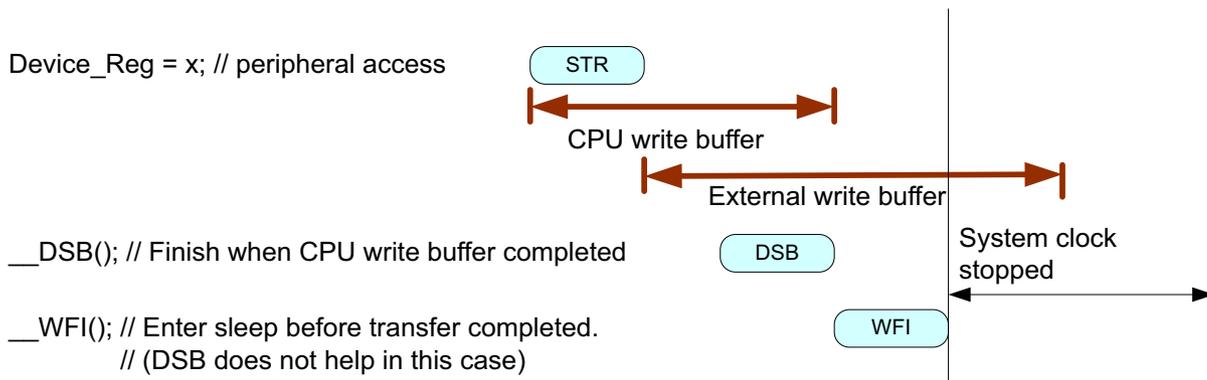


Figure 32 Possible effect of system level write buffer when entering sleep mode

Stopping the clock signals early might not cause an error, this depends on the system level design, the sleep operation being used, and which peripherals are being accessed immediately before entering sleep mode.

ARM recommends that you contact your chip supplier or manufacturer for device specific details.

This problem can normally be resolved by adding a dummy read operation through the write buffer to ensure that the write buffer is drained. Figure 33 shows a possible workaround.

Figure 33 shows a possible workaround using a dummy read of the external write buffer.

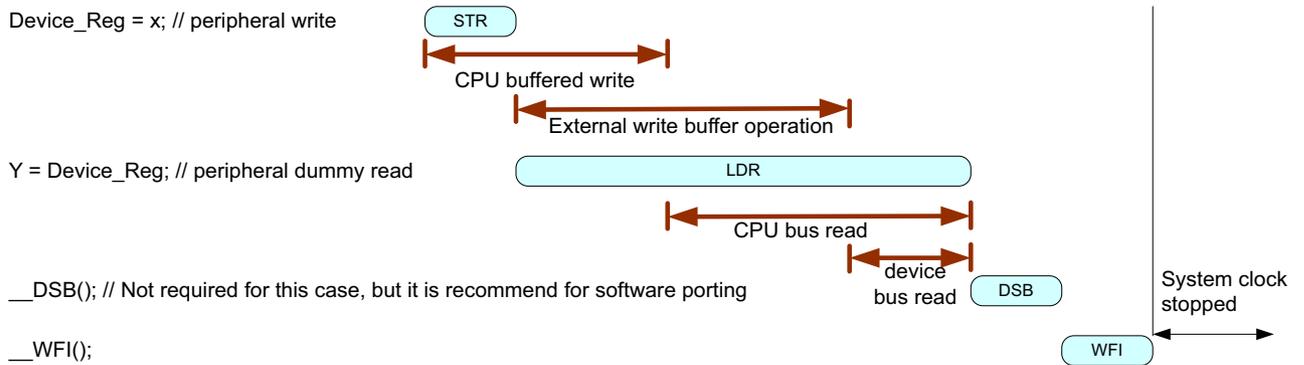


Figure 33 Possible workaround for Implementation specific delay in sleep operation

4.15 Self-reset

The Cortex-M processors include a self-reset feature. You can trigger a system reset using the `SYSRESETREQ` bit in the Application Interrupt and Reset Control Register (AIRCR). In the CMSIS library, the C function `NVIC_SystemReset(void)` is available for using this feature.

Architectural requirement

ARM recommends that the architectural requirements are adopted.

A `DSB` is required before generating self-reset to ensure all outstanding transfers are completed. The use of the `CPSID I` instruction is optional.

Figure 34 shows the architectural requirement.

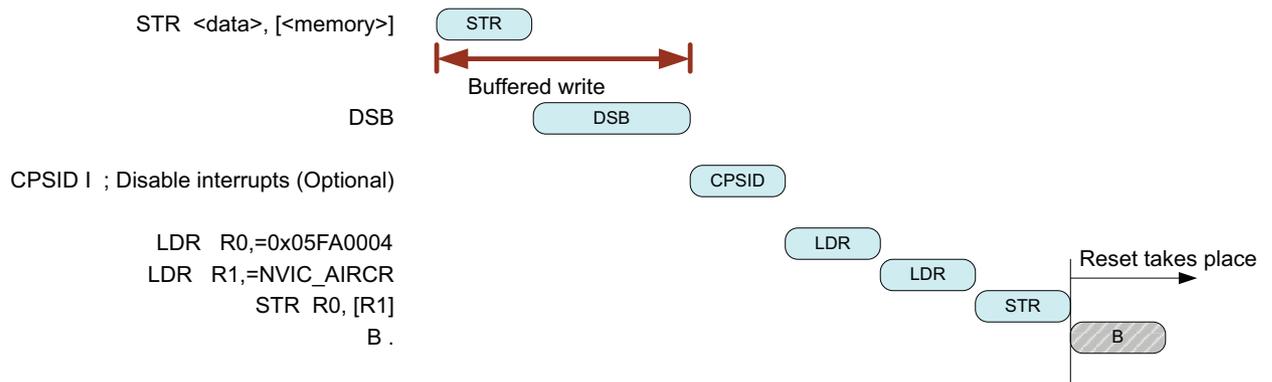


Figure 34 Memory barrier used before self-reset

The `DSB` ensures all outstanding memory transfers are completed before starting the self-reset sequence. The `CPSID I` is optional, it prevents an enabled IRQ triggering in the middle of the self-reset sequence.

If the self-reset is generated by a peripheral, the requirement is similar.

Figure 35 shows the requirement for a peripheral generated reset.

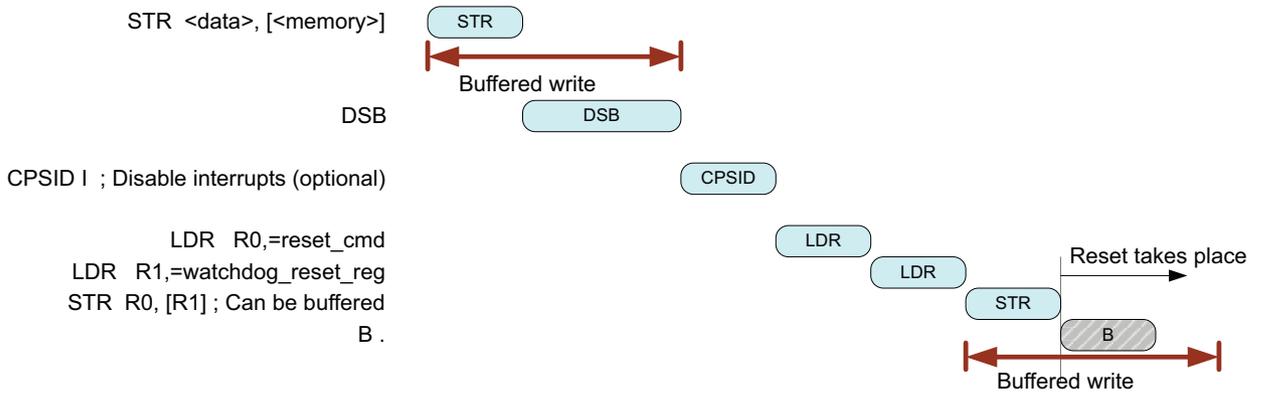


Figure 35 Peripheral generated reset

Implementation requirements

On Cortex-M processors, the DSB instruction is optional if the CPSID instruction is not used. Because the accesses to the SCS already have the DSB instruction behavior, the self-reset cannot start until the write operation is completed.

Figure 36 shows that Cortex-M processors do not reorder transfers, this permits the DSB instruction to be omitted.

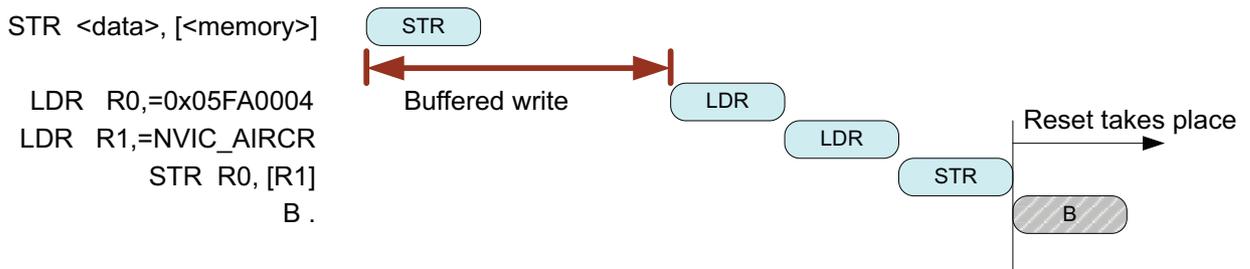


Figure 36 Existing Cortex-M processors do not reorder transfer so the DSB instruction can be omitted

If the CPSID instruction is used, then the DSB instruction should be inserted. This ensures CPSID is executed only when the previous transfer is finished. In this way, an imprecise bus fault resulting from previous transfers, if any, can take place before disabling the interrupts.

———— **Note** ————

The bus fault exception is not available on ARMv6-M and therefore not available in the Cortex-M0 processor.

Figure 37 shows the use of the DSB instruction if CPSID is used.

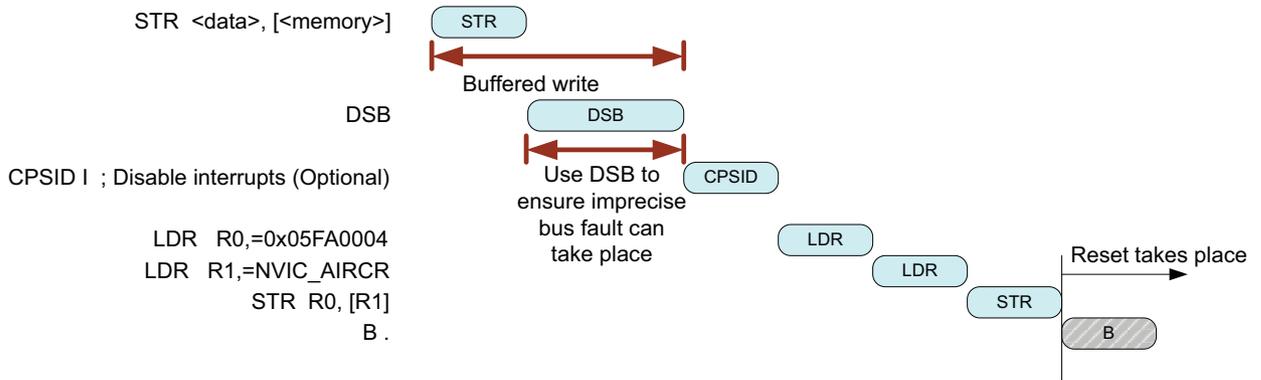


Figure 37 DSB is required if CPSID is used

If the system contains a write buffer at the bus level, a dummy read operation through the write buffer can be used to ensure that the system level write buffer is drained before executing CPSID instruction and carrying out a self-reset.

Figure 38 shows the use of a dummy read to drain the system level write buffer.

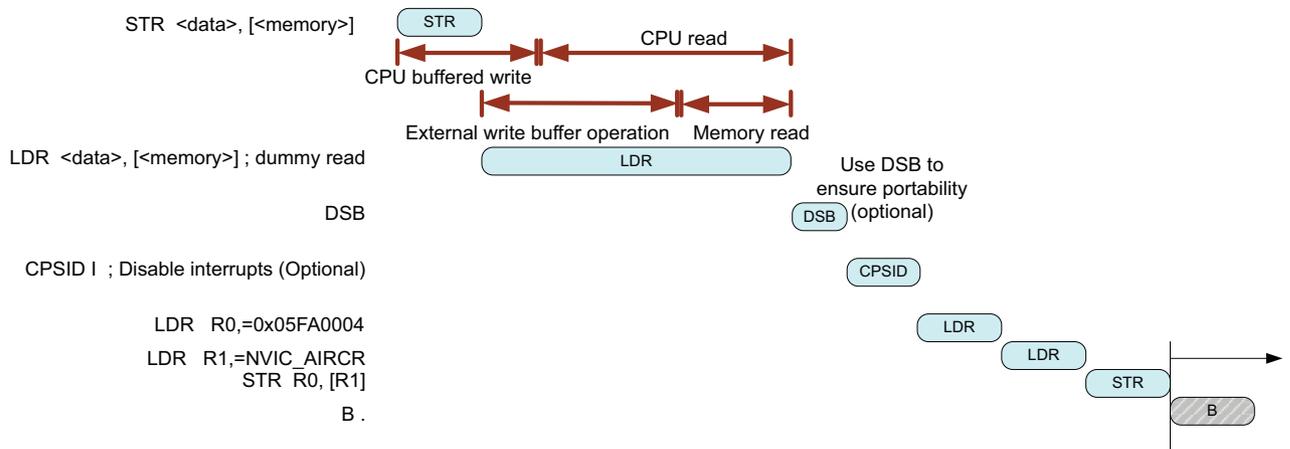


Figure 38 A dummy read ensures that the system level write buffer is drained

Note

If you are using CMSIS 2.0 or later, the NVIC_SystemReset(void) function already includes the DSB instruction.

4.16 CONTROL register

The CONTROL register is one of the special registers implemented in the Cortex-M processors. This can be accessed using MSR and MRS instructions.

Architectural requirement

ARM recommends that the architectural requirement is adopted.

Architecturally, after programming the CONTROL register, an ISB instruction should be used.

[Example 6](#) shows the code required to switch from privileged to unprivileged execution. This is not implemented in the Cortex M0 processor.

Example 6 Switching from privileged to unprivileged execution

```

MOVS R0, #0x1
MSR CONTROL, R0 ; Switch to non-privileged state
ISB ; Instruction Synchronization Barrier
...

```

The ISB instruction ensures that subsequent instructions are fetched with the correct privilege level.

[Figure 39](#) shows how the ISB instruction is used to ensure the correct privilege level is maintained.

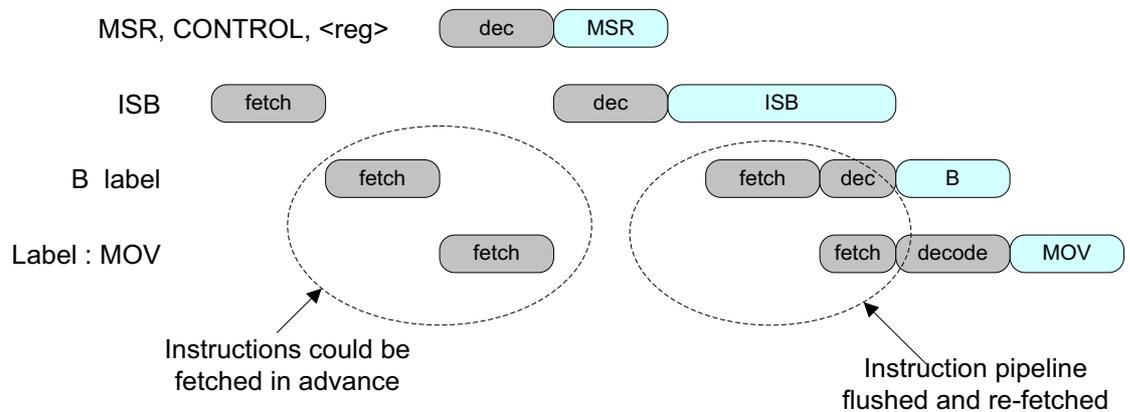


Figure 39 ISB is used after writing to the CONTROL register

You can also use the CONTROL register to select which stack pointer to use in thread mode.

Implementation requirement

In the Cortex-M processors, the omission of the ISB after writing to the CONTROL register does not cause a program error, apart from the issue that instructions could have been pre-fetched using the previous privilege level. The ISB instruction is only required if it is necessary to have the subsequent instructions fetched with the correct privilege level.

4.17 MPU programming

The MPU is an optional feature available on the Cortex-M0+, Cortex-M3, and Cortex-M4 processors.

Architecture requirements

ARM recommends that the architectural requirements are adopted.

Architecturally, the following conditions apply:

- The MPU configuration registers are in the SCS that is Strongly-ordered so there is no need to insert memory barrier instructions between each step of the MPU programming.
- Strongly-ordered memory does not enforce ordering related to Normal memory accesses. Architecturally, a DMB is needed *before* the MPU programming sequence. and *after* the MPU programming is completed, a DSB should be used to ensure all settings are visible.
- An ISB instruction should also be added if the MPU setting change affects program memory to ensure the instructions are re-fetched with updated MPU settings.

Figure 40 shows the MPU setup requirement.

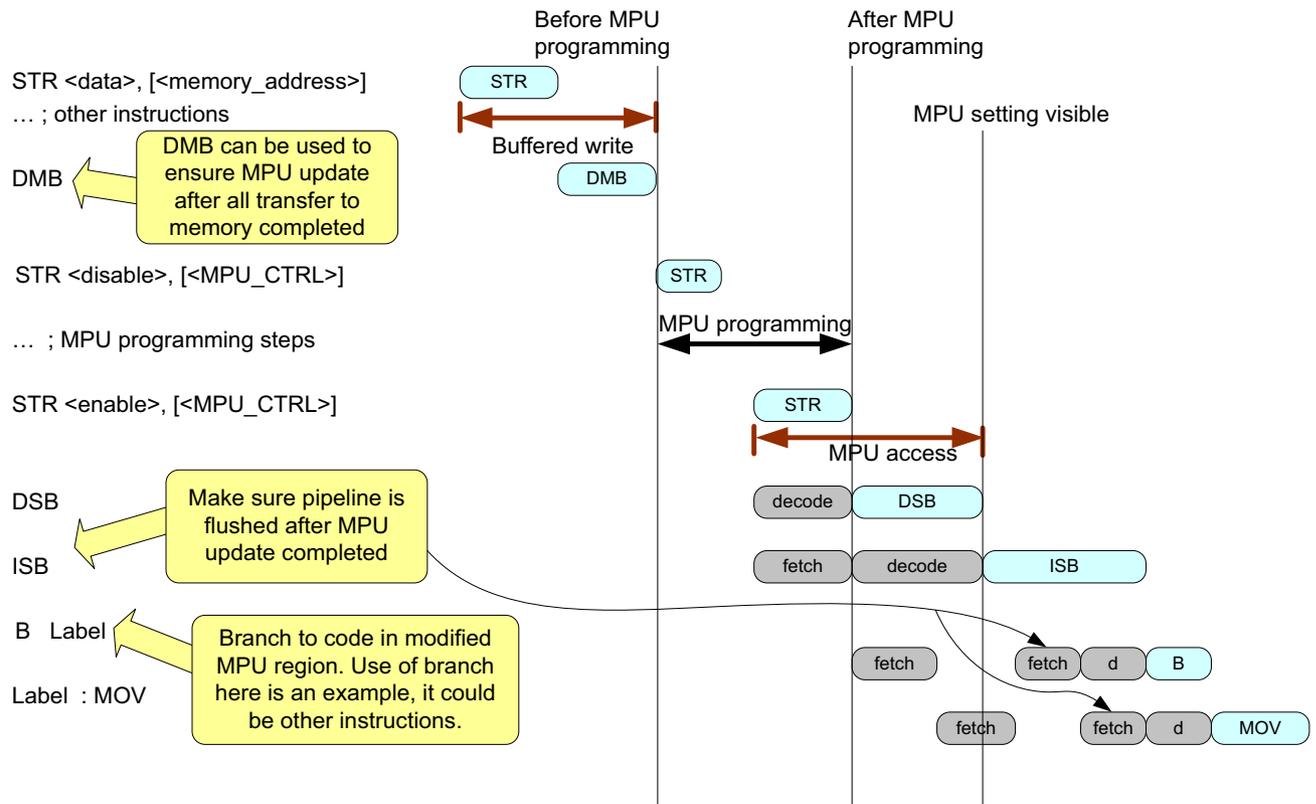


Figure 40 MPU setup requirement

If the MPU programming steps are carried out in an exception handler, then the ISB instruction is not required because the exception entry and exit boundary already has ISB behavior. For example, in an application with RTOS running, the MPU region settings for user threads can be updated during context switching that is carried out inside the PendSV exception handler. Switching the exception exit sequences between the PendSV and the user threads ensures the MPU settings take effect. This applies to both the architectural behavior and the implementation of current Cortex-M processors.

Implementation requirements

The omission of the DMB before entering MPU programming code on Cortex-M0+, Cortex-M3, and Cortex-M4 processors, does not cause an issue.

The omission of the DSB after completing MPU programming code on Cortex-M0+, Cortex-M3, and Cortex-M4 processors, does not cause an issue.

If the change to the MPU settings only affects data memory and not program memory, the ISB instruction is not required on the Cortex-M processors. If it is necessary for the subsequent instructions to be fetched using the new MPU settings, the ISB instruction is required.

4.18 Multi-master systems

The use of memory barrier instructions can be important in multi-master systems if the code must be portable.

Architectural requirement

ARM recommends that the architectural requirements are adopted.

Either a DMB or DSB instruction is needed when handling shared data and when the memory ordering needs to be preserved.

For example, if a *Direct Memory Access* (DMA) controller is available in the system, the DMB instruction is required before starting DMA operations.

Figure 41 shows an example of DMB use in a multi-master system.

———— Note ————

You can use DSB instead of DMB.

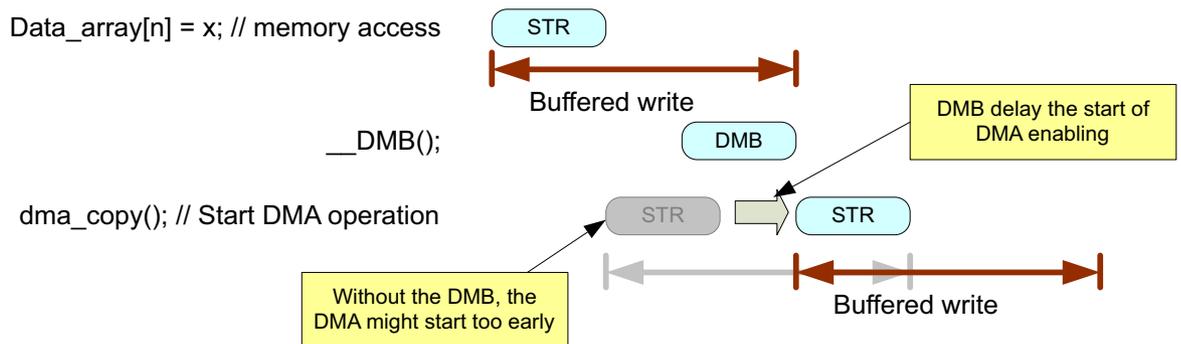


Figure 41 The use of DMB in a multi-master system

Without the DMB, architecturally the two stores can be re-ordered or overlapped, with the possibility of the DMA starting before the data array update completes.

Another multi-master example is the communication of information between two processors on a shared memory. When passing data to another program running on a different processor, it is common to write the data to a shared memory, and then set a software flag in the shared memory. In this case you should use DMB or DSB instructions to ensure correct memory ordering between the two memory accesses.

Figure 42 shows an example of DMB use for shared data in a multi-master system.

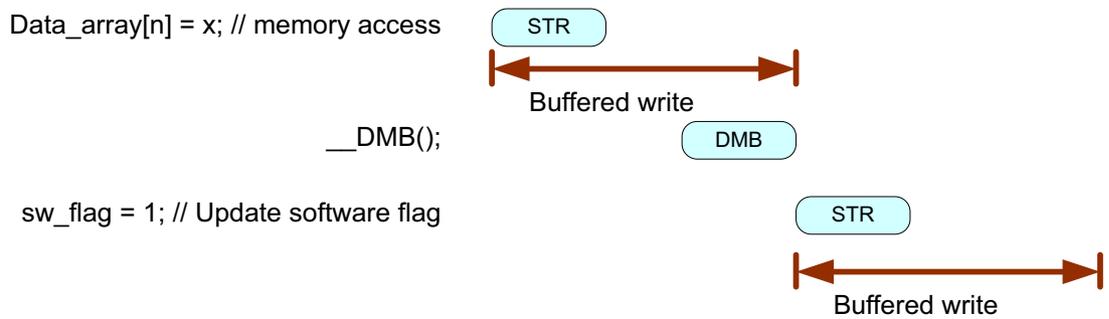


Figure 42 The use of DMB for shared data in a multi-master system

The interactions between two processors are not limited to shared memory. Another possible interaction is an event communication. In this case, the DSB instruction might be needed to ensure that the correct ordering between memory transfers and events is preserved.

Figure 43 shows an example of DSB use for other multi-master interactions.

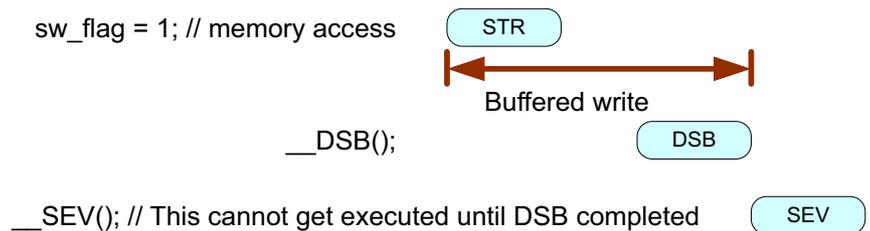


Figure 43 The use of DSB for other multi-master interactions

Implementation requirements

Omitting the DMB or DSB instruction in the examples in Figure 41 on page 47 and Figure 42 would not cause any error because the Cortex-M processors:

- do not re-order memory transfers
- do not permit two write transfers to be overlapped.

In the Cortex-M3 and Cortex-M4 processors, the DSB instruction is required for the example in Figure 43.

In the Cortex-M0 processor, the omission of the DMB or DSB instruction does not cause errors in any of the three examples because there is no write buffer in the Cortex-M0 processor.

4.19 Semaphores and Mutual Exclusives (Mutex) - uncore and multicore

Semaphore and Mutex operations are essential in many *Operating Systems* (OS). They might be used in single processor environments or multi-processor environments.

In a multi-processing environment, semaphore operations require software variables to be placed in memory that is shared between multiple processors. In order to ensure correct operation, memory barrier instructions should be used.

If cached memory is available in a multi-processor system, you must ensure correct cache configurations are used so that the data in shared memory is coherent across all processors.

Architectural requirements

ARM recommends that the architectural requirements are adopted.

The DMB instruction should be used in Semaphore and Mutex operations.

[Example 7](#) shows simple code for getting a lock. A DMB instruction is required after the lock is obtained.

Example 7 Simple code for getting a lock

```
void get_lock(volatile int *Lock_Variable)
{ // Note: __LDREXW and __STREXW are CMSIS functions
  int status = 0;
  do {
    while (__LDREXW(&Lock_Variable) != 0); // Wait until
    // Lock_Variable is free
    status = __STREXW(1, &Lock_Variable); // Try to set
    // Lock_Variable
  } while (status!=0); //retry until lock successfully
  __DMB();// Do not start any other memory access
  // until memory barrier is completed
  return;
}
```

Similarly, code for releasing the lock should have a memory barrier at the beginning.

[Example 8](#) shows simple code for releasing a lock. A DMB instruction is required before the lock is released.

Example 8 Simple code for releasing a lock

```
void free_lock(volatile int *Lock_Variable)
{ // Note: __LDREXW and __STREXW are CMSIS functions
  __DMB(); // Ensure memory operations completed before
  // releasing lock
  Lock_Variable = 0;
  return;
}
```

Implementation behavior

On microcontroller devices using the Cortex-M3 and Cortex-M4 processors, omission of the DMB instruction in semaphore and Mutex operations does not cause an error. However, this is not guaranteed if:

- the processor has a cache
- the software is used in a multi-core system.

ARM recommends that the DMB instruction should be used in semaphores and Mutex operations in OS designs.

The Cortex-M0 and Cortex-M0+ processors do not have exclusive access instructions.

4.20 Self modifying code

If a program contains self-modifying code, memory barriers would be required if the changed program code is to be executed soon after being modified. Since the program code can be pre-fetched, you should execute a DSB instruction followed by an ISB instruction to ensure the pipeline is flushed.

Architectural requirement

ARM recommends that the architectural requirement is adopted.

The architectural requirement is to use a DSB instruction followed by an ISB instruction after modifying program memory.

Example 9 The use of DSB and ISB instructions after modifying program memory

```
STR <new_instr>, [<inst_address1>]
DSB ; Ensure store is completed before flushing pipeline
ISB ; Flush pipeline
B <inst_address1> ; Execute updated program
```

Figure 44 shows the memory barrier instructions required to meet the architectural and implementation requirements for self-modifying code.

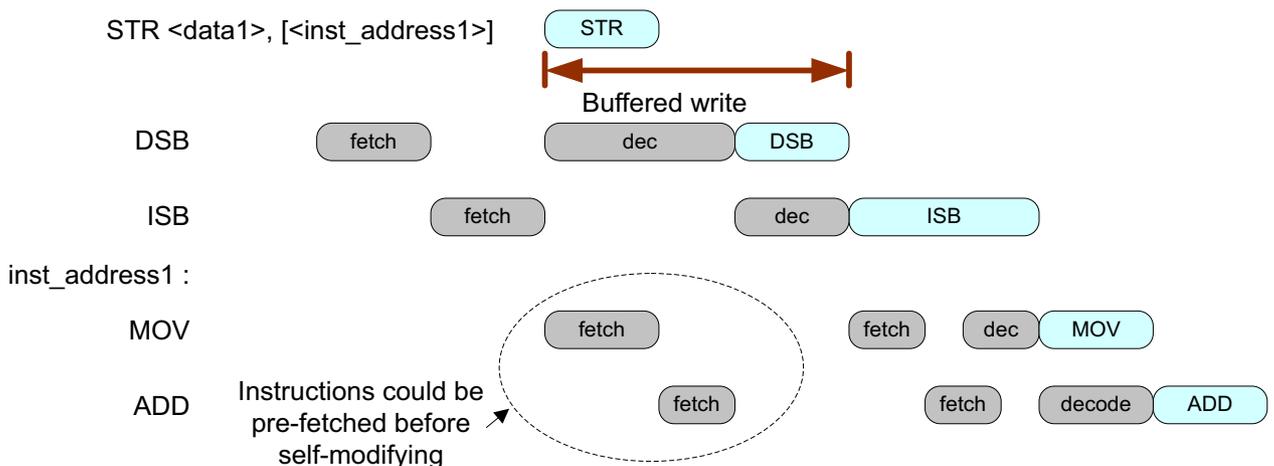


Figure 44 Memory barrier instructions required for self-modifying code

If cached memory is present in the system, an additional cache flush operation should be used to ensure that the instruction cache is updated.

Implementation requirement

In general, use DSB and then ISB after modifying program memory.

The DSB could be omitted if there is no write buffer or cache in the processor or at the system level, for example, in Cortex-M0 based microcontrollers.

The Cortex-M3 and Cortex-M4 processors can pre-fetch up to six instructions. If an application modifies an instruction in program memory and executes it shortly afterwards, the outdated instruction could have been used. If the instruction is not used for some time after the modification, the program might work without failure, but this cannot be guaranteed.

Some Cortex-M3 and Cortex-M4 designs might have an implementation specific program cache to accelerate program memory accesses. Additional steps might be required to ensure the program cache memory is invalidated after the program code modification.

5 Additional Reading

This section lists publications from both ARM and third parties that provide additional information. See the *ARM Information Center*, <http://infocenter.arm.com> for access to ARM documentation.

5.1 User guides

The following user guides can be downloaded from the ARM Infocenter:

- *Cortex-M0 User Guide Reference Material* (ARM DUI 0467)
- *Cortex-M3 User Guide reference Material* (ARM DUI 0450)
- *Cortex-M4 User Guide Reference Material* (ARM DUI 0508).

5.2 Architecture reference manuals

The following ARM Architecture Reference Manuals can be downloaded from the ARM Infocenter subject to user registration:

- *ARMv7-M Architecture Reference Manual* (ARM DDI 0403)
- *ARMv6-M Architecture Reference Manual* (ARM DDI 0419).

5.3 Application notes

The following application note can be downloaded from the ARM Infocenter:

Application Note 179 - Cortex-M3 Embedded Software Development (ARM DAI 0179).

5.4 Software standards

The following software package and its documentation can be downloaded from the ARM website subject to user registration:

- *Cortex Microcontroller Software Interface Standard*, <http://www.arm.com/cmsis>.

5.5 Other publications

The following books are available from various book shops and on-line retailers:

- *The definitive Guide to the ARM Cortex-M3, Second Edition* (Joseph Yiu ISBN: 978-1-85617-963-8)
- *The Definitive Guide to the ARM Cortex-M0* (Joseph Yiu ISBN: 978-0-12-385477-3).