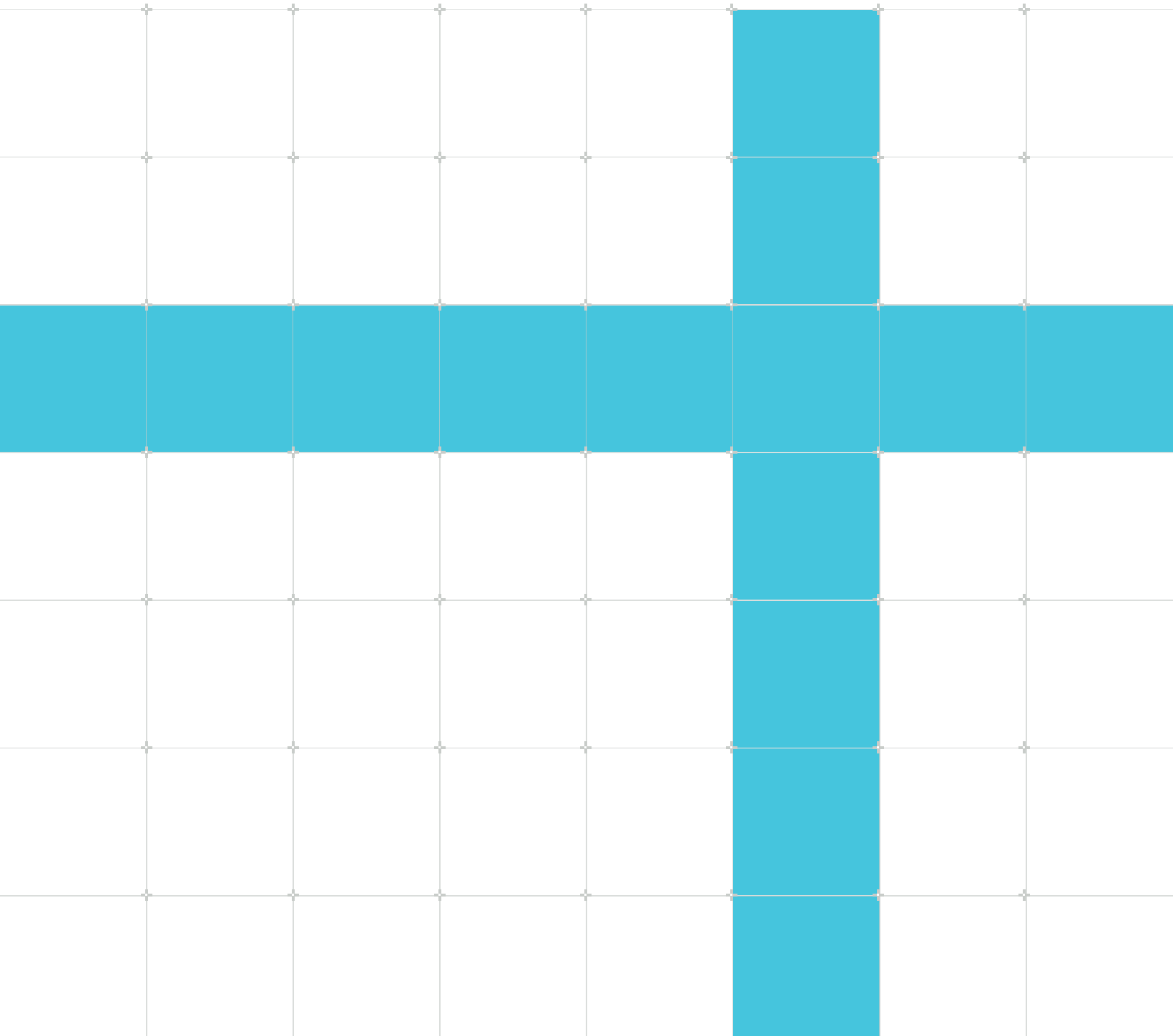
Armv8-M Barriers and Synchronization

Version 1.0

User Guide

Issue 01
111493_100_01_en



Armv8-M Barriers and Synchronization

User Guide

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	16 December 2025	Non-Confidential	First release

Proprietary Notice

makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, any patents, copyrights, trade secrets, trademarks, or other rights.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Reference by Arm to any third party's products or services within this document is not an express or implied approval or endorsement of the use thereof.

This document consists solely of commercial items. You shall be responsible for ensuring that any permitted use, duplication, or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to Arm's customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

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.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introducing Armv8-M Barrier and Synchronization

This guide describes the barrier instructions that were introduced in Armv8-M Memory Model and Memory Protection User Guide.

This guide is written to help software developers work with the instructions quickly and efficiently. For full definitions and strict architectural requirements, refer to the *Armv8-M Architecture Reference Manual*.

2. 1: Barriers

We use memory barriers because modern processors often optimize performance with techniques such as instruction pipelining, speculative memory accesses, caching, and store buffering. Without proper synchronization, these optimizations can cause inconsistent or incorrect results, especially when multiple parts of the system are accessing the same memory or memory-mapped registers.

2.1 1.1: The need for barriers

In Armv8-M architecture, barrier instructions, like ISB, DSB etc, enforce ordering or completion of accesses, ensuring critical data accesses are seen by other parts of the system, sometimes referred to as an Observer, before other actions are performed.

Barriers also synchronize instruction execution of the processor. This is especially required when changes are made that affect instruction execution or system behavior. For example, when the privilege level of the processor is changed by setting CONTROL.nPRIV, by the time the write completes, several instructions following the one that triggered the write can have already executed or have been partially completed.

The need for barriers is often associated with high performance Out-of-Order processors. However, they are still important on Armv8-M processors to ensure correct behavior and enable code portability.

The type of memory, as defined by the Memory Protection Unit, or MPU, being accessed also affects the ordering of memory operations and whether barriers are needed. For a description of the different memory types supported in Armv8-M architecture and their basic ordering properties, see the Memory types and attributes section of the *Armv8-M Memory Model and Memory Protection User Guide*.

2.2 1.2: Barrier instructions

This section describes the ISB, DMB, DSB, and ESB instructions.

The Armv8-M architecture also includes barriers to mitigate speculative execution vulnerabilities. For more information on the need for Speculative Barrier instructions, see the Arm CPU Security Bulletin: Spectre/Meltdown

2.2.1 1.21: Instruction Synchronization Barrer (ISB)

An ISB ensures that all instructions following it in the program are fetched and executed from memory or cache only after the ISB has finished. It therefore makes sure that any changes that have been completed, such as permissions updates, made by instructions before the ISB, are applied to the instructions that come after it. When the ISB is executed, it flushes the pipeline in

the processor and refetches any in-flight instructions, so any new settings are applied to them. If you change a register through memory-mapped access, insert a DSB before the ISB. The DSB ensures the update completes, and the ISB ensures the update affects later instructions. See 1.3.3: Setting processor control registers.

In an application using CMSIS, you can use the __ISB() function to insert the ISB instruction.

The following events within the processor have the same effect as an ISB, so an explicit ISB is not required in all cases:

- Exception entry and return
- Halting or restarting the processor in a debugger

All the special purpose registers in the processor, except the CONTROL register, are accessed directly with an MRS or MSR instruction. These accesses are self-synchronizing, so they do not need a barrier to force a change to be applied.

2.2.2 1.22: Data Memory Barrier (DMB)

A DMB ensures that all accesses before the DMB in program order are observed before any access to the same memory or peripheral that comes after it. That is, a DMB enforces the ordering of accesses, but not when they complete. This is important when other devices in the system, for example, other processors or direct memory access peripherals, can access memory also used by the processor. See 1.3.1: Shared memory communication between processors. The DMB barrier do not force dirty data in any caches to be cleaned. As a result, cache maintenance operations may also be required if shared data is present in a private cache. See 1.3.4: Performing cache maintenance.

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

2.2.3 1.23: Data Synchronization Barrier (DSB)

A DSB ensures that all memory accesses before the DSB are completed before the DSB itself completes. No instruction that comes after the DSB in program order can be executed until the DSB has fully completed.

The DSB is a stronger barrier than the DMB. In addition to the memory ordering enforced by a DMB, a DSB ensures that all previous memory operations have completed. One consequence of the stronger ordering provided by a DSB is that you can use it to force access ordering to different memories or peripherals. A DSB is also often used after writing to memory-mapped control registers to ensure that the requested change is complete. For example, a DSB is used after a write to the cache maintenance registers to wait for the maintenance operation to be completed. See 1.3.4: Performing cache maintenance.

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

2.2.4 1.24: Error Synchronization Barrier (ESB)

The RAS Extension, an optional part of the Armv8.1-M architecture, introduces the ESB instruction. The ESB instruction acts as a barrier to latent bus errors caused by in-flight instructions. It also forces the processor to recognize any present but undetected faults at the same time as the ESB instruction. This instruction with other architecture features enables some hardware errors to be contained within parts of the software, for example, containing an error to an unprivileged thread. The ESB also acts as a DSB where there is a BusFault or HardFault exception pending and ensures subsequent loads of the syndrome registers such as BFSR, RFSR, and SHCSR return the correct and updated values.

CMSIS does not have an intrinsic function for ESB.

ESB instructions isolate errors on demand. The Armv8.1-M Architecture introduces the Implicit Error Synchronization (IESB), to automatically insert ESB at critical points. When enabled, an IESB is inserted on every exception entry, exception return, and around lazy floating-point stacking operations to ensure that faults arise in the appropriate processor state. IESB can isolate the error and show where it occurred. If the error affects only a background thread, you need to terminate just that thread. If the error affects the RTOS or other parts of memory, you must terminate and restart the whole system. In contrast, ESB usage is more intentional. Use it if you are isolating errors between different parts of software in the same thread.

2.3 1.3: Typical barrier instruction usages

In the following section we will discuss some example barrier instructions.

2.3.1 1.3.1: Shared memory communication between processors

The following code shows a simplified example of software running on two different processors using shared uncached memory to pass a message.

Processor A

```
// Global variables ready and data placed in the same memory
bool ready = false;
int  data  = 0;

void send(void) {
    data  = 10;
    __DMB();
    ready = true;
}
```

Processor B

```
int receive(void) {
    while (!ready); // wait until data has been updated
    __DMB();
    return data; // will return value 10
```

```
}
```

The DMB in the `send()` function is required to ensure that the setting to the ready flag is observed after the data variable has been updated. Similarly, the DMB in the `receive()` function is required to ensure that the reading of the data variable is observed after the ready flag has been seen to be set.

2.3.2 1.3.2: Shared memory communication to peripherals

The following example shows a DMA accelerator being used to copy data in a shared uncached memory. The same barrier requirement also applies if the target memory location is a Tightly Coupled Memory (TCM).

```
#define BUFFER_LEN 100

int fromBuf[BUFFER_LEN];
int toBuf[BUFFER_LEN];

void dmaCopy(void) {
    // Fill the from array with some test data
    for (int i = 0; i < BUFFER_LEN; i++) {
        fromBuf[i] = i;
    }
    // Wait until writes to fromBuf[] are complete
    __DSB();
    // Write to the DMA accelerators registers to
    DMA_FROM_ADDR = fromBuf;
    DMA_TO_ADDR   = toBuf;
    // Setting the length register triggers the DMA hardware to
// start the copy.
__DMB();
    DMA_COPY_LEN  = BUFFER_LEN;
}
```

Because the `fromBuf` array is stored in a different peripheral or memory to the DMA accelerators control registers, a DMB is insufficient to ensure that the DMA accelerator sees the updated values. Instead, you must use a DSB. The code above assumes that DMA accelerators control registers are in Device-nGnRnE memory. A DMB isn't required between setting up the from and to registers and starting the transaction with the write to the length register.

2.3.3 1.3.3: Setting processor control registers

Barriers are often required when changing the processor configuration. The following code shows an update to the MPU registers but also applies to other memory-mapped processor registers. Accesses to other memory-mapped system registers in the System Control Space (SCS) such as the AIRCR, for example, will also require DSB and ISB instructions, as shown in the MPU example below.

```
MPU->CTRL |= 1; // enable the MPU
<code before the ISB MAY be subject to MPU checking>
// The DSB waits until the update to MPU->CTRL has happened
__DSB();
// Flush the pipeline to force all instructions to be checked
```

```
// by the MPU
__ISB();
<code after the ISB WILL be subject to MPU checking>
```

The following example shows an update to a special-purpose register.

```
__set_CONTROL(__get_CONTROL() | 1); // set nPRIV
<code before the ISB MAY be subject to new nPRIV value>
__ISB();
<code after the ISB WILL be subject to new nPRIV value>
```

The CONTROL register is not memory-mapped and is accessed directly with MRS and MSR instructions. The change to the nPRIV bit occurs immediately and does not require a DSB to wait for the register update to happen. However, the ISB is still required to make sure that the new value is used by the instructions following the ISB.

The code below shows some examples of where barriers are not required.

```
// Handler for PendSV exception
void PendSVHandler(void) {
    __set_FAULTMASK(1);
    // ISB not required as FAULTMASK is self-synchronizing
    <some critical code that must not be interrupted>
    __set_FAULTMASK(0);
    // Set UBIT_EN to enable BTI checking for background code
    __set_CONTROL(__get_CONTROL() | 1 << 5);
    // No ISB required to force UBIT_EN checking as the
    // exception return performed after this function returns
    // will act like an implicit ISB, forcing the update to
    // CONTROL to be seen.
}
```

We strongly recommend using CMSIS Core functions to access system registers when available because they already correctly use barrier instructions where necessary.

2.3.4 1.3.4: Performing cache maintenance

After you perform cache maintenance, insert a DSB followed by an ISB. This sequence ensures that any program running afterward observes the effect of the cache maintenance.

```
void cleanInvalidateDCache(uint32_t *addr, int32_t size) {
// Clean and invalidate a range of data cache lines
SCB_CleanInvalidateDCache_by_Addr(addr, size);
// Ensure the cache maintenance completes.
__DSB();
// Ensure instructions after the ISB query the cache
// after the maintenance has completed.
__ISB();
}
```

2.3.5 1.3.5: Updating a vector table

When modifying a vector table entry or relocating the vector table, you must include barrier instructions to ensure the changes take effect.

The code below shows relocating a vector table from flash memory to SRAM and updating the Vector Table Offset Register (VTOR).

```
// The original vector table defined in the linker script
extern uint32_t __Vectors[];

// Defining the number of vector table entries
#define NUMBER_OF_VECTORS 16

// Allocate a vector table in SRAM
Uint32_t vectorTable[NUMBER_OF_VECTORS];

void relocateVectorTable(void) {
// Copy the vector table from flash to SRAM
memcpy(vectorTable, __Vectors, NUMBER_OF_VECTORS * sizeof(unit32_t));

// Force the memcpy() to complete before updating the
// VTOR register. As vectorTable is in a different
// memory/peripheral to the SCS registers a DMB isn't
// sufficient.
__DSB();

// Set the VTOR to point to the new vector table location
SCB->VTOR = (uint32_t)vectorTable;

// Ensure the changes are seen by subsequent instructions
__DSB();
__ISB();
// Interrupts or exceptions that are taken after this
// point are guaranteed to use the updated vector table.
}
```

2.3.6 1.3.6: Going into low power modes

Wait For Interrupt (WFI) and Wait For Event (WFE) instructions do not automatically perform memory synchronization. You will need a DSB before WFI or WFE if you have pending memory transactions, such as writes to peripheral registers or cache maintenance operations that must be completed before entering low-power state. Without the DSB, memory operations might still be in progress when the processor enters WFI or WFE. This can cause inconsistent states or delayed peripheral responses after wake-up.

```
void goToWFI(void) {
while (!wakeupInterruptOccured()) {
      __set_PRIMASK(1);
// DSB to ensure there are no outstanding memory
// transactions prior to executing WFI and going to
// sleep.
__DSB();
__WFI();
}
__set_PRIMASK(0);
}
```

2.3.7 1.3.7: Debug authentication and barriers

Depending on the platform, there might be a need to change debug authentication signals dynamically through software to enable debugging. This can be useful in events where platforms in the field with debug authentications disabled are recalled back into the lab for further testing. The below example assumes control to the debug authentication signals are mapped to a system peripheral register in device memory. Barrier instructions are needed to ensure the changes take effect immediately.

The barriers requirement when changing debug authentication signals is the same as any memory-mapped peripheral register.

```
// The debug authentication control register in some device memory
#define EXTDBGAUTHCTRL_REG (*(volatile uint32_t *)0x<example address>)

void setDebugAuthentication(uint32_t auth_state) {
    // Write the desired state to the EXTDBGAUTHCTRL register
EXTDBGAUTHCTRL_REG = auth_state;
    // Barrier instructions to ensure completion and
    // synchronization
__DSB();
__ISB();
}
```

3. 2: Synchronization

Programs combine memory access and data processing operations to form compound operations, frequently creating read-modify-write sequences. When code running on multiple CPUs, on different threads, or exception handlers access the same memory structures, atomicity of these read-modify-write sequences is often required to avoid data corruption.

3.1 2.1: The need for exclusive access

In Armv8-M Architecture, exclusive access instructions such as LDREX, STREX, and their variants support atomic operations. They work by detecting if another thread or CPU modifies a memory location during a read-modify-write attempt.

The read-modify-write can then be aborted and retried. With the help of a Local Exclusive Monitor and optionally a Global Exclusive Monitor, this mechanism enables creation of software semaphores and other synchronization primitives without requiring explicit bus locking. For a detailed description of the mechanics of Armv8-M exclusive accesses, see the "Exclusive accesses" section of the Armv8-M Memory Model and Memory Protection User Guide.

3.2 2.2: Armv8-M exclusive access instructions

The Armv8-M architecture includes Load-Exclusive, Store-Exclusive, and Clear-Exclusive instructions, in byte, halfword, and word variants.

Load-Exclusive:

Byte	LDREXB
Halfword	LDREXH
Word	LDREX

Store-Exclusive:

Byte	STREXB
Halfword	STREXH
Word	STREX

Clear-Exclusive (no other variants):

CLREX

Most commonly, software developers choose to use the CMSIS-provided intrinsic functions for generation of these instructions:

LDREX	`uint32_t __LDREXW (uint32_t *addr)`

LDREXB	`uint8_t __LDREXB (uint8_t *addr)`
LDREXH	`uint16_t __LDREXH (uint16_t *addr)`
STREX	`uint32_t __STREXW (uint32_t value, uint32_t *addr)`
STREXB	`uint8_t __STREXB (uint8_t value, uint8_t *addr)`
STREXH	`uint16_t __STREXH (uint16_t value, uint16_t *addr)`
CLREX	`void __CLREX (void)`

The Armv8-M exclusive instructions implement a “measured-atomicity” methodology. This means that a Store-Exclusive instruction will fail if another processor or memory observer has modified the memory location. The program must test and see whether any Store-Exclusive has succeeded by evaluating the return value of the Store-Exclusive instruction.

Arm strongly recommends keeping the Load-Exclusive and the Store-Exclusive operations as close together as possible in a single thread of execution to minimize the chance of the Store-Exclusive instruction failing.

Note: Store-Exclusive CMSIS function variants all return a value of 0 to indicate the store has succeeded.

The clear-exclusive instruction CLREX moves the exclusive monitor to the open exclusive state regardless of which state it is in initially to break any previous load-exclusive instructions.

Example: Atomic counter

In multi-threaded software it is often necessary to keep a global event counter to increment on any incidence of a specific event. Such a counter must ensure that it increments atomically to avoid potential race conditions from different threads.

```
uint32_t atomicIncrement(volatile uint32_t *ptr) {
uint32_t old_val, new_val, status;
do {
    old_val = __LDREXW(ptr);
    new_val = old_val + 1;
    // __STREXW() returns value 0 into “status” on success
    status = __STREXW(new_val, ptr);
} while (status != 0);
return old_val;
}
```

Example: Simple spinlock using exclusive accesses

In multi-threaded or multi-processor implementations, there might be shared peripherals resources, for example, an UART, that must be accessed exclusively. We can create a simple spinlock for threads to get access to before accessing the shared peripheral resource. When done with peripheral, we then call the release function to free the lock so that other threads can access it.

Note: This is not a lock on the specific peripheral resource; the lock is a software contract that all threads with access to the peripheral must abide by.

```
static volatile uint32_t lock = 0;
```

```
void lockAcquire(void) {
    uint32_t status;
    do {
// Reading a non-zero value from the lock variable
// indicates that some other thread has access to the // lock, so wait
        while (__LDREXW(&lock) != 0);
        status = __STREXW(1, &lock);
    } while (status != 0);

// Barrier to ensure subsequent memory operations are
// observed after we've acquired the lock
    __DMB();
}

void lockRelease(void) {
// Barrier to ensure prior accesses are observed before
// releasing the lock
__DMB();
// No need to use store-exclusive because we have already
// claimed the lock and have sole access
    lock = 0;
}
```

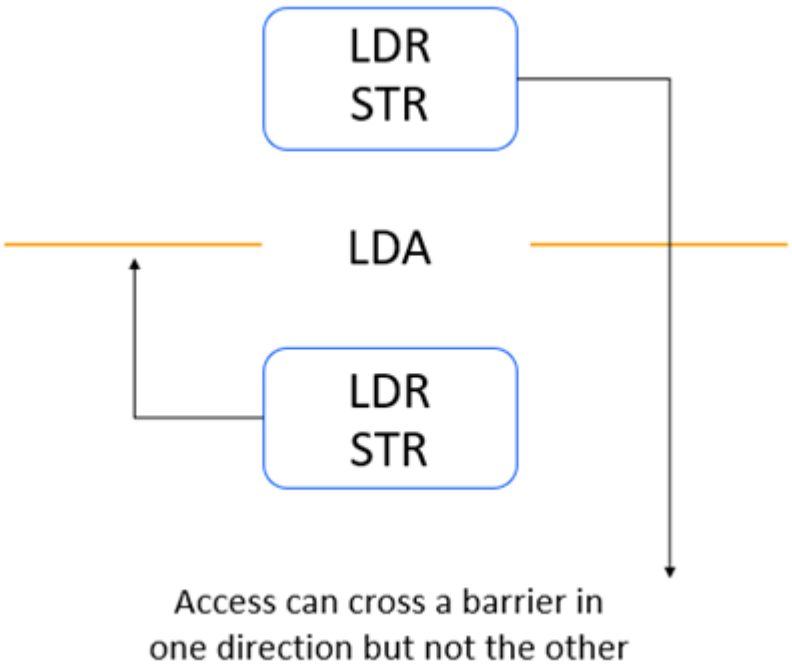
3.3 2.3: Acquire and Release semantics

Armv8-M architecture introduces the Load-Acquire and Store-Release instructions. These are Load and Store instructions augmented with implicit barrier semantics. Load-Acquire and Store-Release instructions contain lighter versions of the DMB and enforce memory ordering in only a single direction.

A Load-Acquire (LDA) instruction ensures that all subsequent loads and stores appearing after the LDA are observed after the LDA. Accesses before the LDA are not affected.

The diagram below shows the effect LDA has on other memory access instructions around it.

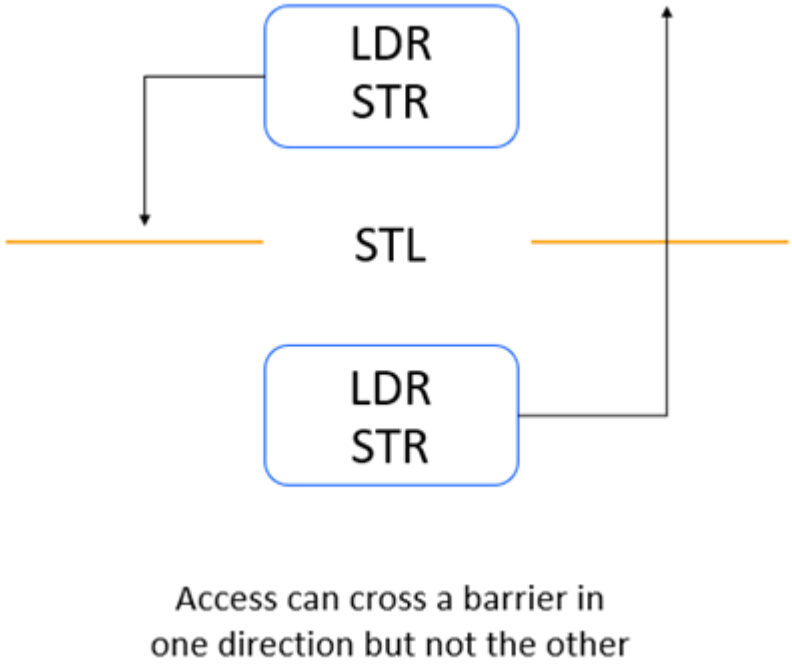
Figure 3-1: Figure 1. Load-Acquire



In a similar but opposite way, a Store-Release (STL) instruction ensures that all prior loads and stores appearing before the STL are observed before the STL. Accesses after the STL are not affected.

The diagram below shows the effect STR has on other memory access instructions around it.

Figure 3-2: Store-Release



You can use LDA and STL instructions in combination to protect critical sections of code, to ensure memory accesses in between the LDA and STL instructions are observed within the LDA and STL.

The diagram below shows the effect of combining LDA and STL to protect critical code.

Figure 3-3: Combining LDA and STL to protect critical code

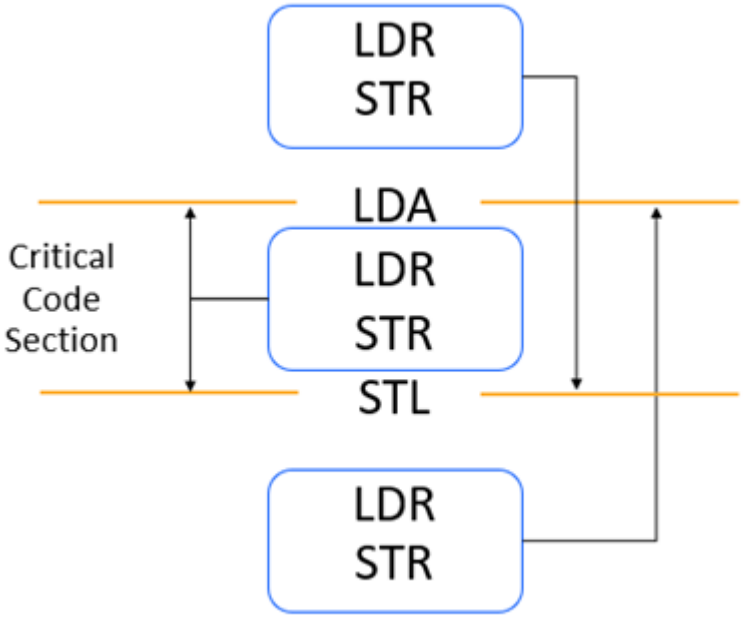


Figure 3: Combining LDA and STL to protect critical code

CMSIS provides the below intrinsics for Load-Acquire and Store-Release instructions and their variants.

Instruction	CMSIS Function
LDA	`uint32_t __LDA (volatile uint32_t * ptr)`
LDAB	`uint8_t __LDAB (volatile uint8_t * ptr)`
LDAH	`uint32_t __LDAH (volatile uint32_t * addr)`
STL	`void __STL (uint32_t value, volatile uint32_t * ptr)`
STLB	`void __STLB (uint8_t value, volatile uint8_t * ptr)`
STLH	`void __STLH (uint16_t value, volatile uint16_t * ptr)`

You can rewrite the example from 1.3.1 Shared memory communication between processors with Load-Acquire instructions instead:

```
// Global variables ready and data placed in the same memory
int ready = 0;
int  data  = 0;
```

Processor A

```
void send(void) {
    data = 10;
    // STL has an implicit uni-direction barrier and will ensure
    // ready is set after write to data is observed
```

```
    __STL(1, &ready);
}
```

Processor B

```
int receive(void) {
    // We use LDA to ensure the return and subsequent memory
    // accesses happen after ready is observed to be 1
    while (__LDA(&ready) != 1);
    return data; // will return value 10
}
```

Because the LDA/STL barriers are only in one direction, the overheads in this example are less than those in the example in 1.3.1 where a DMB is used.

3.4 2.4: Load-Acquire Exclusive and Store-Release Exclusive instructions

The Armv8-M architecture includes instructions that combine the Acquire and Release semantics with exclusive operations.

Load-Exclusive:

Byte	LDAEXB
Halfword	LDAEXH
Word	LDAEX

Store-Exclusive:

Byte	STLEXB
Halfword	STLEXH
Word	STLEX

The exclusive instructions using CMSIS intrinsics are shown below:

Instruction	CMSIS Function
LDAEX	`uint32_t __LDAEX (volatile uint16_t * ptr)`
LDAEXB	`uint8_t __LDAEXB (volatile uint8_t * ptr)`
LDAEXH	`uint16_t __LDAEXH (volatile uint16_t * ptr)`
STLEX	`uint32_t __STLEX (uint32_t value, volatile uint32_t * ptr)`
STLEXB	`uint8_t __STLEXB (uint8_t value, volatile uint8_t * ptr)`
STLEXH	`uint16_t __STLEXH (uint16_t value, volatile uint16_t * ptr)`

The Load-Acquire Exclusive and Store-Release Exclusive instructions, same as their counterparts Load-Acquire and Store-Release instructions, have one-way barrier functionalities built into them.

Example: Simple spinlock using acquire and release semantics

You can rewrite spinlock example in 2.2: Armv8-M exclusive access instructions using acquire and release semantics. The LDAEX ensures that all memory accesses in program order after the lock has been obtained cannot be observed before the lock has been obtained. In the unlock function, the implicit one-way barrier in the Store-Release instruction ensures that the memory accesses that occurred in the critical section after the lock was obtained are observed before the Store-Release instruction. This removes the need for the heavier DMB barriers in the lock and the unlock functions.

```
static volatile uint32_t lock = 0;

void lockAcquire(volatile uint32_t *lock) {
  uint32_t status;
  do {
      // Wait until the lock appears free (lock == 0)
      while (__LDAEX(lock) != 0);
      // Try to atomically set flag from 0 → 1
      status = __STREXW(1, &lock);  // returns 0 on success
  } while (status != 0);
}

void lockRelease(volatile uint32_t *lock) {
// ensure all critical#section accesses are observed before
// the lock is released

__STL(0, lock);

}
```

Example: Publishing nodes to a stack without a lock

In a multi-threaded software example, where different threads share accesses to global data structures, any thread can insert a new node into a stack. To avoid a potential race condition between different threads trying to add a node, you can use a lock. However, the example below shows using exclusives with release semantics to do this with less overhead.

```
typedef struct Node {
    int32_tvalue;
    struct Node *next;
} Node;

static volatile Node *head = NULL;

int32_t list_push(int32_t value)
{
    Node *new = malloc(sizeof(Node));
    if (new == NULL) return -1;
    new->value = value;

    uint32_t status;
    do {
        Node *old = (Node *)__LDREXW((uint32_t *)&head);
        new->next = old;
        status = __STLEX((uint32_t)new, (uint32_t *)&head);
    } while (status != 0);

    return 0;
}
```