ARM® Synchronization Primitives

Development Article

DHT0008A (ID081709)

ARM Synchronization Primitives
Development Article

Release Information

The following changes have been made to this book.

Change history

Date	Issue	Confidentiality	Change
18 August 2009	A	Non-Confidential	Issue 1

Proprietary Notice

Confidentiality Status

Product Status

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

Web Address

`http://www.arm.com`

Contents
ARM Synchronization Primitives Development Article

Chapter 1 **ARM Synchronization Primitives**
1.1 Software synchronization 1-2
1.2 Exclusive accesses 1-4
1.3 Practical uses 1-9

Appendix A **SWP and SWPB**
A.1 Legacy synchronization instructions A-2

Appendix B **Revisions**

Chapter 1
ARM Synchronization Primitives

This article introduces the hardware synchronization primitives available in the ARM architecture, and provides examples of how a system-level programmer can use them. It contains the following sections:

- *Software synchronization* on page 1-2
- *Exclusive accesses* on page 1-4
- *Practical uses* on page 1-9

1.1 Software synchronization

When access to a shared resource must be restricted to only one agent at a time, software must be synchronized. Typically, the shared resource is a shared memory location or peripheral device, and the agents might be processors, processes or threads. This synchronization is often managed by *atomically* modifying a variable that holds the current state of the resource. That is, the modification is always entirely successful, or not successful at all. It must also be visible to all agents that might access the variable at the same time.

In a simple system, atomic modifications can be performed safely by disabling interrupts around critical sections of code. In a multitasking or multi-core system, this is not an efficient or safe solution. Modern computer architectures provide hardware *synchronization primitives* as a safe way of atomically updating memory locations.

1.1.1 Software synchronization interfaces

Operating systems or platform libraries hide these low-level hardware primitives from application developers behind hardware-independent.funcions that form part of *Application Programming Interfaces* (APIs).

This article covers the following high-level software synchronization primitives:

Mutex A variable, able to indicate the two states `locked` and `unlocked`. Attempting to lock a mutex already in the locked state blocks execution until the agent holding the mutex unlocks it. Mutexes are sometimes called *locks* or *binary semaphores*.

Semaphore A counter that can be atomically incremented and decremented. Attempting to decrement a semaphore that holds a value of less than 1 blocks execution until another agent increments the semaphore.

In addition to the blocking operations, an API can define non-blocking variants. A non-blocking function returns an error condition instead of blocking if it fails to perform the requested action.

1.1.2 Synchronization in a multitasking system

In a multitasking operating system, any synchronization operation must be guaranteed to behave correctly even if interrupted by a context switch. When no synchronization with other processors to is required, software can achieve this by disabling interrupts while it updates the synchronization variable. This can be a useful method for implementing synchronization in the operating system kernel, but the performance overhead of a system call makes this an impractical solution for application software. Also, it is not a good solution when low interrupt latency is important.

1.1.3 Synchronization in a multi-processor system

Multi-core and multi-processor systems introduce a new problem, because they can require a mutex to be locked, or a semaphore to be modified, atomically across the whole system. This might require the system to maintain global state that tracks active synchronization operations.

1.1.4 Historical synchronization primitives in the ARM architecture

The `SWP` and `SWPB` instructions atomically swap a 32-bit word or a byte between a register and memory. From the ARMv6 architecture, ARM deprecates the use of `SWP` and `SWPB`. This means that future architectures are not guaranteed to support these instructions. ARM strongly recommends that all software use the new synchronization primitives described in this article. For developers targeting older systems, Appendix A *SWP and SWPB* provides some information about these instructions.

1.1.5 Additions in ARMv6 architecture

The ARMv6 architecture introduced the concept of *exclusive accesses* to memory locations, providing more flexible atomic memory updates. *Exclusive accesses* on page 1-4 describes the new instructions and architectural concepts.

It also introduced the concepts of memory types, memory access ordering rules, and barrier instructions for explicit ordering of memory accesses. For information about these concepts, see the ARM Architecture Reference Manual for your architecture version and profile. This document is available on request from `http://infocenter.arm.com`.

1.2 Exclusive accesses

The ARMv6 architecture introduced Load Link and Store Conditional instructions in the form of the Load-Exclusive and Store-Exclusive synchronization primitives, LDREX and STREX. From ARMv6T2, these instructions are available in the ARM and Thumb instruction sets. Load-Exclusive and Store-Exclusive provide flexible and scalable synchronization, superseding the deprecated SWP and SWPB instructions.

1.2.1 LDREX and STREX

The LDREX and STREX instructions split the operation of atomically updating memory into two separate steps. Together, they provide atomic updates in conjunction with *exclusive monitors* that track exclusive memory accesses, see *Exclusive monitors* on page 1-5. Load-Exclusive and Store-Exclusive must only access memory regions marked as Normal.

LDREX

The LDREX instruction loads a word from memory, initializing the state of the exclusive monitor(s) to track the synchronization operation. For example, `LDREX R1, [R0]` performs a Load-Exclusive from the address in R0, places the value into R1 and updates the exclusive monitor(s).

STREX

The STREX instruction performs a conditional store of a word to memory. If the exclusive monitor(s) permit the store, the operation updates the memory location and returns the value 0 in the destination register, indicating that the operation succeeded. If the exclusive monitor(s) do not permit the store, the operation does not update the memory location and returns the value 1 in the destination register. This makes it possible to implement conditional execution paths based on the success or failure of the memory operation. For example, `STREX R2, R1, [R0]` performs a Store-Exclusive operation to the address in R0, conditionally storing the value from R1 and indicating success or failure in R2.

Alternative exclusive access sizes

The ARMv6K architecture introduced byte, halfword and doubleword variants of LDREX and STREX:

- LDREXB and STREXB
- LDREXH and STREXH
- LDREXD and STREXD.

The ARMv7 architecture added these to the Thumb instruction set in the A and R profiles. ARMv7-M supports the byte and halfword but not the doubleword variants. ARMv6-M does not support exclusive accesses.

The architecture requires that each Load-Exclusive instruction must be used only with the corresponding Store-Exclusive instruction, for example `LDREXB` must only be used with `STREXB`.

1.2.2 Exclusive monitors

An exclusive monitor is a simple state machine, with the possible states *open* and *exclusive*. To support synchronization between processors, a system must implement two sets of monitors, local and global. A Load-Exclusive operation updates the monitors to exclusive state. A Store-Exclusive operation accesses the monitor(s) to determine whether it can complete successfully. A Store-Exclusive can succeed only if all accessed exclusive monitors are in the exclusive state.

Figure 1-1 shows an example system consisting of one Cortex™-A8 processor, one Cortex-R4 processor, and a memory device shared between the two.

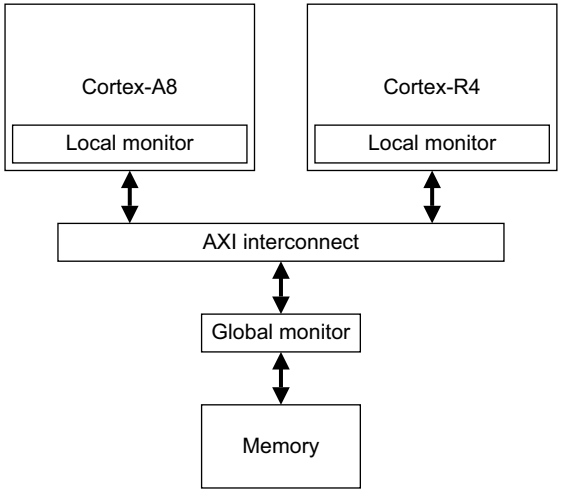

Figure 1-1 Local and global monitors in a multi-core system

Local monitors

Each processor that supports exclusive accesses has a local monitor. Exclusive accesses to memory locations marked as Non-shareable are checked only against this local monitor. Exclusive accesses to memory locations marked as Shareable are checked against both the local monitor and the global monitor.

For example, if software executing on the Cortex-A8 processor in Figure 1-1 on page 1-5 must enforce synchronization between applications executing locally, it can do this using a mutex placed in Non-shareable memory. The resulting Load-Exclusive and Store-Exclusive instructions only access the local monitor.

A local monitor can be implemented to tag an address for exclusive use, or it can contain a state machine that only tracks the issuing of Load-Exclusive and Store-Exclusive instructions. This means a Store-Exclusive to a Shareable location might succeed even if the preceding Load-Exclusive was from a completely different location. For this reason, portable code must not make assumptions about exclusive accesses performing address checking.

Note

If the location is cacheable, the synchronization might take place without any external bus transactions, and without the result being visible to external observers, for example other processors in the system.

The global monitor

A global monitor tracks exclusive accesses to memory regions marked as Shareable. Any Store-Exclusive operation that targets Shareable memory must check its local monitor and the global monitor to determine whether it can update memory.

For example, if software executing on one processor in Figure 1-1 on page 1-5 must synchronize its operation with software executing on the other processor, it can do this using a mutex placed in Shareable memory. The resulting Load-Exclusive and Store-Exclusive instructions access both the local monitor and the global monitor.

It is also possible for a global monitor, or part of the global monitor, to be implemented combined with the local monitor, for example in a system implementing cache coherency management. See *Use in multi-core systems* on page 1-8.

The global monitor can tag one address for each processor in the system that supports exclusive accesses. When a processor performs a Load-Exclusive to a Shareable location, the global monitor tags the accessed address for exclusive use by that processor. The following events reset the global monitor entry for processor *N* to open state:

- processor N performs an exclusive load from a different location
- a different processor successfully performs a store, or a Store-Exclusive, to the location tagged for exclusive use by processor N.

Other events can clear a global exclusive monitor, but they are implementation defined and portable code must not rely on them.

Note

If a region configured as Shareable is not associated with a global monitor, Store-Exclusive operations to that region always fail, returning 0 in the destination register.

Exclusives Reservation Granule

When an exclusive monitor tags an address, the minimum region that can be tagged for exclusive access is called the *Exclusives Reservation Granule* (ERG). The ERG is implementation defined, in the range 8-2048 bytes, in multiples of two bytes. Portable code must not assume anything about ERG size.

Resetting monitors

When an operating system performs a context switch, it must reset the local monitor to open state, to prevent false positives occurring. ARMv6K introduced the Clear-Exclusive instruction, `CLREX`, to reset the local monitor.

Note

In ARMv6 base architecture and ARMv6T2, the local monitor must be reset by performing a dummy Store-Exclusive to a dedicated address.

The state of monitors is architecturally undefined after a Data Abort exception. Therefore, ARM recommends that the exception handling code executes a `CLREX` or dummy Store-Exclusive instruction.

If a context switch schedules out a process after the process has performed a Load-Exclusive but before it performs the Store-Exclusive, the Store-Exclusive returns a false negative result when the process resumes, and memory is not updated. This does not affect program functionality, because the process can retry the operation immediately.

For these reasons ARM recommends that:

- the Load-Exclusive and Store-Exclusive are no more than 128 bytes apart
- no explicit cache maintenance operations or data accesses are performed between the Load-Exclusive and the Store-Exclusive.

1.2.3 Memory barriers

To ensure a consistent memory view, it is architecturally defined that software must perform a *Data Memory Barrier* (DMB) operation:

- between acquiring a resource, for example through locking a mutex or decrementing a semaphore, and making any access to that resource
- before making a resource available, for example through unlocking a mutex or incrementing a semaphore.

Note

The Data Memory Barrier existed before ARMv7 as a cp15 operation, but ARMv7 introduced a dedicated instruction, `DMB`.

1.2.4 Use in multi-core systems

The model for using Load-Exclusives and Store-Exclusives for synchronization is the same for single-core and multi-core systems, but in a multi-core system there are some system-wide implications you must be aware of.

Systems with coherency management

ARM MPCore™ multi-core processors contain a *Snoop Control Unit* (SCU), that maintains Level 1 data cache coherency across memory regions shared by the processors. In this setup, the local monitor of each core operate together with the SCU to provide a combined local and global monitor for synchronization operations in those regions marked as coherent.

Note

This might occasionally lead to false negatives, or delays caused by transferring data between caches when several processors attempt to access synchronization variables within the same ERG block at the same time. For performance reasons, it might be worth to explicitly place very frequently accessed synchronization variables at least the size of the ERG apart in memory.

Systems without coherency management

Memory regions used for synchronization operations between processors, or cores in a multi-core processor, must be marked as Shareable. When coherency management is not available or disabled, this means that such regions cannot be cached, and a global monitor must be implemented to permit synchronization.

1.3 Practical uses

This section gives examples of using the exclusive access hardware synchronization primitives to implement a simple mutex and semaphore. The assembly language examples are written in the format accepted by the ARM RealView® Compilation Tools assembler, `armasm`.

1.3.1 Power-saving features

If a piece of code fails to lock a mutex, or to decrement a semaphore, it can either:

- retry until successful
- return an error code, indicating that the operation could not succeed at this time.

In many situations, you want the function to return only when it has acquired the lock. However, looping and retrying consumes power without performing any useful work. Because it is not possible to acquire the resource until an external agent modifies the synchronization variable, a better solution is to put the processor into a low-power state, or to request that the operating system schedules in a new process, and retry at a later point.

It can also be useful to have a form of mutex and semaphore operations that return an error code, making it possible for the application to perform alternative actions if it fails to acquire a specific resource.

Wait For Interrupt

ARMv6K introduced the Wait For Interrupt, `WFI`, instruction. `WFI` is a hint, and might not have any effect on some processors. On processors that implement the behavior, WFI indicates that the processor can enter a low-power state until a wake-up event occurs.

The wake-up events for WFI are:

- an interrupt, even if masked
- an asynchronous abort
- a debug event, when invasive debug is enabled and permitted in the current state.

In an operating system, the scheduler is often invoked by an exception handler based on an interrupt triggering. When you execute `WFI` in such an environment, this can put your processor into low-power state for the remainder of its execution slot. When the timer interrupt triggers, the scheduler is invoked and context switches in a different process. The next time the process is scheduled to run, it can retry to acquire the resource, and go back to `WFI` if it fails.

Note

WFI existed as a CP15 operation in many earlier processors. ARMv7 redefines the CP15 operation as a NOP.

Example 1-1 shows the `WAIT_FOR_UPDATE` and `SIGNAL_UPDATE` macros implemented using the `WFI` instruction.

Example 1-1 Power-saving macros using WFI

```
MACRO
WAIT_FOR_UPDATE
WFI                    ; Indicate opportunity to enter low-power state
MEND

MACRO
SIGNAL_UPDATE          ; No software signalling operation
MEND
```

Wait For Event and Send Event

ARMv6K also introduced the Wait For Event, `WFE`, and Send Event, `SEV` instructions. These are hints, and might not have any effect on some processors. On processors that implement the behavior, `WFE` indicates that the processor can enter a low-power state until a wake-up event occurs. Portable code must not rely on `WFE` blocking execution.

The wake-up events for WFE are:

- the execution of an SEV instruction on any processor in a multi-core system
- an interrupt, unless masked
- an asynchronous abort, for example a buffered write generating an access fault
- a debug event, when invasive debug is enabled and permitted in the current state.

In an operating system, the scheduler is often invoked by an exception handler based on an interrupt triggering. When you execute `WFE` in such an environment, this can put your processor into low-power state for the remainder of its execution slot. When the timer interrupt triggers, the scheduler is invoked and performs a context switch. The next time the process is scheduled to run, it can retry to acquire the resource, and go back to `WFE` if it fails.

This is the only use for these instructions. Software must not use them for synchronization.

Example 1-2 shows the WAIT_FOR_UPDATE and SIGNAL_UPDATE macros implemented using the WFE and SEV instructions.

Example 1-2 Power-saving macros using SEV/WFE

```
MACRO
WAIT_FOR_UPDATE
WFE                     ; Indicate opportunity to enter low-power state
MEND

MACRO
SIGNAL_UPDATE
DSB                     ; Ensure update has completed before signalling
SEV                     ; Signal update
MEND
```

The SIGNAL_UPDATE macro begins with a Data Synchronization Barrier to ensure that the update to the synchronization variable is visible to all processors before SEV is executed.

Rescheduling as a power-saving feature

If your target operating system provides a way for a process to yield execution by invoking the scheduler manually, this might be the best action to take when acquiring a resource fails, especially in a multi-core system. This means that instead of the blocked process waiting for a resource until it is made available, or until the next time the scheduler is invoked, a different process can execute in its stead, potentially increasing the system responsiveness. This also enables the system to get the total work done faster, so that it can return to a low-power state.

Example 1-3 shows the WAIT_FOR_UPDATE and SIGNAL_UPDATE macros implemented using system calls. It does not use a real system call interface, it only illustrates how one might look.

Example 1-3 Power-saving macros using system calls

```
MACRO
WAIT_FOR_UPDATE
PUSH {r0-r3,r7,r12}     ; Push AAPCS corruptible registers, plus r7
LDR  r7, =yield         ; Parameter to SVC call passed in r7
SVC  #0
POP  {r0-r3,r7,r12}
MEND

MACRO
```

```
    SIGNAL_UPDATE
    PUSH {r0-r3,r7,r12}    ; Push AAPCS corruptible registers, plus r7
    LDR  r7, =released     ; Parameter to SVC call passed in r7
    SVC  #0
    POP  {r0-r3,r7,r12}
    MEND
```

Both macros:

- stack the registers that might be corrupted by the SVC handler
- load a system call id into R7, indicating to the SVC handler what operation to perform
- execute an SVC instruction to trigger a supervisor call exception
- restore the stacked registers.

If the address of the synchronization variable is in R0, the operating system could make use of this to ensure the blocked process does not execute again until an update is signalled for that specific variable.

1.3.2 Implementing a mutex

The functions in Example 1-4 show an implementation of a simple blocking mutex:

- lock_mutex acquires a mutex, blocking indefinitely until it acquires it. If blocked, it invokes the WAIT_FOR_UPDATE macro before retrying.
- unlock_mutex releases a mutex, invoking the SIGNAL_UPDATE macro to notify waiting processes or processors of the change.

WAIT_FOR_UPDATE and SIGNAL_UPDATE are described in *Power-saving features* on page 1-9.

Example 1-4 implementing a mutex

```
locked    EQU 1
unlocked  EQU 0

; lock_mutex
; Declare for use from C as extern void lock_mutex(void * mutex);
    EXPORT lock_mutex
lock_mutex PROC
    LDR     r1, =locked
1   LDREX   r2, [r0]
    CMP     r2, r1         ; Test if mutex is locked or unlocked
    BEQ     %f2            ; If locked - wait for it to be released, from 2
    STREXNE r2, r1, [r0]   ; Not locked, attempt to lock it
    CMPNE   r2, #1         ; Check if Store-Exclusive failed
    BEQ     %b1            ; Failed - retry from 1
```

```
    ; Lock acquired
    DMB                 ; Required before accessing protected resource
    BX      lr

2   ; Take appropriate action while waiting for mutex to become unlocked
    WAIT_FOR_UPDATE
    B       %b1         ; Retry from 1
    ENDP


; unlock_mutex
; Declare for use from C as extern void unlock_mutex(void * mutex);
    EXPORT unlock_mutex
unlock_mutex PROC
    LDR     r1, =unlocked
    DMB                 ; Required before releasing protected resource
    STR     r1, [r0]    ; Unlock mutex
    SIGNAL_UPDATE
    BX      lr
    ENDP
```

The mutex variable passed to these functions must be 32 bits in size, located at a 4-byte aligned address. You must initialize it to `locked` or `unlocked` before first use.

`lock_mutex` performs a Load-Exclusive from the address passed in `R0`. If this location holds the value `locked`, the function invokes `WAIT_FOR_UPDATE` before retrying. If the location holds any other value, it performs a Store-Exclusive of the value `locked`. If the Store-Exclusive fails, the function retries immediately from the Load-Exclusive step. When the Store-Exclusive succeeds, it executes a `DMB` and returns.

`unlock_mutex` stores the value `unlocked` to the address passed in `R0`. Because only a thread or process currently holding the mutex must unlock it, it can use a normal `STR` for this operation. However, it must execute a `DMB` before updating the mutex location. It then invokes `SIGNAL_UPDATE`, to notify any blocked processes or processors, and returns.

Example 1-5 shows how you can call these functions from C source code. The function `putstr` uses the mutex `output_mutex` to ensure than the entire string passed in `str` is printed together, even if several threads call it simultaneously. It returns the total number of characters printed from `str`.

Example 1-5 synchronizing text printout

```
#define locked   1
#define unlocked 0

extern void lock_mutex(void * mutex);
```

```
extern void unlock_mutex(void * mutex);

unsigned int output_mutex = unlocked;

int putstr(char * str)
{
    int i;

    /* Wait until the output mutex is acquired */
    lock_mutex(&output_mutex);

    /* Entered critical section */

    /* Output each individual character from str */
    for(i=0 ; str[i] != '\0' ; i++) {
        putc(str[i]);
    }

    /* Leave critical section - release output mutex */
    unlock_mutex(&output_mutex);

    return i;
}
```

1.3.3 Implementing a semaphore

The functions in Example 1-6 show an implementation of a simple blocking semaphore:

- `sem_dec` decrements a semaphore if its value is greater than 0, or blocks until it is able to decrement it. If blocked, it invokes the `WAIT_FOR_UPDATE` macro before retrying.
- `sem_inc` increments a semaphore, invoking the `SIGNAL_UPDATE` macro to notify blocked processes or processors of the change if the previous value was 0.

`WAIT_FOR_UPDATE` and `SIGNAL_UPDATE` are described in *Power-saving features* on page 1-9.

Example 1-6 implementing a semaphore

```
; sem_dec
; Declare for use from C as extern void sem_dec(void * semaphore);
    EXPORT sem_dec
sem_dec PROC
1   LDREX   r1, [r0]
    CMP     r1, #0        ; Test if semaphore holds the value 0
    BEQ     %f2           ; If it does, block before retrying
```

```
    SUB     r1, #1        ; If not, decrement temporary copy
    STREX   r2, r1, [r0]  ; Attempt Store-Exclusive
    CMP     r2, #0        ; Check if Store-Exclusive succeeded
    BNE     %b1           ; If Store-Exclusive failed, retry from start
    DMB                   ; Required before accessing protected resource
    BX      lr

2   ; Take appropriate action while waiting for semaphore to be incremented
    WAIT_FOR_UPDATE       ; Wait for signal to retry
    B       %b1
    ENDP


; sem_inc
; Declare for use from C as extern void sem_inc(void * semaphore);
    EXPORT sem_inc
sem_inc PROC
1   LDREX   r1, [r0]
    ADD     r1, #1        ; Increment temporary copy
    STREX   r2, r1, [r0]  ; Attempt Store-Exclusive
    CMP     r2, #0        ; Check if Store-Exclusive succeeded
    BNE     %b1           ; Store failed - retry immediately
    CMP     r0, #1        ; Store successful - test if incremented from zero
    DMB                   ; Required before releasing protected resource
    BGE     %f2           ; If initial value was 0, signal update
    BX      lr

2   ; Signal waiting processors or processes
    SIGNAL_UPDATE
    BX      lr
    ENDP
```

The semaphore variable passed to these functions must be 32 bits in size, located at a 4-byte aligned address. You must initialize it to a value greater than or equal to 0 before first use.

`sem_dec` performs a Load-Exclusive from the address passed in `R0`. If this location holds the value 0, the function invokes `WAIT_FOR_UPDATE` before retrying. If the location holds a value greater than zero, it decrements it and attempts to update the semaphore value using a Store-Exclusive. If the Store-Exclusive fails, for example because another agent has modified the variable after the Load-Exclusive step, it retries from the start. When the Store-Exclusive succeeds, it executes a `DMB` and returns.

`sem_inc` performs a Load-Exclusive from the address passed in `R0`. It then increments this value and attempts to update the location using a Store-Exclusive. If the Store-Exclusive fails, it retries immediately from the Load-Exclusive step. If the Store-Exclusive succeeds, it checks what the value of the semaphore was before

incremented, and then executes a DMB. If the value was 0, there could be processes or processors waiting for this semaphore to be incremented, so it invokes the SIGNAL_UPDATE macro before returning.

Example 1-7 shows how you can call these functions from C source code. The function add_task.adds an object of type struct task to a queue and calls sem_inc to atomically increment task_semaphore to signal that a task has been added. The function get_task calls sem_dec to atomically decrement task_semaphore, or block until it is able to do so, and then takes an object off the queue.

Example 1-7 synchronizing a task queue

```
extern void sem_inc(void * semaphore);
extern void sem_dec(void * semaphore);

unsigned int task_semaphore = 0;

void add_task(struct task * task)
{
    /* Add task to queue */
    ...

    /* Increment semaphore to show task has been added*/
    sem_inc(&task_semaphore);

    return;
}

struct task * void get_task(void)
{
    struct task * tmptask;

    /* Decrement semaphore, or block until it indicates a task is available */
    sem_dec(&task_semaphore);

    /* Take task from queue */
    ...

    return tmptask;
}
```

1.3.4 Lockless programming

Apart from using Load-Exclusives and Store-Exclusives to implement software synchronization primitives, you can also use them to directly update affected data atomically, rather than obtaining a lock to a shared variable. This is often described as *lockless* programming.

For example, a linked list can be implemented using atomic updates of pointers instead of using synchronization variables to protect the pointers while modifying them. The exclusive synchronization primitives in ARMv6K and later architectures support all data sizes that can be natively processed, so are well suited to implement lockless algorithms.

Lockless programming is an advanced topic, beyond the scope of this article. Many independent publications describe the concepts and implementations in great detail.

Appendix A
SWP and SWPB

This appendix describes the legacy SWP and SWPB instructions, used for synchronization on processors based on ARMv5 architecture and earlier. It contains the following section:

- *Legacy synchronization instructions* on page A-2

Note

The SWP and SWPB instructions are deprecated from ARM architecture version 6 onwards. This appendix is included for historical completeness only. Use Load-Exclusive and Store-Exclusive for all new software development for processors implementing ARMv6 or later.

A.1 Legacy synchronization instructions

The SWP and SWPB instructions were used for synchronization before the introduction of exclusive accesses into the ARM architecture. This section describes how they work.

A.1.1 SWP and SWPB

SWP (Swap) and SWPB (Swap Byte) provide a method for software synchronization that does not require disabling interrupts. This is achieved by performing a special type of memory access, reading a value into a processor register and writing a value to a memory location as an atomic operation. Example A-1 shows the implementation of simple mutex functions using the SWP instruction. SWP and SWPB are not supported in the Thumb instruction set, so the example must be assembled for ARM.

Example A-1 binary mutex functions

```
    EXPORT lock_mutex_swp
lock_mutex_swp PROC
    LDR r2, =locked
    SWP r1, r2, [r0]       ; Swap R2 with location [R0], [R0] value placed in R1
    CMP r1, r2             ; Check if memory value was 'locked'
    BEQ lock_mutex_swp     ; If so, retry immediately
    BX  lr                 ; If not, lock successful, return
    ENDP

    EXPORT unlock_mutex_swp
unlock_mutex_swp
    LDR r1, =unlocked
    STR r1, [r0]           ; Write value 'unlocked' to location [R0]
    BX  lr
    ENDP
```

In the SWP instruction in Example A-1, R1 is the destination register that receives the value from the memory location, and R2 is the source register that is written to the memory location. You can use the same register for destination and source.

The requirements for memory barriers mentioned in *Memory barriers* on page 1-8 still apply for processors implementing architecture versions earlier than ARMv6. Where required, use the Drain Write Buffer or Drain Store Buffer CP15 operation on processors implementing versions of the architecture earlier than ARMv6.

A.1.2 Limitations of SWP and SWPB

If an interrupt triggers while a swap operation is taking place, the processor must complete both the load and the store part of the instruction before taking the interrupt, increasing interrupt latency. Because Load-Exclusive and Store-Exclusive are separate instructions, this effect is reduced when using the new synchronization primitives.

In a multi-core system, preventing access to main memory for all processors for the duration of a swap instruction can reduce overall system performance. This is especially true in a multi-core system where processors operate at different frequencies but share the same main memory.

Because of these problems, ARMv6 and later deprecate using `SWP` and `SWPB`. The *Multiprocessor Extensions* to ARMv7 introduce the `SW` bit in the CP15 System Control Register. On processors that implement these extensions, after power-up or a reset, software must set this bit to 1 to enable use of the `SWP` and `SWPB` instructions.

Appendix B
Revisions

This appendix describes the technical changes between released issues of this book.

Table B-1 Issue A

Change	Location	Affects
First release	-	-