

```
1  __attribute__((section(".isr_vector")))
2  void (* const g_pfnVectors[])(void) =
3  {
4      (void (*)(void)) (0x20000000 + 64*1024), // 64k of RAM starting at 0x20000000
5                                          // The initial stack pointer
6      ResetISR,                          // The reset handler
7      NmiISR,                             // The NMI handler
8      FaultISR,                           // The hard fault handler
9      IntDefaultHandler,                  // The MPU fault handler
10     IntDefaultHandler,                   // The bus fault handler
11     IntDefaultHandler,                   // The usage fault handler
12     0,                                  // Reserved
13     0,                                  // Reserved
14     0,                                  // Reserved
15     0,                                  // Reserved
16     SVCHandler,                          // SVC call handler
17     IntDefaultHandler,                   // Debug monitor handler
18     0,                                  // Reserved
19     IntDefaultHandler,                   // The PendSV handler
20     /* etc. */
```

Jul 05, 10 12:39

svcdemo.c

Page 1/1

```

1 // You must write an exception handler for the SVC exception
2 // that calls handleSVC() with the 8-bit integer encoded in
3 // the SVC instruction.
4 void handleSVC(int code)
5 {
6 // NOTE: iprintf() is a bad idea inside an exception
7 // handler (exception handlers should be small and short).
8 // But this is the easiest way to show we got it right.
9 switch (code & 0xFF) {
10     case 123:
11         iprintf("Do the 123 thing\r\n");
12         break;
13
14     case 234:
15         iprintf("Do the 234 thing\r\n");
16         break;
17
18     default:
19         iprintf("UNKNOWN SVC CALL\r\n");
20         break;
21 }
22 }
23
24 extern void SVCHandler(void) __attribute__((naked));
25 void SVCHandler(void)
26 {
27     asm volatile (
28 " LDR R0, [R13, #24] @ R0 <-- AddressToReturnTo\r\n"
29 " SUB R0, R0, #2 @ R0 <-- Address of SVC instruction\r\n"
30 " LDRH R0, [R0] @ R0 <-- 16-bit encoding of SVC instruction\r\n"
31 " B handleSVC @ R0 is first parameter to handleSVC()\r\n"
32 " @ Note that 'bx lr' at the end of handleSVC()\r\n"
33 " @ serves to return from the SVC exception.\r\n"
34 );
35 }
36
37 /*
38 NOTES:
39 This works well enough when the entire application is using a
40 single stack (the Main stack).
41
42 How would this code need to change if threads are using the Process
43 Stack? Remember: exception handlers always run using the Main Stack,
44 but automatically-saved registers (including AddressToReturnTo) are
45 saved on the Thread's stack (i.e., the Process Stack).
46
47 Thus, this line is no longer valid:
48
49 LDR R0, [R13, #24] @ WRONG R13!!!!
50 */

```

```
1 New Ways of Structuring Code
2 =====
3
4 E. Time-sliced threads, with interrupts to handle hardware events and
5 global variables used to communicate from ISR's to each thread,
6 and between threads
7
8 Advantages:
9 * No need to explicitly yield to a scheduler...the scheduler forcibly
10 interrupts (i.e., pre-empts) a thread when its "time slice" is up.
11 This is a natural use for the SysTick timer.
12 * The code is minimally intertwined and there are only well-defined
13 interfaces between threads<-->scheduler and thread<-->thread.
14 * If time slices are small enough, the system provides the "appearance
15 of concurrency" and has very good latency and responsivity.
16
17 Disadvantages:
18 * Complex: interrupt-driven system is easy to get wrong, debugging is
19 difficult.
20 * Each thread needs its own stack.
21 * The "concurrency problem". When program state is saved/restored at
22 an explicit function call boundary (i.e., yield) then behavior is
23 (mostly) deterministic. But when program state is saved/restored
24 at any assembly-language instruction boundary, then behavior is
25 non-deterministic and previously-true assumptions no longer hold.
26
27 What is this "concurrency problem"???
```

May 07, 10 11:39

main.c

Page 1/1

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <pthread.h>
5
6  unsigned buffer[65536];
7  #define BUF_SIZE (sizeof(buffer)/sizeof(buffer[0]))
8  volatile unsigned head = 0;
9  volatile unsigned tail = 0;
10 volatile unsigned count = 0;
11
12 void *T1(void *arg)
13 {
14     unsigned token = 0;
15
16     while (1) {
17         if (count < BUF_SIZE) {
18             count++;
19             buffer[head++] = token++;
20             if (head >= BUF_SIZE) head=0;
21         } else usleep(10);
22     }
23 }
24
25 void *T2(void *arg)
26 {
27     unsigned i = 0;
28     unsigned lastval = -1, newval;
29     int failflag = -1;
30
31     while (1) {
32         if (count) {
33             printf("%8u ", (newval=buffer[tail++]));
34             if (tail >= BUF_SIZE) tail=0;
35             count--;
36
37             if (++i == 8) {
38                 printf("|%u\n", count);
39                 i = 0;
40             }
41
42             if (failflag > 0) {
43                 if (--failflag == 0) {
44                     exit(0);
45                 }
46             } else if (newval != lastval+1) {
47                 failflag = 32;
48             } else {
49                 lastval = newval;
50             }
51
52         } else usleep(10);
53     }
54 }
55
56 int main(void)
57 {
58     pthread_t thread1, thread2;
59
60     pthread_create(&thread1, 0, T1, 0);
61     pthread_create(&thread2, 0, T2, 0);
62
63     while (1) ;
64 }

```

Jul 05, 10 12:19

failure.txt

Page 1/1

1	Failure #1:								
2	104288	104289	104290	104291	104292	104293	104294	104295	65525
3	104296	104297	104298	104299	104300	104301	104302	104303	65517
4	104304	104305	104306	104307	104308	104309	104310	104311	65531
5	104312	104313	104314	104315	104316	104317	104318	104319	65523
6	104320	104321	104322	104323	104324	104325	104326	104327	65515
7	104328	104329	104330	104331	104332	104333	104334	104335	65524
8	169872	169873	104338	104339	104340	104341	104342	104343	65528
9	169880	169881	104346	104347	104348	104349	104350	104351	65528
10	104352	104353	104354	104355	104356	104357	104358	104359	65520
11	104360	104361	104362	104363	104364	104365	104366	104367	65512
12									
13	Failure #2:								
14	169960	169961	169962	169963	169964	169965	169966	169967	65513
15	169968	169969	169970	169971	169972	169973	169974	169975	65528
16	169976	169977	169978	169979	169980	169981	169982	169983	65520
17	169984	169985	169986	169987	169988	169989	169990	169991	65512
18	169992	169993	169994	169995	169996	169997	169998	169999	65504
19	170000	170001	170002	170003	170004	170005	170006	170007	65496
20	170008	170009	170010	170011	170012	170013	170014	170015	65535
21	235552	235553	235554	235555	235556	235557	235558	235559	65527
22	235560	235561	235562	235563	235564	235565	235566	235567	65519
23	235568	235569	235570	235571	235572	235573	235574	235575	65533
24									
25	Failure #3:								
26	41368	41369	41370	41371	41372	41373	41374	41375	65520
27	41376	41377	41378	41379	41380	41381	41382	41383	65528
28	41384	41385	41386	41387	41388	41389	41390	41391	65520
29	41392	41393	41394	41395	41396	41397	41398	41399	65528
30	41400	41401	41402	41403	41404	41405	41406	41407	65520
31	41408	41409	41410	41411	41412	41413	41414	41415	65512
32	106952	106953	106954	106955	106956	106957	106958	106959	65528
33	106960	106961	41426	41427	41428	41429	41430	41431	65520
34	106968	106969	106970	106971	106972	106973	106974	106975	65528
35	106976	106977	106978	106979	106980	106981	106982	106983	65529
36									

Critical section

From Wikipedia, the free encyclopedia

In concurrent programming a **critical section** is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution. A critical section will usually terminate in fixed time, and a thread, task or process will have to wait a fixed time to enter it (aka bounded waiting). Some synchronization mechanism is required at the entry and exit of the critical section to ensure exclusive use, for example a semaphore.

By carefully controlling which variables are modified inside and outside the critical section (usually, by accessing important state only from within), concurrent access to that state is prevented. A critical section is typically used when a multithreaded program must update multiple related variables without a separate thread making conflicting changes to that data. In a related situation, a critical section may be used to ensure a shared resource, for example a printer, can only be accessed by one process at a time.

How critical sections are implemented varies among operating systems.

The simplest method is to prevent any change of processor control inside the critical section. On uni-processor systems, this can be done by disabling interrupts on entry into the critical section, avoiding system calls that can cause a context switch while inside the section and restoring interrupts to their previous state on exit. Any thread of execution entering any critical section anywhere in the system will, with this implementation, prevent any other thread, including an interrupt, from getting the CPU and therefore from entering any other critical section or, indeed, any code whatsoever, until the original thread leaves its critical section.

This brute-force approach can be improved upon by using semaphores. To enter a critical section, a thread must obtain a semaphore, which it releases on leaving the section. Other threads are prevented from entering the critical section at the same time as the original thread, but are free to gain control of the CPU and execute other code, including other critical sections that are protected by different semaphores.

Some confusion exists in the literature about the relationship between different critical sections in the same program.^[*citation needed*] In general, a resource that must be protected from concurrent access may be accessed by several pieces of code. Each piece must be guarded by a common semaphore. Is each piece now a critical section or are all the pieces guarded by the same semaphore in aggregate a single critical section? This confusion is evident in definitions of a critical section such as "... a piece of code that can only be executed by one process or thread at a time". This only works if all access to a protected resource is contained in one "piece of code", which requires either the definition of a piece of code or the code itself to be somewhat contrived.

Contents

- 1 Application Level Critical Sections

- 2 Kernel Level Critical Sections
- 3 See also
- 4 External links

Application Level Critical Sections

Application-level critical sections reside in the memory range of the process and are usually modifiable by the process itself. This is called a user-space object because the program run by the user (as opposed to the kernel) can modify and interact with the object. However the functions called may jump to kernel-space code to register the user-space object with the kernel.

Example Code For Critical Sections with POSIX pthread library


```
/* Sample C/C++, Unix/Linux */
#include <pthread.h>

/* This is the critical section object (statically allocated). */
static pthread_mutex_t cs_mutex = PTHREAD_MUTEX_INITIALIZER;

void f()
{
    /* Enter the critical section -- other threads are locked out
    pthread_mutex_lock( &cs_mutex );

    /* Do some thread-safe processing! */

    /*Leave the critical section -- other threads can now pthread_
    pthread_mutex_unlock( &cs_mutex );
}
```



Example Code For Critical Sections with Win32 API

```
/* Sample C/C++, Windows, link to kernel32.dll */
#include <windows.h>

static CRITICAL_SECTION cs; /* This is the critical section object
                             it cannot be moved in memory */
                             /* If you program in OOP, declare this

/* Initialize the critical section before entering multi-threaded
InitializeCriticalSection(&cs);
```

```

void f()
{
    /* Enter the critical section -- other threads are locked out
    EnterCriticalSection(&cs);

    /* Do some thread-safe processing! */

    /* Leave the critical section -- other threads can now EnterCr
    LeaveCriticalSection(&cs);
}

/* Release system object when all finished -- usually at the end o
DeleteCriticalSection(&cs);

```

Note that on Windows NT (not 9x/ME), the function **TryEnterCriticalSection()** can be used to attempt to enter the critical section. This function returns immediately so that the thread can do other things if it fails to enter the critical section (usually due to another thread having locked it). With the pthreads library, the equivalent function is **pthread_mutex_trylock()**. Note that the use of a CriticalSection is not the same as a Win32 Mutex, which is an object used for *inter-process* synchronization. A Win32 CriticalSection is for *intra-process* synchronization (and is much faster as far as lock times), however it cannot be shared across processes.

Kernel Level Critical Sections

Typically, critical sections prevent process and thread migration between processors and the preemption of processes and threads by interrupts and other processes and threads.

Critical sections often allow nesting. Nesting allows multiple critical sections to be entered and exited at little cost.

If the scheduler interrupts the current process or thread in a critical section, the scheduler will either allow the process or thread to run to completion of the critical section, or it will schedule the process or thread for another complete quantum. The scheduler will not migrate the process or thread to another processor, and it will not schedule another process or thread to run while the current process or thread is in a critical section.

Similarly, if an interrupt occurs in a critical section, the interrupt's information is recorded for future processing, and execution is returned to the process or thread in the critical section. Once the critical section is exited, and in some cases the scheduled quantum completes, the pending interrupt will be executed.

Since critical sections may execute only on the processor on which they are entered, synchronization is only required within the executing processor. This allows critical sections to be entered and exited at almost zero cost. No interprocessor synchronization is required, only instruction stream synchronization. Most processors provide the required amount of synchronization by the simple act of interrupting the

current execution state. This allows critical sections in most cases to be nothing more than a per processor count of critical sections entered.

Performance enhancements include executing pending interrupts at the exit of all critical sections and allowing the scheduler to run at the exit of all critical sections. Furthermore, pending interrupts may be transferred to other processors for execution.

Critical sections should not be used as a long-lived locking primitive. They should be short enough that the critical section will be entered, executed, and exited without any interrupts occurring, neither from hardware much less the scheduler.

Kernel Level Critical Sections are the base of the software lockout issue.

See also

- Lock (computer science)

External links

Critical Section documentation on the MSDN Library homepage: <http://msdn2.microsoft.com/en-us/library/ms682530.aspx>

Retrieved from "http://en.wikipedia.org/wiki/Critical_section"

Categories: [Concurrency control](#) | [Programming constructs](#)

- This page was last modified on 8 June 2010 at 18:59.
- Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. See Terms of Use for details.
Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.
- Privacy policy
- About Wikipedia
- Disclaimers

```

1  /*
2   Once a critical section is identified, how do we ensure that no
3   more than one thread is executing it at any one time:
4
5   1. Turn off interrupts before the c.s., turn them on again
6   at the end. This prevents pre-emption of the thread.
7
8   NOTE: You must also NOT make any system/kernel calls within
9   the c.s. as these may cause a context switch.
10  For example, usleep() in Pthreads demo does so.
11
12  2. Serialize access to the c.s. using "clever code". Don't bother
13  trying to write your own -- you will fail because it's
14  REALLY HARD to get right. Fortunately, there's Peterson's Algorithm:
15  */
16
17  int flag[0] = 0; // writable in process0
18  int flag[1] = 0; // writable in process1
19  int turn    = 0; // writable in both processes
20
21  void process0(void)
22  {
23      flag[0] = 1; // "I want the lock..."
24      turn = 1;   // "...but you can have it first"
25
26      // "If you want the lock and it's your turn, I'll wait"
27      while( flag[1] && turn == 1 ) /* NULL */ ;
28
29      // "Either you don't want the lock or it's my turn"
30      // critical section
31      ...
32      // end of critical section
33      flag[0] = 0; // "I don't want the lock anymore"
34  }
35
36  void process1(void)
37  {
38      flag[1] = 1; // "I want the lock..."
39      turn = 0;   // "...but you can have it first"
40
41      // "If you want the lock and it's your turn, I'll wait"
42      while( flag[0] && turn == 0 ) /* NULL */ ;
43
44      // "Either you don't want the lock or it's my turn"
45      // critical section
46      ...
47      // end of critical section
48      flag[1] = 0; // "I don't want the lock anymore"
49  }

```

```
1  /* Code copied from www.yolinux.com, with slight modifications. */
2
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <pthread.h>
6
7  pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
8  int counter = 0;
9
10 void *functionC(void *p)
11 {
12     (void) p;
13     pthread_mutex_lock( &mutex1 );
14     counter++;
15     printf("Counter value: %d\n", counter);
16     pthread_mutex_unlock( &mutex1 );
17 }
18
19 int main(void)
20 {
21     int rc1, rc2;
22     pthread_t thread1, thread2;
23
24     /* Create independent threads each of which will execute functionC */
25
26     if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
27     {
28         printf("Thread creation failed: %d\n", rc1);
29         exit(1);
30     }
31
32     if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
33     {
34         printf("Thread creation failed: %d\n", rc2);
35         exit(1);
36     }
37
38     /* Wait till threads are complete before main continues. Unless we */
39     /* wait we run the risk of executing an exit which will terminate */
40     /* the process and all threads before the threads have completed. */
41
42     pthread_join( thread1, NULL);
43     pthread_join( thread2, NULL);
44
45     return 0;
46 }
47
48 /* Output is:
49     Counter value: 1
50     Counter value: 2
51 */
```

A3.4 Synchronization and semaphores

Exclusive access instructions support non-blocking shared-memory synchronization primitives that allow calculation to be performed on the semaphore between the read and write phases, and scale for multiprocessor system designs.

In ARMv7-M, the synchronization primitives provided are:

- Load-Exclusives:
 - LDREX, see *LDREX* on page A6-106
 - LDREXB, see *LDREXB* on page A6-107
 - LDREXH, see *LDREXH* on page A6-108
- Store-Exclusives:
 - STREX, see *STREX* on page A6-234
 - STREXB, see *STREXB* on page A6-235
 - STREXH, see *STREXH* on page A6-236
- Clear-Exclusive, CLREX, see *CLREX* on page A6-56.

Note

This section describes the operation of a Load-Exclusive/Store-Exclusive pair of synchronization primitives using, as examples, the LDREX and STREX instructions. The same description applies to any other pair of synchronization primitives:

- LDREXB used with STREXB
- LDREXH used with STREXH.

Each Load-Exclusive instruction must be used only with the corresponding Store-Exclusive instruction.

STREXD and LDREXD are not supported in ARMv7-M.

The model for the use of a Load-Exclusive/Store-Exclusive instruction pair, accessing memory address *x* is:

- The Load-Exclusive instruction always successfully reads a value from memory address *x*
- The corresponding Store-Exclusive instruction succeeds in writing back to memory address *x* only if no other processor or process has performed a more recent store of address *x*. The Store-Exclusive operation returns a status bit that indicates whether the memory write succeeded.

A Load-Exclusive instruction tags a small block of memory for exclusive access. The size of the tagged block is IMPLEMENTATION DEFINED, see *Tagging and the size of the tagged memory block* on page A3-15. A Store-Exclusive instruction to the same address clears the tag.

A3.4.1 Exclusive access instructions and Non-shareable memory regions

For memory regions that do not have the *Shareable* attribute, the exclusive access instructions rely on a *local monitor* that tags any address from which the processor executes a Load-Exclusive. Any non-aborted attempt by the same processor to use a Store-Exclusive to modify any address is guaranteed to clear the tag.

A Load-Exclusive performs a load from memory, and:

- the executing processor tags the physical memory address for exclusive access
- the local monitor of the executing processor transitions to its Exclusive Access state.

A Store-Exclusive performs a conditional store to memory, that depends on the state of the local monitor:

If the local monitor is in its Exclusive Access state

- If the address of the Store-Exclusive is the same as the address that has been tagged in the monitor by an earlier Load-Exclusive, then the store takes place, otherwise it is IMPLEMENTATION DEFINED whether the store takes place.
- A status value is returned to a register:
 - if the store took place the status value is 0
 - otherwise, the status value is 1.
- The local monitor of the executing processor transitions to its Open Access state.

If the local monitor is in its Open Access state

- no store takes place
- a status value of 1 is returned to a register.
- the local monitor remains in its Open Access state.

The Store-Exclusive instruction defines the register to which the status value is returned.

When a processor writes using any instruction other than a Store-Exclusive:

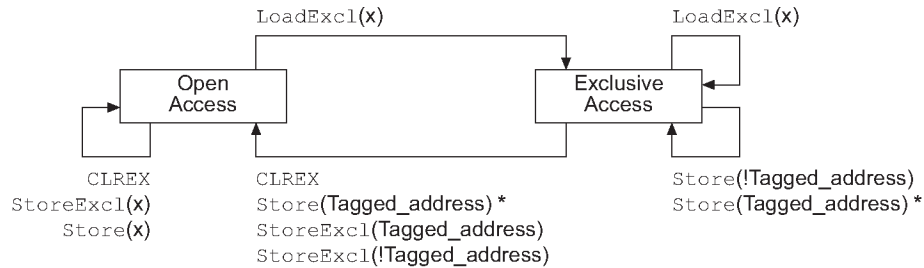
- if the write is to a physical address that is not covered by its local monitor the write does not affect the state of the local monitor
- if the write is to a physical address that is covered by its local monitor it is IMPLEMENTATION DEFINED whether the write affects the state of the local monitor.

If the local monitor is in its Exclusive Access state and a processor performs a Store-Exclusive to any address other than the last one from which it has performed a Load-Exclusive, it is IMPLEMENTATION DEFINED whether the store succeeds, but in all cases the local monitor is reset to its Open Access state. In ARMv7-M, the store must be treated as a software programming error.

———— **Note** —————

It is UNPREDICTABLE whether a store to a tagged physical address causes a tag in the local monitor to be cleared if that store is by an observer other than the one that caused the physical address to be tagged.

Figure A3-2 on page A3-10 shows the state machine for the local monitor. Table A3-6 on page A3-10 shows the effect of each of the operations shown in the figure.



Operations marked * are possible alternative IMPLEMENTATION DEFINED options.

In the diagram: LoadExcl represents any Load-Exclusive instruction
 StoreExcl represents any Store-Exclusive instruction
 Store represents any other store instruction.

Any LoadExcl operation updates the tagged address to the most significant bits of the address x used for the operation. For more information see the section *Size of the tagged memory block*.

Figure A3-2 Local monitor state machine diagram

Note

- The IMPLEMENTATION DEFINED options for the local monitor are consistent with the local monitor being constructed so that it does not hold any physical address, but instead treats any access as matching the address of the previous LDREX.
- A local monitor implementation can be unaware of Load-Exclusive and Store-Exclusive operations from other processors.
- It is UNPREDICTABLE whether the transition from Exclusive Access to Open Access state occurs when the STR or STREX is from another observer.

Table A3-6 shows the effect of the operations shown in Figure A3-2.

Table A3-6 Effect of Exclusive instructions and write operations on local monitor

Initial state	Operation ^a	Effect	Final state
Open Access	CLREX	No effect	Open Access
Open Access	StoreExcl(x)	Does not update memory, returns status 1	Open Access
Open Access	LoadExcl(x)	Loads value from memory, tags address x	Exclusive Access
Open Access	Store(x)	Updates memory, no effect on monitor	Open Access
Exclusive Access	CLREX	Clears tagged address	Open Access
Exclusive Access	StoreExcl(t)	Updates memory, returns status 0	Open Access

Table A3-6 Effect of Exclusive instructions and write operations on local monitor (continued)

Initial state	Operation ^a	Effect	Final state
Exclusive Access	StoreExcl(!t)	Updates memory, returns status 0 ^b	Open Access
		Does not update memory, returns status 1 ^b	
Exclusive Access	LoadExcl(x)	Loads value from memory, changes tag to address to x	Exclusive Access
Exclusive Access	Store(!t)	Updates memory, no effect on monitor	Exclusive Access
Exclusive Access	Store(t)	Updates memory	Exclusive Access ^b
			Open Access ^b

a. In the table:

LoadExcl represents any Load-Exclusive instruction

StoreExcl represents any Store-Exclusive instruction

Store represents any store operation other than a Store-Exclusive operation.

t is the tagged address, bits [31:a] of the address of the last Load-Exclusive instruction. For more information see *Tagging and the size of the tagged memory block* on page A3-15.

b. IMPLEMENTATION DEFINED alternative actions.

A3.4.2 Exclusive access instructions and Shareable memory regions

For memory regions that have the *Shareable* attribute, exclusive access instructions rely on:

- A *local monitor* for each processor in the system, that tags any address from which the processor executes a Load-Exclusive. The local monitor operates as described in *Exclusive access instructions and Non-shareable memory regions* on page A3-8, except that for Shareable memory, any Store-Exclusive described in that section as updating memory and/or returning the status value 0 is then subject to checking by the global monitor. The local monitor can ignore exclusive accesses from other processors in the system.
- A *global monitor* that tags a physical address as exclusive access for a particular processor. This tag is used later to determine whether a Store-Exclusive to the tagged address, that has not been failed by the local monitor, can occur. Any successful write to the tagged address by any other observer in the shareability domain of the memory location is guaranteed to clear the tag.

For each processor in the system, the global monitor:

- holds a single tagged address
- maintains a state machine.

The global monitor can either reside in a processor block or exist as a secondary monitor at the memory interfaces.

An implementation can combine the functionality of the global and local monitors into a single unit.

Table A3-7 Effect of load/store operations on global monitor for processor(n) (continued)

Initial state ^a	Operation ^b	Effect	Final state ^a
Exclusive	Store(t,n)	Updates memory	Exclusive ^e
			Open ^e
Exclusive	Store(t,!n)	Updates memory	Open
Exclusive	Store(!t,n), Store(!t,!n)	Updates memory, no effect on monitor	Exclusive

a. Open = Open Access state, Exclusive = Exclusive Access state.

b. In the table:

LoadExcl represents any Load-Exclusive instruction

StoreExcl represents any Store-Exclusive instruction

Store represents any store operation other than a Store-Exclusive operation.

t is the tagged address for processor(n), bits [31:a] of the address of the last Load-Exclusive instruction issued by processor(n), see *Tagging and the size of the tagged memory block*.

c. The result of a STREX(x,!n) or a STREX(t,!n) operation depends on the state machine and tagged address for the processor issuing the STREX instruction. This table shows how each possible outcome affects the state machine for processor(n).

d. After a successful STREX to the tagged address, the state of the state machine is IMPLEMENTATION DEFINED. However, this state has no effect on the subsequent operation of the global monitor.

e. Effect is IMPLEMENTATION DEFINED. The table shows all permitted implementations.


A3.4.3 Tagging and the size of the tagged memory block

As shown in Figure A3-2 on page A3-10 and Figure A3-3 on page A3-13, when a LDREX instruction is executed, the resulting tag address ignores the least significant bits of the memory address:

```
Tagged_address == Memory_address[31:a]
```

The value of a in this assignment is IMPLEMENTATION DEFINED, between a minimum value of 2 and a maximum value of 11. For example, in an implementation where a = 4, a successful LDREX of address 0x000341B4 gives a tag value of bits [31:4] of the address, giving 0x000341B. This means that the four words of memory from 0x000341B0 to 0x000341BF are tagged for exclusive access. Subsequently, a valid STREX to any address in this block will remove the tag.

The size of the tagged memory block is called the *Exclusives Reservation Granule*. The Exclusives Reservation Granule is IMPLEMENTATION DEFINED between:

- one word, in an implementation with a == 2
- 512 words, in an implementation with a == 11 

A3.4.4 Context switch support

It is necessary to ensure that the local monitor is in the Open Access state after a context switch. In ARMv7-M, the local monitor is changed to Open Access automatically as part of an exception entry or exit sequence. The local monitor can also be forced to the Open Access state by a CLREX instruction.

———— **Note** —————

Context switching is not an application level operation. However, this information is included here to complete the description of the exclusive operations.

A context switch might cause a subsequent Store-Exclusive to fail, requiring a load ... store sequence to be replayed. To minimize the possibility of this happening, ARM recommends that the Store-Exclusive instruction is kept as close as possible to the associated Load-Exclusive instruction, see *Load-Exclusive and Store-Exclusive usage restrictions*.

A3.4.5 Load-Exclusive and Store-Exclusive usage restrictions

The Load-Exclusive and Store-Exclusive instructions are designed to work together, as a pair, for example a LDREX/STREX pair or a LDREXB/STREXB pair. As mentioned in *Context switch support*, ARM recommends that the Store-Exclusive instruction always follows within a few instructions of its associated Load-Exclusive instructions. In order to support different implementations of these functions, software must follow the notes and restrictions given here.

These notes describe use of a LDREX/STREX pair, but apply equally to any other Load-Exclusive/Store-Exclusive pair:

- The exclusives support a single outstanding exclusive access for each processor thread that is executed. The architecture makes use of this by not requiring an address or size check as part of the `IsExclusiveLocal()` function. If the target address of an STREX is different from the preceding LDREX in the same execution thread, behavior can be UNPREDICTABLE. As a result, an LDREX/STREX pair can only be relied upon to eventually succeed if they are executed with the same address.
- An explicit store to memory can cause the clearing of exclusive monitors associated with other processors, therefore, performing a store between the LDREX and the STREX can result in a livelock situation. As a result, code must avoid placing an explicit store between an LDREX and an STREX in a single code sequence.
- If two STREX instructions are executed without an intervening LDREX the second STREX returns a status value of 1. This means that:
 - every STREX must have a preceding LDREX associated with it in a given thread of execution
 - it is not necessary for every LDREX to have a subsequent STREX.
- An implementation of the Load-Exclusive and Store-Exclusive instructions can require that, in any thread of execution, the transaction size of a Store-Exclusive is the same as the transaction size of the preceding Load-Exclusive that was executed in that thread. If the transaction size of a Store-Exclusive is different from the preceding Load-Exclusive in the same execution thread, behavior can be UNPREDICTABLE. As a result, software can rely on a Load-Exclusive/Store-Exclusive pair to eventually succeed only if they are executed with the same address.

- An implementation might clear an exclusive monitor between the LDREX and the STREX, without any application-related cause. For example, this might happen because of cache evictions. Code written for such an implementation must avoid having any explicit memory accesses or cache maintenance operations between the LDREX and STREX instructions.
- Implementations can benefit from keeping the LDREX and STREX operations close together in a single code sequence. This minimizes the likelihood of the exclusive monitor state being cleared between the LDREX instruction and the STREX instruction. Therefore, ARM recommends strongly a limit of 128 bytes between LDREX and STREX instructions in a single code sequence, for best performance.
- Implementations that implement coherent protocols, or have only a single master, might combine the local and global monitors for a given processor. The IMPLEMENTATION DEFINED and UNPREDICTABLE parts of the definitions in *Pseudocode details of operations on exclusive monitors* on page B2-8 are provided to cover this behavior.
- The architecture sets an upper limit of 2048 bytes on the size of a region that can be marked as exclusive. Therefore, for performance reasons, ARM recommends that software separates objects that will be accessed by exclusive accesses by at least 2048 bytes. This is a performance guideline rather than a functional requirement.
- LDREX and STREX operations must be performed only on memory with the Normal memory attribute.
- If the memory attributes for the memory being accessed by an LDREX/STREX pair are changed between the LDREX and the STREX, behavior is UNPREDICTABLE.

A3.4.6 Synchronization primitives and the memory order model

The synchronization primitives follow the memory ordering model of the memory type accessed by the instructions. For this reason:

- Portable code for claiming a spinlock must include a DMB instruction between claiming the spinlock and making any access that makes use of the spinlock.
- Portable code for releasing a spinlock must include a DMB instruction before writing to clear the spinlock.

This requirement applies to code using the Load-Exclusive/Store-Exclusive instruction pairs, for example LDREX/STREX.

Jul 05, 10 15:56

lockacquire.S

Page 1/1

```

1      .syntax unified
2      .text
3      .align 2
4      .thumb
5      .thumb_func
6
7      /* Returns 0 if unsuccessful, 1 if successful in acquiring lock
8         Equivalent to:
9
10     unsigned lock_acquire(unsigned *lockaddr) {
11         if (*lockaddr == 0) { // LDREX instruction instead of LDR
12             CLREX;           // Release exclusive lock
13             return 0;        // Failure: lock is already acquired
14         }
15         *lockaddr = 0;       // STREX instruction instead of STR
16         if (STREXreturn==1) { // STREX returns 1 if unsuccessful
17             return 0;        // Failure: lock is under exclusive lock
18         } else {            // and 0 was NOT written to *lockaddr
19             return 1;        // STREX successful, we have the lock,
20                             // exclusive lock is released, 0 WAS
21                             // stored in *lockaddr
22     }
23     */
24     .type lock_acquire, function
25     .global lock_acquire
26 lock_acquire:
27     MOV     r1, #0
28
29     LDREX   r2, [r0]         @ R2 ← lock value
30     CMP     r2, r1           @ Is it already 0? (hence locked?)
31     ITT    NE
32     STREXNE r2, r1, [r0]    @ If not, try to claim it by writing 0
33                                     @ R2←0 if successful, 1 if failure
34     CMPNE  r2, #1           @ and check success
35     BEQ    lf               @ Branch taken if lock was already 0
36                                     @ (so the previous two xxxNE instructions
37                                     @ did not execute) or STREXNE returned 1.
38     MOV    R0, #1           @ Indicate success
39     BX     LR
40
41 1:                                     @ Local label...branch here from above with destination 'lf'
42     CLREX                                     @ We did not get the lock. Clear exclusive access.
43     MOV    R0, #0           @ Indicate failure
44     BX     LR

```