Exclusive Primitives of the Cortex-M

© Remo Markgraf, info@tnms.de, www.tnms.de

V1.1, 14.02.2025

Intro

Concurrency plays a central role in the booming programming language RUST. In C and C++ it is the responsibility of the designer to apply appropriate mechanisms to avoid the dreaded race conditions, even in small embedded bare metal projects. Disabling interrupts is commonly used to avoid these race conditions. The Cortex-M provides with Exclusive Primitives a better mechanism, without the need to disable interrupts and thus without affecting the interrupt latency. The right application of these Exclusive Primitives is not trivial and therefore we take a deeper look at the application of the Cortex-M Load and Store Exclusive instructions.

Scenario

Assuming you just want to set a flag in an interrupt routine. To ensure that the interrupt duration is as short as possible, the real treatment of the interrupt is carried out in the foreground as a reaction to the setting of the flag in the interrupt service routine. To save storage space, only a single bit is used as a flag in the embedded SW. Whether you address this bit via a mask or via a bit field structure, the setting and reset of the flag always consists of a Read-Modify-Write command sequence that can be interrupted. The root thing about this scenario is that setting and clearing the flag works nicely. Only the remaining bits which are temporarily stored between the read and the write part of the instruction sequence are overwritten if their value was changed in an interrupt taken during this read-modify write sequence. The probability of that race condition is extremely low, because the interrupt must occur exactly during the short read-modify write sequence and in addition one of the bits must be changed. So it is not surprising that a project might be successfully in use for years and that the root cause analysis of a malfunction might become a very lengthy task. To avoid that this happens in your case you better protect the read-modify-write sequence by a Critical Section. Disabling interrupts during the Critical Section is the only option in the Armv6-M architecture, e.g. for the Cortex-M0 or M0+. In the Armv7-M and the Armv8-M architecture, e.g. on the M3, M4. M7. M23, M33, M35P, M52, M55 and the M85 there is a better mechanism to realize such Critical Section with the Load and Store Exclusive Instructions. Unfortunately, the mechanism is not preventing the concurrency, but only to recognize them safely and to react to them with an appropriate algorithm. This makes it a little more complicated, but with the help of this guide doable.

Instruction Set

The Exclusive Instructions use a tag that is realized in the core hardware. The LDREX command not only reads the content of an address in the memory map into a core register, but also sets this tag. The STREX command only performs the write if the tag is set. By means of the value of a core register, the STREX command returns to the caller whether the write command was executed successfully and resets the tag. For resetting the tag without a write operation, there is the CLREX command. This results in the following command sequence for a Read-Modify-Write action:

1. Reading a value from the memory into a register with LDREX. The tag is set.
2. Manipulating the value in the register. For example, set a certain bit.
3. Write the changed register value back to memory with STREX. The tag is reset.

If this command sequence was interrupted and a read-modify write operation has also been performed within the interruption, then the tag would have already been reset when step 3 were to be executed and the write operation would not be performed. The conflict would not be prevented, but recognised. The easiest means of reacting to the detection is to put the read modify write sequence in a while loop until the return value of the STREX reports that it has been successfully written. In practice, this while loop will leave after the second round at the latest. Since the conflict occurs very rarely, it is thus not of importance that there is usually only one tag for the entire memory map. As a matter of fact scenarios might therefore be treated as conflicts that are actually not. For example, in the foreground exclusive read-modify write of address A and in the interruption exclusive read-modify write of address B would be detected as a conflict. However, the consequence is only the unnecessary execution of a loop cycle and therefore negligible. The chip designer has the opportunity to configure more than one tag. The corresponding tag is determined by the memory map. This is applied in multicore designs, for example, to provide each core an own tag, which is used for the private memory areas and the shared memory area gets assigned a system tag that is shared between all cores. The SW developer has no influence on this, so it helps to look into the manual of the used multi-core. Figure 1 shows the various exclusive LDREX and STREX Thumb2 machine code instructions of the Cortex-M.

- **`LDREXB`** and **`STREXB`** for 8-bit
- **`LDREXH`** and **`STREXH`** for16-bit
- **`LDREX`** and **`STREX`** for 32-bit.
- **`CLREX`** is only available once
- A 64-bit variant is not available in the Cortex-M.

Figure 1: Exclusive Instructions

CMSIS

In order to avoid inline assembler in C and C++, there are corresponding functions available in CMSIS. Please refer also to:

- https://github.com/ARM-software/CMSIS_5 und
- https://github.com/ARM-software/CMSIS_6

```
void      __CLREX( void );
uint8_t   __LDREXB( volatile uint8_t  *addr);
uint16_t  __LDREXH( volatile uint16_t *addr);
uint32_t  __LDREXW( volatile uint32_t *addr);
uint32_t  __STREXB( uint8_t  value, volatile uint8_t  *addr);
uint32_t  __STREXH( uint16_t value, volatile uint16_t *addr);
uint32_t  __STREXW( uint32_t value, volatile uint32_t *addr);
__STREX return value: 0 Function succeeded, 1 Function failed
```

Figure 2: CMSIS Functions

```
1     static volatile uint32_t flags = 0;
2
3     void atomic_function_set_bit(uint8_t bitno)
4     {
5         uint32_t val = 0;
6         do {
7             val = __LDREXW(&flags);
8             val |= (0x1UL << (bitno & 0x1Fu));
9         } while(__STREXW(val, &flags) != 0);
10    }
11
```

Figure 3: Code Example

Figure 3 shows a code example of a read-modify write sequence. The read-modify write sequence is placed in a do-while loop, which is embedded in a function. The bit number to be modified is provided as a parameter and has the value range 0 to 31. The corresponding bit in the flag should be set without occurrence of race conditions that might set the value of the remaining bits to incorrect values. For example, when setting the bit 1, the setting of the bit 2 could be lost if the bit 2 were set within an interrupt treatment and the interrupt occurs within the read-modify write sequence. The old value of bit 2 would be written out again after modifying bit 1. The use of the Exclusive commands in both the function itself and in the interrupt routine fixes this error.

```
1     static volatile uint32_t flags = 0;
2
3     void atomic_function_set_bit(uint8_t bitno)
4     {
5         uint32_t retval = 1UL;
6         while( retval != 0)
7         {
8             uint32_t val = __LDREXW(&flags);
9             val |= (0x1UL << (bitno & 0x1Fu));
10            retval = __STREXW(val, &flags);
11        }
12    }
13
```

Figure 4: Code Example after applying Clean Code rules

```
14    void atomic_function_clear_bit(uint8_t bitno)
15    {
16        uint32_t retval = 1UL;
17        while( retval != 0)
18        {
19            uint32_t val = __LDREXW(&flags);
20            val &= ~(0x1UL << (bitno & 0x1Fu));
21            retval = __STREXW(val, &flags);
22        }
23    }
24
```

Figure 5: Code Example extended by a clear bit function

Figure 4 shows the same code example again, this time with two Clean Code improvements. The function call of the

1. __STREXW() CMSIS function is removed from the condition of the while loop by introducing the local retval variable.
2. The unpleasant do-while loop can be converted into a while loop by initializing the retval value to 1.

Figure 5 shows the extension of the code example by a function to clear one of the bits. This requires only two small changes to the copy of the set function. In line 14, the name of the function is changed, in line 20 the “or” of the bit is replaced by an “and” with the bit inverse value to be set.
In addition, the value range of the bitno parameter should be checked with an assertion. For this purpose, an include statement is inserted at the beginning of the module. `#include <assert.h>` and in the body of the two functions the value is checked with:: `assert(bitno < 32);`

Application in RTOS context

Counting semaphore are used to synchronize tasks. A producer task could for example read data from a sensor and a consumer task could use the read values for calculations. Since the two tasks can run asynchronously with different priorities, synchronization is necessary. The producer increments a counter variable when data is read in and stored in a buffer. The consumer decrements the counter when it reads and processes data from the buffer. If the counter has the value 0, there is no data for processing, due to the asynchrony of the tasks, the value can become larger than 1, i.e. several data packets are waiting to be processed. Both increment and decrement must be protected by critical sections because of potential race conditions. The same applies to MUTEX, which usually enable access protection to resources. The

main difference of MUTEX to a semaphore is that a MUTEX is always assigned to a single task, i.e. only the task that got a MUTEX can release it again. For example, in the open() function a UART interface is protected by a MUTEX from access by other tasks until the “owner” releases the MUTEX again. Whereas semaphores might be accessed by several tasks. This difference does not affected the application of Exclusive Primitives to the realization of the Critical Sections. Therefore the following code examples are limited to a potential realization of semaphores.

Have fun with Cortex-Mx 😊.

Finally, please find below two code examples for OS_Semaphore.h and OS_Semaphore.c

```c
/*****************************************************************************//**
 * @file OS_Semaphore.h
 * @version V1.1
 * @date 14th of Feb 2025
 * @author Remo Markgraf
 * Copyright (C) 2025 Remo Markgraf info@tnms.de All rights reserved.
 * @warning
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS,
 * IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT TO
 * THIS SOFTWARE. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY CLAIM,
 * DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
 * OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE
 * USE OR OTHER DEALINGS IN THE SOFTWARE.
 *****************************************************************************/
#ifndef OS_SEMAPHORE_H_
#define OS_SEMAPHORE_H_

#include <stdint.h>

enum
{
	OS_SEMAPHORE_MAXVALUE = 0xFF,
};
enum // Error Codes
{
	OS_SUCCESS = 0,
	OS_TIMEOUT = 1,
	OS_SEMAPHORE_OVERFLOW = 14,
	OS_SEMAPHORE_NULL = 15,
};

typedef struct
{
	uint32_t value;
} OS_Semaphore_t;

//prototypes
uint32_t OS_Semaphore_Put(OS_Semaphore_t *pSema);
uint32_t OS_Semaphore_Get(OS_Semaphore_t *pSema, uint32_t timeout);
uint32_t OS_WaitForSemaphore(OS_Semaphore_t *pSema, uint32_t timeout);

#endif //__OS_SEMAPHORE_H

```

```c
/*****************************************************************************//**
 * @file OS_Semaphore.c
 * @version V1.1
 * @date 14th of Feb 2025
 * @author Remo Markgraf
 * Copyright (C) 2025 Remo Markgraf info@tnms.de All rights reserved.
 * @warning
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS,
 * IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, THE WARRANTIES OF
 * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT TO
 * THIS SOFTWARE. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY CLAIM,
 * DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
 * OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE
 * USE OR OTHER DEALINGS IN THE SOFTWARE.
 *****************************************************************************/
#include <stdint.h>
#include "OS_Semaphore.h"

//OS_Semaphore_Put increments the Semaphore value
//returns OS_SUCCESS or a corresponding ERRORNUMBER otherwise
//
uint32_t OS_Semaphore_Put(OS_Semaphore_t *pSema)
{
	uint32_t retval = 1UL;
	while( retval != 0)
	{
		uint32_t val = __LDREXW(&pSema->value);
		if( val >= OS_SEMAPHORE_MAXVALUE )
		{
			__CLREX();
			return OS_SEMAPHORE_OVERFLOW;
		}
		val++;
		retval = __STREXW(val, &pSema->value);
	}
	return OS_SUCCESS;
}

//OS_Semaphore_Get decrements the Semaphore value
//when the value was 0 it waits for timout ticks and returns
//returns OS_SUCCESS or a corresponding ERRORNUMBER otherwise
//
uint32_t OS_Semaphore_Get(OS_Semaphore_t *pSema, uint32_t timeout)
{
	uint32_t retval = 1UL;
	while( retval != 0 )
	{
		uint32_t val = __LDREXW(&pSema->value);
		if( 0 == val )
		{
			__CLREX();
			uint32_t retwait = OS_WaitForSemaphore(pSema, timeout);
			if( retwait != OS_SUCCESS )
			{
				return retwait;
			}
			val = __LDREXW(&pSema->value);
			if( 0 == val ) //should not happen
			{
```

```
60                                          __CLREX();
61                                          return OS_SEMAPHORE_NULL;
62                                  }
63                          }
64                          val--;
65                          retval = __STREXW(val, &pSema->value);
66                  }
67                  return OS_SUCCESS;
68          }
69
70          //OS_WaitForSemaphore waits for a Semaphore value > 0
71          //Returns with OS_SUCESS if it happens within the timeout
72          //Returns with OS_TIMEOUT otherwise
73          //
74          uint32_t OS_WaitForSemaphore(OS_Semaphore_t *pSema, uint32_t timeout)
75          {
76                  uint32_t wait_ticks = timeout;
77                  while( 0 == pSema->value )
78                  {
79                          if( 0 == wait_ticks )
80                          {
81                                  return OS_TIMEOUT;
82                          }
83                          OS_WaitForTick(); //wait for next tick
84                          wait_ticks--;
85                   }
86                  return OS_SUCCESS;
87           }
88
```

Hi, my name is Remo Markgraf and I train and consult extensively in projects related to Cortex-M, embedded software development and testing, test-driven development and agile development for embedded systems. Training and Consulting is provided in German and English language. My experience covers numerous areas of software development, test engineering, system architecture, project, product, lifecycle and business management and of course Live und Online Trainings www.tnms.de