Exclusive Primitives des Cortex-M

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

V1.1, 14.02.2025

Intro

Nebenläufigkeit ist ein Begriff, der in der boomenden Programmierung mit RUST eine zentrale Rolle spielt. In C und C++ liegt es in der Verantwortung des Entwicklers geeignete Mechanismen anzuwenden, um selbst in kleinen Embedded Bare Metal Projekten, die gefürchteten Race Conditions zu vermeiden. Das Sperren von Interrupts ist ein übliches Mittel zur Vermeidung. Der Cortex-M bietet dafür bessere Mechanismen, die ganz ohne das Sperren von Interrupts auskommen und dadurch die Interrupt Latency nicht verschlechtern. Die Richtige Anwendung dieser Exclusive Primitives ist nicht trivial und daher werfen wir hier einen tieferen Blick in die Anwendung der Load- und Store-Exclusive Befehle des Cortex-M.

Scenario

Nehmen wir an in einer Interrupt Routine wollen Sie nur ein Flag setzen. Damit die Unterbrechung möglichst kurz ist soll die eigentliche Behandlung des Interrupts im Vordergrund als Reaktion auf das Setzen des Flags erfolgen und das Flag wieder zurückgesetzt werden. Um Speicherplatz zu sparen wird in der Embedded SW gerne nur ein einziges Bit als Flag genutzt. Egal, ob sie dieses Bit über eine Maske adressieren oder über eine Bitfeldstruktur, das Setzen und Rücksetzen des Flags besteht immer aus einer Read-Modify-Write Befehlsfolge, die durch Interrupts unterbrochen werden kann. Das Gemeine an diesem Scenario ist, dass das Setzen und Rücksetzen des Flags problemlos funktioniert, nur die durch den Read-Anteil kurzfristig bis zum Write-Anteil der Befehlsfolge zwischengespeicherten restlichen Bits, werden im Write-Anteil gnadenlos überschrieben, falls sich Ihr Wert in einer Interrupt Behandlung während der Read-Modify-Write Befehlsfolge geändert hat. Das Tückische an dieser potentiellen Race Condition ist ihr extrem seltenes Auftreten, denn der Interrupt muss ja genau während der kurzen Read-Modify-Write Befehlsfolge auftreten und eines der Bits auch noch verändern. So wundert es nicht, dass ein Projekt jahrelang erfolgreich im Einsatz ist und die Fehlersuche eines unerklärlichen Verhaltens zu einer sehr langwierigen Aufgabe wird. Damit Ihnen das nicht widerfährt schützen Sie die Read-Modify-Write Befehlsfolge vor unerwünschten Unterbrechungen durch eine Critical Section. Interrupts innerhalb der Critical Section sperren ist in der Armv6-M Architektur, also z.B beim Cortex-M0 oder M0+ leider die einzige sinnvolle Möglichkeit. In der Armv7-M und der Armv8-M Architektur also z.B. beim M3, M4. M7. M23, M33, M35P, M52, M55 und dem M85 gibt es mit den Load- und Store-Exclusive Befehlen eine geschicktere Methode solche Critical Section zu realisieren. Leider besteht

der Mechanismus nicht darin, die Nebenläufigkeit zu verhindern, sondern nur darin diese sicher zu erkennen und mit einem geeigneten Algorithmus darauf zu reagieren. Das macht das Ganz etwas komplizierter, mit dieser Anleitung aber beherrschbar 😊.

Befehlssatz

Die Exclusive Befehle benutzen ein Tag, das in der Core Hardware realisiert ist. Mit dem LDREX Befehl wird nicht nur ein Registerinhalt aus der Memory-Map gelesen, sondern auch dieses Tag gesetzt. Der STREX Befehl führt den Schreibbefehl nur durch, wenn das Tag gesetzt ist. Über den Return-Wert in einem Core Register gibt der STREX Befehl dem Aufrufer zurück, ob der Schreibbefehl erfolgreich ausgeführt wurde und setzt das Tag zurück. Für das Zurücksetzen des Tags ohne eine Schreiboperation gibt es noch den CLREX Befehl. Damit ergibt sich für eine Read-Modify-Write Aktion folgende Befehlssequenz:

1) Lesen eines Wertes aus dem Speicher in ein Register mit LDREX. Dabei wird das Tag gesetzt.
2) Manipulieren des Wertes im Register. Z.B. ein bestimmtes Bit setzten.
3) Schreiben des veränderten Registerwertes zurück in den Speicher mit STREX. Dabei wird der Tag zurückgesetzt.

Sollte diese Befehlsfolge unterbrochen worden sein und innerhalb der Unterbrechung ebenfalls eine Read-Modify-Write Operation ausgeführt worden sein, dann wäre das Tag bereits zurückgesetzt, wenn der Schritt 3 ausgeführt werden sollte und die Schreiboperation wäre nicht ausgeführt worden.
Der Konflikt wäre damit nicht verhindert, aber erkannt. Einfachstes Mittel der Reaktion auf die Erkennung liegt darin, die Read-Modify-Write Sequenz in eine While-Loop zu setzen, bis der Return-Wert des STREX meldet, dass erfolgreich geschrieben werden konnte. In der Praxis wird diese While-Loop spätestens nach der zweiten Runde verlassen. Da der Konflikt nur sehr selten auftritt ist es kaum von Belang, dass es i.d.R. nur ein einziges Tag für die komplette Memory-Map gibt und damit Konflikte als solche behandelt werden, die eigentlich keine sind. Z.B. im Vordergrund exklusives Read-Modify-Write der Adresse A und in der Unterbrechung exklusives Read-Modify-Write der Adresse B würde als Konflikt erkannt. Die Folge ist aber nur die unnötige Ausführung eines Loop-Durchgangs und daher vernachlässigbar. Der Chipdesigner hat die Möglichkeit mehr als einen Tag einzubauen. Die Wahl des verwendeten Tags wird über die Memory-Map bestimmt. Dies wird z.B. dazu verwendet in Multicore Designs jedem Core ein eigenes Tag zu geben, das für die privaten Memory Bereiche verwendet wird und dem Shared Memory Bereich ein System Tag, das von allen Cores gemeinsam benutzt wird. Darauf hat der SW Entwickler keinen Einfluss, es hilft hier also nur der Blick ins Manual des verwendeten Multi-Cores. Bild 1 zeigt die verschiedenen exklusiven LDREX und STREX Thumb2 Maschinenbefehle des Cortex-M.

- **`LDREXB`** und **`STREXB`** für 8-bit
- **`LDREXH`** und **`STREXH`** für16-bit
- **`LDREX`** und **`STREX`** für 32-bit.
- **`CLREX`** gibt es nur ein Mal
- Eine 64-bit Variante gibt es im Cortex-M nicht.

Bild 1: Exclusive Befehle des Cortex-M

CMSIS

Um sich in C und C++ nicht mit inline Assembler herumschlagen zu müssen gibt es unter CMSIS entsprechende Funktionen. Siehe dazu auch:

- 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);
Returnwert: 0 Function succeeded, 1 Function failed
```

Bild 2: CMSIS Funktionen

```
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
```

Bild 3: Code Beispiel

In Bild 3 ist ein Code Beispiel für eine Read-Modify-Write Sequenz dargestellt. Dabei befindet sich die Read-Modify-Write Sequenz in einer do-while-Loop, die wiederum in einer Funktion eingebettet ist und als Parameter die Bitnummer im Wertebereich 0 bis 31 in einem Flag erwartet. Das entsprechende Bit in dem Flag soll gesetzt werden, ohne dass Race Conditions den Wert der restlichen Bits auf fehlerhafte Werte setzen können. Beim Setzen des Bits mit der Nummer 1 könnte das Setzen des Bits mit der Nummer 2 verloren gehen, falls das Bit 2 innerhalb einer Interrupt Behandlung gesetzt wird und der Interrupt innerhalb der Read-Modify-Write Sequenz auftritt. Der alte Wert des Bit 2 würde nach dem Lesen zum Modifizieren verwendet und wieder herausgeschrieben werden. Die Verwendung der Exclusive Befehle sowohl in der Funktion als auch in der Interrupt Routine beheben diesen Fehler.

```
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
```

Bild 4: Code Beispiel nach Clean Code

```
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
```

Bild 5: Code Beispiel Erweiterung um eine clear bit Funktion

Bild 4 zeigt das gleiche Code Beispiel nochmals, diesmal mit zwei Clean Code Verbesserungen.

1) Der Funktionsaufruf der __STREXW() CMSIS-Funktion ist durch die Einführung der lokalen retval Variable aus der Bedingung der while-Loop herausgelöst.
2) Die unliebsame do-while-Loop, kann mit der Initialisierung des retval Wertes auf 1 in eine while-Loop gewandelt werden.

Bild 5 zeigt die Erweiterung des Code Beispiels um eine Funktion zum Löschen eines der Bits.

Hierzu sind nur zwei kleine Änderungen an der Kopie der set-Funktion erforderlich. In Zeile 14 wird der Name der Funktion geändert, in Zeile 20 das „Verodern“ des Bits zum Setzen in ein „Verunden“ mit dem bitinversen Wert ersetzt

Zusätzlich sollte der Wertebereich des bitno Parameters mit einer Assertion abgesichert werden. Dazu wird am Anfang des Moduls eine Include Anweisung eingefügt `#include <assert.h>` und im Body der beiden Funktionen der Wert überprüft mit: `assert(bitno < 32);`

Anwendung im RTOS Kontext

Als Beispiel hier abschließend noch Beispiele für Anwendungsfälle für die Exclusive Primitives im RTOS Kontext. Counting Semaphore werden zum Synchronisieren von Tasks verwendet. Eine Producer-Task könnte z.B. Daten von einem Sensor einlesen und eine Consumer-Task könnte die eingelesenen Werte für Berechnungen verwenden. Da die beiden Tasks asynchron mit verschiedenen Prioritäten laufen können ist eine Synchronisierung nötig. Der Producer inkrementiert dazu eine Zähler-Variable, wenn Daten eingelesen und in einem Puffer abgelegt werden. Der Consumer dekrementiert den Zähler, wenn er Daten aus dem Puffer liest und verarbeitet hat. Wenn der Zähler den Wert 0 hat, sind keine Daten zur Verarbeitung

vorhanden, durch die Asynchronität der Tasks, kann der Wert durchaus größer als 1 werden, d.h. mehrere Datenpakete warten darauf verarbeitet zu werden. Sowohl das In- als auch das Dekrementieren muss wegen potentieller Race Conditions durch Critical Sections geschützt werden. Ähnliches gilt für MUTEX, die üblicherweise Zugriffsschutz auf Resourcen ermöglichen. Der Hauptunterschied des MUTEX zu einem Semaphore liegt darin, dass ein MUTEX immer einer Task zugeordnet ist, d.h. nur die Task, die den MUTEX besitzt kann ihn wieder freigeben. Z.B. wird eine UART Schnittstelle beim open() durch einen MUTEX vor dem Zugriff anderer Tasks geschützt, bis der „Besitzer“ den MUTEX wieder freigibt. Beim Semaphore können das mehrere Beteiligten sein. Die Verwendung der Exclusive Primitives zur Realisierung der Critical Sections ist von diesem Unterschied nicht betroffen, daher beschränken wir uns in dem folgenden Code Beispiel auf eine potentielle Realisierung von Semaphoren.

Viel Spaß mit den Cortex-Mx-ern 😊 .

Zum Abschluss hier noch die zwei Codebeispiele für OS_Semaphore.h und 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
```

Hallo, ich heiße Remo Markgraf und biete intensive Trainings und Consulting in Projekten rund um Cortex-M, Embedded Software Entwicklung und Test, Test-Driven Development und agile Entwicklung von Embedded Systemen. Training und Consulting ist auf Deutsch und Englisch möglich. Ich verfüge über Erfahrung und Kenntnisse in verschiedensten Bereichen der Software Entwicklung, Test Entwurf und Umsetzung, System Architekturen, Projekt-, Produkt-, Lifecycle- und Business-Management und natürlich Live und Online Schulungen. www.tnms.de